University of Porto • Faculty of Engineering

Doctoral Thesis

Increasing the Dependability of Internet-of-Things Systems in the context of End-User Development Environments

João Pedro Matos Teixeira Dias

Scientific Supervisor:
Hugo Sereno Ferreira, PhD
Assistant Professor

Scientific Co-Supervisor:
João Pascoal Faria, PhD
Associate Professor

In partial fulfillment of requirements for the degree of
Doctor of Philosophy in Informatics Engineering
by the
Doctoral Program in Informatics Engineering (ProDEI)

April 2022

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Increasing the Dependability of Internet-of-Things Systems in the context of End-User Development Environments

João Pedro Matos Teixeira Dias

In partial fulfillment of requirements for the degree of
Doctor of Philosophy in Informatics Engineering
by the
Doctoral Program in Informatics Engineering (ProDEI)

Approved by:

President of the Jury:

- PhD Rui Filipe Lima Maranhão de Abreu, Full Professor at the Department of Informatics Engineering of the Faculdade de Engenharia da Universidade do Porto.

Members:

- PhD Dariusz Mrozek, Associate Professor at the Department of Applied Informatics at Silesian University of Technology, Poland;
- PhD Pedro Nicolau Faria da Fonseca, Assistant Professor at the Department of Electronics, Telecommunications and Informatics of Universidade de Aveiro;
- PhD André Monteiro de Oliveira Restivo, Assistant Professor at the Department of Informatics Engineering of the Faculdade de Engenharia da Universidade do Porto;
- PhD Hugo José Sereno Lopes Ferreira, Assistant Professor at the Department of Informatics Engineering of the Faculdade de Engenharia da Universidade do Porto (Supervisor).

April 1, 2022

Hugo Sereno Ferreira, PhD

Contact Information

João Pedro Matos Teixeira Dias
Faculdade de Engenharia da Universidade do Porto
Departamento de Engenharia Informática

Rua Dr. Roberto Frias, s/n
4200-465 Porto
Portugal

Email: jpmdias@fe.up.pt
Web: https://jpdias.me

This thesis was typeset on a Lenovo ThinkPad T460s running Manjaro Linux, using the Microsoft Visual Studio Code editor (with the LaTeX Workshop extension), and the LaTeX typesetting system. The style is based on the Masters/Doctoral Thesis template by Vel and Johannes Böttcher, available under LPPL v1.3c License. Most of the diagrams were drawn using draw.io and Inkscape, using free for personal use icon packs and assets, including ones from freepik.com.

"Increasing the Dependability of Internet-of-Things Systems in the context of End-User Development Environments"

This work was partially funded by the Portuguese Foundation for Science and Technology (FCT), under the research grant SFRH/BD/144612/2019.

"In the new era, thought itself will be transmitted by radio."

Guglielmo Marconi, Nobel laureate in Physics

Abstract

The ubiquitousness of computing, known as Internet-of-Things (IoT), has reshaped the way people interact with the physical world. However, the scale, distribution — both logical and geographical —, density, heterogeneity, interdependence, and quality-of-service requirements of these systems make them complex, posing several challenges from both operational and development viewpoints.

While there is a consensus that the widely used software engineering practices are inadequate for IoT development, they remain the go-to solutions for most practitioners. This aspect has severely compromised their dependability, centralizing most of the computation of these (soft) real-time systems in cloud infrastructure. Likewise, as these systems scale in terms of devices and applications, it outreaches existing technical resources to manage and operate them, becoming of paramount importance, making them as most self-managed as possible while empowering the ability of system operators (including end-users) to configure and understand them — mainly using solutions that do not require high technical expertise, viz. low-code development solutions — including the configuration of fail-safe measures.

This thesis primary focus is to research how to improve the current status quo on the dependability of IoT. However, this is a manifold endeavor: (1) what are the best practices for developing IoT dependably, and what is their scientific soundness, (2) do the current solutions give the fundamental building blocks that allow to design and construct dependable systems, and, if not, what contributions are needed to overcome the existing limitations, and, lastly, (3) giving that these systems are operated by humans with limited technical expertise, it is required that their users can use and configure them without compromising their correct operation. As we set ourselves to tackle these challenges, we claim that:

> *It is possible to enrich IoT-focused end-user development environments in such a way that the resulting systems have a higher dependability degree, with the lowest impact on the know-how of the (end-)users.*

As preliminary research, to understand what end-users want to automate and how they wish to perform such automations, a study was carried to collect automation scenarios. These scenarios showcased the complexity of the automations that some end-users want to perform and the interdependencies between different information sources, devices, and persons. It also supported the view that some of the appliances that end-users want to automate can have nefarious effects if a malfunction happens or a misconfiguration is performed.

We followed extensive literature research and experimental process to *mine* a set of patterns that can be used to improve IoT systems by making them more dependable, documenting them as *patlets*, which summarily describe solutions that address some particular problem within a specific context. We further studied a subset of these patterns as a *self-healing* pattern language that contemplates the use of more than one pattern in tandem to address systems' operational concerns autonomically.

Adopting these patterns depends on supporting foundations, which include architectural and functional aspects of the target systems. A key aspect is that most of the current solutions do not provide any features to readjust their intrinsic behaviors during runtime — with the

software that runs on edge devices being mostly set on stone, delegating all the computational needs to cloud-based services. The research on fog and edge computing attempts to mitigate this by leveraging computational resources across architectural tiers, making the resulting systems more dependable and improving their scalability. Taking on these foundations, we explored and asserted the feasibility of using serverless functions in the IoT context, optimizing the choice of execution contexts according to a priori preferences, constraints, and latencies.

To understand how these paradigms can be leveraged in widely used solutions, we select the open-source Node-RED solution as the experimental base, given its popularity. It provides a visual programming interface that increases its target user base across different expertise levels. Like other available solutions, Node-RED does not provide any feature that allows it to orchestrate tasks across devices or deal with system parts' failures, limiting the dependability of systems built with it. Nonetheless, given its open-source and extensible nature, we proceed to address some of its limitations. We proceed to evaluate empirically, both in virtual and physical setups, the feasibility of using Node-RED as an orchestrator, where computational tasks are allocated to the available resources, and failures are mitigated by re-orchestrating as devices fail and recover. We also implemented a set of extensions for Node-RED that allows one to enrich the existing programs (*i.e.*, flows) with self-healing capabilities — allowing the detection errors of different parts during runtime, and readjust its behavior to keep delivering correct service by recovering to normal operation, or, at least, maintain its operation within acceptable Quality-of-Service levels.

As IoT users have different expertise levels, we also attempt to improve the interaction with these systems in a way that the users can understand what the configured automations are (viz. inspection), how it is behaving (viz. observability and feedback), and increase their capability to know what was the possible cause behind certain events (viz. causality). In the first study, we extended the visual notations and functionalities of Node-RED to improve the development process using it. We proceed to empirically evaluate the performance of our solution against a non-modified version of Node-RED, observing statistically significant improvements in the users' ability to evolve existing IoT deploys. Lastly, we explored the use of voice assistants as an alternative way of configuring, understanding, and interacting with IoT-enriched environments, with a particular focus on the ability of a user to understand the cause behind some events. We assert the feasibility of our solution by covering all the different automation possibilities that Node-RED supports, with a considerable extension of the interaction possibilities due to multi-message dialogs support. We proceeded to empirically validate the feasibility of users using the voice assistant to complete different tasks, and all the users were able to finish the tasks. While some valid sentences were incorrectly recognized, forcing the user to repeat their intent, participants expressed a preference for voice interfaces over visual ones in terms of subjective perception.

These contributions materialize into a core set of building blocks that, in assemble, can be used to improve the dependability of IoT systems while leveraging abstractions that do not hinder the (end-)user capability to configure, use, and evolve them. The experimental counterparts of the contributions provide empirical supporting evidence for the plausibility of the hypothesis.

Resumo

A ubiquidade da computação, conhecida como *Internet-of-Things* (IoT), tem moldado a maneira como as pessoas interagem com o mundo físico. No entanto, a escala, distribuição lógica e geográfica, densidade, heterogeneidade, interdependência, e os requisitos de segurança, fazem com que estes sistemas sejam complexos, criando vários desafios operacionais e de desenvolvimento. Embora exista um consenso alargado de que as práticas de engenharia de *software* amplamente utilizadas são inadequadas para o desenvolvimento de IoT, estas continuam a ser as soluções mais usadas. Este aspeto tem vindo a comprometer a confiabilidade destes sistemas, centralizando a maior parte da computação em infraestruturas *cloud*. Adicionalmente, a larga-escala destes sistemas em termos de dispositivos e aplicações supera os recursos técnicos existentes para a sua gestão e operação, sendo crucial torná-los o mais *self-managed* possível. No entanto, tem de ser dada a capacidade aos operadores do sistema (incluindo utilizadores finais) de configurá-los e entendê-los — principalmente usando soluções que não requerem alto conhecimento técnico, viz. soluções de desenvolvimento de *low-code* — incluindo a capacidade de configuração de medidas de resiliência em caso de falhas.

O foco principal desta dissertação é investigar como melhorar o status quo da confiabilidade em sistemas IoT. Este é um esforço multifacetado sendo necessário (1) perceber quais são as melhores práticas para desenvolver IoT de forma confiável e qual é sua solidez científica, (2) perceber se as soluções atuais fornecem mecanismos fundamentais que permitem desenhar e construir sistemas confiáveis, e, se não, que contribuições são necessárias para superar as limitações existentes e, por último, (3) dado que esses sistemas são operados por humanos com conhecimento técnico limitado, é necessário que os seus utilizadores sejam capazes de usar e configurar os sistemas sem comprometer o correto funcionamento dos mesmos. Enquanto nos propomos a enfrentar os desafios acima mencionados, afirmamos que:

> *É possível enriquecer os ambientes de desenvolvimento de IoT focados em utilizadores finais de tal forma que os sistemas resultantes têm um maior grau de confiabilidade, reduzindo o impacto no know-how destes utilizadores.*

Para se melhor entender o que (e como) os utilizadores finais desejam automatizar os seus sistemas de IoT foi realizado um estudo para reunir cenários de automação, como pesquisa preliminar. De seguida, uma extensa pesquisa bibliográfica foi realizada de forma a extrair um conjunto de padrões que podem ser usados para melhorar os sistemas IoT do ponto de vista da sua confiabilidade. Estes padrões foram documentados em formconfiabilidadea sucinta, expondo problemas particulares dentro de um contexto específico. As relações de um subconjunto desses padrões foram estudadas, resultando numa linguagem de padrão de *self-healing*, endereçando as preocupações de confiabilidade em tempo de execução de forma autonómica.

A adoção destes padrões depende de aspetos arquiteturais e funcionais dos sistemas. Um dos aspetos limitadores é que a maioria dos sistemas e soluções atuais não fornece nenhum mecanismo para reajustar o comportamento do sistema durante o tempo de execução. Dado isto, os paradigmas de computação *fog* e *edge* foram explorados de forma a aproveitar os recursos computacionais em todas as camadas do sistema, com o objetivo de tornar os sistemas mais confiáveis e mais escaláveis. Com estas contribuições de base, exploramos e afirmamos a

viabilidade de usar funções serverless no contexto de IoT, otimizando a escolha de contextos de execução de acordo com preferências, restrições e latências.

Para melhor entender como é que estes paradigmas podem ser potencializados para soluções amplamente utilizadas, selecionamos Node-RED como caso de estudo, dado ser um dos sistemas de desenvolvimento open-source mais utilizados. Este fornece uma interface de programação visual que potencia que utilizadores com diferentes níveis de conhecimento técnico o possam usar. E, tal como outras soluções, o Node-RED não fornece mecanismos para orquestrar tarefas entre dispositivos nem lidar com falhas de partes do sistema dinamicamente, limitando a confiabilidade dos sistemas construídos.

Começamos por avaliar empiricamente, tanto em configurações virtuais como físicas, a viabilidade de usar o Node-RED como um orquestrador, onde as tarefas computacionais são alocadas para os recursos disponíveis e as falhas de dispositivos são mitigadas com re-orquestrações. Também foi desenvolvido um conjunto de extensões para Node-RED que permitem enriquecer os programas existentes (*flows*) com mecanismos de self-healing — permitindo a deteção de diferentes erros durante o tempo de execução e reajustando o comportamento do sistema para este manter a sua operação dentro de níveis aceitáveis de Qualidade de Serviço.

Dado que os utilizadores de IoT têm diferentes níveis de conhecimento técnico, tentamos melhorar a interação com os ambientes de desenvolvimento destes sistemas para que os utilizadores pudessem melhor entender quais são as automações configuradas (viz. inspeção), como estas se estão a comportar (viz. observabilidade e *feedback*), e aumentar a sua capacidade de perceber qual foi a possível causa por trás de certos eventos (viz. causalidade). No primeiro estudo, começamos por estender as notações visuais e as funcionalidades do Node-RED para que pudéssemos melhorar o processo de desenvolvimento. De seguida avaliamos empiricamente o desempenho da solução desenvolvida em relação a uma versão do Node-RED sem modificações, observando melhorias estatisticamente significativas na capacidade dos utilizadores de evoluir sistemas de IoT existentes. Por fim, exploramos o uso de assistentes de voz como maneira alternativa de configurar, compreender e interagir com ambientes IoT, focando-nos particularmente na capacidade de um utilizador perceber a causa por trás de alguns eventos.

Verificamos a viabilidade do uso de sistemas de voz como forma alternativa de desenvolvimento de sistemas IoT, cobrindo todas as diferentes possibilidades de automação que o Node-RED suporta, com uma extensão considerável das possibilidades de interação devido ao suporte de diálogos multimensagem. Passamos a validar a viabilidade dos utilizadores usarem o assistente de voz para realizar diferentes tarefas, verificando que todos os utilizadores foram capazes de terminar as tarefas. Enquanto algumas frases válidas foram incorretamente reconhecidas pelo mecanismo de reconhecimento de voz, o que forçou alguns participantes a repetir a sua intenção, os participantes expressaram uma preferência por interfaces de voz preterindo interfaces visuais em termos de perceção subjetiva.

Essas contribuições materializam-se num conjunto básico de blocos de construção que, em conjunto, podem ser usados para melhorar a confiabilidade dos sistemas de IoT, ao mesmo tempo que potencializam a capacidade do utilizador de os configurar, usar e desenvolver sistemas deste tipo. As contrapartes experimentais destas contribuições fornecem evidências empíricas que sustentam a plausibilidade da hipótese.

Contents

List of Figures

List of Tables

Algorithms and Code Snippets

Acronyms

AAL	Ambient Assisted Living
A/C	Air conditioning
AC	Alternating Current
ADC	Analog-to-digital converter
API	Application Programming Interface
AWS	Amazon Web Services
CD	Continuous Deployment
CE	Conformitè Europëenne
CEP	Complex Event Processing
CI	Continuous Integration
CPS	Cyber-physical Systems
DC	Direct Current
DNS	Domain Name System
DSL	Domain Specific Language
DSML	Domain Specific Modelling Language
FCC	Federal Communications Commission
HTTP	Hypertext Transfer Protocol
IIoT	Industrial Internet-of-Things
I/O	input/output
IoT	Internet-of-Things
IP	Internet Protocol
ISO	International Organization for Standardization
JSON	JavaScript Object Notation
MDSE	Model-Driven Software Engineering
MQTT	Message Queuing Telemetry Transport
MTBF	Mean Time Between Failures
MTTF	Mean Time To Failure
MTTR	Mean Time To Repair
MWS	Minimal Working System
N/A	Not Applicable or Not Available
NATO	North Atlantic Treaty Organization
NFC	Near-field communication
NLP	Natural Language Processing
OTA	Over-the-air
PLC	Programmable logic controller
POC	Proof of Concept
pub/sub	publish–subscribe
QoS	Quality-of-Service
REST	Representational state transfer
RTOS	Real Time Operating System
SBC	Single-Board Computer
SDLC	Software Development Life-Cycle
SDN	Software-defined Networking
SDR	Software-defined Radio
SoC	System on a Chip
SoS	Systems of Systems
SPOF	Single point of failure
SSH	Secure Shell
SUT	System Under Testing
TAP	Trigger-Action Programming
TCP	Transmission Control Protocol
UI	User Interface
URL	Uniform Resource Locator
VPL	Visual Programming Language
VPS	Virtual Private Server
WoT	Web of Things
WSN	Wireless Sensor Networks
WWW	World Wide Web

Preface

> This is our world now… the world of the electron and the switch, the beauty of the baud.
>
> *The Conscience of a Hacker*, The Mentor, 1986

I always had this thing for *engineering*, what I see as the practical side of *science*, of designing and building, of problem-solving, and of *hacking*. But not so much for informatics and computers, at least in the *beginning*. The first things that I remember to build were these little toy houses with construction materials that I *borrowed* from the near construction site. Building these little toy houses gave me an enormous feeling of achievement, as I built them with sand and cement in a way that they could outlive me for far more years. However, things change quickly, and what was once toy houses is now a little garden.

A few years later, I got my first computer, a Pentium 4, which provided me some hours of entertainment playing Pinball on Windows XP. However, there was not so much information available on how it worked nor what it was capable of beyond playing what seemed simple games. Since the Internet was still not commonly available, the only way to get new *stuff* for the computer was using 3.5" floppy diskettes to transfer data from the old Internet-connected computers that existed, at the time, in the parish council building back to my home computer. Quickly after, I *broke* my computer. After being repaired at some IT store, it took me less than a month to break it again. But this time I could not tell my parents again, since it was so shortly after the last time. So the problem arose, *how could I fix it?* I had recently read something about formatting the hard drive as a way to get the computer to a clean default state. After finding the Windows XP CD, I put it in, followed some instructions in scary white letters over blue screens, and I managed to restore the computer. However, for some reason, there remained a naggy window saying "This copy of Windows did not pass genuine validation". That scared me, but after getting to the Internet, I found out that there was some magic register key that could be changed using the *Registry Editor*, that would validate Windows and make the nagging window go away. *And it was.* Operating systems, like Windows, are getting every day more complex and harder to break — and more limited in terms of ownership and modification ability —, and there are more safeguards and built-in diagnostic and repair tools than ever.

I was a *skid* back then, at least more than I am now. But I was able to do things rapidly, which provoked immediate results (even if more perishable than my toy houses). Fast-forwarding, Windows XP became legacy software, and Internet not only became a commonplace, but it is no longer a desktop computer only thing, but it is now present in the *handled terminals* that we carry in our pockets. With the reduction of the size of *computers* with connectivity, it was a matter of time to have computers everywhere, not only in our pockets but in our watches, in the coffee machines, in the lightbulbs, …

Nevertheless, I know well the constant struggle to get a computer printer to work when needed, even if it is a product available since the 1930s. Although the amount of investment and *innovation*, the problem remains. The question arises, are we — and should we be — ready to have similar issues with our coffee machine?

I, for one, think that this is not a problem of knowledge. We, as a species, build systems such as the NASA's Voyager 1 and Voyager 2 spacecraft that are able to continue working for more than 40 years straight, being now at an approx. 22.5 billion km away from earth. Even so, they are not *perfect*, and malfunctions appear. The difference lies in how the system is prepared to deal with such issues. At the beginning of 2020, an anomaly happened in the Voyager 2 spacecraft, resulting from an *unexplained delay in the onboard execution of the maneuver commands inadvertently left two systems that consume relatively high levels of power operating at the same time.* However, soon after, the spacecraft was stable once again thanks to the built-in *fault protection software routine*, which turned off the operating science instruments to compensate the power deficit [NAS20].

The problem with the IoT is a different one, as Sean Smith puts it in the book *The Internet of Risky Things* [Smi17]:

> *If we build this new Internet the way we built the current Internet of Computers (IoC), we are heading for trouble: humans cannot effectively reason about security when devices become too long-lived, too cheap, too tightly tied to physical life, too invisible and too many.*

I would rewrite the sentence to focus not only on security, but also on safety/dependability. IoT systems have been typically built in the same fashion as software-only systems, mostly disregarding the accumulated knowledge from other fields, such as mission-critical space systems. Such disregard does not come without motivation, since, as it happens to software-only systems, IoT devices and appliances are under similar economic forces that make products to be rushed to market with insufficient testing [Smi17]. Moreover, while the hardware of some of these devices has, typically, to be certified under the current standards (*e.g.*, European CE and American FCC marks), this is not common for the software counterparts.

In the same way that the Voyager spacecrafts were built, sharing the same software/hardware inter-dependency nature, these new systems must be trustable, potentially capable of outliving their creators, with automatic built-in ways to avoid — or at least reduce — the impact that their malfunctions have in the people's everyday lives. Things should not stop working because the manufacturing company entered in bankruptcy, nor because their genuine usage license ended (*they are things, not subscription-based software*). This implies a mind-shift from vendors, that instead of trying to make it harder — or impossible — for *hackers* to build their own *smart home* systems (while mining their users' privacy for profit), should be leveraging and encourage open standards, interoperability, and repairability on their devices and systems. This dissertation does not go by any means close as to give answers to all of these issues and problems, but, by *standing on the shoulders of giants*, attempts to give some *engineering* insights on how to address some of them while being supported by *scientific evidence*.

Acknowledgments

First and foremost, I am utmost grateful to my supervisor and friend, Hugo Sereno Ferreira, for all the mentorship, guidance, endless unusual ideas, discussions, tinkering, and unconditional support. Without your constant incentives to aim higher and confidence in my capabilities to

pursue such goals, this work would not be possible. My token of appreciation to my second supervisor, João Pascoal Faria, whose experience and availability played an essential role in this work.

To André Restivo that during this thesis became to be like a *third* supervisor as the shared authorship in several of the published articles part of this work proves, but also a friend. The endless discussions, most shared with Hugo, and the countless *almost* all-nighters revising articles and discussing research directions played a crucial role in achieving this work. Your aesthetic input and skills resulted in some of the most artistic illustrations of this thesis, and for these reasons and many others, my sincere appreciation.

To Filipa, to whom I am grateful for the patience and efforts to keep me at *acceptable sanity* levels through the years, hoping to share with you the years to come. I am also grateful for my family's unconditional, unequivocal, and loving support during all this journey.

To all my friends* that supported me through this journey, and had the patience to keep listening to my constant rants about academia and my research endeavors. A token of appreciation for the ØxOPOSEC Meetup community that allowed me to share the stage with fellow hackers and tinkers, listening to my ramblings about IoT, and where I found several people sharing the same *curiosity* about computing systems, security, and privacy.

To all the others, Professors and students, that collaborate directly or indirectly with me in this research, being co-authors of at least one publication or related master's thesis, in alphabetic order: Ademar Aguiar, Ana C. R. Paiva, Andreia Rodrigues, André Sousa Lago, Ângelo Martins, Bruno Lima, Bruno Piedade, Diogo Amaral, Diogo Torres, Duarte Duarte, Duarte Pinto, Filipe F. Correia, Flávio Couto, Gil Domingues, Guilherme V. Pinto, José Magalhães Cruz, José Pedro Pinto, José P. Silva, Luís Reis, Margarida Silva, Miguel P. Duarte, Pedro Costa, Pedro Lourenço, Rosaldo J. F. Rossetti, Rui Nóbrega, Tiago Boldt Sousa, Tiago Fragoso, and Zafeiris Kokkinogenis.

I also acknowledge the financial and operational support from the Foundation for Science and Technology (FCT), the Faculty of Engineering of the University of Porto, and INESC TEC. A special thanks to the Department of Informatics Engineering (DEI) where I have been lecturing for the past four years, and to the Professors who have put their confidence and trust in me to lecture their practical classes, namely, Jorge Alves Silva, José Magalhães Cruz, Ademar Aguiar, and, again, Hugo Sereno Ferreira and André Restivo. A last word of appreciation to Pedro Miguel Silva, Sandra Reis, and the remaining staff of the department for guiding me through the kafkaesque bureaucracies of academia.

Feedback

Any feedback on this dissertation, including ideas for future work or collaborations, are welcome. Please feel free to find me at http://jpdias.me and contact me via jpmdias@fe.up.pt for any questions and comments.

*A special thanks to the members of the underground chat groups whose ramblings and conversations about the most diverse subjects helped to lighten the mood, especially during the SARS-CoV-2 age.

1 | Introduction

As a mostly direct consequence of the evolution of computing capability *versus* size, in accordance to Moore's Law — *"the number of transistors in an integrated circuit doubles approximately every two years"* — the creation of ever-smaller computing-able devices has opened doors to ubiquitous computation, making possible to embedding logic in everyday devices [Pu11]. This possibility, along with the explosion in networking solutions and protocols way beyond the traditional TCP/IP stack, boosted the concept of ubiquitous connectivity, making the possibility of having Internet-connected devices at large. This computation and connectivity outreach has enabled the so-called Internet-of-Things (IoT), creating a wide range of applications across industries and domains, posing challenges for both academia and industry, mainly due to the cross-domain nature of the field, which calls for a need of software-hardware co-design [Bon08], since pending challenges in both hardware- and software-level research have the potential to influence the field's evolution. As some fear a new software crisis [Fit12], others have been aggravating the knowledge gap between different parts of the IoT ecosystem by creating an ever-growing fragmentation [Gui16]. This work delves on the challenges and current practices for IoT development mostly from a software engineering perspective but without completely disregard the hardware constraints that mold the possibilities of building dependable IoT systems are. The presented research contemplates the systematization of widespread knowledge across domains that influence these systems, empirical insights on how different development solutions fit the field, and how different strategies — from computation distribution to autonomic computing — can improve IoT systems.

1.1 On the Analog and Digital Worlds

The common definition of a computer is the one of a "programmable electronic device that can process, store and retrieve data" [ORe16]. The first computers were *analog*, where data is represented by physical quantities' (*e.g.*, electric voltage). *Digital* computers are based on binary digits, processing data one step at a time. Their fundamental architecture remained the same since the work of von Neumann, Eckert, Mauchly, and others on the design of the Electronic Discrete Variable Automatic Computer (EDVAC), the successor of the Electronic Numerical Integrator and Computer (ENIAC)[1]. Moreover, while the original computers were expensive and filled entire rooms of space, today, as envisioned by Gordon Moore (*i.e.*, *Moore's law*), computers are smaller, powerful and cheaper [ORe16].

Digital computers consist of two essential parts: hardware, the physical part of the machine, and software, the set of instructions that specifies what the computer does. The software itself can be divided into two major classes: system software, which drives the computer hardware and computer system itself — including operating systems, device drives, and other — and application software that provides services to the users — *e.g.*, word processing, web browsing, and others. Typically, both kinds of software are written in high-level programming languages, which are then later converted to *machine code*, and are easily replaceable — *e.g.*, upgrade the operating system version or change the default web browser — thus being more temporary and malleable.

Nevertheless, as computers get more complex, with hardware that can go from advanced peripherals to personalized hardware accelerators, the need for specific software that allows them to have some level of *hardware-independence*, so-called *firmware*, which is device-specific software, arises, typically specified in a low-level language. Firmware is not limited to hardware counterparts of computers. It can act as a device's complete operating system, performing all control, monitoring, and data manipulation functions — *e.g.*, the embedded software that runs in the PLCs in industrial shop floors. Historically, the most significant difference from firmware to software has been the ability to change it after being programmed into a devices' non-volatile memory (*e.g.*, ROM, EPROM, or EEPROM), requiring a higher level of skill, effort, and cost to do so.

As computers became ubiquitous and the range of applications skyrocketed, especially in terms of highly-specific constrained devices, it became a common need to replace the firmware of the devices after being manufactured, *e.g.*, for bug fixing or adding new features. With the IoT, it became common practice to update the software that is running on, for example, *smart watches* or *smart lightbulbs*.

The empowering of low-cost devices with easy programming features, and high connectivity, resulted in distinguishing features of what has been called firmware became closer to a regular piece of software, easier to update, with more complexity and fewer restrictions.

As the line between firmware and software became more blurred, the separation between the analog world and the digital has also shortened. While the software is intangible, its ability to perform changes in the real world by analog means was limited to some fields of application

[1] EDVAC major differences to ENIAC was the use of binary arithmetic instead of decimal and its stored-program computer design.

such as robotics and industrial systems. With IoT it became common, and one of its utmost importance characteristic, that the analog aspects of the real-world — seen and measured in terms of light, pressure, temperature, proximity, and others — play a direct role on the software that runs on the devices and *vice-versa* [Avn19].

The common barrier between hardware and software is also shifting. The birth of Software-defined Everything (SDE) has been expanding and growing the importance and impact of software, becoming a key and essential part of the technological world [BC15]. The dissemination of Software-defined Networking (SDN), Software-defined storage (SDS), and Software-defined data centers (SDDC) as core components of Cloud computing are strong indicators of the Software-defined tendency. Despite this, it is not limited to the scope of Cloud computing and similar, one can observe that approaches like Software-defined Radio (SDR) are evidence that even components that have been traditionally implemented in hardware are now being implemented in software [Tut03].

Developers that want to develop on top of IoT systems need to have a deep knowledge of layers much closer to the hardware, and even take into account the impact that the *real world* has on its operation. Also, similar knowledge is needed due to the *software-based everything* tendency and its close relation with the IoT ecosystem.

1.2 From the Internet-of-Computers to the IoT

Nearly five decades have passed since the first public demonstration of the ARPANET[2], the world's first packet-switched network, that connected more than 30 institutions computers'. Soon after (1973), with satellite-based radio networks, the need for connecting the ARPANET with other networks appeared, which lead to the definition of well-established protocols on how these computer networks should work, namely, the Transport Control Protocol (TCP) and the Internet Protocol (IP) — also known as the TCP/IP network stack. In the mid-1980s there were around 2000 hosts on the TCP/IP network [HL96; ORe16].

In the late 1980s, the decision to shut down the ARPANET was made, and the National Science Foundation (NSF) commenced work on its successor with higher data transfer speeds, the NSFNET, which later became known as the Internet. The Advanced Network Services (ANS) not-for-profit company was formed circa 1991, and it replaced the NSFNET with a new network, the ASNNET, which was a distributive network architecture operated by several commercial providers [ORe16]. Later, in the 1980s, there were over 160000 Internet-connected computers.

A turning point was the invention of the World Wide Web, circa 1990, by Sir Tim Berners-Lee while working at the European Organization for Nuclear Research (CERN), by introducing the concept of Universal Resource Locator (URL) to access pages formatted in the Hypertext Markup Language (HTML) over a Hypertext Transfer Protocol (HTTP) on top of the already existing TCP/IP networks. A client application — browser — allows users to view the HTML pages and interact with them [ORe16].

[2]The ARPANET acronym is based on the name of the agency that coordinated the project efforts; namely, the US Department of Defense founded the Advanced Research Projects Agency (ARPA), nowadays known as Defense Advanced Research Projects Agency — DARPA.

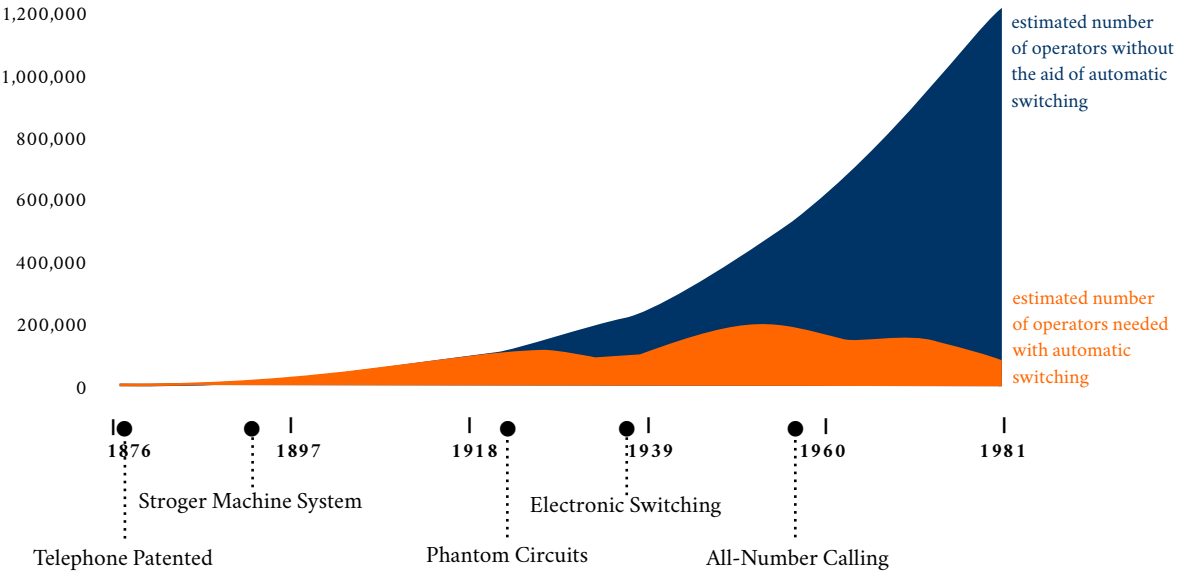


Figure 1.1: AT&T/Bell System's introduction of automated switching protocols in the 1920s allowed it to meet operational demands without requiring manual operators [Hor01].

The *Internet*, as of today, corresponds to the vastness of applications and protocols that run on top of advanced and interconnected computer networks, available 24/7, serving above 4.66 billion users worldwide circa January 2021 [sta22]. Although its reach, the Internet was, historically, mostly made of computers and servers, so-called the Internet-of-Computers (IoC) [FF11; Smi17].

As predicted by Mark Weiser in the early 1990s [Wei02], an era of ubiquitous computing and connectivity is here, as *processors, communications modules and other electronic components are being increasingly integrated into everyday objects* [FF11]. This shifted the world from an IoC to an Internet-of-Things, which points towards a *seamless integration of people and devices to converge the physical realm with human-made virtual environments* [BD16].

The IoT spread quickly, with the first scientific conference in the area happening in 2008 [Flo+08]. In 2009 a dedicated EU Commission action plan presented IoT as a general evolution of the Internet — "from a network of interconnected computers to a network of interconnected objects" [Eur09].

1.3 The Role of Automation

IBM, circa 2001, in their report IBM's Perspective on the State of Information Technology, identified the complexity of computing systems as *"the single most important challenge facing the IT industry"* [Hor01; GC03]. By creating increasingly powerful computation systems to automate essential task and processes, systems have become increasingly complex as a byproduct [PD11; Hor01].

Historically, the complexity created by this automation was mainly attained and managed by human intervention and administration. However, as the number of systems, devices, and

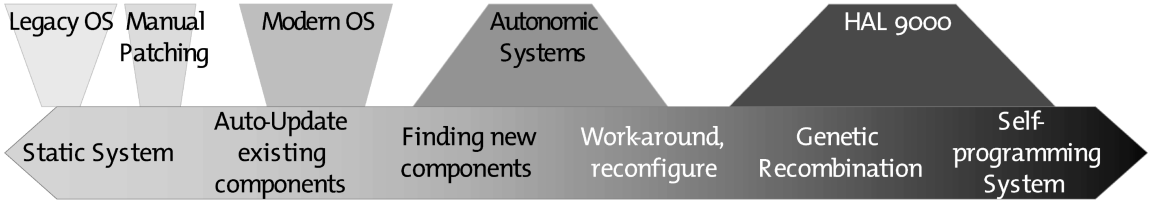


Figure 1.2: Self-management continuum as presented by Fink *et al.* [FF07]. *HAL 9000* is a fictional artificial intelligence character in Arthur C. Clarke's Space Odyssey series.

applications keep increasing[3], the ability to manage the resulting complexity with traditional approaches is impracticable, mainly due to the lack of enough skilled personnel in IoT and IT in general [Mic19; Hor01]. Similarly to what happened around 1920 with the introduction of the automated switching protocol in telephony (*cf.* Figure 1.1, p. 4), the introduction of IoT across domains will, potentially, lead to similar scenarios where most manual tasks will be automatized, being this automatization already noticeable in industrial scenarios where new ways of operation are being drafted as a result of the IoT opportunities [LLD18].

Nevertheless, even in a situation with enough skilled individuals, complexity is outreaching the human ability to manage it [Hor01; Smi17]. As Paul Horn stated back in 2001: *"As computing evolves, the overlapping connections, dependencies, and interacting applications call for administrative decision-making and responses faster than any human can deliver. Pinpointing root causes of failures becomes more difficult while finding ways of increasing system efficiency generates problems with more variables than any human can hope to solve."* [Hor01].

Automation plays a crucial role in different perspectives. Examples are common, such as the following: (1) for an end-user of an IoT system, one of the most valuable features is the ability to automate recurrent tasks that have historically been performed manually (*e.g.*, turn on the air conditioning (A/C) system when the temperature drops below a given threshold); (2) for a cloud computing administrator, a key feature is the ability to configure a deployed system to automatically scale its resources (automatic provisioning) to meet service demands; and (3) for a developer, every time a new feature is developed the deployment process is automatized in pipelines that automatically check for code quality, verifications (test automation) and re-deploy the system with the new feature without downtime (continuous integration and delivery).

Inspired by the autonomic nervous system of the human body, IBM Research introduced the autonomic computing initiative [Hor01]. The main focus of autonomic computing is to progressively make computing systems more self-managed, hiding the intrinsic complexity of the systems away from operators and other users. Systems should also be capable of adapting to unpredictable changes in their operational environment while *increasing predictability, speed of response, and reliability* of the computing systems [FF07]. Envisioning a pervasive computing world (*i.e.*, IoT), autonomic computing becomes paramount, allowing the computers in our environment (from small to large units) to constantly adjust to our needs [FF07].

[3] "By the end of 2018, there was an estimated 22 thousand million Internet of Things (IoT) connected devices in use around the world" [Tan18].

Fink *et al.* in their self-management continuum vision circa 2007 (*cf.* Figure 1.2, p. 5), present an evolution view from the legacy — mostly static — systems to autonomic computing while identifying the current systems (*Modern OS*) as a stepping stone towards an autonomic world. The authors also envision a future, inspired by the *HAL 9000* fictional artificial intelligence figure, with computing systems that can re-program themselves.

1.4 Software Crisis and the Technology Fragmentation

Back in 1958, the statistician John W. Tukey was the first author to publish the term *software*[4], extolling its importance in a world that was, at the time, taking the first steps away from analog systems [HKN02]. Soon after, the term *software engineering* was coined by Anthony Oettinger[5] and then used as the title of the first conference on software engineering sponsored by NATO's Science Committee. The conference that leaned on the issues regarding software's development and delivery witnessed the birth of the phrase *software crisis* [NR68].

As Prowell *et al.* states, despite what is known about software engineering, software fails, and a frequent source of this problem is the use of informal approaches of developing software, *viz* [Pro+99]:

> *The vast majority of the software today is handcrafted by artisans using craft-based techniques that cannot produce consistent results. These techniques have little in common with the rigorous, theory-based processes characteristic of other engineering disciplines. As a result, software failure is a common occurrence, often with substantial societal and economic consequences. Many software projects collapse under the weight of the unmastered complexity and never result in usable systems at all.*

Over the years, several studies had confirmed the Software Crisis 1.0., establishing that the first indicators of such crisis arose in the 1960s, with software taking longer to develop, costing more than estimated, and not working correctly when eventually delivered [Fit12]. As an example, an IBM study circa 1994 reported that 68% of all software projects overran their schedules and, also, the initial estimated budget was surpassed in 55% of them all [Mis11].

Although the consecutive CHAOS[6] reports [Sta15] do not reveal a significant shift in the scenario, with a 29% success rate of software projects, challenged rate of 52% and 19% impaired (canceled) in 2015 (back in 1994 the report pointed out a 16% success rate, 53% challenged and 31% impaired)[7]. Brian Fitzgerald states that, as of today, we are beyond the Software Crisis 1.0., and that the myriad, incremental advances improving software development over the past 50

[4] John W. Tukey, "The Teaching of Concrete Mathematics", American Mathematical Monthly 65/1 (1958): 1-9 at 2: "Today the *software* comprising the carefully planned interpretive routines, compilers, and other aspects of automative programming are at least as important to the modern electronic calculator as its 'hardware' of tubes, transistors, wires, tapes and the like."

[5] There is some discussion around who has coined the term *software engineering*, with some giving such credit to Margaret Hamilton, lead NASA flight software designer for Apollo missions, circa 1961 [Com17].

[6] CHAOS acronym stands for *Comprehensive Human Appraisal for Originating Software*.

[7] Although the Standish Group findings and methodology have been challenged [EV10], it seems that either their results are heavily biased, or even a moderate change in the accuracy of the success ratio, *e.g.*, from 29% to 50%, would probably still render the field as in crisis.

years have *changed our lives for the better* [Fit12]. He also states that now software is routinely developed largely on time, within budget, and within user expectation [Fit12]. Nevertheless, Brian Fitzgerald points out a new crisis, a Software Crisis 2.0. [Fit12]:

> *The demand for data from digital natives, coupled with the huge volume of data now generated through ubiquitous mobile devices, sensors, and applications, has led to a new software crisis.*

On the one hand, resulting from the significant advancements in hardware capability (processing power, storage) over the last years, allied with the dramatic reductions in costs, a proliferation of devices was bound to happen, shifting from a world of traditional computers to an era of ubiquitous computing, with computing-able devices widespread among everyday devices, playing a more profound role on people's everyday life. Altogether with the dissemination of data-collecting sensors and applications, the result is a heavy *push* factor towards a new crisis. One can easily associate the birth of the term Internet-of-Things as a direct consequence of such dissemination. On the other hand, the desire to consume new technology by the *digital natives* becomes an enormous *pull* factor for such a crisis. For Fitzgerald, these are the disruptive factors for the Software Crisis 2.0., since the software engineering field has not seen similar advances in the last years that can attend to the needs of such landscape [Fit12].

> We face a new software crisis. In 1968, computer scientists learned that developing robust software requires skills, methods, and tools. Today, software and hardware engineers realize that developing a robust Internet-of-Things also pushes the states of their art and practice. Recent news illustrates the many problems faced by IoT: from lack of interoperability to broken updates to massive security attacks.

As we enter an epoch of ubiquitous computing, with computation-able devices being present anytime and everywhere, mostly due to the *push* and *pull* factors aforementioned, we are also witnessing a widespread fragmentation of the technological solutions in the market. The co-existence of multiple highly-incompatible technologies and technology stacks leads to two main implications: it forces the technology users to commit to an entire product ecosystem, as in vendor *lock-in* and vastly increases the efforts need by software developers to develop systems that encompass different technologies and technology stacks [Mic19].

Similar to any other paradigm-shift, and, in this case, from the perspective of the software engineering community, there is no consensus, or standards, on what are the best practices for developing for IoT systems. Further, it is even noticeable that the creation of standards, instead of solving the problem, leads to *race* between competing ones, thus not solving the original problem and contributing to an even more significant fragmentation[8]. This leaves several questions open such as, how to select the more suitable architecture, which are the more reliable and optimal communications protocols for each scenario, and what are the best approaches in terms of security and privacy.

[8] As pointed by Andrew S. Tanenbaum in his book COMPUTER NETWORKS circa 1981: "The nice thing about standards is that you have so many to choose from; furthermore, if you do not like any of them, you can just wait for next year's model." [Tan02].

As envisioned by Fitzgerald circa 2012, recently several authors have been raising awareness to the problems of designing and building software for the IoT, focusing on the complexity of these systems [KW17] empowered by the lack of standardization [Spi17] and lack of suitable development methods, languages, and tools [TM17]. While these problems can appear nothing more than mere troublesome for developers and other technical users that have to design, construct and test these systems, the side effects of building these systems poorly and not-future-proof lead to several issues with a most direct impact on the comfort, well-being, and safety of the system' users, ranging from privacy and security issues to other system malfunctions with origins in both software and hardware [Smi17].

1.5 Complexity: Essential *versus* Accidental

Brooks has established the concept of *complexity* in the context of software development circa 1986 [Bro86]. As he considers that there is *no silver bullet* to address systems' complexity, he also considers that there are two different types of complexity: *essential* and *accidental*.

Complexity, resulting from particularities such as *heterogeneity*, *large-scale* and a wide range of application scenarios of IoT can be considered as *essential* complexity, given that it is complexity which the developer must always face from the beginning of developing on or for such systems, which is inherent and unavoidable. In contrast, *accidental* complexity appears as a result of dealing with the *essential* complexity, viz. a specific approach was chosen to solve a specific problem. Such complexity arises in computer artifacts, or their development process, which is non-essential to the problem being solved.

The ever-increasing complexity (both inherent and accidental) of building and managing software for the IoT landscape leads to the constant creation of solutions by both the industry and the scientific community, being disseminated by a wide-range of mechanisms (*e.g.*, books, research papers, web pages and other types of communication media).

From the known problems of software engineering that led to the birth of the term *software crisis* until the issue of technological fragmentation, it is noticeable the complexity of the task that is software development. Furthermore, when considering the details of the IoT paradigm, new developer-facing challenges appear, such as the need of dealing with *heterogeneity* and *large-scale* inherent to it, plus, on some occasions, the need of having considerations about lower-level details of hardware systems, it is noticeable the emergent *complexity*.

This fast-growing knowledge body should be the base for an engineer to choose the *best design* for a *specific situation*, but, as Christopher Alexander pointed out, any specific body of knowledge is "hard to handle, widespread, diffuse, unorganized, and ever-changing" [AIS77]. Thus, developing software for IoT that is secure, interoperable, modifiable, and scalable becomes a challenge since developers are unable to understand which are the best practices and the most suitable architecture and communication protocols to be used [Dig+19].

1.6 Towards Dependable Systems

Avižienis *et al.* [Avi+04] in their paper Basic Concepts and Taxonomy of Dependable and Secure Computing circa 2004 introduce dependability as a global concept that subsumes the attributes of availability, reliability, safety, integrity, confidentiality, and maintainability[9]. A dependable system is a system that is able to (1) *"deliver service that can justifiably be trusted"* and (2) *"avoid service failures that are more frequent or more severe than is acceptable"*. The dependability of a system is threatened by faults, which can be (1) development faults, (2) physical faults, and (3) interaction faults.

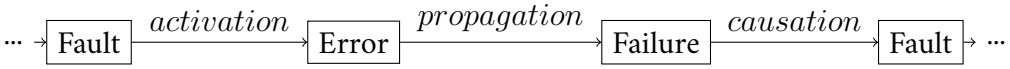


Figure 1.3: Fundamental chain of dependability threats. The arrows express a causality relationship between faults, errors, and failures, but they should be interpreted generically since that, by propagation, several errors can be generated before the occurrence of a failure [ALR01].

Faults, when activated, lead to errors — errors being the deviations from correct service. The propagation of errors leads to failures — a failure being the delivery of incorrect service. This is known as the chain of threats as introduced by Avižienis *et al.* [ALR01], and is also known as chain `Fault`→`Error`→`Failure` (*cf.* Figure 1.3, p. 9) as introduced by Jean-calude Laprie circa 1992 [Lap92]. Although different fields of study are known to consider different definitions of what a fault is, error, failure, and their relationships, for the purposes of this work, we will consider these concepts as they are introduced and defined by Avižienis *et al.* in their works [ALR01; Avi+04].

When designing a dependable system, one of the key objectives is to reduce the probability of a system deviating from delivering the correct service. Fault prevention, tolerance, removal, and forecasting techniques have been used across application domains to improve system dependability. While these techniques were, historically, applied and based on hardware-only mechanisms, nowadays, it also includes software-based techniques [Han07]. A particular example of the crucial role these techniques play are mission critical systems — *e.g.*, manufacturing floors, aerospace industry, emergency services — which malfunction can endanger people's well-being and comport significant financial losses.

As the systems' complexity increases — even originating Systems of Systems [ASD16] — attaining dependability becomes an ever-more arduous task due to the ongoing increase in the moving parts of the system — either by design or through their lifecycle. This counterpoints with hardware-only systems that were mostly set in stone and suffer little to no modifications (beyond maintenance ones) through their lifecycle. Further, while historically systems were deployed, used, and maintained by personnel with specific technical expertise, computer systems — including IoT ones — are nowadays used by everyone without requiring specific knowledge.

Ensuring always correct service becomes unattainable as the system complexity increases, a fact which led to principles such as graceful degradation — *"ability of a system to automatically*

[9] Security is defined as a subset of dependability with a focus on the confidentiality, integrity, and availability attributes (also known as the CIA triad).

decrease its level of performance to compensate for hardware and software faults" [Joh89] — to gain traction, reducing the impact of non-recoverable (or slow to recover) failures. However, what services can be degraded or not to best fulfill the user needs and requirements, *i.e.*, Quality-of-Service, seems to be an open question. Prioritizing services that endanger the well-being can be consensual, but beyond that, it becomes a user-specific and use-case-specific question; thus, the *one size fits all* approach may be unsuitable — even when considering the additional management efforts and costs it could bring to practitioners.

Among the fault-tolerance approaches that have been used, IBM's concept of self-healing — as part of their autonomic computing vision [Hor01] — has been increasingly adopted by both the research community and industry. One of the main drivers of this adoption is the possibility of maintaining larger connected computer systems through automation of their management, *i.e.*, defining autonomic behaviors to maintain or restore a system's health without (or with minimal) human intervention.

As IoT systems permeate our physical surroundings with ever-growing complexity, several authors suggest that automation, including self-healing, is the most viable solution to ensure minimum levels of QoS [Ang15; DAA16; SHA17; AA19; SBC20]. However, what is the most suitable balance between the user intents and such automations? Should users be limited in what they can *program* their system to do if it compromises the system dependability? As these questions remain unanswered, in this work, we attempt to aid, inform, and explain how to improve their system dependability without limiting what they can *program* or not.

1.7 Motivation and Scenarios

Even though the disseminated prospects of what has been described as an IoT utopia, there is a widespread discussion on the utopia/dystopia dichotomy of IoT. Some raised concerns fall within the scope of the noticeable technology fragmentation [Gui16], but also refer to the devices' numbers and *moving parts*, their expected lifetime, their intertwinement with the physical life, the influence of the economic forces, and the unprecedented attack surface size [Smi17].

To exemplify and further illustrate some central ideas and concepts of this thesis, let us consider a real-world IoT scenario, a *smart lab*, partially depicted as a diagram on Figure 1.4 (p. 11). Briefly, at one end, we have a space filled with several devices with different capabilities, from sensing features, acting features, or both. Such devices can partially or fully control the space (*e.g.*, door security, lights) and its environment (*e.g.*, temperature, humidity). At the other end, the laboratory inhabitants have a set of appliances that allows them to see the home status information and giving them control over it. An intermediate layer (gateways, routers and other networking and computing equipment) is responsible for connecting the owner to its home, anywhere, anytime. Other parts (internal or external services) enrich the system with external information (*e.g.*, weather) and intelligent services (*e.g.*, automation shopping and delivery based on *smart-fridge* information).

As a motivational example, we can consider the following, only representative, set of sensing-acting trigger rules that play a role in the space:

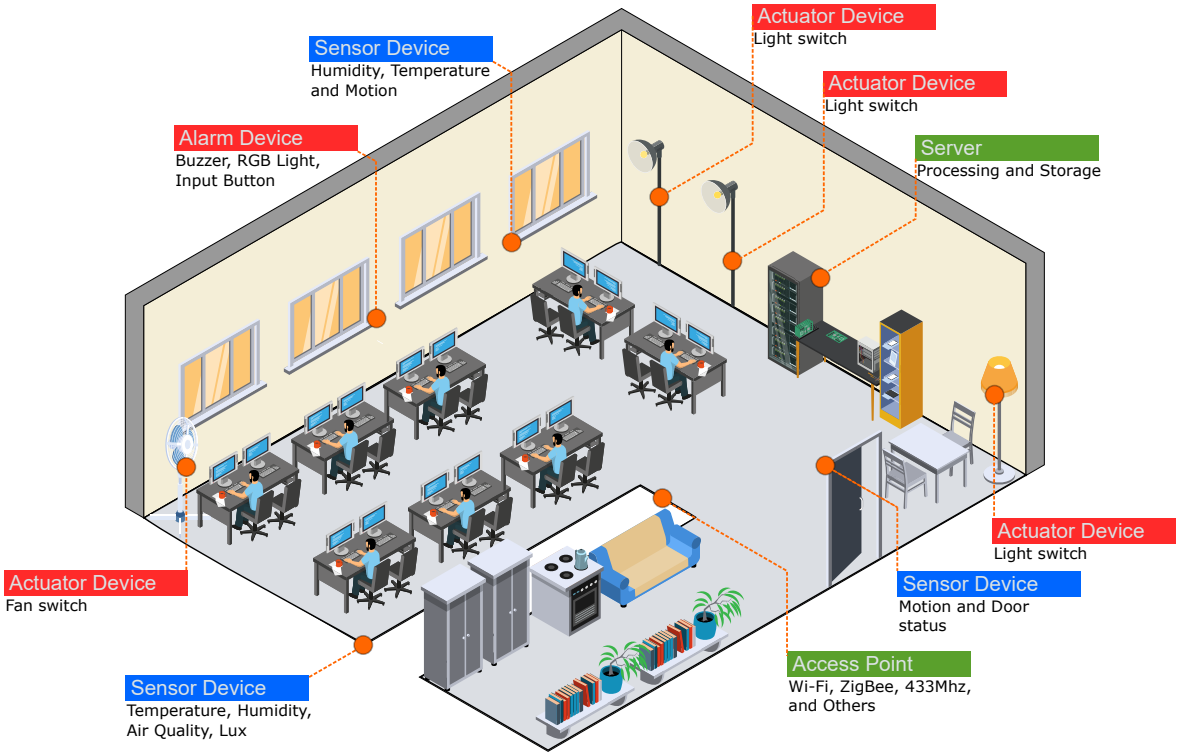


Figure 1.4: *SmartLab* motivational scenario with interconnected sensors and actuators.

- When somebody enters the laboratory (door status magnetic sensor), and it is dark (lux sensor), turn on the lights (light switch actuator);
- If there is poor air quality (air quality sensor), turn on the fans (fan switch actuator). Depending on the hazardousness of the values read, turn on the alarm (buzzer or RGB light actuators).
- Depending on the laboratory temperature — *e.g.*, if greater than 35 °C — (temperature sensor), turn on the fans (fan switch actuator) until the temperature drops — *e.g.*, bellow 30 °C.

Although the *smartlab* can be considered as a base example of the complexity on IoT systems, this complexity increases considerably as the number of devices, people, and services, increases. Consider the *smart home* depicted in Figure 1.5 (p. 12). When compared with the *smartlab*, the number — and heterogeneity — of devices being used is considerably higher, mostly resulting from the different necessities of the different rooms that are part of it (one can consider the *smartlab* as a *single* room). Likewise, the diversity — and number — of sensing-acting trigger rules that can be configured in such *smart space* is considerably higher and, potentially, more complex, opening doors to ever-more hard to understand misconfigurations and unexpected behaviors.

As *things* play a role in the environment, dependability becomes a significant concern. The reliability of the components and the system as a whole can have direct implications on the system's availability, and, thus, the comfort and well-being of its users. There exists a considerable

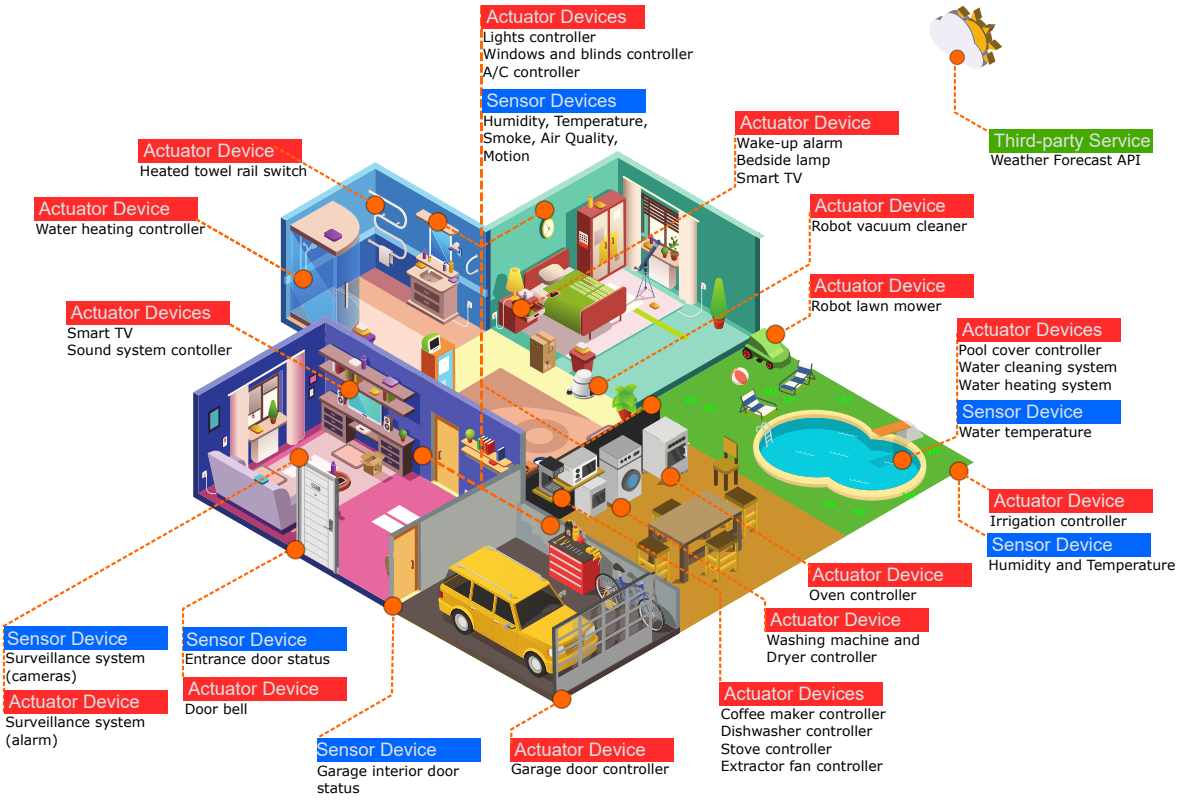


Figure 1.5: *Smart home* motivational scenario with interconnected sensors and actuators.

number of reports which sustain such observation, as the ones in The RISKS Digest, moderated by Peter G. Neumann [Neu20]. The roadmap proposed by Ratasich *et al.* [Rat+19] groups these and other issues in three categories, namely: security (*e.g.*, eavesdropping, jamming, and denial of service), dependability (*e.g.*, data corruption, protocol violations, and misusage) and long-term concerns (*e.g.*, aging and environmental effects and end-of-life — unsupported — hardware or Software).

Although many authors focus on such concerns (as discussed in Chapter 3, p. 64), the deployment and use of their proposed solutions and approaches in scenarios such as the presented one (*smart home*) are most scarce. Using mission-critical-grade components that are more heavily tested and reliable is unpractical due to the increase in cost, development efforts and maintenance complexity (*e.g.*, redundant hardware), which increases development time and testing needs (which can be an issue in time-to-market practitioners' concerns). Nonetheless, dependability concerns must not be disregarded, and *best practices* should be adopted to safeguard the safety and well-being of *smart*-space users.

Although one can argue that these motivational scenarios, Figure 1.4 (p. 11) and Figure 1.5 (p. 12), are mostly isolated systems' without relying on or communicating with any third-entity (at least at a critical level), this is not so-common, due to the typical IoT systems' dependency on external services (*e.g.*, in the event of an alarm being triggered the system must notify the proper authorities and the house owner or validate an individual identity for opening a *smart lock* by checking their online profile).

Let us consider a neighborhood of *smart homes*. The system will scale linearly to the number of houses in it, and, further, considering that we are living among different and co-existing

smart spaces interconnected and, sometimes, inter-dependable, the scale will stop increasing linearly and become closer to exponential growth.

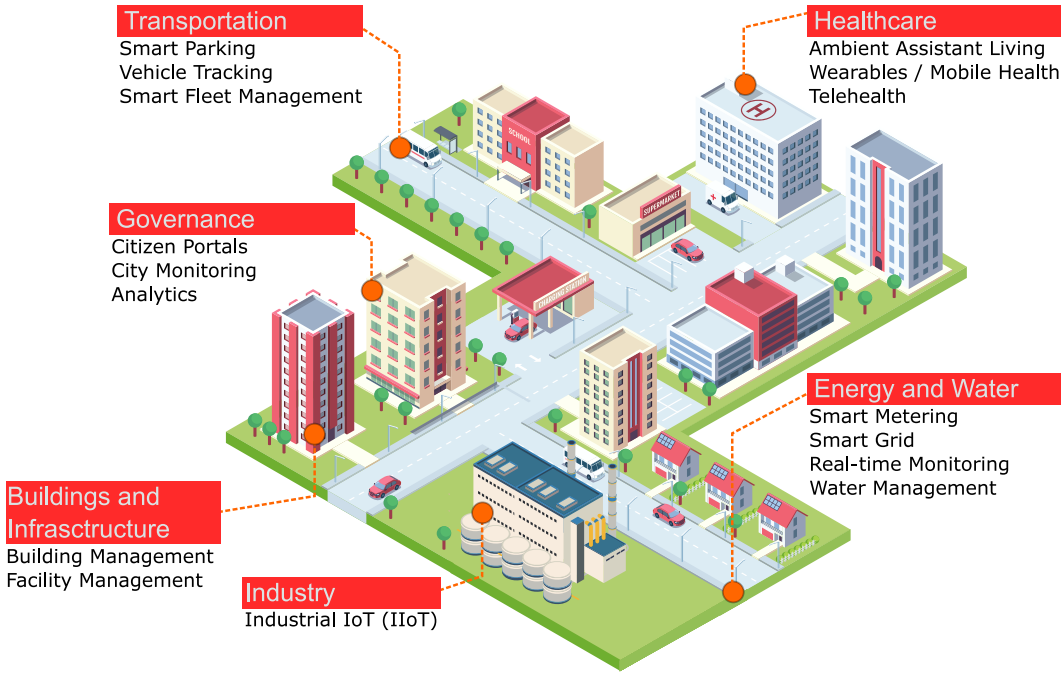


Figure 1.6: Conceptual view of a *smart city*, showcasing some possible *smart*, interconnected and inter-dependable parts and services.

Taking the example of a *smart city*, as conceptually displayed on Figure 1.6 (p. 13), we have to consider the wide-range of *smart* components involved in such scenario, from *smart homes* and other *smart buildings*, the *smart transportation* and *smart retail*, down to the supporting infrastructures like *smart grid* and *smart energy* that power everything, from buildings to streetlights.

In these examples, the reader can easily observe a key propriety of an IoT system: *large-scale*. From the number of devices that a *smart home* can have with different purposes, to all the services needed to collect, analyze and act upon the data, until the consumer appliances that handle them the control of his home.

The technological fragmentation, resulting from different companies producing different components for the *smart home*, each one of them with a different set of supporting services and consumer appliances [Gui16]. The short life-span in terms of supporting these devices by the different companies that produce them, creates, beyond the technological fragmentation, version fragmentation. These elements, combined with the lack of a de-facto standard for the IoT ecosystem, led to a new key propriety of IoT systems: *heterogeneity*.

Although heterogeneity and large-scale are the most widespread key challenges of IoT-based systems, it must be considered that, depending on the use-case, aspects such as real-time, fault-tolerance, and human-in-the-loop have a crucial impact on the systems. These aspects have a significant impact on developers approaching IoT systems as a base to develop their services and applications. Even though the advances in distributed computing, dealing with *large-scale* systems is a problem in itself, that, on top of the *heterogeneity* typical on IoT, also leads to a complexity that it is not yet fully covered by any known system.

1.8 Emerging Challenges

IoT systems share characteristics that make them mostly-unique from a complexity perspective — making their development challenging and resource-consuming — as can be concluded by analyzing the presented scenarios of the previous section. Several authors have been enumerating what they consider to be the main characteristics, and, although there is a lack of consensus, most of them identify the following: (1) large-scale, as the result of the unprecedented — and increasing — number of devices and services that made up the systems, distributed both physically and logically; (2) heterogeneity and lack of interoperability, mostly resulting from the lack of standards, technology fragmentation and vendor-lock attempts; (3) highly dynamic and unknown nature of the network topology; (4) need to be mostly end-user-centric; and (5) real-world intertwinement, since these system's *sense* and *act* upon their physical surroundings [Gui16; Smi17; HTI11; UK18a; And+21].

While these characteristics, when combined, result in a complex system, their by-product points to an even *complexer* ecosystem. Ensuring these systems security and privacy becomes challenging as a result of having dozens of *smart* devices spread among houses and cities while collecting and processing enormous amounts of data creates an unprecedented attack surface and raises several concerns regarding the confidentiality and integrity of the data being collected and stored. From a dependability perspective, ensuring the delivery of correct service when there are too many variables to take into account, *e.g.*, battery life, hardware degradation, connectivity issues, misconfigurations and de-calibrations. Similarly, other aspects such as the volume and velocity of data being collected and the real-time needs of some IoT application scenarios also pose complex challenges that need to be met.

From a technical viewpoint, design, construct, test, deploy, evolve, and maintaining these systems while tackling the observations mentioned above remains as a mostly *unsolved* challenge given that there is no silver-bullet that addresses all the present challenges [Bro86].

As Andrade *et al.* [And+21] point out, the traditional development approaches are mostly unsuitable for the IoT domain. As aforementioned, in the first large automations, all the system's lifecycle was managed by a handful of people with specific technical knowledge that made the system as it would be used by end-users (*i.e.*, bottom-up), and the ability of these users to modify the in-place system logic was very limited or nonexistent. In order words, traditionally, end-users were expected to interact with applications, wherein users with administrator privileges and specific technical knowledge configured them by defining the possible interactions and features.

In modern systems, due to a plethora of constraints (*e.g.*, large-scale) and motivations (*e.g.*, user-specific personalizations), an increasing number of features have been introduced that empower users with limited technical expertise to personalize and, even, program[10] their systems in a way that comply to their specific needs and requirements (*i.e.*, top-down). IoT can be considered one example of such modern systems, given that these systems are required to

[10]While the origins of low-code/no-code go as back as the 1980s with the introduction of Rapid Application Development (RAD) tools as an alternative to text-based development, the lack of technical human resources and the growing need for applications (*e.g.*, digitalization and digital transformation) have been driving its large adoption across application domains (so-called end-user development and citizen development) [BRH20; PJ21; Won+21].

be in constant adaptation — *e.g.*, a user defining new behaviors for its *smart* home — it is not feasible *to go back to the early stages of the development cycle when changes need to be implemented*. Thus, the systems' users must have the ability to configure and change their system when they wish to do so, without requiring specific expertise. The underlying system should also ensure its correct functioning at all times, adapting itself when required.

1.9 Research Goals

The technological and historical context on how IoT is proclaimed to be the next evolution of the Internet, resulting from the interconnection of objects (things), people, systems, and information resources at a large scale has been briefly discussed in the previous sections, complemented by a short discussion on the emerging challenges of this *evolution* of the Internet.

We observe that there exists a wide-spectrum of challenges that need to be tackled and that most of them will require contributions from different research communities, including joint-efforts. In this work, we will delve into the mostly unique characteristics that threaten the dependability of IoT systems and the underlying research challenges, with a special focus on end-user development environments.

To concretely understand the underlying pending research challenges that (in-)directly influence the current state of IoT development we start by addressing the following research question:

RQ1 What are the unique characteristics of IoT systems that make them complex, and how does such complexity impact the end-user ability to configure their dependable systems?

The review of the state-of-the-art, as well as the carried end-user studies, allows us to grasp the origins and motivations behind the current complexity of the IoT systems. The end-user surveys give some insights on what are the most common automations that users' want to perform, and, thus, showcasing how the complexity of IoT systems can limit their ability to configure such automations dependably. This led us to the second research question of this thesis:

RQ2 Are there recurrent problems concerning the lifecycle of IoT systems, and what are the prevalent solutions that address them?

Most of the problems faced in the IoT ecosystem are not novel per se, but it is their combination at large scale that must be tackled. Concerning dependability, IoT has been tormented by the same problems that have been long solved by other fields of research and industries (*e.g.*, mission-critical and real-time systems), but the adoption of the solutions created by such related fields have not been re-thinked and adapted to the IoT scope. By thoughtfully systematizing this existing knowledge as *patterns*, taking into account the particularities of the IoT system we provide a set of design guidelines that can be used to enhance the dependability of the system while avoiding *reinventing the wheel*.

RQ3 What can be improved concerning the IoT systems' dependability?

While the state-of-the-art (RQ1) presents several widely-used techniques and methodologies to improve a system dependability (*e.g.*, fault-tolerance), there are strategies used in other research areas (*e.g.*, cloud computing) that are still in their early stages of adoption in IoT, sharing several pending issues that hinder their adoption. The use of the available computing resources in an automated fashion, *i.e.*, distributing computing tasks among computing, has been leveraged in cloud computing as a way of both distribute system load and avoid single-point-of-failure (SPOF). Proving such mechanisms in IoT can bring similar advantages. By proving the system with the mechanisms to dynamically allocate computational tasks while adapting to the environment constraints within widely-used development environments (*e.g.*, visual programming environments such as Node-RED), we provide mechanisms to build systems with improved dependability.

RQ4 How can the mechanisms identified in RQ2 be leveraged by the end-users of IoT systems?

Even though most of the dependability problems that are common to IoT can be solved, to a certain degree, by leveraging existing knowledge, this does not directly address the problem of having users without specific technical knowledge building their IoT systems dependably. By providing a set of visual abstractions that can be used in already widespread end-user programming solutions, we can both leverage the existing knowledge and provide it as easy-to-use build-blocks to the end-users.

Even so, most of the existing end-user programming tools are mostly opaque and provide little-to-no feedback about the running system to their users, which limits their ability to change (re-program) their system to meet their ever-changing requirements. This led us to our final research question:

RQ5 How can the end-user's ability to manage the IoT systems' lifecycle be improved without requiring specific expertise nor hindering the systems' dependability?

By providing mechanisms that aid the end-user to understand the behavior of their IoT system (*e.g.*, the messages that flow in the system, the configured trigger-action rules or the cause of some state change) the end-user can more swiftly re-adjust their system and *see* the consequence of those changes (*i.e.*, improved visual notations).

Taking into account these research questions, we can now state our research hypothesis as follows:

> **H:** *It is possible to enrich IoT-focused end-user development environments in such a way that the resulting systems have a higher dependability degree, with the lowest impact on the know-how of the (end-)users.*

As a reference implementation of an *IoT-focused end-user development environments* we will be using Node-RED, as it is one of the most widely-adopted development environments in the scope of IoT. By *(end-)user* we consider all IoT system users that have the minimal set of knowledge to configure their own systems, giving a special focus to the ones that are used to visual programming languages (independently of the domain where they use them).

These research goals and their motivation — taking into account the current state of practice and available literature — is further discussed in detail in Chapter 5 (p. 132).

1.10 Research Contributions

In the scope of this thesis, a motivational framing on the software development challenges that have tormented the software industry for long is presented, contextualizing with the challenges posed by this *new* Internet, which can, potentially, lead to a new software crisis [Fit12].

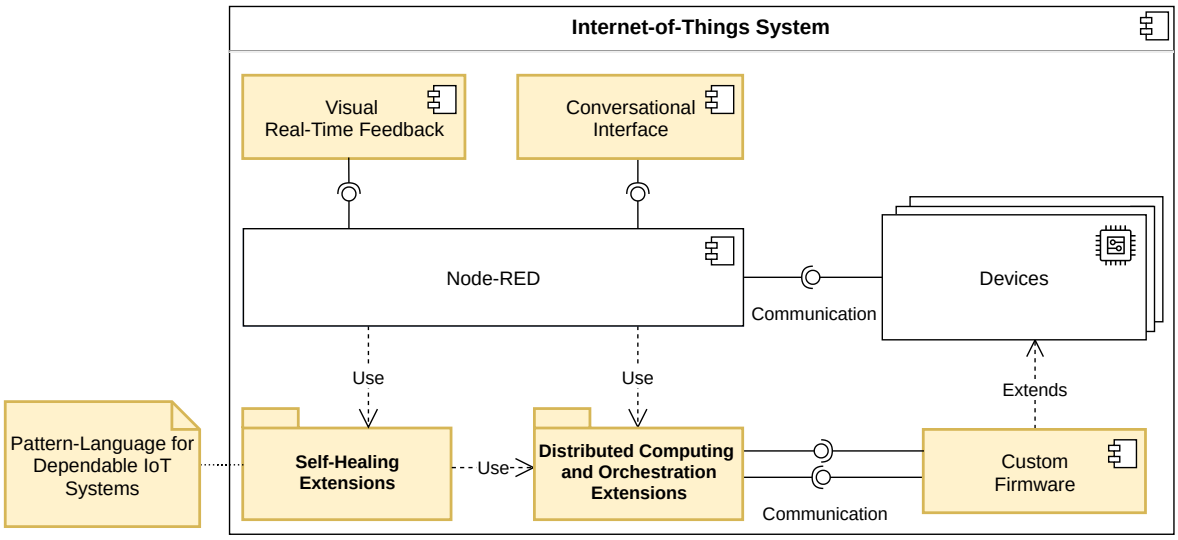


Figure 1.7: High-level overview of the approach detailed in this thesis.

With this work, we do not aim to provide a *silver-bullet* for all the challenges of IoT development, but to provide valuable, and empirically validated, insights on how to address some of them. The main contributions to the IoT field of this research can summarize as:

A comprehensive review of state of the art. A thoughtful analysis in both fundamental concepts, technologies, and development practices that are key for IoT are presented, discussed, and open challenges summarized. More details in Part I (p. 21).

Patterns and pattern-language. Best practices for designing and developing dependable IoT systems are described, inspired in both current practices from the field and related fields. More details in Part II (p. 144).

Dependable and autonomic computing. A compendium of several experiments into the distribution of computing task in constrained devices and the role of autonomic computing — with a focus on self-healing — in assuring dependability in IoT systems. These contributions try to meet the quality-of-service demands of end-users while attempting to empower them to configure their dependable systems. More details in Part III (p. 184).

End-user development. Two studies on the use of visual and voice-based low-code solutions for IoT system development and configuration. These studies focus on enabling the end-user to build/configure systems that are more dependable while improving the ability of the user to understand the running systems. More details in Part IV (p. 246).

The contributions present in this work, as enumerated, are highlighted in Figure 1.7 (p. 17), which presents an overview of the complete approach that satisfies the stated hypothesis. Node-RED [Ope19b] is the foundation block of our work, and we attempt to minimize the changes to this component, building the contributions as extensions or extra-modules. The devices were both *off-the-shelf* and *do-it-yourself* ones tailored to our use cases (*i.e.*, *SmartLab*), with most of them accepting custom firmwares (*i.e.*, allowing to program the device behavior). No relevant contributions were done in regards to communications, sensing and actuating.

These contributions were summarized and published in several venues given the cross-field nature of IoT research. More details on the publications resulting from this work are presented in Appendix A (p. 321).

1.11 How to Read this Document

The target audience of this work includes software engineers, designers, architects, developers, system integrators, power-users, or anyone involved in the conception, design, construction, maintenance, or evolution of IoT systems. This research (1) gives the fundamental background knowledge to the reader on the IoT ecosystem, (2) empowers the reader with a set of best practices that can aid the development of dependable IoT systems, (3) proves insights on how to develop and integrate these systems — in a dependable and autonomic fashion — while reducing the required specific technical knowledge. We also target this work at IoT researchers, especially the ones focused on dependability in IoT, end-user development environments and dependable autonomic systems, extensively discussing both virtual and physical observations on the aforementioned contributions.

Due to the cross-domain contributions of this thesis, the remaining of this document is organized into four parts, which, although related, can be (mostly) read independently. These parts and their contents are depicted in the following paragraphs.

Part I (p. 21) presents a summary of the key background concepts of this work, an analysis of the state-of-the-art, the insights of an end-user study, and, finally, it delves on the concrete research challenges tackled by this work.

- Chapter 2 (p. 21) gives an overview of the key concepts of IoT, the most common architectures, fault-tolerance systems, autonomic computing, and some core concepts of the software development lifecycle.

- Chapter 3 (p. 64) provides an extensive analysis of the state-of-the-art regarding IoT systems design, development, testing. Relevant works in the research fields of fault-tolerance and autonomic computing are also analyzed.

- Chapter 5 (p. 123) presents some insights on what are the end-user intents of use and preferences for IoT systems by summarizing the results of a two-part survey on home automation and low-code approaches.

- Chapter 5 (p. 132) discusses the finding of previous chapters, systematizing the emergent challenges by different viewpoints. It is then presented the goals of this research, elaborating on the main hypothesis, the followed methodology and the different validation steps.

Part II (p. 144) introduces our pattern-language for dependable IoT systems. It presents an overview of the patterns (*cf.* Chapter 6, p. 144), followed by enumerating a total of 34 patterns: 7 supporting patterns (*cf.* Chapter 7, p. 151), 13 error detection patterns (*cf.* Chapter 8, p. 159), and 14 recovery and maintenance of health patterns (*cf.* Chapter 9, p. 171).

Part III (p. 184) presents a three-part work towards more dependable IoT systems by both ensuring the ability to distribute computing tasks during runtime and provide mechanisms by which the system can inspect the runtime conditions and adapt as it changes. A first study (*cf.* Chapter 10, p. 184) delve into the ability to automatically distribute computation tasks among the available resources given both operational constraints and runtime conditions. A follow-up study (*cf.* Chapter 11, p. 199) delves on how to provide such computational distribution in existing visual programming systems while leveraging the computational resources of constrained devices. Finally, a set of contributions to self-healing IoT systems are provided which leverage the pattern-language presented in the previous part.

Part IV (p. 246) details two studies on the enhancement of current mechanisms by which the users' interact, configure, and build their own IoT systems. The detailed enhancements focus on giving users low-code development solutions that provide more feedback about the running system (*cf.* Chapter 13, p. 246) while improving their overall capability of understanding the (already defined) system behaviors (*cf.* Chapter 14, p. 258). The presented contributions leverage both visual notations and voice-based interactions to improve the ability of the end-user to build more capable — and more dependable — IoT systems.

There are often references to programming language's lexical elements, standard libraries, framework functions and variables which are identified using `typewriter` font. References to pattern names (both from this work or relevant literature) are type-faced in Small Caps. Book and article titles are type-faced in smallcaps. References and citations appear inside [square brackets] and are highlighted — when viewing this document on a computer, these will also act as hyperlinks.

Some contributions of this thesis were supported by one or more master's degree dissertations, with some being part of one or more published works. These works are duly acknowledged in the first paragraphs of the corresponding chapter in which the corresponding master's work contributions are presented, and the author diverse contributions to the published work are presented using the *CRediT author statement* [AOK19].

Part I

Fundamentals

2 | Background

This chapter introduces the IoT concept and its key aspects, the notion of dependability and fault-tolerance in computing systems, autonomic computing, and patterns. This chapter can be skipped by experts in these subjects. We revisit the origins of IoT, delving into its main characteristics, reference architectures, and the paradigms of fog and mist (edge) computing. The theory of dependable computing is introduced as well as its central notions such as fault-tolerance. Autonomic computing fundamental view is also introduced, as well as its key elements. Finally, the notion of Design Patterns and Pattern Language is presented as a source of systematized knowledge of observable practices to aid the design process of software systems.

2.1 Internet-of-Things

The term Internet-of-Things (IoT) is claimed to be coined by Kevin Ashton[1] circa 1999, during a presentation about supply-chain management and the use of Radio-Frequency Identification (RFID) technology to enable computers to observe, identify and understand the world (without the limitations of human-provided data), long before anything but computers were actually connected to the Internet (Internet of Computers) [FF11]. Ashton, by the time, already presented concerns about the need of standards and their role in the sucess of IoT, suggesting the need of "*a standardized way for computers to understand the real world*" [Sch02].

From its birth, a crucial requirement for IoT, beyond ubiquitous computing, was ubiquitous connectivity. Any computing-able object that is aware of its context and that can communicate with other entities fall under the umbrella of IoT. Initially, RFID was the dominant technology behind IoT development, but, as of today, WSN and Bluetooth-enabled devices augmented the mainstream adoption of the IoT trend [Kev09].

[1] Citing the author, "I (Kevin Ashton) could be wrong, but I am fairly sure the phrase *Internet-of-Things* started life as the title of a presentation I made at Procter & Gamble (P&G) in 1999" [Kev09].

The joint technical committee of the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) — ISO/IEC JTC 1 — defined[2] IoT as [ISO14]:

> *An infrastructure of interconnected objects, people, systems, and information resources together with intelligent services to allow them to process information about the physical and the virtual world and react.*

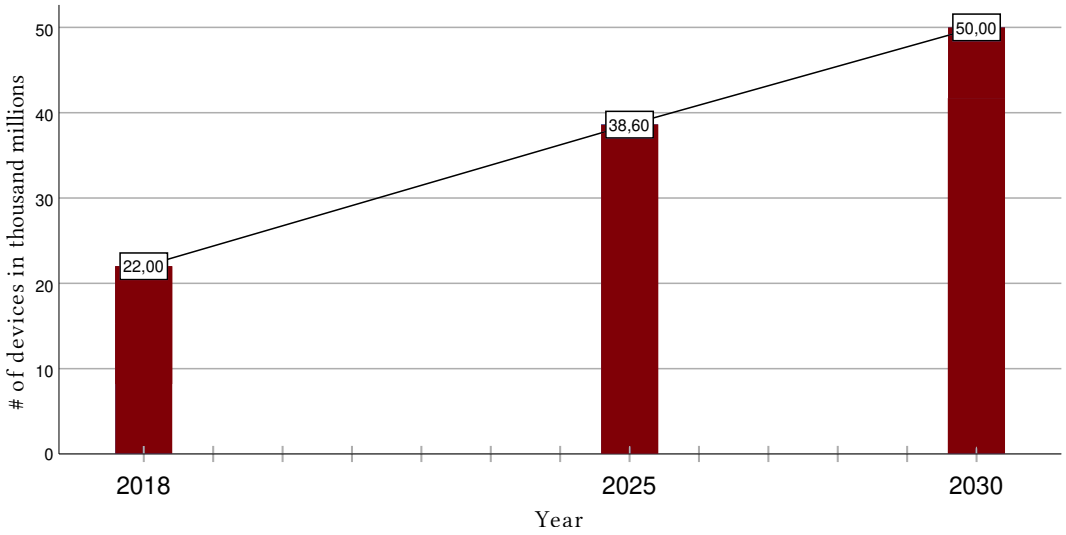


Figure 2.1: Actual count and estimation of the number of IoT devices per year (from Statista) [Tan18].

IoT shifts computing from traditional devices (IoC) towards a ubiquitous and pervasive computing world of systems, and, Systems of Systems (SoS), where heterogeneous and highly-distributed objects (*things*) are computing-capable and Internet-connected [ASD16]. The number of Internet-connected devices under the IoT umbrella is growing (*cf.* Figure 2.1, p. 22), expected to reach around 50 thousand million of connected devices by 2030 [Tan18].

From an innovation point-of-view, IoT has been enhancing the interactions among *things* and humans, enabling the realization of smart cities, infrastructures, and services for enhancing the quality of life and utilization of resources. Thus, IoT envisions a new world of connected devices and humans in which the quality of life is enhanced because of the management of the city and its infrastructure is less cumbersome, health services are conveniently accessible, and disaster recovery is more efficient [BD16]. In regard to the fourth industrial revolution, so-called Industry 4.0, IoT, more specifically known as IIoT, is considered the *supporting backbone* of the vision of *"a fully describable, manageable, context-sensitive and controllable or self-regulating manufacturing systems"*, making it possible by embedding Internet-powered devices in the production systems that are part of the life cycle of a product [LLD18].

From a technical point-of-view, one can consider that a major role of the IoT consists of the delivery of highly complex knowledge-based and action-oriented applications in real-time. To

[2]Note that the precise definition of IoT and its relationship/boundaries with the concept of Cyber-physical Systems (CPS) and less-used terms such as Intelligent Internet-of-Things (IIoT) or Internet-of-Everything (IoE) is yet an open discussion [Bor+17].

be able to reach such an end, several considerations should be done regarding the full lifecycle of these systems, from conceptualization to development, from test to deployment and maintenance. These include, but are not limited to, (1) development of scalable architectures, (2) moving from closed systems to open systems, (3) dealing with privacy and ethical issues (due to data collection and storage practices), (4) heterogeneity support, (5) data storage, (6) data processing, decision-making, (7) designing interaction protocols, (8) autonomous management, (9) communication protocols, (10) smart objects and service discovery, (11) programming frameworks and languages, (12) resource management, (13) data and network management, (14) real-time needs, (15) power and energy management, (16) governance, and (17) interoperability [BD16].

Despite the ongoing discussion about the concrete definition[3] of the IoT, and consequently, what devices comprise its ecosystem, the total count of connected devices is growing fast[4].

2.1.1 Application Domains

IoT has been one of the main drivers of technological innovation in different contexts and scenarios since it works as the foundation for any *smart space*. This observation is supported by the work carried by the Cluster of European Research Projects on the Internet of Things (CERP-IoT) which has identified numerous application domains for IoT [Sun+10].

Table 2.1: Summary of IoT application domains [Sun+10].

Domain	Description	Indicative Examples
Industry	Activities involving financial or commercial transactions between companies, organizations and other entities.	Manufacturing, logistics, service sector, banking, financial governmental authorities, intermediaries, etc.
Environment	Activities regarding the protection, monitoring, and development of all-natural resources.	Agriculture & breeding, recycling, environmental management services, energy management, etc.
Society	Activities and initiatives regarding the development and inclusion of societies, cities, and people.	Governmental services towards citizens and other society structures (e-participation), e-inclusion (*e.g.*, aging, disabled people), etc.

The CERP-IoT report [Sun+10] defines three IoT application domains, as they are described in Table 2.1 (p. 23). Within these application domains, several fields with open opportunities are presented such as aerospace and aviation (systems status monitoring, green operations), automotive (systems status monitoring, vehicle-to-vehicle, and vehicle-to-infrastructure communication), telecommunications, intelligent buildings (automatic energy metering, home automation, wireless control), healthcare (personal area networks, monitoring of parameters, positioning, real-time location systems), independent living (wellness, mobility, monitoring of an

[3] Defining IoT is *fuzzy because of breathless hype, including attempts to anticipate demand for devices have yet to be invented or commercialized* [Nor16].

[4] Amy Nordrum states that the exact total number of devices will be *somewhere between Gartner's estimate of 6.4 thousand million (excluding smartphones, tablets, and computers), and IHS Markit's estimate of 17.6 thousand million (with all such devices included)* by the year of 2020 [Nor16].

elder population), retail, logistics, supply chain management, people and goods transportation, media, entertainment, and insurance.

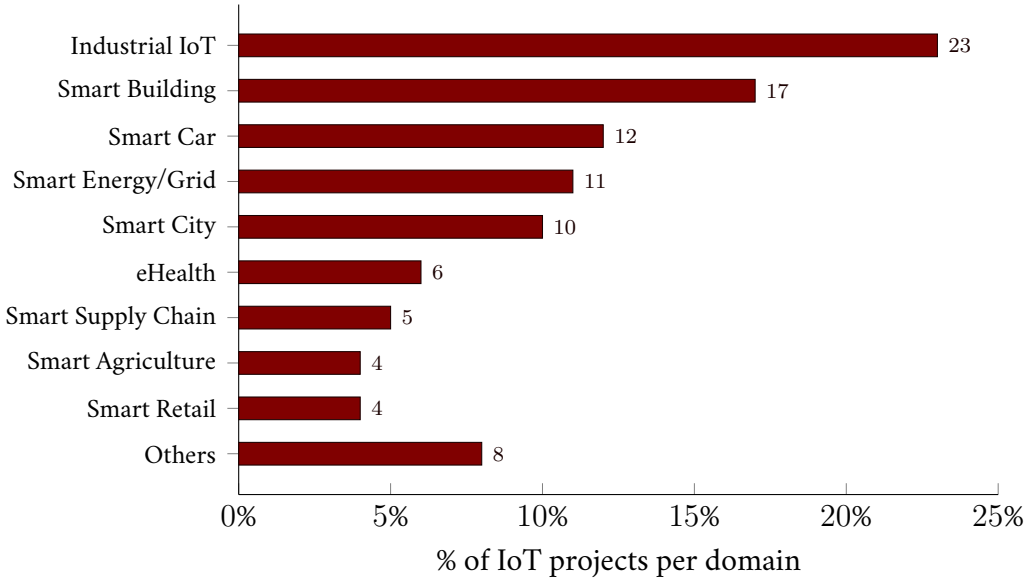


Figure 2.2: Number of IoT enterprise projects per application domain. Statistics based upon 1600 public known enterprise IoT projects circa 2018 (not including consumer-grade IoT projects such as wearables and smart homes) [Scu18].

The IoT Analytics GmbH Report [Scu18] points that the most relevant enterprise-level IoT segments are Smart City, IIoT, Smart Building, Smart Car, Smart Energy/Grid, eHealth, Smart Supply Chain, Smart Agriculture, Smart Retail, and their relevance is shown in the chart on Figure 2.2 (p. 24). However, this report does not count with consumer-level IoT segment (*e.g.*, wearables and smart homes).

IoT enterprise applications can also be aggregated in three major categories, depending on their role, namely [BD16]: (1) monitoring and actuating, (2) business process and data analysis, and (3) information gathering and collaborative consumption.

The IIoT, core technological component of the Industry 4.0 initiative, consists on the adoption of Internet-enabled *things* with sensing and actuating capabilities as a way to gather data about production processes, thus enabling companies to detect and resolve problems faster — resulting in overall money and time savings [BD16]. As an example, in a manufacturing company, IIoT can be used to efficiently track and manage the supply chain, perform quality control and assurance.

Altogether, with the recognized impact that IoT can have on the industry, it is also envisioned the impact that IoT can have on improving the quality of life [BD16]. From a healthcare perspective, IoT can be a facilitator of data collecting (*e.g.*, heart rate) which enables remote patient monitoring, viz. ambient assisted living (AAL) [Doh+10]. Further, monitoring hazard environmental conditions can give data insights for authorities to act accordingly and alert the population.

Holistically, exploiting the open IoT opportunities and different application domains can lead to, on the one hand, improve people's quality of life, and, on the other hand, improve the industry and the enterprise world.

2.1.2 Sociotechnological Context

The initial vision of IoT — enabled by achieving ubiquitous computation[5] and ubiquitous connectivity — has now become a reality. Our mobile phones, our security cameras, our watches, our fridge, toaster, and coffee machine, all have (micro-)processors more powerful than the one used to send the first man to the moon [LY02]. They all tend to be (inter-)connected for sending alerts, transferring new updates, or even downloading new *mocha* recipes from the Internet [Kru16]. We continuously push annotated geolocation (collected almost simultaneously by the GPSs present in our mobile phone, watch, fitness band, and car) to a multitude of recipients that consume vast volumes of personal data for the sole purpose of displaying ads [Aks+18]. Even wired Internet access is becoming obsolete due to faster, cheaper, convenient, and more energy-efficient wireless technologies (4G and 5G) [And+14]. The *status quo* is ubiquitous connectivity, available to everyone, everything, every time, everywhere.

From the above examples, one can understand why IoT has been promoted and widely used as an enabler of *machine-to-machine*, *human-to-machine*, and *human-with-environment* interactions [BD16]. IoT has a key role in the scope of *human-in-the-loop* systems, in which humans and *things* operate synergistically and cooperatively. As new applications intimately involve humans, a range of new opportunities for a broad range of applications, including energy management and automotive systems, will appear.

Munir *et al.* categorize these *human-in-the-loop* applications and systems based on the controls that they employ: (1) applications where humans directly control the system (*e.g.*, switching one room lights on/off), (2) applications where the system passively monitors humans and takes appropriate actions (*e.g.*, surveillance systems or sleep monitors), and (3) hybrid approaches based on the aforementioned (*e.g.*, energy optimization systems that both use information from sensors and commands from operators) [Mun+13].

However, as pointed out by John Stankovic, several challenges also arise from the *human-in-the-loop*: (1) the need for a comprehensive understanding of the complete spectrum of types of *human-in-the-loop* controls, (2) the need to extend current identification systems, or other techniques to derive models of human behaviors and determine how to incorporate human behavior models into the formal methodology of feedback control [Sta14].

2.1.3 Physical Platforms and Devices

The hardware (*e.g.*, devices, computers) in which IoT systems (or components) *live*, ranges from low computing power edge devices (edge tier) to high-performance cloud servers (cloud tier).

Most of the cloud part of IoT systems use software as a service (SaaS), platform as a service (PaaS), and infrastructure as a service (IaaS) provided by several vendors such as Microsoft Azure, Amazon Web Services (Amazon AWS) and Google Cloud Platform (GCP) [IKK13; KM16; PK16; Ray16]. These and other providers have been developing new and tailored solutions (*i.e.*, computational platforms) to meet the requirements of IoT in terms of cloud resources.

Different surveys on IoT cloud providers identify and compare the different available solutions, including the AWS IoT Platform, Kaa IoT Platform, IBM Watson IoT Platform, Microsoft

[5] Ubiquitous computing is also known as *ubicomp* or pervasive computing.

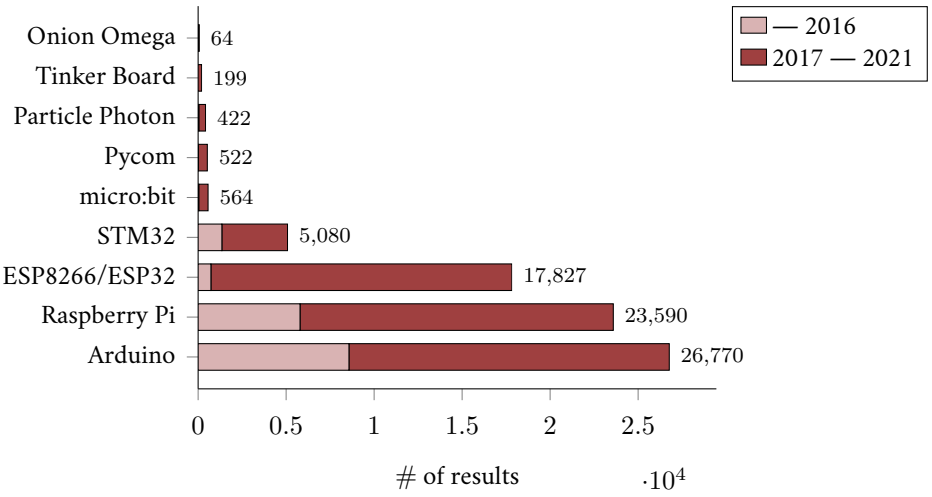


Figure 2.3: Number of results on Google Scholar for most-known hardware boards. Query `("<board_name>") AND ("IoT" OR "Internet-of-Things" OR "Internet of Things")` circa November 2021.

Azure IoT Platform, Bosch IoT Suite, Google Cloud IoT, ThingSpeak and ThingWorx. P.P. Ray survey of 26 different IoT clouds exhaustively compares the available solutions in terms of suitability and applicability in terms of support for application development, device management, system management, heterogeneity management, data management, analytics, deployment management, monitoring, and visualization [Ray16]. In his work, he points out several issues across different solutions, namely, standards conformance, heterogeneity adaptation, context awareness, incentives to the proliferation of vertical silos, issues dealing with things identities, and lack of considerations for fault-tolerance and energy management.

While different cloud providers favor different protocols (*cf.* Section 2.1.4, p. 30, and Section 25, p. 34), operating systems (*e.g.*, Amazon FreeRTOS and Microsoft Windows 10 IoT), and provide different Software Development Kits (SDK), they provide a common set of services thus reducing the heterogeneity between them (although having different commercial names).

Regarding the lower tiers of the IoT system — devices closer to the physical environment that they sense and act upon —, namely the *fog* and *edge* tiers, the device and platform heterogeneity sharply increases. Most of these devices can be placed into three groups which are presented next, along with some examples of hardware devices that were searched on Google Scholar for a reference metric of popularity (*i.e.*, number of results), plotted in Figure 2.3 (p. 26). The most common device taxonomy is (1) Single-Board Computers, (2) Single-Board Microcontrollers, and (3) Networking Devices. Most of these devices have several aspects that should be considered when choosing the most appropriate for a given scenario, such as (1) computing capabilities, (2) power consumption, and (3) connection range (*cf.* Figure 2.4, p. 30) [SK17].

Single-Board Computers

Single-Board Computers (SBC) are complete computers built on a single circuit board, with microprocessor (typically ARM-based), memory, I/O and other features required to have a fully-functional computer, capable of running a complex operating system such as different

Linux distributions [UHM11]. These devices are typically part of the fog tier (*cf.* Figure 2.7, p. 39), but can also be used in the edge tier. Some examples are:

Raspberry Pi One of the most popular SBCs, firstly designed for educational purposes, has been used for the most diverse purposes, including IoT system. It uses a Broadcom system on a chip (SoC) and has an integrated ARM-compatible CPU [Fou19b; KGC16; SA16; PKB15].

Onion Omega Board This SBC has an even smaller size when compared to Raspberry Pi, uses the MIPS CPU architecture, and has a power-efficiency typical of embedded devices [Cor19; Raj+18].

Asus Tinker Board An ASUS-designed competitor for the Raspberry Pi, has a Rockchip SoC and an integrated ARM-compatible CPU [Glo19; ZEB18].

The presented list is limited in number it showcases the most common fog tier devices. Other solutions exist, but, typically, they share similar hardware capabilities and features.

Single-board Microcontrollers

Single-board Microcontrollers are built onto a single printed circuit board providing the minimum requirements for computing and control tasks, namely: microprocessor, I/O circuits, a clock generator, RAM, stored program memory, and any necessary supporting integrated circuits (ICs). Some of these boards are capable of running operating systems[6] such as FreeRTOS and RIOT [GT15]. These boards, jointly with some communication microchips, are the most common devices in the lower tier of IoT systems (*cf.* Figure 2.7, p. 39). The following presented microcontrollers are some well-known microcontrollers used:

Arduino and Arduino-like boards Boards based on Atmel AVR microcontrollers that can have some way to communicate such as serial ports, Wi-Fi microchips, or even SigFox microchips, and a set of digital and analog I/O pins that can be used to connect to various expansion boards (shields), other circuits and sensors and actuators. The boards are typically programmed using the Arduino language and framework, but they are compatible with other firmwares, and, even, support some real-time operating systems [Ard19b; Dou12; Ben18].

Particle It is a set of boards and development kits with built-in Wi-Fi, Bluetooth, and/or cellular connectivity, that can work in a mesh fashion (board versions: Argon, Boron, Xenon, Photon, Electron). It has an ARM-based processor, runs a customized OS called Device OS and can be programmed both by a direct serial over the USB connection or using the Particle's Cloud IDE [Par19].

[6] The most common operating systems for these boards and embedded devices are Real Time Operating System, which are intended to serve real-time applications that are required to processing data as it comes in with minimal delays, running within a deterministic execution pattern.

ESP8266/ESP32 Microcontrollers from Espressif Systems with built-in Wi-Fi and/or Bluetooth connectivity that has a set of General Purpose input/output pins (GPIOs). The devices can be programmed using Lua, MicroPython, and C (Arduino) [Esp20; Esp19; Ben18; Bel17].

BBC micro:bit An ARM-based board designed by the BBC for educational purposes with Bluetooth LE connectivity that can be programmed in C, JavaScript or a visual programming language using MakeCode Editor that is similar to Blockly from Google [Fou19a; CP16].

STM32 boards STM-based set of ARM SoC based boards by STMicroelectronics that are part of several IoT devices and can be personalized in terms of connectivity, computing capabilities, and GPIOs [STM19; Lin+13; LZZ14].

Nordic boards Set of boards designed by Nordic Semiconductor that use an ARM SoC having a special focus on ultra-low power wireless communication capabilities, supporting protocols such as Bluetooth low energy, Bluetooth Mesh, ANT+, 2.4GHz, IEEE 802.15. 4, Thread, Zigbee and LTE-M/NB-IoT. The BBC micro:bit is an example of such boards, being based on the Nordic's nRF52833 SoC [Sem20].

This is not an extensive list, and there are new microcontrollers being designed and created nonstop to meet new operational requirements or improve certain features (*e.g.*, power consumption and computational power).

Networking Devices

Networking Devices appeared because most of the IoT solutions depend on routers or other kinds of protocol-specific gateways to provide the interconnection of different devices and systems components, by forwarding data packets between different networks. These are typically part of the fog tier (*cf.* Figure 2.7, p. 39). As an example we can consider home automation solutions that use the ZigBee protocol (*e.g.*, Philips Hue [Pen20]), a specific gateway needs to connect and control the edge devices.

Due to the increase in the number of features and computing power of these devices, some of them also act as IoT *hubs*, connecting to multiple devices and allowing its management in centralized fashion [Cir+15; Fan+14]. As an example, some routers are capable of running integration platforms such as Domoticz [TVM19]. TP-Link Kasa Smart Home Router is an example of a consumer-level router that also acts as an IoT hub, integrating devices with ZigBee, Z-Wave, and Wi-Fi connectivity [TP-19].

Summary

The plethora of devices and other physical platforms, (re–)designed and adapted to different application domains (*cf.* Table 2.1, p. 23, and Figure 2.2, p. 24), coupled with the fact that they are being produced and commercialized by different vendors, have resulted in a highly-fragmented market [Gui16].

Regarding the cloud tier (*i.e.*, servers and other platforms), IoT solutions are being provided as Everything-as-a-Service (XaaS) [Dua+15][7], where out-of-the-box, service-oriented architectures are offered as a panacea for any application domain. However, the dependability of this approach can be severely compromised given that centralizing all the systems' core logic in the cloud makes it a single point of failure and dependent on an always-on Internet connection [Kle+19]. Additionally, several authors raised concerns regarding the security and privacy of such cloud-centric approaches [Kle+19; SMM19].

Another research focus is the concept of Software-defined Everything (SDE), which has been expanding the importance and impact of software in a previous hardware-only world[8], becoming a vital part of the IoT ecosystem [BC15]. The dissemination of Software-defined networking (SDN), Software-defined storage (SDS) and software-defined data centres (SDDC) as core components of Cloud-tier are a reliable indicator of the Software-defined tendency [BC15]. However, this also impacts the lower-tiers of IoT systems reducing the need to physically upgrade and change hardware to meet new requirements or correct existing flaws.

It is in the lower-tiers of IoT systems (*i.e.*, fog and edge tiers) where the peak of heterogeneity is observed, resulting in a vast mix of Single-board Computers (SBC), networking devices, and embedded/constrained devices. These devices, tailored to the different application domains and working environments, are built with a variety of central processing units (CPUs), communication radios, and power consumption needs. They also provide different end-user interactions, and include different sensing and actuating capabilities.

Current research efforts are focused on the development of ever-smaller boards with lower power needs and built-in power-efficiency strategies[9] (*e.g.*, on/off cycles), low-cost and highly-reliable data transceivers, improved sensing capabilities, and overall more efficient processing units [Aky+02]. Few, if any of these, would be of help towards the de-fragmentation of the IoT.

2.1.4 Communication Protocols

Networking is a mandatory requirement of IoT devices. Every sensor and actuator device should be connected anytime, anywhere. However, these devices are limited by design concerning processing power, energy consumption and management, and available connectivity [Cen+16]. The Wireless Sensor Networks (WSN) research field[10] has long been developing networking solutions that attempt to address these highly stringent constraints. This constant connectivity is one of the critical building blocks of IoT that suffers the most from the fragmentation mentioned earlier. Similarly to what happens in terms of physical platforms and devices, different vendors are trying to develop their own communication radios, protocols, and ecosystems leading to an ever-growing heterogeneous network ecosystem, thus increasing its accidental complexity. Ray *et al.* point out that such a scenario *enhances the complexity among various types of devices through different communication technologies showing the rude behavior of the network to be fraudulent, delayed, and non-standardized* [Ray18].

[7] Also known as Anything-as-a-Service (*aaS) depending on the authors [Maa+19].

[8] As stated by Marc Andreessen circa 2011, "software is eating the world" [And11].

[9] Including the usage of energy-harvesting mechanisms [GG17].

[10] It is hard to clearly define a boundary between the WSN and IoT research fields due to their overlap in several research directions.

Lower-Layers Communication Protocols

Although wire-based solutions are still widely used due to their reliability, with one of the most common ones being Ethernet (*e.g.*, gateways and other fog devices are, typically, connected via Ethernet to routers or other local network devices), these solutions are limited to physical installations and limit mobility. Other wired-protocols[11] that still play a role in IoT include the Controller Area Network (CAN) bus protocol which is widely used in the automotive industry, Modbus which become a *de facto* standard for industrial PLC communications, and power-line communication (also known as power-line carrier or PLC) widely used for transmitting data using AC power lines [AS17].

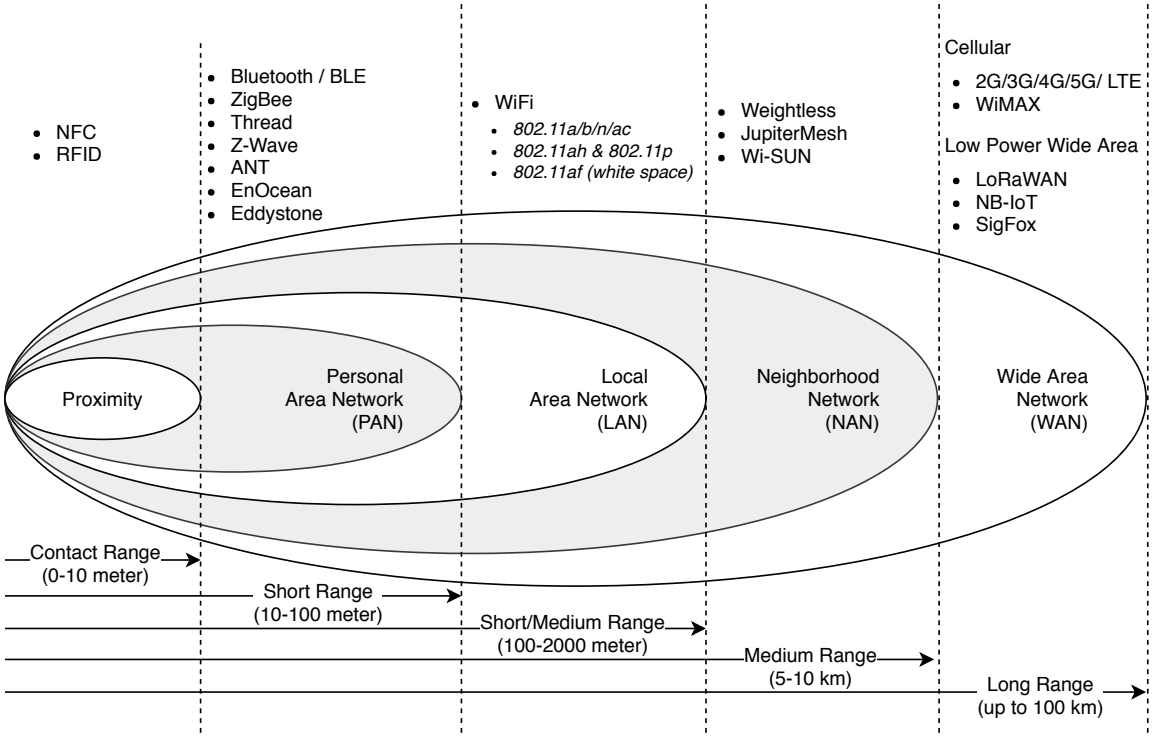


Figure 2.4: An overview of the IoT-enabling network protocols pointing their fitness to each spatial scope (*i.e.*, coverage range), namely: Personal Area Network (PAN), Local Area Network (LAN), Neighborhood Area Network (NAN) and Wide Area Network (WAN).

Protocols targeting lower-layers (per the OSI model[12]) have been specifically designed to address necessities shared by IoT systems. An overview of some of these protocols is given on Figure 2.4 (p. 30). Some of these are transparent to mechanisms used in higher-layers while others pose practical limitations or enforce specific strategies. A few of these protocols are the following:

[11] While there are several other lower-level wired communication protocols used for establishing communication between sensors, actuators, and microcontrollers — *e.g.*, I2C, JTAG, SPI, RS232, and several other serial-based communication protocols — these are not of relevance for this work as their complexity impact is, typically, limited to the embedded systems' development.

[12] Open Systems Interconnection model (OSI model) splits network communication protocols into seven layers — from the physical lower-level layer to the higher-level application layer.

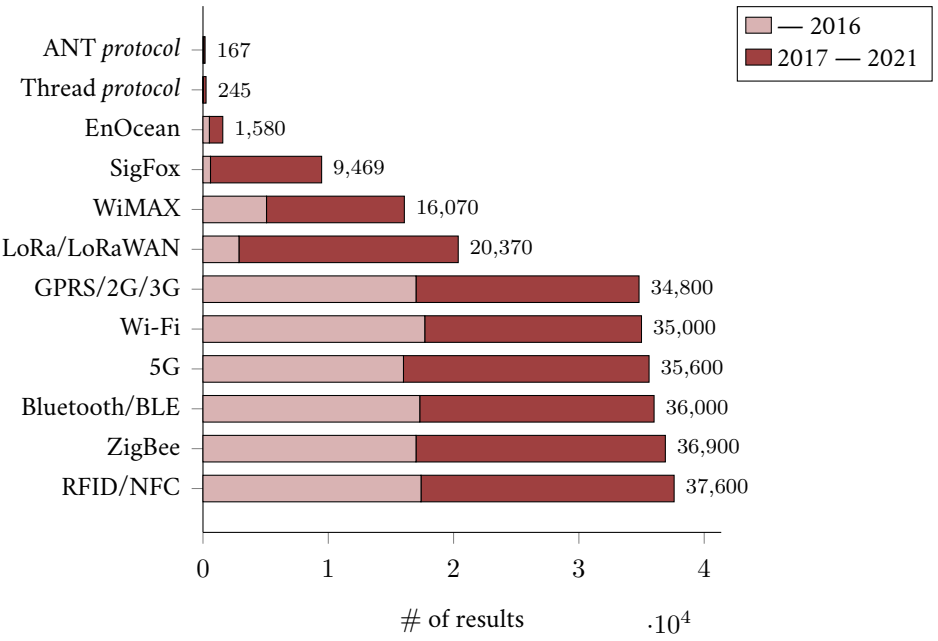


Figure 2.5: Number of occurrence of each protocol in the literature gathered using Google Scholar and the search query `<protocol_name> AND ("IoT" OR "Internet-of-Things" OR "Internet of Things")`, where `<protocol_name>` is the name of the protocol or a combination of two versions of it (*e.g.*, `Bluetooth OR BLE`). Some protocols required the use of the word *protocol* in the query to disambiguate the term. Data collected in November of 2021.

RFID *Radiofrequency identification* is the foundation technology for several solutions that provides automated wireless identification and tracking capabilities. It is more robust than the barcode system and consists at least of a reader (interrogator) with a reader antenna and tags (transponders) which are microchips combined with an antenna in a compact package. A popularized RFID-based solutions is the high-frequecy Near Field Communication (NFC) which operates at a close range of about 10 cm or less [Vaz+12].

Bluetooth/BLE *Bluetooth*[13] objective was to replace short-range wired communication, typically used in point-to-point or star topologies. However, it has a weak security layer. More recently, *Bluetooth low energy* (BLE) appeared, providing considerably reduced power consumption and cost while maintaining a similar communication range [Jeo+18].

Eddystone, iBeacon, AltBeacon, and GeoBeacon A set of protocols that uses BLE to communicate with local beacons[14] for exchanging URL, unique identifiers and other metadata [Ray17]. These can be used to allowing to determine a device physical location, track customers, or trigger a location-based action on the device such as a check-in on social media or push notification.

[13] Bluetooth was introduced in 1994 by telecom vendor Erickson as a wireless communication standard for interaction between computers and mobile phones [BS01].

[14] Bluetooth's beacons are hardware transmitters, which is a class of BLE devices that broadcast their identifier to nearby portable electronic devices.

LoRa and LoRaWAN *LoRa Low Power Wide Area* proprietary technology that supports the use of low power, low data requirements, and long-range operation devices. LoRaWAN defines the communication protocol and system architecture for the network while the LoRa physical layer enables the long-range communication link [Gou+17]. A single LoRaWAN gateway can collect data from thousands of nodes deployed kilometers away [Ade+17].

Weightless The Weightless is an open technology (controlled by the Weightless Special Interest Group) pointed to low-cost M2M applications that require a wide coverage with minimal power usage [MZU16].

SigFox SigFox aims to deploy a controlled network dedicated to IoT, much like a cellular network. A Sigfox certified transmitter must be added to the devices, then the data transmitted by the device is first routed to Sigfox server to scrutinize for integrity and security of the data, following which, it is routed back to applications IT network [MZU16].

ZigBee A common wireless networking standard for building sensor networks. It consists of a packet-based open and global protocol that has been designed to provide a secure and reliable communication architecture transmitting small data at low power over short distances [MZU16].

JupiterMesh A proprietary low-power industrial-grade wireless mesh network with flexible data rates. It includes authentication, encryption and key management, and advanced radio layer technologies such as frequency hopping, and multi-band operation [Zig18].

Thread Based upon the IPv6 and 6LoWPAN protocols, *Thread* is a low-power mesh network protocol. It provides suppliers with the freedom to use proper application layers according to their application needs [Lan+19].

EnOcean A wireless energy harvesting technology (self-powered) that delivers high data rate at low energy consumption, adequate for short-range, less data-heavy applications. Thus, *EnOcean*-compliant devices employ energy converters that reap power from surroundings by harnessing temperature differences, light energy, and mechanical motion, requiring no batteries [MZU16].

WiMAX A wireless technology intended for high-speed data communication applications. It provides a low-cost alternative for cable and data subscriber links while maintaining a data rate comparable to cable rates and the ability to maintain a dedicated link for each subscriber [MZU16].

Z-Wave A proprietary protocol promoted by the *Z-Wave Alliance* that is specifically designed to transmit short messages from the control unit to slave devices at minimal power consumption, useful for automation in commercial and residential environments [MZU16].

ANT Proprietary wireless sensor network technology that enables semiconductor radios operating in the Industrial, Scientific and Medical allocation of the RF spectrum (*ISM*

band) to communicate by establishing standard rules for co-existence, data representation, signaling, authentication, and error detection.

Wi-SUN *Wireless Smart Utility Network* has low power and middle data rate physical layer specification and support of a bi-directional multi-hop transmission. A typical Wi-SUN setup consists of two types of wireless stations, *i.e.*, the devices (collect and transmit data) and the coordinators (control the devices) [Moc+17].

GPRS/2G/3G/4G/5G cellular Using the already-in-place ubiquitous cellular networks for machine-to-machine communications in the IoT domain has been explored in the last few years, with several technologies being introduced for cellular IoT networks, such as Extended Coverage GSM IoT (EC-GSM-IoT), Long Term Evolution - Machine (LTE-M), Narrowband Internet of Things (NB-IoT) [MZU16; And+19]. The next fifth-generation (5G), Radio Access technology, is promised to be a key component of the Networked Society. 5G will support massive numbers of connected devices and meet the real-time, high-reliability communication needs of mission-critical applications [Gou+17].

NB-IoT A technology, also part of the 5G family, is expected to be the leading IoT over LTE technology in the next years, focusing on further optimizations on top of the LTE-M standard, leading to narrowband LTE-M [Gou+17].

Wi-Fi Is one of the most well-established protocols for wirelessly connecting local area devices, typically used to provide Internet connection to them. It is based on the IEEE 802.11 family of standards and operates in the 2.4GHz and 5GHz bands (Wi-Fi 6E also uses the 6Ghz band) [Gas05]. Some extensions to the protocol have been proposed, including the support for mesh networking [Rey+13]. Another related protocol is the WiMAX one, based on the IEEE 802.16, which focus on providing last mile wireless broadband access [BC13].

Despite this extensive list of protocols enabling communication between different parts of the IoT systems across different tiers (*viz.* Figure 2.7), new protocols are still being designed and implemented due to limitations imposed by the scope of operation of these systems. As an example, most of the presented communication protocols are not suitable for Vehicle-to-Vehicle Communication (V2V), and several solutions such as vehicle ad hoc networks (VANETs), and mobile ad-hoc networks (MANETs) have been studied as a solution for such issues [Too+08].

Several works have been focused on using ultrawideband (UWB) as a low energy level for short-range and high-bandwidth communications, with some commercial-grade devices (*e.g.*, mobile phones) already using it for precision locating and tracking purposes [JOJ21]. Some sub-GHz ISM bands (license-free Industrial, Scientific, and Medical frequency bands) are also popular amongst IoT, specifically 433MHz, 868MHz, and 915MHz — with more than 5270 works on Google Scholar by November 2021. The data transmitted in these frequencies can be modulated using a wide range of protocols (*e.g.*, LoRa runs on 433MHz in the USA and 868MHz in Europe) and they are typically low-powered.

Additionally, other *wireless* protocols, including light-based ones, have been proposed as a way of mitigating some of the issues (*e.g.*, electromagnetic interference) that affect most of the

protocols above at some degree. Among such initiatives, one can identify Li-Fi (Light-Fidelity), a derivative of optical wireless communications (OWC) technology, that uses light-emitting diodes (LED) for transmitting data [TVH14]. Another widely-used light-based communication technology is infrared (IR), as standardized by the Infrared Data Association (IrDA) in the early 1990s, which has been historically used as a short-range communication protocol solution and still plays a role in IoT due to its low power consumption [Ran+17; SAA16].

We are still far from a reference (standard) set of protocols a designer can objectively choose from when developing IoT systems. As can be observed in Figure 2.5 (p. 31), several network protocols appear recurrently in the literature. While not every protocol can be used for all kinds of networks, there are some overlapping protocols in terms of range and functioning (*e.g.*, BLE and ZigBee). They are indistinguishable in terms of popularity, which goes accordingly with the current fragmentation (*i.e.*, heterogeneity) of the IoT ecosystem.

Higher-Layers Communication Protocols

Some of the most common higher-layers communication protocols (per the OSI model) that are used in these systems are [Diz+19]:

HTTP *Hyper Text Transfer Protocol* is one of the most common protocols used in the application layer. However, its inherent complexity and verbosity make it an undesirable candidate for IoT applications [MZU16]. HTTP/2 further reduces the network latency, compresses the header-field data, and allows parallel data exchanges on the same network. While it is still complex and verbose, it provides advantages regarding bandwidth, reliability, and messaging mechanisms. Representational State Transfer (REST) architectural style is often used in both [Kar+15].

The third-version of HTTP (HTTP/3 or HTTP-over-QUIC[15]) improves the current version 2 by supporting multiplexing (reducing the impact of packet loss on data transfer speeds) and lessen the handshake scheme [Bis19], however there is still no references to the protocols in the context of IoT.

MQTT *Message Queue Telemetry Transport* was released by IBM, being primarily designed for lightweight M2M communications. It is an asynchronous publish-subscribe protocol that runs on top of the TCP stack and a typical network configuration consists of a server and several clients [SM17]. Small modifications on this protocol originated the *Message Queuing Telemetry Transport for Sensor Networks* (MQTT-SN) [ST13], MQTT over WebSockets that leverage the use of MQTT on top of WebSockets [Del+17], and MQTTw/QUIC which uses QUIC as the transport protocol [KD18].

AMQP *Advanced Message Queuing Protocol* provides asynchronous publish-subscribe communication with messaging. Its main advantage is its store-and-forward feature that ensures reliability even after network disruptions, which is a common issue of IoT networks. It also provides flexible routing and transactions support [Nai17a].

[15] QUIC is a *multiplexed and secure general-purpose transport protocol* on top of UDP [IT19].

CoAP *Constrained Application Protocol* was designed for IoT devices being deployed in low-power lossy networks, working in a REST-style but using datagrams for data transmission [MZU16]. CoAP holds various functional characteristics such as multicast support, small data overhead, Uniform Resource Indicator (URI) support, content-type support, subscription of a resource, and surfing push notifications [Ray17]. Although it was originally designed to use UDP as underlying transport protocol, it can be used over TCP, TLS, and WebSockets.

XMPP *Extensible Messaging and Presence Protocol* is an open protocol that provides facilities such as lightweight middleware, instant messaging, real-time communication, voice/video call, and multi-party chat, and content syndication of XML data [MZU16]. XMPP runs over TCP, and it is suitable for IoT applications due to its publish-subscribe (asynchronous) and also request/response (synchronous) messaging systems, thus being extensible and flexible [Kar+15].

DDS *Data-Distribution Service* is a solution for real-time systems providing a middleware to help real-time publish-subscribe communications. It differs from centralized publish-subscribe architectures by using a set of decentralized nodes of clients across a system that can identify themselves as subscribers or publishers using a localization server [CK16].

STOMP *Streaming Text Orientated Messaging Protocol* is a simple and easily interoperable text-based protocol. Due to its simplicity, there is almost no enforcement over a message semantic, requiring an extra effort in terms of configuration [Mag15].

OMA LwM2M *Light-weight Machine-to-Machine* is a standard by the Open Mobile Alliance that provides an interface between M2M devices and M2M Servers to build an application-agnostic scheme for management and communication between a variety of devices. It is built on top of the CoAP protocol [Al-+15].

Matter Formerly Project CHIP (Connected Home over IP) is a protocol under development by the Zigbee Alliance (also known as Connectivity Standards Alliance) attempting to unify the technological diversity in smart home connectivity. It runs on existing technologies (Ethernet, Wi-Fi, and Thread) by levering the IP protocol stack [All20].

UPnP Universal Plug and Play (UPnP) build on top of the Plug and Play peripheral model and uses existent protocols (*e.g.*, UDP, HTTP and XML), to enable devices to dynamically join a network (automatically setting all the necessary information, including IP addresses). UPnP also defines mechanisms for devices to announce themselves and their capabilities to other UPnP-enabled devices, and automatically retrieving the relevant information about other compliant devices in the network, without requiring any manual configuration (*i.e.*, zero configuration or *zeroconf*). Device Profile for Web Services (DPWS) was introduced as successor of UPnP, but being fully aligned with the vision of Web Services. It defines a set of requirements that enable secure web-base messaging, discovery, description, and eventing, with a special focus on resource-constrained devices [Rei13].

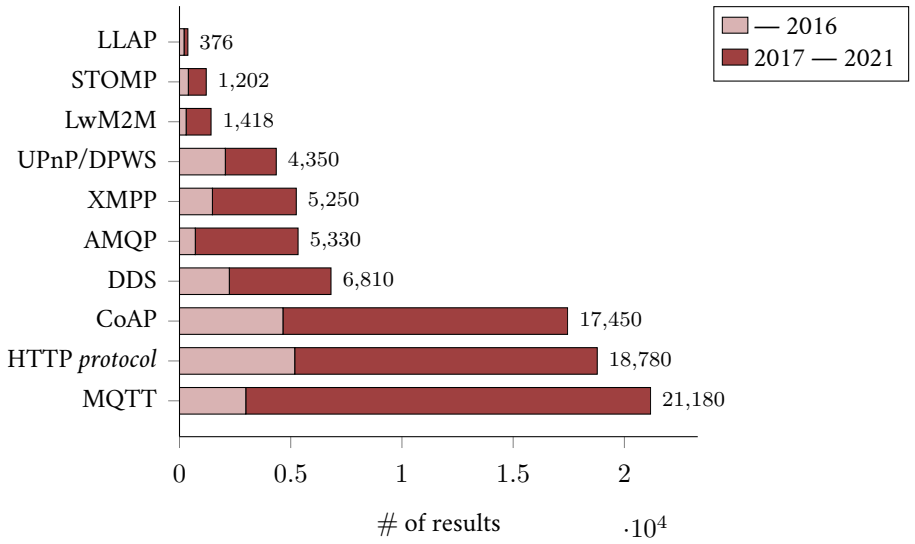


Figure 2.6: Number of occurrence of each protocol in the literature gathered using Google Scholar and the search query `<protocol_name> AND ("IoT" OR "Internet-of-Things" OR "Internet of Things")`, where `<protocol_name>` is the name of the protocol or a combination of two versions of it (*e.g.*, `Bluetooth OR BLE`). Some protocols required the use of the word *protocol* in the query to disambiguate the term. Data collected in November of 2021.

Other protocols include the Data-Distribution Service (DDS) and Streaming Text Orientated Messaging Protocol (STOMP). Several studies have been carried to evaluate metrics on some above-listed protocols, such as power consumption, bandwidth, and latency [Nai17a; Cen+16; CK16].

A popularity study based on the number of results on the literature (from Google Scholar) can be found in Figure 2.6, and while there is a considerable amount of results for most protocols, there is a clear tendency in using MQTT, HTTP, and CoAP. MQTT and CoAP are lightweight and simple-to-use protocols that were designed for IoT, thus their popularity. HTTP is one of the long-establish protocols, and its use is common when IoT devices have the necessary computational resources.

Summary

Research in network protocols keeps growing, mostly focusing on addressing factors such as fault tolerance, scalability, cost, hardware, mutable topology, environmental conditions, and power consumption and efficiency [Aky+02]. For example, Gubbi *et al.* points out that it is common for the network counterpart of a constrained device to stop working, and thus requiring the network to be self-adapting and allow for multi-path routing, being that these both aspects are still in their early stages of maturity [Gub+13].

The SDE trend also provides communication solutions such as Software-defined Radios (SDR) and Software-defined Networks (SDN). This is another evidence that components traditionally implemented in hardware are now being supplanted by others implemented in software, mainly due to their inherent agility (*e.g.*, changing between different communication

frequencies or protocols happens without the need of changing the hardware itself) [BMV17; Qin+14]. This brings advantages such as (1) addressing heterogeneity better, (2) being (more) future-proof, and (3) allowing updates for devices that are situated in inaccessible locations.

From a network services perspective (*cf.* Section 2.2.4, p. 44), there has not been enough investment towards defining a future standard Service Description Language (SDL) [Dar+15]. Such an effort would improve service development, deployment, and resource integration among IoT solutions. Besides this much-needed standard, the development, and usage of powerful service discovery mechanisms and object naming services are also crucial to transforming the *resources of physical objects into value-added services* [Ray18].

Despite the full range of wireless technologies available, each one fulfills different requirements, and each one has a set of advantages and disadvantages (*e.g.*, communication range, power, and bandwidth needs). However, from a system designer's point of view, it is hard to conclude which is the most suitable for a particular scenario or to satisfy a given constraint. This leads to the age-old question [Al-+17; NAG19] of *which technology is the best one for my application?* Recently, some practitioners have tried to partially answer this question by unifying some low-power personal area network protocols — including ZigBee and Thread — under a novel protocol called Matter [All20], thus easing the process of selecting a protocol, in this specific case, for short-range communications (*i.e.*, smart home).

More open challenges in the communications field include the need for new wireless *ad hoc*[16] networking techniques (WANET) or mobile *ad hoc* networks (MANET) [Aky+02; ML15]. Finally, most of the communication protocols currently being used suffer from limitations and flaws regarding privacy and security, pointing out that there is a need for more research on such sensible topics (*cf.* Section 3.4.5, p. 106) [XHL14].

2.1.5 Technology Maturity

A report from Microsoft circa 2019 provides some insights on the challenges faced by practitioners that want to utilize IoT-based products and services, identifying the complexity of the field and other technical challenges (38%), lack of budget and resources (29%), lack of knowledge (29%), lack of suitable IoT solutions (28%) and security considerations (19%) as driving issues [Mic19].

Focusing on the complexity of the field and other technical challenges, the wide-range of application domains and technological advancements that empower IoT makes it depend on several areas of knowledge, thus inheriting their particularities and challenges while also posing new ones. Inherent particularities such as high-heterogeneity, logical and geographical distribution, human-in-the-loop concerns, real-time needs, and power constraints play a key role on the design, development, testing, and maintenance of IoT systems, but are also impacted by other, more common, concerns such as the ones of large-scale systems such as interoperability, security, and scalability [UK18a].

The lack of maturity of several technologies, architectures, and approaches used in IoT (as the lack of the maturity of the field by itself), makes it hard to know how these will evolve,

[16] *Ad hoc* networks do not rely on pre-existing infrastructures (*e.g.*, routers or access points) [ML15].

survive or suffice in the long run. Both the technological challenges (complexity) and unpredictability of the field makes it hard for practitioners to find workers with the right skills and experience to embrace IoT[17] [Mic19].

Given the high data volume being generated by IoT (*e.g.*, sensor data, operational telemetry, and user inputs), plus the variety of objects that make part of it, several issues/requirements need to be addressed to provide a good QoS. Examples of these are minimizing the latency[18], conserving network bandwidth (since it is not practical to transport vast amounts of data from thousands of edge devices to the cloud) and increasing local efficiency (*e.g.*, collecting and securing data across a wide geographic area with different environmental conditions may not be useful) [Han+17; Vas+19].

2.2 Software Architecture Context

Most of the logical architectural concerns of IoT systems are unique to each application domain, such as application features, classes, and relationships, timing constraints, and states. However, the physical architecture tends to exhibit more commonalities across different IoT systems as it is mainly concerned with [Toa18]:

- *On which devices do the system's components live?*
- *How are the computers and hardware devices connected, and how do they communicate?*
- *In which software modules, packages, or components (physical containers) is the system divided? And what are the system dependencies?*

The physical architecture is, in this way, highly-influenced by the advancements of hardware capabilities, communication features, computational resources, energy management, and reliability among others.

2.2.1 Architectural Tiers

Traditional IT computing models (*e.g.*, direct connection between end-devices and the cloud[19]) do not suffice for the QoS needs of IoT systems. For most of the scenarios, these requirements are not met due to limitations in bandwidth in last-mile IoT networks, high latency, network unreliability, and increasing volume of the data being generated, transmitted and, *a posteriori*, analyzed [Han+17; Vas+19].

Fog computing has been proposed and used as a way to mitigate some of the challenges mentioned earlier. The architectural vision coined by Flavio Bonomi and Rodolfo Milito of Cisco Systems, circa 2015, focuses on distributed data management throughout the IoT system,

[17] "*47% of companies that have adopted IoT report that they do not have enough skilled workers, and 44% do not have enough available resources to train employees*" [Mic19].

[18] Latency is critical in several scenarios, *e.g.*, milliseconds matter for many types of industrial systems, such as when you are trying to prevent manufacturing line shutdowns or restore electrical service.

[19] Approach popularized by the commercialization of Platforms as a Services (PaaS) IoT solutions, also known as CoT (Cloud of Things) [Vas+19].

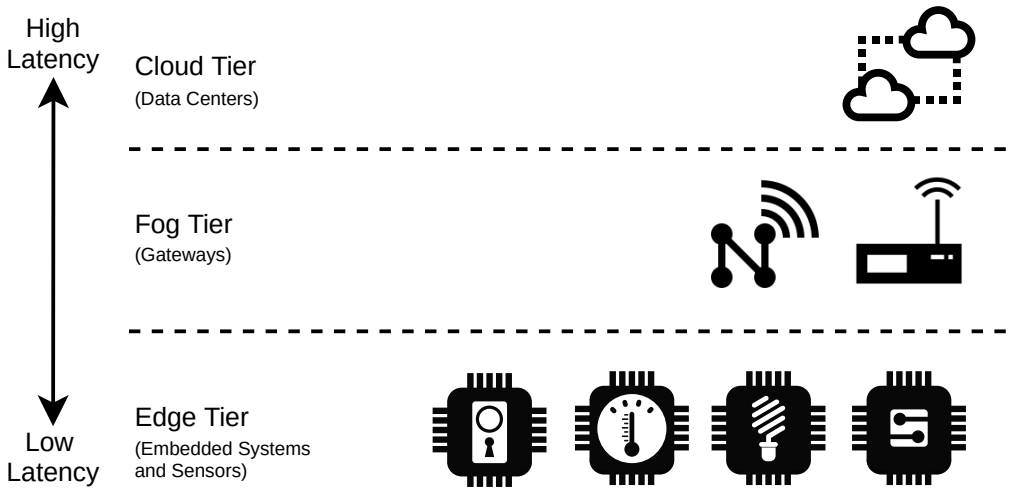


Figure 2.7: Typical architectural tiers composition of an Internet-of-Things system. The upper tiers have more latency and more computational power than the lower tiers of the stack.

as close to the edge of the network as possible [Han+17]. The National Institute of Standards and Technology (NIST) states that the fog computing model "facilitates the deployment of distributed, latency-aware applications and services, and consists of fog *nodes* residing between smart end-devices and centralized (cloud) services" [Ior+18].

Fog computing presents several advantages in terms of latency (*i.e.*, real-time interactions support), local proximity (*i.e.*, contextual location awareness), geographical distribution (*i.e.*, potentially increasing resilience), and battery life improvements. However, some disadvantages are also identified such as the increase in heterogeneity (*i.e.*, when compared to the homogeneousness of cloud computing) and the transience of their permanence in the network (*i.e.*, high-dynamic topology due to, *e.g.*, device mobility) [Vas+19; Ior+18; Han+17]. The *fog nodes* responsibilities can include data (pre-)processing, hosting middleware and other applications, and maintain communications between end-devices and the cloud (including external third-party services) [Vas+19].

Another, complementary, concept, is the one of edge computing (also known as mist computing). While both fog and mist/edge devices are at the edge of the network (on-premises), their differences reside on the device typology. Fog devices have, typically, more processing power, have a direct Internet connection, and can act as gateways for other devices. All the other devices that do not share this features fall under the mist category of devices. These devices, which are, mostly, the IoT sensors and actuators, have, typically, high computational constraints and leverage network protocols beyond the typical TCP/IP based ones [Vas+19].

However, as computing capabilities increase, some mist devices have enough computing capabilities to perform at least low-level analytics and filtering to make fundamental decisions, allowing for distributing computing tasks among them (making these constrained devices more autonomous and fault tolerant) [Han+17; Vas+19].

Cloud, fog, and edge computing paradigms opened doors to the widespread dissemination of the three-tier architectural hierarchy common in IoT systems, as depicted on

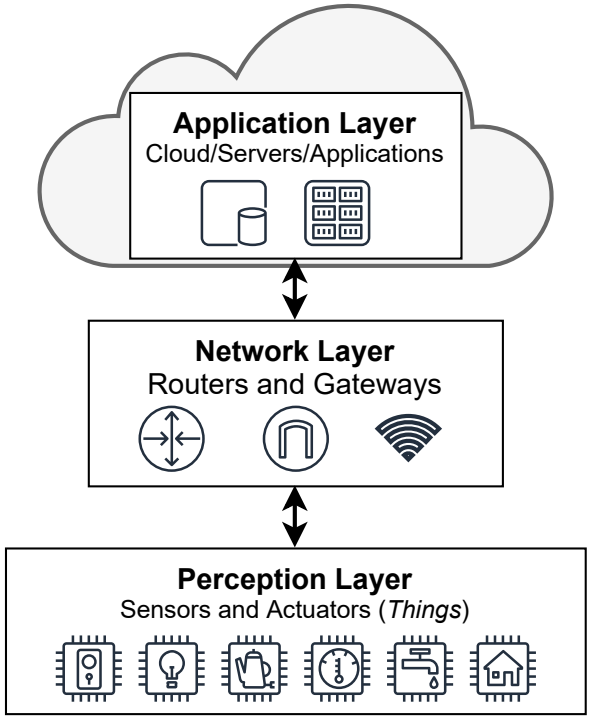


Figure 2.8: Logical view of the common layers of IoT systems.

Figure 2.7 (p. 39). This architecture is the foundation behind the architectural pattern *Fogxy* [STB18] (where these *tiers* are considered *layers*) and is observable in diverse IoT systems [Lin+17].

In the lower level are the IoT devices (*i.e.*, embedded systems and sensors/actuators), the edge/mist tier devices. After, and close to the edge-devices are the fog *nodes* which together make the fog tier. On the top are the data centers, the cloud tier. The applicability of this architectural hierarchy is visible in enterprise solutions such as the RedHat IoT enterprise architecture [Lin+17].

Despite the appearance of the presented concepts of fog and edge computing in several scientific works and practitioners' communications, there is still (1) a lack of consensus on the concrete definitions of what constitutes a fog or edge device and (2) a certain degree of ambiguity for the concept of both Fog Computing and Edge Computing (as an example, Edge Computing appears in the Cloud Computing literature as the use of computational resources geographically-closer to their clients/users). In this work, the presented concepts of fog and edge will be the ones used.

2.2.2 Architectural Styles

The Internet-of-Things leverages well-known building blocks of several software architectures in a unified fashion, where smart objects and humans responsible for operating them (if needed) are capable of universally and ubiquitously communicating with each other. Like almost any other computational system, IoT ones have an organizational structure, viz. system architecture.

At a high-level view, the architecture of IoT systems has the aforementioned three-tiered organization, *i.e.*, edge, fog, and cloud (*cf.* Figure 2.7, p. 39). However, when considering the specificities of IoT systems, the architecture needs to guarantee flawless operation of its components and *fuse* the physical and virtual realms. For reaching this objective, the architecture needs to be dependable, adaptable, highly scalable, human-centric, and handle dynamic interactions [BD16].

Some architectures are more suitable for the requirements of several IoT applications, such as low-power consumption, lightweight data transmission, messaging mechanisms, and reliability.

Regarding the software modules, packages, or components of IoT systems, the architectural interpretation of its division depends on the abstraction layer being used. Most modern IoT systems of modest-to-high complexity now tend to follow a Service-Oriented Architecture (SOA). Notwithstanding, several other architectures are being used for designing these systems, which can be used both jointly or individually. A mapping survey by Muccini *et al.* identifies the following as the most common architectural styles in IoT while mentioning that most systems use an overlay of one or more styles in their architecture [MM18]. The following presented some styles resulting from their mapping study with some additional relevant styles.

Layered Systems are built by a combination of several, heterogeneous, parts that interact between them. The number of layers goes from 3 to 6 across implementations, where the 3-layer architecture has a *perception*, *processing and storage*, and *application* layers and the 6-layer one presents the additional *adaptation*, *network*, and *business* layers [MM18].

Service-Oriented Architecture in which services are provided to other components or subsystems by using a common communication channel. As such, it is suitable for the connectivity, interoperability, and integration needs of IoT systems, where sensing and actuating services are highly distributed by design [ZDC14a; ZDC14b; UK18b; Tei+11].

Microservices Applications that follow a microservices architecture are structured as a set of services that are, as per Chris Richardson: (1) highly-maintainable and testable, (2) loosely coupled, (3) independently operable, and (4) organized around business capabilities [Ric17]. In IoT context, this architecture fits the SoS paradigm, adding the ability to encompass the different components and system parts that can be spread in both logical and geographical terms, while making it possible to assure scalability, extensibility, fault-tolerance, and scatter of services among tiers [UK18b; KJP15; SLM17; Lu+17].

Event-driven Event-driven Architectures (EDA) focuses on events — significant changes in the state of the system — and deals with the production, detection, consumption, and reaction to those events. IoT systems typically are built-on architectures that are already event-based, such as publish-subscribe (also known as pub/sub) — *architecture that states that producers publish events on an event bus (or message bus) and consumers subscribe to the events they want to receive from that bus* [Eug+03] — and Complex Event Processing — *architecture to combine events from diverse sources, looking for patterns in these event streams and then typically responding in real-time* — which provides aid in dealing with the ever-growing scale of IoT systems [Che+14; ACN10; ZDC14a; BGT16; STB18].

Broker and Mediator Broker architecture is commonly used for decoupling higher system tiers from the underlying tiers, *i.e.*, decoupling applications from the underlying sensing and actuating devices. A broker is a core component of most of the publish-subscribe architectures due to the dependency of a middle-man that receives messages from publishers and delivers them to different subscribers [Pre+16; Bla+10; CCV12]. Mediator architectures, supported by gateways and implemented using bridges and adapters, assure the communication between the devices and applications and can have responsibilities such as collecting and pre-processing data from multiple devices, device management capabilities, and assuring security and privacy [MZ14; Dav+16].

Client-Server One of the most common architectures on the Internet is that of a client and a server connected by some protocol such as Representational State Transfer (REST[20]) or Simple Object Access Protocol (SOAP) [GIM11; GTW+10; BD16]. Some IoT systems also use this architecture, and due to the constraints of some devices, new protocols have been developed as a way to meet this constraint, making them able to use the client-server architecture such as CoAP [Cas+11; Nai17b].

In their work, Muccini *et al.* , also lists the Cloud-based style — which refers to the use of cloud computing as a core part of the system — and Information-Centric Networking (ICN) style — that focus on making device communication information-centric — however, these overlap the already presented styles [MM18]. Other but less relevant architectures that also appear in this context are Peer-to-Peer, Space-based architecture, and Shared Nothing architecture. However, their repeated usages in the literature are mostly unnoticeable [CSM17; Bec+19].

2.2.3 Reference Architectures

Reference architectures and models give a bird's eye view of the whole underlying system; hence, their advantage over other architectures relies on providing a better and higher level of abstraction. They provide template solutions for an architecture for a particular domain (in this case, the IoT domain), hiding specific constraints and implementation details, aiming to help developers meet the development challenges of the domain [Clo+09].

Several entities are proposing and pushing into the market different reference architectures on how to develop systems that meet IoT characteristics and requirements. Weyrich *et al.* [WE16] survey circa 2016 refers that a fast-growing number of initiatives have been working towards standardized architectures for the IoT domain, aiming to facilitate interoperability between different solutions, simplify development, and ease implementation. Their work points out five major IoT reference architectures, namely: Reference Architecture Model Industrie 4.0 (RAMI 4.0), Industrial Internet Reference Architecture (IIRA), Internet of Things— Architecture (IoT-A), Standard for an Architectural Framework for the Internet of Things (IoT) and Arrowhead Framework. Also, they note the importance and relevance of initiatives to define machine-to-machine (M2M) communication standards and other initiatives related to the IoT domain. Saemaldahr *et al.* [Sae+21] survey circa 2020 found some other IoT-focused reference

[20] Also known as RESTful, which is used to describe services that follow a REST architecture.

Table 2.2: Reference Architectures for the IoT [WE16; Sae+21].

Initiative Name	Description
Reference Architecture Model Industrie 4.0 (RAMI 4.0)	A reference architecture for smart factories dedicated to IoT standards, which started in Germany and today is driven by all major companies and foundations in the relevant industry sectors.
Industrial Internet Reference Architecture (IIRA)	The Industrial Internet Consortium (founded by AT&T, Cisco, General Electric, IBM, and Intel) has delivered a reference architecture with a special focus on the industrial application of IoT.
Internet of Things — Architecture (IoT-A)	The IoT-A delivered a detailed architecture and model from the functional and information perspectives of the IoT system.
Standard for an Architectural Framework for IoT	The IEEE P2413 project has a working group on the IoT's architectural framework, highlighting protection, security, privacy, and safety issues.
Arrowhead Framework	Focus on enabling collaborative automation by open-networked embedded devices. It is a major EU project to deliver best practices for cooperative automation.
WSO2's Reference Architecture	A five-layer architecture with a special focus on modularity, making it suitable to wide array of use cases. It also suggest a core set of communication protocols to be used.
IoT Architecture Reference Model (ARM)	Based on IoT-A, it focuses on presenting a reference organization of the basic functional components without detailing their interactions. From an information viewpoint, the core view are virtual entities (*i.e.*, digital twins).

architectures, including the WSO2's Reference Architecture and IoT Architecture Reference Model (ARM). A summary of the hereby enumerated reference architectures for IoT is given on Table 2.2, detailing their focus and presenting some additional information [WE16; Sae+21].

Some practitioners also present their vendor-specific reference architectures which are tailored to their services and solutions, including: IBM Internet of Things architecture [IBM21], Azure IoT reference architecture [Azu21], and AWS Architecture Best Practices for IoT [AWS21].

Aware of the limitations and issues of the current architectures used in IoT, which are reflected in these reference architectures, Yaqoob *et al.* [Yaq+17] enumerates three challenges that need to be addressed by the next generation of architectures, including (1) resource control, *i.e.*, devices should be accessible and configurable remotely, (2) energy awareness, *e.g.*, automatically lowering processing power when system load is low, and (3) interference management, *i.e.*, with the number of devices communicating by different manners, mostly of them radio-based, interference can become a real problem. European Union projects, such as SENSEI [Pre+09], Internet of Things-Architecture (IoT-A) and FIWARE [Pre+16], have been addressing these challenges from a WSN viewpoint and have been successful in defining some common grounds [Gub+13].

But Weyrich *et al.* also conclude — a recurrent topic in this domain — that the scientific communities and industry leaders need to agree on standards to avoid systems that are crippled due to their base architecture (including the lack of interoperability, *cf.* Section 2.2.4, p. 44);

since not even a *de facto* reference architecture encompassing most kinds of IoT systems across different application domains was reached so far.

2.2.4 Interoperabilty

Different entities have been working on different standards to ensure a semantic interoperable Internet-of-Things, establishing a common language that devices can speak between them and different applications, envisioning a reduction on IoT fragmentation. A summary of the most known initiatives is given on Table 2.3 [GR19], and the most active ones are analyzed in the following paragraphs.

Table 2.3: Overview of the IoT interoperability enabling models and APIs.

Name	Description
Web of Things (WoT) [Fra17]	Common data model and API for the Internet-of-Things, with a focus on web-based interaction (*i.e.*, Web of Things).
IOTDB [Jan17b]	A semantic layer for the IoT, which includes definitions for all the data, to provide both,formal definitions for all important items and unlimited expandability.
Web Thing Model [TGC17]	A common model to describe the virtual counterpart of physical objects in the Web of Things.
OGC SensorThings API (SensorML) [LHK16]	An open, geospatial-enabled and unified way to interconnect IoT devices, data, and applications over the Web.
SENML [Jen+13]	The Media Types for Sensor Markup Language is a standard for representing simple sensor measurements and device parameters.
LsDL [Ray17]	The *Lemonbeat smart Device Language* is a XML-based smart devices encoding language that is read as Lemonbeat smart Device Language (LsDL).

Web of Things (WoT) A standard data model and API for the Web of Things[21]. The Web Thing item provides a vocabulary for describing physical devices connected to the World Wide Web in a machine-readable format with a default JSON encoding. The Web Thing REST API and Web Thing WebSocket API allow a web client to access the properties of devices, request the execution of actions, and subscribe to events representing a change in state. Some fundamental Web Thing Types are provided, and additional types can be defined using semantic extensions with JSON-LD. Also, the proposal includes details on Web of Things Gateway Protocol Bindings which proposes non-normative bindings of the Web Thing API to various existing IoT protocols and a set of Web of Things Integration Patterns which contains advices on which design patterns can be used for integrating connected devices with the Web of Things, and where each pattern is most appropriate [Fra17].

OGC SensorThings API by OGC An open, unified and geospatial-enabled way to interconnect the Internet-of-Things (IoT) devices, data, and applications over the Web purposed

[21] Web of Things (WoT) was part of the Mozilla WebThings initiative which is now an independent project.

by the Open Geospatial Consortium (OGC). At a high level, the OGC SensorThings API provides two main functionalities, and a part handles each function. The two parts are the Sensing part and the Tasking part. The Sensing part provides a standard way to manage and retrieve observations and metadata from heterogeneous IoT sensor systems while the Tasking part provides a standard way for parameterizing (*i.e.*, *tasking*) task-able IoT devices [LHK16].

IOTDB Things are described by semantically annotating the *data associated with the Thing*, being this item built from *composition* of *atomic* elements (that cannot be meaningfully subdivided further) and are *extensible* (allowing to add in elements from other Semantic ontologies). The *things* can have many *bands* of data associated with them (*e.g.*, the metadata, the actual state) [Jan17b].

There is, however, no consensus on what is the most suitable standard for a particular scenario, neither exists a commonly accepted standard that ensures semantic interoperability among IoT systems and applications [TTJ18].

The lack of consensus on interoperability standards is one of the most pressing issues that IoT faces, influencing all system stakeholders. From an end-user viewpoint, they do not know for how long the devices that they brought will be supported by the different system parts (*e.g.*, a light bulb that is no longer supported by the integration gateway). The same happens from the vendor point-of-view, since designing a system using the latest available standard proposal does not ensure that it will be adopted, or deprecated even before it reaches the market (which pressures vendors to define their in-house solutions contributing to today's fragmentation) [Buj+18].

Noura *et al.* [NAG19] defines a taxonomy for interoperability in IoT systems, analyzed from different perspectives: (1) device, (2) network, (3) syntactic, (4) semantic, and (5) platform level. They conclude that most IoT interoperability proposals focus on a single, specific perspective rather than embracing different ones. They also propose the use of semantic web technologies along with inter-working APIs as a good foundation for providing cross-platform interoperability. From the analyzed standards we believe that the one that goes more accordant with such view is the Mozilla Web Thing proposal.

The future seems gloomy, though, and most authors share the belief that it is not likely that a common set of standards will emerge in the near future and be universally accepted among academia, industry, and standardization bodies.

2.3 Fault-tolerant Systems

Because of our present inability to design and create error-free software systems, the software is hardly *perfect*. Due to the complexity of the developed systems altogether with the difficulty of asserting its correctness, software fault-tolerance is and will continue to be an essential concern on software lifecycle [Tor00]. Software faults are the root cause of a high percentage of operative system failures [Cho97]. The repercussions of such failures highly depend on the application and the particular characteristics of the fault or faults.

The reflection of Peter Neumann on more than 24 years of the ACM Risks Forum on its work Computer-Related Risk Futures, points out that even after 4 decades of sharing wisdom on how to reduce many risks of computer-related, this *wisdom* has been largely ignored in practice [Neu09].

2.3.1 Definition of Dependability

Avižienis *et al.* work Fundamental Concepts of Dependability defines dependability as a global concept that encompasses three different parts: (1) the threats to, (2) the attributes of, and (3) the means by which dependability is attained, which are dissected in the diagram of Figure 2.9 (p. 46) [ALR01]. Further, Avižienis *et al.* address security, as it is given by the CIA triad: confidentiality, integrity, and availability. It is verifiable that security shares some key attributes of dependability plus confidentiality [ALR01].

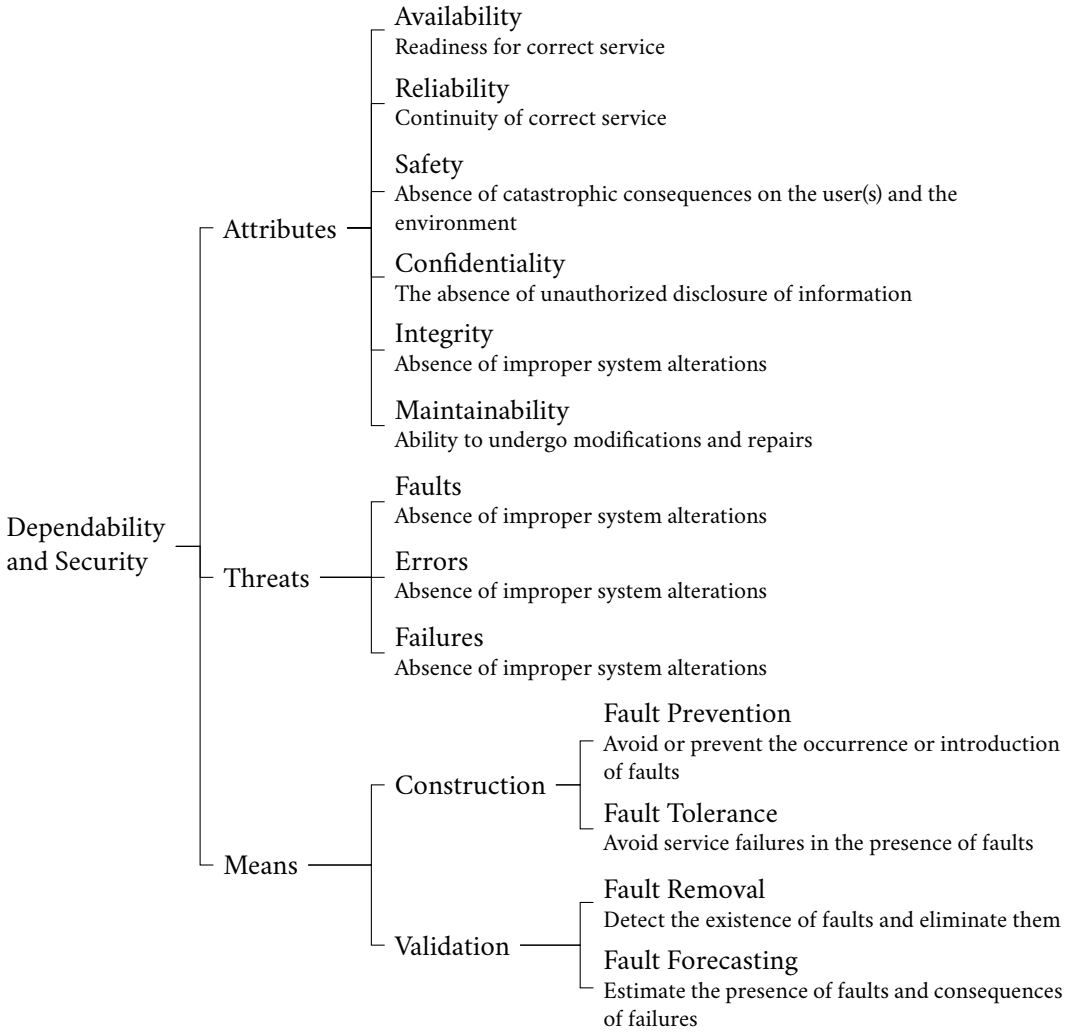


Figure 2.9: The dependability and security tree [Avi+04].

Dependability of a computing system is the ability to deliver service that can justifiably be trusted. A service is dependable if it has the *ability to avoid service failures that are more frequent and more severe than is acceptable* [ALR01].

A system is an entity that interacts with other entities and an environment is given by the array of entities or other systems, that surround a given system. A system and its environment are separated by a shared frontier, known as the system boundary.

The service delivered by a system is its behavior as it is perceived by its user that interacts with it at the service interface. The system function is *what the system is intended to do*. The user (physical, human) is always part of that system's environment, being the system the service provider for the user [Avi+04].

A system is said to be performing correct service when the service implements the system function, and, when a delivered service deviates from correct service, a system failure event occurs. The delivery of incorrect service is a system outage [Avi+04].

2.3.2 On Faults, Errors, and Failures

The threats to the dependability of a system are faults, errors, and failures. An overview of the chain of threats, also known as chain `Fault`→`Error`→`Failure`, is given in Figure 1.3 (p. 9).

A fault can then be described as the *the adjudged or hypothesized cause of an error* [Kni12]. Faults can be classified according to the behavior of the failed components. Avižienis *et al.* proposal on fault classification identifies 16 *elementary fault classes*, dissected on Figure 2.10 (p. 47), discriminating them in regard to their phase of creation/occurrence, place of origin, phenomenological cause, intent, persistence, and capabilities of the originators. The faults or their combinations belong to three major partially overlapping groupings, namely: development faults, physical faults, and interaction faults.

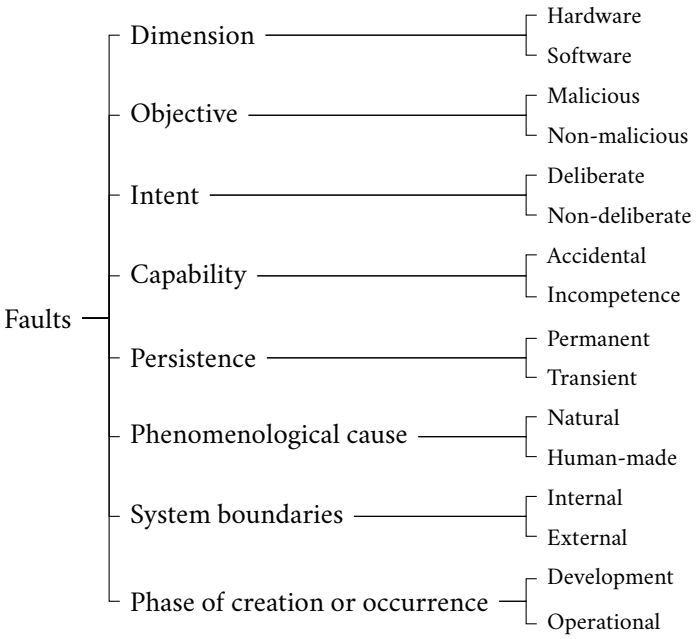


Figure 2.10: Dissection of the elementary fault classes. Avižienis *et al.* identified that there are 31 likely combinations of the 8 elementary classes [Avi+04].

In software scope, the only type of fault possible is a design fault introduced during the software development phase itself. Software faults are typically known as *bugs* [Tor00], but a

fault can be a variety of other things, such as, for example, a garbled message received on a communications channel.

A fault is active when it produces an error, otherwise, it is dormant. An active fault can be an internal, and previously dormant, that has been activated by the computation process or environmental conditions, or it can be an external fault. The application of input (the activation pattern) to a component that causes a dormant fault to become active is defined as fault activation. Often, internal faults cycle between their dormant and active states [ALR01].

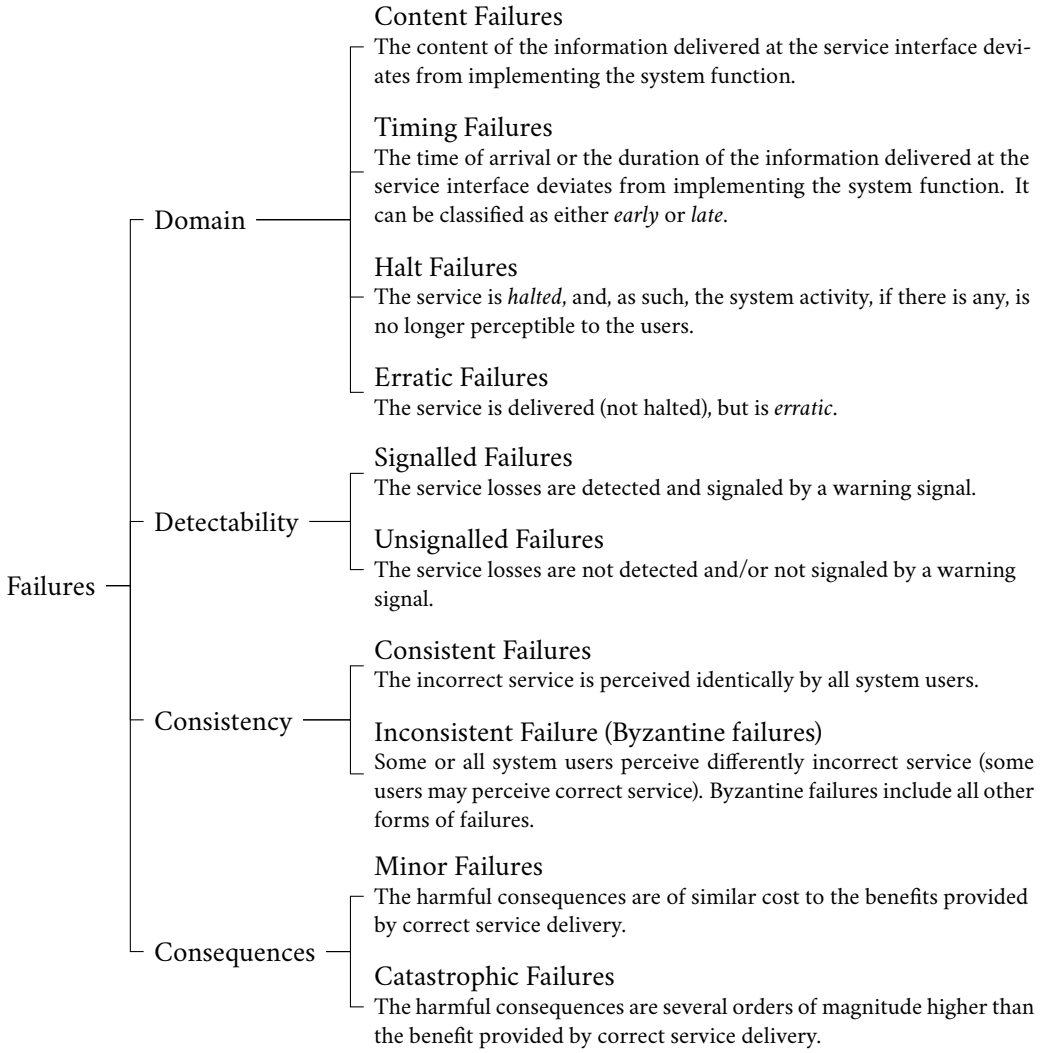


Figure 2.11: Overview of the service failure modes as defined by Avižienis *et al.* [Avi+04].

An error is the part of the system state that is liable to lead to a failure. Error propagation within a given system's component (internal propagation) is caused by the computation process, thus, an error is successively transformed into other errors. An error can be in one of two states: (1) detected if it is indicated by an error message/signal or (2) latent if it is present yet not detected [ALR01]. Further, when a single fault causes multiple errors they are called multiple related errors, and, in contrast, if it affects one component only are called single errors [Avi+04].

Failure happens when the system's behavior deviates from the intended behavior. It is the result of error propagation to the service interface in a way that unacceptably alters the service

delivered by the system. The different ways in which the deviation is manifested are a system's service failure modes. Failures can be classified in terms of domain, detectability, consistency, and consequences, as it is further analyzed in the diagram of Figure 2.11 (p. 48) [ALR01].

From a system perspective and in what concerns dependability, two software quality measures are reliability and safety. Reliability, as it is defined by Pressman *et al.* , is the *probability of failure-free operation of a computer program in a specified environment for a specified period of time*, where the failure-free operation in the context of software is interpreted as compliance to its requirements [Pre01; Tor00]. A measure of software reliability is the Mean Time Between Failures:

$$MTBF = MTTF + MTTR \tag{2.1}$$

Where:

$MTTF$: is the Mean Time To Failure.
$MTTR$: is the Mean Time To Repair.

MTTF is the measurement of how long a software item is expected to operate properly before the occurrence of a failure, and MTTR is the measurement of the maintainability of the software (*i.e.*, the degree of difficulty in repairing the software after failure).

2.3.3 Design of Fault-Tolerant Systems

For a system to be dependable, mechanisms should be put in place that detect, prevent, react, and forecast faults. Avižienis *et al.* in their work Basic concepts and taxonomy of dependable and secure computing present fault prevention, fault-tolerance, fault removal, and fault forecasting as the key concepts to attain dependability and security [Avi+04].

Fault prevention is concerned with the use of design methodologies, techniques, and technologies aimed at preventing the introduction of faults into the system, specially during design stages. Further, it is complemented by a process of fault removal that considers the use of techniques like reviews, analyzes, and testing to check an implementation and remove any faults thereby exposed. Proper use of software engineering during the software development process must contemplate fault prevention and fault removal.

As there is it hard to guarantee that complex software designs are free of design faults, fault-tolerance is used as a way improve the confidence that the system continues operational even when errors and failures ocurr. Fault-tolerance is the use of techniques to enable the continued delivery of services at an acceptable level of performance and safety after a design fault becomes active [Tor00]. It is achived by the use of mechanisms for error detection and system recovery as the diagram of Figure 2.12 (p. 50) exposes, giving an overview of the techniques for achiving fault-tolerant systems.

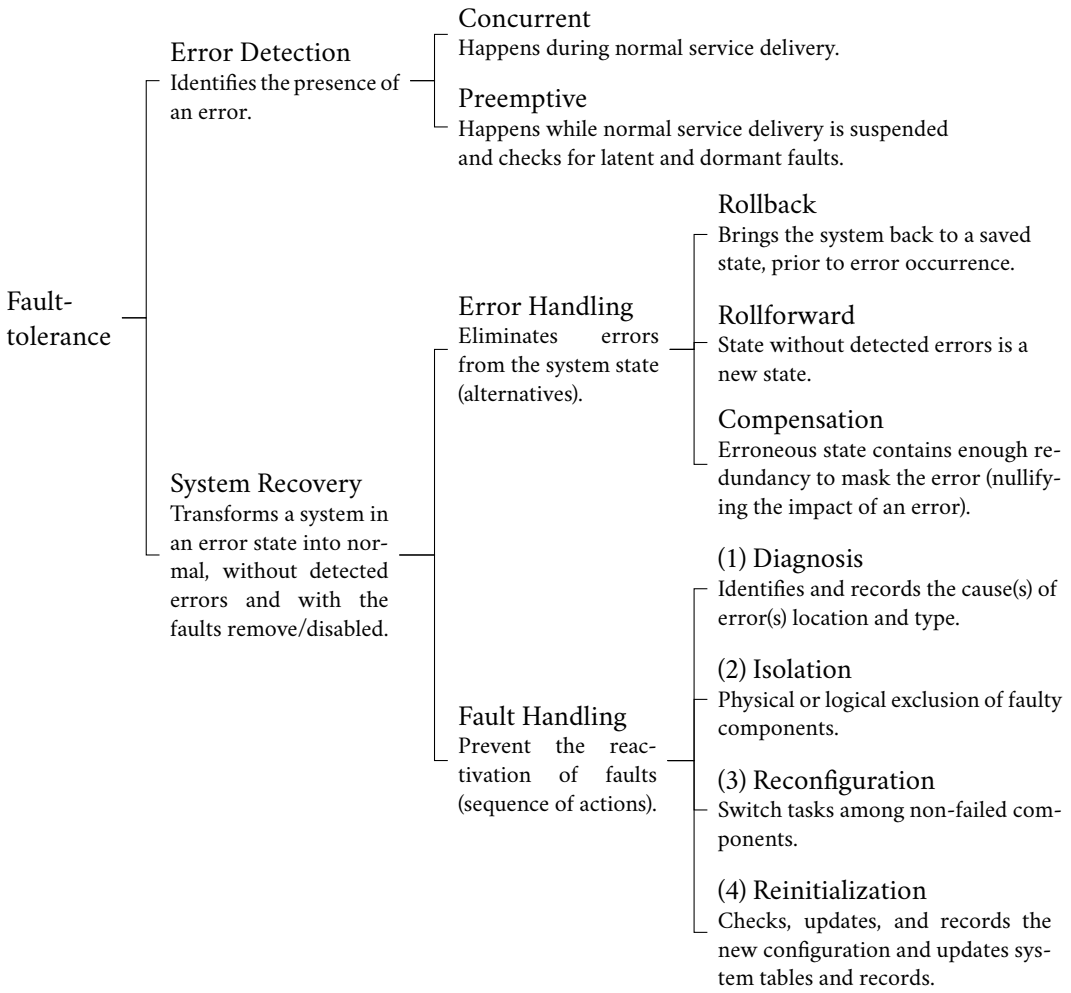


Figure 2.12: Summary of the fault-tolerance techniques as defined by Avizienis *et al.* [Avi+04].

Error detection refers to the set of techniques that are used to identify the presence of errors in a system. This can be of two kinds: concurrent detection which takes place during normal service delivery (*e.g.*, cyclic redundancy checks (CRCs)[22] in messages transmitted through a network) and preemptive detection which takes place while normal service delivery is suspended (*e.g.*, checks of the consistency of a file system during the booting of an operating system) [Avi+04].

System recovery is the process of transforming a *system state that contains one or more errors and (possibly) faults into a state without detected errors and without faults that can be activated again*, and consists of two different tactics which target different phases of the chain of threats (*cf.* Figure 1.3, p. 9), fault handling deals with the activation of faults thus preventing errors (intervenes between fault and error stages of the chain of threats) and error handling (also known as error processing) deals with the propagation of errors thus preventing failures (intervenes between error and failure stages of the chain of threats) [Avi+04].

[22] Technique invented by W. Wesley Peterson in 1961, CRC is a hash function that detects accidental changes to raw computer data commonly used in digital telecommunications networks and storage devices such as hard disk drives.

In order to deal with errors, *i.e.*, error handling, there is a set of typically used techniques. Rollback (*i.e.*, backward recovery), consists in restoring the system back to a previous saved correct state (*e.g.*, restoring corrupted files from a backup system). Roll-forward (*i.e.*, forward recovery), consists of replacing an erroneous state by a new state without errors (*e.g.*, overwriting an erroneous sensor value stored in memory with a new reading). Finally, compensation consists on masking faults (fault-masking), being the error concealed from the service delivered to a user by taking advantage of component redundancy (*e.g.*, having several identical components produce a result each and then applying a majority vote on these results) [Avi+04].

Fault handling (*i.e.*, fault treatment) consists of preventing failures by disabling the reactivation of faults. The fault handling process consists of four stages, namely: fault diagnosis, fault isolation, reconfiguration, and re-initialization, as described in the diagram of Figure 2.12 (p. 50). As such, fault handling is similar to the process that a repair person would do to fix a system (*e.g.*, a car): diagnose what is causing errors, isolate the responsible component, reconfigure the system so that it can continue delivering a correct service, and reinitialize any necessary components. The key difference is that fault handling does not involve a repair person, nor any other external agent, but is performed by the system.

The measure of the effectiveness of any given fault-tolerance technique is called its coverage. In order to reach a higher level of dependability, the fault-tolerance coverage should be the closest to perfect possible [Avi+04].

2.4 Autonomic Computing

IBM Research, in their efforts to address the problem of complexity, introduced the concept of autonomic computing as a way of coping with the continuous growth in the complexity of operating, managing, and integrating computing systems [PD11; GC03]. And, while the original manifesto for autonomic computing focused mostly on traditional computing systems, the same rationale applies for IoT with the factor that in these systems the scale is unprecedented when compared to any other type of system.

A truly autonomic computing system needs to *"know and understand itself"*, thus must be (1) automatic, meaning that they must be capable of controlling their own operations without any manual external intervention, (2) adaptive, meaning that the system should be able to adapt its operation to cope with temporal and spatial changes in its operational environment and (3) aware, thus the system must be able to monitor both internal and external operating conditions to assert if its current operation meets the service goals [Ric04]. Detailing these three vertices, designing and building systems that are capable of *running themselves* — self-governance — requires them to be able to have certain capabilities, namely [GC03; Hor01; Ric04]:

1. the system needs to *know itself* — awareness — in terms of components, operational status, capacity in terms of resources (what are its capabilities and limitations), and interconnections/interdependencies (why and how it is connected to other systems);

2. it must be capable of configure (*setup*) automatically, and re-configuring — adapt — itself to under varying, and, sometimes, unpredictable conditions to handle the new system environmental conditions;

3. it must monitor its constituent parts and act upon such metrics to find and implement optimizations (feedback control mechanisms that monitor the system parts and take appropriate action), ensuring the achievement of predetermined system goals (fine-tuning);

4. the system must be capable of discovering issues, or potential issues, within the system while it is running and proactively find ways to address the issue, *i.e.*, , find alternative ways to attain the system goals or (re-)configuring the system to keep it running while minimizing system degradation;

5. it must be capable of "*detect, identify and protect itself against various types of attacks to maintain overall system security and integrity*" without impacting the delivery of service;

6. the system should have contextual awareness of its surroundings, allowing it to adapt to changes with or within neighboring systems, converging for the best way of interacting with them. The systems should also be able to describe themselves (*e.g.*, in terms of features, resources, …) and automatically discover other devices or systems in the environment;

7. the system can not exist in a hermetic environment and must function in a heterogeneous world and implement open standards, thus can not be a proprietary solution nor rely upon proprietary parts/protocols;

8. it should be capable of anticipating the demand on its resources while keeping its complexity hidden, interacting seamlessly with other systems to respond to such demands.

A system that achieve these capabilities acts in accordance to four self-* (self-star) properties [GC03; Ric04]:

Self-configuring Systems are able to readjust themselves on-the-fly to cope with dynamically changing environments. This includes the ability to dynamically add new features, software, and hardware parts to a system without disruption of the provided services (plug-and-play).

Self-healing Ability to automatically discover, diagnose, and react to, or recover from, failures. Failing components should be identified, and their malfunctions should be solved without any apparent service disruptions (continuous availability).

Self-optimization Optimize resource utilization to improve the quality of the service. Such optimization should be carried out seamlessly across heterogeneous systems, providing a single collection of computing resources that can be managed by a logical workload manager, allowing dynamic redistribution of workloads even in unpredictable environments (increasing flexibility).

Self-protection Anticipating, detecting, identifying, and protecting themselves from attacks. Systems should be able to defend themselves against unauthorized resource access, be able to manage all user access, and detect, report, and prevent malicious activities as they occur.

Some authors expand these self-* properties with additional ones such as self-security, self-adaptation, and self-organization, but these are already encompassed by the original four properties, being only more specific views of certain sub-properties of them [TMD19].

Most of the self-* properties can leverage a model similar to feedback control loops such as OODA (observe, orient, decide, act), MAPE or MAPE-K (Monitor, Analyze, Plan and Execute with a Knowledge base), or cognitive cycle (sensing, analysis, decision, action) [Muc+18; Sei+19]. Even if having system parts that follow the principles of autonomic computing is a start, is *"the self-governing operation of the entire system, and not just parts of it, that delivers the ultimate benefit"* [Hor01]. The following paragraphs analyze in detail the four above-mentioned self-* properties.

2.4.1 Self-configuring

Autonomous systems must automatically adapt their configurations to dynamically changing environments *on-the-fly* to be considered self-configurable. The self-configuring nature of the system should encompass all the lifecycle of the system, meaning that the system should be able to both *setup* itself and adapt its configurations during its operation (*e.g.*, add or remove features, software, and hardware) without disrupting the delivery of normal service [GC03; Ric04].

As an example, in a IoT scenario, self-configuring primarily consists of (1) neighbor systems and service discovery, thus finding available resources and system capabilities, (2) network organization both in terms of establishing connections (*e.g.*, adjusting protocols and frequencies) and topology adjustments and (3) resource provisioning (*e.g.*, by request more or fewer resources in the system cloud counterpart to meet operational demands) [ADT13]. Further, the term orchestration, popularized by cloud computing, focuses on configuring large-scale systems and, sometimes, complex architectures; doing this orchestration dynamically also falls under the umbrella of self-configuration [Feh+14].

Self-configuring systems features include the ability to *plug-and-play*, mostly automatic configuration wizards, and remote management. *Plug-and-play* is especially important when we consider the mobility of certain parts of the network (*i.e.*, high-dynamic network topology), where the system must (re-)configure itself in the presence or absence of certain devices or other resources [GC03].

Systems that have this property allow all of these (re-)configurations to happen with minimal to no human intervention. This also requires the system to implement dynamic software configuration techniques along with awareness capabilities that allow it to dynamically identify and document the characteristics of the configurable parts, while guaranteeing that the configuration changes allow the system to comply with the required service levels (*i.e.*, feedback-loop) [Ric04].

2.4.2 Self-healing

Self-healing is the capability of a system to discover, diagnose, and react to disruptions such as failures of systems' parts. This process can be both predictive, using, for example, heuristics to predict failures and act accordingly (preventive measures), and reactive, act upon the discovery of a failure [GC03].

Self-healing is already present to some degree for some time in computing systems. As an example, error checking and correction have been long used to keep data transmission reliable (*e.g.*, TCP/IP) and redundant storage data recovery capabilities (*e.g.*, RAID). However, the growing complexity of computing systems makes it hard to diagnose the root cause of a problem (*i.e.*, root-cause analysis[23]), even in simple scenarios [Hor01]. But, since the goal of self-healing is to minimize interruptions and service restoration, there is the need for an *action-oriented* approach that determines what immediate actions are required to stabilizing the system and proceed with the recovery.

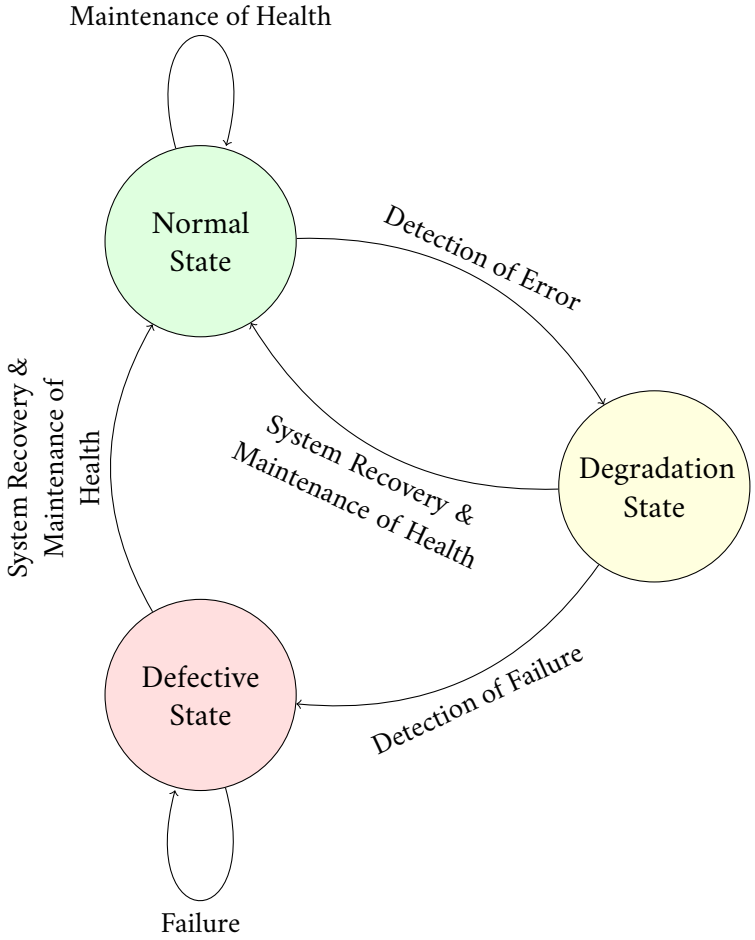


Figure 2.13: State transactions of a self-healing system as defined by Khalil *et al.* [Kha+19].

Ghosh *et al.* [Gho+07] describe systems with self-healing capabilities to be those that can deal with disruptions in their operation by (1) detecting system failures and possibly diagnosing

[23] Root-cause analysis is the "*attempt to systematically examine what did what to whom and to home in on the origin of the problem*" [Hor01].

the root cause of the problem, (2) determining a fix (*i.e.*, maintenance of health), and (3) recovering (even if only to a less capable but safe and healthy state), viz. Figure 2.13 (p. 54). Self-healing may use models (*external* or *internal*) that monitor the system's behavior (*probes*), allowing it to adapt to environmental or operational circumstances. These approaches can be *intrusive*, if implemented internally within the system itself, or *non-intrusive*, if they consider the guarded system as a complete unit; they are *closed-loop* when they try to avoid all a priori known failure sources (*i.e.*, all possible states are known before recovery), or *open-loop* otherwise [PD11]. The typical recovery mechanisms employed include reconfiguration and replication of components (hardware and software) and degradation of the QoS [AA19].

2.4.3 Self-optimization

Optimization of available resources to meet service demands is also a key part of computing systems. The core idea of self-optimization is that these systems are able to continuously monitor the running system — and its resource usage — to check if the predefined system goals or performance levels are met while continuously search for opportunities to optimize resource utilization and tune the system accordingly, automatically, even in unpredictable environments [Hor01].

Allowing a system to self-optimize requires the use of feedback control mechanisms to monitor the system metrics and take appropriate action, similar to the control theory feedback loops used in industrial systems[24]. Another common use of self-optimization is the dynamic and automatic provision of computing resources common in cloud computing to meet operational demands [Feh+14].

And, while self-optimization can play a key role in the optimization of resources such as network load, processor load and storage, and memory, in IoT other metrics come at play than can also be optimized such as battery/energy consumption (*e.g.*, devices' sleep and wake-up cycles) and deciding what is the best frequency/protocol taking into account the surrounding conditions.

2.4.4 Self-protection

Autonomic systems should self-protect themselves against various types of attacks (both internal and external) to maintain overall system security and integrity. To do so, the system should anticipate, detect, identify attacks — which often occur on a daily basis — with diverse origins. Systems should be able to defend themselves against unauthorized resource access, be able to manage all user access, detect, report, and prevent malicious activities as they occur, and also detect security issues on devices and systems and automatically respond to them by following (pre-)defined mitigation plans [Hor01; GC03].

Self-protection should address all the system aspects of security, including platforms, operating systems, networks, applications, and infrastructure. All the system events should be

[24] In control systems, a feedback loop (closed-loop control) is used to control the output of a system or device by feeding back all or some portion of the output back into the system input [Sam16].

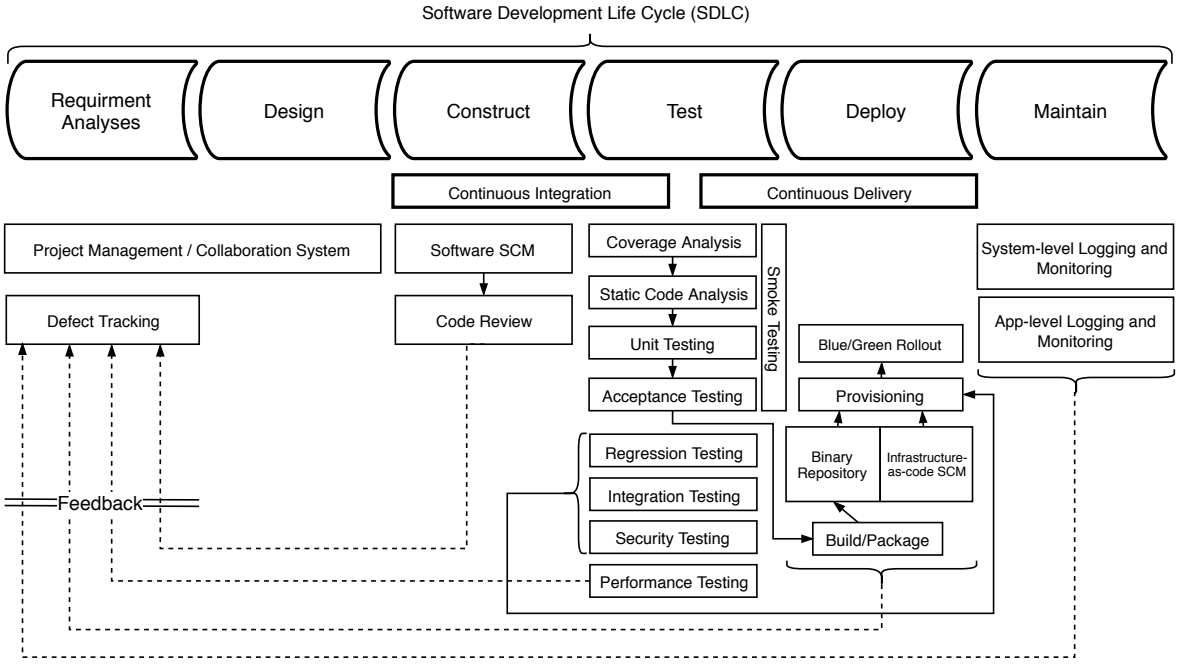


Figure 2.14: Software Development Life Cycle holistic overview, detailing most common processes and development stages [Fee16].

continuously recorded to allow both automatic and manual audits to be performed if an indicator of system compromise is detected. Threat levels should also be defined, allowing the system automatic response to be different in accordance with the severity of the threat [Mur04].

2.5 Software Development Life-Cycle

Software Development Life-Cycle is a process of building or maintaining software systems. Typically, it includes various phases, from preliminary development analysis (*e.g.*, requirements, architectural design) to post-development software testing and evaluation (*e.g.*, verification and validation) [Lea+12].

SDLC also encompasses the models and methodologies that the development teams use to develop software systems, in which the methodologies form the framework for planning and controlling the entire development process. Currently, there are two principal methodologies categories, the Traditional Software Development ones (*e.g.*, waterfall) and the AGILE Software Development ones (*e.g.*, SCRUM) [Lea+12]. A mostly complete view of the phases and tasks that are part of a SDLC is depicted on Figure 2.14 (p. 56).

The application of the widespread SDLC processes to design, construct, test, deploy and maintain IoT systems face different challenges than the ones that are faced when developing traditional software systems, due to the inherent peculiarities of the IoT ecosystem.

Taivalsaari *et al.* delves into this question and identifies the following as the key challenges for developing software in the context of IoT [TM17]: (1) multi-device programming (from tiny devices to cloud servers), (2) the reactive and always-on nature of the system (real-time and *human-in-the-loop*), (3) heterogeneity and diversity, (4) the distributed, highly dynamic,

and potentially migratory nature of software and hardware components, and (5) the general need to write software in a fault-tolerant and defensive manner (from a safety and security perspective). These challenges, along with the unsuitability of the existing languages and tools, have a most direct impact on software development and its lifecycle.

Summarly, on the one hand, from a technological viewpoint, there is a considerable amount of gaps in the software engineering body of knowledge regarding IoT. On the other hand, from a developers' viewpoint, there is the need for the developers to have a broader base of knowledge that ranges from large-scale systems to embedded system's programming.

2.5.1 Patterns

The current body of knowledge on IoT systems' design is *widespread, diffuse, hard to handle, redundant, ever-changing* and sometimes even *unorganized* as different authors chose different vocabulary to target the same things (ambiguity). When a single domain reaches this level of complexity, the possible decisions start increasingly growing beyond the reach of single designers [AIS77]. The purpose of *design, as an intentional act of choice, is continuously overwhelmed by the sheer quantity of available information* [Fer11].

Christopher Alexander, circa 1977, was faced with a similar challenge regarding civil architecture and came up with the idea of *patterns* — recurrent problems with recurrent solutions — to capture this widespread knowledge coherently. In his book A Pattern Language, he uses this concept of a pattern as a way to document the architecture and urban design solutions at such time[25]. He further stands that [AIS77]:

> *Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.*

In his research, he further extended the notion of patterns beyond the <*problem, forces, solution*> triplet, towards a *pattern language*, which also considers the relationship between different patterns in a specific domain.

These concepts were later borrowed by the software engineering community as a way to capture and share practical knowledge and experience [MD97; Gam+95; Gab96; HW04; Fow02]. It is widely accepted that a pattern corresponds to a recurrent solution for a specific problem, that is able to achieve an optimal balance among a set of forces in a specific context, yet taking into account the consequences of it.

In software engineering the use of patterns ranges from the high-level *architectural patterns*, through *design patterns* until low-level *idioms* [BMR96]:

- **Architectural Patterns** express fundamental structural organization schemes for software systems, decomposing them into subsystems, along with their responsibilities and interrelations.

[25] Alexander had an M.Sc. in Mathematics and was a civil architect, and his theories on patterns emerged from the observation of cities and buildings.

- **Patterns** are medium-scale tactical patterns, specified in terms of interactions between elements of object-oriented design, such as classes, relations, and objects, providing generic, prescriptive templates to be instantiated in concrete situations. They do not influence overall system structure but instead define micro-architectures of subsystems and components.

- **Idioms** (*i.e.*, coding patterns) are low-level patterns that describe how to implement particular aspects of components or relationships using the features of a specific programming language.

The knowledge contained in these patterns and pattern languages is a result of a synthesizing process of systematic analysis and documentation of scattered empirical knowledge, and has, as of today, a profound impact on the way that developers design, build and manage software artifacts.

2.5.2 Development Approaches

While the design activity of the software development lifecycle focuses on reaching suitable and high-level solutions for a given problem, the construction (also known as implementation) activity of the lifecycle focuses on actually building — developing — the systems (*i.e.*, *create an executable version of the software* [Som10]).

The construction of software systems may involve writing code in one or more high– or low-level programming languages (at different abstraction levels), or tailoring and adapting generic, *off-the-shelf* systems to meet the specific requirements of a given project, organization, or scenario [Som10].

Visual Programming

Diagrams, and other graphical logic and model representations, have been playing a role in software development since the appearance of modern digital computers in the 1940s. In the beginning, diagrams were paper-based aids, used to design and understand the software structure, but then, the direct use of diagrams as a solution to improve software development tools as started gaining momentum. This led to the appearance of visual software project management tools, visual editors for graphical interface creation, visual tools for software modeling and engineering, and visual programming languages [Cox07].

A VPL can be defined, as described in the *Wiley Encyclopedia of Computer Science and Engineering* [Cha02], as:

> *A language in which significant parts of the structure of a program are represented in a pictorial notation, which may include icons, connecting lines indicating relationships, motion, color, texture, shading, or any other non-textual device.*

Visual programming makes use of an extensive set of icons and diagrams to convey information and to allow for multi-modal communication and interaction between humans and computers [Cha02].

VPLs have been explored and used in several domains, including, but not limited to, educational activities (*e.g.*, learning to program), multimedia, video game development, system design and development, simulations, automation, data warehousing, and business analytics [Ray17].

In favor of using visual programming, several researchers point out the dramatic reductions in time and cost when developing with this approach within industry scenarios [Ost02]. Further, the achieved improvements in productivity and reliability were considerably noticeable [Ost02].

However, visual programming, having several benefits and drawbacks, is not the *silver bullet* for software development; some authors even suggest that the solution for the adoption of visual notations lies in a hybrid visual-textual approach. Visual notations are more effective when dealing directly with an application domain (*e.g.*, IoT), and have several drawbacks when applied to general purposes, especially when dealing with complex control structures and recursion [EM95]. Further, the developer experience, when using such languages, depends significantly on its previous programming experience, how quickly they can create and debug programs, and how easy they can maintain applications over time [MM01].

There is a long-established relationship between models and visual programming, as is pointed out by the work of Lau-Kee *et al.* [Lau+91] circa 1991 that mentions the use of data flow as the computational model underlying the visual programming features of systems such as the LabVIEW. Such relationship between models and visual programming languages is visible as of today in systems such as the Executable UML (xtUML or xUML), which are at the same time a visual notation and a visual programming language [MBF02].

Model-driven Development

A software engineering paradigm that debuted back in 2001 [Bel+03] as an answer to the growing complexity of system architectures is known as Model-Driven Software Engineering (MDSE), a subset of Model-Driven Engineering (MDE). Douglas Schmidt [Sch06] points out that, the growth of complexity in software systems, together with the lack of an integrated view of the system under development *often forces developers to implement sub-optimal solutions that unnecessarily duplicate code, violate key architectural principles, and complicate system evolution and quality assurance*.

The use of (mostly graphical) domain-specific models, allows the different stakeholders to improve their understanding of the system. Further, the transformation engines provide automatic mechanisms to generate new models from existing models (or to update those models), thus facilitating the transfer of models between phases of the software development lifecycle, while ensuring consistency between them [Ams+12].

A key concept associated with MDSE is one of the meta-models. Meta-models are definitions of the *source* and *target* of a model transformation. The elements of the meta-models are used in the model transformation definition for defining the model transformation, by the transformation engines [Küs11]. The model transformations can be unidirectional (a source model is transformed into a target model) or bidirectional (a source model is transformed into a target model and *vice-versa*) [Küs11]. Further, model transformations can be, if the source and target modeling language are not the same, exogenous, otherwise are endogenous. An overview o process of model transformation is given on Figure 2.16 (p. 60).

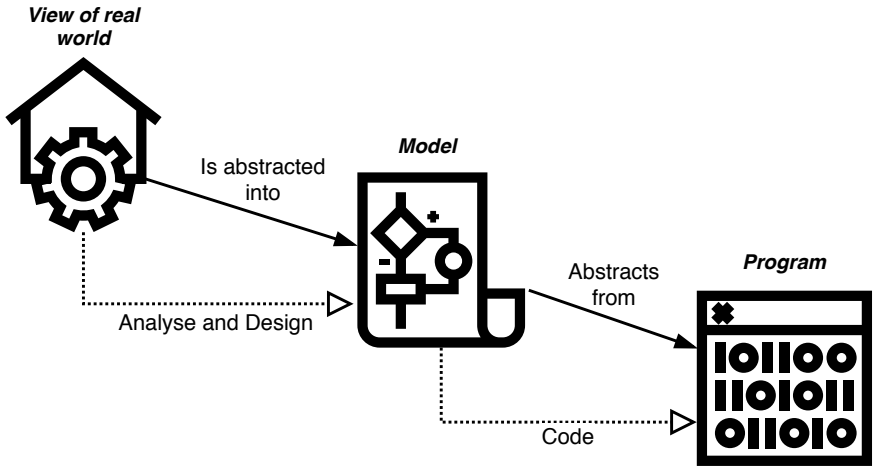


Figure 2.15: Real-world, model and program relationship overview [Küs11].

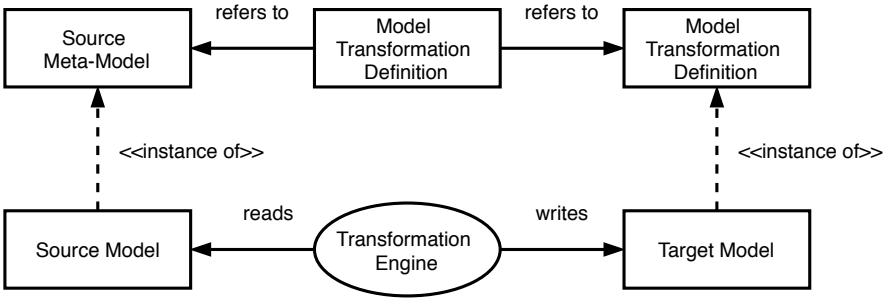


Figure 2.16: Basic concepts of model transformation in MDSE [CH06].

Model-driven approaches are considered to have several advantages over traditional approaches to software development, including the following [Rie+01]:

Shorter time-to-market Users model their domains rather than implement them. A modeling language like UML is more suitable to express domain models than a programming language like Java or Smalltalk;

Increased reuse and fewer bugs The tools hide the details of how the models are hooked up into the runtime system, freeing users from knowing intricate details about used frameworks or system components;

Straightforward system and up-to-date documentation As the design and implementation are in sync, documentation can be maintained up-to-date, making the system easier to understand at any point of its life-cycle.

However, the same authors point out that code-generation does not solve all the problems; identifying as the main drawback the *delay between model change and model instance execution*. They continue arguing that *generating code from models, compiling the code, shutting down the existing system, installing and configuring the new system, and starting it up can take from minutes to hours* [Rie+01]. Nonetheless, such a drawback is a common property of typical deployment and evolution of software systems, not being a particular downside of MDSE.

More recently, work has been carried out under the banner of *models@run.time*, embracing the vision of models as an abstraction of a system, and pushing it further, seeking to understand the roles that such models can play at runtime [BBF09]. The idea of having high-level specifications of, for example, data or routines that are interpreted at runtime is research since long ago. Examples include the use of meta-data and meta-specification in Aspect-Oriented Programming (AOP) [Fer11; WLA06; Amo+12].

Blair *et al.* define *models@run.time* as *a causally connected self-representation of the associated system that emphasizes the structure, behavior, or goals of the system from a problem space perspective* [BBF09]. System users can use runtime models to support dynamic state monitoring and control of systems during execution, to dynamically observe the runtime behavior, to understand a behavioral phenomenon, to promote the semantic integration of heterogeneous software elements on runtime, or even to fix design errors or adopt new design decisions during runtime.

The potential support of *models@run.time* for the evolution of software design can blur the line between development models (MDSE) and runtime models, thus becoming a live development model [Agu+19] that enables dynamic growth and the realization of software designs, support adaptation decisions by humans or by agents embedded in the system itself or through combinations of both [BBF09].

Mashup-based Development

The term mashup is vast and widespread. Using as definition the one purposed by Rümpel *et al.* [RM12], mashup tools are solutions that allow developers to construct applications and systems in a component-based fashion (*e.g.*, Widgets) or Web service composition (*e.g.*, REST APIs), *i.e.*, building applications by *mashing-up* existing components. Early examples of mashup tools are Microsoft Popfly [Lot08] and Yahoo Pipes [Pru07].

According to Maximilien *et al.* [Max+07], mashup tools are characterized by providing the following facilities out-of-the-box:

Data mediation by providing mechanisms to convert, transform and combine data from one or more services (*e.g.*, different APIs) to satisfy the needs of another service.

Process mediation also known as *protocol mediation*, are essentially choreographing mechanisms for interconnecting the different services to create a new process. As an example, process mediation can include invoking different service methods, waiting for asynchronous messages, and dispatching the needed confirmation messages.

User interface customization which are mechanisms to build interfaces that can display progress or final process information to the end-user (*e.g.*, build Web-based dashboards to display information supporting different user interactions).

Typically, mashup solutions are Web-based and provide a visual editor for composing the different services into an application or system. As such, several mashup tools provide a visual programming environment that allows the definition of the message flow between components, viz. *nodes* (*e.g.*, sensors and actuators, processing units, aggregations, and external Web-based

services). Due to this fact, most of the mashup-based solutions are also visual programming environments [PC13].

2.5.3 Software and System Testing

Testing is the process of identifying faults (also known as *bugs*) that push systems into an error state and can eventually lead to failures [ALR01]. Faults are (mostly) human-related mistakes that exist in the system, and testing focus on finding them before deploying the system into production where failures can have nefarious effects.

As the development/deployment barrier can be significantly eroded for IoT, the traditional separation of these phases may no longer remain suitable, since that for a system to work as a whole, there exists a dependency on different software and hardware components, modules, and architectures, produced by many manufacturers and with different working properties.

Testing can be done at different levels, depending on the scope of the test and its objective. Different test levels [Bei03; Com90] are defined as follows:

Unit Testing Testing of a single hardware or software unit (or groups of related units). It consists of isolating each part of the system and demonstate that individual pieces work, at least, in isolation.

Integration Testing Software and hardware components are combined and tested to check the interaction between them and how they perform together.

System Testing Testing a complete, integrated system to check the system's compliance and behavior within the specified requirements.

Acceptance Testing Formal testing is conducted to determine whether a system satisfies its acceptance criteria and to enable a customer, a user, or other authorized entity to determine whether to accept the system.

Different methods can be used to test the System Under Testing (SUT), namely, white-box testing [Ost02], gray-box testing [Lin+04] and black-box testing [Edw01]. These methods are described as:

White-box Testing The internals of the SUT are all visible and known, and, as such, this information can be used to create test scenarios. White-box testing is not restricted to fault detection but is also able to detect error states.

Black-box Testing The SUT internal contents are hidden, and only knowledge about the system's or module's inputs and outputs is known, being closer to real-world use situations. Black-box testing can only detect failures; the program needs to inspect the code to find the fault that caused it.

Gray-box Testing A mix of the two previous techniques. Information about the internals of the SUT is used, but tests are conducted under realistic conditions, where only failures are detected.

The different types of tests can be carried using different (and more than one) methodology. The confidence that a given system will comply with its requirements depends highly on the number and variety of tests at different levels and leveraging different methodologies.

2.6 Summary

This chapter introduces some key background concepts that are considered relevant knowledge to fully grasp this work. It also attempts to reduce the ambiguity of some terms and concepts by presenting definitions that will be the ones used in this work. Starting by presenting the main subject — Internet-of-Things — we provide some historical context to IoT and its core vision, while focusing on its broadly adoption across application domains. We also briefly analyze the technological context, from both hardware and software perspectives, that facilitated and open doors to the ubiquity of IoT, including, but not limited to, the communication protocols, single board computers, and microcontrollers.

A mostly brief overview of some software development concepts, including the most widely used architectural styles, patterns, software verification and validation techniques, and other aspects of the software development lifecyle is also presented, while providing some insights from an IoT system perspective.

The topic of fault-tolerance — both in software and in hardware — is introduced, by presenting the some key works that present and systematize the available knowledge of the field, defining some of the glossary that is used along this document. Lastly, the concept of autonomic computing as defined by IBM circa 2004 is presented, including its core vision, proprieties and self-* aspects.

3 | State-of-the-Art

Internet-of-Things have been reshaping the way individuals interact with their surroundings, automatizing manual tasks, and continuously gathering information that allows them to make data-supported previsions and decisions. As these systems widespread, questions arise on how to effectively design, construct, test, orchestrate, operate, and manage them at a large scale while being widely heterogeneous [Dig+19]. Ensuring the systems' dependability also poses novel challenges since it requires efforts from both hardware and software communities as it implies considerations that go beyond the everyday challenges of developing *software-only* systems (*e.g.*, mobile and web applications). Although well-known software practices can be harnessed and adapted to the IoT development challenges, there is still a need to develop new approaches, environments, and best practices [Lar+17]. Zambonelli *et al.* [Zam17] goes further by proposing a novel *IoT-oriented software engineering* field that brings together the overlapping engineering areas involved in IoT which eventually could *lead to the identification of general models, methodologies, and tools* for developing these systems.

In this chapter, we present the current approaches that play a key role in the lifecycle of IoT systems, while providing insights on the current efforts on fault-tolerance and autonomic computing in the context of this type of systems. Along the different sections, we discuss the current gaps in the body of knowledge while highlighting open research challenges. Summarily, we conclude that there are still several challenges that must be addressed to fulfill the IoT potential, some of which can, potentially, leverage knowledge from other, and, related, bodies-of-knowledge, such as cloud computing and mission-critical systems.

> Parts of this chapter were published in the works A BRIEF OVERVIEW OF EXISTING TOOLS FOR TESTING THE INTERNET-OF-THINGS [Dia+18], A REVIEW ON VISUAL PROGRAMMING FOR DISTRIBUTED COMPUTATION IN IOT [Sil+21], and DESIGNING AND CONSTRUCTING INTERNET-OF-THINGS SYSTEMS: AN OVERVIEW OF THE ECOSYSTEM [DRF22].

3.1 Designing IoT Systems

The design of software systems is the *creative activity in which you identify software components and their relationships* based on the system's requirements (*i.e., design is about how to solve a problem*), and the most common outputs of this process are the system's models and documentation [Som10].

While design encompasses several concerns including hardware considerations, picking the most suitable protocol and communication standard, as well as architecture considerations, evolving the current body of knowledge in these regards fall out of the scope of this thesis. Taking this into consideration, in the following paragraphs a focus will be given on patterns and pattern-languages in IoT context.

3.1.1 Patterns

Internet-of-Things is a relatively recent field for both academia and industry, resulting in large amounts of knowledge being created rapidly and disseminated in a variety of formats. The system's designers are tasked to pick and represent the best solutions that cope with the desired scenario mostly based on empirical evidence, usually captured as standard reference architectures (*cf.* Section 2.2.2, p. 40) and case studies.

The work by Washizaki *et al.* attempt to map the landscape of IoT patterns resulted in a total of 136 patterns scattered among abstraction levels and IoT layers, and the identified works are included in Table 3.1, but their work does not encompass several works due to the reduced number of scientific databases considered, while encompassing others that do not focus on design patterns per se [Was+19; Was+20].

Thus, using as knowledge base the *software engineering pattern literature*[1], we surveyed it for works specifically concerning IoT or for nearby fields (such as cloud computing) that we believe share a considerable subset of concerns with our domain. We only considered works which identify patterns as a primary outcome.

A summary of the analyzed publications is presented in Table 3.1 (p. 66), and the identified design patterns in the context of IoT systems are listed in Table 3.2 (p. 67). An overview of the literature shows that various works focusing on patterns for these systems and their lifecycle, primarily focus on the edge tier, viz. the *things* themselves. Even though we also found existing work tackling cloud computing, there is a (somewhat variable-length) gap concerning the IoT-specific cloud and fog tiers, and on how all of them come together.

This latter fact raises the hypothesis that there may exist a non-negligible amount of knowledge in other areas of software engineering that is not being researched for IoT. Being aware of this, could probably help bootstrap well-known software development practices (*e.g.*, continuous integration, continuous development, isolation/containerization, fuzzing and fault injection) for our target systems, as they share similar, if not equal, characteristics, such as distributed computing, fault-tolerance, and large-scale systems' orchestration.

[1] Patterns and pattern languages are usually published in the PLoP conference series and made available in the ACM-DL.

Table 3.1: The landscape of relevant literature about design patterns for the Internet-of-Things, and their relevance for each IoT architectural layer by ranking, from no relevance ( ) to most relevant (•••).

		IoT Tier		
		Cloud	Fog	Edge
IoT Focused	IoT Patterns for Device Bootstrapping and Registration [Rei+17a]	•	•	•••
	IoT Security Patterns [Luk+17]		•	•••
	Internet of Things Patterns [Rei+16]		•	•••
	Internet of Things Patterns for Devices [Rei+17b]			•••
	IoT design patterns: Computational constructs to design, build and engineer edge applications [Qan+16]	•	••	•••
	Design patterns for the industrial Internet of Things [Blo+18]	••	••	••
	Patterns for Devices: Powering, Operating, and Sensing [Rei+17c]			•••
	Fogxy — An Architectural Pattern for Fog Computing [STB18]	•••	•••	•••
	Cataloging design patterns for Internet of Things artifact integration [Tka+18]		•••	•••
	Model-driven development of user interfaces for IoT systems via domain-specific components and patterns [BUA17]	••	••	•••
	An ontology design pattern for IoT device tagging systems [Cha+15]			•••
Relevant for IoT	Cloud Computing Patterns [Feh+14]	•••		
	A Pattern Language For Microservices [Ric17]	•••	•	
	Continuous Integration: Patterns and Anti-Patterns [DGM07]	•••		
	Patterns for Fault Tolerant Software [Han07]			•••
	Designing Distributed Control Systems: A Pattern Language [Elo+14]	••	••	•
	Patterns for software orchestration on the cloud [SCF15]	•••		

The adoption of these existing patterns and practices in a new context should not be taken lightly. As an example, consider the case of continuous integration/delivery. The typical CI/CD approach assumes that low-friction tools for delivery exist (*e.g.*, a containerized pipeline) and that software is made to work in a well-defined set of hardware and/or platform configurations [SAZ17].

3.1.2 Discussion

When taking into account the number of open technological challenges, despite the pattern mining effort done work by Reinfurt *et al.* [Rei+17b; Rei+17c; Rei+17a; Rei+16], the number of captured patterns is still residual. Existing patterns do not encompass the ecosystem as a whole and are tied to specific architectural tiers and to specific hardware/software perspectives. Extensive work should be pursued on the systematization of existing solutions (in both academia and enterprises). We also believe that fields close to IoT should be studied as they might be of relevance when considering practices that can be adopted, *e.g.*, MICROSERVICES PATTERNS by Chris Richardson [Ric18], PATTERNS FOR FAULT TOLERANT SOFTWARE by Robert

Table 3.2: List of the IoT-related design patterns. While this list does not encompass all the patterns presented by the literature (*cf.* Table 3.1, p. 66), it showcases some patterns that we consider key for this work.

Concern	Pattern
Energy Supply	Mains-Powered Device [Rei+17c]
	Lifetime Energy-Limited Device [Rei+17c]
	Energy-Harvesting Device [Rei+17c]
	Period Energy-Limited Device [Rei+17c]
Operation Mode	Always-On Device [Rei+17c]
	Normally-Sleeping-Device [Rei+17c]
Processing	Rules Engine [Rei+16]
Sensing	Event-Based Sensing [Rei+17c]
	Schedule-Based Sensing [Rei+17c]
Bootstrapping	Factory Bootstrap [Rei+17a]
	Medium-based Bootstrap [Rei+17a]
	Remote Bootstrap [Rei+17a]
Registration	Device Registry [Rei+17a]
	Manual User-Driven Registration [Rei+17a]
	Automatic Client-Driven Registration [Rei+17a]
	Automatic Server-Driven Registration [Rei+17a]
Device Model	Device-Driven Model [Rei+17a]
	Predefined Device-Driven Model [Rei+17a]
	Server-Driven Model [Rei+17a]
Security	Trusted Communication Partner [Luk+17]
	Outbound-Only Connection [Luk+17]
	Permission Control [Luk+17]
	Personal Zone Hub [Luk+17]
	Whitelist [Luk+17]
	Blacklist [Luk+17]
	Remote Lock and Wipe [Rei+16]
Comms.	Delta Update [Rei+17b]
	Device Gateway [Rei+16]
	Device Shadow [Rei+16]
	Device Wakeup Trigger [Rei+16]
	Visible Light Communication [Rei+17b]
	Remote Device Management [Rei+17b]
Deploy	Edge Code Deployment [Qan+16]
Architecture	Fogxy [STB18]

Hanmer [Han07], and PATTERNS FOR SOFTWARE ORCHESTRATION ON THE CLOUD by Boldt *et al.* [SCF15].

Looking at the specific case of CI/CD, although one can leverage existing best practices (including patterns), IoT poses special needs that make the automation of these processes more expensive and risky; Jim Ruehlin [Rue18] identifies several *physical* needs and considerations beyond the *software-only* world needs: (1) Physical deployment of new sensors, (2) manual deployment of software (which implies the need for more resources and increased costs), (3) sensors or devices that are physically compromised, (4) weather conditions, (5) geographic considerations and constraints, and (6) devices and systems connectivity characteristics and issues.

These factors might hinder the feasibility on delivering new versions in a short regular schedule when compared to other software systems; which is unfortunate, since the criticality of fixing security vulnerabilities and bugs in IoT is increasingly paramount [LB16; Bei18].

Ultimately, the existence of enough IoT-specific design patterns and the discovery of relationships between them, has the potential to produce a *pattern language* that can be shared and improved among practitioners. Pattern languages would help IoT developers by providing them with a proper vocabulary, abstractions, and insights into the solutions other developers

have recurrently found.

3.2 Constructing IoT Systems

While the design activity of the software development life-cycle focuses on reaching suitable and high-level solutions for a given problem, the construction (also known as implementation) activity of the life-cycle focuses on actually building the systems (*i.e.*, *create an executable version of the software* [Som10]). The construction of software systems may involve writing code in one or more high– or low-level programming languages (at different abstraction levels), or tailoring and adapting generic, *off-the-shelf* systems to meet the specific requirements of a given project, organization, or scenario [Som10].

The heterogeneity and vastness of devices, platforms, and services that are part of IoT require development approaches that attend to this ecosystem particularities. Traditional programming — procedural computer programming — using code editors and integrated development solutions has been the go-to solution for developers and other technical individuals. However, as the heterogeneity of devices keeps increasing, as well as the number of application scenarios and environments, it became necessary to build abstractions of sensors, actuators, and whole devices as a way to reduce the complexity of developing and manage IoT systems. This need for abstractions leads to the birth of several, IoT-focused, model-driven development solutions as well as mashup-based development tools [PC13].

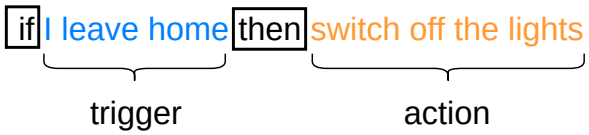


Figure 3.1: Example of a trigger-action rule for turning off the lights (action) whenever the user leaves the house (trigger).

With the growing number of non-technical users that use IoT systems, even such solutions did not suffice, since they required a certain level of technical knowledge that most end-users do not possess. This lead to the birth of several low-code solutions in the field that leverage visual programming concepts, natural language processing tools, and voice assistants as a way for the users to configure (viz. program) their own systems. One of the most common strategies used by these approaches is the use of *if-then* rules programming solutions (*e.g.*, IFTTT and Zapier [Hoy15])— also known as Trigger-Action Programming — where rules are defined as a sequence of trigger-action *flows*, as exemplified in Figure 3.1 (p. 68) [Rah+17].

An overview of the popularity of some tools used for IoT development is given on Figure 3.2 (p. 69), disregarding abstraction levels and application focus.

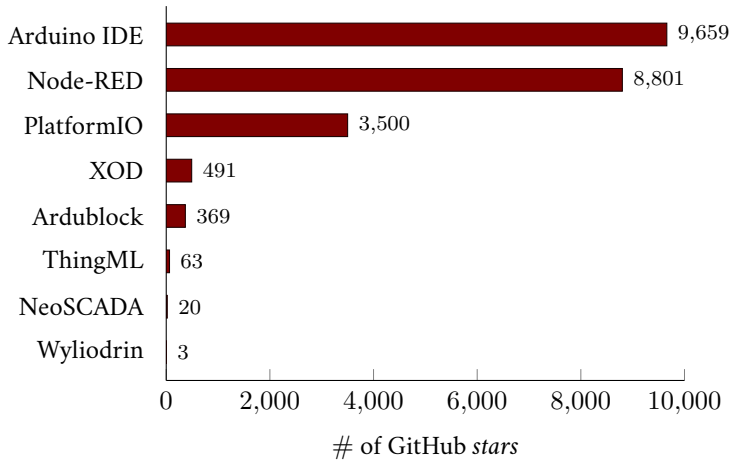


Figure 3.2: Number of *stars* in the GitHub repositories of the *open-source* IoT-focused development tools (circa 2020).

3.2.1 Traditional Development

Most IoT systems result from a combination of already existing technologies and systems (viz. Systems-of-Systems). As a consequence, there has been for a long time a set of development facilities (*e.g.*, programming languages, Integrated Development Environments, and other toolsets) that empower the construction of components for each tier of IoT systems (*cf.* Figure 2.7, p. 39) and the systems as a whole. Some of these tools are vendor-specific and/or board-specific.

From a programming languages perspective, an IoT Developer Survey carried by the Eclipse IoT Working Group, AGILE IoT, IEEE, and the Open Mobile Alliance circa 2019 enumerates the most common languages for each IoT systems tier, namely (by order of relevance): (1) C, C++, Java and JavaScript for the edge tier, (2) Java, Python, C++ and C for the fog tier and (3) Java, JavaScript, Python, and PHP for the cloud tier [Ecl19].

Regarding development environments, it is noticeable that the most popular ones (*e.g.*, Eclipse IDE) can be extended and adapted (*e.g.*, via plugins) to encompass the reality of IoT systems development. However, new tools are developed to deal with the particularities of each tier (*i.e.*, adapted to the heterogeneity, scale, and specific concerns).

We can identify some tools as reference development environments for IoT development, such as the following ones:

PlatformIO An IDE for the Internet-of-Things with a particular focus on the programming of edge-tier devices. It has support for more than 650 development boards, and comes with a built-in library manager, and has a *C/C++ Intelligent Code Completion and Smart Code Linter* suitable for embedded devices. It also has debugging features (*PIO Unified Debugger*) alongside with testing features (*PIO Unit Testing*) and support for integration in local — or cloud-based CI/CD pipelines [Pla19]. `PlatformIO` can also be used as a standalone Command Line tool or as a plugin in several code editors and IDE's (*e.g.*, Visual Studio Code and Atom).

Arduino IDE An IDE focusing on the programming of microcontrollers based on the Arduino framework (not to be confused with the Arduino development boards based on Atmel AVR), commonly found on the edge tier. Includes support for C and C++ programming languages and is pre-loaded with several examples and libraries [Ard19b].

NeoSCADA A project from the Eclipse IoT Industry Working Group targeting SCADA (Supervisory Control and Data Acquisition) systems development. This type of system is omnipresent in manufacturing and control operations and is now, typically, Internet-connected. Eclipse NeoSCADA (previously known as OpenSCADA [PCB18]) is not an out-of-the-box solution for SCADA systems but instead a set of development libraries, interface applications, mass configuration tools, and other facilities that allow the construction of these systems [IBH19].

There are also several micro-controller-specific development tools including, among others, (1) the ESPlorer, an Integrated Development Environment (IDE) for ESP8266-based boards using Lua or microPython languages [esp19], (2) Atmel Studio 7, a tool for developing and debugging AVR and SAM microcontroller applications using C/C++ or assembly code [Inc19a], (3) Pymakr, an IDE extension to code editors to aid the development of IoT edge devices that run microPython [Pyc21] and (4) Particle IDE, a development environment for Particle's IoT boards [Par21].

Additionally, several Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS) solutions — both cloud-based and on-premises — have appeared in the last few years as a response to the needs of IoT systems development, in a fashion similar to that of other kinds of software systems. The survey by Mineraud *et al.* identifies 23 different solutions that fit this category [Min+16], such as the EvryThng, ThingWorx, and SensorCloud. The same survey points out several gaps in the existing solutions, mostly resulting from the nonexistent standardization and lack of cross-platform support [Min+16] (*i.e.*, fragmented ecosystem, no interoperability between different IoT solutions [Brö+17]). These solutions generally provide a mostly complete programming environment with an out-of-the-box set of Web-based Internet-of-Things, Software Development Kits (SDKs), libraries for specific hardware boards (*i.e.*, devices), and dashboards for device and data management.

Most of these solutions, ranging from the PaaS to the board-agnostic IDEs are somewhat limited in scope, both in terms of the supported devices and/or frameworks and in terms of system tiers (*e.g.*, tools focused on developing software for micro-controllers disregard the developments towards other tiers completely).

3.2.2 Model-driven Development

Several works have studied the application of Model-Driven approaches like Domain-Specific Languages (DSL) and model transformations for providing IoT-focused low-code programming solutions [Ihi+20]. These try to improve the interaction with IoT systems and make the production of platform-specific code more efficient by abstracting the platforms and allowing users to design systems at a high-level, most of them focusing on visual abstractions.

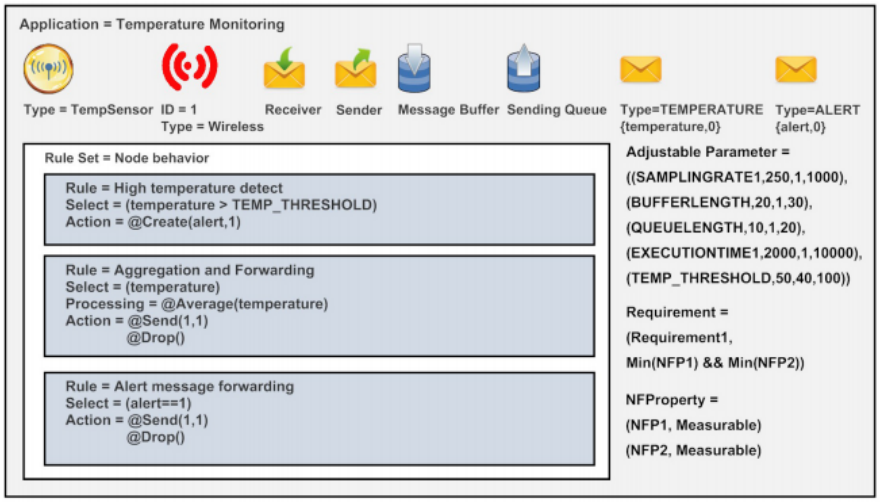


Figure 3.3: FRASAD Platform-Independent Model visual editor.

Some of the most known and comprehensive works that focus on model-driven development for IoT include the following:

FRASAD FRAmework for Sensor Application Development is a framework for model-driven IoT system development, based on a *node-centric, multi-layered software architecture* to hide *low-level details and to raise the level of abstraction* of developing IoT systems. Using model transformations, users can visually define the behaviors of the IoT system, which is then converted into runnable code. The approach uses a DSL and a rule-based model to describe the final applications and provides a graphical interface for users to create those rules, according to the DSL. Figure 3.3 (p. 71) shows an example of a Platform-Independent Model in the visual interface provided by FRASAD. The work authors' evaluated the tool by testing it with novice and intermediate programmers, through the completion of tasks with various difficulties using the tool developed and two other tools (TinyOS and ContikiOS), and also by testing the model transformations to see whether the models generated from the visual programming solution were complete and correct enough to work correctly. The test results with programmers showed that programmers were more successful using FRASAD than using the other two tools, regardless of their experience. The results of testing the model transformations showed that the tool generates complete and correct code in 40% of cases; however, in 41% of cases, the code needed minor fixes to work, and in 18% of cases, the code was not complete nor correct. The authors conclude that the results provide evidence that their tool can be a solution for IoT development. Nevertheless, the solution's focus is on the programming of sensor nodes (constrained devices), not on end-user development and configuration of holistic IoT solutions.

Eterovic *et al.* proposed a visual DSL for IoT, inspired in UML, aiming to provide non-technical end-users with a tool simple enough for them to work on their IoT systems and sufficient complexity for more experienced users to do more complex tasks. However, the authors state that when used in an IoT context, advanced UML expressions need to be used, making it too complex for non-technical users. The authors propose a DSL based

on UML, but with some changes to simplify the advanced expressions. To validate the tool, the authors experimented with different test groups, some with UML experience and others with no experience, but no group had previous IoT experience. The results reveal that everyone could complete the experiment with success, and most participants showed a preference for the DSL. This points out that the developed approach is both simple in terms of interaction and complete in functionality. However, the authors say that the measurements made do not provide evidence about usability problems, only success rate and that the experiment's tasks were straightforward. Therefore, they believe that more experiments should be performed to measure other metrics and using more complex systems and tasks [Ete+15].

Einarsson *et al.* also applied model transformations in IoT, developing a Domain-Specific Modeling Language (DSML) that allows users to specify rules that are then used to generate code for Alexa and SmartThings. The solution uses Text Template Transformation Toolkit (T4)[2] to handle the transformations into platform-specific code. The authors tested the system in a mock cloud service, which allowed them to conclude that it can handle all Alexa's skills and some of SmarthThings', being only limited by the fact that they were only generating code to work with mock cloud-enabled devices [Ein+17].

Sutherland *et al.* developed a text-based DSL for programming social interactions for robots. The authors leverage off-the-shelf modules to build a Python-inspired DSL that could be integrated with the robot. By doing so, they maintain the existing tools' capabilities and build a more straightforward interface for non-technical users to also work on programming robots. With their approach, users can create scripts for robots' social interactions parsed and converted into an Abstract Syntax Tree (AST) to be validated and then consumed by an execution engine that runs on the robots, which results in the robots performing the scripted action. The authors concluded that the DSL is complete enough to be used for various robots with slight changes and also short enough to be quickly understood and used as a cheat sheet. However, the tool was not tested by non-technical users, and the authors mention some potential issues. For example, even though the DSL is short and straightforward, it is similar to the Python programming language, and non-programming users may have difficulties understanding some concepts or code structure. The authors also mention that improving debugging features may help non-technical users since the debugging so far is done by reading logs resulting from the execution [SM19].

UML4IoT A model-driven UML-based approach for IoT system development. It defines a new UML profile, UML4IoT profile, which automates the generation of an IoT-compliant layer, IoTwrapper, that allows the full integration of different IoT components in an IoT system. This wrapper is tailored for IoT manufacturing environments. However, the *UML4IoT profile contains basic key constructs independent of the application domain* so that it can be used in other application domains [TC16].

[2] T4 is a template-based text generation framework included with Visual Studio that can generate text artifacts such as code.

MDE4IoT An approach for modeling things as well as their Emergent Configurations[3] with a DSML. This approach exploits the use of high-level abstraction and separation of concerns to manage heterogeneity and complexity of things, enabling collaborative development, and enforcing reusability of design artifacts. It also provides automation mechanisms by providing model manipulation features enabling intelligence as runtime self-adaptation [CS17].

DSL-4-IoT Editor-Designer A visual domain-specific modeling language (VDSML) approach which allow users to configure their systems by connecting high-level representations of sensing and actuating devices using a built-in *Rules editor*. The visual specifications are used to generate OpenHAB configurations, which work as the target runtime of the system. They validated the feasibility of their solutions using a testbed with 15 heterogeneous devices, asserting that it performs as expected [Sal+15].

SmartHomeML A Domain-Specific Modeling Language (DSML) that allows users to specify new skills targeting Alexa and SmartThings. The solution uses a template-based transformation to handle the generation of platform-specific artifacts. The authors tested the system using mock cloud devices, concluding it covered all existing Alexa's skills and some of SmarthThings' [Ein+17].

Midgar A DSL, a visual editor, and a runtime (named Midgar) to specify and run IoT applications. The authors performed a two-staged validation with 21 participants. In the first stage, they measured the time participants took to interconnect one temperature sensor to two smartphones (≈162s). In the second stage, they performed a Likert-scale based survey to assess the subjective perception of the participants regarding the tool, concluding that respondents generally liked it. The main negative feedback was the limited development characteristics that the platform presented [Gon+14].

WoX: Web of Topics A conceptual model for IoT development based on the notion of *topics*, building block of the event-driven architecture followed by several IoT systems. In WoX, a *WoX Topic* is defined by two parts, namely: (1) *a discrete semantic feature of interest* (*e.g.*, temperature, humidity, air pressure), and (2) a Uniform Resource Identifier (URI) based location. A thing (IoT entity) role within a *Topic* is specified by its technological and collaborative dimensions, *i.e.*, *Topic-Role* [Mai+15].

Fluidware The Fluidware conceptual idea is to abstract the different system parts (including sensors, actuators, and other system components) as *sources, digesters, and target of distributed flows of contextualized events*, maintaining information about the data produced and manipulated over time. In this solution, developing new services and applications implies the definition of *funnel processes* to channel, elaborate, and redirect such *flows* in a fully distributed way, *i.e.*, declaration of how these *flows* consume and produce events over space and time [Zam+19; For+19].

[3] Emergent Configurations is defined as the *set of things with their functionalities and services that connect and cooperate temporarily to achieve a goal.* [ASD17]

ThingML The Internet of Things Modelling Language is an approach for IoT systems' development that encompasses a modeling language — DSML — (including state-charts, an imperative platform-independent action language and IoT-specific constructs), a set of tools (language editor, model transformations to diagrams, code generation framework), and a methodology to develop IoT systems and to extend ThingML itself (*cf.* Figure 3.4, p. 75). This solution is one of the most comprehensive model-driven approaches and has a text-based description language. The primary focus is on the development of highly-distributed and heterogeneous systems by abstracting from the heterogeneous platforms and devices to model the desired IoT system's architecture; however, the solution only provides a design-time specification of the structure (components) and behavior (state machines), and it is not executable per se. Further, their focus is not on improving the end-user ability to develop IoT systems but also on technical users. More recent works attempt to tackle the issue of IoT communications and behavioral modeling replacing the use of state machines with automatic generation by using machine learning techniques [MRG18].

Durmaz *et al.* This work presents a metamodel (*i.e.*, modeling language syntax) and a supporting graphical modeling environment tailored for the specifications of Contiki IoT platform. As a result, this work focus on abstracting the event-driven mechanism and protothread [Dun+06] architecture of Contiki [Dur+17].

Chen *et al.* The work presents a runtime model (*models@run.time*) based approach to IoT application development. It focuses on two facets, namely: (1) *manageability of sensor devices is abstracted as runtime models that are automatically connected with the corresponding systems* and (2) *the customized model is constructed according to a personalized application scenario and the synchronization between the customized model and sensor device runtime models is ensured through model transformation*. Such an approach allows all the *application logic to be carried out by executing programs on this customized model* [Che+15].

Some other works propose DSLs to tackle the shortcomings of the current solutions [SN15; Gom+17; CC20], but only present ideas or minimal proofs-of-concept, without validating their feasibility nor empirical validation with users. A more complete list of other relevant works can be found in [SAA21]. Additionally, other less comprehensive approaches, have been developed as an answer to specific issues, such as the modeling of security and privacy concerns [Nei+14; Yan+12], and critical-mission systems [Cic+17].

3.2.3 Visual Programming

In the context of IoT systems development, several visual programming languages and ecosystems have been created with the main objective of easing the development process, leveraging different visual metaphors, and programming paradigms. These languages can be categorized in being (1) purely visual, (2) hybrid text-visual, (3) programming-by-example, (4) constraint-oriented, and (5) form-based languages [BD04]. Some of the most common examples are:

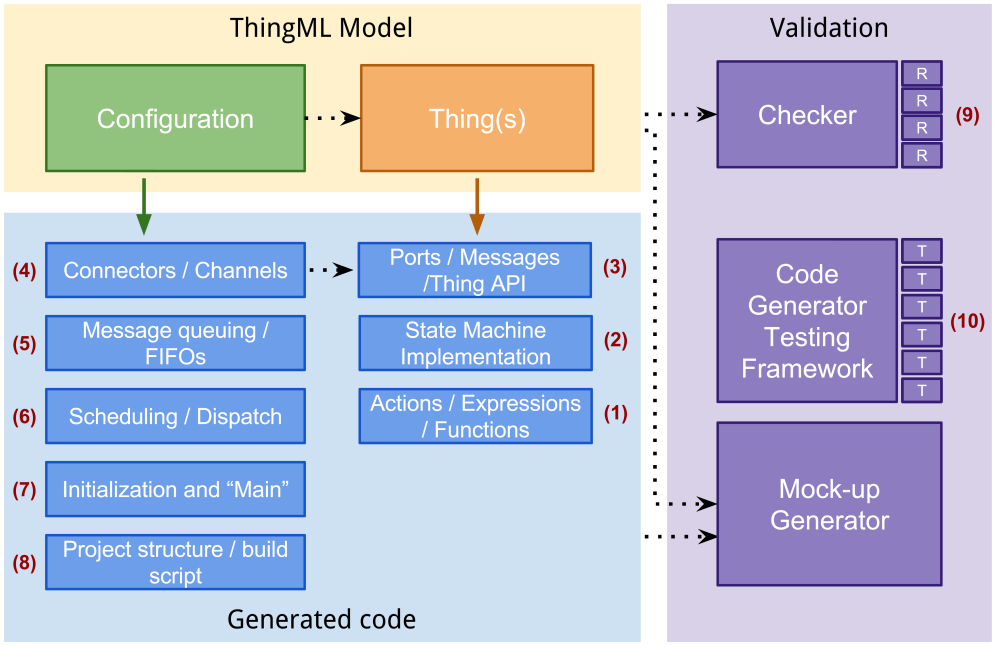


Figure 3.4: The ThingML code generation framework. The 10 variation points of the framework are identified and separated in two groups: the ones responsible for the generation of code for *things* and the ones corresponding to the generation of code for the *applications* (Configuration) [Har+16].

ArduBlock ArduBlock is purely visual programming, based on the Google/MIT Media Lab's Blockly [Fra+13], focused on physical computing devices based on Arduino [Ard19a]. Several similar solutions are available such as Snap4Arduino, Microblocks, and BlocklyDuino [XP18].

S4A S4A is a modified version of the original Scratch educational programming language [Res+09] that targets Arduino-based hardware. It provides purely visual programming, based on the Google/MIT Media Lab's Blockly [Fra+13], keeps the Scratch original focus on educational purposes only, and provides an extra feature of recording and playback the resulting programming *output* [CIT19; Ray17].

XOD XOD is a purely visual microcontroller programming platform. It uses a visual language to program the devices and then generates native code for the target platform. In the language, a node is a block that represents either a physical device (*e.g.*, sensors, motors, or a relay) or an operation (*e.g.*, addition, comparison, or a text concatenation). Each node has one or more typed inputs that accept values to be processed and outputs that return results. Creating a link from an output to an input builds a path for data, allowing one node to feed values into another. The development environment also allows the simulation of the programmed systems before deploying to real hardware [Inc19b].

GraspIO Graphical Smart Program for Inputs and Outputs is purely visual programming, also inspired on the Google/MIT Media Lab's Blockly [Fra+13], targeting the *Cloudio* hardware shield for Raspberry Pi SBCs. It is most suitable for educational purposes and offers the ability to build simple IoT and Robotics systems quickly [Ray17].

Wyliodrin Wyliodrin is an online IDE for Linux-based embedded systems (Raspberry Pi, Intel Galileo) programming. The low-level GPIO-connected components are abstracted by the use of the *Libwyliodrin*, which offers an Arduino-like API, and can be programmed by the use of a hybrid text-visual programming language, in a *drag-and-drop* fashion. It also provides a set of dashboards to visualize the data being collected, and a mechanism to ease the communication between different devices [MR16].

Zenodys Zenodys is a hybrid text-visual programming environment, based on a *drag-and-drop* interface, that can construct both the logic and the user interfaces (UI) of IoT applications. It has built-in debugger features as well as text-based programming mechanisms, allowing fine-tuning of the solutions. Solutions designed in Zenodys can be deployed to both Linux and Windows environments [BV19].

Noodl Noodl is a hybrid text-visual programming environment that allows the creation of interfaces, and both logic and data flow. Although it does not focus on IoT, it also covers IoT system programming. It leverages visual notations to define the behavior of the system, namely, *nodes* are entities that perform computation, *connections* interlink the different *nodes*, and the resulting logic flows are organized into *hierarchies* [AB19; Ray17].

DG Solution Builder DG Solution Builder is a purely visual IoT programming language, and environment, powered by a *drag-and-drop* interface. It also provides a dashboard feature capable of displaying the IoT systems data. It is tailored to be used with the Distributed Services Architecture (DSA) IoT middleware platform that abstracts the *intercommunication, logic, and applications at every layer of the Internet of Things infrastructure* [DGL19; Ray17].

AT&T Flow Designer AT&T IoT Platform Flow IDE is a cloud development environment for IoT systems. The visual language allows the creation of prototypes of IoT solutions, giving the ability to iterate and improve through multiple versions, then deploy the final solution. It gives the developer a set of pre-configured nodes that allow easy access to multiple data sources, cloud services, device profiles, and communication methods. Inspired by the Node-RED programming environment, it shares the same hybrid visual-text approach to development [Pro19; Ray17].

Epidosite A programming-by-demonstration system tailored for smartphones leveraging it as a hub for IoT systems automation. Users can express the wanted system logic by demonstrating the desired behaviors through directly manipulating the corresponding smartphone app for each IoT device. This programming environment also supports the creation of a highly context-aware application by taking into consideration the *smartphone app usage context and external web services as triggers and data for automation's* [Li+17].

IFTTT IFTTT is a web and mobile application that leverages the use of a visual programming language to develop *if-this-do-that* rules in a form-based programming fashion [Kub16]. The language provides integration with several third parties. Despite not

being focused on IoT solutions, it provides integration with several IoT products on the market, allowing the programming of their behavior [IFT19; CDR17]. There are some other similar solutions in the market, such as Microsoft Flow and Zapier [BBS18].

Blynk Blynk is a mobile application that allows the control of Arduino and Raspberry Pi devices using a digital dashboard in a program-by-example fashion. It allows the *drag-and-drop* of widgets and then configures them using a form-like approach [Inc19c; BMS16].

While there are other solutions for developing IoT systems employing a visual programming strategy (Node-RED, WoTKit, glue.things, IoT-MAP, IoTMaaS, and the solution proposed by Yao *et al.* [YSD15]), these tools also employ mashup strategies to build the system and are analyzed in detail in Section 3.2.4.

These visual programming languages and environments are mostly tailored for IoT systems, with some targeting one specific tier (*cf.* Figure 2.7, p. 39) while others are capable of connecting different IoT devices, third-parties, and subsystems across tiers. Regarding the last, these are highly-inter-winded with their visual languages runtime and can be deployed on Linux-based systems, usually on the fog and cloud IoT tiers. They are also web-centric and encourage rapid development [BL12a].

Focusing on the tools that leverage a TAP-like fashion (*e.g.*, IFTTT), there are several works that highlight the issues that users have when configuring and understanding the trigger-action programs built using such tools [HC15; Ghi+17]. Huang and Cakmak in their work identify that ambiguities between *trigger types* (states and events) and *action types* (instantaneous, extended, and sustained actions), lead users to misconstrue and misinterpret their rules (the authors state that “people create different programs given the same prompt and are still in disagreement in their interpretations after having created programs themselves") [HC15]. Ghiani *et al.* mention similar issues in their work and emphasize that different individuals understand the same *concept* or *metaphor* differently, which also increases the proneness to errors and the difficulty to understand the programmed rules [Ghi+17].

3.2.4 Mashup-based Development

Mashup tools enable rapid prototyping by combining the use of rule-based mechanisms, graphical composing interfaces, and templates [PC15]. Some of the most relevant examples of mashup tools specific to IoT systems development are:

Node-RED Node-RED is one of the most widespread visual programming solution targeting IoT systems. It presents a flow-based notion for wiring together hardware devices, APIs, and third-party services, leveraging a visual programming language [Szy+17; Ope19b]. IBM originally developed it, and its runtime is built on JavaScript. Node-RED provides a programming *canvas* through which users can create, edit and delete system rules and connections in an interface that displays rules and connections as a *flow* of information, events, or action by *drag-n-drop* building blocks (*nodes* and *links*) which are made available through an extensive and extensible *node* palette, as exemplified in Figure 3.5 (p. 78). Several entities have merged their solutions with Node-RED providing

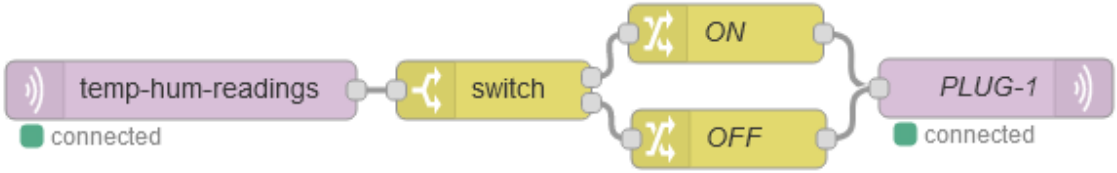


Figure 3.5: Example of Node-RED *flow*, where the status of an electric plug (PLUG-1) changes (ON/OFF) accordingly to the current temperature value (SWITCH), provided by the temperature and humidity sensor (TEMP-HUM-READINGS).

new services, such as the Sense Tecnic IoT Platform with provides FRED, a cloud-based Node-RED service [BL16].

WComp It is a three-part middleware designed for ubiquitous systems that provides a (1) foundational software infrastructure, (2) a service composition architecture, and (3) a compositional adaptation mechanism designed with dynamicity and heterogeneity of the underlying system in mind. Systems are built by composition of Web Services (which can correspond to devices, cloud services, or any other Web Service compliant entity). Their approach also introduces Aspect of Assembly (AA) as a way to define reactive behaviors to respond to changes in the system (*i.e.*, self-adaptation) [Tig+09; Fer+12].

WoTKit It is a lightweight Java web application based on a message broker for programming IoT systems that allow the integration, visualization, processing of data from different devices and subsystems. The environment uses a hybrid text-visual language that allows the connection between different *modules* using *pipes*. Visualizations are based on the concept of *widgets* that are displayed in a dashboard, allowing rapid visualization of sensor data [BL12a; BL12b].

glue.things A Node-RED-inspired mashup tool focused on the composition of Web services data streams and Web-enabled IoT devices (*i.e.*, Web-of-Things). A special focus is given on the delivery and management aspects of device data streams, consumer applications, and their integration within the ecosystem [Kle+14].

IoT-MAP Is a *thing* service composition platform that dynamically discovers devices, deploys drivers, and provides them in the uniform interface to IoT-App. The IoT-App decouples the development of mobile applications from the heterogeneous devices' specificity's by providing an *IoT-App* API that abstracts the functionalities of various things (*i.e.*, abstracted service objects) [Heo+15].

IoTMaaS IoT Mashup as a Service is a mashup platform that provides a (1) thing model, (2) a software model, and (3) a computation resource model. During the mashup process, these three components can be tailored to the specificity's of the system under development (*e.g.*, select *things*, processing software and resource allocation, notification

preferences), while the platform assures interoperability between devices and services (following the service-oriented architecture principles) [IKK13].

Yao *et al.* This work presents a web-based solution for *connect, monitor, control, mashup, and visualize* devices in an IoT system. Their solution provides a layered framework for managing the data produced by the devices and to share it, includes a rule-based system to program the system logic in a context-aware fashion, and exploits the concept of avatars (*i.e.*, virtual representations of the physical things) [YSD15].

Although more popular (*cf.* Figure 3.2, p. 69) when comparing with model-based solutions, most of the mashup tools for IoT disregard the concerns about user interface definition and customization, focusing on the data and process mediation. Further, some mashup tools provide extra features such as simulation mechanisms and support for interoperability with other platforms.

3.2.5 Development of Decentralized IoT Applications

Several of the aforementioned solutions, specifically the visual programming and mashup-based ones (*i.e.*, low-code), provide a certain level of orchestration features — they connect together services and devices, allowing their configuration, coordination, and management. But, in most cases, these solutions go further than that and also perform some level of computation, from data parsing to conditional statements, in a centralized fashion, thus becoming the Single point of failure of the system.

Only a small fraction of those aims to offer a way for distributing computing tasks among devices and other computational resources while dealing with the challenges posed by the nature of these resources, especially the highly-dynamic topology of these networks. Thus, as a way to leverage the existing computational power in edge devices, several proposals have been drafted, mostly resulting from scientific research — thus being more proofs-of-concept and not full-fledge ready-to-use solutions. A non-extensive list of the ones found in the literature is summarized in the following paragraphs.

WoTFlow [BL14], DDF [Noo+19], and subsequent works [Gia+15; GLL18] A set of extensions to Node-RED focusing on Smart Cities use case. Their goal is to make it more suitable for developing fog-based applications that are context-dependent on edge devices where they operate. DDF starts by implementing D-NR (Distributed Node-RED), which contains processes that can run across devices in local networks and servers in the Cloud. The application, called *flow*, is built with a visual programming environment, running in a development server. All the other devices running D-NR subscribe to an MQTT topic that contains the status of the *flow*. When a *flow* is deployed, all devices running D-NR are notified and subsequently analyze the given *flow*. Based on a set of constraints, they decide which nodes they may need to deploy locally and which sub-flow (parts of a *flow*) must be shared with other devices. Each device has characteristics, from its computational resources, such as bandwidth and available storage, to its location. The developer can insert constraints into the *flow* by specifying which device a sub-flow

must be deployed in or the computational resources needed. Further, each device must be inserted manually into the system by a technician.

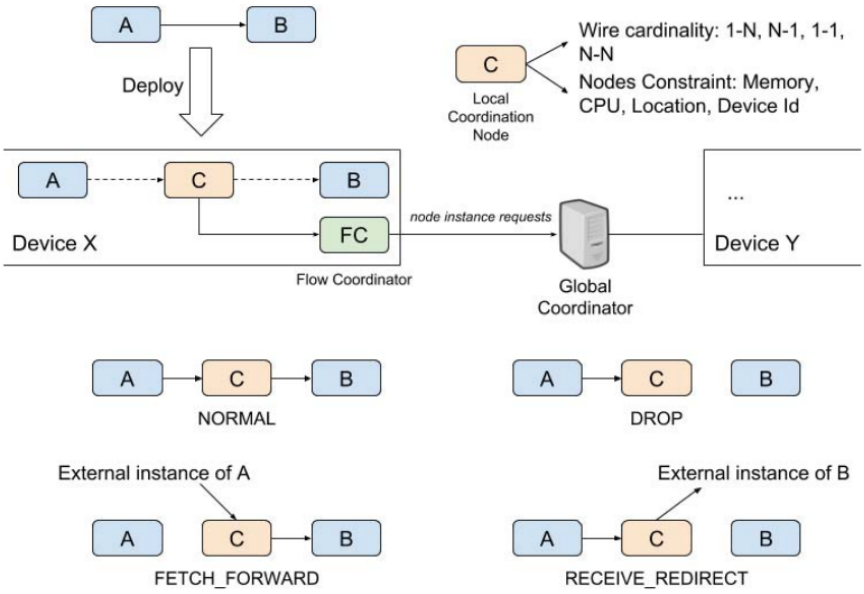


Figure 3.6: Coordination between nodes in Distributed Node-RED (D-NR) [Gia+15].

Subsequent work focused on support for the Smart Cities domain, including the deployment of multiple instances of devices running the same sub-*flow* and the support for more complex deployment constraints of the application *flow* [Noo+19]. The developer can specify requirements for each node on device identification, computing resources needed (CPU and memory), and physical location. In addition to these improvements, the coordination between nodes in the fog was tackled by introducing a coordinator node. This node is responsible for synchronizing the device's context with the one given by the centralized coordinator. In more recent versions of the work support for CPSCN (Cyber-Physical Social Computing and Networking) was added, thus making it possible to develop large-scale CPSCN applications [GLL18]. Additionally, to make this possible, the contextual data and application data were separated so that the application data is only used for computation activities. The contextual data is used to coordinate the communication between those activities. In Figure 3.6 it is possible to see the four possible states of a coordinator node: (1) `NORMAL`, where the node passes the data to its output, (2) `DROP`, in which the node does not pass the data to other node and instead drops it, (3) `FETCH_FORWARD`, where the node gets the input from an external instance of its supposed input and (4) `RECEIVE_REDIRECT` in which the node sends the data to an external instance of its output node.

Szydlo *et al.* [Sen+19] Work focused on the transformation and decomposition of data

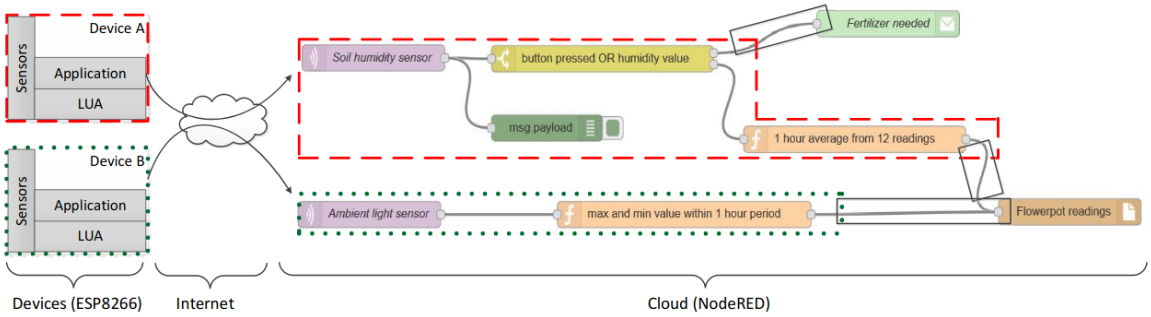


Figure 3.7: Partition and assignment of parts of a Node-RED *flow* [Sen+19].

flow. Parts of the *flow* can be translated into executable parts, such as Lua. Their contribution includes data flow transformation concepts, a new portable runtime environment (uFlow) targeting resource-constrained embedded devices, and its integration with Node-RED (*cf.* Figure 3.8, p. 82). Their solution transforms a given data flow by allowing the developer to choose the computing operations run on the devices. These operations are implemented using uFlow. The communication between the devices requires a Cloud layer, without support for peer-to-device communication. The results are promising, showing a decrease in the number of measurements made by the sensors.

However, there is room for improvement with relation to the automatic decomposition and partitioning of the initial *flow*, and detecting current conditions in deciding when to move computations between fog and cloud. Later, the authors proposed FogFlow [Sen+19], which enables the decomposition into heterogeneous IoT environments according to a chosen decomposition schema. To achieve a certain level of decentralization and heterogeneity, they abstract the application definition from its architecture and rely on graph representations to provide an unambiguous, well-defined computation model. The application definition is infrastructure-independent and only contains data processing logic, and its execution should be possible on different sets of devices with different capabilities. Several algorithms for *flow* decomposition are mentioned [NAA+18; Gup+17], but none were explored/provided results.

FogFlow by Cheng *et al.* [Che+17; Che+18b] Proposes a standards-based programming model for Fog Computing and scalable context management. The authors start by extending the dataflow programming model with hints to facilitate the development of fog applications. The scalable context management introduces a distributed approach, which allows overcoming the limits in a centralized context, achieving considerable improvements in performance, namely in terms of throughput, response time, and scalability.

The FogFlow framework focuses on a Smart City Platform use case, separated into three modules: (1) *Service Management*, typically cloud-hosted, (2) *Data Processing*, present in cloud and edge devices, and (3) *Context Management*, which is separated in a device discovery unit hosted in the Cloud and IoT brokers scattered across edge and cloud tiers (*cf.* Figure 3.8, p. 82). The approach was later improved to empower infrastructure providers

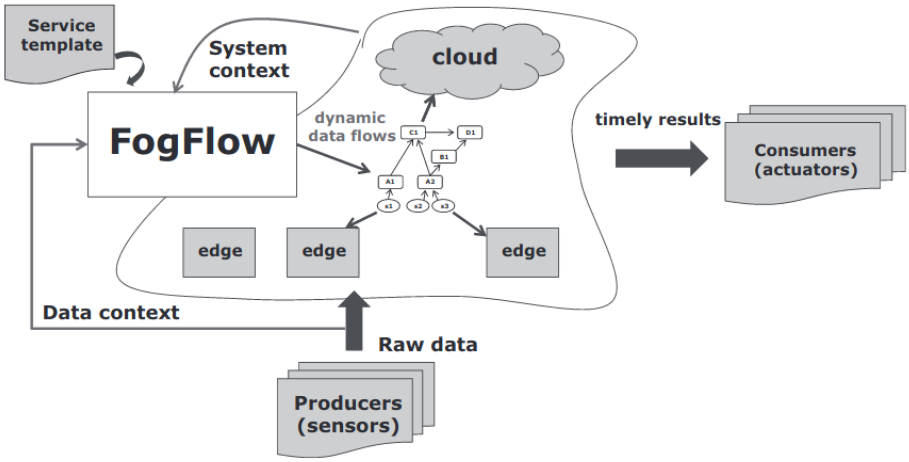


Figure 3.8: FogFlow high level architectural model [Che+18b].

with an environment that allows them to streamline the construction of decentralized IoT systems, with increased stability and scalability. Dynamic data representing the IoT system *flows* are orchestrated between sensors (sources) and actuators (sinks). An application is first designed using the FogFlow Task Designer (a hybrid text and visual environment), which outputs an abstraction called *Service Template*. This abstraction contains details about the resources needed for each part of the system. Once the *Service Template* is defined and inserted, the framework determines how to instantiate it using the context data available. Each task is associated with an operator, and its assignment is based on (1) how many resources are available on each edge node, (2) the location of data sources, and (3) the prediction of workload. Edge nodes are autonomous since they can make their own decisions based on their local context without relying on the cloud. As a downside, the dependency in Docker completely discards constrained devices, requiring edge devices to run a full-fledge operating system.

DDFlow [Noo+19] Presents another distributed approach by extending Node-RED with a system runtime that supports dynamic scaling and adaption of application deployments. The distributed system coordinator maintains the state and assigns tasks to available devices, minimizing end-to-end latency. Dataflow notions of *node* and *wire* are expanded, with a *node* in DDFlow representing an instantiation of a task deployed in a device, receiving inputs and generating outputs. *Nodes* can be constrained in their assignment by optional parameters, *Device*, and *Region*, inserted by the developer. A *wire* connects two or more nodes and can have three types: *Stream* (one-to-one), *Broadcast* (one-to-many), and *Unite* (many-to-one).

In a DDFlow system, each device has a set of capabilities and a list of services that correspond to an implementation of a *Node* (*cf.* Figure 3.9, p. 83). The devices communicate this information through their Device Manager or a proxy if it is a constrained device. The coordinator is a web server responsible for managing the DDFlow applications. It is composed of: (1) a visual programming environment where DDFlow application are built, (2) a Deployment Manager that communicates with the Device Managers of the

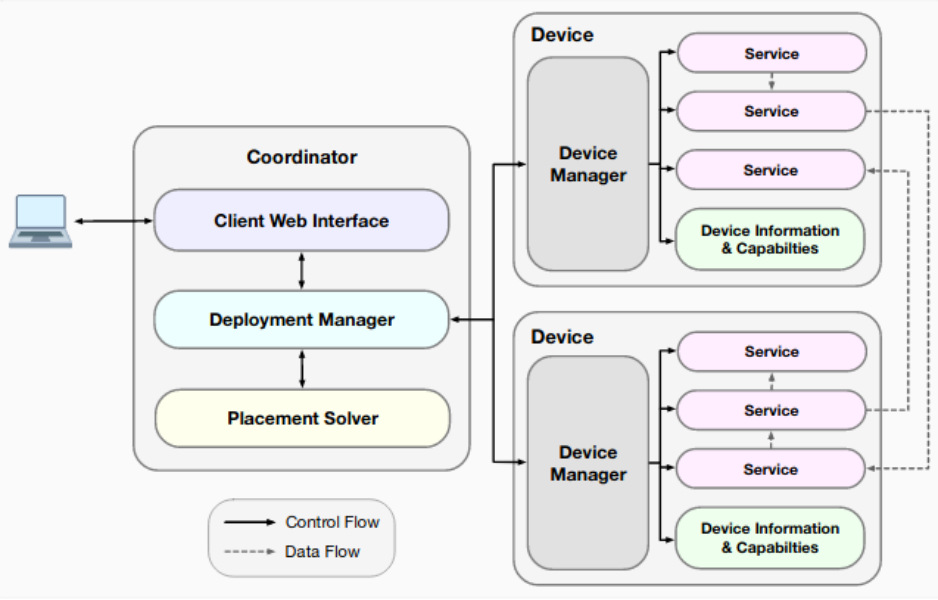


Figure 3.9: DDFlow architectural components, representing both the coordinator components and the device's components [Noo+19].

devices, and (3) a Placement Solver, responsible for decomposing and assigning tasks to the available devices. When an application is deployed, a network topology graph and a task graph are constructed based on the real-time information retrieved from the devices. The coordinator proceeds with mapping tasks to devices by minimizing the task graph's end-to-end latency of the longest path. Dynamic adaptation is supported by monitoring the system; if changes in the network are detected, such as the failure or disconnection of a device, adjustments in the assignment of tasks are made. The coordinator can also be replicated into many devices to improve the system's reliability and fault-tolerance. They also showcase DDFlow recovering from network degradation or device overload, whereas in a centralized system this would likely cause its (total) failure.

Most of these approaches are only POC and not widely adopted, even if most of them are based on Node-RED which is one of the most common IoT-focused development environment and runtime. Only a few of the approaches — FogFlow and uFlow — do in fact leverage the available resources in the edge tier of the system.

3.2.6 End-user Development

There is a considerable amount of literature that focuses on enhancing the interactions between users and smart spaces, taking advantage of several approaches, such as introducing artificial intelligence-based solutions (personal assistants) and exploring Natural Language Processing (NLP). These solutions try to improve the end-users experience on configuring their own IoT systems, by providing easy-to-use mechanisms to program their systems that require little to no technical skills. Some of these solutions that focus on providing a graphical interface for the

user to interact with the system have already being presented in § 3.2.3 (p. 74), thus in the following paragraphs, the focus will be on the approaches that leverage other ways of interaction with the IoT system.

Austerjost *et al.* [Aus+18] recognized the usefulness of voice assistants in home automation and developed a system that targets laboratories. Possible applications reported in their paper include a stepwise reading of standard operating procedures and recipes, recitation of chemical substance or reaction parameters to control, and readout of laboratory devices and sensors. As with the other works presented, their voice user interface only allows controlling devices and reading out specific device data.

Some of the identified approaches that either use NLP or conversational assistants in the context of IoT systems are the following:

Siri, Alexa, Cortana, and Google Assistant There exist a plethora of conversational assistants in the market (see [Mit18] and [LQG18] for a comparison of these tools) which are capable of answering natural language questions. Recently, these assistants have gained the ability to interact with IoT devices, with Ammari *et al.* identifying IoT as the third most common use case of voice assistants [Amm+19]. However, these are general conversational assistants and do not focus on providing IoT-focused features. Further, they are limited in the number of device manufactures and ecosystems they support (relying in the open-source community for supporting other devices and ecosystems).

Kodali *et al.* An approach for a home automation system to *"increase the comfort and quality of life"*, by developing an Android app that can control and monitor home appliances using MQTT, Node-RED, IFTTT, Mongoose OS, and Google Assistant. Their limitations lie in that the *flows* must have been created first in Node-RED, and the conversational interface is used to trigger them, ignoring all the management and configuration activities [Kis+19].

Braines *et al.* An approach based on Controlled Natural Language (CNL) — natural language using only a restricted set of grammar rules and vocabulary — to control a smart home. Their solution supports (1) *direct question/answer exchanges*, (2) *questions that require a rationale as response* such as *"Why is the room cold?"* and (3) *explicit requests to change a particular state*. The most novel part of their solution is in trying to answer *questions that require a rational response*; however, they depend on a pre-defined smart home model that maps all the possible causes to effects [Bra+17].

Rajalakshmi *et al.* A solution using both Node-RED and Alexa to interact with IoT systems, which they claim to simplify the interaction between users and IoT systems, but also manage complex systems. The system proposed by the authors uses Node-RED to create rules and link devices with each other, allowing the user to create complex rules while at the same time taking advantage of Alexa, providing a simple way for users to control their smart devices. This system provides simple interaction with smart homes through Alexa, and the complexity needed for some use cases through Node-RED. However, there is no link between the voice control and the visual programming solution,

which means they cannot create these rules using voice commands (being Alexa, the only way to trigger the Node-RED *flows*) [RS17].

Kang *et al.* In their work explores the use of multi-modal interaction within IoT systems — combining voice and gesture interactions — as a way of addressing the scalability and expressiveness supported by existing IoT-vendors mobile applications and voice assistants. Although most of the participants who took part in the study responded positively to many interaction techniques, one of the identified pitfalls was the lack of robustness of the voice assistant that failed to understand the user commands [Kan+19].

Several other works [Bhe+21; Kim20; KSP20] combine the use of voice assistants with IFTTT, using the latter to define the system rules. While the primary control mechanism over the IoT system is voice-based, it is mostly used to trigger the rules specified in the IFTTT platform, thus depending and being limited by the rules' definition capabilites and syntax of IFTTT.

An empirical study by Ammari *et al.* [Amm+19] identifies IoT as one of the most common uses of voices assistants. In their study, users identified as the main drawbacks of the use of voice assistants (1) the lack of spatial and temporal contextualization and (2) lack of support for dynamic instructions (macros). Concerning (1), such awareness would allow the assistant to know where the user is physically at any point in time, thus acting in accordance (*e.g.*, turn on the lights in the room where the user is located without the need to provide further context). Regarding point (2), the users point to the need of creating macros to simplify their interactions with the devices (*e.g.*, supporting rules such as *when leaving home, turn off all the lights, close the garage door and reduce the thermostat temperature*).

Priest *et al.* [PDM19] elaborated a list of the commands that Alexa knows how to handle, classifying them into some categories, such as smart home, search, entertainment, among others. From this list, it is noticeable that Alexa cannot perform complex actions on smart home devices; instead, it can only perform direct actions like turning lights on or off, changing the lights colors, and adjusting the climate system's temperature. Similarly, Martin *et al.* [Mar+19] gathered Google Assistant's commands in a list, categorized similarly to the one about Alexa, being the results similar to the ones about Alexa. Regarding Apple's assistant, Siri, there is also a list of the commands it can perform, created by Langley [Lan20b], in which it is possible to see the similarities to the other two assistants mentioned. These assistants present a way to create routines (recurring rules); however, it has to be done manually using the mobile application associated with each assistant, instead of voice commands. To sum up, smart assistants' current state does not allow users to create rules for their smart home devices using voice commands.

Clark *et al.* [CDN16], based on previous work that analyzed natural language for patterns in smart home programming, stated that the current smart assistants are too simple and work just as a voice interface for applications that control smart devices. The authors surveyed possible end-users of IoT systems for smart home applications they would want to see implemented, based on a list of smart devices and their capabilities. The survey was split into two, one where the smart home controls were handled by the devices controllers and one where there was artificial intelligence (AI) agent receiving the user's commands and managing the controls, similar to a smart assistant, but without limitations set in terms of capabilities. The survey results reveal a difference between both, with responses being typically more correct when an AI agent

helps smart home management. The authors then analyzed the queries given by respondents. They developed a grammar that could express all those queries, concluding that there were many similarities in sentence structure, which they believe allows them to convert any natural language command to an executable program. The authors conclude that current smart assistants are not developed enough to complete complex tasks and reveal that end-users can create natural language prompts that can be turned into smart home programs.

Other authors studied the applicability of natural language in smart homes, in a more complex fashion than current smart assistants can handle, concluding that end-users can create programmable rules through natural language, even if there is a basic structure attached to the rules, with the example of trigger-action programming (*cf.* Figure 3.1, p. 68), where there is a basic structure to specify rules. However, they can easily be understood by users, even if they are inexperienced, as shown by Ur *et al.* [Ur+14].

Ur *et al.* [Ur+14] carried three studies to understand how users use TAP on smart home scenarios. In the first study, they asked 318 workers on Amazon's Mechanical Turk (MTurk) to list five things that they would want a smart home system to do, concluding that most of them fit into four categories, namely: (1) programming, *e.g.*, "automatically turning on the lights when it is dark outside"; (2) self-regulation, *e.g.*, "adjust the house to my preferred temperature at all times"; (3) remote control, "hitting a button on my phone to turn on the lights"; and (4) specialized functionality, *e.g.*, "a breakfast-making machine". To further check the ability to model the workers' intents, they tried to fit them into the TAP model, finding that 62.6% of the submitted answers fit the model. In a second study [Ur+14], the authors downloaded a dataset of 67169 recipes (TAP rules) from IFTTT [IFT19], focusing only on the recipes related to smart home automation, which corresponded to a total of 1107 (2,1%), concluding that 513 recipes (0.8%) use physical devices as triggers and 594 (1.3%) use physical devices as actions. A third-study required that a sample of 226 MTurk workers complete pre-defined automation tasks using IFTTT, concluding that 80% or more of the participants successfully implement the presented automation cases. However, the authors' study on the diversity and complexity of the rules that end-users want to configure is too open and vague. There is no common base of devices to automate nor sample building schematic. Also, there is no dataset provided with the first study's collected cases (which limits the use of the gathered data).

Mi *et al.* [Mi+17] also carried a survey on IFTTT, including an analysis of over 408 services (third-party services such as Amazon Alexa), 1490 triggers, 957 actions, 320000 applets. Although their work is not IoT-focused, they carry an analysis on the IoT-related subset of the dataset. In their study, they conclude that the majority of the entries by users are trigger-action ones (*e.g.*, "turn on the light"); thus, the resulting applets are, in their majority, relatively simple, mostly due to the limited and simple interfaces exposed by most IoT devices. The authors also add that this is due to "the fact that most tasks (in the smart home context) we want to automate are indeed simple". While we agree that the limitations posed by the devices limit what end-users can program them to do and that the majority of the rules are indeed simple by nature, as the number of inhabitants and devices increases, the resulting operational context can be complex to model and reason about [Man+19].

3.2.7 Node-RED in Detail

Node-RED [Ope19b] is one of the most common (low-code) visual programming solutions (*cf.* Figure 3.2, p. 69) with a special focus on IoT development[4]. This solution was identified and briefly discussed in the previous section as one of the existent mashup tools, but due its popularity and being open-source, we will analyze it in detail as a way to grasp how this kind of solutions — *i.e.*, all-in-one development environments and runtime — work.

Node-RED can be used as a standalone development solution and, also, as an optional extension to other solutions (including Home Assistant [com21], FRED [BL16] and OpenHAB [Ope21]). Although most Node-RED users' state that they use it as recreational activity (*i.e.*, use Node-RED as part of a hobby), there is a considerable part of users that refer its usage as part of commercial solutions (*e.g.*, used in products of Hitachi, Siemens, Samsung, and Particle [OLe20]), or as a component of systems in production (*cf.* Figure 5.1, p. 135).

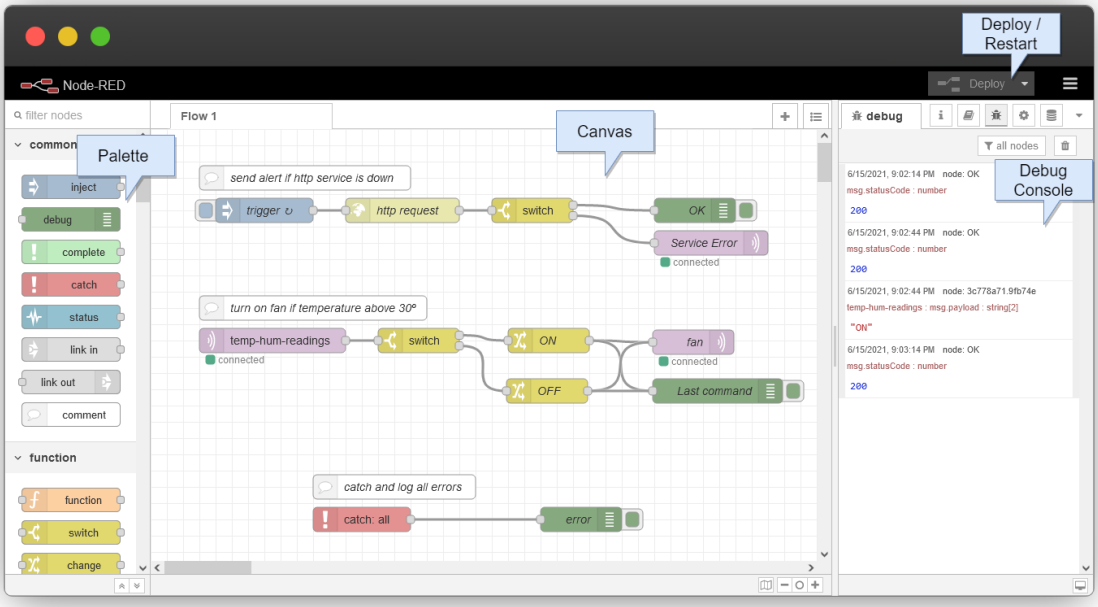


Figure 3.10: Annotated Node-RED development web-based user interface.

As an open-source solution it allows us to experiment with without limitations of closed-source solutions, modifying both its inner works and front-end (the default UI can be seen in Figure 3.10, p. 87). It is also extensible by design, and anyone can create and publish new *nodes*, which can be installed directly within Node-RED UI[5].

Although its widespread usage across application domains, it has several limitations and issues that impact both the end-users interaction and the dependability of systems built using it. In the following paragraphs an analysis from different perspective will be given highlighting

[4]Although the tool was initial only focused on IoT development it now claims to be suitable for developing any category of *event-driven application*.

[5]A search on npmjs [npm21], the package manager used by Node-RED, yields a total of 3644 packages for `node-red-contrib` (circa 2021). This means that there are at least 3644 community-made *nodes* available (since each package can provide more than one *node*).

some pending challenges of these approaches, showcasing how it maps with recurring problems of the IoT ecosystem and other existing development environments.

Technical Perspective

A summarized view on Node-RED was given in § 3.2 (p. 68). We consider Node-RED as mashup-development tool since it leverages a visual programming language but, as per their own definition is "a programming tool for wiring together hardware devices, APIs and online services" [Ope19b]. Node-RED applications are *flows*, composed by different *nodes* connected by *wires*. These *flows* can be organized into different *tabs* and can be directly deployed within the programming interface. The Node-RED runtime, which is based on Node.js, can run multiple *flows* and enables the direct exchange of messages within a *flow*, but also provides mechanisms for indirect inter *flow* and inter *node* communication via the *global* and the *flow* context (*e.g.*, this can be used to maintain a state shared by several *nodes* without direct message passing).

There are four main types of *nodes*: (1) *input nodes* which are *sources* of events (*e.g.*, new sensor reading), having no input and at least one output port, (2) *output nodes* which are *sinks*, having no output and at most one input (*e.g.*, post data to a web service), (3) *intermediary nodes*, having at most one input and at least one output (*e.g.*, check if a condition is met), and (4) *configuration nodes*, having no inputs nor outputs, which are used to share configuration data across *flows* (*e.g.*, MQTT broker configurations).

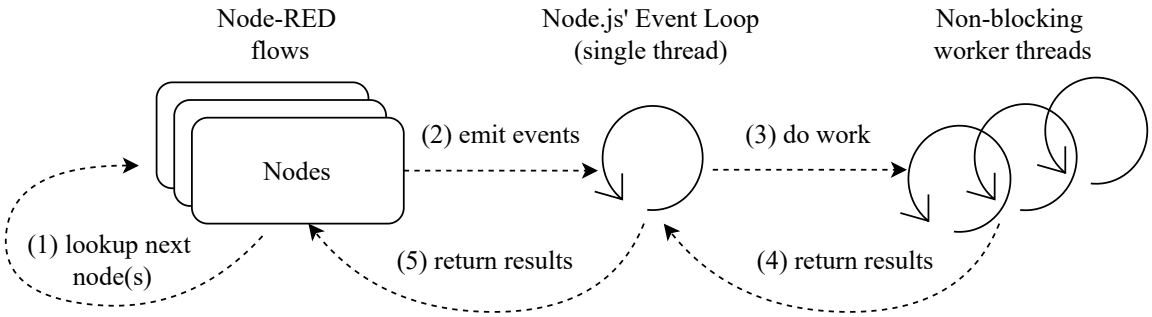


Figure 3.11: Node-RED high-level event processing model [BL14].

The information about a *flow* is stored in JSON files, which include details about the *nodes*, *wires*, and other relevant meta-data (*e.g.*, positioning of the *nodes* in the programming canvas). The JSON files can be used for exporting and importing other *flows* and to version control different versions of a *flow*. An example of a *flow* was given on Figure 3.5 (p. 78).

The high-level event processing model of Node-RED is depicted in Figure 3.11 (p. 88). The `Node` base class (*nodes* inherit from it) is a subclass of Node.js event APIs `EventEmitter`. This class implements an observer design pattern that maintains a subscriber list of all the *nodes* connected to it by *wires* and emits events to them. When a *node* finishes processing data from external sources or another node, it calls the methods `send()` with a JavaScript object. In its turn, this method calls the `EventEmitter emit()` method that sends named events to the subscribed nodes.

When a *flow* is changed, the developer can choose between a full or partial (only changed *nodes* or *flow* parts) deployment. As the developer deploys the system, all the involved *nodes* are

reset (calling `node.on('close',)` event) and any messages received in during deployment are lost. If a new or changed *flow* is received by Node-RED's REST API it is immediately applied and all the *nodes* in the changed *tab* are restarted. It is also possible to restart all the nodes (calling the `close` event).

Although Node-RED presents an easy platform to prototype simple systems, its current implementation has several limitations from our perspective. Concretely, we highlight the following as some main concerns and limitations from a technical viewpoint:

Centralized architecture Node-RED runtime is centralized by default, doing all the messaging passing and processing in a single single-threaded runtime[6] (*i.e.*, there is only one event loop that must be shared between all the *flows*). This poses several limitations and raise several concerns [OLe20], which include (1) all the processing of every *node* must be done in a single instance, thus not allowing distributing the computational tasks among available resources, (2) if one *flow* has a high-resource usage, it will impact the whole system, (3) if there is any issue (fault) with any *node* or *flow*, an error can lock up the event loop, which will lead to a system failure, and (4) there is no isolation of execution contexts (which raises both security and privacy issues). Additionally, there is no focus on having redundancy (multi-tenancy) with several instances of Node-RED (*e.g.*, mechanisms to synchronize state across instances);

Separation of concerns The web-based development interface (Node-RED Editor) and the runtime are highly coupled. This makes it easy for presenting status information about the runtime directly in the programming interface (*i.e.*, logs and *nodes* status messages). However, it makes it harder to interact directly with the runtime using other interfaces, limiting the possibility of using alternative implementations of the programming canvas (this is typically known as loose coupling). Additionally, only recently (version 1.2) the support for pluggable message routing was introduced (decoupling from the internal Node.js messaging routing mechanism);

Multiple inputs Although Node-RED supports several outputs per *node*, they cannot have differentiated inputs. This poses a technical difficulty in defining and readjusting the behavior of *flows* both during the design and runtime phases, including the configuration of test conditions and recovery measures. A strategy followed by some *nodes* is to force inputs messages to have a dedicated field in the JavaScript object called *parts*, with the *node* implementing additional logic to check if all required *parts* are received before proceeding;

Types and static analysis (structural correctness) *Nodes* do not have the notion of types (everything is a JavaScript Object); this allows the user to incorrectly connect two *nodes*, where the destination expects a different data type than the one sent by the origin one. This leads to common (and simple) errors that only make themselves noticeable after deployment of the whole *flow*, possibly introducing severe inconsistencies in the system. This can be even harder to detect as some errors may only appear in specific conditions.

[6] This is how Node.js works by design.

Adding such features would provide foundations for a *linter* that could inform the developer over issues in their *flows* before deploy;

Meta-programming Formal mechanisms of *introspection* and *reification*, essential for effective meta-programming, are nonexistent. This limits the possibility of adjusting *flows* in runtime and forces us to rely on external APIs that were not designed for this particular purpose and which might easily break.

Runtime modification There are no mechanisms to change messages at runtime to check the system's behavior (*e.g.*, when a temperature sensor emits a reading, it should be possible to change it). Thus, it is impossible to inject faulty, or rare, messages to verify if the system reacts as expected;

Testing As the *flows* increase in complexity, it becomes paramount to assert their correctness. This is even harder to assert given that there are no mechanisms for runtime modification. Node-RED lacks of a testing framework within the development environment which could be used to give insights on the correctness and performance of the developed *flows*, but this would imply that should exist some way to distinguish between development and production environments, requiring, *e.g.*, a secondary runtime which would act as the test runner.

Although this is not an extensive list of the technical challenges and limitations of Node-RED, it covers the ones that mostly impact this work (there is, for instance, some work detailing severe security issues in Node-RED [Ahm+21]). Some authors have been focusing some of these limitations, namely the distribution of tasks across devices, and some of these approaches are further detailed in § 3.2.5 (p. 79).

Node-RED lead developers have already considered some of these limitations [OLe20], including the introduction of a testing framework, a *flow* debugger (introducing breakpoints and capturing metrics), a *flow* linter, and (further) decoupling of the UI and the runtime (making foundations for a distributed Node-RED).

End-User Development Perspective

The Node-RED interface has three components: (1) Palette, (2) Workspace, and (3) Sidebar. The Palette contains all the *nodes* installed and available to use, divided into categories. They can be used by dragging them into the canvas, and additional features and configurations for each *node* are accessible by double-clicking them. The Workspace is where the *flows* are created and modified. It is possible to have several *flows* and *sub-flows* accessible with the use of tabs[7] (*cf.* Figure 3.5, p. 78). Lastly, the Sidebar contains information about the nodes, the debug console, *node* configuration manager and the context data.

Node-RED presents several challenges for its users (commonly users with limited technical expertise), specially as the system increases in complexity, thus making it harder to understand what is happening (system's behavior). One particular limitation is that Node-RED does not provide any feedback about the running system to the user during development (*e.g.*, evolution

[7] Although *flows* can be organized in tabs, all *flows* share the same runtime.

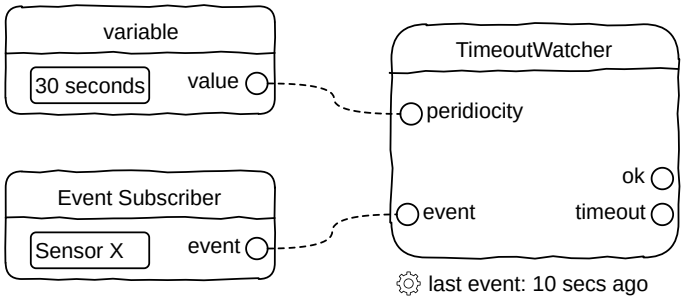


Figure 3.12: Mockup of an alternative *node* interface in Node-RED with annotations, labels and multiple inputs/outputs.

of an existing *flow*), requiring to deploy the system to see the changes. This makes it difficult for a user to create new — and modify existing — rules while ensuring that changes do not break the expected behavior [Hol02].

It also lacks mechanisms to inspect the inner workings of a node, inject or modify messages during runtime (as aforementioned), or even to verify if connections between nodes will not raise runtime errors. Its debug capabilities are also inadequate, relying on *"log to console"* strategies — leading to the proliferation of non-essential debug nodes in the *flows*. Every change the user makes to the system, even to add debug nodes, requires a new deployment. Common mitigation strategies to this problem includes the use of external solutions that provide visualization and monitoring mechanisms that allow understanding how the system is behaving (making the system observable to a certain degree), mostly through log analysis [Baj17; DS18].

Even given that there is a considerable amount of visual programming solutions for IoT (even if they are less popular in terms of usage and community than Node-RED [Ray17; Ihi+20]) they are typically limited in ways similar to Node-RED. We identify and summarize some of these limitations as the following:

Observability There is no out-of-the-box way to visualize the information that *flows* through the system in the development interface. Bypassing this limitation typically requires resorting to external solutions for monitoring, but information is limited (*e.g.*, there is no information about the current data being processed by a given *node*). Node-RED has a built-in dashboard for visualizing metrics, but it also runs completely separated from the development environment (and requires extra *nodes* to configure the visualizations);

Debugging and exploration Besides the provided logging capabilities of Node-RED, using *debug nodes* (*i.e.*, log to console), no other debugging technique is available. This means that breakpoints, *node* inspection, value history, and other apparatus are absent (both at runtime or as a capture-replay fashion), severely hindering the developer ability to understand what went wrong in the internal logic of a node. Additionally, every change in the system requires its deployment to production, including adding debugging nodes;

Support for labels and annotations Nodes do not visually provide sufficient information about their connectors and internal status, making *flows* harder to construct, debug, and adapt. Most (if not all) nodes configurations cannot be set or changed by other nodes.

A solution similar to Figure 3.12 (p. 91) seems more useful, not only in presenting this information but also in terms of flexibility regarding our goals;

Visual notations Node-RED should provide different visual notations (besides simple boxes with text) which could convey contextual information based on the data that the node is processing and its datatype;

Auto placing As *flows* grow in complexity it becomes an arduous task to keep the visual organization of *nodes* and *wires* simple to read and understandable (*e.g.*, overlapping *wires* and *nodes* hinders the reading capability). An automatic *flow* formatter could readjust and reordering the *flow* to make it more understandable.

Some mentioned visual enhancements are depicted in Figure 3.12 (p. 91). Other works, including the one by Ancona *et al.* [Anc+18], attempt to address some of these end-user developing challenges. Though they are capable of providing real-time information about the running system, they do not provide any real-time visual feedback in the Node-RED editor. No existing research, to the best of our knowledge, was found in providing some structural correctness verification in design time, neither any feature regarding runtime modification and exploration in visual development solutions for IoT.

3.2.8 Discussion

Taking into consideration the findings presented in Section 3.2, the next paragraphs match the pending challenges and issues with published works that delve into the same topics.

Traditional Development Approaches

Most typical solutions used for IoT system development are tightly coupled to a specific tier. For example, PlatformIO [Pla19] is arguably one of the most complete solutions for embedded systems development. However, it fails to cover the full spectrum of devices, languages, and vendor-specific technologies that build up IoT systems.

Other tools, platforms, and approaches have been developed in the last few years to find new approaches for developing IoT systems. However, a gap analysis on 39 IoT platforms circa 2016 (from PaaS to on-premises solutions) [Min+16] revealed issues regarding (1) the lack of support for heterogeneous devices, (2) handling different formats and models of device data streams, (3) lack of proper documentation and tooling, (4) ability to simultaneously target different tiers (edge, fog, cloud), (5) lack of semantic-based service discovery, (6) on-intuitive construction mechanisms, and (7) privacy and security.

While most of the traditional development solutions for software-only systems have built-in — or easy integration — with verification and validation mechanisms, this is not common in IoT development solutions that typically consist of independent and vendor-specific tools, which are analyzed and discussed by existing literature, and further discussed in Section 3.3, p. 97.

Several authors conclude that there is a need to evolve towards a *new generation of development environments* to bring software engineering practices up-to-date [Lar+17] with other

domains: *"software developers [need] to realize that IoT development indeed differs from mobile-app and client-side web application development"* and, as such, appropriate tooling is needed to effectively tackle this reality [TM17]. Larrucea *et al.* [Lar+17] also suggest that there is a need to shift from traditional development environments to cloud-based ones: *"development environments in the cloud — not for the cloud, but in the cloud — to enable the massively scalable verification and validation techniques (including simulation) that will be needed for most large mission-critical systems in the IoT"*. This idea has been followed by some practitioners that already offer cloud-based IDEs with built-in validation and verification mechanisms [Pyc21; Par21].

Model, Visual, and Mashup Development

Several development approaches have been suggested as alternative to traditional development regarding IoT systems, due to the specific constraints and necessities of these systems. More specifically, several works focuses on improving the abstractions and reducing the technical knowledge necessary to develop them, as the complexity of these systems keeps increasing. Most of the presented solutions are not mutually exclusive, being in some cases combined to improve the development, evolution, and maintenance processes.

Visual Programming Visual programming aims to abstract low-level concepts and details into high-level logic through the use of visual metaphors (*e.g.*, Programmed Logic Controllers are typically programmed using visual notations such as Ladder Diagrams) [YF03].

When analyzing the range of visual development solutions available, we can notice that while most of them target the fog tier, they can also be used in the cloud tier (as most of them target non-constrained Linux-based systems common to both the fog and cloud tiers) [Ray17; Ihi+20]. However, it is noticeable that those that target physical devices are particular to the edge tier and more coupled to specific hardware (primarily due to the direct interaction with its physical capabilities).

Other typical shortcomings of visual development solutions relate to (1) testing and verification procedures, (2) debugging capabilities, (3) scalability, both in terms of components and in terms of the visual metaphors used, (4) maintainability (*e.g.*, version control), and (5) real-time feedback to the developer (*e.g.*, most of the time the feedback occurs after deployment) [Ray17; Ihi+20].

Mashup-based and Model-based To improve the development of IoT systems, both model-based and mashup-based software development approaches have been pursued (*cf.* Section 3.2.4, p. 77, and Section 3.2.2, p. 70). These allow the abstraction of low-level programming details and processes into higher-level representations and constructions that can be used to manipulate IoT systems [MHF17]. However, these approaches commonly have limitations and unaddressed challenges, such as not capturing the full software development life-cycle [MHF17], having large-scale limitations [MHF17], and suffering from leaky abstractions [Spo04].

Prehofer *et al.* verify this in their work, suggesting that model-based approaches by themselves fall short in attaining the development requirements of IoT systems. The authors also

noticed a need for better tooling to use such approaches in real-world development and that concepts such as models@run.time — usage of models to manage and monitor the execution of systems — are not explored (which would ease the process of dealing with large-scale systems that are very dynamic). They also point out that mashup-based development tools would benefit from the use of model-based concepts. Although, both approaches have a common shortcoming: *if systems are very dynamic, modelling and mashup tools cannot accurately represent the system graphically* [PC15].

By analyzing the selected tools and their roles, both in academia and in the market, we observed that two of them share wide popularity (*cf.* Figure 3.2, p. 69): (1) Node-RED as a mashup-based tool that also leverages visual programming, and (2) ThingML as a model-based solution highly inspired in UML. We consider these two approaches as reference implementations of these two development models (mashup- and model-based), and will analyze and compare them in detail in the following paragraphs.

Node-RED also has several shortcomings, including the lack of a proper way to debug and test *flows*, which becomes essential given that IoT systems are typically large-scale and complex by nature, being easy to end up with highly-complex *flows* that are visually hard to understand and reason. Some works have been trying to improve some of the development issues of Node-RED, either by enriching the visual notations or by adding development mechanisms common to other development solutions, *i.e.*, linting, debugging, and testing[8] [Cle+18; Tor+20]. As of version 2.0, Node-RED added support for linting the visual *flows* and introduced some new debugging mechanisms [OLe20].

The shortcomings of mashup-based development solutions, which are shared with other visual programming approaches, pose a considerable problem regarding scalability — at least from a developing process perspective — due to limitations in the usage of visual metaphors. A study by Petre [Pet02] acknowledged that developers think that *simply repackaging massive textual information into a massive graphical representation is not helpful*, raising the need of adding *means of reasoning about artifacts too enormous to encompass fully in one view*.

Several authors have been trying to improve Node-RED. As an example, Blackstock *et al.* [BL14] and Margarida *et al.* [Sil+20] works include modifications that enable Node-RED to orchestrate the parts of a given *flow* in a *distributed fashion*, *i.e.*, orchestrating tasks between servers, gateways, and devices collaborating and coordinating actions defined in the same *flow*.

ThingML as a model-based development environment does in fact fully leverage models as a way to develop IoT systems [Har+16; FM17]. However, to use some of the devices features it is required to add device-specific code (*i.e.*, leaky abstractions). When comparing to UML, one key difference is that ThingML's primary syntax is lexical and not graphical despite visual notations being the most common approach for Model-Driven Engineering [SK03]. It lacks in covering the full development life-cycle, due to limitations on software deployment and updates. Also, the ability to share computational resources and devices among IoT applications, in a reliable and foreseeable fashion, is not covered.

From the analysis of these two tools, several conclusions can be made that show that both miss some essential features. Node-RED has a simple to use drag-n-drop visual programming

[8] As of version 2.0, Node-RED added support for linting the visual *flows* and introduced some new debugging mechanisms [OLe20]

interface that can be used as a reference in the domain of IoT, but needs to be extended to embrace the necessities of real-time feedback (*i.e.*, development feedback-loop) [Agu+19]. As per the current state of practice, we agree with Prehofer *et al.* [PC15] when they state that a model-based approach can be a suitable abstraction for IoT, as it is already proposed by approaches such as ThingML. However, it must contemplate other aspects, including the use of features common to mashup-based approaches.

From Figure 3.2, and taking into account the observations mentioned earlier, we observe that there is a bigger focus on model-driven solutions, but typically this kind of tools are unpopular and are at a more experimental development phase than mashup-based solutions.

Development of Decentralized IoT Applications

The approaches that focus on providing ground for programming and orchestrating (decentralized) IoT systems were characterized based on their mentions or support for the following features and characteristics:

Table 3.3: IoT decentralized visual programming approaches and their characteristics. Most of the solutions do not provide access to the source code, thus limiting our analysis.

Tool	Leverage edge devices	Capabilities	Open-source	Computation decomposition	Runtime adaptation
DDF [GLL18]	Limited[9]	Yes	Yes	Limited	Yes
`uFlow` [Sen+19]	Yes	Limited[10]	No	Limited	Limited[10]
`FogFlow` [Che+18b]	Yes	N/A	Yes	Limited	Yes
`DDFlow` [Noo+19]	Limited[11]	Yes	No	Limited	Yes

Leverage edge devices A decentralized architecture takes advantage of the computational power of the devices in the network, assigning them tasks. However, some approaches have limitations on the type of supported devices or only focus on the fog tier and not edge devices;

Capabilities The orchestrator must know each device's capabilities so that it can make informed decisions regarding the decomposition and assignment of tasks;

Open-source The license of software or tool is essential in terms of its usability. Open-source allows access to the code, making it possible for its analysis, improvement, and reuse;

Computation decomposition To implement a decentralized architecture, it is important to decompose the computation of the system into independent and logical tasks that can be assigned to devices. This is made using algorithms, which can be specified or mentioned;

[9] Assumes all devices run Node-RED (does not apply to constrained devices).
[10] Communication between devices is made through the cloud (Internet-dependent).
[11] Assumes all devices have a list of specific services they can provide.

Runtime adaptation A system needs to adapt to runtime changes, such as non-availability of devices or even network failure. The system notices these events and can take action to circumvent the problems and keep functioning;

From the analysis of Table 3.3 (p. 95), we can conclude that the current research for visual programming approaches that leverage the decentralized nature of IoT is incomplete. Most of the existing solutions have limitations either in their functioning or in the devices they support. `DFF` assumes that all devices run Node-RED, limiting the types of devices used to computing units capable of running some kind of operating system. `FogFlow` and `uFlow` are the only ones that specify how they truly leverage constrained devices, with the transformation of sub-flows into Lua code. `DDFlow` assumes that all devices have a list of specific services they can provide that should match the node assigned to them.

Regarding the method used to decompose and assign computations to the available devices, `DDFlow` describes the process using the longest path algorithm focused on reducing end-to-end latency between devices. `FogFlow` and `uFlow` mention several algorithms that could be used but do not specify which one was implemented. Both `DDF` and `FogFlow` do not specify the algorithm used besides some constraints, but are the only ones with accessible source code and an open-source license. All the surveyed approaches claim to have some runtime adaptation mechanisms to deal with changes in the system, such as device failures.

End-user Development

Popular end-user development environments are based on hybrid (visual-textual) rule-based programming (*if-this-do-that* or *when-trigger-then-action*). However, these kinds of solutions are not enough to express more sophisticated intents, *i.e.*, the process of expressing rules based on time, space, and fuzzy conditions make it too complicated for the end-user to attain their preferences [BV15].

There has been an effort towards conversational assistants and other NLP-based solutions for users without specific technical expertise to ease their interaction with IoT systems. However, most of the solutions are limited in some aspects: (1) the lack of standards (and the disregard by the existing ones by practitioners), most of the solutions are limited to certain devices or ecosystems, (2) the simplicity offered by some solutions limits the users' ability to configure more complex[12] scenarios and behaviors [He+19], (3) several limitations regarding the test and analysis of the resulting configuration (users have to manually trigger events to check if their program matches their expectations), thus making the system more opaque and hard to understand "why" some events happen — causality — and (4) some existing solutions depend on a first setup by technicians [Ihi+20].

Although there is a considerable amount of research challenges associated with the aforementioned points, there are still issues and challenges that are not being tackled effectively neither by practitioners nor by the research community. This is a bigger problem since it is hard to draw a line between the simplicity of end-user development and the ability to configure and manage complex scenarios without requiring a certain degree of expertise (*e.g.*, power-users).

[12] As an example, the complexity of managing devices schedules which rises with the number of devices and the shared conflicting preferences of household members.

3.3 Testing IoT Systems

Focus on testing the different tiers and components that compose an IoT system — from low-level/hardware specifications to high-level components — is needed to guarantee its performance, scalability, reliability, and, security. Yet, it is hard to define a clear boundary between the low-level and high-level components in IoT since they are strongly connected and dependent. However, the methods and techniques used for testing these are, typically, similar.

Testing IoT systems is a complex task as these systems depend on: (1) different software and hardware components, (2) different modules, architectures, and standards, (3) produced and sold by different manufacturers, and (4) with different working properties. One can also identify several challenges, for example, the high-heterogeneity, large-scale, dynamic environment, real-time needs, security and privacy implications, and the difficulties pertaining to test automation. This makes testing IoT systems especially hard. Moreover, tests have different needs depending on the target artifact (*e.g.*, network, code, and hardware), and different testing needs appear in each of the different IoT tiers (*cf.* Figure 2.7, p. 39):

Edge Testing Concerns the testing of the more low-level parts of the IoT system, like micro-controllers (*e.g.*, Arduino) and PLCs. Testing approaches like embedded system testing can be typically used to perform tests on the edge tier, asserting the edge devices against their specification [Koo11].

Fog Testing Focuses on testing the middle-point tier on IoT systems, normally composed of gateways. Software testing approaches can be seamlessly applied since the devices that belong to this tier have, typically, a comfortable amount of computing power and memory, and run full operating systems (*e.g.*, Linux). Additionally, since this is the connectivity-enabler tier, connecting the devices and the Internet *per se*, it should cover network testing [KK16] and security testing [Zha+14].

Cloud Testing Cloud testing addresses the need to test the unique quality concerns of the cloud infrastructure, such as massive scalability and dynamic configuration. This field has open-challenges and issues of its own, being extensively analyzed in the literature [Bai+11; RTS10].

When considering IoT testing as a cross-cutting concern, there are generally two avenues that can be found in the literature: (1) testbeds, and (2) emulators/simulators. IoT testbeds enable cross-cutting testing ranging from lower to higher level tiers. Although almost every testbed vertically encompasses all the tiers, they are *single-domain*, focusing on a specific domain of application or technological aspect, although there are some *multi-domain* testbeds that combine different technologies into a typical experimental facility. A survey on the currently active and publicly available physical testbeds is given by Gluhak *et al.* [Glu+11].

More recently, and not covered by the survey of Gluhak *et al.* [Glu+11], other testbeds have appeared (some as successors of previous exiexistingstent ones). One example is FIT IoT-LAB, a scientific testbed for testing small wireless sensor edge devices, together with heterogeneous communicating objects, built on large-scale infrastructure, and deployed around six sites in

France with over 2000 sensor nodes. It is the successor of SENSLAB testbed and is part of the *Future Internet of the Things* (FIT) platform [IoT19b; Adj+15]. Another example is the SmartSantander testbed which is a city-scale experimental research facility that supports the typical applications and services of a smart city, designed with the purpose of achieving experimental realism in terms of scale, heterogeneity, dynamic topology (mobility), reliability concerns, and user support and end-user involvement [San+14].

Emulators attempt to replicate with a high-level of fidelity the inner mechanism of the target system (*e.g.*, physical devices' emulation), while simulators only target the externally observable behavior, but not necessarily the intricate mechanisms of the real system (*e.g.*, smart city simulation) [SYB04]. The work pursued by Looga *et al.* [Loo+12] surveys the existing simulators and emulators, revealing issues on their suitability for testing IoT systems and proposing a new emulation platform for IoT (*e.g.*, MAMMotH).

3.3.1 Solutions and Approaches

An overview of these solutions, focusing on different tiers and enabling technologies, is listed below. A comparison of these tools is also depicted in Table 3.4 (p. 101).

PIO Unit Testing PlatformIO testing feature with support for several development boards and embedded systems [Pla19]. This testing feature is based on the Unity Test API by *ThrowTheSwitch.org* [Mik18].

IoTIFY IoTIFY is an application development environment for IoT without any hardware dependencies. By resorting to device virtualization, it provides a virtual laboratory for building embedded prototypes, and a network simulation for system scaling and data generation which can be used for both performance checks and connectivity [IoT19a].

ArduinoUnit ArduinoUnit is a unit testing framework for Arduino libraries. Being a lightweight library, developers can quickly test their systems in an Arduino board, despite their low amount of resources. However, it is up to the developer to upload the testing application to the target board, and the results must be interpreted by them, typically through the use of a serial port monitor [Mur19].

MAMMotH MAMMotH is a large-scale IoT emulator, being able to emulate ten thousand devices per Virtual Machine (VM), whose architecture presumes three distinct scenarios, namely: (1) mobile devices connected via GPRS to a base station forming a star topology, (2) stand-alone wireless sensor networks (WSN) connected to a base station via GPRS, and (3) constrained devices (*e.g.*, sensors) connect to proxies, which in turn connect to the backend — a large-scale IoT emulator. To reproduce the communication problems present in a real IoT environment, the proxy to which the devices are connected simulates a radio link for each node, able to delay and drop messages. Developers can then use this setup to create experiment scenarios, deploy them on a testbed and monitor the results [Loo+12].

SimIoT SimIoT is a toolkit to achieve experimentation on dynamic and real-time multi-user submissions within an IoT scenario. The toolkit is based on SimIC, a system that allows modelers to configure a diversity of clouds in terms of data center hosts and software policies wherein the desired number of users could send single or multiple requests for computational power, software resources, and duration of VM virtualization [Sot+14].

Cooja Simulator Cooja Simulator is an emulation/simulation platform developed for the Contiki OS. It is an extensible Java-based simulator able to simulate the network, operating system, and instruction set. It is also able to emulate the execution of the same firmware that may be uploaded to physical nodes, instead of simulating it. Cooja allows developers to test their code and systems long before running it on the target hardware [Gro19; BE15].

TOSSIM TOSSIM is a wireless sensor network simulator that was built with the specific goal to simulate TinyOS devices. Since TinyOS is event-based, it is easily translated into a simulator engine with discrete events, thus simplifying it and making it more effective. TOSSIM supports two programming interfaces (Python and C++), and has various levels of simulation, from hardware interrupts to high-level system events, such as packet arrivals [SIN19; LL03].

iFogSim iFogSim is a Fog Computing Simulator able to simulate edge devices, cloud data centers, and network links, and perform metrics evaluation on them. With these features, it allows investigation and comparison of resource management techniques based on QoS criteria (*e.g.*, latency, network congestion) [Gup+17; MB19].

MobIoTSim MobIoTSim is a mobile IoT device simulator, developed for Android, designed to help researchers learn IoT device handling without buying real sensors, and to test and demonstrate IoT applications utilizing multiple devices. This system can be connected to a gateway service in a cloud, such as IBM Bluemix Platform and Azure IoT Hub, to manage the simulated devices and to send back notifications by responding to critical sensor values. By using this tool, developers can examine the behavior of small IoT systems, and evaluate IoT cloud applications with a hand-held device [Pfl+16; KPG18].

IOTSim IOTSim is a Cloud simulator built on top of the CloudSim system and designed to support the testing of IoT Big Data processing, resorting to a MapReduce approach. By inherently supporting Big Data systems, it facilitates the understanding and analysis of the impact and performance of IoT systems by researchers and commercial organizations [Zen+17].

DPWSim DPWSim is a simulation toolkit to support the prototyping and development of service-oriented and event-driven IoT applications. It aims to support the OASIS standard Devices Profile for Web Services (DPWS), which, although it enables the use of web services on smart and resource-constrained devices, also reduces its scope to that of IoT devices implementing the referred device profile [Han+14].

SimpleIoTSimulator SimpleIoTSimulator is an IoT device simulator that can create test environments comprised of thousands of sensors on a single computer. It supports many common IoT protocols and can learn from data of recorded packet exchanges from real servers and sensors and model the behavior of its simulated devices from such data [Sim19].

Atomiton IoT Simulator The Atomiton IoT Simulator, built atop Atomiton Stack (a proprietary operating environment for the Internet-of-Things), is a prototyping and testing framework able to simulate virtual sensors, actuators, and devices with unique behaviors. It allows prototyping of an IoT solution and tests its scalability by providing the ability to create boundary test cases, resorting to the simulation of thousands of devices and events such as network interruptions, device response delays, and peak load [KPG18].

MBTAAS The Model-Based Testing as a Service (MBTAAS) allows systematic testing of IoT and data platforms. The approach resorts to a combination of model-based testing (MBT) techniques and service-oriented solutions. The solution has been tested on top of the FIWARE IoT-enabling platform. Further, the modularity of the solutions allows *integration testing* between different IoT platforms [Ahm+16].

CupCarbon CupCarbon is a platform for designing smart-city and IoT Wireless Sensor Networks (SCI-WSN). It is designed around two simulation environments, namely: one that models mobile units (*e.g.*, cars) and natural events (*e.g.*, wildfire, gas), and the other makes discrete event simulation of wireless sensor networks and can take into account the scenario designed in the previous environment. With the integration of the OpenStreeMaps framework, and allowing the programming of each node individually, CupCarbon is a useful tool to design, visualize, debug and validate distributed mechanisms for monitoring and collecting data about the environmental surroundings [Bou16].

EdgeCloudSim EdgeCloudSim is an extension of the cloud simulation solution CloudSim [Cal+11] adapted for simulating two-tier systems (*i.e.*, cloud servers and on-premises servers), focusing on the modeling of the system in terms of computational resources and network realistically with load generation and dynamic topology modeling features. The simulation devices and applications are specified using XML. Further, the simulation system is built in modules, allowing it to be extended [SOE18; Zai+18].

Fortino *et al.* An hybrid IoT system modeling and simulation approach based on the Agent-based Cooperative Smart Object framework (ACOSO framework) [For+13], built on top of the OMNeT++ discrete event simulator [VH08]. The approach allows modeling the IoT system in an agent-based way (multi-agent system), enabling the simulation and testing of the communication between different devices and other parts of the IoT system [For+17].

Despite being extensive, this list only presents the most common approaches and solutions. There is an ever-growing number of tools and approaches being designed to test IoT systems (both as a whole as well as for specific artifacts), most of them being of simulators and testbeds [Che+18c; JJW18; Glu+11].

Table 3.4: Overview comparison of the available tools on the IoT testing landscape. N/A symbolizes that there is no information available, or it was not have been found during our research.

Tool	IoT Tier	Test Level	Test Method	Testing Artifact	Prog. Lang.	Test Env.	Test Runner	Sup. Platf.	Scope
PIO Unit Testing	Edge	Unit	White-box	Code	C/C++, Arduino	Device	Local, Remote	15+	Market
IoTIFY	All	Any	White-box	N/A	N/A	Simulator	Remote	N/A	Market
Arduino Unit	Edge	Unit	White-box	Code	Arduino	Device	Local	Arduino	Academic, Market
MAM-MotH	All	Integration, System	Any	Network	N/A	Emulator	Local	N/A	Academic
Cooja	Edge	Integration	Black-box	Network	C	Emulator	Local	Contiki OS	Academic, Market
TOSSIM	Edge	Integration	Any	Application, Network	Python, C++	Simulator	Local	TinyOS	Academic
SWE Simulator	Edge	System	Black-box	Application, Network	XML, Visual	Simulator	Local	SWE Standard	Academic
SimIoT	Fog	Integration, System	Black-box	Any	N/A	Simulator	Local	N/A	Academic
iFogSim	Edge, Fog	Integration, System	Grey-box	Network	Java	Simulator	Local	N/A	Academic
MobIoT-Sim	Fog, Cloud	Integration, System	Grey-box	Application, Network	N/A	Simulator	Local	N/A	Academic
IOTSim	Cloud	Integration	Any	Application	N/A	Simulator	N/A	N/A	Academic
DPWSim	Fog, Cloud	Integration, System	Any	Application	WSDL	Simulator	Local	DPWS	Academic
SimpleIoT Simulator	Edge, Fog	Integration, System	Any	Network	N/A	Simulator	Local	N/A	Market
Atomiton IoT Simulator	All	Any	Grey-box	N/A	N/A	Simulator	Remote	N/A	Market
MBTAAS	All	Any	Black-box	Model	OCL	Platform	N/A	N/A	Academic
CupCarbon	All	System	Any	Network	SenScript	Simulator	Local	N/A	Academic
EdgeCloud Sim	Edge, Cloud	System	Black-box	Model, Network	XML	Simulator	Local	N/A	Academic
Fortino *et al.*	Edge	System	Black-box	Network	C++	Simulator	Local	N/A	Academic

Some work has been pursued towards hybrid simulation-based testing approaches which have both simulated and physical parts interacting with the system to capture more realistic behavior, thus improving the verification of these systems [Bos+19; Bur17].

3.3.2 Discussion

Taking into consideration the findings presented in Section 3.3, the following paragraphs analyze these findings by matching pending challenges and issues together with several published works that delve into the same topics.

A comparison of the available tools for testing IoT solutions was given in Table 3.4 (p. 101). Testing capabilities of each solution were analyzed according to different parameters.

Tools were divided by the tier they focus on, as presented in Figure 2.7 (p. 39). There is a clear relation between the tier and the artifact being tested. Edge tier tools, such as the PlatformIO and ArduinoUnit, typically focus on testing the code that runs on edge devices (*e.g.*, Arduino). However, to test the edge tier, the already available tools from embedded system testing can be helpful (*e.g.*, UNITY[13]). Fog and cloud-related tools are typically concerned about network or application testing, disregarding the low-level tests on code but testing at the system and integration level.

By analyzing the *Test Level* in which each tool is positioned, we noticed that tools are covering all levels, from *unit testing* to *acceptance testing*, at least partially. We must note that although some tools allow testing all the levels, they do not provide out-of-the-box functionalities to do so. An example of one of those is the FIT IoT-LAB testbed that provides a large-scale platform to test applications across different tiers but requires development efforts in, for example, retrieving and managing data from testing tasks. In other cases, the tools provide only partial support for the testing functionalities, *e.g.*, providing functionalities to collect all network logs and responses but not providing direct insights about that information.

A large part of the available tools focuses on a specific platform, language, or standard, lacking the support for the heterogeneity of the IoT field. An example of such a tool is DPWSim which focuses on the Devices Profile for Web Services (DPWS) standard language and the TOSSIM simulator for the TinyOS compatible devices. Another problem appears from the broad range of network communication protocols and IoT-enabling technologies (*e.g.*, reference architectures) that are now appearing in the market without any standardization, which leads to the lack of tools to test them in a platform-agnostic fashion. However, some have many supported platforms or are open to any implementation requiring extra development efforts.

About the different artifacts that need to be tested, the testing necessities are similar to those of highly-distributed systems. The artifacts that have more coverage by the available solutions (*e.g.*, MAMMotH and iFogSim) are the network and communication variables. Some available tools, such as the MobIoTSim, also provide features to carry application-level testing. The functionality, usability, and consistency can be tested within a real-world scenario, mostly disregarding the business logic. Some solutions are also available for code testing edge devices, such as PlatformIO. However, it is noticeable that there is a lack of tools for testing certain artifacts such as security and privacy, regulatory testing, and firmware/software upgrades (*e.g.*, out-of-the-box continuous integration functionalities).

In the security and privacy scope, there is work being pursued by the OWASP (Open Web Application Security Project) to *help manufacturers, developers, and consumers better understand the security issues associated with the Internet-of-Things, and to enable users in any context to make better security decisions when building, deploying, or assessing IoT technologies* [Pro17].

Testing environments are another distinguishable aspect of the available testing solutions. Most of the environments are purely virtual using emulation (*e.g.*, a virtual representation of an Arduino board) or simulation techniques (*e.g.*, simulation of a smart city or smart house). However, some efforts have been made in the creation of physical testbeds like the FIT IoT-LAB. Also, some traditional software testing tools are available (for unit testing purposes) that mostly rely on physical devices to conduct the testing.

[13] UNITY by ThrowTheSwitch.org, available at https://www.throwtheswitch.org/unity

Another relevant aspect is the maturity stage of the solutions and their openness. Most of the solutions have been presented in the literature. However, most are purely academic, and there is no access to their source code or software package. Comparatively, solutions that are available to be used are scarce, and most of them are closed-source, reducing the possibility of extending the tool functionalities or improving them using extensions or plug-ins. Some tools are only available on remote test runners, reducing the ability to test their specificities and raising privacy concerns.

The key features that differentiate IoT testing needs from those of traditional systems are the heterogeneity of the objects and the large-scale networks and systems. These factors lead to an increase in the complexity and difficulty of the testing process, thus increasing the need for developing new techniques and methodologies to test these kinds of systems similar to the practices already prevalent in other fields of software engineering (*e.g.*, testing automation, continuous integration features, fault-injection, and fuzzing). Nonetheless, there is already some research being conducted on harvesting these existing practices, such as the work by Chen *et al.* on fuzzing IoT systems [Che+18a] and testing frameworks such as Izinto [PLF18].

3.4 IoT Cross-Cutting Challenges

IoT poses a set of implications, barriers, and challenges that have a direct or indirect impact on the development of these systems, including in the design, construction and testing phases.

As long as these gaps and challenges remain unaddressed by the scientific community and market players, both the agility to make new IoT systems or reaching any IoT *utopia* (*e.g.*, interoperability between different IoT systems) will continue to be limited.

The following is a compilation of cross-cutting challenges that represent vertical *constraints* and *concerns* for the software development lifecycle. While research has been done to meet some of these challenges, mainly by the adoption of novel approaches such as blockchains [CVD16] and other distributed ledger technologies [Qin+19], Autonomic Computing [Mur04], Complex Event Processing [Che+14] among others, they do not suffice to meet the new requirements created by the particularities of IoT.

3.4.1 Resource Management

A smart city, as an example of a large-scale IoT scenario, shows the importance of efficient resource management due to the need for robustness, fault-tolerance, scalability, energy efficiency, QoS, and Service-level Agreements (SLA).

IoT can be pictured as a large graph, with numerous nodes with different resource capacity (computing, storage, connectivity). As a consequence, the selection and provisioning of such resources have a significant impact on the QoS of the IoT applications [BD16].

Delicato *et al.* suggest that finding an optimal solution on time, in such large and complex systems as IoT, is non-trivial; they also give hints on some pending open challenges [DPB17]:

- *Which are the proper mechanisms for resource allocation in data stream processing tasks* (e.g., *real-time processing)?*

- *How to ensure the correct application priority in systems where multiple applications coexist, with different requirements and criticality levels?*

Some authors also point out that *context awareness* is a crucial requirement for these systems, allowing them to dynamically adapt to application needs and execution contexts, thus deciding what data needs to be processed at a given time [DPB17; Per+12].

Besides other challenges mentioned before, there are also challenges regarding resource modeling, resource allocation, monitoring, and usage estimation that must be addressed to solve pending issues that affect directly, and/or indirectly, the interoperability and scalability of these systems (*cf.* Section 2.2.4, p. 44).

3.4.2 Identification and Discovery

When talking about discovery mechanisms in IoT, two different perspectives need to be considered. The first is to identify and locate the *things* (Resource Discovery) in the system, which can be achieved by storing and indexing meta-data information about each device. The second is to discover the target service (Service Discovery) that needs to be invoked for a given task [BD16]. Sometimes these two perspectives are implemented as a single mechanism.

Any discovery mechanism for IoT must take into account the amount of consumed energy, latency, and the impact on the end-user experience. Also, the use of centralized and dedicated servers for resource management is not suitable for IoT due to the unfixed and highly-distributed infrastructure, and to the high-volatility of devices (changing the network topology).

There are several challenges in this field as presented by Gubbi *et al.* [Gub+13], that can be summed as follows:

- *How to deal with the heterogeneous nature of the devices, variable data types, concurrent operations and the confluence of data from devices?*
- *What are the suitable approaches for automatic device identification and resource discovery that encompass the unfixed infrastructure of IoT?*

Addressing these challenges is needed to deal with the highly-dynamic topology, distribution, and heterogeneity of IoT systems.

3.4.3 Identity Management and Authentication

As the number of devices that are part of an IoT systems increases, their correct identification and authentication within the ecosystem where they operate becomes a challenge.

IoT devices need to be uniquely identified [Kev09; BD16], and solutions such as *ucode*[14] and Electric Product Code (EPC)[15] reduce the complexity of expanding the local environment and linking it with larger ecosystems [BD16].

[14] *ucode* generates 128-bit codes that can be used in active and passive RFID tags.
[15] EPC creates unique identifiers using a Uniform Resource Identifier (URI) codes.

However, Identity Management (IdM)[16] — the process of identifying, authenticating and authorizing humans or *things* in a system — is not yet adequately met in networks, with only a few proposed solutions. Furthermore, Sicari *et al.* raise some further questions regarding the identity problem and access control perspective [Sic+15]:

- *To manage access control, how could the IoT system deal with the registration of users and things and the consequent issuance of credentials or certificates by authorities?*
- *Could the users/things present these credentials/certificates to the IoT system to be allowed to interact with the other authorized devices?*
- *Could the definition of specific roles and functions within the IoT context address the issue of managing authorization processes?*

Some solutions have been proposed to address these questions suggesting the use of a subscriber method and a group membership scheme to deal with the access control of heterogeneous devices [Sic+15].

From an authentication perspective, a few solutions exist for constrained devices (*i.e.*, most *things*). However, there is no common solution or standard to manage the authentication process, limiting the interoperability between different systems, mostly due to the diversity of underlying architectures and environments that use unique authentication approaches. The development of a common standard is needed to assure a trustworthy environment which ensures a secure environment for communication [KS18].

3.4.4 Data Management and Analytics

IoT brought a new vision on collecting information about systems and their surroundings, through the wide-spreading of sensing capable devices. As a consequence, IoT has become one of the most significant sources of data, for both individuals and organizations [BD16]. It is widely accepted that the real power of IoT resides on the value of the data being collected and analyzed [Ver+11; Mar+17; STH18].

The amount of information generated by some of the sensors that are part of some devices fits on the view of Big Data since the data being generated is characterized by the so-called 3Vs, namely *velocity*, *volume*, and *variety*. As a result, IoT shares the same needs and challenges of a typical Big Data scenario. Although, it also adds the need of dealing with *variable veracity* — which increases the complexity of storing and analyzing the data — to generate useful insights, leading to possible data overload (too much data without value or the inability to analyze it) [RT17]. Several surveys point to research challenges yet to be attended on the same subject [STH18; Qin+16], namely:

- *How to deal with the distribution of data sources and interoperability between those sources (complicated by big data variety)?*
- *How to ensure performance when dealing with the volume and velocity of the data being collected?*

[16] Also known as Identity and Access Management (IAM).

- *How valuable is the knowledge obtained by the analytics process (Analytical Value)?*

Several research challenges remain to be addressed in the context of Big Data for IoT (*e.g.*, traditional SQL-queried relational database management systems (RDBMSs) are usually unsuitable for IoT needs *out-of-the-box*). The problem is even more complex when factors such as integrity are taken into account, not only because of their impact on the QoS but also for its security and privacy-related aspects, especially on outsourced data [RT17].

Recently, several advancements appeared as solutions for IoT needs, such as lambda architectures, stream processing, batch processing, and time-series oriented databases [BD16].

But finally, as Hurlburt *et al.* points, a more fundamental question must be posed that has direct implications on privacy (detailed in the next subsection) [HVM12]:

> *As the IoT becomes ubiquitous, issues of information ownership will become crucial. Who will own the oceans of data IoT will generate?*

Also, by taking into consideration the Open Data[17] momentum, questions arise if anyone should own the data generated at any instance, especially for government-based IoT scenarios (*e.g.*, Smart Cities).

3.4.5 Security and Privacy

The spreading of IoT usage increased the size of the attack surface that should be taken into account by manufacturers, developers, security researchers, and those looking to deploy or implement new IoT applications. Sicari *et al.* points out that there are eight main categories of security concerns that must be considered in the IoT landscape, namely: authentication, access control, confidentiality, privacy, trust, secure middleware, mobile security, and policy enforcement [Sic+15]. The same authors then explore the particularities of each of these concerns and define several open research questions, from which we highlight the following:

- *How heterogeneous devices and users can dynamically interact and agree on the same communication protocols, also ensuring security and privacy?*
- *What are the suitable trust negotiation mechanisms to handle data stream access control?*
- *Is it feasible to reuse the traditional security mechanisms* (e.g.*, encryption algorithms), or is it better to start from new solutions?*
- *How to guarantee access permission in an environment where not only users but also things could be authorized to interact with the system (human-to-machine* vs. *machine-to-machine)?*
- *What is the suitable privacy model (regulation/standards) and policies that must be followed when making IoT systems (both in legal and technological aspects)?*

[17] Open Data is the idea that some information (*e.g.*, datasets) should be freely available to everyone to use and republish as they wish, without restrictions from copyright, patents or other mechanisms of control.

IoT devices typically have significant resource constraints that influence the feasibility of using standard security mechanisms (*e.g.*, cryptography algorithms need considerable processing capabilities, bandwidth, and energy to provide end-to-end protection). As an example, the wireless communication channels used by the majority of these devices are vulnerable to eavesdropping and man-in-the-middle attacks due to the use of non-encrypted communication protocols [Far+15].

On several occasions, the lack of a security layer on top of IoT applications has been leveraged by malicious parties leading to nefarious consequences. One of the most known examples was the use of IoT devices to conduct a Distributed Denial of Service (DDoS) attack against the DNS provider Dyn which supports several relevant Internet platforms and services such as PayPal, Twitter, and VISA circa 2016 [Fru+18].

The DDoS attack was performed using a botnet of devices already infected with the so-called Mirai malware. This malware took advantage of the misconfiguration of IoT devices (default passwords for the telnet or SSH services) to gain shell access [Kol+17; DPC17; Ant+17].

However, there are other examples of vulnerabilities that researchers were able to exploit successfully. Examples included a flaw in the radio protocol ZigBee that allow attackers to control smart Philips light bulbs remotely without authorization [Fru+18] and several 0-day vulnerabilities in commercial IP cameras from vendors such as D-Link and Linksys, which allowed attackers to be able to do remote code execution on those devices [Ser+18].

The Open Web Application Security Project (OWASP) has been working towards reference documentation on how to tackle the IoT security perspective. As self-described, this project *is designed to help manufacturers, developers, and consumers better understand the security issues associated with the IoT, and to enable users in any context to make better security decisions when building, deploying, or assessing IoT technology* [Pro17]. However, there are several open issues with the modeling of the typical security threads, such as physical access to devices [AK14].

IoT is being considered *Insecure by Design* given the lack of procedures being done to ensure the security of the devices and their communication [ONe+16].

From a privacy perspective, IoT introduces a whole new degree of concern for consumers. These concerns are not only due to the ability of these devices to collect personal information like users' names and telephone numbers but because these devices can also monitor user activities, understanding their tendencies and preferences as well as their surrounding environment (*e.g.*, when users are in their houses and what they had for lunch) [Wei+15].

Standardization and regulatory (legal) limitations and gaps are also crucial and need to be addressed in this field. Rolf H. Weber points out that efforts are being made by different entities such as the European Union Commission, trying to encompass the reality of governance by multi-stakeholders (results from having different vendors of IoT solutions) and critical issues such as transparency and non-discriminatory access, accountability, security, and confidentiality [Web09].

3.5 Fault-Tolerant Systems

Robert Hanmer in its book Patterns for Fault Tolerant Software defines fault-tolerance patterns as solution templates for recurring problems of dependable systems. Fault-tolerance

patterns might be suitable for systems that are stateless, stateful or both, are based on observers and monitors (humans/computers) and work orthogonally to the system's primary function.

Fault-tolerance patterns are grouped by Hanmer depending on their purpose or on the phase of the development lifecycle that they fit on, resulting in the following groups of patterns:

Architectural Patterns Patterns that cut across all parts of the system (from an architectural point of view), and, as such, need to be applied in the early stages of system design.

Detection Patterns Patterns which purpose is to detect the presence of root faults, error states, and failures on the system.

Error Recovery Patterns Patterns that allow a system that entered an error state to continue delivering its correct service (moving to a new error-free state).

Error Mitigation Patterns Patterns that allow a system to deliver its correct service using error masking and compensation techniques, without changing the system state.

Fault Treatment Patterns Patterns for preventing the re-occurrence of an error by fault reparation. Such patterns deal with system verification, diagnosis (location and nature of faults), and fault correction techniques.

The following subsection approaches each one of these groups, detailing different patterns as they are presented in the literature. Although the work of Robert Hanmer, PATTERNS FAULT TOLERANT SOFTWARE, circa 2007, being the de-facto *pattern language* for fault-tolerant system's design and development, consisting of a total of 63 patterns, only a selection of those are hereby analyzed, giving their fitness to the research statement of this thesis. More recent works, including the book from Kuhn *et al.* , REACTIVE DESIGN PATTERNS, and the work from Chris Richardson, MICROSERVICES PATTERNS, present some novel details and patterns in this area, but still are closely coupled with the Robert Hanmer works.

3.5.1 Architectural Patterns

Simple Component Pattern

A component shall do only one thing, but do it in full [KHA17].

Derived from the Single Responsibility Principle, if a system performs multiple functions or the functions it performs are of high complexity, the system must be broken into different components or modules, each one responsible for a certain task (but entirely).

In the scope of the SIMPLE COMPONENT PATTERN, we can consider the following patterns as directly related to it.

UNITS OF MITIGATION (Isolation) *Decide during architecture what the units of fault-tolerance are.* Each component (or a set of components) is contained in a unit of mitigation which contains the errors that can arise for these modules and handle their recovery process after an error [Han06].

Error Containment Barrier *Build barriers into your system so that errors cannot propagate from one part of the system to another.* The boundaries of a Unit Of Mitigation are defined by an implementation of an *error containment barrier*, that disables error propagation [Han06].

Error Kernel Pattern

In a supervision hierarchy[18]*, keep important application state or functionality near the root while delegating risky operations towards the leaves.* Systems are constituted by several modules which can have different complexities, proneness to failure, and reliability requirements. Thus, some functions of the system must rarely go down, whereas others are necessarily exposed to failure. The system should be designed in a way that most error-prone modules have been restricted from having an impact on the core system functions [KHA17].

Redundancy Pattern

Provide redundant capabilities that support quick activation to enable error processing to continue in parallel with normal execution. It is expected that services have the highest availability (*uptime*) possible. The best way to reach such a goal is to reduce the MTTR of the system. As such, with redundancy, a system can resume its delivery of correct service before errors are corrected, by using identical copy. Redundancy can be either spatial (replication), temporal, or informational [Han07].

Human Interaction Patterns

People are the cause of many failures in long-running systems through the inappropriate actions they sometimes take [Han06]. As such, requiring humans to intervene in system error correction can lead to procedural errors contributing to even larger system unavailability. However, when there are skillful operators available, the system should be able to count and rely on humans to address these errors. As such, two patterns appear:

Minimize Human Interaction *Humans make mistakes and are slow; to minimize downtime the system should take care of itself, without human intervention* [Han06].

Maximize Human Participation *Design the system to enable knowledgeable operating personnel to participate positively, toward error detection and error processing* [Han07].

Someone in Charge Pattern

All fault-tolerant related activities have some component of the system (someone) that is clearly in charge and that has the ability to determine correct completion and the responsibility to take action if it does not complete correctly. If a failure occurs, this component will be sure that the new failure does not stop the system. Errors can arise even during error processing tasks. When this happens the system might stop all the running tasks, including the error processing jobs. Thus, the need

[18] In a supervision hierarchy, a supervisor is responsible for starting, stopping and monitoring its child processes.

for a component that has higher responsibility and is able to determine correct completion of tasks and intervene in the case of failure, to avoid new failure that could lead to the stop of the system [Han07].

Escalation Pattern

When recovery or mitigation is failing, escalate the action to the next more drastic action. In the case that an attempt to process an error in a given component does not archive the expected results, additional measures are required to be triggered. Escalation should report to Someone In Charge [Han06].

Fault Observer Pattern

Some part of the system should know that a fault is present and report it and maybe escalate actions. If a system delivers correct service even in the case of errors (due to its ability to locate and automatically correct them), there must be a way of tracking what faults and errors have been detected and processed, both currently and in the past. Thus, all errors should be reported to the interested parties [Han06].

Software Update Pattern

The system and its applications must not stop operating, not even to install new software. Even with the use of good fault prevention through software quality methods, faults will be frequent in software, especially in their initial releases. As such, the ability to change the software during its correct service delivery should be built-in from its first release, since that patch's[19] for fault will be release after the software being deployed [Han07].

3.5.2 Fault Detection Patterns

In the following paragraphs, the body of knowledge about fault detection is presented in the form of patterns. As aforementioned, although the constant evolution of the field, these patterns document the most widely used tactics in terms of fault-tolerance.

Fault Correlation Pattern

Analyze multiple error indications to identify the actual active fault. In order to be able to identify which fault is activating, and classify it according to a determined fault category, the unique signature of an error should be identified. Being able to do so allows the activation of error processing mechanisms for that specific fault category [Han07].

System Monitor Pattern

Some errors will only manifest themselves at a system level. Check for them at this level. To guarantee the correct service deliver there should exist a way for the system to verify that the system

[19] A patch, sometimes just called a fix, is a small piece of software that is used to correct a problem, usually called a bug, within a software artifact.

itself or its parts are alive and functioning correctly. A monitor should be put in place that watches the system behavior and, in case of component malfunctioning, report the occurrence to the Fault Observer [Han07].

In order to get complete system monitoring, several patterns can be applied, such as the following:

Heartbeat *Send a status report at regular intervals to let other parts of the system know their status* [Han07; KHA17].

Acknowledgment *Send a reply message to let a communicating party know that the sender is alive* [Han07].

Watchdog *Build a special entity to watch over another to make sure that it is still operating well* [Han07].

Proactive Failure Signal *A component can diagnose some failures itself, thus, a component is assumed to not have failed until it has sent a failure signal* [KHA17].

The system monitor should also consider the Realistic Threshold pattern, defining different time threshold depending on the task (*timeouts*). As per the Robert Hanmer work [Han07], the latencies thresholds should be defined in a way that the monitor will be informed in a timely enough manner to meet the availability requirement, and yet is the maximum possible to avoid false triggers. The following points describe the mechanisms to define *message latency* and *detection latency* threshold values:

Messaging Latency Threshold based upon the worst case communications time combined with the time required to process one Heartbeat message.

Detection Latency Threshold based upon the criticality of the functionality. The value used should be a multiple of the messaging latency. Use a smaller multiple for critical or unique tasks and larger for Redundant tasks.

Voting Pattern

When more than one result is available for a computational result or question or task, vote to pick the correct one. The use of redundancy in space results in multiple answers, thus the need appears of choosing one answer from the available options by devising a voting strategy [Han07].

The voting strategy can be either (1) *exact voting* where decision leads to correct result (or to an uncertainty state alert notification), or (2) *inexact voting* where the comparison of answers might lead to multiple correct results (that can be solved by *non-adaptive voting*, using boundaries on discrepancy minimum or maximum, or *adaptive voting*, where results are ranked based on experience) [Han07].

Maintenance and Exercises Patterns

Some errors can be anticipated and mechanisms put in place to avoid their occurrence (*e.g.*, periodically checking data storage systems). Thus, two patterns can be applied:

Routine Maintenance *Perform routine, preventive maintenance on the system.* Periodically and automatically perform routine, preventive maintenance to prevent faults from silently accumulating [Han07].

Routine Exercises *Routinely exercise, or execute, the system components that will be required in an error situation.* Make sure that Redundant spare components truly work in a failover case (identify components that have latent faults), thus guaranteeing they are available when required [Han07].

Routine Audits Pattern

Check data by a background task to make sure that it is correct. Faulty data can exist in a system for a long time before it causes errors. As such, mechanisms should be put in place to routinely check the data in order to find latent faults (usually by using a low priority maintenance task) [Han07].

One pattern that can be used to detect latent faults is Checksums — add information to data or messages to verify that they are correct. Detection of incorrect data by storing aggregate information along with the data value, or over the block of data (*e.g.*, parity bits or hashing) [Han07].

Riding Over Transients Pattern

Sometimes the prudent thing to do is to ignore an error if it is something that might be due to a transient situation. Transient errors can appear in a system without having long term effects on it, thus, the System Monitor should check the frequency of occurrence but take no other action unless it is occurring more than expected [Han07].

A pattern that can be applied to check that an error is transient or intermittent is the Leaky Bucket Counter. The pattern concrete implementation should keep a counter that is automatically decremented and incremented by each event and/or fault. The exceeding of the predefined upper limit of the bucket identifies a permanent fault.

3.5.3 Error Recovery Patterns

In the following paragraphs, the body of knowledge about error recovery is presented in the form of patterns. As aforementioned, although the constant evolution of the field, these patterns document the most widely used tactics in terms of fault-tolerance.

Quarantine Pattern

Take steps to isolate and confine a sick element to keep it from corrupting the rest of the system. When Units Of Mitigation pattern is followed, if an error occurs its propagation is limited by the barrier, thus quarantining the error [Han07].

Concentrated Recovery Pattern

The system should have as few distractions as possible during error recovery. The system should be able to focus all resources on recovery activity, thus minimizing the unavailability of the system or one of its parts [Han07].

Rollback, Roll-Forward and Restart Patterns

When a system enters an error state three approaches can be followed:

Rollback *Resume normal execution by moving to a state in the execution path but before the error occurred.* Move the system to an error-free state by returning to a state before the error occurred [Han06; Han07]. Timing of the checkpoint or last requests decides about the rollback point to go to.

Roll-Forward *Resume normal execution by advancing to a future state that would have been reached if the error had not occurred.* Move the system to an error-free state by advancing to a future state that does not contain the error [Han07; Han06].

Restart *Resume execution by restarting the program from the beginning. Prefer a full component restart to internal failure handling.* Whenever a component is detected to be faulty, no attempt is made to repair the damage and, instead, all the component resources are released, and it is started up again from scratch [Han07; KHA17]. This pattern is also known as Let-It-Crash.

However, underlying any of these patterns, several considerations should be made, such as the following ones:

Limit Retries *Do not return to the scene of an error without changing something, because the error might reoccur.* In scenarios that faults are deterministic (a latent fault with the same stimuli leads to fault activation) the propagation of error within itself must be stopped by limiting retries [Han06].

Checkpoint *Save state periodically in a way that allows execution to be resumed with a consistent state.* Avoid loss of important data or results during recovery by saving global state information, reducing the need to recreate the entire execution from startup to the point of the saved state. Each process should create a checkpoint at the best time for it to do it (*cf.* Individuals Decide Timing) [Han06].

Failover Pattern

Recover by switching to a redundant unit. In the case that the active part of a group of redundant elements has a fault and restoring of error-free operation in active part did not succeed, the switch must be made to redundant resources. Consideration about quarantining the faulty part of the system and establishing someone in charge of dealing with the change of replica must be taken into account [Han07].

Remote Storage Pattern

Consider REDUNDANCY *and other recovery factors when deciding where to place checkpoints.* Store the saved checkpoints in a centrally accessible location, enabling a new processor to access the saved checkpoint which minimizes the period of unavailability. However, there should be taken into consideration that there must not be a single point of failure in the system [Han07].

Data Reset Pattern

Restore some data to its initial (or a predetermined) value when it is found incorrect. In order to recover from an un-reconstructable, uncorrectable data error, the system must reset the values stored to their initial values or reference points [Han07].

3.5.4 Error Mitigation Patterns

In the following paragraphs, the body of knowledge about error mitigation is presented in the form of patterns. As aforementioned, although the constant evolution of the field, these patterns document the most widely used tactics in terms of fault-tolerance.

Marked Data Pattern

Mark erroneous data so that others are not corrupted by it. Mark erroneous data values as invalid and define rules for how to process these values when encountered later to prevent any part of the system from propagating the error. Basically, the erroneous data should be quarantined, not allowing the system to use it and, consequently, do not derive any actions from it [Han06; Han07].

The Circuit Breaker Pattern

Protect services by breaking the connection to their users during prolonged failure conditions. Different parts of the system should be easily disconnected in case of failure, in such a way that failures do not spread uncontrollably across the whole system. This pattern has been used for a long time (the 1920s) in electrical engineering [KHA17].

Overload Patterns

A system, while delivering its correct service, handles requests and performs work based upon those requests. However, depending on a wide variety of factors (*cf.* Figure 2.10, p. 47), errors can appear on the system, being one of the most common sources of problems the request *overload* (*i.e.*, too many requests at a time). Thus, several patterns can be applied to mitigate the occurrence of errors due to the increase of the system *load*. Some of them are the following [Han07]:

OVERLOAD TOOLBOXES *Have separate collections of techniques for dealing with different kinds of overloads.* Handle overload situation with too many requests for the system by implement dedicated overload treatment for each resource class.

Shed Load *Discard some requests for service to offer better service to other requests.* Throw away a minority of requests to serve the majority.

Finish Work In Progress *Categorize arriving work as either new work or related to something that is already in progress. Give priority to work that continues work that is already in progress work.*

Fresh Work Before Stale *Giving better service to recent requests enables at least some requests to get good service. If all requests wait in a queue* (cf. *Queue For Resources Pattern*) *then none of them receives good service.*
If the requester gives up, his retry eats up even more resources.

Slow It Down *Sometimes the best thing to do when many errors are occurring is to slow down. Transients might clear and the permanent errors will become evident.*

Deferrable Work *Make the routine work deferrable.*
Under heavy load, prioritize essential tasks, and defer non-essential tasks (*e.g.*, Routine Maintenance).

Equitable Resource Allocation *Divide the resources up equitably between all the requestors.* Create a pool for similar requests and allocate resources to pools.

Queue For Resources *Queue requests for resources in a way that will protect the system.*
In order to deal with requests for resources that cannot be handled immediately when they arrive, these requests should be stored in a queue and be handled later.

Further, some protective controls can be put in place to improve the error mitigation in case of overload on the system:

Expansive Automatic Controls *Protect the system from too much work/traffic by providing new ways to do the work.* Design some system parts for only being used in case of overload [Han07].

Protective Automatic Controls *Protect the system from too much work by restricting what work is allowed into the system.* Shed internal work, shed incoming load, do nothing [Han07].

3.5.5 Fault Treatment Patterns

In the following paragraphs, the body of knowledge about fault treatment is presented in the form of patterns. As aforementioned, although the constant evolution of the field, these patterns document the most widely used tactics in terms of fault-tolerance.

Let Sleeping Dogs Lie Pattern

Consider the true benefits to the system of correcting the faults versus leaving the fault present. When a fault is found in the system (by the system itself or its maintainers), the weight of the risks and costs should be evaluated in regard to the benefits and rewards associated with the correction. In some situations it is essential to correct the faults with the highest risk of reoccurring or the highest potential for damage if they do reoccur, and avoid the correction of the faults with lower risks reducing the potential of introduction of new ones into the system [Han07].

Reintegration Pattern

After making the faults passive, return the repaired component to service. In the eventuality of a component enters error state, and after the mechanisms for removing/disabling the fault are activated, the component should be reintegrated into the system. To do so, a predetermined procedure to reintegrate the corrected component into the system should be followed [Han07].

Reproducible Error Pattern

Activate a fault while monitoring its behavior to determine clues to its nature and potential corrective treatments. In order to correct the real fault, stimuli on the fault in a controlled manner should be done to verify that the fault did indeed cause the observed error and that the fault is still present in the system. The result should be an unexpected behavior of the system when checked against its specification [Han07].

Fault Correction Patterns

Faults can arise from a system's implementation or from the system's misuse by a human counterpart. As such, two patterns on how to correct faults exists, namely [Han07]:

Small Patches *Surgically correct an erroneous program part.* Evaluate what Software Update will have the least chance of introducing extraneous faults or bringing in extra capabilities that are not needed, and is able to patch what is needed.

Revise Procedure *After a failure in which people contributed to downtime instead of minimized it, revise the procedures that they followed to avoid the problem in the future.* When operating personnel following the system's predetermined procedures contribute to failure durations, revise the procedures to avoid repeating the same sequence of errors.

3.5.6 Fault-Tolerance in IoT Systems

IoT and its applicability across different scenarios led to several considerations depending on their criticality.

The use of IoT in mission-critical scenarios, such as the healthcare or ambient assisted living, where connected devices monitor and, even, act, upon data about the health of the patient,

should consider that the system components can not tolerate any type of failure during its mission time[20]. In other words, there is zero-tolerance for failures, and, thus, the MTTF for these components should be strictly greater than the mission time, and the MTTR should be close to zero during mission time [BD16].

However, there are scenarios where IoT systems can tolerate faults, thus recover from error states. Here, patterns such as the Restart, Roll-Forward or Rollback can be used to recover from such states. As an example, if an embedded component on a vehicle fails it can afford to restart to avoid a catastrophic failure (*e.g.*, an accident). Thus, in such scenarios, the goal is to make MTTR as small as possible, more than reducing the MTTF [BD16].

Further, in some scenarios, the system can tolerate the erroneous input for some time, within the user-defined safety limit before getting it fixed (this is commonly known as graceful degradation or, in the automotive literature, as *limp* mode [Ros+20].). The application of patterns such as the Riding Over Transients Pattern and Let Sleeping Dogs Lie are applicable. Further, Small Patch Pattern can be applied to get rid of the fault (if identified).

When in IoT systems, we have to consider that there are three main sources of errors, namely [BD16]:

- **Infrastructure Fault** Given that IoT devices can end up operating in unanticipated scenarios, and, as such, some of these scenarios can lead the system to suffer from infrastructure failures.

- **Interaction Fault** IoT is, by nature, highly distributed in terms of devices and applications and heterogeneous. The need to communicate and share data between the different entities on the system can lead to operational failures due to several reasons (*e.g.*, network unreliability).

- **Fault Service Platform** Most of the IoT reference architectures propose the existence of one or more service platforms (*hubs*) that integrate the different devices and applications. However, these platforms will rarely be built from scratch, integrating with many third-party products (heterogeneity) and with several external systems. Thus, even if it is assumed that these components have been thoroughly tested for their own functionality, many transient faults can be due to off-the-shelf components.

One of the most used approaches in IoT systems to *nullify the impact of a fault* is to reduce the single points of failure in the system [BD16], thus, the application of the Redundancy Pattern at different levels such as the infrastructure, the network, and in the software.

Further, the use of loosely coupled components, Simple Component Pattern (*cf.* Units Of Mitigation Pattern), so that the failure of one component does not bring the entire platform down. Also, the use of built-in graceful degradation mechanisms (fail-safe) in each component can make an IoT system survive for longer.

[20] Mission time is considered to be the time during which the system it is actively working.

3.5.7 Discussion

The need for dependability to have a proper QoS is fundamental to any software system. Fault-tolerance research has been inter-winded with the need of increasing the system's *uptime*, reduce response time to the user, while guaranteeing integrity, confidentiality, safety, and easiness of maintaining such systems during their lifecycle.

The design patterns community has worked for a long on the systematization of the means by which dependability can be attained, and the extensive work of Robert Hanmer summarizes these efforts. However, some more recent contributions are also considered, and it is noticeable that patterns from other fields are being adapted for the new reality of systems, highly dependent on software instead of hardware-only solutions (*i.e.*, *software-defined everything*).

3.6 Autonomic Computing

Many components (*e.g.*, devices and services) that are part of the IoT systems make it challeging to deploy manually, setup, manage and maintain each component, thus exceeding the human ability to manage all connected devices [TMD19]. As a motivational scenario, *Internet of Underwater Things* [Dom12], which describes the use of things in under-water environments is a prime example of a system expensive to maintain due to the remote localization and environmental factors. Further, Ramakrishnan *et al.* identify other motivations for autonomic computing, namely [Ber+13]:

- Increase the performance by deploying heavyweight application components on faster hardware;
- Reduce the amount of communication and network latencies between distributed components;
- Optimize the overall energy consumption of the application components on the different platforms.

Even with the vision of autonomic computing gaining momentum, current approaches are limited with issues ranging from dependency on high-computing power units (when edge devices are typically power-constrained), lack of evidence on how such solutions work at large-scale, and only focus on one or, at most, two, self-* properties, not approaching autonomic computing from a holistic point-of-view.

Thair *et al.* identify in their work several challenges and future research directions [TMD19], which can be mapped to each one of the self-* properties proposed by IBM. Nonetheless, to achieve the plateau of autonomic computing, all self-* should be conciliated.

Regarding self-configuration, IoT devices and networks should be able to automatically configure themselves accordingly with high-level policies (*e.g.*, access control). This concerns not only the configuration of each component individually but also the system as a whole (*e.g.*, including network). With the ever-growing number of different protocols and architectures (ecosystem fragmentation), there is no cross-cutting solution that addresses these problems. Chatzigiannakis *et al.* suggest the use of semantic data instead of domain-specific data formats

to express configurations that can be understood by the different IoT things themselves, and, when a new thing is added to the system, it can question similar devices in the local network to “deduce their own state, configuration, and purpose" [Cha+12]. Similar solutions should be developed to reduce the complexity of configuring things by avoiding the manual process.

Self-optimization focuses on improving the performance and efficiency of the IoT system. The goal is to optimize the usage of computational resources (*e.g.*, optimize energy consumption) and improve response times (*i.e.*, reduce lag), reducing the need to make all the computation happen in the cloud (being distributed among tiers, *i.e.*, fog and edge) [TMD19; Ber+13]. Self-optimization should enable the optimization of computing tasks post-deployment, taking into account several time-varying variables [Ber+13]. Self-optimization is, however, not straightforward, due to the heterogeneity of the devices, highly-specified systems according to the operational domain, interoperability limitations and dynamic nature of the systems (*i.e.*, devices can join and leave the network). Further, considerations need to be made related to the dependency in centralized orchestrator (single-point-of-failure), which points to the need for distributing the decision engine among tiers.

Self-healing focuses on automatically detect, diagnostic, and repair software and hardware issues using recovery and maintenance of health mechanisms. While there is some work done in the field of Wireless Sensor Networks, it focuses on the network level. In IoT systems, some authors propose the use of choreographies that, taking into account several parameters (*e.g.*, response time), readjust the system automatically to mitigate the issue [Seo+17]. While literature about fault-tolerance and self-healing in other application domains has been disregarded for most of the IoT-focused research, existing well-established strategies that have been used for long to mitigate failures in software and hardware systems can be adapted and used [PD11].

Self-protecting systems should anticipate, detect, identify and protect themselves from attacks (both internal and external). Several works have proposed the use of state-of-the-art security and privacy methodologies along with known-to-work “classic” solutions to enable this self propriety. Most work being done focuses on the use of Intrusion Detection Systems (IDS) that continuously inspect incoming network traffic and take actions when suspicious activity is detected [PQW10; Pat+11; Vie+19]. However, due to the enormous threat landscape that IoT systems face (from device compromising to external services security issues) [AH15a], there is the need to continue researching and to provide new self-protection mechanisms at different levels of the systems (*e.g.*, remote lock-and-wipe devices [Rei+16]).

3.6.1 Autonomic Computing and IoT

Although both IoT and the concept of automatic computing have been around for a while now, their application is yet uncommon among deployed computing systems. Nonetheless, several works are showing the potential of autonomic computing (or some of their self-* properties), including in high-complex IoT systems.

Athreya *et al.* in their work suggest that IoT devices should be able to manage themselves both in terms of configuration (self-configuration) and resource utilization (self-optimization), proposing a high-level framework for measurement-based learning and adaptation that allows the system to adapt itself to changing system contexts and application demands (context-aware

adaptation) [ADT13]. They also identify the need for re-programmable interfaces to comply with changing requirements and the need for energy-awareness, but they do not refer to any self-healing considerations.

Angarita *et al.* introduce the concept of responsible objects where IoT things are self-aware of their context (passage of time, the progress of execution and resource consumption) and can apply smart self-healing decisions taking into account component transaction properties (backward recovery and forward recovery features) [Ang15]. Their approach shows limitations due to the dependency on the transaction properties that implies that the system must be continuously making checkpoints before and after specific actions. Further, in critical systems, some decisions are neither retryable nor compensable.

Aktas *et al.* are among the first to propose the use of self-healing mechanisms in the IoT context, proposing the use of runtime verification mechanisms to identify issues on the running system. To do so, they use a Complex Event Processing (CEP) techniques by applying rule-based pattern detection on the events generated in real-time [AA19]. Their approach focuses on the events, caused by the regular system operation, that are non-intrusive, being all messages processed by a predictive maintenance service. However, their work does not provide any insight on how to use the outputs of the runtime verification mechanism to self-heal the system, just relaying an overview of (possible) problems to human operators.

Savaglio *et al.* in their work model, the IoT things as agents and treat the IoT system as a Multi-Agent System (MAS). They suggest that the agent abstraction is suitable to give smartness and autonomy to each thing in the system, making it possible to realize distributed computation autonomically, leveraging the different heterogeneous components. They further present Agent-based COoperating Smart Object (ACOSO) middleware as a way to develop interoperable, autonomic and cognitive IoT systems, validating the resulting system in a simulation environment [For+14; SFZ17; For+18]. However, their implementation depends on heavy computation being done at the thing level, which would not fit most of the low-powered things that compose IoT systems.

3.6.2 Self-healing for IoT

IoT systems have been primarily identified as a core example of a system that must contemplate autonomic components [AH15b; Ang15; Ver+11]. These components — which can range from single devices (*e.g.*, smart locks) to whole systems (*e.g.*, smart homes) — should be capable of self-management, reducing the need for frequent human operation [Kop11]. This becomes even more important in critical systems and when devices are deployed in remote (*e.g.*, wildfire control) or other hard to access areas (*e.g.*, in the user's home).

Some IoT systems are *close-loop* systems. These act based on sensors measurements in order to maintain a predictable output (feedback-loop). Examples are Cyber-Physical Systems (CPS) and some Industrial IoT systems [Bor+17]. Other systems are *open-loop*. These take input under consideration but do not react only based on those inputs (no feedback-loop) [Blo+18]. As a result, making IoT *open-loop* (there is no verification that an actuator performed the required operation) systems resilient is harder than *closed-loop* ones, due to the lack of feedback.

Nonetheless, any IoT system should be capable of reconfiguring itself to recover from failures. A self-healing enabled system should be able to *detect disruptions, diagnose the failure root cause and to derive a remedy, and recovering with a sound strategy* in a timely fashion (*cf.* Figure 2.13, p. 54) [PD11].

The existing approaches for fault-tolerance (and *self-healing*) typically follow a *reactive* methodology where errors are detected and then recovered from, using strategies such as complex event processing, system watchdogs, and supervisors (*cf.* Device Error Data Supervisor [Ram+17]), or a *proactive* (also known as *preventive*) methodology where errors are *predicted* and avoided before faults being triggered using machine learning and other predictive mechanisms (*cf.* Predictive Device Monitor [Ram+17]). A combination of both can also be used [PD11].

Athreya *et al.* [ADT13] suggest devices should be able to manage themselves both in terms of configuration (self-configuration) and resource usage (self-optimization), proposing a measurement-based learning and adaptation framework that allows the system to adapt itself to changing system contexts and application demands. Although their work has some considerations about resilience to failures (*e.g.*, power outages, attacks), it does not address self-healing concerns.

The concept of *responsible objects*, introduced by Angarita *et al.* [Ang15], states that *things* should be self-aware of their context (passage of time, the progress of execution and resource consumption), and apply *smart* self-healing decisions taking into account component transaction properties (backward and forward recovery). Their approach shows limitations, viz. (1) when applied to time-critical applications, as it is not clear how much time we should wait for a transaction to finish, (2) some processes, such as those triggered by emergencies, cannot be compensated, and (3) when is it acceptable to perform *checkpoints* in a continuously running system that cannot be *rolled-back*? It also disregards the typical capability of devices (*e.g.*, limited memory, power) that might challenge the implementation of transactions.

As above mentioned, Aktas *et al.* [AA19] were also among the first to purpose the use of autonomic computing to detect problems at runtime by using CEP. They, however, do not address *self-healing* and only convey a summary of problems or possible problems to human operators. Leotta *et al.* [Leo+18] also present runtime verification as a testing approach by using UML state machine diagrams to specify the system's expected behavior. However, their solution depends on the definition of a formal specification of the complete system, which is unfeasible for highly-dynamic IoT environments (*e.g.*, dynamic network topology).

We could not find any work that focuses on bringing runtime verification mechanisms for visual programming environments. This is not unexpected, as Leotta *et al.* [Leo+18] point out that *"software testing (in IoT) has been mostly overlooked so far, both by research and industry,"* and later corroborated by Seeger *et al.* [SBC20], claiming that most of the research being conducted in visual programming for IoT has been disregarding failure detection and recovery.

3.7 Summary

In this chapter, an extensive revision of the relevant literature for this work is presented. While IoT is a relatively recent research topic, it already exists a large body of knowledge regarding

the lifecycle of IoT systems.

An overview of the state-of-the-art on designing IoT is presented, with a specific focus on patterns of problems and solutions of building this kind of system. The IoT system's construction aspects are also revised by introducing the different development environments present in the literature, their core aspects, advantages, and shortcomings from different perspectives, use-cases, and user expertise levels.

An analysis of the literature focused on testing IoT systems is also provided, systematically analyzing the different available solutions while comparing them, since verification and validation is one of the key aspects to guarantee the dependability of any system, IoT included.

IoT emerges from a combination of several socio-technical aspects, and while most of them fall out of the scope of this work, a brief study on the existent cross-cutting challenges that IoT faces is provided, as these challenges can have an (in)direct impact on the work carried.

A revision of the current literature around fault-tolerance software/hardware systems is done. Given that the literature already has several works that systemize the fault-tolerance knowledge in the form of patterns, we leverage these summarizing works to grasp the most common problems and solutions used at large while contextualizing them from an IoT perspective.

Lastly, a review on the topic of autonomic computing is done, focusing on the works that leverage autonomic computing concepts, *i.e.*, self-*, for addressing IoT issues and challenges.

4 | End-user Automation Survey

Automating IoT systems, including smart homes, is not without its challenges, especially when most end-users have little to no technical knowledge [Ghi+17]. The heterogeneity and number of devices, platforms, and services used in IoT, together with the need for end-users to configure and automate them, require a different approach. While traditional programming (using code editors and integrated development environments) has been the go-to solution for developers and other technical individuals, as the number of IoT application scenarios, environments, and non-technical users increased, it became necessary to build abstractions of sensors, actuators, and whole devices, with additional supporting solutions as a way to reduce the complexity of developing and managing them [BG10; Ghi+17]. This led to the (re)birth of several low-code programming strategies for end-user development (*cf.* Section 3.2, p. 68).

While several authors [Ihi+20; Amm+19] state that these low-code solutions for end-user development still have considerable limitations, they also point out their growth in the variety and the number of users. Thus, it becomes of paramount importance to understand what end-users wish to automate, state their intents, and grasp the users' programming mental models. Knowing this can provide valuable information to future research, allowing researchers and industry alike to find and model their systems' limitations. As far as we could find, there is a lack on the literature of a systematic study on the concrete rules that users would define for smart home automation given a base set of devices and a minimal but realistic definition of a home (*i.e.*, akin to a house floor plan).

> Parts of this chapter were partially based on the master thesis work of Danny Soares entitled MODEL-TO-MODEL MAPPING OF SEMI-STRUCTURED SPECIFICATIONS TO VISUAL PROGRAMMING LANGUAGES [Soa20] and were published in the work PROGRAMMING IOT-SPACES: A USER-SURVEY ON HOME AUTOMATION RULES [Soa+21]. The author's main contributions were on the formal analysis and data curation, visualization, and writing of the published versions of the work.

4.1 Home Automation User Study

Some studies already reflect on the automation rules that end-users program into their spaces [Ur+14; Amm+19; Mi+17]. However, these studies are limited by the number of devices and ways of interaction that the development tool under study supports (which, in most cases, is limited to IFTTT online service [IFT19] due to the easy access to the applet dataset).

A survey was envisioned as the most effective way to gather as many automation scenarios as possible in a timely fashion. The methodology was based on the one presented by Molléri et al. [MPM20].

We surveyed 20 participants for home automation rules given a standard house model and a base set of IoT devices. The study intended to gather as many and as varied home automation scenarios as possible from individuals with different backgrounds and technical know-how while maintaining a certain level of similarity with real-world scenarios and not limiting their creativity and resulting automation complexity.

We proceeded to split the gathered scenarios into categories according to a systematic study of all automation possibilities. This survey also added knowledge on how users typically describe their home automation scenarios using text, allowing us to understand if different individuals use different phrases to describe the same scenarios.

4.2 Methodology

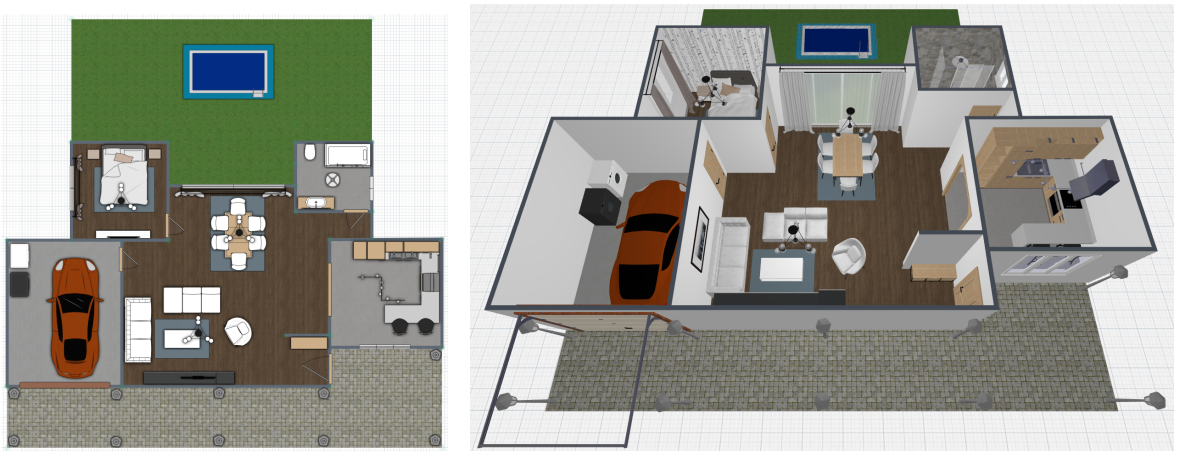


Figure 4.1: 2D and 3D floor plan of the smart house used for the survey [Soa20].

To have a pre-defined, common foundation from where the participants could base themselves to draft their own automation scenarios, we designed a house floor plan and 2D/3D models of it, as shown in Figure 4.1 (p. 124). The house has a total of 8 *spaces*: (a) a *garage*, (b) a *front patio*, (c) a *pool*, (d) a *garden*, (e) a *living room*, (f) a *kitchen*, (g) one *bedroom*, and (h) a *bathroom*.

Along with the home model, we provide a list of smart devices containing various types of sensors and actuators for the participants to use. Namely, across all home divisions, there are the following IoT devices: (1) motion, temperature, humidity, smoke, and air quality sensors,

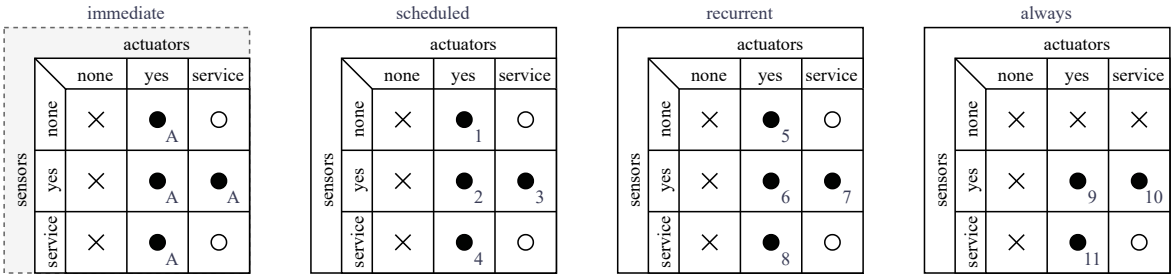


Figure 4.2: Systematic process used to select the categories of the scenarios. All immediate scenarios, even if IoT related, were discarded as they represent actions (A). Scenarios that are not applicable (×), and scenarios that are not IoT related (○) were also discarded. Applicable scenarios (●) were numbered from 1 to 11.

(2) security cameras, (3) controllable lights, (4) controllable windows and blinds, (5) A/C system, (6) robot vacuum cleaner, and (7) sound system.

The (a) *garage* has (a.1) controllable outside and inside doors, (a.2) washing machine, and (a.3) a dryer machine. The (b) *front patio* has only a (b.1) controllable entry door. The (c) *pool* has a (c.1) automated pool cover, (c.2) cleaning system, (c.3) water temperature sensor, and (c.4) water heating system. The (d) *garden* has a (d.1) water sprinkler system, (d.2) soil moisture sensor, and (d.3) robot lawnmower. The (e) *living room* has a (e.1) smart TV. The (f) kitchen has a (f.1) stove, (f.2) oven, (f.3) exhaust hood, (f.4) dishwasher, and (f.5) coffee machine. The (g) *bedroom* has (g.1) a smart TV, and (g.2) controllable bedside lamps. Lastly, the (h) *bathroom* has a (h.1) heated towel rack.

We also allowed and instigated participants to include other devices in their home automation scenarios as long they were available as off-the-shelf (consumer-grade) IoT solutions. No limitations on the interoperability of the IoT system parts nor in the end-user programming interface were defined nor presented.

An online form was picked as a data collection method, given the study's motivation to gather as many automation scenarios as possible while attempting to reducing any bias on the respondent population. The form presented the smart home model, *i.e.*, the house floor plan and the list of available devices. Users had only one open question where they could insert as many scenarios as they wished to without any limitation in size or form.

The survey was then disseminated among 20 participants with different educational backgrounds and ages. All the answers were collected in a spreadsheet, anonymized, and the individual scenarios identified, allowing further analysis.

4.3 Scenarios Categories

To be able to categorize the participant-submitted scenarios, we first need to concretely define the ways by which the end-users interact with their smart home systems (*i.e.*, interaction scenarios). More specifically filtering the ones which of those can be *programmed* (*i.e.*, automation rules), and are, indeed, IoT-related.

We decided to systematically enumerate each possible case — corresponding to a category — by cataloging them according to three different axes: (1) the type of the sensors involved,

ID	PRD	SNS	ACT	EXAMPLE
1	sch	—	YES	Tomorrow, at 11pm, turn on the watering system for 10 minutes.
2	sch	YES	YES	In 2 hours, if no one is in the living room, turn the TV off.
3	sch	YES	SER	In 2 hours, if the door is still open, send me an SMS.
4	sch	SER	YES	Tomorrow, at 9am, if the forecast is sunny, turn on the sprinkler.
5	REC	—	YES	When it is 7am, turn on the coffee machine, the hot water system and kitchen lights.
6	REC	YES	YES	When it is 9am, if the soil moisture is below 60%, turn on the sprinklers.
7	REC	YES	SER	When it is 12pm, if there's mail in the mailbox, send me an SMS.
8	REC	SER	YES	When it is 9am, if the forecast is sunny, close the office and living room blinders, and open the room ones.
9	ALW	YES	YES	When there is no one in the pool for 10 min., cover the pool and turn off the water heater.
10	ALW	YES	SER	When someone rings the door bell, send me an SMS.
11	ALW	SER	YES	When a close windows request is received from Alexa, close the windows.

Table 4.1: List of scenario categories, which take into account their periodicity (PRD) — that can be scheduled (sch), recurrent (REC), or always (ALW) — usage of sensors (SNS), and usage of actuators (ACT) — that can be none (—), yes (YES), or service (SER).

(2) the type of the actuators, and (3) the periodicity of the rule. For the first two axes, we have three possibilities, either (a) there is no sensor/actuator involved, (b) there is a sensor/actuator involved, or (c) the sensor/actuator is actually an external service. For simplicity's sake, we decided to ignore hybrid cases where local sensors/actuators are combined with external services as these are just compositions of different types of cases. Regarding the third axis, rule periodicity, there are four different possible values, (a) it is an instantaneous action (*i.e.*, now), (b) the rule is scheduled to run in the future, in one or more well-defined occasions (*e.g.*, tomorrow at 10 am), (c) the rule has a well-defined periodic schedule (*e.g.*, every Wednesday at 9 pm), and might be subject to a time-frame, and (d) the rule is always active, just waiting for a trigger to initiate it, and might also be subject to a time-frame.

Having this framework in mind, we started writing sample scenarios for each axis intersection (for a total of $3 \times 3 \times 4 = 36$ possibilities). This exercise enabled us to eliminate some combinations straight away: (1) we completely eliminated the *immediate* periodicity as this category only contains *actions* — since we are only looking for *rules* —, (2) we eliminated any combination of external sensors (services), and external actuators (services) as these have little to do with IoT, (3) any combination without an actuator — if nothing happens then there is little sense in creating a rule for it — was also removed, and (4) we eliminated always-on scenarios where there is no sensor to trigger the rule. We ended with 11 different valid scenarios. This process is depicted in Figure 4.2 (p. 125), and examples for the remaining 11 scenarios categories can seen in Table 4.1 (p. 126).

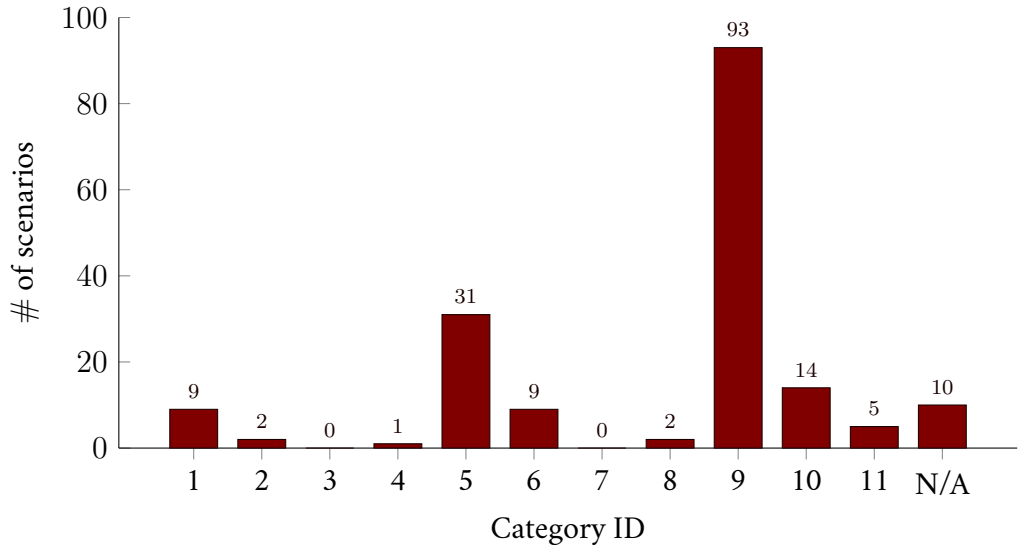


Figure 4.3: Number of automation scenarios per category. N/A stands for scenarios that were too generic, one-time actions or non-categorizable (*e.g.*, hard to understand what the participant wanted to accomplish).

4.4 Results and Analysis

The survey resulted in a total of 177 scenarios. These were grouped into categories according to the categories presented in previous section, being the results' distribution presented in Figure 4.3 (p. 127). We considered all the submitted entries valid smart home automation scenarios, and we were able to categorize most of them (≈94.3%) into one of the 11 defined categories.

With the information collected, the house plan, and devices provided to the participants, we created a **resulting ecosystem** that represents the house with all the devices that the participants used. This ecosystem is replicated in the representation presented in Figure 1.5 (p. 12), showing the house plan with all the devices used by the participants. Some respondents also mentioned using an external weather forecast API and wearables (which are not represented in the isometric visualization).

The scenarios differ (1) in the granularity of application (*e.g.*, with some of them being specific to a house part or domestic appliance), (2) in complexity (rules range from direct triggers to one device to triggers of multiple devices depending on several conditional statements) and (3) in writing fashion (with most of them being close to a conditional programming logic). It is noticeable the usage of Boolean operators (in ≈ 29% of the submissions) to trigger several devices, or to write more complex conditionals (*e.g.*, "If no one is at home for more than 2 hours and the stove is on, turn off the stove."). There are also scenarios (≈7%) were chaining of actions was used to state complementary actions, *e.g.*, "When someone enters a room, turn the lights on. When someone leaves a room, turn the lights off.". We also notice that most rules use at least one, and mostly two, components (*e.g.*, one sensor and one actuator), as depicted in Figure 4.4 (p. 128).

Responses such as *"Intensity of lights based on the available natural light"*, *"Blinds inclination*

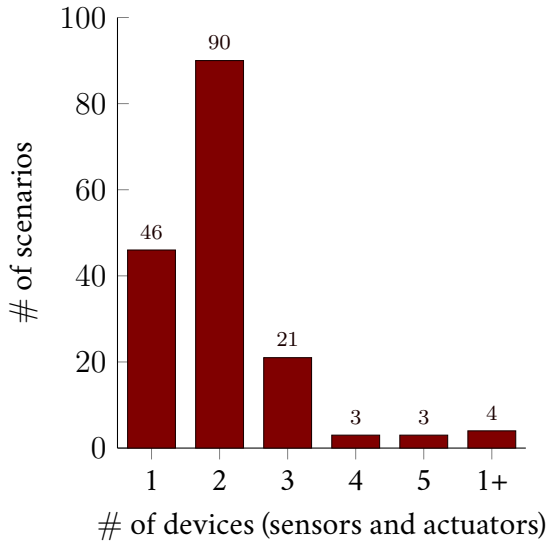


Figure 4.4: Number of scenarios per number of system components in use. We consider each sensor and actuator as a component, even if they are part of the same device (*e.g.*, an A/C system can have both the capability of measuring and adjusting the current ecosystem temperature). The 1+ label refers to scenarios that point to several devices (but at least one).

system based on outside light", *"On schedule turn on the coffee machine"* would need modifications to be closer to a programming-like format to be possible to implement them in usual end-user programming solutions. For example, these should look more like *"When the luminosity in the living room is below <value>, then increase the light's intensity by <increment>"*, or *"When time is 7:00, then turn on the coffee machine"*. These responses (and respective scenarios) are still valid because the information portrayed is enough to understand their meaning and to which category they belong. There are also examples of rules that are too generic, *e.g.*, "Shut down all unnecessary devices", which would require that the IoT system had some degree of contextual awareness, and, even, user preferences (*e.g.*, "Turn the heating system on, set to the preferred temperature, on schedule."), to be able to execute them (≈27%).

The most common way of specifying scenarios is by using the structure "when *condition*[1], then *action*" or "*action* when *condition*" (with the recurrent use of "if" instead of "when"). This is close to the representation commonly used by the widely available TAP services, which also However, some scenarios are depicted differently, mainly for scheduled actions, such as "*action* at *time*", or "every day at *time*, *action*". There are also some example of *loops*, mostly in the form of "*do action until condition*".

Looking at the dataset, we can see that there are home areas more frequently identified. In contrast, others are almost unseen, as depicted in the chart of Figure 4.5 (p. 129). We consider direct mentions all the mentions to specific rooms in the submitted scenarios, *e.g.*, garage or bedroom. All the indirect mentions consist of remarks about certain things that are, typically, only present in certain rooms, namely: washing machine mentions are part of the garage (by the given device list); entrance is considered front patio; lawnmower references considered

[1]For simplicity, we consider a *trigger* as a specific type of *condition*.

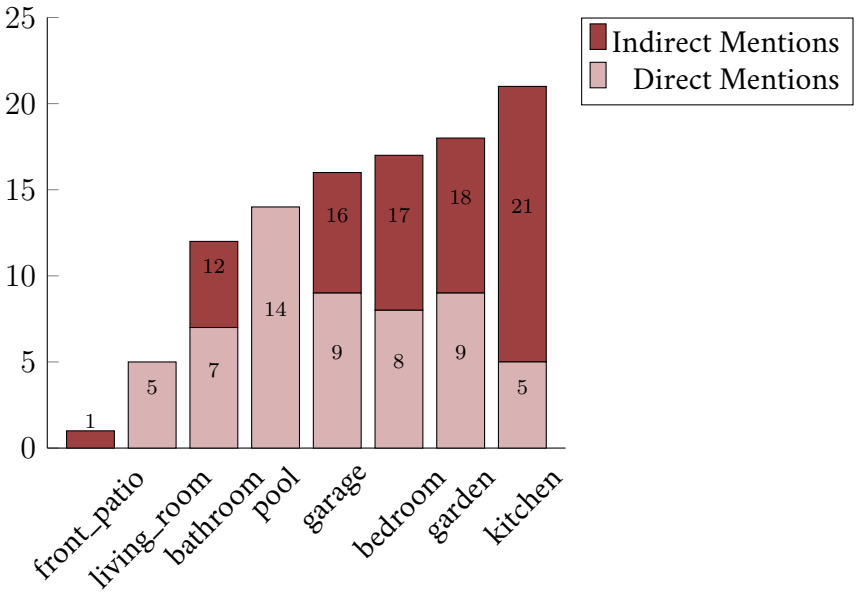


Figure 4.5: The total sum of mentions to specific home parts in the submitted scenarios. Direct mentions consider situations where the surveyed participants directly mentioned a given house part. Indirect mentions include references to certain things that are, typically, only present in certain rooms.

as part of the garden; kitchen includes mentions to the dishwasher, coffee, and oven; alarm clock, waking up, sleep are all related to the bedroom; and all mentions to shower and toilet are considered part of the bathroom.

Users also tend to specify similar (or equal) scenarios using different expressions, granularity, and forms. This was expected since the participants' background (*e.g.*, educational level, previous experience with IoT, age, and the way of expressing ideas) was homogeneous. As an example, "Turn the heating system on, set to the preferred temperature, on schedule.", "Maintain room temperature in between a specified permissible range.", and "Turn on the A/C when the temperature is higher than a given value." transmit the same rule — adjusting the house temperature according to a preference value — in different fashions, either in format or precision. Approx. 28% of the scenarios mention "turn on" actions, and there are 28 direct mentions to *lights*, 23 to *water*, 21 to *blinds*, and 20 to *temperature*.

It is noted the use of pre-conditionals in some scenarios, more specifically, defining some condition that should be met before enforcing the rule, *e.g.*, "With a solar thermal collector (for heating water); when the sun is expected during the following hours, turn off traditional water heating system". The use of macros that aggregate a set of tasks and sub-rules is also visible, *e.g.*, "*Holiday mode*: when any device is used/triggered notify the owner" and "*Garden automation*: stable soil moisture level, temperature stabilization in adverse weather".

Although integration with external services is only mentioned once, there are several rules that, when implemented, would depend on some information provider. For example, we can consider "If the hot water system is based on electricity, heat when electricity is cheaper" would depend on knowing the market prices of electricity. Further, voice control, *e.g.*, "Voice control

over coffee machine/blinds/lights/etc.", is typically accomplished by integrating with a third-party voice assistant such as Amazon Alexa, Apple Siri, or Google Assistant [Amm+19].

Finally, it is also noticeable that some participants already present some degree of concern about the failure of the system parts and the use of IoT to detect them, *e.g.*, "Send SMS alert if faulty freezer/fridge".

4.5 Threats to Validity

For this survey, we have identified some threats that may affect the validity of the results attained.

We asked participants for home automation scenarios and did not give them any **structure for the phrases**, to understand how they would write the scenarios, which resulted in many scenarios being just a brief description and not specific enough. Perhaps, having requested the participants to provide more detailed scenarios would have resulted in more concrete scenarios. However, constraining participants to use a specific format or having certain degrees of detail for the scenario descriptions would not have allowed us to evaluate whether users tend to follow a pattern or typical structure.

Although we let participants use any off-the-shelf device, we provided them with an initial list of devices to pick from. This, together with the specified houseplant, might have introduced a bias into the chosen scenarios.

The **sample size** for this survey was relatively low size, corresponding to 20 participants. Having a larger sample could have resulted in more varied scenarios. This could enrich our analysis and provide more insights into the typical way that users express their automation rules and what these rules typically consist of.

The **level of expertise** of the participants could impact the scenarios provided by them, and, as such data was not collected, we consider it a threat to the conclusions drafted. For example, participants with more experience with IoT should provide more complex and realistic scenarios than participants with no experience in that field. To tackle this threat, we attempted to choose participants with different levels of expertise with low-coding programming solutions and IoT. This resulted in having scenarios from participants whose experience ranged from participants who never used or experimented with home automation, to participants that had already implemented IoT systems and worked with Node-RED extensively.

Even though the participants had different levels of expertise in home automation, the results reveal **little variety in categories** for the scenarios. As aforementioned, increasing the number of participants in the survey would probably, have resulted in more varied scenarios and more categories. Another possible mechanism to mitigate this threat would be to provide a wide-range of example automations, but this could induce some bias in the participant's answers.

4.6 Summary

Notwithstanding the global growth in the number of IoT devices across all fields of application, configuring, programming and managing these systems remains a challenge, especially by non-technical end-users.

Vendor apps designed to help with device configuration and interaction are device/brand specific, thus not offering the flexibility required by a heterogeneous field which use cases require interoperability between different devices and systems. This leads to the existence of low-code programming solutions targeting this need. However, their inherent simplicity limits the complexity of the automation scenarios that are possible to specify. Other, more advanced solutions, such as Node-RED, provides the capability to work with multiple devices/brands in one place. This comes with an extra cost in terms of usability, as these solutions turn out to be too overwhelming and complicated for most novice users. Smart assistants, which are typically more user-friendly, are another common alternative to provide comfortable interactions with IoT systems. But they lack in the ability to provide sufficiently complex interactions, which prevents users from managing complex systems.

In this chapter we present the results of a survey conducted with the goal of collecting home automation scenarios, which resulted in 177 scenarios created by 20 participants. We conclude that the most common pattern used by users to define their automation scenarios shares a similar structure close to conditional programming — "when *condition*, then *action*", or "*action*, when *condition*" — which is compatible with the trigger-action programming model used by several market solutions including IFTTT. This shows that it is intuitive for regular users to describe home automation scenarios in a mostly-structured fashion, easily transposed to a conditional programming syntax. Besides, the users tend to use (or mention) macros and/or aliases that represent more than one device (*e.g.*, a group of lights) or more than one action (*e.g.*, garden control).

Taking into account the available solutions in the market for end-user programming and their programming strategies, we can consider that while most of the scenarios could be easily mapped into TAP rules, the rules that do not follow such model appear as a challenge which is mostly ignored by existing solutions, especially the ones that focus users without specific technical knowledge. In this case, voice assistants can become of utmost importance, allowing users to create automation rules in a conversation, adding complexity by steps instead of specifying everything in one statement or by a diagram [Amm+19; DLF20].

The full dataset, the 3D model of the house, floor plan, and suggested device list are available as a replication package (*cf.* Chapter B, p. 337) to ease the study's replication and allow further analysis.

5 | Research Statement

Building software systems is an arduous task as it can be seen from the CHAOS reports [Sta15] which states that software projects have a challenged rate of 52%, a 19% impaired (canceled) status, and only a 29% success rate, circa 2015. Even with Fitzgerald [Fit12] considering that we are beyond the early *software crisis*, such reports reveal that even if we are now facing another *software crisis*, the *crisis* leads to the same poor success rates. One can identify complexity as the main reason behind the problem with developing software [Bro86].

Within the nature of IoT systems, there are several particularities that, although not new or unique, congregate at a large scale in terms of interconnected devices, people, systems, and information resources, leading to an ever-increasing complexity that affects developers and end-users alike. These systems — typically built with heterogeneous parts, mostly resulting from the integration of different, and, sometimes, already existing, systems (*i.e.*, SoS [Del+13]) — are not only logically distributed but also geographically and, typically, have to deal with power constraints and real-time needs.

In this chapter, we present the main challenges that drive this thesis taking into account the analysis of the literature (*cf.* Chapter 3, p. 64) and the end-user survey (*cf.* Chapter 5, p. 123), while discussing the adequate research methodologies and strategies that provide empirical evidence for our hypothesis.

> A preliminary version of this research statement was presented and discussed at the *doctoral symposium on software engineering* part of the *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*, with the work entitled A REACTIVE AND MODEL-BASED APPROACH FOR DEVELOPING INTERNET-OF-THINGS SYSTEMS [DFF18].

5.1 Emerging Challenges and Viewpoints

The scenarios described in § 1.7 (p. 10) showcase some different ways in which IoT plays a role already or can/will be part of soon, while presenting some different technical challenges that influence different intervening parties, from end-users to practitioners in a mostly-unique

fashion. Although we could consider some concerns as cross-cutting, the ability of each one of the different types of users to address them is, most of the time, limited in several ways.

In an end-user perspective, several challenges appear when the user wants to interact with their system, ranging in complexity accordingly with the number and variety of devices and other ecosystem constraints (*e.g.*, number of household members). From this viewpoint, and in the context of this work, we can enumerate the following as some motivational issues:

1. End-users have, typically, limited technical knowledge thus resorting to the vendors' out-of-the-box tools — that come mentioned in the users' manual — to configure their systems. While these tools (ranging from mobile applications to web dashboards) provide minimal features that allow one to use their systems, they (1) limit the ability to configure complex sensing-acting rules thus limiting the ability to satisfy the system users' necessities if they do not fall in the typical device use cases, (2) lack of ability to verify if the current configuration performs as expected when a condition is met, and, (3) limitations in understanding why something happened in the system *a posteriori*. Further, when the user attempts to control their system with everyday solutions such as voice assistants, the available feature-set is even more limited.

2. While it is common to have electronic spares of some kind around the home (*e.g.*, lightbulbs), it is not so common to have IoT devices *spares* lying around, because they are, typically, more expensive and not expected to fail so frequently. However, these devices and systems can fail due to problems with hardware parts — communication issues, sensors, and actuators malfunctions — as well as software counterparts — broken updates, misconfigurations, and security problems. Due to the reliance on single devices of each kind, the failure of one can render the whole system unusable, only recovering by replacing it. Further, as the complexity of the system increases, understanding what the faulting part was can be a challenge by itself.

3. As the number of devices, integrations, system users, and environment variables increases the ability of the users to reason about the system, the configured automations, and their side effects weaken (or completely disappears), resulting in poorly configured and managed systems that can be troublesome, and, in extreme cases, threaten the comfort and well-being of the house inhabitants.

4. IoT systems heavily depend on Internet connectivity to proper work. The ability for these systems to work as expected can be impacted by (1) the Internet service quality, which varies between locations, and (2) the vendors' Internet-based services that can have disruptions themselves or suffer from disruptions due to problems with the providers they use[1]. Due to these, end-users can be unable to use their systems at all if the vendors do not provide alternative ways to interact or *minimal service* features that work offline.

[1] In some more *extreme* cases, the entity (*e.g.*, vendor) behind certain IoT devices can decide to deprecate them and disable the supporting services (*e.g.*, cloud services), thus making the devices severely limited or, even, unusable.

From a perspective of both systems integrators and power-users — individuals that have enough technical knowledge that allows them not to be limited by the vendors out-of-the-box solutions — a new set of challenges appear:

5. Similarly to what have, historically, happened to other fields of application such as industrial automation and systems modeling, there has been a rise in the use of solutions that leverage high-level abstractions in the scope of IoT as a way of taming the complexity of configuring and managing these systems. Such solutions, although their apparent usage simplicity, have limitations when (1) have to deal with the heterogeneity of the systems and devices, limiting their use to a few supported vendors, (2) the abstractions are *leaky* [Spo04], thus, in most cases, lower-level errors and configurations require users to manually intervene in the system, requiring more advanced knowledge manually, (3) do not provide features to introspect the running system making it hard to impossible to debug them, and, (4) make the systems more opaque (limit observability) and harder to test, *i.e.*, one can only observe misconfigurations when the system is running since these tools lack proper system validation and verification capabilities.

6. Available solutions for configuring and managing IoT systems are typically centralized — one central unit that orchestrates (*i.e.*, configures and manages), and, in some cases, processes the data from the different devices and triggers actuators accordingly while taking into consideration human interaction (*e.g.*, human-in-the-loop). This makes users unable to leverage the distributed nature of IoT systems, leading to wasting local computing capabilities and making it hard to impossible to adapt the system in the case of disruptions (*e.g.*, fault of a central computation node).

7. Most IoT systems and the abstractions used to build them mostly disregard system/device failure situations. The disregard for such scenarios — along with the lack of building blocks to attain these scenarios requirements — makes it hard to impossible for users to configure their systems to sustain failures within the system or with the 3rd-parties they rely on.

As a system designer and/or developer, the aforementioned technical challenges and open issues, ranging from technology fragmentation to lack of standards, are a source of problems that are being addressed differently by practitioners and research mostly without any consensus, leading to several challenges:

8. The vastness of practitioners, researchers, and organizations putting efforts on both evolving and standardizing IoT has created an immense body of knowledge, ambiguous concepts and created several competing standards and technologies that become a major barrier to those who have to pick the best solution (*e.g.*, communication protocols and system's architecture) for a specific case or system.

9. As developing IoT systems depends both on hardware and software considerations, those who design and build these systems need to consider failures on software, but on hardware also.

As there is no *silver-bullet* that could encompass all of these challenges and many others that fall out of the scope of this work, we focus our contribution on the systematization of best practices for design and developing IoT systems dependably from a software engineering perspective, while reducing the users' specific expertise required to configure their own systems, by improving the existent development environments and by leveraging concepts of autonomic computing, such as self-healing.

5.2 A Perspective on Node-RED

Node-RED [Ope19b] is one of the most common (low-code) visual programming solutions, with more than 8800 stars on GitHub (*cf.* Figure 3.2, p. 69) and ≈4600 mentions in research artifacts[2]. It is also open-source and has a special focus on IoT development[3].

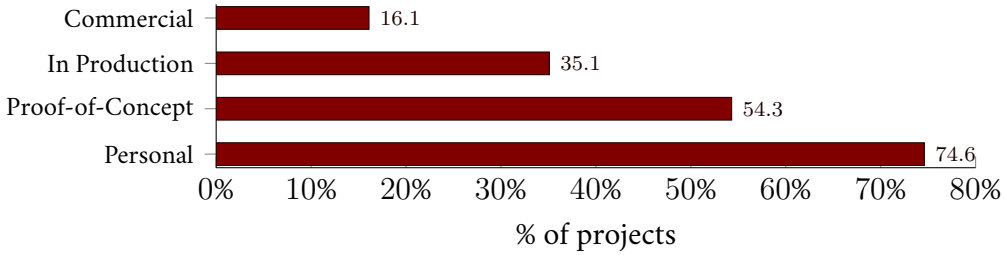


Figure 5.1: Usage of Node-RED by the user community per project maturity stage. Results gather from 871 surveyed participants where respondents could pick all answers that applied to them [Ope19a].

Node-RED vast majority of users[4], as per Figure 5.1 (p. 135), use it for home automation purposes (≈75%) and building dashboards (≈48%), there is a considerable number of users that state that they use it for defining edge device's logic (≈43%), controlling PLC devices (≈24%), and gathering and transform data (extract, transform, and load — ETL workflows, corresponding to ≈7%) [Ope19a]. While this solution was identified and discussed as part of our literature review (*cf.* Section 3.2.7, p. 87), given its role in the context of this thesis — which mostly direct impacts the work carried — a review on some key challenges and limitations is presented in the following paragraphs.

From a technical perspective, Node-RED has several limitations in terms of dependability and computational distribution, mostly due, but not limited, to its centralized architecture. This architecture results in (1) all the computation being performed in one instance (limiting computational distribution), (2) computational heavy tasks will impact the performance of the whole system, and faults in a single *flow* can lead to system disruption, (3) there is no isolation of execution contexts which can raise both security and privacy issues. There is also no out-of-the-box mechanisms for redundancy (multi-tenancy) with several instances of Node-RED.

[2]Considering a search on *Google Scholar* with the query `("Node-RED" OR "NodeRED") AND ("IoT" OR "Internet-of-Things" OR "Internet of Things")` circa November 2021.

[3]Although the tool was initial only focused on IoT development it now claims to be suitable for developing any category of *event-driven application*.

[4]Considering a survey with 868 respondents [Ope19a].

Complementarity, there are some design decisions that pose limitations to the use of Node-RED, namely: (4) the web-based development interface (Node-RED Editor) is highly-coupled with the runtime, limiting the use of alternative development interfaces and making it more difficult to interact directly with the runtime (*i.e.*, lack of an runtime API), (5) there is no support for multi-inputs per *node*, making it harder to define and readjust the behavior of *flows* both during the design and runtime phases, and (6) there are no mechanisms to ensure the structural correctness of the developed *flows* (*e.g.*, types and static analysis) nor to verify if the developed *flows* operate as expected (testing).

From an end-user perspective over the Node-RED development environment, there are a few key missing features that could improve the environment and enhance the end-user interaction. Concretely, (1) the system's internal functioning (*e.g.*, flow of the messages) is opaque, thus limiting the observability over the system operation, (2) lack of debug mechanisms such as breakpoints, (3) labels, annotation, and other documentation features are poor, (4) there is no feature for auto-placing and auto-routing of nodes and resulting *flows* (making the organization of *nodes* and *wires* a manual endeavor), and (5) the visual notations are very similar (*i.e.*, all *nodes* look the same) making it harder to understand the different functionalities of each node.

Although there is a considerable amount of visual programming solutions and, concretely, mashup-development tools, for IoT (even if they are less popular in terms of usage and community) [Ray17; Ihi+20], they are typically limited in ways similar to Node-RED.

5.3 Thesis Statement

As the number of interconnected devices permeating our daily lives increases, their operation becomes strongly tied to our physical surroundings, regardless of the domain of application. The unavailability or malfunctioning of these devices and the systems they are part of could threaten human life, cause environmental damage and significant financial loss, at large scale — as a mostly direct result of moving from mostly-segregated and mostly-independent systems to ever-more connected and dependent ones.

Traditional systems typically consider the existence of non-technical users who interact with interfaces (*e.g.*, applications) behind which users with technical and domain knowledge configure the underlying system and its features. In this view, when new types of interaction requirements are found, the design and development teams go through the development life-cycle (from its early stages) to meet these new user-requested features or modifications.

Applying this development approach in context of IoT systems becomes a challenge due to several aspects, including: (1) the number of technical human resources does not scale to meet all the IoT users' needs — especially when they are user/domain/scenario specific [Mic19]; (2) the tendency for market-solutions to be one-size-fits-all, that do not consider interoperability[5] among different vendors and not specific for each niche scenario [Gui16; Aly+19]; (3) the dynamic nature of these systems, where users keep changing their *automations* and

[5] Some vendors go further limit the interoperability on purpose to lock users to their ecosystem and products (*i.e.*, vendor-lock).

adding/removing a device, making developing using traditional approaches too slow and unsuitable [And+21], and, (4) evolving these systems could imply changes in hardware components which would force individuals to buy and upgrade their systems continuously.

Thus, as current approaches do not scale to the dimension of IoT — in number of users, application domains, and devices —, several solutions have appeared that enable users without specific expertise to configure and automate their systems without requiring technical knowledge (or reducing this knowledge to a bare minimum)[6]. Most of these solutions are low-code development platforms (*cf.* Section 3.2, p. 68), ranging from visual programming platforms to text-based automation rules (some even using conversational assistants). However, these solutions typically limit[7]: (1) what the system users can automate and how these can be carried out (*i.e.*, vendor-specific apps or limited scope of features and integrations) [Aly+19; MHF17], (2) have limited features aiding users with limited technical knowledge to understand what is happening in their systems (*i.e.*, what the automation rules do), and, (3) mostly disregard dependability considerations which lead to a highly fragile ecosystem (*e.g.*, total dependency on an always-on Internet connection).

Additionally, as systems' complexity increases, it inevitably results in people becoming *"overwhelmed by the effort to properly control the assembled collection,"* [PD11] increasing the probability of human-induced errors and failures; developing becomes hard, labor-intensive, and expensive, no matter how *low-code* the infrastructure is [JEC14].

While most IoT development environments limit the possible configurations and integrations that the users without specific technical knowledge can do — thus limiting the complexity of the system and the possible human-induced errors — we believe that by enriching the systems with safeguards and feedback mechanisms (similar to debug), one could improve their ability to configure their systems, including the ability to configure more complex behaviors (*e.g.*, automation that involves several devices, third-parties, and several users). However, we also believe that the dependence on end-users without any technical knowledge to define *dependable* behaviors (and others, such optimizations) is a fallacy since it always require some degree of technical knowledge.

We consider that a balance is needed between development environments that enable end-users without specific technical expertise to define more complex automations, and the requirement of dependability of the resulting system (given that most of end-users do not possess the specific knowledge to define mechanisms from scratch for ensuring such dependability). Several authors have been proposing the adoption of autonomic computing [Hor01; AH15b; Ang15; Ver+11], as a way of enabling systems to self-adapt to their operational ecosystem and current context mostly without technical and manual intervention[8]. Autonomic computing principles, namely self-healing, can thus be used to improve the system's dependability without requiring additional know-how from the end-users. Additionally, the autonomic behaviors

[6] This is a tendency observable in other fields with low-coding application development solutions being a commodity [Won+21].

[7] While these are the concerns that most impact this thesis statement, other considerations, including privacy, should not be discarded by the research community (*cf.* Section 3.4, p. 103).

[8] With the current widespread of IoT systems it is already unfeasible to have (timely) technical human intervention.

built into the system would also provide safeguards on the possible end-user configurations and automation.

Given the aforementioned context, we consider the fundamental research question of this thesis as:

> *What mechanisms should be provided to IoT end-users that enable them to build systems tailored to their own needs without compromising the overall system dependability while minimizing the additional know-how required?*

This question leads us to our main research hypothesis:

> **H:** *It is possible to enrich IoT-focused end-user development environments in such a way that the resulting systems have a higher dependability degree, with the lowest impact on the know-how of the (end-)users.*

As a reference implementation of an *IoT-focused end-user development environments* we will be using Node-RED, as it is one of the most widely-adopted solutions in this scope.

By *enriching IoT-focused end-user development environments*, we consider a two-fold endeavor: (1) from a technical perspective, the environment should provide the fundamental building-blocks that allow a user to address dependability concerns in a IoT system, and (2) from a user's perspective, the environment should provide abstractions and functionalities that improve the user's ability to construct, understand, and evolve their own systems — which include the ability to address their dependability concerns in their specific use case.

By *higher dependability degree* we consider systems that can manage errors — by either recovering or neutralize the impact of errors, and, possible, failures — of some of their working counterparts with minimal disruption of normal service, *i.e.*, without impacting the system users'.

By *with the lowest impact on the know-how of the (end-)users*, we mean that the resulting environment should not increase — and, ideally, decrease — the complexity of achieving systems that perform as the (end-)user requires — even when errors (or, even, failures) occur. By *(end-)users* we are considering all IoT system users that have the necessary knowledge to configure their own systems, with a special focus on the ones that want to configure behaviors for increasing the system dependability.

5.4 Research Questions

With the widespread use of the so-called IoT, new ways to design, construct, deploy, evolve, and maintain these systems have been created, mostly due to the unfitness or incompleteness of existing approaches. However, as these approaches appear, they are not *silver-bullets*, and we are still far behind a widespread consensus on what are the best practices for designing, developing, operationalize, and manage reliable IoT systems.

To enable our contribution to the current body of knowledge, we set ourselves to understand what are the mechanisms and approaches that can be leveraged, adapted, or created that enhance the management of lifecycle IoT system, while providing users with the necessary

mechanisms to make them more dependable. While a focus is given on providing such to users with limited technical expertise, the mechanisms should contemplate enhancements to both end-users and experts.

We consider as the fundamental desideratum of this work to **tackle the current complexity of managing the lifecycle of IoT systems in a dependable fashion** given their inherent characteristics, without disregarding the lack of specific technical expertise of the most common IoT users. We consider that the lifecycle of an IoT system encompasses the design, construction, evolution, and maintenance of these systems from both hardware and software perspectives. While in software-only systems, one can mostly disregard hardware concerns (*e.g.*, cloud computing), in IoT, hardware specifications and operational modes can directly impact the software that runs both in the devices and in higher layers. Given that, we identified five (5) Research Questions (RQs) that guide this research, as follows:

RQ1 **Considering the unique characteristics of IoT systems that makes them complex, how suitable are the existent solutions for the end-user to develop a dependable system?** What makes IoT systems more complex than software-only systems? What are the current end-user-focused solutions to develop such systems? Do they provide any mechanisms regarding system dependability?

RQ2 **Are there recurrent problems concerning the lifecycle of IoT systems, and what are the prevalent solutions that address them?** What recurrent problems can we identify with current approaches? What is the impact of these problems through the lifecycle of these systems? What are the common solutions and best practices that attempt to address such issues?

RQ3 **What can be improved concerning the IoT systems' dependability?** Given the existing literature on developing dependable systems (RQ1), which aspects have been overlooked in their application to the IoT domain? How can the widely used IoT computational architectures be improved towards higher availability?

RQ4 **How can the mechanisms identified in RQ2 be leveraged by the end-users of IoT systems?** Do current approaches provide mechanisms to build more dependable systems? What form would a concrete implementation of such a mechanism take in a way that empowers end-users (RQ1) to build dependable systems? Is it possible to extend the existing development environments in a way that satisfies the dependability needs?

RQ5 **How can the end-user's ability to manage the IoT systems' lifecycle be improved without requiring specific expertise while promoting the systems' dependability?** Given that current end-user development solutions limit their ability to understand the IoT systems, how can we improve these environments with information about what is happening (feedback-loop) and why (diagnostic)? How can IoT development environments be improved to benefit the management of the systems' lifecycle — without requiring specific technical expertise — while promoting the systems' dependability?

Answering these questions will provide us with evidence of the falsifiability of the aforementioned hypothesis. The scientific methods and validation approaches that will be used are presented and discussed in the following sections.

5.5 Research Methodology

As new paradigms emerge, such as IoT, the repercussions spread through the different areas influenced by its creation. This leads to efforts, more or less coordinated, among different individuals, teams, and institutions towards its adoption, in order to create new tools, producing scientific content, improve processes, or generate profits. At the same time, there is a parallel creation of knowledge by the ones working in evolving the paradigm *per se* in mostly disconnected and diverse fashion. The area of software engineering is one that may be regarded as inherently coupled with *human activity*, since as a such paradigm-shift happens, the knowledge generated around it is directly dependent on the methods by which it was obtained.

In order to assess the practice of software engineering, particularly in field research, approaches based on reductionism are a complex activity and sometimes unsuitable; thus this thesis is aligned with a *pragmatist* view of knowledge acquisition, valuing acquired practical assets and observations.

This work follows the Engineering research method[9] — as defined by Zelkowitz and Wallace based upon the discussion on the Dagstuhl workshop on FUTURE DIRECTIONS IN SOFTWARE ENGINEERING [ZW98; THP93] — by developing and test a solution to the thesis hypothesis, concretized by the design and implementation of several proofs of concept which were thoroughly validated using the appropriate methods.

Auxiliary methods are used to empirically gather evidence for some parts of this work. An *historical method* was used during the literature review, since the knowledge comes from the analysis of data that already exists [ZW98].

The work pursued on patterns follows an *inductive method*, where knowledge, as defined by Kohls and Panke [KP10], results from the observation and analysis of existing cases, being based on practical experience and not deduced from theories.

A *controlled method*[10], which consists on collecting "multiple instances of an observation for statistical validity of the results", was also used along this work, mostly consisting of *replicated experiments*, *synthetic environment experiments*, and *simulation experiments* [ZW98].

The concrete strategies used to validate, or, at least, improve knowledge about the questions raised are the ones that seem to be adequate, provided their scientific significance. Essentially, a mix of different strategies was used, namely:

1. User-studies to grasp the current practices and challenges that end-users face when developing IoT systems;

[9] The Engineering method is defined as "Engineers develop and test a solution to a hypothesis. Based on the results of the test, they improve the solution until it requires no further improvement" [ZW98].

[10] The controlled method is the most classical approach for experimental design in other scientific fields.

2. Systematization of widespread, diffuse, and organized empirical best practices through observational and historical methods on both published academic research and widespread enterprise solutions;

3. Feasibility experiments to measure what are the implications, both beneficial and negative, of the proposed approaches, by evaluating one (or more) possible implementation(s);

4. Laboratorial (*Quasi-*)controlled experiments, both by (1) leveraging physical testbeds and simulators to carry out replicable experiments and (2) guided experiments with individual participants to assert both their capacity to carry out predefined tasks and to gather perception metrics.

Carrying some aforementioned validation steps required a realistic setup to test upon. While simulations and other kinds of mocks can be used to grasp how the approaches and implementations would perform in a real-setup, they are limited by their capability to mimic real-world constraints. A physical testbed was defined, implemented, and deployed in the Software Engineering laboratory at the Faculty of Engineering, University of Porto, to attain this limitation partially. The testbed, so-called *SmartLab*, is composed of several sensors, actuators, and data storage and processing units (*i.e.*, servers), resembling a real-world IoT system.

As the goal of the testbed was to test new approaches and ideas, mainly asserting their feasibility, most of the devices were implemented from scratch, giving control both over the firmware and the circuitry. Due to the costs involved in adding more devices to the testbed, it is of small scale[11]. As the testbed is deployed in the laboratory shared by other individuals, these users are part of the testbed and interact — directly or indirectly — with it.

As a complement to this *do-it-yourself* testbed, several *out-of-the-shelf* devices were acquired, which allows better understanding of their functioning, architecture, limitations, and other relevant details. This allows us to compare the testbed with commercial solutions continuously.

5.6 Summary

There are fundamental research questions directly related to the state-of-the-art of IoT systems, the suitability of current software development best practices to them, and how end-users can or cannot interact or adapt their systems to their own needs. We acknowledge that these research questions are transversal to several fields of research that are combined in current IoT systems. Thus, in the scope of this thesis, we focus on tackling the current complexity of creating dependable IoT systems, from a software engineering perspective, without disregarding the lack of specific technical expertise of the most common IoT users. To that end we identified four main goals/questions to drive our research: (1) identify the unique characteristics of these systems that make them complex and how does that affect the end-user ability to build dependable systems, (2) what are the recurrent problems — and solutions — when dealing with

[11] For some experiments, larger physical testbeds — in device number — were used, but they were not deployed as part of the laboratory.

the dependability of these systems, (3) how can those recurrent solutions be adapted and implemented in widely-used IoT development solutions, specifically low-code ones, and (4) how can the users without specific technical expertise interaction with these systems be enhanced in a way that their ability to build and evolve IoT systems is improved, while maintaining or improve these systems' dependability.

Valuing acquired practical knowledge, we are set to answer the above goals/questions while providing empirical evidence that supports those answers. Thus, to validate our claims a mix of different validation methodologies were used, among which are (1) *observational* and *historical* methods for the contributions regarding patterns and pattern languages, (2) feasibility experiments (*engineering* methods) for contributions that depend on various POC and their suitability to the hypothesis in question, (3) user-studies (*empirical* methods) to validate contributions that depend on gathering empirical evidence on the end-users perception over a concretization of a hypothesis, and (4) laboratorial (quasi-)controlled experiments, both using testbeds or simulators, to gather observations for statistical validity.

Part II

Pattern Language

6 | Patterns for Dependable IoT

IoT as a constantly evolving field across application domains have been lead to a widespread creation of knowledge in the most various forms. This knowledge, coming from both academia and industry (and, even, hobbyists), is paramount to the evolution of the field. However, without a proper systematization of this knowledge it becomes an arduous task to understand what are the best or most suitable practices, architectures or methodologies that applies to a specific project, concern, or problem. Patterns and pattern-languages have been used for long as a way of accomplishing such systematization [AIS77; Gam+95]. Initial contributions in the context of design patterns for IoT were published mainly by Reinfurt *et al.* [Rei+16; Rei+17b; Rei+17a; Rei+17c]. The published patterns mostly address the operation mode of IoT systems, including concerns about energy supply modes, bootstraping (self-configuration), interoperability, communication modes and security controls. These contributions, from Reinfurt *et al.* and others, are analyzed and discussed in § 3.1 (p. 65), which, although preliminary and focused only in a subset of concerns, already unify some of the existing body-of-knownledge.

In this chapter we introduce a pattern-language[1] for self-healing on IoT systems. This pattern-language *capture important structures, practices, and techniques that are key competencies in a given field, but which are not yet widely known*, regarding resilience, reliability, and fault-tolerance for these systems, and from an autonomic computing perspective (*i.e.*, self-awareness and self-adaptation). The patterns here presented are further detailed in the following chapters.

> Parts of this chapter were published in the works A PATTERN-LANGUAGE FOR SELF-HEALING INTERNET-OF-THINGS SYSTEMS [Dia+20a], PATTERNS FOR THINGS THAT FAIL [Ram+17], and, TESTING AND DEPLOYMENT PATTERNS FOR THE INTERNET-OF-THINGS [DFS19]. Some parts also appear as part of the subsequent works VISUAL SELF-HEALING MODELLING FOR RELIABLE INTERNET-OF-THINGS SYSTEMS [Dia+20b] and EMPOWERING VISUAL INTERNET-OF-THINGS MASHUPS WITH SELF-HEALING CAPABILITIES [DRF21].

6.1 How To Read These Patterns

The patterns presented in the following chapters are described as *patlets* of *problem-solution* pairs instead of the more structured traditional fashion of patterns. Thus, these become more

[1] A *pattern-language* is defined as the group of related patterns that can be used together to address a larger problem/concern, typically forming an hierarchy or network [Han12; Sei17].

general *guidelines* of design, providing a small insight on how to improve an IoT system reliability through self-healing, but are not prescriptive implementation solutions, mostly due to the wide range of application domains, competing standards, and abstraction levels (ranging from hardware concerns to software issues). Giving concrete implementations of each pattern would force us to drill down to the specifies of the technologies used (and, even, specific proprietary protocols and solutions), operational context, and users' capability to interact with the system. The patterns share the following structure:

Name A meaningful and easily memorable way to refer to the pattern, ranging from a one to two words.

Context and Problem Presents conjunction of influencing factors and settings that result in the manifestation of the problem, giving context to the reader for what were the problem main driver(s). The context limits the range of action of the solution; *i.e.*, a context change might invalidate the solution. It also presents supporting examples for highlighting problem and ends with a formalization of the problem in the form of question. Readers with experience on the field will often relate the pattern context with previous or undergoing experiences.

Therefore Presents a succinct solution to the presented problem (*i.e.*, answers the question), enumerating some solution strategies.

Rationale Discusses the problem and solution taking into account the pattern forces, presenting both the benefits and liabilities of its usage. It also details some of the solution strategies and high-level guidelines for its implementation.

Also see Presents related patterns that address similar problems in other contexts or domains of application and enumerates several usages of the pattern in the wild, showcasing its usage (as per the *rule of three*).

This synthesised format for presenting patterns appear repeatedly in the literature as a <*problem, forces, solution*> triplet (*cf.* Section 2.5, p. 56).

6.2 Methodology

The patterns here introduced were *mined* through an enumerative and eliminative inductive endeavor as presented by Kohls and Panke [KP10]. By enumerative we consider the process of inferring from a number of observed positive cases the properties of new cases — both by (1) extrapolation, *i.e.*, *if a given design worked several times, it will work in this case as well*, and by (2) generalization, *if a given design has worked several times, it will always work in similar contexts*. During the process of extrapolation or generalization we also perform an ultimately eliminative task as *there are no other forces which (in general) influence the observed relations*. Even when new aspects are introduced to the context (which can imply new forces and adjustments), the essential information captured in the patterns do neither miss critical forces nor contain forces that only existed as a detail of a specific situation.

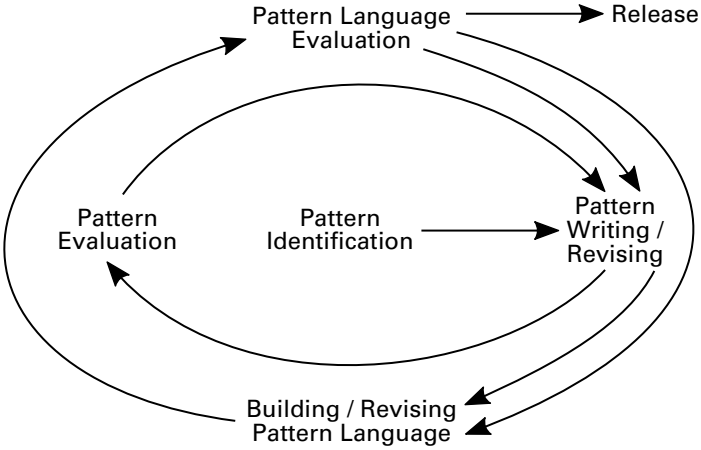


Figure 6.1: Spiral process for elaborating patterns and pattern language, from Seidel [Sei17].

Generally, during the pattern mining process we respect the *rule of three* [KP10], presenting at least three independent examples of the application of the pattern, whether them come from industry or scientific literature.

To provide evidence of the relevance of each *mined* pattern we followed the framework proposed by Seidel [Sei17], evaluating each pattern in terms of: (1) completeness — *is the description complete?* (2) briefness — *does it succinctly presents the context, problem and solution?* (3) validity — *is the solution presented valid and sustained by observation?* (4) usability — *it is easy to interpret both the problem and solution?* (5) feasbility — *does it contain enough information to be implemented?* and (6) impact — *does it have any relevance for the application domain?* The definition of the pattern language followed the same framework. The spiral process of evolution and iteration over the patterns is depicted in Figure 6.1 (p. 146).

The patterns here presented were peer-reviewed by the software engineering community through publication in several of the Pattern Languages of Programs (PLoP) series of conferences.

6.3 Pattern Language

The pattern language hereby presented is composed of a total of 34 patterns. These patterns are grouped into four categories: architectural patterns (4), error detection patterns (13), recovery & maintenance of health patterns (14), and integration and deployment patterns (3). Figure 6.2 (p. 147) maps this pattern groups and the relations between the groups and the IoT architectural tiers, and the remaining of this section briefly describes each category and its patterns.

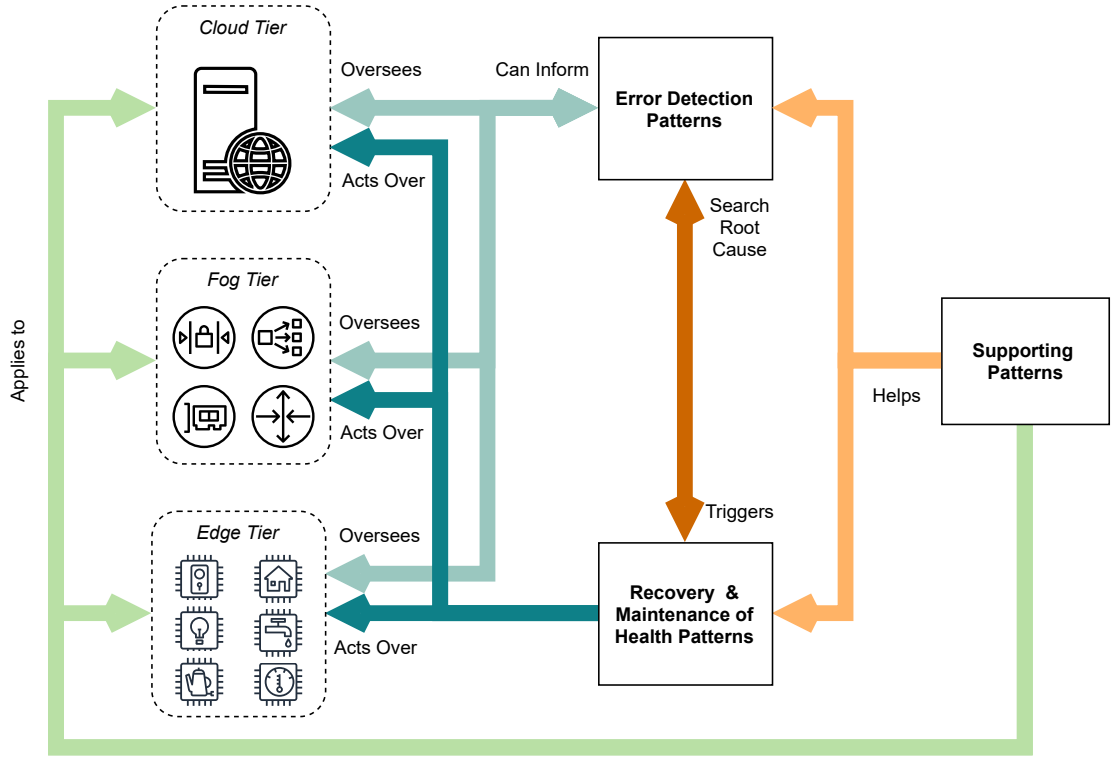


Figure 6.2: Pattern-map of the self-healing pattern-language and their role across IoT architectural tiers.

6.3.1 Supporting Patterns

In this pattern language, we identified patterns regarding observability of IoT systems, namely: Device Registry[2], Device Raw Data Collector, Device Error Data Supervisor, and Predictive Device Monitor. These patterns provide architectural guidelines for certain parts of IoT system, in terms of registry, monitoring, supervision and preventive prediction, and are further described in Chapter 7 (p. 151).

In Figure 6.3 (p. 148) the role of these patterns in a IoT system is depicted. As an answer to the high-volatility common to IoT systems network, Device Registry enables the existing of a common registry to all the devices on the network along with their services, allowing the system to dynamically adjust to the connection and disconnection of devices (keeping track of the devices' identification).

All the different parts of the system produce operational logs (either by the devices themselves — telemetry — or by external probing). The amount of data produced, along with its heterogeneity (as the result of the heterogeneity of the devices themselves) creates an enormous volume of raw data. To keep track of all the data produced by the system parts we defined the Device Raw Data Collector which address the issue of dealing with such volume and heterogeneity of operational data.

As this data is collected, processing it becomes fundamental to make it useful. A common

[2] Device Registry was documented at the same time by us and by Reinfurt *et al.* [Rei+17a].

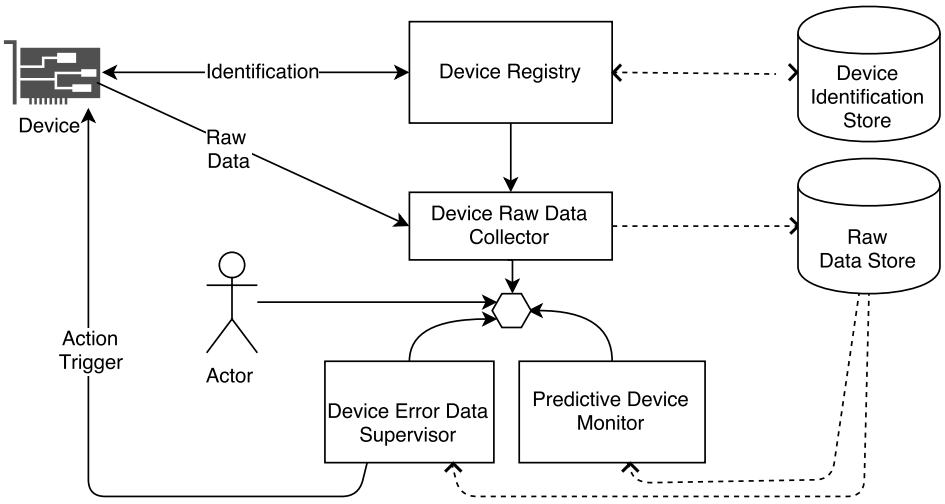


Figure 6.3: Pattern mapping of the four different design patterns identified, including their interactions and relationships.

approaches is to supervise this data in *quasi*-real-time in order to provide information to operators (or other supervisor system components) about the current operation of the system. This is defined as Device Error Data Supervisor, and it enables the processing of the data to trigger errors handling procedures, countermeasures or broadcasting alerts to operators.

Lastly, Predictive Device Monitor enables the processing of raw data not for immediate remedy, but for enabling predictive maintenance based on historical operational data. Similarly to the previous pattern, this enables the processing of data to trigger errors handling procedures, countermeasures or broadcasting alerts to operators about events that may happen in the future.

As IoT systems are not set in stone, they require updates to guarantee their normal functioning or the addition of new features. These updates are required to meet the requirements of the system, thus tested and validated. We, additionally, introduce 3 patterns in this context that reflect on the inner challenges of both validating and delivering updates (deploy) in IoT systems, that are further detailed in Chapter 7 (p. 151).

The first stages of development are done in quick and short iterations during which new features are added and removed until the system (*e.g.*, product) requirements are satisfied. During these iterations, quick feedback is required to assert if the system comply with the requirements. Simulated-based Testing pattern enables this by simulating (and/or emulating) the IoT system's infrastructure and its inputs/outputs in order to test new (or newer versions) of the IoT systems.

Later stages of the development (pre-market validation) can required more real-world data on how the system operates (*e.g.*, capturing unexpected scenarios). Testbed enables the developers to test their systems with real devices and end-users interactions that provides an ecosystem to test new (or newer versions) of IoT systems. Testbed and Simulated-based Testing are complementary patterns in the way they address the issue of validating the system, while gathering operational feedback, at different stages of development.

When a new software version (*e.g.*, firmware) is ready to be delivery to the already deployed

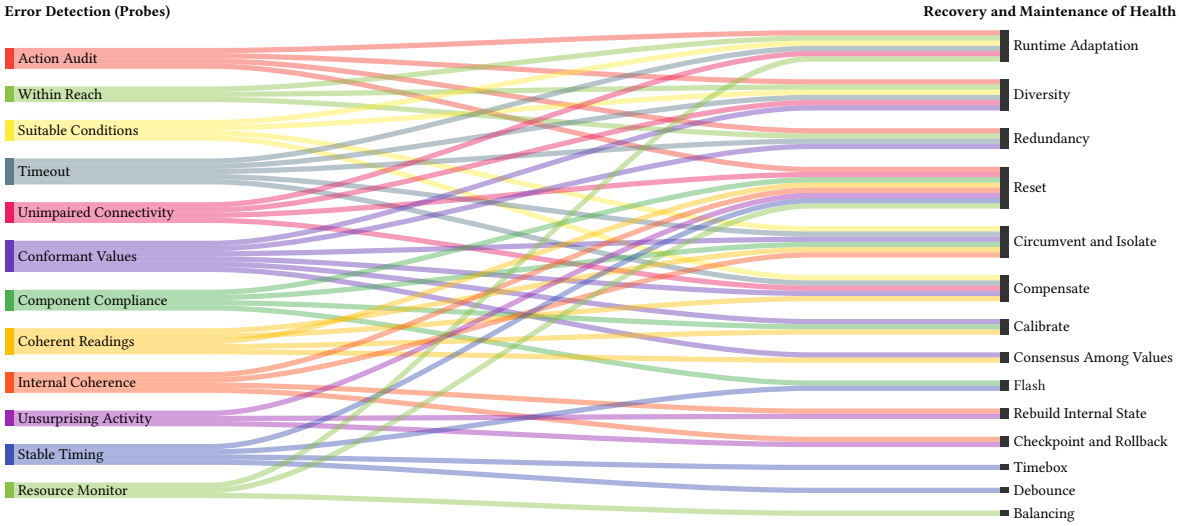


Figure 6.4: The self-healing pattern language for IoT systems. Each *error detection* pattern on the left can identify issues that can be solved by one or more *recovery & maintenance of health* patterns on the right. Although these mostly direct relations between the two pattern categories, the *error detection* patterns can be used together to enhance diagnosis and *recovery & maintenance of health* patterns can also be used together to recover the system to a healthy state (*cf.* Figure 9.1, p. 172).

systems, traditional update strategies (to which patterns have already being discussed in the literature) might not suffice, due to the constraints of these devices (*e.g.*, data transfer rate and no Internet connectivity). Middleman Update addresses this by describing the use of a *middleman* to deliver new software versions to IoT system devices', that is able to establish a link with the target device (*e.g.*, using a low-power communication protocol).

Patterns such as the Flash., Reset. and Runtime Adaption. are examples of patterns that can be complemented by these patterns, specially the Middleman Update, as they can involve the delivery of validated new software versions (*e.g.*, firmware) to the devices.

6.3.2 Self-Healing Patterns

As IoT systems increase in complexity, keeping track of all the moving parts, their behavior, and finding and addressing failures becomes paramount. As these systems seamlessly merge the physical and virtual realms this becomes a crucial concern, from one perspective how to detect and discover root causes of problems, and from another, complementary, perspective, how to enable these systems to autonomously respond and adapt to the operational errors and failures. Autonomic computing (*cf.* Section 2.4, p. 51) defines self-heal as the capability of a system to automatically discover, diagnose, and react to, or recover from, failures. In this section we present two pattern categories, error detection patterns (13), and recovery & maintenance of health patterns (14), which totalize 27 patterns.

The patterns described in Chapter 8 (p. 159) — *error detection* patterns — and Chapter 9 (p. 171) — *recovery & maintenance of health* patterns — are inter-winded and form

a pattern-language by themselves. The patterns in this language are presented in Figure 6.4 (p. 149). The relationships specified in Figure 6.4 (p. 149) are *first-degree* relations which point to possible recovery & maintenance of health patterns that address issues identified by certain error detection patterns. Relationships between the patterns in the same category also exist, and are illustrated to some extent in the following sections.

We hypothesize that several pattern sequences can exist, combining the patterns that are part of the language here presented. On the one hand, several *error detection* patterns are used in sequence to diagnosing a problem (or improve the knowledge about a problem). On the other hand, several *recovery & maintenance of health* patterns can be used in sequence to recover the system to a healthy state. Further, several *recovery & maintenance of health* patterns can be used to maintain the system stable while others (*e.g.*, concurrently) try to recover certain parts of the malfunctioning system.

Most of these patterns can be used at different layers as per the specification given in Figure 2.7 (p. 39). As an example, missing sensor data can be compensated at the perception layer, where the device has some mechanism to fill in the missing values (*e.g.*, average, maximum or minimum in a timebox) or at the application layer, where, with more computing capability (*e.g.*, cloud), data mining strategies can be used to guess the missing values.

Futhermore, these patterns can both be used at an integration level, where the user is building the system by "connecting" *boxes* together and some of these are (or have) self-heal features (*e.g.*, visual programming) [Dia+20b], or at a device/system design and developing level where the logic of the devices, gateways, and servers is coded with these patterns in mind.

6.4 Summary

The lack of maturity of most of the IoT products and services make them at the ideal stage for pattern adoption, since most of these have little to no focus on best practices, specially when regarding the primary focus of these patterns: dependability and autonomicity. While adopting a pattern language can force adjustments to a system architecture and development, namely, in terms of organization, processes, and, even, teams, this pattern language and its patterns allow for iterative (even only partial) integration of them along the development lifecycle, not enforcing a disruptive change.

In this chapter an overview on the pattern language and pattern categories was presented. A total of 34 patterns, part of three different categories, namely: supporting patterns (7), error detection patterns (13), and, recovery & maintenance of health patterns (14). The patterns can be used independently or jointly, regardless of the category they are part of, in order to improve (or at least maintain) the dependability of an IoT system.

The supporting patterns are cross-tier, detailing best practices that can be used at different levels with the purpose of improving the observability, interoperability, evolution, and testability of the system as a whole. The error detection patterns go hand by hand with the recovery & maintenance of health patterns, and can be used to improve the system dependability leveraging self-healing strategies.

7 | Supporting Patterns

As the number of moving parts of a system increases, distributed both logically and geographically, it becomes a requirement to keep a registry of all of them and their operational status. The same rationale applies to IoT systems, as an example by the excellence of such scale and distribution. As the devices and services part of the IoT systems can spread among different computational tiers, large amounts of operational data need to be properly collected and analyzed to provide operators with information about the running system. This information can be used to monitor service disruptions or even make predictions about future needs (*e.g.*, predictive maintenance).

Furthermore, as a mostly-direct consequence of the increasing number of IoT-enabled domains, malfunctions, and bugs of software and hardware faults can have a significant impact in the physical world, *e.g.*, in health scenarios, devices, and systems used are becoming *increasingly complicated and interconnected*, and the mix of manual and machine operations evolved are ever-growing in complexity, and therefore become a risk [Wiz+11]. It is even argued that "we may already reach the point where testing as the primary means to gain confidence in a system is impractical or ineffective" [Wiz+11].

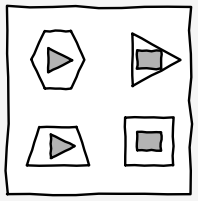
This chapter focuses on these issues, presenting a total of seven patterns. The first set of four patterns focus on maintaining track of what happens in an IoT system. These patterns describe supporting architectural blocks that can be integrated into new or already existing IoT systems, which can improve the observability over them (*e.g.*, monitoring). The three patterns latter describe focus on testing and integration on IoT systems. We believe these patterns are strongly interrelated and may become part of a future pattern language of CI/CD for the Internet-of-Things, a need that was already identified by the Philips Hue team [Bei18].

> This chapter summarizes the contributions present in the works PATTERNS FOR THINGS THAT FAIL [Ram+17] and TESTING AND DEPLOYMENT PATTERNS FOR THE INTERNET-OF-THINGS [DFS19].

7.1 Device Registry

An IoT implementation mostly consists of a group of devices that form a network. It should be possible to leverage the visibility of a set of them to provide more complex features (*e.g.*, the introduction devices whose focus is to process data). Nonetheless, as an administrator of the network, one is aware that one of the main requirements is scalability. Therefore, the devices should be Plug & Play [Yan+15]. Regarding security and control over the network, it should be possible to disconnect the devices from the network if they get hacked, promote mutual awareness, or, in the opposite case, limit the exposure of the devices among each other.

Therefore, build and maintain a registry with all the information about the devices in the system, which scales and adapts as the devices connected to the system can change.



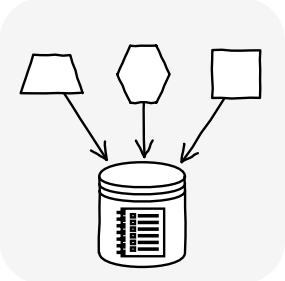
Rationale. Have a single module known and reachable by the whole network of components. This module may be queried for the address of a specific device (or group of devices) having a specific set of features. Also, a device joining the network will have to register itself in this module. In order to leave the network, the procedure is the same. The devices may subscribe to certain events (*e.g.*, registration of devices with a specific capability), so they make the best use of the network. It introduces the following advantages: (1) easiness to scale since registration is made on a single module with some metadata regarding the device capabilities; (2) the subscription of certain events empowers the network to change itself at runtime; (3) a device may only be aware of the rest of the network if registered; and (4) there's a clearer separation of concerns and more awareness of the network capabilities. We also identify the following drawbacks: (1) a single point of failure is introduced, since a single module holding the registry in which the system heavily relies on to properly function; and (2) grouping all the information of the devices in the network in a single module increases the desirability of attacks to it in order to retrieve information of the network. **Also see:** Service Registry [SB08].

7.2 Device Raw Data Collector

Both the raw data collected by the devices and the logs they produce can be the source of a wealth of information about the status of the system and the functioning of the devices. This pattern facilitates the compilation of data and logs from the devices in a singular, known location, from where it can later be extracted and further processed as edge devices in IoT systems are constantly reporting data about their functioning. It is therefore important to have an efficient way to store it consistently. The problem arises due to the fact that the devices are spread geographically, and the information needs to be sent to a remote location due to the limitation of the devices' storage. That is, there needs to be a way to receive, collect and distribute raw data and logs produced by the devices. It may be desired to attach some labels to the data, such

as timestamps. Lastly, the delay between the moment the data is generated, and its storage is crucial in order to allow real-time monitoring by other entities. Nonetheless, ensuring data integrity is also a key point, but it is as important to have security and abstraction layers in order to prevent outsiders from having access to the stored data.

Therefore, use a collector to gather all the operational information of the system, collecting raw data produce by the system, going from device logs to network traffic.

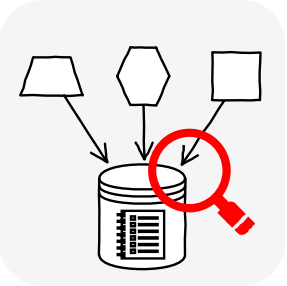


Rationale. Have a centralized collector server responsible for capturing logs and raw data from every edge device. The collected data can later be used for a variety of purposes such as error detection (*cf.* Device Error Data Supervisor), system status analysis, or extracting information from the data. This collector server may exist locally or be located in the cloud, but its location needs to be known to every edge device (*cf.* Device Registry). Each sequence of raw data or log captured should be stored by the collector along with a reference of which device it was emitted by. Additional labels may be attached either requested by the device or by the server. It facilitates the detection of errors in edge devices. Raw data received from the devices can be used as a source of information for posterior analysis. By having the raw data stored, a more powerful device, or even a dedicated one, can process it. It is possible to identify the device from which a sequenced piece of data originates, thus making it possible to analyze individual devices, the whole system, or parts of it. The server described will need enough disk space to store a history of the device data and logs. **Also see:** Raw Data [DeL98], Centralized Data [DeL98], Log Aggregator [BW16], Centralized System Logging [BWK13].

7.3 Device Error Data Supervisor

IoT-based systems are typically constituted by many nodes, highly complex and distributed. This makes the occurrence of errors in edge devices more common due to the inherent heterogeneity as well as the wide-range of deployment scenarios where these nodes can coexist. By collecting information from the data streams associated with the edge devices, such as *logs*, it becomes possible to notice errors that occur in the system. The handling and processing of such errors are fundamental in order to mitigate failures and, consequently, reduce downtimes and other nefarious effects in the deployed IoT systems. Edge devices in IoT systems are susceptible to errors which can affect the reliability of the data collected by sensors and influence the correct functioning of actuator systems. When the errors reported by the devices themselves or edge-device monitoring systems are not correctly handled and processed, this can lead to the appearance of malfunctions in the systems, with consequences depending on the criticality of these systems.

Therefore, use a supervisor to continuously processing data about the devices' operation and trigger advisory or correctional actions when required, actions which can include handling errors with existent mechanism, triggering countermeasures, or broadcasting alerts.



Rationale. Since devices themselves, or even monitoring systems, are capable of reporting information about their current status and activities, and such data is collected, it is possible to continuously analyze and process it in order to detect explicit or implicit errors on the data. Whenever an error occurs, some actions must be taken, as countermeasures, to mitigate or reduce its impact on the IoT system. To be able to offer Device Error Data Supervisor functionalities, the edge devices must be continuously giving feedback about their current functioning status. Such information can be given by the devices or by some third-party monitoring mechanism and can be accomplished by implementing the Device Raw Data Collector. By analyzing the continuous flow of data, we can detect and process errors, enabling its correct handling and activating warning mechanisms. Therefore, we can consider certain benefits on the error data handling, namely, by activating available countermeasures in order to mitigate or reduce the impact of device failures in IoT systems. **Also see:** Device Raw Data Collector, Predictive Device Monitor, Rules Engine [Rei+16], Service Monitor [SB08].

7.4 Predictive Device Monitor

Device maintenance in IoT systems can be considerably costly. The possible geographical dispersion and difficult access to the edge components can negatively impact the time required to repair said components. Thus, it would be advantageous to efficiently, and pre-emptively estimate the *Remaining Useful Life* of an edge component. Maintenance on edge devices is an important operation for improving the hardware endurance over time. Whether it be security vulnerabilities, software malfunction, or hardware malfunction, such problems are usually not detected until after they occur, which results in longer unexpected downtime. In that sense, it would be an improvement to the management of an IoT system to become aware of a possible issue as soon as possible so that actions can be taken to mitigate it.

Therefore, use predictive failure detection techiniques as a way of predicting when and how a device will fail, providing information to carry maintenance actions to maintain the health of the system.

Rationale. Based on the Raw Device Data Collector, through the use of monitoring operations like the ones provided by the algorithms of machine learning and data mining, analyze the data in such a way that allows the detection of abnormal entries in the data records associated

with the devices. These kinds of indicators should be known states or data samples that have a high likelihood of preceding a failure, so the fault prediction will learn the state that precedes an error with every error that occurs in the system. Upon detection of these indicators, the system must be able to warn the user of the likelihood of a malfunction, as well as have a schedule of predictable device failure. It becomes possible to avoid errors instead of simply repairing them after their occurrence; thus, unexpected downtime due to hardware or software faults is decreased. However, the location where the data used for the prediction is stored as well as the component generating that prediction, is centralized. Further, the use of machine learning algorithms may produce false results initially, either positive or negative, before the patterns leading to those errors are learned by the model. **Also see:** Device Raw Data Monitoring, AnyCorrectiveAction Stable [SF10].

7.5 Testbed

When developing software for IoT systems, in resemblance to any other software-only system, there is the need for a testing phase before deploying the system. However, since IoT systems work in the realm of virtual and real-worlds, this testing phase should be performed in real-world-like scenarios, representative of the similar real situations where these devices will operate. Consider a *smart home*, composed of several actuators and sensors, which can be remotely and locally controlled by an end-user or another system (*e.g.*, *smart assistant*). The software that runs on the connected devices can, like any other system, suffer from bugs resulting from the development phase that can potentially negatively impact the comfort and well-being of the end-user.

Therefore, test the system on a physically deployed setup similar to or equal to the final deployment environment and observe abnormal behaviors or failures.

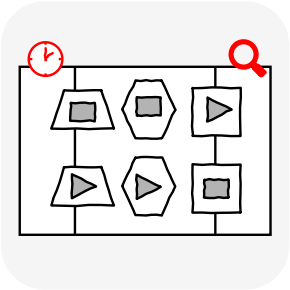
Rationale. Testing IoT systems using the traditional software-only tool, methods, and approaches do not suffice the needs of testing IoT systems in real-like scenarios. This is mostly due to the issue of simulating realistic and complete information about the sensors' data *input* and actuators *outputs*, which limits what can be tested with traditional test methods. The Testbed introduces the following benefits: (1) the observability inherited from using a testbeds facilitate understanding how developers behave and communicate; (2) by having real users interacting with the devices on a daily basis, abnormal behaviors will eventually be identified; (3) system responses to real incidents are captured (*e.g.*, power surge). The pattern also introduces the following limitations: (1) replicating a physical production environment might present an elevated cost, further increase by the need to maintain and evolve it; (2) the feedback about the system operation is slow to obtain, given the time needed for the user to naturally interact with all parts of the system; (3) replicating user behavior requires human intervention, preventing test automation; and (4) it is not possible to replicate all production environments,

preventing the TESTBED from ensuring that the software will always succeed. **Also see:** AUTOMATED TESTS [DGM07], DON'T TRUST SIMULATIONS PATTERN [Ris98], DEVICE RAW DATA COLLECTOR [Ram+17].

7.6 Simulation-based Testing

Each device that is typically part of an IoT system is mostly unique since they are designed and built by several entities, use different communication protocols and standards, and have different responsibilities and features (identified as the *technological fragmentation*) [Gui16] which difficult the process of testing such systems. Further, these devices can be scattered geographically, operating in the most diverse environments. Testing IoT systems become even harder because the corresponding hardware may not be readily available. Consider an automated irrigation system. To assert its performance under diverse weather conditions (*e.g.*, different geographical locations), the system needs to be setup in a multitude of operating scenarios, allowing the developer to capture information about the system's performance under different and changing environmental conditions. In these scenarios, the developer depends on the setup of the system in a real-world setup in order to assert its correct functioning even in the early stages of development.

Therefore, use software to simulate the deployment scenario, including its failures. Test the software in the simulator and observe abnormal behaviors or failures while validating the expected behavior.



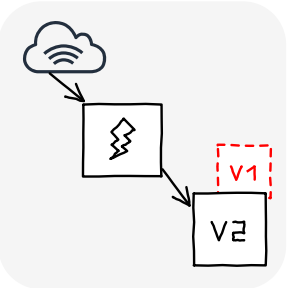
Rationale. Using simulation to run the new IoT system, we can test the normal operation of the system, test extreme cases, and replicate extreme conditions and incidents. We can further easily tune the test scenario by, for example, scaling the number of devices under testing or introduce or remove devices. Further, we can parallel several testing scenarios and rapidly collect information about the tests, and return that feedback to the developer. This pattern is more suitable for too early stages of development since it does not have real-world implications and allows the collection results quicker. It introduces the following benefits: (1) feedback about how the system performs is obtained quicker; (2) tuning the test setup is quick and mostly costless; (3) tests can be performed in parallel with different configurations (improving feedback); (4) tests on edge-cases[1] can be easily performed; and (5) hard to reproduce scenarios can be simulated (*e.g.*, fire). The pattern also introduces the following drawbacks: (1) abnormal interactions are not reproduced in simulations since they are not known at the time of the test specification (*e.g.*, no human-in-the-loop); and (2) real-world device failures are hard to reproduce in simulated setups. **Also see:** AUTOMATED TESTS [DGM07], DON'T TRUST SIMULATIONS PATTERN [Ris98].

[1] An edge case is a problem or situation that occurs only at an extreme (maximum or minimum) operating parameter, usually corresponding to an *unlikely or weird situation* [NBB14].

7.7 Middleman Update

IoT devices have different communication mechanisms which influence the mechanisms and processes needed to update their running software versions to newer versions. The most common form of updating edge devices is the use of serial port[2], and, in this case, there is a need for a manual effort to update each device. Consider a *smart watch*, which is connected to a mobile phone by Bluetooth and uses an application to interact with it. As such, when a new software version is available, the *smart watch* is not capable of connecting directly to the remote update server, always depending on the *middle-man* application. This application is responsible for checking for a new software version and delivering the update to the watch. In this scenario, the watch is not able to download its own updated software version due to the connectivity constraints (it does not have a capable Wi-Fi connection to do so).

Therefore, update the running software version of a device by establishing a connection to a middle-man that delivers the new software version from a remote update service.



Rationale. The update process of edge devices becomes a responsibility of the devices closer to them — fog devices — with higher connectivity facilities and storage. These devices work as a *middle-man* which checks for new software versions from the remote update server, and, checking the edge devices' software version, delivers the new artifact and manages their update procedures. It introduces the following benefits: (1) distribute the responsibility of managing the edge devices fleet software update procedures from an update service to devices closer to the edge devices; (2) allows the use of low range and slower data transfer protocols (*e.g.*, Bluetooth Low Energy); and (3) having a *middle-man* device with higher computational resources than the edge device allows improved verification of the update packages from a *security-wise* viewpoint. The pattern also introduces the following drawbacks: (1) dependency on a *middle-man* can introduce a new vulnerable point (*security-wise*) in the update chain; and (2) the update of edge devices can suffer a delay due to the dynamic topology of the network (*e.g.*, the fog device can be disconnected from the edge device at the time that the update is made available). **Also see:** Edge Code Deployment [Qan+16], Remote Deployment [DGM07], Continuous Deployment [DGM07], Deployment Manager [SCF15].

7.8 Summary

This chapter introduces seven patterns that can be used in IoT systems to improve architectural and operational aspects. More concretely, the first four introduced patterns can be used to improve the observability over the IoT system. These patterns enable the capture and process of

[2]Serial port refers to an asynchronous communication interface that transmits data one bit at a time, mostly used for embedded device communication and programming [Axe07].

information about the running system that can be used either by the system administrator or by the system itself to control the delivery of correct service and, possibly, improve its functioning.

The last three patterns focus on strategies to verify and validate the correct functioning of the system (compliance with its specification) and provide mechanisms that allow any deviation to be mitigated even after deployment (*i.e.*, updates and upgrades).

8 | Error Detection Patterns

Enabling IoT systems with self-healing capabilities require systems capable of detecting errors and failures. In this work, we consider error detection the process of detecting issues (both errors and failures) on the running system that can make it enter in a *degradation state* or *defective state*. An error can be in one of two states: *detected* if an error message/signal indicates its presence, and *latent* if undetected. Error detection can be threefold: (1) identifying that there exist an error somewhere in the system, (2) identifying an error root cause, and (3) trigger reactive measures to recover and maintain the system health [Han14].

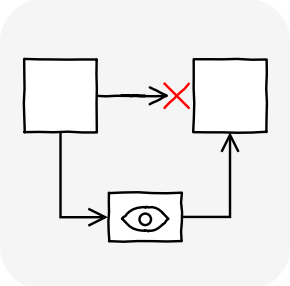
The patterns described in the following paragraphs are not recurrent solutions to problems *per se*, but recurrent solutions on *how* to detect different types of operational issues in a system, mainly focusing on one or more parts of the three-part process of *error detection*. The application of these patterns in a system enables it to provide information, mostly at runtime, to *recovery & maintenance of health* mechanisms (*cf.* Chapter 9, p. 171). Most of the *probe* patterns here presented can be enhanced their diagnostic *precision* by using other *probes*, potentially improving the diagnostic of a specific malfunction or unexpected behavior. Nonetheless, each *probe* has a well-defined target error; thus, they are sufficient to detect the error they target (but each can have limitations in finding the root cause).

> Parts of this chapter were published in the work A PATTERN-LANGUAGE FOR SELF-HEALING INTERNET-OF-THINGS SYSTEMS [Dia+20a] and in the subsequent works VISUAL SELF-HEALING MODELLING FOR RELIABLE INTERNET-OF-THINGS SYSTEMS [Dia+20b] and EMPOWERING VISUAL INTERNET-OF-THINGS MASHUPS WITH SELF-HEALING CAPABILITIES [DRF21].

8.1 Action Audit

In an empty home, a smoke detector has been activated; two alarms were triggered in response: one turning on a loud siren, and the other messaging the homeowner. There is a real possibility of a fire raging on, and time is of the essence. The system automatically attempted to perform these actions, but sometimes things do not go as planned. Maybe the message never reached the owner; maybe the siren was broken; maybe both of these things happened. Some actions are critical, and when they fail, countermeasures must be taken. Comfort, and even well-being, can depend on the proper functioning of components. How to guarantee that required actions are triggered when needed?

Therefore, implement a mechanism that validates each action. The siren makes a sound, so it should be audible by a noise sensor. The message can request human acknowledgment via a reply. If these action checks fail, one can resort to alternate pathways to mitigate potential issues, like triggering a light strobe or directly calling 911 (*cf.* Runtime Adaption, Circumvent And Isolate), or try resetting the device (*cf.* Reset).



Action Audit

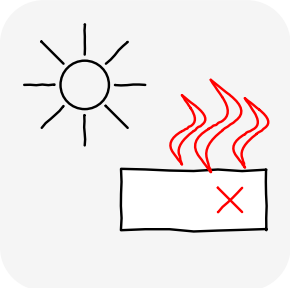
- Component Compliance
- Conformant Values

Figure 8.1: Diagnostic enhancement tree for the Action Audit pattern.

Rationale. *Smart home* applications and their underlying platforms take a *fire-and-forget* approach when issuing commands to actuators. The mistake lies in the assumption that everything will work between the triggering of a command and the intended purpose, *e.g.*, the message was not corrupted, the communication layer is working, and the hardware is not faulty (*cf.* Figure 8.1, p. 160). From a control-theory perspective, IoT systems could learn to benefit from a closed-loop approach in which, based on observational feedback, commands are not simply assumed to have worked until the desired effective state is achieved [Ter16; PLF18]. However, performing such checks might need additional infrastructure that allows one to probe the effect. Depending on the nature of the action, this might be accomplished through different hardware (the noise sensor in our example) by performing a reverse computation to check if the output matches the input constraints. [Tor00]. **Also see:** Acknowledgment [Sar02; Han07], Test Alerts [PLF18], Test Actions [PLF18], Indirect Response Check [RK15], Check Physical Response [RK15].

8.2 Suitable Conditions

It is a particularly hot day outside. So hot that one can fry an egg on top of any of their devices. The outdoor temperature sensors point to an average temperature of 41 °C, but, mysteriously, despite its A/C unit being blasting cold air for the last two hours, the indoor garage sensor still points to an abnormally high temperature of 38 °C. Maybe the window is broken, but the shattering glass sensor did not trigger. On further inspection, the specification of this particular temperature sensor state that the *Recommended Operating Conditions* over operating free-air temperature range goes from −10 °C to 50 °C[1]. How can the system or its parts became aware that they can operate correctly?



Therefore, monitor if surrounding conditions are suitable for device operation, usually known as *operation thresholds* or *recommended operating conditions*. If values start getting unacceptable close limits, there is a likelihood that the device could stop working, or (even worse), malfunction[2]. Take preemptive actions to mitigate potential failures (*cf.* Runtime Adaption, Circumvent And Isolate).

Rationale. Most commercial Integrated Circuits (*i.e.*, *chips*) have recommended operating conditions that do not perform well below the water freezing point but go as far as 70 °C. The reason most IoT systems disregard this problem is that devices are assumed to be placed indoors. Nevertheless, this assumption can still be broken in a cascading scenario such as the one presented above; 70 °C is not unheard of if the device is behind a glass window with sun shining on it. Checking for environmental constraints is a relatively straightforward process, and most IoT devices must state their operating conditions in the manuals for FCC or CE approval [CH11]. The capability of flagging a device in an *unreliable state* allows mitigation strategies to be preemptively triggered. However, this depends on having external data sources (independent, and redundant, sensors or third-party services) that provide the data that allows one to detect if some device is operating in ideal conditions or not[3]. **Also see:** Dependability requirements [Boa+16], Environment-aware communication protocols [Boa+16].

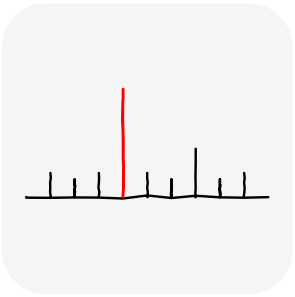
8.3 Reasonable Values

Things tend to act as expected. Even unexpected events usually present a pattern. An outside luminosity sensor is expected to follow a curve according to the sun's position; unless there are clouds (or a solar eclipse). Temperatures do not drop from 20 °C to 0 °C in the blink of an eye, and then immediately recover back; there is expected inertia to it. Humidity exhibits reasonable gradients; unless someone is taking a shower in the bathroom. All these situations present *reasonable* patterns of readings one is expecting from sensors. If the readings do not fit these patterns, then they might be untrustable.

[1] *Recommended Operating Conditions* are detailed descriptions of the conditions in which the device performs as expected is typically documented in the device's user manual or datasheet.

[3] Some approximation can always be made if the read values by the device itself, *e.g.*, humidity, are too close to its the operating limits

Therefore, consider the reasonableness of the readings before blindly accepting them as valid values. Use different checks (or a combination of them) depending on the particular sensor to detect unreasonable situations. There will always be a degree of confidence in this assessment, that can vary from *suspicious activity* (spikes in luminosity), to outright *impossibility* (readings outside working intervals). Once they are detected, different strategies can be employed to deal with erroneous values adequately (*cf.* Runtime Adaption, Circumvent And Isolate, Consensus Among Values, Compensate, Calibrate, Reset).



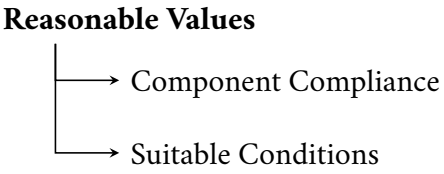


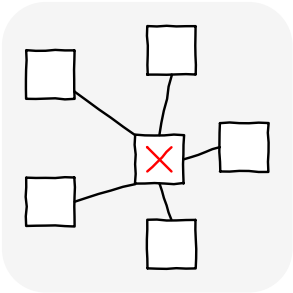
Figure 8.2: Diagnostic enhancement tree for the Reasonable Values pattern.

Rationale. Something that deviates from what is standard, normal, or expected, is usually called an *anomaly*. There is a whole sub-field of computer science and mathematics called *anomaly detection* that might employ sophisticated algorithms to search for situations that deviate from normality, and the data itself defines what establishes as normal [CBK09]. Notwithstanding, there are other scenarios where normality is well-known: reading intervals and physics are two significant information sources. In these scenarios, specific strategies might be employed to assess the *degree of confidence* in the values, up to the point of *unreasonableness*. Once we identify these situations, mitigation techniques might be able to extract a *workable* value by filtering out the noise; otherwise, isolation techniques could flag the devices as *unreliable* (*cf.* Figure 8.2, p. 162). One should note that operating outside the recommended conditions might lead to unreasonable readings, but the reverse implication is not true. **Alias**: *Plausible Values*. **Also see:** Test Periodic Readings, Test Triggered Readings [PLF18], Mean and Variance, Correlation, Gradient and Distance from other readings [Ni+09], Realistic Threshold [Han07], Complex-Event Processing (CEP) [PK19].

8.4 Unimpaired Connectivity

An edge device is attempting to send its reading back to the server (*i.e.*, the message recipient), but the server is not answering back. Depending on the reading's importance, the value might be discarded or saved to be resent later. However, memory is not infinite, and the urgency of the message might require immediate action. If the failure remains, the device might be forced to decide a secondary course of action (*i.e.*, Diversity). How to ensure that the different entities in a system are alive and can communicate with other parts?

Therefore, start by checking if the infrastructure supports the intended connectivity by attempting communication to a secondary target, but through the same connectivity tissue. This can be done by any system part (*e.g.*, a coordinator trying to access a device or a device trying to reach a third-party service). If the message recipient is on the cloud, ping a different known server on the Internet can ping another edge device if it is local. If it is concluded that the communication infrastructure is defective, try resetting it or using another one (*cf.* Runtime Adaption, Reset); if you can communicate with other devices through that same medium, look for the problem elsewhere (*cf.* Within Reach).



Unimpaired Connectivity

└→ Suitable Conditions

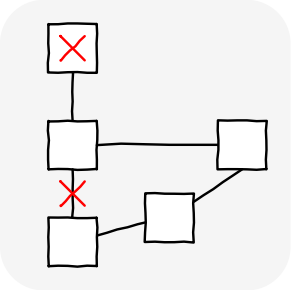
Figure 8.3: Diagnostic enhancement tree for the Unimpaired Connectivity pattern.

Rationale. A simple diagnostic (ping to a secondary service) might discard a connectivity problem. Knowing the difference between the two conditions (the message recipient being down, and the connectivity tissue malfunctioning) might allow the system to take different actions; if the connectivity is down, one might consider using an alternate radio to find the recipient (*e.g.*, GSM, Lora or ZigBee). A typical example would be a fog device that triggers a rule that would make an alarm go off via a Wi-Fi connection. Additional checks (*e.g.*, Action Audit) reveal that the order was not fulfilled. Most devices in the local network fail the heartbeat checks, and attempts to connect to other cloud-based services and edge devices are failing. Switching to a secondary radio protocol (*e.g.*, 433MHz) might allow the intended goal to be carried (*cf.* Figure 8.3, p. 163). **Also see**: Dead Spots (*RF holes*) [Ady+04], Locate disconnected client [Ady+04], Performance Isolation [Ady+04]

8.5 Within Reach

Edge sensors that report frequent reading, such as temperature ones, are usually working continuously, and the system can readily observe that they exist because of their constant message throughput. Edge actuators, like alarms, might only actuate in rare circumstances. The rough confidence that a device will not fail to operate when needed (in this case, disregarding failures of the device mechanical parts) is directly proportional to how well (and often) previous communications were successful. How can one know that some system part is available and responsive when required as it is designed to idle most of the time?

Therefore, if two devices are going to trade messages infrequently, establish a way to increase the confidence in their communication, forcing them to communicate event if to demonstrate that they are operational (*cf.* Runtime Adaption).



Within Reach

- Component Compliance
- Suitable Conditions
- Unsurprising Activity

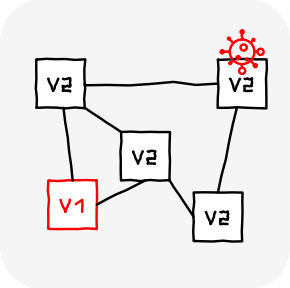
Figure 8.4: Diagnostic enhancement tree for the Within Reach pattern.

Rationale. There are (broadly speaking) two types of connectivity checks: (1) a deliberate, scheduled broadcast of connectivity is called a Heartbeat [Han07; Elo+14], and it usually occurs from devices (edge) to servers/nodes (cloud/fog); (2) a point check of connectivity is called a Ping, and it usually occurs in the opposite direction of a heartbeat, *i.e.*, from the servers/nodes (cloud/fog) to the (edge) devices (*cf.* Figure 8.4, p. 164). The first is mostly used to preemptively capture potential connectivity failures before action is needed (*e.g.*, a failed heartbeat from a siren might imply the system entering a warning state). The second is mostly used as a diagnostic mechanism to find out if the device is out-of-reach or in an abnormal state. Several mechanisms can be used to meet these *alive* checks, such as the periodic broadcast of status messages or push/pull of telemetry data between system parts. However, one must consider that in low-power solutions (*e.g.*, battery-powered devices), forcing the devices to make themselves *alive* when it is not needed can have a drastic impact on their battery-life (*i.e.*, devices that support *deep-sleep* will drain more energy due to the more frequent power-cycles). **Also see:** Acknowledgment [Han07], Are You Alive [Sar02], I Am Alive [Sar02], NACK [Com+15], ACK [Com+15].

8.6 Component Compliance

Software is not set in stone, which is the most significant advantage of programmable things and their ultimate curse. We expect things to behave and perform the same unless physical tearing and breakdown occur. Nevertheless, the software can also tear and breakdown, both through usage and time, as well as through malicious intentions or users' configurations. Devices also gain new capabilities, have their configurations changed (*e.g.*, Reset), have their vulnerabilities patched, and their bugs fixed via software updates. Moreover, these can happen more frequently, over-the-air (OTA), and unassisted, with recent advancements. How can we be sure

that the devices are executing the software they are expected to be running while complying with the current system configurations (interoperability)?

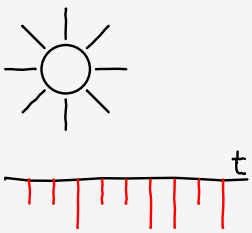


Therefore, check if a particular device is running what it should in the way it should, by frequently observing their software versions, configurations, firmware hashes, and any other checksum mechanisms. This check can be carried out by different system parts (*e.g.*, a gateway that periodically checks the edge devices versions for updates and checksums for corruptions) or by devices' self-checks (that can detect *modifications* at runtime).

Rationale. Several reasons can render devices running unexpected software, namely tampered devices (detected via integrity checks), newer versions (detected via update checks), known vulnerabilities (detected via audit checks) or misconfigurations (detected using different types of configuration checks [Le+06; SNS15]). Typical recovery actions include factory resets, reboots (*cf.* Reset), firmware re-installs, re-configurations, updates and downgrades (*cf.* Flash). In some situations, where recovery is not possible, contingency actions must be done (*cf.* Circumvent And Isolate). Ensuring the correct use of this pattern depends on having entities (*e.g.*, servers) that provide the latest stable software packages along with verification checksums[4]. These also should have security standards that enhance the confidence of the checks. **Also see:** Firmware Integrity Assurance [Al-+18], Update [Al-+18], OTA upgrade [Gan16] and OTA downgrade [Gan16], Middleman Update [DFS19], Protocol Version Handshake [Elo+14].

8.7 Coherent Readings

Sometimes, a fact comes to your attention that, although entirely plausible, you know it is probably wrong. Not because of its absurdity *per se*, but because you have other evidence contradicting it. Sensors are the same; sometimes, readings might be perfectly fine on their own, but when confronted with values coming from different sensors, they are not. For example, it is expected that multiple temperature sensors inside the same environment to report slightly different values. Still, when one of them communicates a wildly different one, something must be wrong. How can one ensure that the readings are faithful and that inaccurate/incorrect readings are flagged as such?



Therefore, compare values from different sources, and check if they are in accordance so that erroneous readings can be detected (*cf.* Redundancy, Diversity, Consensus Among Values). If one sensor is consistently reporting widely different values, maybe you should try resetting or calibrate it (*cf.* Reset, Calibrate), or maybe just stop trusting it altogether (*cf.* Circumvent And Isolate).

Rationale. By crosschecking values coming from different sources, we can detect problems that might not be apparent in any other way. Multiple sources must report approximated values

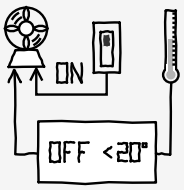
[4]Nonetheless, these checks can have some intermediaries, *i.e.*, Middleman Update [DFS19]

if they are deployed in similar conditions. Even if the sources are entirely different (*e.g.*, humidity and rain detection), inconsistencies can still be inferred (detecting rain, while the humidity sensor reports a dry environment would be a strange occurrence). Nonetheless, a sensor that provides *unexpected* values consistently can point to an abnormal situation, *e.g.*, if a fire starts in a home division, only the sensor deployed there will be triggered. **Alias**: *Consistent Readings*. **Also see:** N-Version Programming [CA78; Tor00], Fail-Stop Processor [Sar02], SICO First And Always [Ada+98].

8.8 Internal Coherence

Actions are being performed in your system; *e.g.*, turning switches on, regulating the A/C and the windows blinds, activating the irrigation system… All these actions lead to changes in the system, whose state is usually mirrored internally (*i.e.*, instead of continually asking if a switch is *on* or *off*, one usually stores the latest known state). Sometimes, though, devices act on their own (*e.g.*, due to a reset or human intervention) and change the state without (or failing to) informing the rest of the system. As an example, consider a light power switch that can be controlled by (1) manually toggling the switch, (2) a mobile application, and (3) a configured light-sensing trigger action. Depending on message delays, packet losses, system reboots or, even misconfigurations, the system can enter an inconsistent state, where it no longer knows the state of the lights and can lead the user to make incorrect decisions. The problem increases when there are more configurations beyond a simple on/off state, such as using the same example, light colors/temperature. How can we make sure the internal representation of the system reflects its actual state?

Therefore, perform regular checks of the system's internal representation when possible, making sure that it correctly mirrors the actual devices' state. This is specially important after a Reset or a Flash and might require the device to Rebuild Internal State.



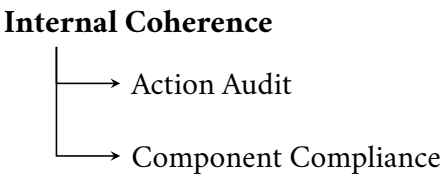


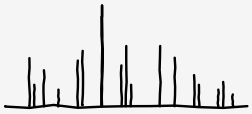
Figure 8.5: Diagnostic enhancement tree for the Internal Coherence pattern.

Rationale. The maintenance of an internal representation of the system exists for several purposes, the most common one being *performance* or to avoid constantly querying a device about its status. However, this assumption that changes in the system are always successfully reported (*cf.* Figure 8.5, p. 166). Any small overlook in reporting can easily create inconsistencies between the physical setup and its *internal representation*, which might eventually cascade in the

decision process leading to a degraded/defective state. **Alias**: *Internal Consistency*. **Also see:** Device Registry [Ram+17], System Monitor [SK09], Resource Discovery for Fault Detection [Zho+15].

8.9 Stable Timing

Your devices are continuously talking with each other, sending messages to inform about a particular value, or asking another device to execute a specific action. However, timing is everything. Two messages in the wrong order are enough to leave the system in a defective state; delayed readings can be the difference between taking the appropriate action in time to prevent damage or the information no longer being relevant. How can we detect data that is not arriving on time, on an irregular basis, or in the wrong order?

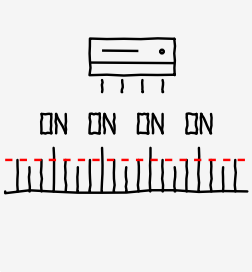


Therefore, have mechanisms that can detect if devices are sending messages at the expected intervals, and taking the expected time to respond to the messages they are receiving. If they are not, you can try to reset them (*cf.* Reset), use other sensors (*cf.* Runtime Adaption, Circumvent And Isolate), or mitigate the problem (*cf.* Debounce).

Rationale. Timing can be critical in IoT systems, and system degradation might cause devices to start taking more time to act upon the messages they are receiving. These delays can cause mischief. Sometimes the time between a reading and a device carrying the appropriate action can be the difference between preventing a fire or irreparable damage, sometimes two steps in the reverse order might be the difference between a vital switch staying on or off. **Also see:** Data-Driven Synchronization [BGJ17], Bubble Razor [Foj+12], DFix [Li+19].

8.10 Unsurprising Activity

Home is where you know how to set the thermostat just right, and you expect that perfect temperature to be maintained indefinitely. That should be easy enough, turn the A/C off if it is too hot, and the heater if it is too cold. At first glance, these two simple rules might seem obvious enough, but they hide a serious problem. As the temperature fluctuates around that desired temperature, a futile and power-hungry battle between the heater and A/C rages on, with each one taking turns trying its best to push the problem into the other's hands. How can this type of suspicious activity be detected?



Therefore, check if any device is sending a suspicious number of messages, as that might indicate severe hardware or logic problem.

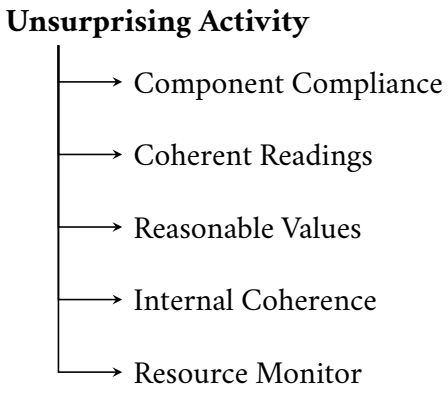


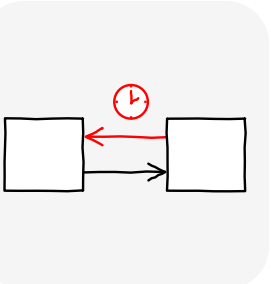
Figure 8.6: Diagnostic enhancement tree for the Unsurprising Activity pattern.

Rationale. The inappropriate usage of a device might degrade its lifetime or result in an unexpected behavior. Such usage can result from a damaged or malicious device or entity that would continuously ask the device to perform the same operation (*cf.* Figure 8.6, p. 168). By monitoring the messages being sent to a device, and establishing a reasonable usage restriction to it, it would be possible to identify misuse patterns, informing the *recovery & maintenance of health* mechanisms in place (*e.g.*, Circumvent And Isolate and Reset). **Also see:** Blacklist [Rei+16], Whitelist [Rei+16].

8.11 Timeout

You want to trigger an action and be sure that it executes within a given time frame. If it does not, an error must have occurred. For example, you want to alert a homeowner that they did not enable his home alarm, despite being out of the house. If the alarm is not enabled within 20 minutes, you want to call him to ensure that they are aware that the alarm is disabled. How can one be sure that a particular action is executed?

Therefore, keep a timer running since the first action and observe if a reaction happened, if the timer runs out without the reaction, an error has occurred. The root cause can range from a device issue, *i.e.*, Action Audit to a network disruption *i.e.*, Unimpaired Connectivity, Within Reach.



Rationale. When a device requests an action from another, the time it takes for the action to be completed might be critical (*cf.* Figure 8.7, p. 169). If the action fails to complete within the acceptable time frame, the triggering device should be able to trigger an alternate action. For that, it should run an internal timer, during which it observes if the action has concluded. When the timer runs out, if the desired reaction is not observed, an alternate action is triggered, to which a Timeout can be used as well, working as a *maintenance of health* mechanism. **Also see:** Limit Retries [Han07; Elo+14].

Timeout

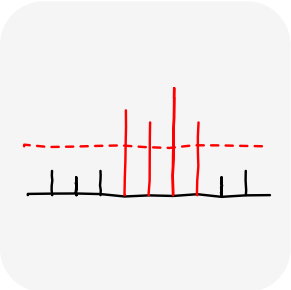
- Component Compliance
- Suitable Conditions
- Unsurprising Activity

Figure 8.7: Diagnostic enhancement tree for the Timeout pattern.

8.12 Conformant Values

Devices can sometimes have faults in their hardware (and, sometimes, in their software) that can lead them to behave out of their specification. For example, sensor devices that use percentage values in their reading can not output negative value nor values above 100%. How can we assure that sensors are operating accordingly to their manufacturer specification?

Therefore, check if the device readings are in conformance with the device reading thresholds, which are stated in the device specification.



Conformant Values

- Coherent Readings
- Reasonable Values
- Component Compliance
- Suitable Conditions

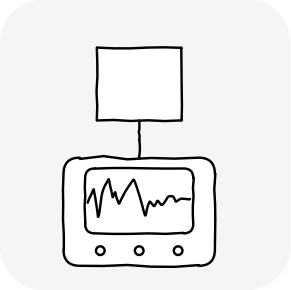
Figure 8.8: Diagnostic enhancement tree for the Conformant Values pattern.

Rationale. Most common hardware present in IoT devices is low-cost and has, typically, a high-proneness to failure. One of the possible outcomes of failure can be, for example, a sensor producing values that are not part of their specification (*cf.* Figure 8.8, p. 169). In such scenarios, the system should be able to distinguish between valid values or out-of-spec ones, allowing *recovery & maintenance of health* measures to be put in place (*e.g.*, Compensate and Circumvent And Isolate) **Also see:** Threshold Check [Tor00], Reasonableness Check [Tor00].

8.13 Resource Monitor

Entities in IoT system, ranging from low-power devices to powerful servers, have constraints such as processing power, storage capacity, bandwidth, among others. Spikes in system usage can lead to malfunctions and severally impact the QoS. The need appears to oversee the resources that the system is consuming both in real-time and historically.

Therefore, monitor the system resources at all times, ranging from battery levels to network operation and resource usage, providing insights on the system's bottlenecks and related issues.



Rationale. System need for resources varies during its operation. As an example, a *smart home* system can sit mostly idle during the time that the house is empty; however, as inhabitants arrive home and interact with the system, the resource usage can spike. While most cloud-based systems can scale resources on-demand, the devices spread around the house (*i.e.*, fog and edge devices), which are typically resource constrained (*e.g.*, processing power), can easily provoke issues in the system (such as QoS degradation). Thus, resource monitor can both provide insights on current issues on the system and on potential issues, allowing them to perform actions accordingly (*e.g.*, Balancing). **Also see:** External Monitoring [Sou+18], Resource Monitoring [Mar05].

8.14 Summary

This chapter introduces a total of 13 error detection patterns, *i.e.*, probes, that focus on checking the health of the system and its parts. These patterns can be used individually or combined as a way to better understand the errors found (*i.e.*, root cause analysis). The information and events collected by the probes can be used by trigger (and inform) recovery and maintenance of health mechanisms to ensure the delivery of correct service (*cf.* Chapter 9, p. 171). These patterns are contextualized within the scope of IoT systems such as *smart home*, *smart farming* and *smart cities* ones. The relationships between the different *probes* are presented in the form of diagrams that appear together with the patterns *patlets*.

Given the highly heterogeneous nature of IoT systems, which affects both hardware and software components, the difficulty in adopting these patterns can vary greatly. The adoption of the supporting patterns introduced in the previous chapter (*cf.* Chapter 7, p. 151) can ease the adoption of some of the error detection patterns detailed in this chapter.

9 | Recovery & Maintenance of Health Patterns

If errors are detected in a system, enabling the system to *self-heal* requires *recovery & maintenance of health* mechanisms. These mechanisms act per the *probes*, reacting to the information reported by them (*i.e.*, if a probe detects an error recovery or maintenance of health mechanism should be triggered, avoiding system degradation or, even, recovering from a defective state).

The following paragraphs delve into the patterns that correspond to the common *recovery & maintenance of health* mechanisms. These mostly rely on the information provided by the error detection mechanisms described in the previous section; thus, *probes* and *recovery & maintenance of health* mechanisms work in tandem to enable a system to self-heal. Most of these patterns can work together to restore the system to a healthy state. As in previous chapters, *smart spaces* such as the *smart home* presented in Chapter 1 (p. 1) are recurrently used as a motivational scenario. The following paragraphs describe the pattern of the *recovery & maintenance of health* (right side of the Figure 6.4, p. 149).

> Parts of this chapter were published in the work A PATTERN-LANGUAGE FOR SELF-HEALING INTERNET-OF-THINGS SYSTEMS [Dia+20a] and in the subsequent works VISUAL SELF-HEALING MODELLING FOR RELIABLE INTERNET-OF-THINGS SYSTEMS [Dia+20b] and EMPOWERING VISUAL INTERNET-OF-THINGS MASHUPS WITH SELF-HEALING CAPABILITIES [DRF21].

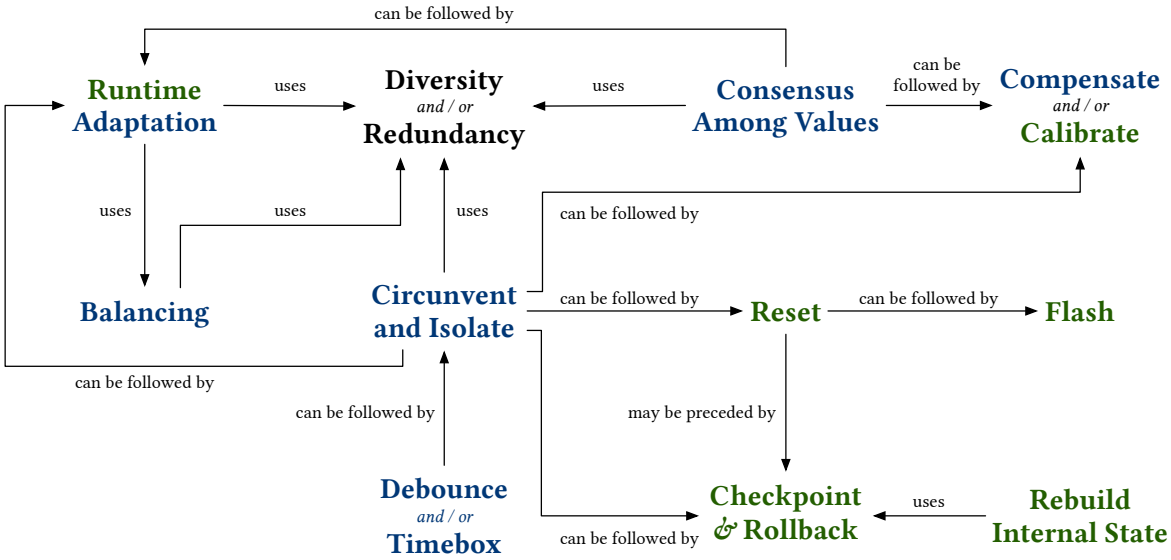
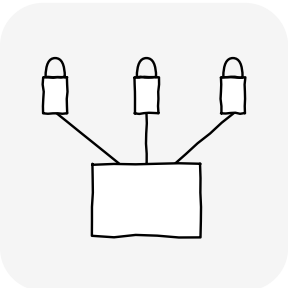


Figure 9.1: The *recovery & maintenance of health* patterns and their sequences of actions (service restorations) towards normal state. Some patterns provide the foundations for others (no color), others work as maintenance of health (blue-colored) and, finally, the remaining work as system recovery mechanisms (green-colored).

9.1 Redundancy

Things are prone to fail, both in hardware (*e.g.*, power-spikes) and software (*e.g.*, corrupted software update). Even when the different system entities report what seems to be reasonable values (*cf.* Reasonable Values), there is no assurance that it is the real value, since there is no comparison point. In cities, if the air quality control was made by only one sensor, there was no way to distinguish a strange reading (*e.g.*, due to a broken sensor or by a spike provoked by a heavy-duty vehicle passing by) if there is no other record to compare with. How can we ensure that the system provides *correct* service at all times?

Therefore, use redundant mechanisms to achieve the same goal, allowing one to both make decisions on which report to believe in, or to trigger the same action using another way.



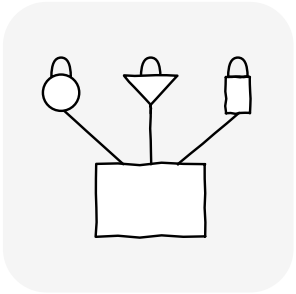
Rationale. In more sensitive scenarios, there is a need to deploy redundant units (*i.e.*, redundancy in space) that can report the same measurements or trigger the same actions to make the system survive to partial failures into account the extra costs. Such redundancy can be achieved by deploying similar or equal sensing or acting devices, communication channels (*e.g.*, different radios), processing units (*e.g.*, microservices), third-party services providers (*e.g.*, using both Amazon Alexa and Microsoft Cortana), and even different power-sources to support the running system (*e.g.*, solar and batteries). In sensing scenarios (*e.g.*, environmental) it

can be possible to use *redundancy in time* which consists on taking several measurements in a time-window and only report the most *correct* (*e.g.*, common) reading (*i.e.*, dropping outliers), viz. Reasonable Values. **Also see:** Failover [Han07], Remote Storage [Han07], Passive Replication [Sar02], Semi-Passive Replication [Sar02], Semi-Active Replication [Sar02], Active Replication [Sar02], Different types of wide-area networks [Ter16], 1+1 Redundancy [Elo+14].

9.2 Diversity

Having multiple components, such as light bulbs, in the same space for the same purpose have the side-effect of acting as redundant components when one of them fails. However, this is tightly coupled with the cost of the device; more expensive objects, such as A/C units, are usually deployed in a minimum-required number in the same environment. How can we improve the recovering and maintenance of health capabilities in a system where some components are not redundantly deployed?

Therefore, use different entities to achieve a common goal and reduce the impact of faulty parts. There are sometimes *alternative* ways of achieving a common goal without adding new entities to the system. During the daytime, a way to compensate a broken light can be to open the windows' blinds and let in sunlight. Lowering the temperature can also be achieved by opening a window instead of turning on the A/C.

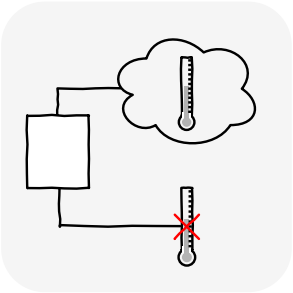


Rationale. Redundancy *per se*, *e.g.*, triple modular redundancy, is rare in IoT systems due to the associated costs for mostly non-critical tasks. Nevertheless, alternative (and possibly already existing) mechanisms can be used to accomplish tasks that are not part of their primary functions, reducing the systems' *points of failure*. Diversity does not need to be applied only to devices; mechanisms such as communication channels (*e.g.*, Wi-Fi, Bluetooth, 433Mhz) can also be the target of diversity, mitigating the effects of Unimpaired Connectivity. Nonetheless, adding diversity to the system will increase its *complexity*, thus possibly impacting the system's overall cost, maintainability, and understandability. Some authors have been proposing the idea of *automatic workarounds* to leverage the already existing diversity (and redundancy) on the system to recover from failures [CGP08]. **Also see:** Circuit Breaker [KHA17], Design Diversity [Avi+04], Automatic Workarounds [CGP08], Protocol Switching [RR12].

9.3 Runtime Adaption

Devices typically have different operating modes which are enabled in different setups or operation conditions. Consider the example of a lightbulb that is connected to the main hub using ZigBee; if there is no hub present, the bulb can be controlled by a dedicated protocol-specific remote control. Another example is the case of Wi-Fi-connected *smart plugs*; the first time that they are turned on, an Access Point (AP) is created (or a Bluetooth connection) that allows the

user to enter the home Wi-Fi credentials, and then the *plug* connects itself to the home network. However, after things are configured once, the devices typically do not employ fallback measures. In the *smart plug* case, if there are disruptions on the Wi-Fi service, the user will not be able to access their smart home system, even if changing the *smart plug* to AP-mode would temporarily fix the issue. Since IoT systems typically rely on low-cost components that might be deployed in harsh environments, which increase the likelihood of *soft-errors* (*e.g.*, intermittent erroneous sensing data or issues on communication) [Cha+18], fallback strategies can be crucial in preserving system operation. How can the system deal with different system degradation (partial failures) while still providing services to the user?



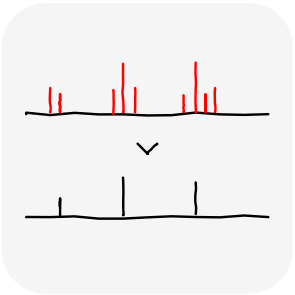
Therefore, enable the system to adapt during runtime, allowing the system to use different infrastructure seamlessly (physical or virtual, *i.e.*, Avatar or Digital Twin [Rei+16]) during operation. When the usual infrastructure recovers to a healthy state, the system must change back to the usual channels [PD11].

Rationale. There are several devices/services that already have the built-in capabilities to provide services even when facing some degree of service degradation. However, typically, these capabilities are not taken advantage in a resilience perspective. The *soft-errors* can allow the system to continue operating but can require a certain amount of adaptation to the runtime conditions. If the device is low on battery, its transmission power (*e.g.*, Wi-Fi) can be impacted. However, changing to a more low-powered communication protocol can allow the system to continue operating for more time without human intervention. Further, if a device cannot connect to the communication infrastructure, it can create its own AP for giving users the ability to control or get data from the device. Similarly, if the system depends on a device to report *e.g.*, environment temperature, if there is a failure in receiving data from that device, the system can fall back automatically to a different source (*e.g.*, third-party weather service). Several solutions employ the concept of Avatar or Digital Twin, virtual representations of devices that, beyond making the bridge between the virtual and real-worlds, are capable of some adaptation such as using other physical units in case of detecting errors or simulate the behavior of the real counterpart if none is available (*cf.* Compensate). It must be considered that these adaptations can compromise the system's capabilities (*e.g.*, data-rate), which may not be viable in certain scenarios. **Alias**: *Dynamic Binding, Reconfiguration*. **Also see:** Circuit Breaker [KHA17], Reintegration [Han07], Device Shadow [Rei+16].

9.4 Debounce

A new temperature sensor was added to the *smart home* to improve the control of room temperature (with more precision than an already existing sensor). However, the cooling system began to present unpredictable behaviors from time to time. The sensor might have been broken; yet, replacing it with a new sensor unit did not solve the issue. Further investigation, mostly by reading the cooling system user manual, showed that the system was trying to collect sensing

data from the sensing device at a rate above its sensing period, which lead the cooling system to misbehave (since it had no data to decide upon). How can we meet the device operational constraints without compromise the system operation?

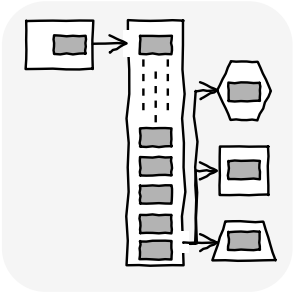


Therefore, filter or aggregate events to meet operational timing constraints, ensuring that the target device receives (or is able to collect) data at the expected frequency. Optionally aggregate them with an average, maximum, minimum, or other strategies to provide the device's relevant data.

Rationale. Both sensors and actuators have several operating constraints, such as power supply, accuracy, and operation range. Among those constraints, there is some that limit how many times a device can trigger a certain action, namely, how often readings can be *collected* from a sensor, and how many times or with which frequency an actuator can be triggered, namely: (1) sensing periods and (2) mechanical/electrical life and operation/release time. If a system does not respect these parameters, in the case of sensors, it can result in *undefined behavior* (*e.g.*, ranging from failures to collect values to collecting random data). In the case of actuators, this can reduce the life-spawn of devices, or even, having hazard repercussions. However, in most cases, humans will not notice if the system *delays* the triggering of a device (or delay the report of a value) a second or so (*e.g.*, any delay will possibly lead users to keep pressing the `ON` button until the lights turn on, *cf.* Timebox) [Ter16]. Thus, the system must implement mechanisms that debounce events to meet the system's operational constraints. While these issues are often dealt with at system development and testing phases, end-users can be impacted by the nonexistence of such mechanisms when upgrading their systems. **Also see:** Queue For Resources [Han07], Request Delay [PD11], Protective Automatic Controls [Han07], Shed Load [Han07], Slow It Down [Han07].

9.5 Balancing

A street access control sub-system (part of a *smart city* system) has variable load. While the system is mostly idle during the night and weekends, during the work-days, the system is at its usage peak (*e.g.*, due to commuting). During this time, other hardware in the *smart city* might be sitting mostly idle. How can we ensure that the system is responsive at all times?



Therefore, distribute software and load between available resources to meet service demands. Do so by abstracting the underlying hardware and distributing the computational units automatically between the available hardware.

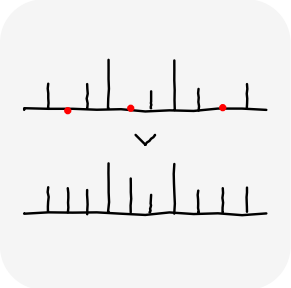
Rationale. The processing demands of a system can change rapidly due to peaks in usage (*e.g.*, access control or increase in the home inhabitants' activity) or the appearance of heavier

computational tasks (*e.g.*, surveillance video processing). However, even during peaks, there can exist parts of the system that are idle (or, even, available redundant parts, *cf.* Redundancy) that can be leveraged to meet the system usage demands during peaks (returning then to normal operation when they are not further needed, *cf.* Runtime Adaption). **Also see:** Share The Load [Han07], Protective Automatic Controls [Han07], Orchestration Manager [Bol20].

9.6 Compensate

A dehumidifier was brought to eliminate musty odors and prevent the growth of mildew in the house. However, to avoid having it turned on at all times, an additional humidity sensor was brought that provides information that allows controlling when and for how long the dehumidifier is active. However, in several situations, the dehumidifier was not turning on due to issues with its sensing counterpart (due to, as an example, Unimpaired Connectivity or Stable Timing). How to ensure that devices perform as expected even when there have some operational problems in their sensing counterpart?

Therefore, have mechanisms that can compensate missing or erroneous information, at least during an established period of time.

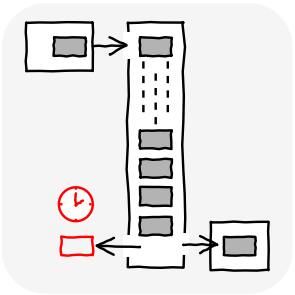


Rationale. While the system operation is best when all of its parts are working correctly, the malfunction of one part (*cf.* Reasonable Values, Unimpaired Connectivity and Suitable Conditions) should not jeopardize the entire system. In these scenarios, mechanisms should be put in place to compensate for the missing information, allowing the system to keep operating. In the case that there is another source of data that can be used (*cf.* Runtime Adaption and Redundancy), they should be adopted. However, when none of these alternatives exists (or are available), strategies such as using an average of the last measurements can be considered and used [Avi+04]. However, the confidence in this calculated values lowers as the time passes by, thus, additionally, considerations about a *graceful degradation* should be taken (*e.g.*, by setting and using a default — reference — value) [FSM12]. **Also see:** Kalman Filter [Lai+19], Interpolation and Correlation [SMG19; Zho+15], Linear and Non-Linear models [Zho+15].

9.7 Timebox

Having components that are both manually controlled *on-site* and remotely (*e.g.*, using a mobile application) leads the user to expect the same type of behavior in both. However, while operating devices remotely, several constraints can slow the *feedback-loop*, such as network lag. This can lead to the user to keep sending the same request (*e.g.*, turn on the sprinklers) repeatedly until the application reflects the request (*e.g.*, showing a message informing that the sprinklers

are on). However, this behavior can provoke malfunctions (even failures) in the system. How to prevent repetitive and similar actions (user or system-induced) in a short period from damaging the system?

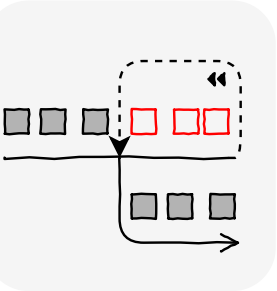


Therefore, only process an order in a specific period that respects the system operational constraints, filtering (*e.g.*, dropping) the remaining requests within the timebox.

Rationale. When a state change request is made, sending the same request repeatedly to the make the action happen *faster* does not have real effects (since the system will always take some time to change to the requested state). However, if this behavior is not controlled, it can compromise the system; thus, similar or equal requests made within a pre-defined timebox should be discarded. Even if the system is required to respond to all the requests (or if the requests are different), it can only go as quickly as the system operational thresholds (*cf.* Debounce). **Also see:** Limit Retries [Han07], Limit Number Of Retries [RK15].

9.8 Checkpoint

A *smart home* system can keep information about several aspects, such as lights state (on/off), presence (detection of motion in the last minutes) and last device activation's (*e.g.*, vacuum cleaner robot). However, in the case of a general system restart (by some error or on purpose), the system will act as new, changing all devices to default values, thus changing the lights accordingly, resetting the last time presence was detected, and starting to clean the house again (*e.g.*, activating the vacuum cleaner robot) even if it was cleaned just before the system restart. How to avoid the fallback to default values in such cases?

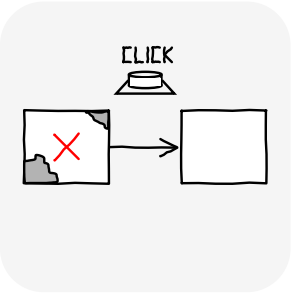


Therefore, preserve the current system state, avoiding the repetition of actions or changing devices states to default values after a disruptive recovery action.

Rationale. The correct functioning of an IoT system depends on preserving parts (or all) of the system current state (*i.e.*, checkpoint), enabling the system to restore to the last known state if a system error or restart/reset happens (*cf.* Reset). This allows the system to recovery seamlessly (*i.e.*, rollback), without repeating tasks and/or bothering the user to restore to the most current configurations. **Also see:** Checkpoint [Han07], What To Save [Han07], Rollback [Han07], Roll-forward [Han07], Checkpoint [Elo+14], Snapshot [Elo+14].

9.9 Reset

As the system operates, several faults can happen without triggering an *error* (*i.e.*, latent faults), going unnoticed by the users and, even, by the system itself. However, these faults can build up, leading to system errors and, possibly, system failures. How can we reduce the probability of errors and failures being triggered as time passes? Further, several issues can appear during service delivery that can compromise the correct operation of devices (*e.g.*, wrong user inputs, electromagnetic radiation, or power spikes) that can lead the device to present *undefined behavior*. How can we restore the device to a healthy state?



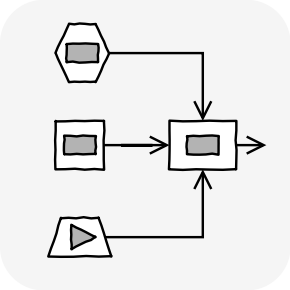
Therefore, perform system resets, periodically (*e.g.*, during idle periods) or when some error is detected, working both as maintenance of health mechanism and, possibly, as a fault removal procedure.

Rationale. Reboot and reset strategies have been used for a long as a way to improve systems reliability both in terms of software (*e.g.*, application restart [Ter16]) and hardware (*e.g.*, hardware watchdog timers) concerns [Cun+02; Abd+17; Aba19]. The continuous system operation can lead to the creation of several latent faults that can be triggered, leading to system errors and failures. Even if one could argue that with more resilient (*e.g.*, rigorously tested and verified) software and hardware, the probability of a system entering in an error or failure state is reduced, IoT systems are known to be built with low-cost components with high failure rates (*e.g.*, communication issues, sensing imprecision's) [Cha+18; KC18]. These resets can be both *soft-resets* and *hard-resets*, depending on what they preserve in terms of the device's internal state. For example, *soft-resets* provably will only work for non-permanent faults (*e.g.*, if a fault is preserved in a checkpoint — *cf.* Checkpoint —, a hard-reset should be performed). However, depending on the device or system, rebooting/reset the system periodically can introduce inconsistencies in operation (*e.g.*, system state synchronization) than can negatively impact the system. Further, since reboot/reset can restart all its processes, any dormant storage corruption can provoke malfunctions. **Also see:** Routine Maintenance [Han07], Routine Exercises [Han07], Data Reset [Han07], Roll-forward [Han07].

9.10 Consensus Among Values

Triggering an alarm can have serious consequences, so, commonly, alarm systems rely on several and, sometimes, different sensors to keep the environment under check. However, a misconfigured or malfunction sensor can report some nefarious condition (*e.g.*, CO2 levels) without being correct. How can the system deal with such a report without erroneously triggering the alarm system?

Therefore, evaluate information from several sources before taking a decision, increasing the level of confidence on the action to take, minimizing the likelihood of mistakes.

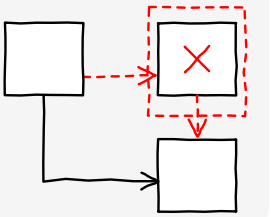


Rationale. When several sources report data upon which the system acts, all the information should be considered. In most cases, malfunctions can be easily identified and discarded by comparing the collected information and only considering the information with which the majority agrees with (*i.e.*, voting). More advanced mechanisms can be used that also take into account the trustability of the reported values (*cf.* Compensate) [Ter16]. Nonetheless, there can be situations where the majority of the sensors are reporting erroneous values. In such situations, depending on factors such as the importance of such readings, more sensors can be added and/or the misreports should be considered and, at least, communicated to the system owner/administrator. **Also see:** Voting [Han07; Elo+14], Anomaly Detection [Uki+16].

9.11 Circumvent and Isolate

As the system operates along time parts of it can reach end-of-life (*e.g.*, an A/C which no longer can achieve the target temperature), having high-failure rates which can impact the overall system operation and, possibly, threaten the user comfort and well-being. How can the system deal with failing parts without compromising its correct operation?

Therefore, circumvent and isolate failing parts, by disabling faulty components and reconfiguring the system to ignore those, avoiding system-wide disruptions.

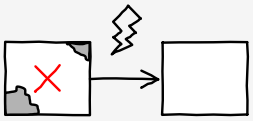


Rationale. System parts can have errors and failures, ranging from hardware/software issues to provoked malfunctions (both intentional and unintentional) that can impair the system's operation as a whole. When such problems are detected, and their origin is identified, mechanisms should be triggered that circumvents and isolate the failing part. Suppose that the A/C unit is defective. In that case, the system must not rely on it anymore (until there is a repair intervention) and use alternative ways to achieve the same goal, *cf.* Runtime Adaption (*e.g.*, during summer, to lower the temperature, opening the home windows can be an acceptable solution). **Also see:** Roll-forward [Sar02; Han07], Error Containment Barrier [Han07], Riding Over Transients [Ada+98; Han07], Leaky Bucket Counter [Ada+98; Han07], Quarantine [Han07], Input Guard [Han06], Output Guard [Han06], Loose Affiliations [Han14], Circuit Breaker [KHA17], Output Interlocking [RK15].

9.12 Flash

Things can be deployed in remote areas with only periodic maintenance cycles (*e.g.*, every three months). During this time, the devices are exposed not only to environmental conditions, but they also can be stolen or modified by an adversary (*i.e.*, user with malicious intent). In such cases, if the device remains part of the system, it cannot be trusted (since software modifications could have been performed), *cf.* Circumvent And Isolate. A Reset is not enough if the running software modifications were permanent. How to regain control of the device?

Therefore, flash the device with a trusted software version, remotely (if possible), or by collecting and redeploying the equipment. This can also be known as factory reset or wipe *and* reinstall.



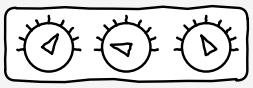
Rationale. Most of the low-cost devices that are part of IoT systems are prone to physical attacks due to the limitations of encryption (mostly resulting from the limited computational power). When a device shows suspicious activity (*cf.* Unsurprising Activity), mechanisms should be triggered to reclaim control of the device, remotely if possible to flash the device with a trusted software version *over-the-air* or physically by collecting and redeploying the device. When recovery is not possible, there should exist a *kill switch* that erases/destroys the device, limiting what the attacker can do to the system. **Also see:** Remote Lock And Wipe [Rei+16], Blacklist [Rei+16], Whitelist [Rei+16], Bumpless Update [Elo+14], Updateable Software [Elo+14].

9.13 Calibrate

Devices sensors and actuators can deviate from their expected behavior due to decalibrated elements (both in software and hardware). Typically, decalibration errors are consistent, showing up every time a new measurement/action is taken. Regarding sensors, even if they are designed to have high-accuracy, the storage, transport, setup of the devices, along with being subject to heat, cold, shock, humidity, and other nefarious conditions, can lead to sensor[1] decalibration. We can consider, as an example, that most of the sensors require ADC calibration for accurate readings, and power monitoring chips must be calibrated before being used for measuring the energy consumption [LD18]. Regarding actuators, as an example, a potentiometer can decalibrate with its usage and lead to unexpected outcomes. How can we ensure the accuracy of the data collected by the sensing devices and the actions carried out by actuating devices?

[1] It is important to note that in a sensing device, the sensor itself is only one component in the measurement system.

Therefore, (re)calibrate the device to meet the expected behavior, by remote or *in-situ* hardware tuning, potentially supported by cooperating sensors.

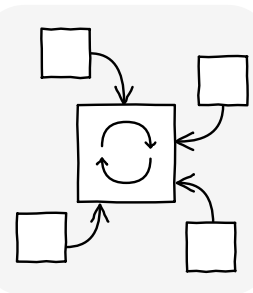


Rationale. Several sensing devices have calibration requirements that must be realized to make the devices properly function. Further, as time goes by, some of these devices can suffer some decalibration due to several causes, such as usage. As an example, a motion sensor must be calibrated in terms of trigger-periodicity and detection range. If one or both of these configurations are erroneously done or suffer from some decalibration (*e.g.*, malicious modifications), it will cause the device to misfunction, Unsurprising Activity, thus a recalibration is needed. Additionally, we can consider that several communication infrastructure and protocols can require calibration to work properly in the environments they are deployed to (*e.g.*, find the right Wi-Fi channel that is less used can improve communication reliability). **Also see:** ProCal [LD18], SensorTalk [Lin+19].

9.14 Rebuild Internal State

System internal state (partially or as a whole) can suffer from inconsistencies due to the concurrent nature of the IoT systems (*e.g.*, due to multiple ways of interact with the system parts). Refer to Internal Coherence for an example on the incoherence that can be introduced by the concurrent inputs of a light system. How can we restore to a stable and coherent system state that reflects the real-world system state?

Therefore, rebuild the internal state of the system to comply with the current system state. Internal Coherence probe can trigger this. The system can be restored by querying the existing devices about their state or recover from external state storage.



Rationale. IoT systems are concurrent, with several *inputs* that can come from a wide range of origins (*e.g.*, mobile applications, external APIs or physical device triggers — buttons). However, as the system operates, events (*e.g.*, power-surge) can lead the system to an inconsistent state. In these cases, a Checkpoint is not enough since the state of different system parts can be different from the existing checkpoint. Further, there is no need for a Reset. Thus, the system part can rebuild its internal state by observing the current environment or system state. **Also see:** Safe State [Elo+14], Bootstrapper [Elo+14], Start-Up Negotiation [Elo+14].

9.15 Summary

This chapter introduces a total of 14 patterns for recovery and maintenance of health in IoT systems. From a self-healing perspective, these patterns consist of *healing* procedures that can occur (*i.e.*, be triggered) in accordance with the errors detected by the probes presented in the previous chapter (*cf.* Chapter 8, p. 159).

While some patterns address cross-cutting issues to almost any other computing system, these are presented and discussed under the IoT point-of-view, giving this kind of system constraints and features. The adoption of some of these patterns can be straightforward if the system implements some of the patterns introduced in Chapter 7 (p. 151).

Part III

Dependable and Autonomic Computing

10 | Dynamic Allocation of Serverless Functions in IoT

While the majority of the developers in IoT opt for using the cloud for every need, disregarding the computing power that exists locally (*cf.* Local-First [Raw+18] and NoCloud [Raw+18]), there exists literature that already tries to leverage the concept of serverless in IoT domain — which some authors have called as *deviceless*[1] [GND17; ND18; Gus+19] — proposing the offloading of computing tasks during runtime horizontally (across devices in the same tier) or vertically (across devices in different tiers). Gusev *et al.* [Gus+19] analysis suggests that using serverless in IoT (*deviceless*) can bring advantages in terms of energy efficiency, scalability, and elasticity, and increased fault-tolerance (*i.e.*, opportunistic computing). Due to the nature of serverless functions, some of them can be executed locally, using the joint processing power of the multiple IoT devices or leveraging specific device capabilities. The hardship comes with using this power efficiently, having multiple serverless functions, and knowing where to execute each one, locally or remotely. It is not feasible for the developer to manually analyze performance across the different runtime environments and decide where the function should be executed, mainly due to the highly volatile nature of this performance (*e.g.*, dependency on lag and computing power availability).

Given the existing limitations on using serverless in IoT — *i.e.*, dynamically distribute computation tasks and workloads in IoT systems — this chapter explores and defines an architecture for a serverless IoT system, evaluating its feasibility by building a reference POC.

> Parts of this chapter were published in work DYNAMIC ALLOCATION OF SERVERLESS FUNCTIONS IN IOT ENVIRONMENTS [PDS18], and were partially based on the master thesis work of Duarte Pinto entitled SERVERLESS ARCHITECTURAL DESIGN FOR IOT SYSTEMS [Pin18]. The author's main contributions were on the conceptualization and supervision of the work, formal analysis and data curation, visualization, and writing of the published versions of the work.

[1] Glikson *et al.* [GND17] defines *deviceless* as a new computing paradigm — *Deviceless Edge Computing*.

10.1 Approach Overview

Nastic *et al.* [ND18] in their work summarize what they consider as the main research challenges of *deviceless computing*, which include: (1) resource pooling and rapid elasticity, (2) security across execution environments, (3) automated provisioning and management at scale due to the dynamic nature, heterogeneity, and distribution of IoT systems, (4) ensuring correct scheduling on loosely coupled and scarce resources, (5) lack of supporting platforms and abstractions to develop IoT systems that leverage serverless principles, and (6) governance of these systems become complex, as computing tasks can be allocated to be executed in contexts where different regulatory laws are enforced (*e.g.*, different countries).

Given the aforementioned challenges and the limitations of existent serverless solutions, an approach that fulfills the goal of dynamically distribute computation tasks and workloads in IoT systems should:

- Have a serverless *orchestrator* (*i.e.*, handler proxy) in the fog tier — *i.e.*, being part of the local network — capable of allocating task-execution requests from the different entities of the system to devices on the different tiers;
- Make use of the local processing power of the available resources regardless of the tier they are part of;
- Support the existence of multiple IoT devices with different functions (*i.e.*, capabilities), making them capable of interacting with both the cloud and fog tier to execute different tasks.

The core block of our approach is the introduction of a proxy between the entity (*e.g.*, device, mobile application, conversational assistant, or any other system part) requesting the function execution and the serverless function. This proxy will analyze each function's history by looking at the time taken in past requests and decide of which runtime environment[2] should the request be forwarded to (*cf.* Figure 10.1, p. 186). The proxy should be able to decide between the local network of devices and one of the many available servers.

In order to improve fault-tolerance, in case of no Internet connection (or if one of the servers is not available), in the case of request failure (*i.e.*, no response after some time), the proxy should fall back to the local network. This way, even if the request to execute the function is forwarded to the server and fails, the function will still be executed locally.

The proxy is situated inside the local network of IoT devices. It will forward the request for a specific function to a gateway that forwards the function execution to one of the IoT devices capable of executing the function. The load management, containerization, replication, and clustering of the serverless functions are not handled by the proxy. However, it still has to be aware of the serverless functions installed in the local network or other runtime environments.

[2] Runtime environment is the system where the serverless function will be executed, *i.e.*, local network or one of the servers available in the cloud

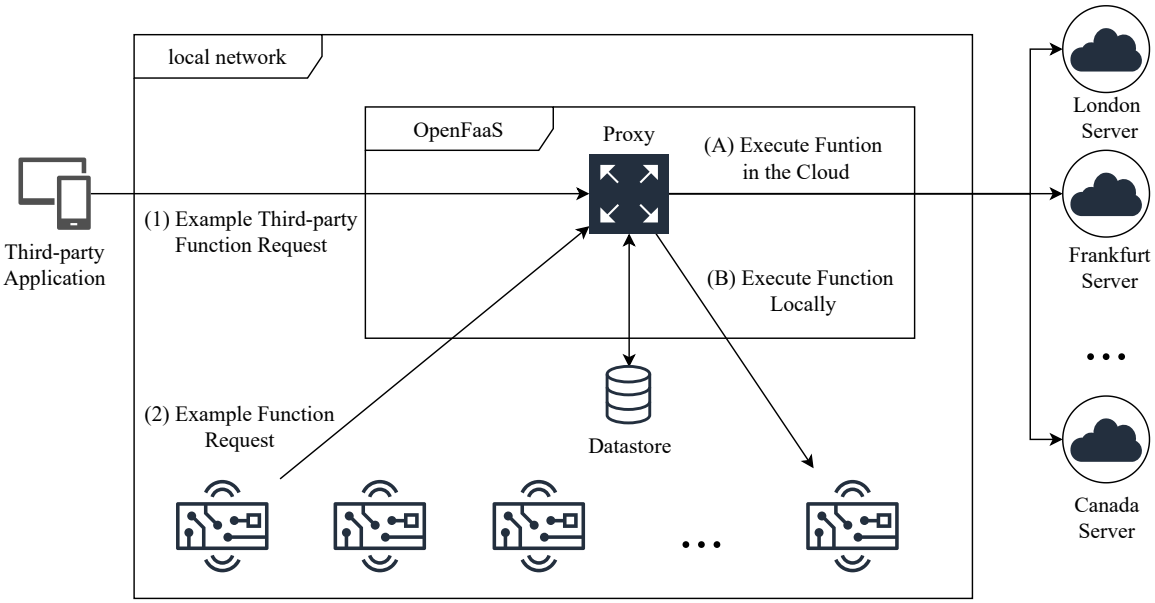


Figure 10.1: Overview of the system operation. A function execution request can be made by: (1) an external third-app (connected locally or over the Internet) or by (2) a local device on the network. The proxy can allocate the function to run on: (a) local computational resources, or (b) cloud computing resources.

10.1.1 Motivational Scenarios

The following examples explain the expected results and decision-making process of the resulting proposed solution. The decisions taken by the proxy are based only on previous metrics of the time taken for the runtime environment to execute the function (including network latency).

Forward Function Execution to the Cloud

The scenario in Figure 10.2 (p. 187) exemplifies a situation where the requested serverless function is hardware intensive, therefore taking too much time to execute locally. Due to the high processing power of the cloud servers, it is beneficial to forward the execution request to one of the available cloud servers, even when considering the connection latency. The multiple available servers will opt for the one that is physically nearest (lower latency).

Forward Function Execution to the Local Network

Contrary to the previous case, the Figure 10.3 (p. 188) portrays a scenario where the requested serverless function is considerably lightweight, being more beneficial to execute the function locally and avoid network latency. Despite the difference in power between the two environments, the previous metrics illustrate that the local environment can satisfy the request more quickly.

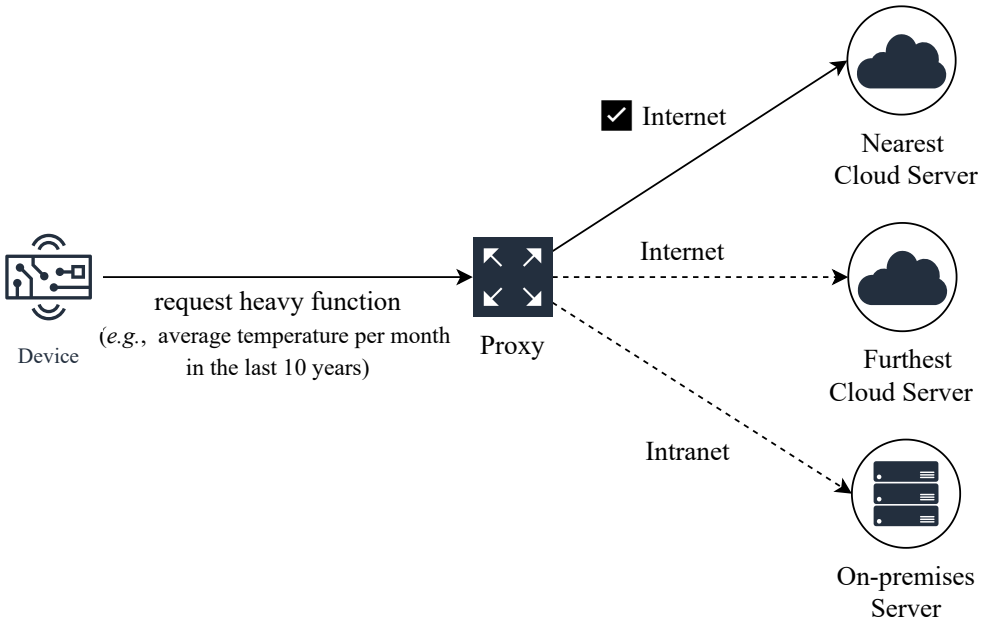


Figure 10.2: Request for the execution of a demanding function. The proxy will forward the request to the cloud because due to the high processing power of the cloud server, the function will be executed more quickly. The nearest server was chosen because of latency.

Fallback to Local Network

The Figure 10.4 (p. 189) depicts a scenario where the proxy first tries to forward the request to one of the cloud servers (because of its better overall performance) but fails in doing so. The proxy then decides to forward the request to the local network, completing the request. There are certain situations where it is more favorable for the function to be executed on the cloud, but it could still be executed locally. Because it is not possible to guarantee an always working network connection, in these cases, if the connection fails, the proxy will fallback to execute the function locally, assuring fail redundancy and the reliability of the system.

Manual Forward by Bypassing the Weighting Process

It should also be possible for the developer to bypass the weighting process (the evaluation of the different runtime environments) and manually choose where to forward the request. This option should be possible either in the function setup process or as an argument for the function execution request.

10.1.2 Weighting Runtime Environments

When receiving a request, the proxy will first pick from a list of various runtime environments — the list must include the local network of devices and one or more cloud servers — where to forward the request to execute a given serverless function. The decision is based on previous metrics of the time taken for the function to execute in the different runtime environments

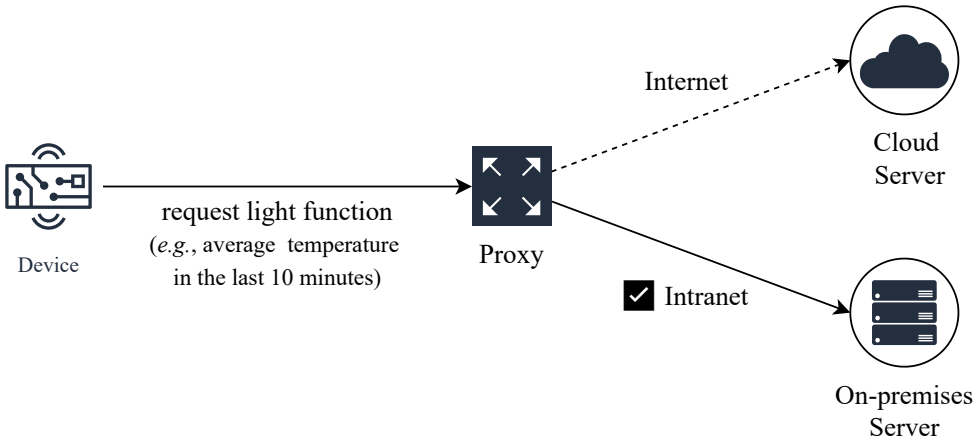


Figure 10.3: Request for the execution of a simple, light function. The proxy will forward the request to be run locally, as there is no benefit in executing the function on the cloud.

and will aim to choose the fastest one (the one with less time taken). This can be modeled as a common *Exploration vs. Exploitation* problem.

Exploration vs. Exploitation is a common decision-making dilemma, where there is the need to choose between a known good option or taking the risk of trying an unknown option to finding the best possible result. This is done in a way that tries to minimize the total opportunity loss [Sil20b].

If we had access to all the information about the universe in question, we could either brute-force or use other smart approaches to achieve the best results. In this situation, the problem comes from only having *incomplete* information. In order to make the best overall decisions, we need to simultaneously gather enough about the system and keep the regret value at a minimum. Exploitation will choose the best-known option to avoid any regret. Exploration will take the risk of choosing one of the less explored options to gather more information about the universe in question, reduce short-term success for long-term success. A good strategy will use both options, exploration and exploitation, to achieve the best results.

The Multi-Armed Bandit is a known problem that exemplifies the *Exploration vs. Exploitation* dilemma. The problem places us with multiple slot machines, each with a different reward probability. Given the setting, the objective is to find *the best strategy to achieve the highest long-term reward* [Wen18].

The goal is to maximize the total reward, $\sum_{t=1}^{T} r_t$, or in other words, minimize the regret of not taking the optimal action in every step.

The optimal reward probability θ^* of the optimal action a^* is:

$$\theta^* = Q(a^*) \tag{10.1}$$

$$= \max_{a \in A} Q(a) \tag{10.2}$$

$$= \max_{1 \leq i \leq K} \theta_i \tag{10.3}$$

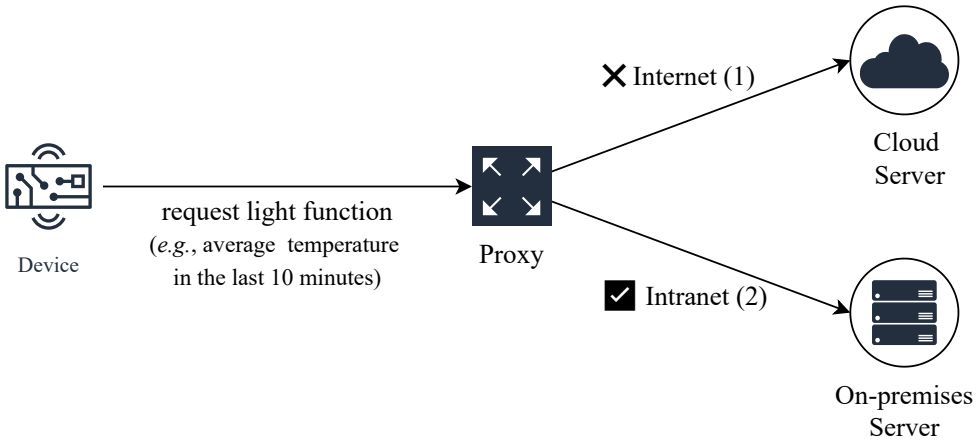


Figure 10.4: The request for the function execution to be in the cloud could not be satisfied (*e.g.*, no Internet connection). The proxy will then forward the request for it to be executed in the local network.

In the serverless task allocation use case, the decision of which runtime environment to forward the function must have some weighting process that will assign the weights for each runtime environment and compare them. This process will gather information about the different runtime environments and then accurately estimate which one is the best choice (less time took). Therefore, we considered the algorithms presented in the following paragraphs to handle the weighting process.

Greedy This algorithm only takes into account the average time taken for a given task to complete in a given running environment. It has no exploration; thus, it directly assigns the weight as the mean of the time taken by previous runs.

ϵ-**Greedy** This algorithm will choose the best-known action most of the time, but it will also explore randomly from time to time. The value of an action is given by [Wen18]:

$$N_t(a) = \sum_{\tau=1}^{t} \mathbb{1}[a_\tau = a] \tag{10.4}$$

$$\hat{Q}_t(a) = \frac{1}{N_t(a)} \sum_{\tau=1}^{t} r_\tau \mathbb{1}[a_\tau = a] \tag{10.5}$$

Where:

$\mathbb{1}$: is a binary indicator function.

$N_t(a)$: represents the total number of times that a given action as been selected.

In this algorithm, we take the best-known action most of the time, $\hat{a}_t^* = \arg\max_{a \in A} \hat{Q}(a)$, or, with a probability of ϵ, we take a random action. The best-known action will be taken with a probability of $1 - \epsilon$.

Upper Confidence Bounds (UCB) Random exploration might not be the best option because it might lead to trying an action that was previously concluded as bad. One way of avoiding this is to give priority to options with a *high degree of uncertainty*, actions for which there is no confident value estimation yet [Wen18].

UCB will translate this potential into a value, the upper confidence bound of the reward value, $\hat{U}_t(a)$. The true will be below $Q(a) \leq \hat{Q}_t(a) + \hat{U}_t(a)$. $\hat{U}_t(a)$ will vary with the number of trials, and a larger sample of trials will result in a smaller $\hat{U}_t(a)$. With the UCB algorithm, the next action to take in will be:

$$a_t^{UCB} = \arg\max_{a \in A} \hat{Q}(a) + \hat{U}(a)$$

To estimate the upper confidence bound, if prior knowledge of how the distribution looks like can be discarded, then it is possible to apply *Hoeffding's Inequality*, a theorem that can be applied to any bounded distribution [Sil20b].

Applying Hoeffding's Inequality to the rewards of the bandit will result in:

$$U_t(a) = \sqrt{\frac{-\log p}{2N_t(a)}}$$

UCB1 To ensure that the optimal action is taken as $t \to \infty$, p can be reduced with each trial [Sil20b]. Thus, the UCB1 algorithm replaces $p = t^{-4}$, which results in:

$$U_t(a) = \sqrt{\frac{2\log t}{N_t(a)}} \tag{10.6}$$

$$a_t = \arg\max_{a \in A} Q(a) + U_t(a) \tag{10.7}$$

$$= \arg\max_{a \in A} Q(a) + \sqrt{\frac{2\log t}{N_t(a)}} \tag{10.8}$$

Where:

a_t : is the final weight as calculated by UCB1.

Bayesian UCB The previous method, UCB1, does not make any assumptions about the reward distribution R, only relying on Hoeffding's Inequality to make an estimation. Knowing the distribution would allow for closer estimates.

In the Bayesian UCB it is assumed that the reward distribution is Gaussian, $R_a(r) = N(r; \mu_a, \sigma_a^2)$. The action that will give the best result is the action that maximizes standard deviation of $Q(a)$ [Sil20b]:

$$a_t = \arg\max_{a \in A} \mu_a + \frac{c\sigma_a}{\sqrt{N(a)}} \tag{10.9}$$

Where:

c : is an adjustable hyper-parameter.

Bayesian UCB relies on a considerable amount of prior knowledge of the rewards, and for it to be accurate [Sil20b], otherwise, it would be the best fit.

10.1.3 Implementation Details

The proposed solution is constituted of a main component, the `proxy`, and an auxiliary package, the `sample_functions`, which contains sample functions for validation purposes.

The `proxy` package is the main package of the POC. It is responsible for (1) receiving the requests for the execution of a function, (2) the weighting of the environments in which the functions can run (locally or in the cloud in one of the multiple servers), and (3) the storage and retrieval of all metrics regarding previous function executions that are used for future weighting processes.

The developer can define a priori, or in runtime (*e.g.*, device selects) the weighting algorithms will be used for a request as a parameter, but the default algorithm is `UCB1`. Despite Bayesian UCB's improved performance, UCB1 requires no previous knowledge about the environment; thus, it was picked as the default setting.

The `sample_functions` component was created to fulfill the validation needs. It contains the serverless functions whose execution is going to be requested to the proxy package. This package is purely a sample to simulate and analyze resource-demanding serverless functions (either light or heavy) and could be replaced by any other set of functions. The functions inside this package can be executed on the local environment or in one of the cloud servers (remote environments).

Proxy

The decision of function allocation is based on previous metrics of the time taken for the function to execute in the different runtime environments. It will aim to choose the fastest one (the one with less time taken). Nonetheless, it is also possible for the runtime environment to be manually configured in the request options or when setting up the proxy.

If the proxy chooses to forward the request to the local network, it will just wait for the function to execute. Alternatively, if the decision is to execute the function in one of the cloud servers, it will request the cloud server for it to execute the serverless function. Lastly, if the request is done to a cloud server and fails to complete (*e.g.*, because there is no connection to the server), there will be an attempt to execute the function in the local network of devices. After having the response from the function execution, the proxy will answer with the corresponding result.

The proxy communicates through HTTP with the different entities (*e.g.*, devices or cloud servers) and expects to receive the content as `application/json`. The main proxy modules are the following:

- `proxy` The main function through which all requests go through first. After receiving the list of weights associated with the execution of the requested function in each environment, it will choose the environment with the least weight and forward the execution of the serverless function to that runtime environment;
- `weight_scale` This function will analyze all the collected metrics of the requested function and assign a weight to each runtime environment. It allows more than one algorithm for weight estimation;
- `get_duration` Retrieves the list of all the collected metrics of a function;
- `insert_duration` Store the time taken for a function to execute;
- `get_overall_stats` Function that returns the summarized records of all the collected metrics for each function in each environment. Useful for analysis and evaluation of the results.

These provide enough functionality for both allocating tasks as they are requested and to collect/store information that can be used to (re-)calculate the strategy used in the next function execution request of the same type.

Sample Functions

A set of *sample functions* was created to simulate serverless functions with different purposes and (forcing) different execution times. The functions are aware of the runtime environment they are being executed on, and it is possible for them that the answer differs according to this. Here, the different time taken is simulated using a *wait* and using different values for different runtime environments.

- `func_light` A very lightweight function which answers almost instantly; there should be no difference between executing the function locally or in the cloud other than connection latency;
- `func_heavy` In this function, there is a *wait* of 2 seconds if it is executed locally or a *wait* of 1 second if it is executed on the cloud. There should be no difference in the time taken across different cloud servers other than connection latency;
- `func_super_heavy` Similarly to `func_heavy` but here the difference in time is bigger. There is a *wait* of 4 seconds if the function is executed locally or 2 seconds if the function is executed in the cloud;

- `func_obese_heavy` This function can only be executed in the cloud, and its execution has been flagged as cloud-only. The proxy will not even try to run the function locally, and it will always forward the request to the cloud. Because of this, there is no fallback to run locally.

This base set of *sample functions* allowed us to simulate our motivational scenarios with different combinations, allowing us to carry several experiments and observe how our solution works while dealing with different constraints (*e.g.*, network lag).

10.2 Experiments and Results

In this series of experiments, the setup was configured with three runtime environments: one local virtual machine running all the functions with Lubuntu 16.04, 2 cores and 1Gb of RAM, and two cloud servers, one in London and the other in Frankfurt, both hosted in an EC2 instance in AWS[3]. They are both *t2.micro*, running Ubuntu 16.04, with one core and 1Gb of RAM.

Both servers were pinged before the test to verify the connection latency. The experiment was carried in Porto, Portugal (circa 2018), giving the following latency values:

- London (*eu-west-2*) : 71.153ms
- Frankfurt (*eu-central-1*) : 52.297ms

This result serves as a basis for these experiments, but other network providers and places around the same city the tests were performed (Porto) would possibly present different latency values.

10.2.1 Stable Internet Connection

For this experiment, both cloud servers were up and running and performed 99 iterations of requests. In each iteration was requested for every single one of the serverless functions in `sample_functions` (*cf.* Section 10.1, p. 185) to be executed. The system does not know the environment. The main objective of this experiment is to check the feasibility of the scenarios presented in Section 10.1.1, p. 186, Section 75, p. 186, and Section 77, p. 187.

It is expected that the proxy will forward requests for `func_light` to be executed locally, for `func_heavy` requests to be executed either locally or in the cloud. It will depend on the impact of the connection latency. However, the connection latency should generally be less than 1 second (difference in time that takes for the function to execute locally and remotely). The expected result is for the proxy to choose to forward to one of the cloud servers. The `func_super_heavy` is expected to be executed in the cloud most of the time due to the large difference in time taken, and the function `func_obese_heavy` should always be executed in the London server because it is configured that way. Apart from `func_obese_heavy`, some exploration is expected to occur for each one of the different environments and not only exploitation of the runtime environment that the system considers as the best option. Considering the

[3]An EC2 instance is a virtual server in Amazon's Elastic Compute Cloud (EC2) for running applications on the Amazon Web Services (AWS) infrastructure, https://aws.amazon.com/pt/ec2/instance-types/.

Table 10.1: Stable Internet connection experiment results.

		`func_light`	`func_heavy`	`func_super_heavy`	`func_obese_heavy`
On-premises	avg. time (s)	0.085602	2.111840	4.100240	-
	count	94	20	25	-
London	avg. time (s)	3.423012	5.741277	6.169448	3.384546
	count	3	10	20	99
Frankfurt	avg. time (s)	0.336398	1.253591	2.265615	-
	count	2	69	54	-

baseline observed latency, and when choosing between servers, the approach should pick the Frankfurt server because it is the one with less latency (despite being physically further).

It is expected that the proxy will forward requests for `func_light` to be executed locally, for `func_heavy` requests to be executed either locally or in the cloud. It will depend on the impact of the connection latency. However, the connection latency should generally be less than 1 second (difference in time that takes for the function to execute locally and remotely). The expected result is for the proxy to choose to forward to one of the cloud servers. The `func_super_heavy` is expected to be executed in the cloud most of the time due to the large difference in time taken, and the function `func_obese_heavy` should always be executed in the London server because it is configured that way. Apart from `func_obese_heavy`, some exploration is expected to occur for each one of the different environments and not only exploitation of the runtime environment that the system considers as the best option. Considering the baseline observed latency, and when choosing between servers, the approach should pick the Frankfurt server because it is the one with less latency (despite being physically further).

The obtained results matched the expected results, as can be observed in Table 10.1 (p. 194). For `func_light`, the time taken was so small (less than 1/10 of a second) that proxy immediately converged in the best option. The results for the execution of the function `func_light` translate the results expected for the scenario in Section 75, p. 186.

For `func_heavy` and `func_super_heavy`, it kept a ratio of *Exploration vs Exploitation* of 3/7 and 5/6, respectively, but always choosing the fastest of the cloud servers. The exploration rate also increases with the execution duration, meaning that the proxy will look for possibly more fitting options (*i.e.*, in terms of time-to-complete) the longer it takes for a function to execute. The results for these two functions correspond to the ones expected in the scenario presented in Section 10.1.1, p. 186.

Also, as expected, `func_obese_heavy` had a 100% accuracy, thus matching the expected outcome stated in scenario presented in § 77 (p. 187).

10.2.2 Broken Internet Connection

Both servers were turned off for this experiment, and the Internet connection was cut, leaving the system only operational locally. The system still keeps all the knowledge acquired in the previous experiment. The aim here is to identify that it is accomplishing the scenario in § 76 (p. 187).

In this experiment, it is expected for the weighting algorithm to suggest executing the serverless function in one of the cloud servers because it will lead to faster execution. Because there is no Internet connection, it is expected for the proxy to try to execute the function remotely, fail, and then fall back to the local runtime environment. In the end, the function should be executed in the local runtime environment leading to the request being answered successfully.

```
{
  status: "success",
  londonServer: 3.5747403999957266,
  frankfurtServer: 1.1756422708191938,
  local: 2.090245031544607
}
```

Listing 10.1: Weights of the different servers, for the second experience, using the Bayesian UCB algorithm.

First, the function *weight_scale* is queried to know which of the runtime environments the proxy is going to choose (because the proxy chooses the runtime environment with less weight, knowing the weights allow us to know which option the proxy is going to take). Because there is more information about the system, the weighting algorithm used was the Bayesian UCB. As it can be seen in the Listing 10.1 (p. 195), the runtime environment that is going to be chosen is Frankfurt's server.

Even though the chosen runtime environment was Frankfurt's server, because there was no Internet connection, it had to fall back to execute the function locally to complete the request successfully, as seen in Listing 10.2 (p. 195). The observed results match the ones expected, and also, the proxy proceeded as stated in the scenario presented in § 76 (p. 187).

```
{
  nodeInfo: "61e20a65b48e",
  swarm: "local",
  message: "I was able to achieve this result using HEAVY calculations",
  status: "The light is ON"
}
```

Listing 10.2: The request was executed locally, as indicated by the key *swarm*, which is the swarm (runtime environment) where the function was executed. The *local swarm* corresponds to the local network of devices, as configured when setting up the proxy.

10.2.3 Intermittent Internet Connection

During this experiment, the main purpose is to run a cycle of requests and then turn off the Internet access in the middle of the cycle to observe how this will affect response times. It will be run 99 iterations of requests, and in each iteration, it will be requested the execution

of `func_heavy`. After 50 requests, the Internet connection will be cut off, leaving the system only operational locally. The system will keep all the knowledge gathered in the previous experiments. The aim here is to identify that it is accomplishing the scenario in § 76 (p. 187) and how results vary over time.

In this experiment, it is expected for the weighting algorithm to suggest executing the serverless function in one of the cloud servers because it will lead to faster execution. Because of this, the mean total time of the first 50 requests should be smaller. After the 50th request, because the Internet connection was cut off, the proxy will try to execute the function remotely, fail, then fallback to execute the function locally, resulting in a considerable mean total time.

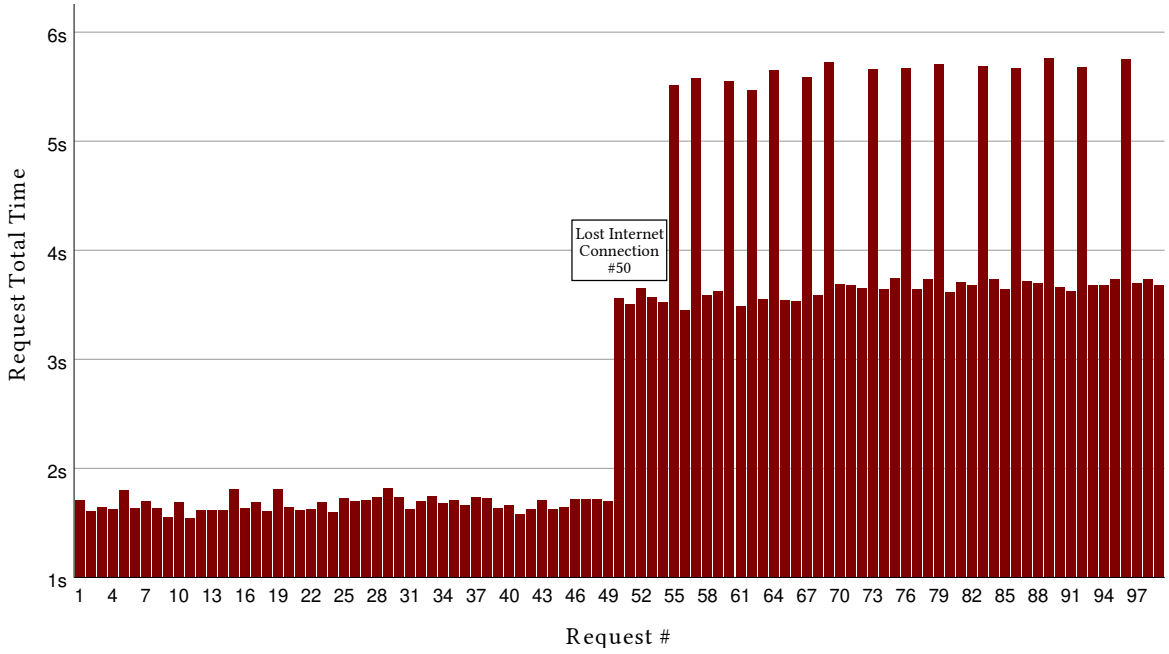


Figure 10.5: Response time per request before and after Internet connection drop.

The obtained results, illustrated in Figure 10.5 (p. 196), match the expected results. The mean total time of the request when there was an Internet connection was **1.714666** seconds, and, at iteration number 50, it jumped to **4.253786** seconds when the Internet connection was cut off. Despite having no Internet connection, the system could still complete the request, just with an added delay. The added delay was since it had to try to execute the function remotely and also because of the increased time it takes to execute the function locally (2 seconds).

10.2.4 Adapting to Network Lag

A series of requests will be performed, and after a while, the Internet connection will be purposely slowed to see how the system reacts in situations of lag and slow connection. It will be run 249 iterations of requests, and in each iteration, it will be requested the execution of `func_heavy`. After 50 requests, the Internet connection will be slowed down (28 Kbps UP, 14 Kbps DOWN), and the system will continue to be asked to execute the functions. The system

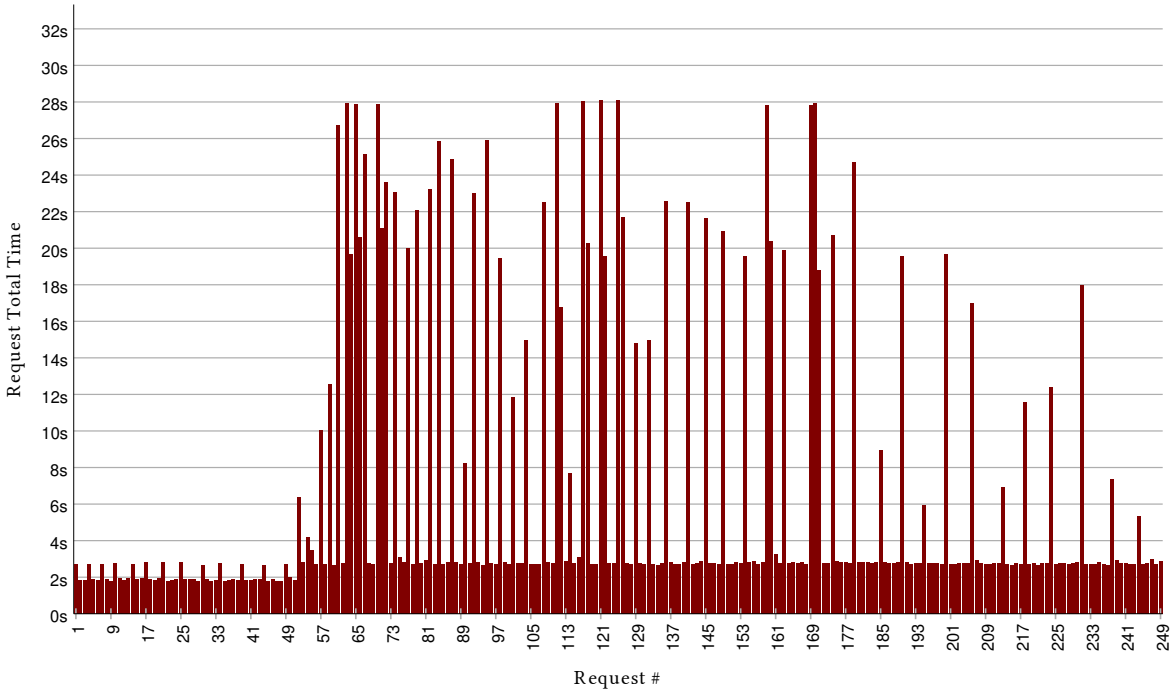


Figure 10.6: Requests total time throughout the various iterations of the fourth experiment.

will have none of the knowledge gathered in the previous experiments. The aim here is to identify that it is accomplishing the scenarios in sections 10.1.1 and 75, and also that it is capable of adapting to changes in the network.

In this experiment, it is expected that the system goes through three phases: (1) in the first 50 iterations, while the connection to the server is working as expected, the system is supposed to gather information about the environment and converge to the best option (one of the cloud servers); (2) the Internet connection is slowed down, and the function's execution remotely should take longer than the execution of the function locally. In this phase, the system is supposed to still converge to one of the cloud servers but gradually diminishing the frequency in which it chooses the cloud serves as the best option. After this phase, the system will enter the third phase, (3) where the results gathered after the introduction of network lag outweigh the results gathered in the first phase. Here, the system should start to converge to the local network as the best option.

10.3 Discussion

The gathered results can be observed in Figure 10.6 (p. 197). As expected, the results for the first 50 iterations are the expected results in a common situation. After introducing network lag, we start to observe spikes in the total time it takes for the function to be executed. These spikes refer to the execution of the function remotely. In the first 30/40 requests after the introduction of network lag (iterations 50 to 90), the frequency of requests that are executed remotely is still high. The frequency starts to diminish from that point on, and at around iteration 175, the system chooses the local network more frequently than the cloud servers.

Nevertheless, it took around 125 iterations (from iteration 50 to iteration 175) for the system to adapt. After 50 iterations where the system gathered information that became invalid, it took the system 250% more iterations to adapt to the new conditions. These results reveal that although capable of adapting, the system will take a considerable amount of time to adapt to new conditions.

10.4 Summary

While serverless has been largely adopted in cloud computing, its usage beyond the cloud is still scarce. However, several authors have been suggesting that serverless across architectural tiers will play a crucial role towards fulfilling the view of Edge Computing — *"enabling IoT and Edge devices to be seamlessly integrated as application execution infrastructure"* [GND17].

The experiments carried showcase the feasibility of using serverless (*i.e.*, deviceless) in IoT-systems, as it was showcased by the defined experimental scenarios of § 10.1.1 (p. 186). The developed approach can analyze the knowledge it has acquired over time of the ecosystem performance, optimizing allocation decisions that lead to a faster execution time while exploring different options that might lead to improved performance. Additionally, the developed solution can also detect failures in the remote execution of the serverless function and fallback to executing the function locally, thus improving the request-response success rate.

This chapter summarizes exploratory work towards distributing computing tasks across devices and computational tiers, foundational supporting block towards orchestrating IoT applications, as further explored in Chapter 3.2.5 (p. 79).

11 | Visual IoT Dynamic Orchestration

Although most IoT systems are of large-scale, they are typically designed and built around centralized architectures (as most of the existing Web services), where one main component executes most of the computation on data provided by edge devices (*e.g.*, QNAP QIoT Suite, Home Assistant, and OpenHAB) [Min+16; BCC18]. We also observe a tendency for centralized cloud services in cloud-based IoT architectures, mostly due to the advantages of management and costs (*e.g.*, *the economics of scale when building datacenters, automatic backup of all data, and enforce physical security* [WSJ15]). Examples include IoT PaaS such as Amazon Web Services (AWS) IoT, IBM Bluemix, and Microsoft Azure IoT Suite [PIS17]. These centralized approaches have several downsides, including: (1) computation capabilities of the edge devices are being ignored, (2) it introduces a single point of failure, and (3) data is being transferred across boundaries. Ideas such as the one of Local-First and NoCloud [Raw+18] have been mostly ignored.

In this chapter, we explore how the computation capabilities of heterogeneous devices capable of running custom code can be leveraged to improve the resiliency, efficiency, and scalability of IoT systems. For this purpose, a set of (1) extensions and modifications to the Node-RED system were made, and (2) a MicroPython-based firmware that runs on the edge devices that can receive, interpret and execute the orchestrator-assigned computational tasks was developed. The approach was evaluated in terms of functionality, resilience to hardware/software errors, and efficiency; by scaling up the number of available devices and computational tasks. We concluded that the system's resilience to failures was improved, and, once orchestrated, the system operated in a distributed fashion (even without the orchestrator's presence). We verified that the system scales at least up to 50 devices (affirming its suitableness for most *smart home* setups). Additionally, we observed that our approach increases the delay in communication between *nodes*, mostly due to changes in the *channel* (*i.e.*, from a Node-RED *in-process* communication to a decentralized Wi-Fi MQTT-based).

> Parts of this chapter were published in work entitled VISUALLY-DEFINED REAL-TIME ORCHESTRATION OF IOT SYSTEMS [Sil+20], and were partially based on the master thesis work of Margarida Silva entitled ORCHESTRATION FOR AUTOMATIC DECENTRALIZATION IN VISUALLY-DEFINED IOT [Sil20a]. Further improvements were part of the master thesis work of Pedro Costa entitled DECENTRALIZED REAL-TIME IOT ORCHESTRATION [Cos21]. The author's main contributions were on the development of the software, formal analysis and data curation, visualization, and writing of the published versions of the work.

11.1 Approach Overview

Although the prevalence of centralized architectures in IoT, Edge and Fog Computing have been suggested to solve some of these limitations by pushing some processing tasks away from the cloud and into lower-tier devices [MK16]. Nonetheless, most of the issues remain unaddressed, as the central instance, if it exists, should orchestrate the system so that the computational tasks are divided into independent blocks that could then be executed by other devices instead of running everything in a centralized way. Node-RED is an example of an open-source IoT development environment and runtime that follows a centralized view, being the solo event processing engine for the whole IoT system, according to the user-defined rules and triggers (*cf.* Figure 11.4, p. 206) [Ope19b; Ihi+20].

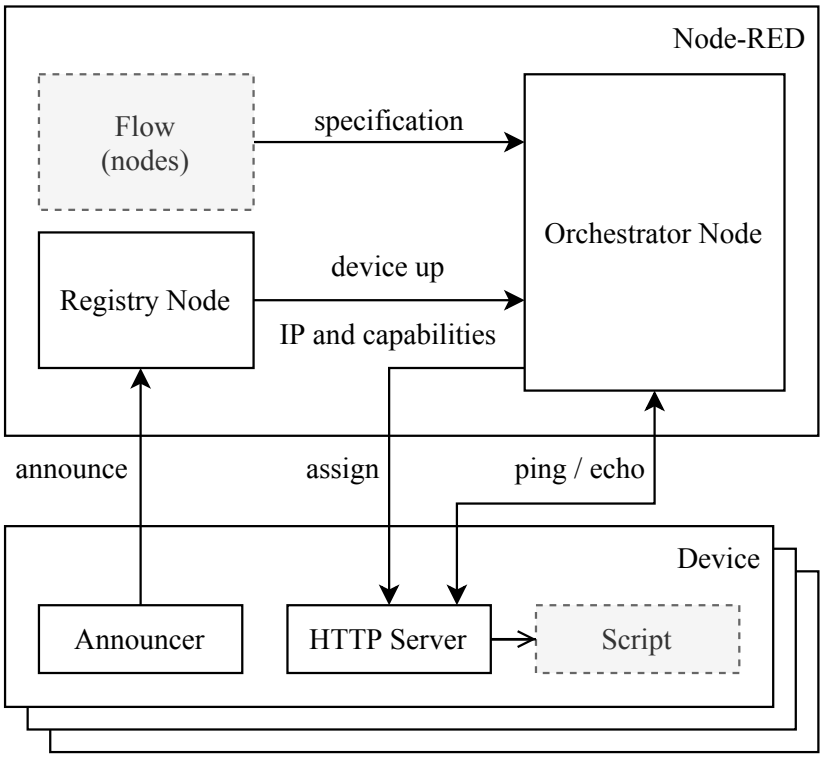


Figure 11.1: High-level overview of the proof of concept implementation.

In this approach for reducing the centralization of computation in IoT systems we leverage — and modify — Node-RED to (1) define programs (as *flows*) and (2) send tasks to other devices in the network, acting as a system's orchestrator (*cf.* Figure 11.1, p. 200). The network devices make themselves known by announcing their address and capabilities to a particular registry *node*. Consequently, Node-RED assigns *nodes* to devices (taking into account their capabilities) and communicates each node's assignment via HTTP. Constrained devices cannot directly run Node-RED *flows*, so the orchestrator translates the nodes' JavaScript code to artifacts that can be interpreted by these devices.

Two main components were introduced to the *palette*: (1) the *Registry node*, which maintains a list of available devices and their capabilities and, (2) the *Orchestrator node*, which partitions

and assigns computation tasks to the available devices. The capability of generating MicroPython code for supported nodes was added, and a MicroPython-based firmware was developed that receives and runs Python code generated by the orchestrator. The centralized Node-RED built-in node's communication was also replaced by an MQTT-based one, which is also used by the developed MicroPython firmware.

11.1.1 Devices Setup

Table 11.1: Comparison between the Espressif Systems ESP32 and ESP8266 SoCs as per the devices' datasheet [Esp20; Esp19].

	ESP8266	**ESP32**
MCU	Single-core 32bit	Dual-Core 32bit
Frequency	80 MHz	160 MHz
SRAM	160kBytes	512kBytes
Flash	SPI, 4MBytes	SPI, 4MBytes
802.11 b/g/n Wi-Fi	Yes	Yes
Bluetooth	No	Yes
Programming Language	Lua, Python and C	

We consider constrained devices that are capable of running custom code. Among the available hardware solutions, taking into consideration both costs and features, we picked two IoT development devices based on the Espressif Systems ESP32 and ESP8266 System On Chip (SoC), which technical specifications are presented in Table 11.1 (p. 201). The first challenge is to find a way to take advantage of the constrained devices by making them run arbitrary scripts of code and communicate with other devices. Since both selected devices can run MicroPython firmware, Python language was used [Bel17]. MicroPython already packs a small-footprint HTTP server, and packages are available to implement asynchronous operations (`uasyncio`) and MQTT publisher-subscriber (*i.e.*, pub/sub) communication (`MicroPython-mqtt`).

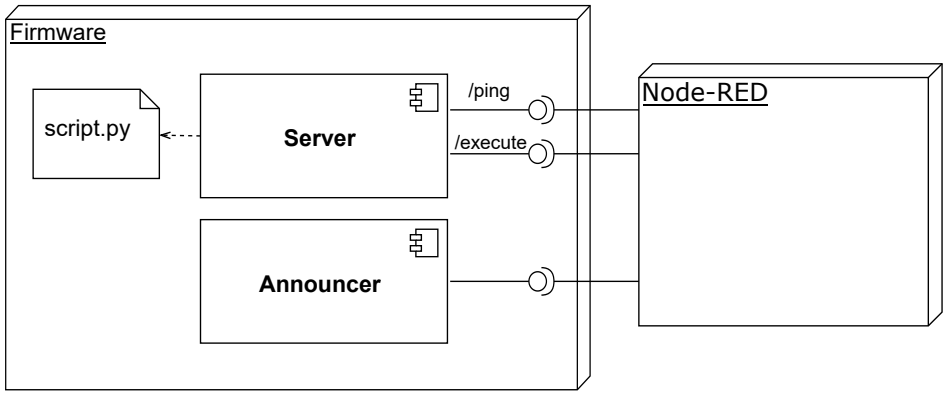


Figure 11.2: Summarized firmware component diagram.

As the devices must receive arbitrary Python scripts (sent by Node-RED) and run them, an HTTP server was used to receive the Python payloads and save them into the device memory

to be executed later. The same HTTP server is used to implement an endpoint that returns the state of the device and an announcing mechanism (*cf.* Section 11.1.3, p. 204). These features were built as an integral part of the firmware that runs on the devices. An overview of the components of the firmware can be seen in Figure 11.2 (p. 201).

The firmware also includes a `Fail-Safe` mechanism, safeguarding against several errors (including `Out-of-Memory`) that may happen during the device's lifespan (SRAM usage). This mechanism resets all running tasks and recovers the HTTP server and communication channels, being essential due to the high probability of these errors occurring due to the device's memory constraints.

11.1.2 Decentralizing Node-RED

Node-RED is centralized by design, taking advantage of events to allow communication between *nodes* in a *flow*. Implementing a decentralized architecture required some changes to its runtime. These changes consisted mainly of (1) implementing a different communication channel for *node-to-node* communication and (2) add code generation features (*i.e.*, JavaScript to MicroPython).

Node-RED Node-to-Node Communication

Node-RED *nodes* communicate using events — node.js `EventEmitter`. The communication is forward-only, with *nodes* only sending data to the following *nodes* in the *flow*. Output wires are used to access which *nodes* a message must be sent to by calling its `receive()` method. This method triggers the `emit()` event, which will be caught by a specific method, implementing its own logic, in each *node*. This implementation is local and JavaScript specific, making it hard to be used in a decentralized architecture where *nodes* will be executed outside Node-RED. It was necessary to implement a way of communicating between *nodes* external to Node-RED that could be supported by constrained devices.

As a first proof of concept, Node-RED `Node` class was modified to use MQTT pub/sub communication [SM17; Nai17a] instead of in-place communication. Since the modifications were made at the base class level (from which every *node* derives from) all the existing *nodes* and sub-*flows* became compatible with this modification without further changes.

In a later improvement, the MQTT communication channel was merged into the *nodes* themselves — removing the need for changes in the internals of Node-RED, allowing this approach to be used on any vanilla Node-RED, and not requiring any changes to it beyond installing the required *nodes* [Cos21].

Each *node* publishes messages to a unique and addressable topic generated at the start of the *flow* and subscribed by the next *node*. This happens for every *node* except for *producer nodes* that only act as publishers and *consumer nodes* that only act as subscribers. However, a *node* to be *orchestration-enabled* should be modified to comply with the orchestration process and generate the corresponding MicroPython code (*cf.* Section 85, p. 203).

Code Generation

To orchestrate Node-RED *nodes* among devices, we need to generate MicroPython-compatible code from the existing JavaScript (*i.e.*, code generation). It is also necessary to support multiple *nodes* in one script; thus, we defined a generalized strategy appropriate for any *node* type. This was accomplished by adding specific code generation methods to each *orchestration-enabled node*, which provide (1) their functionality and (2) input/output capabilities. Since every *flow* communication is now MQTT-based, the only input and output a *node* can have is its topics.

Code generation happens after the *node*-device assignment. This generation creates device-specific code that carries out the tasks assigned to the device (which can correspond to several *nodes*), adding some wrapping code that is responsible for subscribing to all input topics of all nodes, stopping the script's processes, and forwarding the messages to the respective *nodes*.

Device- and *node*-specific code is generated by (1) taking a basic, and generic, Python specification corresponding to the *node* functionality, which is then (2) complemented (filling the gaps) by the parameters passed through the Node-RED programming canvas (*node*-specific configurations). The result is valid Python code with specific device definitions (*e.g.*, I/O pin specification) which can be deployed to the devices.

Custom Nodes

As previously mentioned, all the existing *nodes* can still be used without modifications. Nonetheless, for a *node* to be *orchestration-enabled* it must be modified to comply with code generation needs. Each of these *nodes* has two available properties: *Predicates* and *Priorities*. Similar to the Kubernetes logic of assigning containers to machines [BT18], the *predicates* dictate constraints that cannot be violated, and *priorities* are requests that are advisable and recommended but can be ignored if needed.

As POC, the following *nodes* were modified (or created) to have a MicroPython representation:

- `if` which receives an input and verifies if it complies with all the given rules, returning true or false;
- `and` which receives a given number of inputs and verifies if all of them are true or false, returning the corresponding Boolean;
- `temperature-humidity` that read the temperature and humidity from a DHT sensor present in a specific pin;
- `fail` that raises a `MemoryError` exception (used for testing purposes);
- `nothing` that simply redirects the received message in its input to its output;
- `mqtt-in` and `mqtt-out` which subscribe and publish to MQTT topics, respectively.

These *nodes* provide enough base functionality to wire simple *smart home* scenarios and validate our approach. We believe those *nodes* to be enough to provide the basic functionality that would allow us to validate the correct function of our POC, and the feasibility of the approach as a whole.

11.1.3 Device Registry

IoT systems are typically built on top of heterogeneous parts, with different capabilities and resources, and their network can be highly-dynamic (devices can go off/on due to battery levels, hardware/software failures, and communication issues). To maintain a *list* of available network devices and their capabilities, we need a Device Registry [Ram+17] inside Node-RED.

When a device becomes available, information about itself is sent to an MQTT topic. This information contains the device's IP address, its capabilities, and status (*e.g.*, if the device has failed before). Node-RED contains a *Registry node* that listens to the announcements MQTT topics and saves the devices' information. If this *node* is connected to an *orchestrator node*, each time a new device appears, a message is sent to the orchestrator to consider the new resources in the following orchestration.

When a device has an `Out-of-Memory` error, it triggers a `Fail-Safe`, where it reboots the HTTP server, stops running any script and restarts all communications. After this action, the device announces itself again but with a flag that indicates that it has failed. This way, the *orchestrator node* knows that a device is active but not running any code and that it has possibly failed due to having too many allocated *nodes*. In that case, it can dynamically adapt and assign fewer *nodes* to the device, reducing the chances of causing another `Out-of-Memory` error.

11.1.4 Computation Orchestration

The requirements to achieve this are two-fold: (1) an *orchestrator node* should act as coordinator, which when provided with an available devices list, along with their respective capabilities (*cf.* Section 11.1.3, p. 204), should decide which device should execute specific *computation nodes* (*i.e.*, Node-RED blocks) and, (2) the *orchestration-enabled nodes* should provide both *Predicates* and *Priorities* that must be met to assure their correct execution (*cf.* Section 85, p. 203).

Algorithm 11.1: Greedy algorithm for *node* assignment.

Input : deviceList, node, $\alpha = 0.5$, $\beta = 0.4$, $\gamma = 0.1$
Output : bestDevice

onInput
  electible $\leftarrow \{d \in \text{deviceList} \mid \text{hasMem} \wedge \text{isReady} \wedge \text{isCapable}\}$
  where
    hasMem $\leftarrow |d.\text{nodes}| < |d.\text{lastError.nodes}|$
    isReady $\leftarrow d.\text{status} = \text{OK}$
    isCapable $\leftarrow \text{node.predicates} \subseteq d.\text{capabilities}$
  return $\underset{d \in \text{electible}}{\arg\max}\ \text{fitness}(d) = \alpha \cdot \text{overlap} + \beta \cdot \text{vacancy} + \gamma \cdot \text{specificity}$
  where
    overlap $\leftarrow \frac{|d.\text{capabilities} \cap \text{node.priorities}|}{|\text{node.priorities}|}$
    vacancy $\leftarrow (|d.\text{nodes}| + 1)^{-1}$
    specificity $\leftarrow \frac{|d.\text{capabilities} \cap \text{node.predicates}|}{|d.\text{capabilities}|}$

The assigning algorithm uses the device's capabilities and each node's *Predicates* and *Priorities* to assign *nodes* to devices. With a greedy approach, the algorithm filters the devices that comply with each node's *predicates* and assigns the one having the best fitness (*cf.* Algorithm 11.1, p. 204). The fitness takes into account the number of *priorities* the device can provide ($\alpha = 0.5$), the number of already assigned *nodes* the device has ($\beta = 0.4$), and the specialization of a device ($\gamma = 0.1$) — meaning that a device capable of satisfying *priorities* not requested by a given node should be left for a next node that might request them (*i.e.*, need them) in the future. We decided to opt for these particular values of these hyper-parameters, as they performed well in preliminary tests; their optimization is out-of-scope of this work. The goal is to assign each *node* to the best possible device, spreading the tasks through all the available devices. An example of a possible assignment can be seen in Figure 11.3 (p. 205), where the assignment matches the nodes' *priorities* with the devices' tags while spreading the *nodes* over the devices.

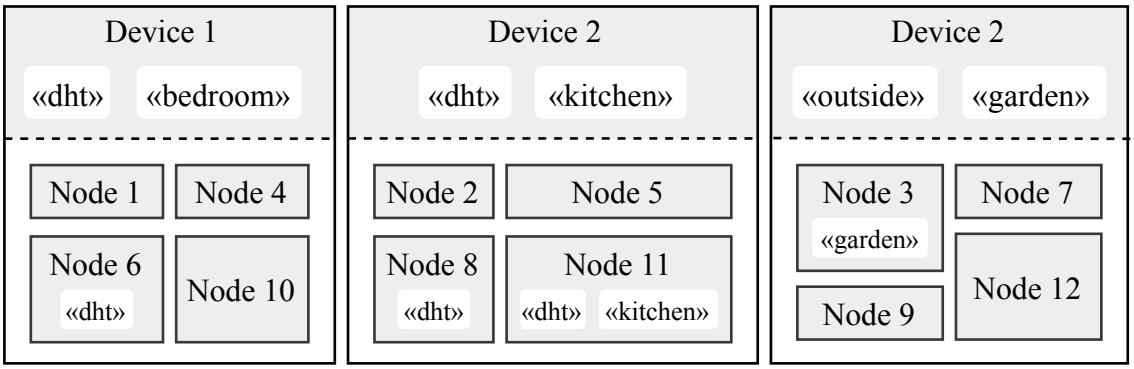


Figure 11.3: Node assignment representative resulting example after an successfull orchestration.

After assigning all *nodes* to a specific device, a code script is generated for each one (*cf.* Section 85, p. 203). Due to the constrained memory of the devices, the number of *nodes* assigned to a device may exceed their resources. In that case, it will `Fail-Safe` and return an error to the assignment request. The orchestrator will receive this information and repeat the process, assigning fewer *nodes* to the ones that returned an `Out-of-Memory` error. If a device does not return any response, the orchestrator will assume that the device is unavailable and not assign any *node* to it.

The *orchestrator node* can be triggered — proceeding to a system (re)orchestration — by the following events: (1) start of the system, when there is already a defined *flow* in the configuration, the assignment start after a period of 3s, to give time for the devices to be registered by the registry node, (2) deployment of the entire *flow* using the Node-RED editor or API, (3) appearance of a new device detected by the *Registry node*, and (4) failure or recovery of a device, which, working as a complement to the *Registry node*, is detected using Ping/Echo pattern [SK09] which periodically *pings* the devices in the system to assert their operational status.

11.2 Experiments and Results

We evaluate our approach in scenarios using both virtual and physical setups. Physical setups used ESP8266 and ESP32 devices connected to the same Wi-Fi network. Virtual setups used Docker containers with constrained resources. The experiments were performed in an i5-6600K at 3.5GHz with 16Gb RAM, Linux-based OS (kernel 5.6.16), Node-RED 1.0.6, Mosquitto 1.6.10, and MicroPython 1.12.

11.2.1 Experimental Scenarios

We defined two experimental scenarios. In **ES1**, a room has three sensors that provide temperature and humidity readings every minute. There is a virtual sensor that compares these readings and triggers depending on certain thresholds. An A/C reads it and decides (a) if it *switches on/off*, and (b) its operating mode: *cool*, *heat*, or *dehumidify*. The Minimal Working System consists of (a) one temperature sensor, (b) one humidity sensor, (c) one *node* capable of making the decision, and (d) a working communication channel among them. For **ES2**, the system has 20 devices that are responsible for propagating an injected message in a long chain of nodes until it reaches a specific sink MQTT topic. The goal of **ES1** is to isolate the features of our work with a moderately simple, although realistic, Node-RED *flow* (*cf.* Figure 11.4, p. 206). **ES2** aims to measure possible overheads of our solution.

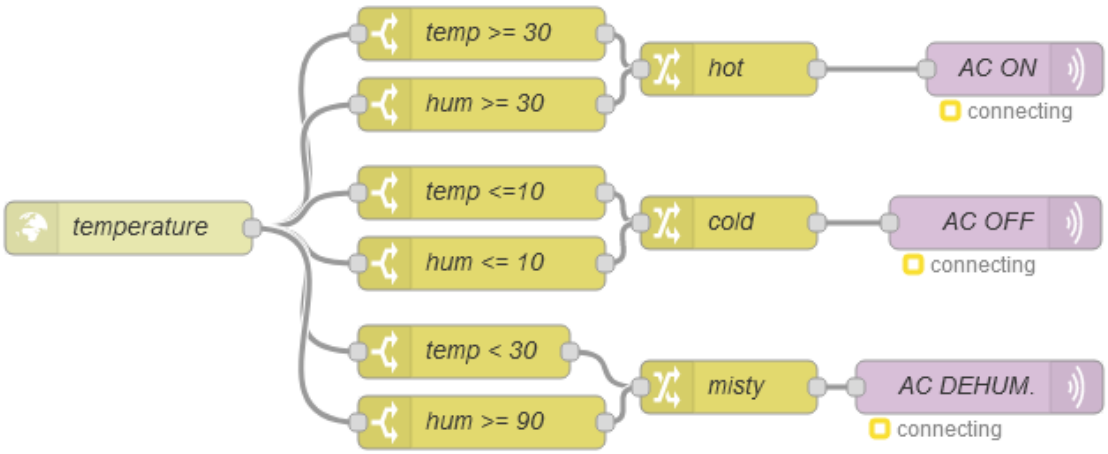


Figure 11.4: Partial *flow*, which is repeated three times to enable *consensus* and *fault-tolerance* strategies in Experimental Scenario 1 (ES1).

11.2.2 Experimental Tasks

For each one of the experimental scenarios (**ES1** and **ES2**), we defined a set of experimental tasks, detailed in the next paragraphs.

Experimental Scenario 1 (ES1)

Two sanity checks were performed, namely (**ES1-SC1**) with virtual devices and (**ES1-SC2**) with physical devices. A set of readings and message forwarding tasks were performed with no

compensation or any other fault-tolerance strategies. Each sensor only provided environmental readings to the system. Orchestration is centralized. We expect all roundtrips to take less than the smallest part that can be resolved (measurement capability estimated to be < 1s). We then defined a set of (re-)orchestration experiments where the system must allocate computation tasks among the available resources:

(A) Minimal working system (MWS) is achieved via multiple possible configurations by provoked device failure (fail-stop) using only virtual devices;

(B) MWS is achieved via multiple possible configurations by provoked device failure (fail-stop) using physical devices;

(C) Inconsistent device behavior, *e.g.*, appearing and disappearing in intervals shorter than the time needed for orchestrating convergence, possibly impacting the MWS;

(D) With four devices, each with different processing capabilities. During orchestration, some devices will throw an `Out-of-Memory` error because they cannot handle all the processing tasks assigned to them (*i.e.*, the size of the provided script). The orchestrator should decide to send fewer tasks to these devices and converge to a working solution;

(E) With four devices, some of them exhibit a memory leak from an unknown cause. These problematic devices stop working with an `Out-of-Memory` error at a random time. The orchestrator should assume these devices cannot handle the number of processing tasks assigned to them and assign them fewer tasks. Since the devices will keep breaking, the orchestrator should eventually ignore them;

(F) With four devices, there is a device that is sensitive to a particular *node*. Whenever the orchestrator assigns this *node* to that specific device, it throws an `Out-of-Memory` error. The orchestrator should eventually converge to a solution where the specific *node* is not assigned to that device;

(G) With 50 devices, there is a given probability of a particular device failing in each second. The downtime can go from 0s to 10s at random. The orchestrator must deal with the devices' failure and re-orchestrate. This experiment is considered a *stress* test, since it forces constant re-orchestrations.

During these experiments, we should verify that (a) **any restrictions (*predicates*) are enforced**, by checking every obtained configuration, and (b) that ***priorities* are honored**, by checking that all specified *priorities* were taken into account, and only violated if necessary. If specified *priorities* must be violated, (a) edge devices should be used first, and (b) the level of decentralization should be maximized by using the most available devices.

Experimental Scenario 2 (ES2)

Regarding **ES2**, a total of 20 devices were connected in a line topology. A message is sent to the starting device, which will propagate it to its output. All the devices implement this propagation

logic, which should result in the initial message reaching the end of the line. The propagation time is measured, starting when the message is sent and ending when the message reaches the last node. This scenario was implemented with different experimental configurations, namely:

(A) Non-modified version of Node-RED, using the default *node*-to-*node* communication channel (`EventEmitter`), with all the *nodes* sharing the same runtime;

(B) Modified version of Node-RED that uses MQTT as the *node*-to-*node* communication channel, with all the *nodes* sharing the same runtime;

(C) MQTT-based modified Node-RED, where each *node* of the *flow* is assigned to a different virtual device (*i.e.*, a MicroPython-running Docker instance). The Docker instances and MQTT broker run in the same host machine;

(D) MQTT-based modified Node-RED, where each *node* of the *flow* is assigned to a different virtual device. The Docker instances are in one host, but the MQTT broker is in another one. All parts are connected to the same Wi-Fi network;

(E) Each physical device runs a simple script that performs the desired behavior, on top of a non-modified MicroPython firmware image, communicating over MQTT. Node-RED is not used, and there is no orchestration being performed;

(F) MQTT-based modified Node-RED, along with the modified MicroPython firmware running on physical devices. Each *node* is assigned to a different device. The devices communicate by MQTT over the same Wi-Fi network.

11.3 Discussion

The results from the experimental tasks are presented, analyzed, and discussed in the following paragraphs. Data about the execution is collected and aggregated in Node-RED, based in telemetry messages from the devices. The payload size corresponds to the size of the payload sent to the devices from the orchestrator perspective.

11.3.1 Scenario 1 (ES1) Sanity Checks

To assert the correct functioning of the dynamic orchestration approach under normal operation conditions two sanity checks were defined and run, as detailed in the next paragraphs. These were mostly to observe if the approach performed similarly in both virtual and physical devices.

Virtual Devices Sanity Check (ES1-SC1)

This experiment was used to observe the overall approach in a controlled fashion. By using virtual devices we reduced the chance of hardware failures. The free flash size decreases by ≈150Kb when the device receives a script for executing, *i.e.*, matching the size of the payload. As the orchestrator assigns the *nodes*, the corresponding scripts are built and sent to them.

The time it takes to deliver the script averaged 0.303 ± 0.165s. All exchanged messages were captured, which enabled us to check that the system behavior was the expected by (1) spreading the computation among available resources, and (2) resulting in a system with the expected functional behavior.

Physical Devices Sanity Check (ES1-SC2)

The previous experiment was repeated using physical devices (four ESP32 devices). The orchestrator attributed 9 *nodes* to each device. The RAM usage in physical devices was smaller than in virtual devices[1]. The free flash space was also smaller, as expected. The time to deliver the script was longer than in the first experiment, averaging 6.776 ± 0.476s. This is mainly caused by the *nodes* being in different devices, with the Wi-Fi communication and hardware specs having a non-negligible impact.

11.3.2 Scenario 1 (ES1) Experimental Tasks

The following paragraphs detail the experiments carried using as reference the Experimental Scenario 1 (ES1).

Simulated Device Failure (ES1-A)

This experiment evaluates if the system is able to re-orchestrate when a device fails. A set of virtual devices were turned off one by one until only one was left running. It was expected for the system to detect when a device became unavailable and to re-orchestrate by assigning *nodes* to the remaining devices. In the end, we expected only one device to be running, with all the *nodes* assigned to it. Figure 11.5 (p. 210) shows the uptime of the devices, allowing us to identify the moment each one fails. We can also observe an increase in the payload size and the number of allocated *nodes* in the remaining devices each time a device was turned off.

Physical Device Failure (ES1-B)

This experiment repeats **ES1-A** with physical devices. The payload size is similar (with minimal differences due to inner works of MicroPython in the virtualized *versus* physical environments) and number of *nodes* assigned through the experiment is the same (*cf.* Figure 11.6, p. 211). However, we observe that *Device 2* (the last remaining active), first fails when receiving the payload containing the code for all the *nodes* of the system. This was *expected*, as its constrained memory cannot handle the full payload. It then enters a `Fail-Safe` state, reporting an `Out-of-Memory` error, and forcing the *Orchestrator* to assign it fewer *nodes*.

Early Device Failure (ES1-C)

Similar to **ES1-A** and **ES1-B**, this experiment focuses on testing the system's ability to recover when devices fail and then recover. In Figure 11.7 (p. 212), we can observe *Device 3* and *Device 4*

[1] Which may be due to optimization differences between the Docker-compatible and ESP-compatible MicroPython firmware, garbage collector runs, and libraries.

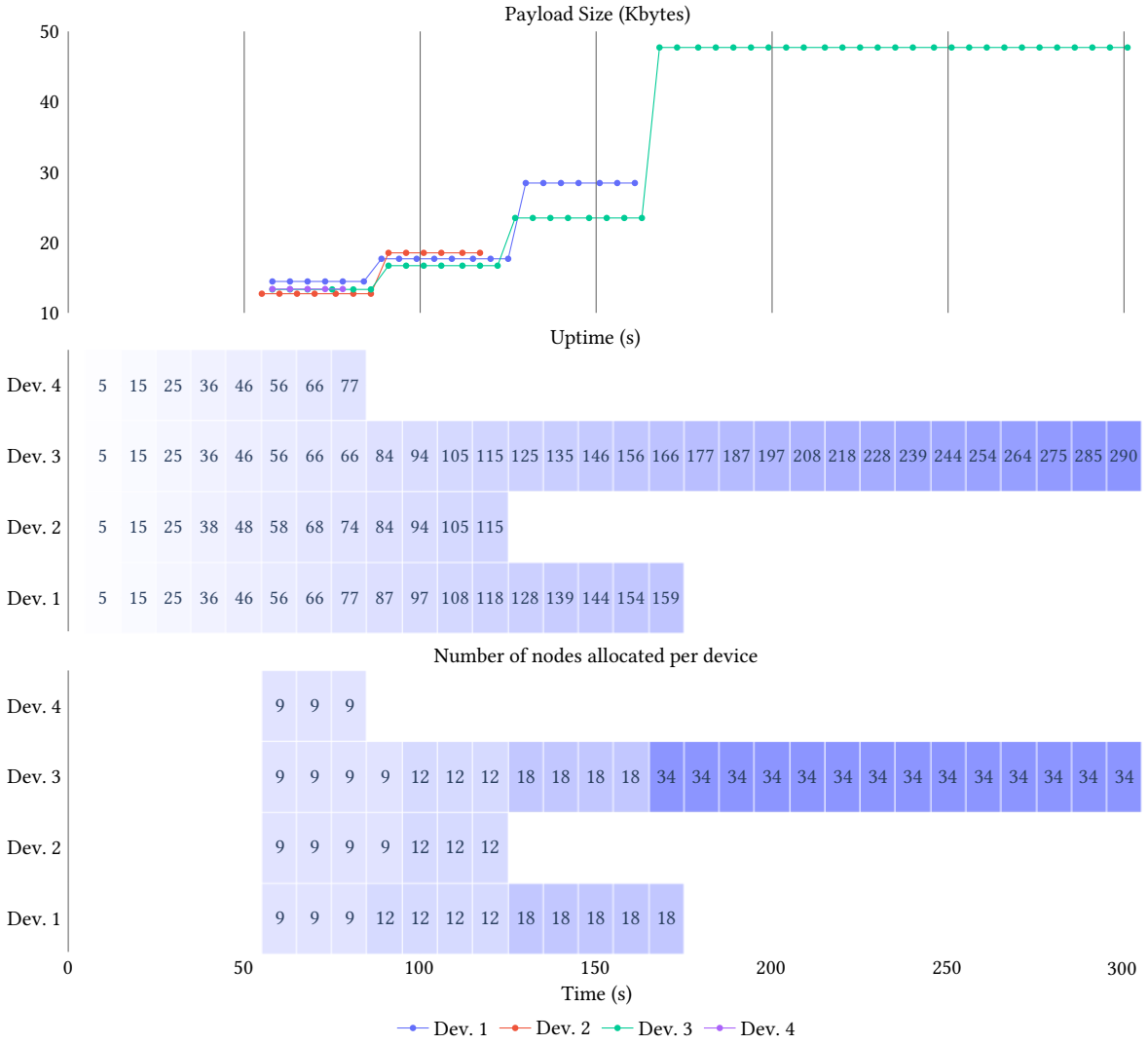


Figure 11.5: Simulated Device Failure (ES1-A) measurements from the perspective of the orchestrator.

failing prematurely. The system recovers by assigning the corresponding *nodes* to other devices. *Device 4* then (1) recovers around 100s, (2) fails again, and (3) recovers. The system disregards this swift failure and only re-orchestrates the second time *Device 4* recovers. During this experiment, *Device 3* and *Device 4* continue to fail and recover in a predictable pattern, and the system keeps re-configuring itself. The precise decision heuristic might need further investigation, as a device that enters a fail/recover loop might introduce continuous re-orchestrations (essentially an unintended Denial-of-Service).

Out-of-Memory Issues (ES1-D)

The memory constraints of IoT devices can negatively impact the functioning of the system by raising out-of-memory errors when writing the received script into the device's SPI flash. This experiment assesses how the system recovers and adapts to these conditions. Figure 11.8 (p. 213) depicts the system behavior due to the constrained memory of *Devices 2* and *4*. When the first assignment is made, at ≈50s, both these devices enter a `Fail-Safe` state due

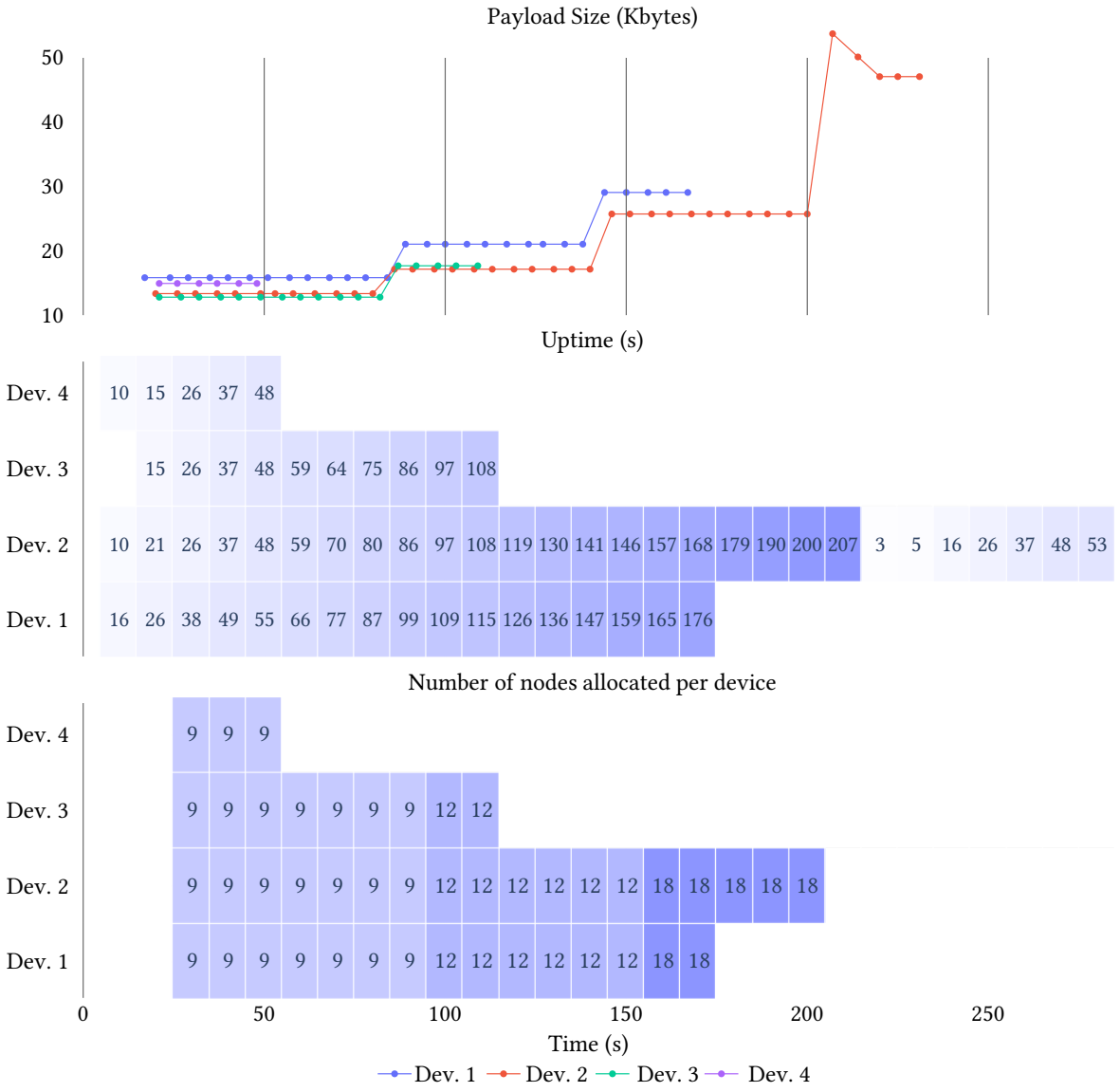


Figure 11.6: Physical Device Failure experiment results (ES1-B) from the perspective of the orchestrator.

to `Out-of-Memory` errors. The number of *nodes* present on these devices are the ones assigned after they communicate to the orchestrator their limitations. We then turn *Device 2* off, and later on. As it can be observed, once *Device 2* stops, *nodes* are distributed to the other devices, except for *Device 4*, which is memory constrained. After the recovery of *Device 2*, the system re-orchestrates, and the same number of *nodes* are assigned to the devices. The fact that *Device 4* fails after *Device 2* recovered implies that the system repeated the original assignment decision, ignoring previously known information about memory constraints. This is a known limitation to be addressed in the future.

Memory Leak Issues (ES1-E)

Besides memory limitations, we also expect the system to be capable of handling unhealthy devices with memory leaks. *Device 2* was modified to always generate an `Out-of-Memory`

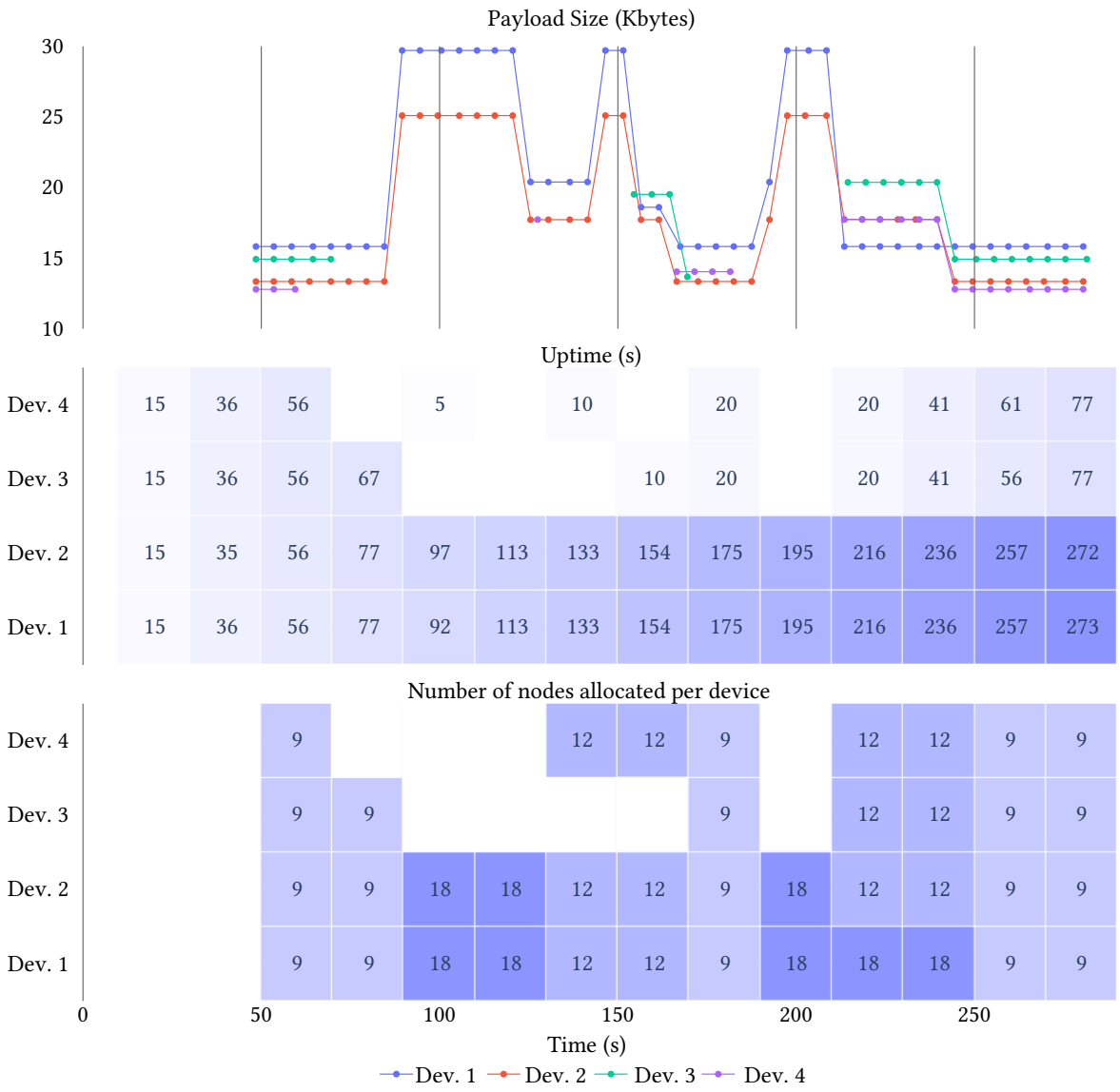


Figure 11.7: Early Device Failure (ES1-C) measurements from the perspective of the orchestrator.

error after a random period. We expected the system to eventually exclude this device during the assignment process. The Figure 11.9 (p. 214) shows that *Device 2* consistently fails after the first assignment of *nodes* at ≈75s. The number of *nodes* assigned keeps decreasing until the device is excluded from consideration. This is currently a simple process, in which the system will decrease the number of *nodes* it assigns to a device every time it reports an `Out-of-Memory` to the orchestrator. Once the minimum number of *nodes* reaches zero, the device is excluded from the assignment process.

Sporadic Fault Injection (ES1-F)

To further assess the resilience of the system, *nodes* causing errors in specific devices were deliberately added. We expected for the system to re-orchestrate and converge to a solution

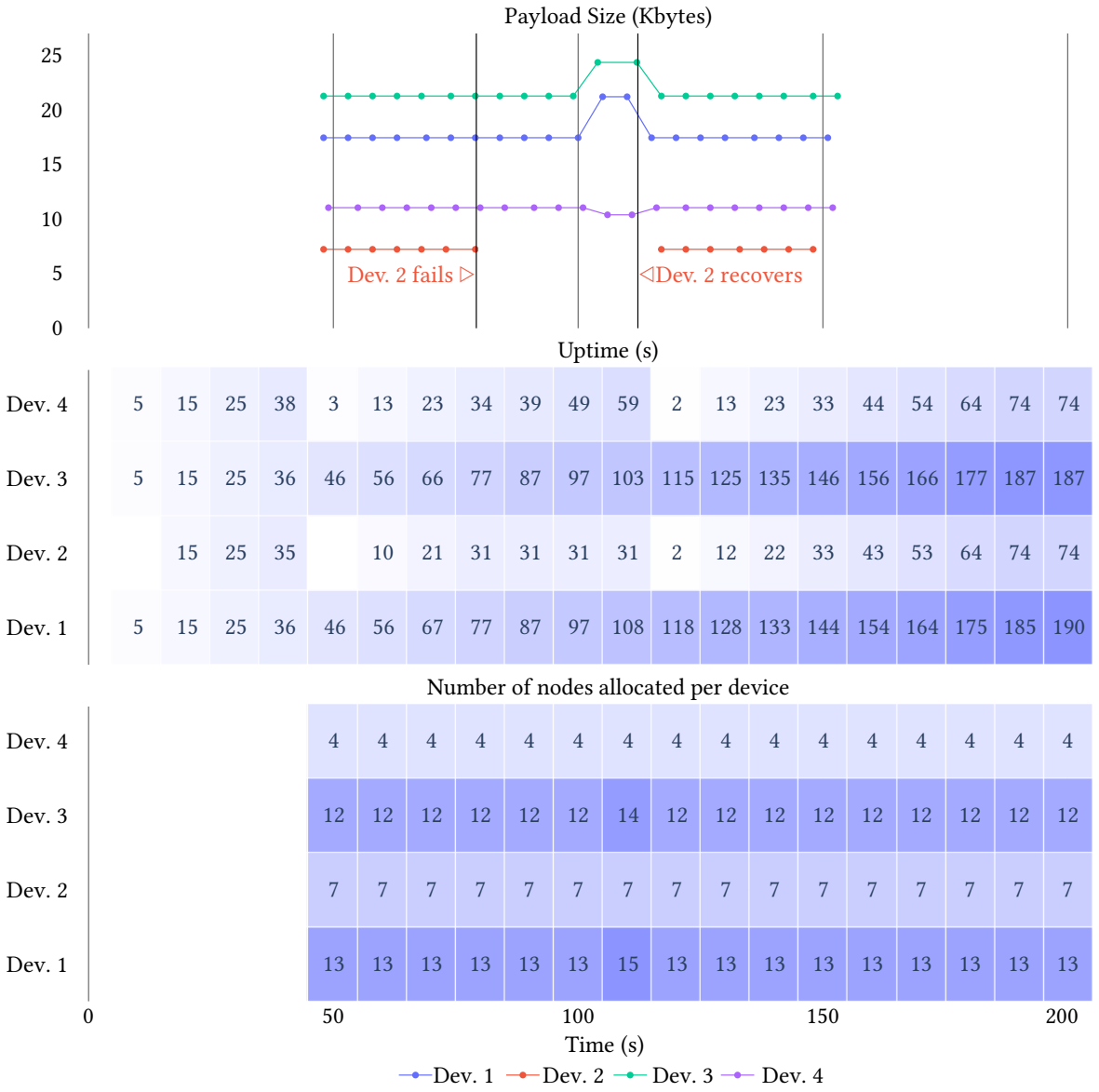


Figure 11.8: Out-of-Memory Issues (ES1-D) measurements from the perspective of the orchestrator.

where those specific *nodes* were assigned to devices not affected by the problem. Neither the system nor the devices know which specific *node*/device combinations are causing the faulty behavior, so this scenario is overall interpreted by the orchestrator as a device problem. As the first assignment could be correct by sheer chance, we forced a system re-orchestration by turning off and on all devices in a random order and repeating the process three times. The Figure 11.10 (p. 215) shows these on/off events at the ≈125s, ≈200s and ≈275s timestamps. In this case, the devices affected by the faulty *nodes* were *Device 2* and *Device 4*. The event we aim to test occurs at ≈300s. As can be seen in Figure 11.10 (p. 215), 10 *nodes* are assigned to *Device 4*. The uptime of *Device 4* resets in this small-time period (the next uptime is less than 20s), meaning that an `Out-of-Memory` occurred and the device entered a `Fail-Safe` state. The system updates, allocating the 10 *nodes* previously assigned to *Device 4* through all the available

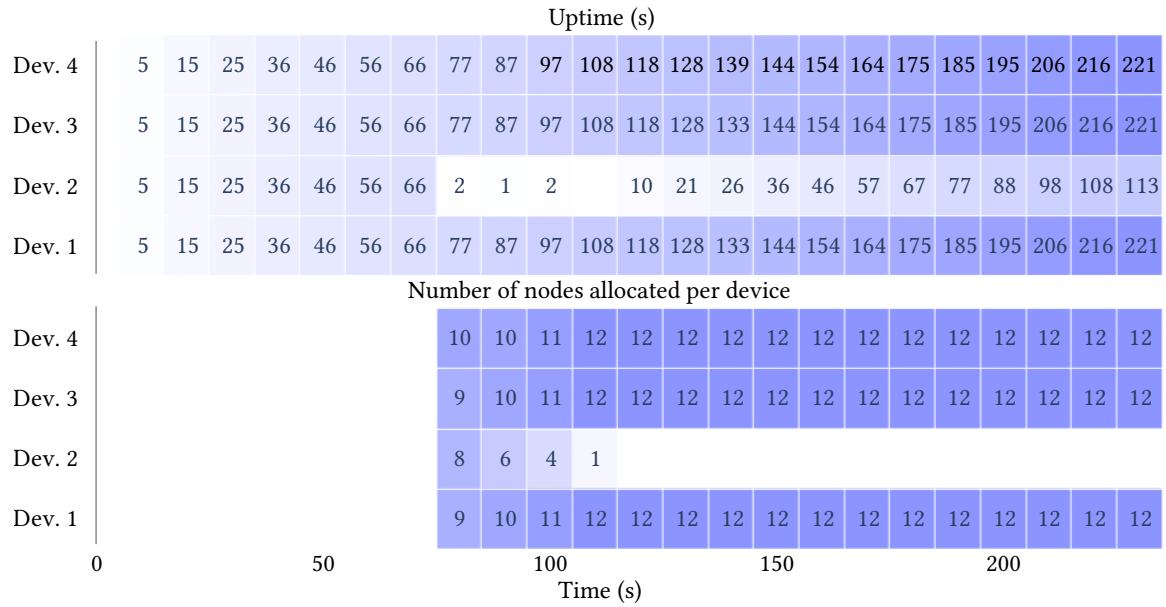


Figure 11.9: Nodes' assignment over time with memory leaks (ES1-E) from the perspective of the orchestrator.

devices. Since Figure 11.10 (p. 215) shows the data in intervals of 20s, the assignment in *Device 4* happens before the assignment present in the other devices. When the system receives information that *Device 4* is available again, it already knows that it has a limitation, so it only assigns 9 *nodes* to it. It can be seen that the missing *node* is assigned to *Device 1*. Since *Device 4* does not enter a `Fail-Safe` state, the *node* assigned to *Device 1* must have been the faulty one.

Recurrent Fault Injection (ES1-G)

To further investigate possible limitations in our current approach, we proceeded to inject constant failures in the available devices. Every second, each device has a $p = 5\%$ of becoming unavailable for 0–10s. During this period, the device becomes unresponsive, announcing itself only when it recovers. The Figure 11.11 (p. 216) shows that the system is kept continuously re-orchestrating. But once the majority of devices fail, the system becomes unstable, *i.e.*, the system keeps trying to allocate nodes across available devices (as it can be seen by the shifting number of nodes per device), even if the devices are unable to allocate them. It is important to note that, similar to previous experiments, once a device fails, the number of *nodes* does not update to zero. We conclude that devices with the same number of *nodes* during the total execution of the system failed early on and continued to fail, stopping the orchestrator assignment. Despite that at ≈100s, there is a period where all devices are available, the *orchestration* does not converge during that time. This is due to the system re-orchestrating whenever a device becomes available (since each device announces itself individually, each announcement triggers a new orchestration). This process takes time and results in several failed orchestrations due to outdated data on the device's operating status, being also taxing for the devices, causing an overload of received assignments that will stop the system function as a whole.

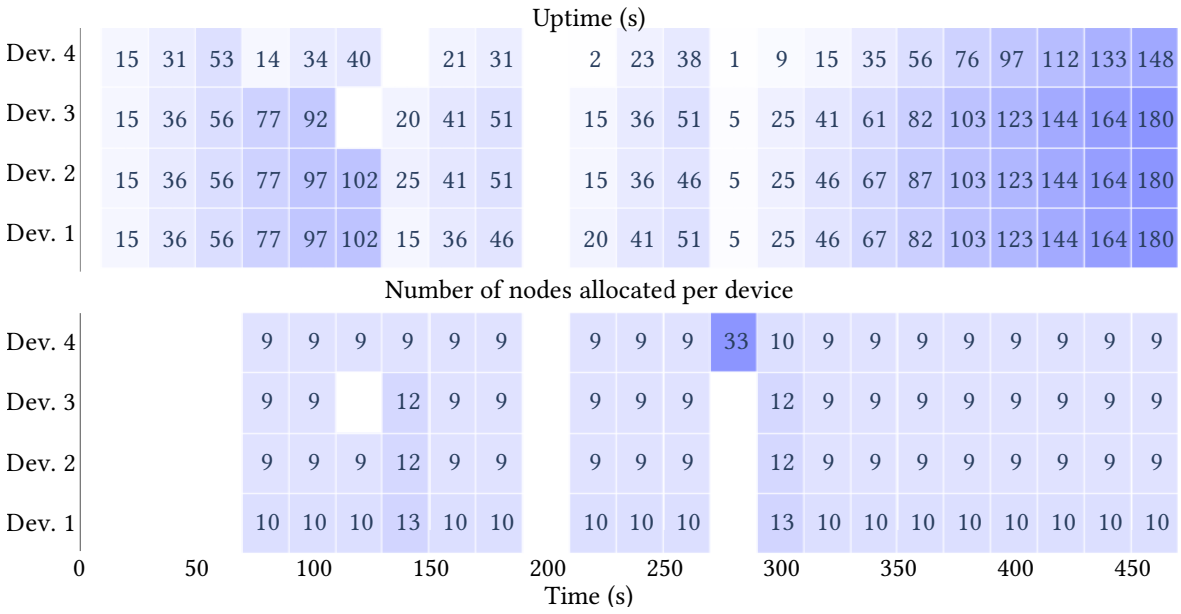


Figure 11.10: Nodes' assignment over time, from the perspective of the orchestrator, with fault-injection on the devices (ES1-F).

11.3.3 Scenario 2 (ES2) Experimental Tasks

To benchmark the impact of our approach, we proceeded to incrementally instrument Node-RED with partial implementations and measure each of them. Our setup consists of a *flow* that passes a message through several devices, recording the total round trip. The `NOP` *nodes*[2] execution consists of only redirecting their input to their output. A message containing only the current timestamp is inserted into the system by triggering the *Inject node*, and the same message is expected to appear in the Node-RED debug console.

Table 11.2: Approach benchmark (ES2) with elapsed time measurements.

Label	Min	Q1	Q2	Avg	Q3	Max
ES2-A	3	8	10	10	13	15
ES2-B	134	353	431	489	711	883
ES2-C	1217	1260	1318	1400	1574	1665
ES2-D	1445	2332	2536	2392	2708	3059
ES2-E	3616	4031	4142	4133	4372	4452
ES2-F	4168	4357	4569	4751	5088	5940

This experiment was run with different configurations (**ES2-A** to **ES2-F**) to assert the impact of each modification/module, as described in Section 11.2.2. Each experiment was replicated ten times, and the resulting measurements are shown in Table 11.2 (p. 215).

When the decentralization is applied inside Node-RED (*cf.* **ES2-B**), it is possible to see that the introduction of the MQTT communication (Mosquitto broker) running in the same host

[2] NOP is an assembly instruction for no-operation (no-op), an operation that does nothing.

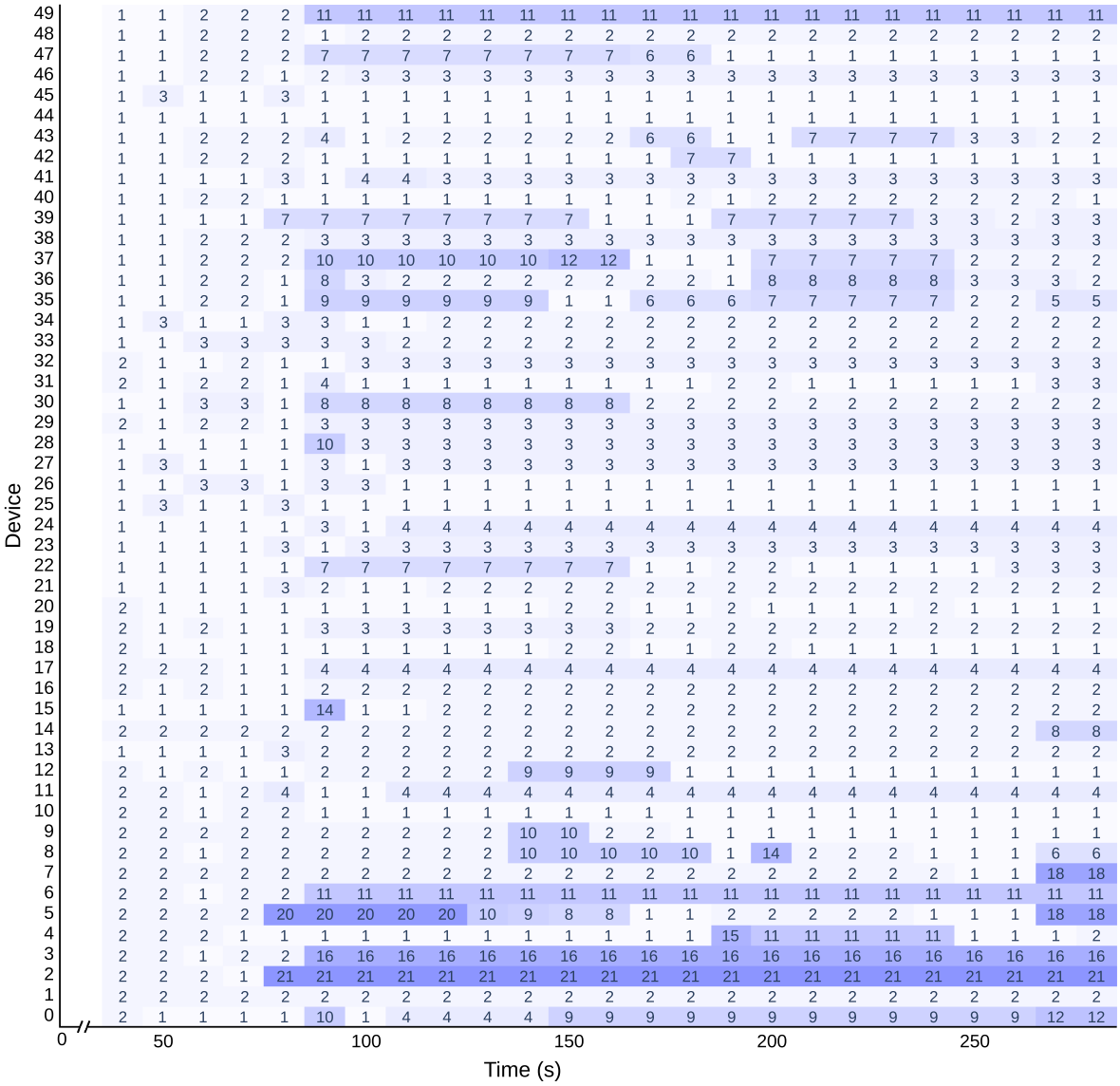


Figure 11.11: Nodes assignment distribution among available devices over time with recurrent device failures from the perspective of the orchestrator (ES1-G).

causes some latency. The introduction of Dockers running the firmware in the same host as the Node-RED instance and MQTT causes additional latency (*cf.* **ES2-C**), making it possible to conclude that the MicroPython-based developed firmware also delays the communication. By repeating the same experiment but with the broker running in another machine (same network) (*cf.* **ES2-D**), it is noticeable that the times are more spread out, and the overall latency of the system increases. As the Node-RED and the broker run in different machines connected over Wi-Fi, we conclude that this is the leading cause for the additional delay.

The experiment was repeated in physical devices: (1) by running a simple code in the MicroPython flashed devices and injection of messages directly in the broker (*cf.* **ES2-E**), and (2) by using our approach as a whole, *i.e.*, modified Node-RED and designed firmware (*cf.* **ES2-F**). The results reveal that the use of physical devices produces higher times (as expected) but that the developed firmware has little impact, visible by the comparison of their results. We conclude that our approach, including the *node*-to-*node* communication change, is slower than

the original Node-RED, but it mostly results from the Wi-Fi communications and the base MicroPython firmware. Also, this modification makes Node-RED more *modular*, allowing the other communication mechanisms.

11.4 Summary

This chapter presented both a method and an extension to Node-RED that provides automatic decentralized orchestration of constrained devices in an IoT network. We proceeded to evaluate it through 2 experimental setups, divided into 13 scenarios. We have shown that our approach is able to provide a decentralized substrate for computation and dynamic adaptation of the system via self-reconfiguration.

There were three main quality attributes which were targeted as improvement points: (1) resilience, to which we provide evidence that it is moderately robust, handling device failures and memory constraints dynamically; (2) elasticity, by showing a moderate-sized IoT system functioning in a decentralized fashion with devices being added and removed in runtime; and (3) efficiency, where we investigate the overheads introduced by our approach and conclude that most of them come from the extra latency introduced by the communication channels, and the proposed firmware has little to no impact.

Some limitations of our approach include: (1) the algorithm used to orchestrate the *nodes* among the available resources can fail to find a suitable configuration, (2) there's room for optimizations, *e.g.*, bypassing the communication substrate between nodes assigned to the same device (although this might hinder observability) and trying to increase the likelihood of such sets of nodes being assigned to the same device (via static and/or dynamic analysis of the data flow [Mat+20]), (3) other firmware approaches could be explored beyond MicroPython, and (4) the (re)orchestration currently redeploys the entire system and does not attempt to take into consideration a set of minimal changes.

12 | Self-Healing for IoT

As systems' complexity increases, it inevitably results in people becoming *"overwhelmed by the effort to properly control the assembled collection,"* [PD11] increasing the probability of human-induced errors and failures; and developing becomes challenging, labor-intensive, and expensive [JEC14].

Providing systems with the ability to reconfigure to recover from, or, at least, mitigate the impact of, failures introduced by faulty parts, misconfigurations, or any other disruption, can improve the systems' dependability, even when their complexity increases. Achieving this requires that the running system to be able to inspect itself to identify the faulty components during its operation (*i.e.*, runtime) without the need for human inspection. In this chapter, we provide the foundations to enable users to visually model diagnosis and recovery/maintenance of health mechanisms to improve IoT systems' reliability, thus enabling them to be *self-healing*. These mechanisms have been developed, applied, and tested as extensions that we named Self-Healing Extensions for Node-RED (SHEN). We validated our approach by executing a set of scenarios on top of a live, physical setup, called *SmartLab*. The in-place-based system was first upgraded with the designed extensions; then, a set of common scenarios was executed, and the resulting system behavior was observed. Our experiments present supporting evidence to the feasibility of the approach, showcasing improvements in terms of system reliability and dependability, despite several limitations and challenges that this particular VPL language poses and which limit the full potential of our approach.

> Parts of this chapter were published in the work A Pattern-Language for Self-Healing Internet-of-Things Systems [Dia+20a] and in the subsequent works Visual Self-Healing Modelling for Reliable Internet-of-Things Systems [Dia+20b], Empowering Visual Internet-of-Things Mashups with Self-Healing Capabilities [DRF21], and Evaluation of IoT Self-healing Mechanisms using Fault-Injection in Message Brokers [Dua+22]. Miguel Duarte carried additional experiments in the context of the Master's thesis entitled MQTT Chaos Engineering for Self-Healing IoT Systems [Dua21].

12.1 Approach Overview

Node-RED has several limitations regarding the verification of *flows* — which limits runtime verification, by not providing *out-of-the-box nodes* capable of doing these tasks. Thus, we encapsulate runtime verification and self-healing mechanisms into a new set of Node-RED *nodes*. The following paragraphs detail our approach and implementation, by detailing how certain *nodes* correspond to the implementation of one or more of the self-healing pattern presented in previous chapters (*cf.* Chapter 6, p. 144).

Regarding runtime verification capabilities, we created *nodes* that allow inspecting the SUT, *i.e.*, *probing* the system (*cf.* Figure 2.13, p. 54), including the test patterns presented in Pontes *et al.* [PLF18] and detailed in § 12.2 (p. 221). Some devices and services (*e.g.*, message brokers, data stores, third-party services) can only be tested by implementing *black-box* reachability checking, such as the new `MQTTBrokerTimeout` node that asserts if the broker is still alive.

Following the self-healing loop described by Psair *et al.* [PD11], our *detection* component is composed by nodes that allow *runtime testing* and provide *diagnosis* information (*cf.* Figure 2.13, p. 54), after which the *recovery* process is accomplished by nodes that implement *maintenance of health* and *recovery* mechanisms.

The list of implemented Node-RED *nodes* is presented in Table 12.1 (p. 220). These *nodes* which can correspond to one or more self-healing patterns of the presented pattern language (*cf.* Chapter 6, p. 144), or even to a specific use case of a general pattern (*e.g.*, `kalman-filter` *node*). Additionally, some *nodes* do both error detection and recovery (or maintenance of health tasks). As an example, consider the `compensate` *node*: if it is configured in *active* mode, it checks that sensor readings are coming at an expected frequency (viz. detection) and compensate missing values if they do not arrive as expected (viz. maintenace of health); however, if it is configured in *passive* mode, an external event (*e.g.*, a message with a pre-defined payload) is required for the *node* to emit compensation values.

The `compensate` node, which corresponds to an implementation of the Compensate pattern within the Node-RED boundaries, is a node that provides features in the view of maintaining the health of the system, compensating missing values — typically sensing data — using one of the available compensation strategies (*cf.* Algorithm C.2, p. 341). When configuring the system, the user can select the most suitable compensation strategy (*e.g.*, maximum, minimum, or average) of the last n (where $n > 1$) already read values. The user also defines the expected sensor reading periodicity to be able to detect disruptions[1], typically providing some extra margin to deal with network latency spikes. After deploying a Node-RED *flow* with this node connected to a sensor, it will always provide a value, even when the sensor stops producing values. Additionally, if of the user's interest, a confidence formula can be defined, that, taking into account the number of compensated values and the array of previous readings, can be used to stop the production of values when the confidence drops below a given threshold.

Some *nodes* leverage *meta-facilities* that allow changing a system's behavior during runtime. As Node-RED does not formally provide them, we found a workaround by resorting to its *external* REST API from *inside* our nodes, thus gaining the ability to create, delete and change

[1] It can be considered that the `compensate` node leverages Stable Timing pattern within itself.

Table 12.1: Self-Healing Extensions and their map to self-healing patterns (*cf.* Chapter 6, p. 144). The corresponding algorithms are presented in Appendix C (p. 340).

Node name	Description	Enabled Patterns
`action--audit`	After a certain event happens, a secondary event that acknowledges the action is waited for until a pre-defined timeout occurs (*cf.* Algorithm C.11, p. 346).	Timeout, Action Audit
`balancing`	Distributes messages among available resources using a *Round Robin*, *Weighted Round Robin*, or *Random* strategy (*cf.* Algorithm C.15, p. 349).	Balancing, Redundancy
`checkpoint`	Stores the last input message of a node, replaying it in case of Node-RED failure, typically with a time-to-live threshold (*cf.* Algorithm C.9, p. 345) .	Checkpoint
`compensate`	Compensate missing values using pre-defined strategies (*e.g.*, minimum, average or maximum) complying with the expected readings periodicity. Also provides a confidence level based on the number of consequent compensations (*cf.* Algorithm C.2, p. 341).	Compensate
`debounce`	Adjusts periodicity of messages to meet target periodicity requirements by operating as a rate-limit with aggregation features (*cf.* Algorithm C.12, p. 347).	Stable Timing, Debounce, Timebox
`flow--control`	Enable/disable *flows*, allowing to adapt to changes/disruptions in the system (*cf.* Algorithm C.6, p. 343).	Circumvent And Isolate, Runtime Adaptation
`heartbeat`	Heartbeat that check the alive status of system parts connected over HTTP and MQTT (*cf.* Algorithm C.10, p. 345) .	Within Reach, Timeout, Unimpaired Connectivity
`http-aware`	Periodically probes the network for running services (*i.e.*, open ports) in order to find changes, *i.e.*, new or disconnected service (*cf.* Algorithm C.13, p. 348).	Within Reach, Unimpaired Connectivity, Device Registry
`kalman-filter`	Kalman filter [BRH16] uses statistical predictors to reduce the effect of random noise on measurements (*cf.* Algorithm C.1, p. 340).	Compensate
`network-aware`	Periodically scan of the local network for finding new or disconnected devices and hosts (*cf.* Algorithm C.17, p. 351).	Within Reach, Unimpaired Connectivity, Device Registry
`redundancy`	Manage redundant instances of Node-RED, setting a new master instance in the case of disruption of a master instance and reconfigure in case of recovery (*cf.* Algorithm C.8, p. 344).	Redundancy
`readings-watcher`	Check if sequential sensor readings are meaningful and correct by checking for minimum changes, maximum changes or stuck-at anomaly, *i.e.*, same sequential reading (*cf.* Algorithm C.18, p. 352).	Reasonable Values
`replication-voter`	Selects a value (message) taking into account several input messages (*e.g.*, array of sensor readings), based on a consensus, *e.g.*, majority (*cf.* Algorithm C.5, p. 342).	Redundancy, Diversity
`resource-monitor`	Checks telemetry data reported by the different system parts against near-maximum (or near-minimum) thresholds (*cf.* Algorithm C.14, p. 348).	Resource Monitor
`threshold-check`	Checks if measurements (*e.g.*, sensor readings) are within the operational specifications of the device. Also can be used to check if the surrounding conditions allow correct device operation (*cf.* Algorithm C.3, p. 341).	Reasonable Values, Suitable Conditions
`timing--check`	Checks if the periodicity of incoming messages matches the expected rate (*cf.* Algorithm C.16, p. 350).	Unsurprising Activity
`device-registry`	Maintains a registry within Node-RED with all devices in the system which adapts and triggers events as connected devices change (*cf.* Algorithm C.7, p. 343).	Within Reach, Device Registry

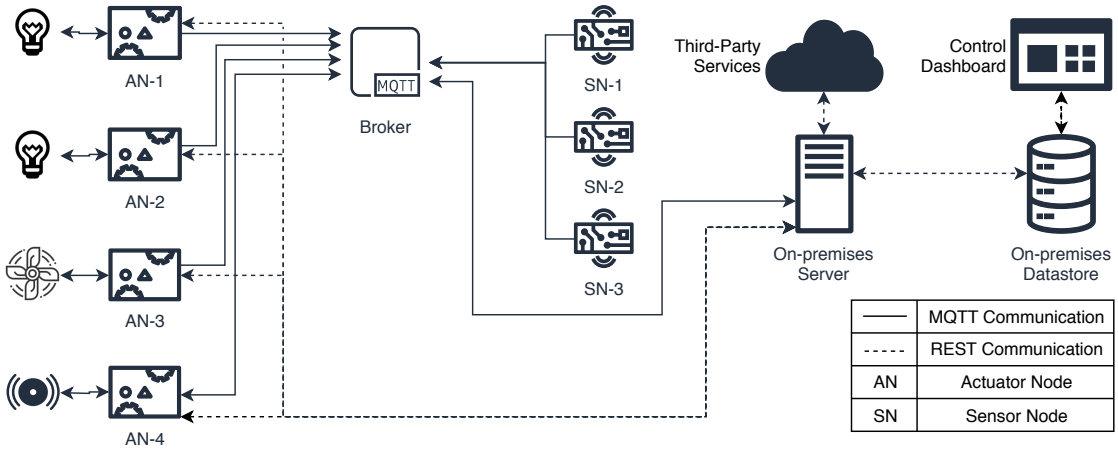


Figure 12.1: System component diagram, showing the main system parts, along with the different devices (actuators and sensors) and the enabling communication protocols.

the configuration of *flows* and other *nodes*. This is exemplified by the `flow-control` node, which allows toggling *flows on* and *off*, allowing Runtime Adaption.

An example of this is the `flow-control` *node* (*cf.* Algorithm C.6, p. 343). While the node's internal algorithm is rather minimal, it allows redefining the behavior of Node-RED at runtime within itself (meta-programming). Several *flows* can be defined in a Node-RED instance that carry rather similar tasks using different strategies or pathways (*e.g.*, turn on the lights using Wi-Fi or Bluetooth). While it is mostly irrelevant to have two *flows* carrying the same task simultaneously — which could, eventually, provoke malfunctions —, having these two mutually exclusive *flows* can be useful to fulfill system goals when some disruption happens. The `flow-control` node allows, in this sense, to enable or disable particular *flows* at runtime (*cf.* Runtime Adaptation), which can even be running in a different host. Using the same example, if Wi-Fi is not available, the task of turning on the lights can be carried by another wireless medium (*e.g.*, 433Mhz).

There are also some *self-healing* patterns which do not have a direct representation as Node-RED *nodes*, since they depend on specific edge device features and capabilities. As an example, to implement patterns such as Flash, Reset and Calibrate, the target device should be capable of receiving and execute instructions that accomplish the expected outcome: for the Flash, the device should be capable of receiving (*e.g.*, OTA) a new firmware version and update itself; the Reset also requires that the device is able to receiving and execute a reset instruction (*e.g.*, rolling back to factory defaults); and the Calibrate most of the time requires human-interaction to adjust mechanical parts.

12.2 Experiments and Results

Validating new solutions for runtime verification and self-healing requires scenarios representative of the characteristics, issues, and challenges of real-world IoT environments, such as heterogeneity and real-time needs. We carried experiments on *SmartLab*, an experimental *testbed* with four actuators and three sensing devices (each having more than one sensor) deployed

in a laboratory (*cf.* Figure 12.1, p. 221), responsible for a set of user-interaction features. Additional experiments were carried using existing IoT system collected data, allowing to carry experiments considering operation of IoT devices along larger timelines, *i.e.*, using data collected during larger time windows by reducing the time between readings for experimental purposes.

12.2.1 Scenario-based Experiments

We devised six scenarios to demonstrate the necessity of runtime verification and self-healing mechanisms. Although these scenarios do not cover all possibilities, we believe them to be sufficient to illustrate the complexity, challenges, and, in this case, Node-RED limitations and trade-offs. They also showcase the feasibility of the solution for improving the dependability of IoT systems.

Unavailability of Message Broker

MQTT is the base of most of our *SmartLab* communications; thus, it needs a message broker. Typically, the defined *flows* are triggered when a new message is received (the *flow* subscribes to a specific topic). In this scenario (described in two versions in Figure 12.2, p. 222, and Figure 12.3, p. 223), the message broker is both the *bottleneck* and a Single point of failure of the system; if it fails, the functionality of the system is compromised.

To verify the availability of the broker (*i.e.*, health status), a *heartbeat* pattern was followed. Two different strategies can be followed to accomplish this: (1) an *active* strategy where the monitoring component (*i.e.*, Node-RED node) probes a system part (*e.g.*, message broker) and checks for an answer in a timely fashion, or (2) a *passive* strategy, where the monitored component proactively informs (*e.g.*, continuous flow of telemetry data) the monitoring component.

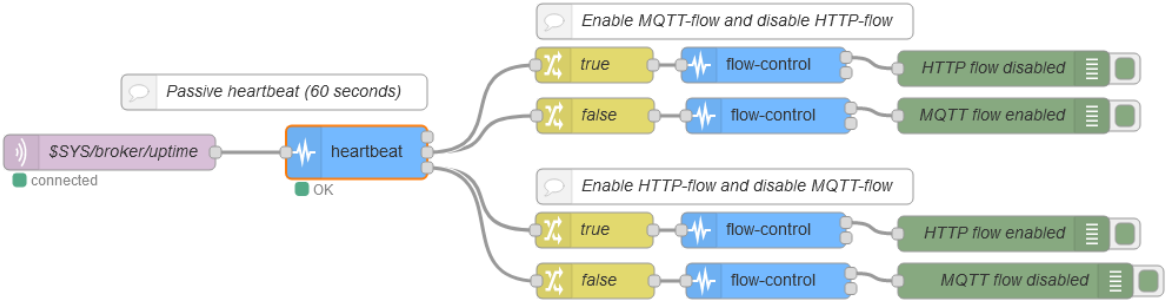


Figure 12.2: Usage of the `heartbeat` and `flow-control` nodes for detecting and healing from a broker failure, for detecting (*i.e.*, Unimpaired Connectivity or Within Reach) and healing (*i.e.*, Runtime Adaption) a potential unavailability of the broker.

In our *SmartLab* scenario, the two different strategies were put in place to assert their behavior. In Figure 12.3 (p. 223), a passive strategy was used, taking advantage of the MQTT *uptime* telemetry message which is periodically emitted by the message broker[2]. If the `heartbeat`

[2]Mosquitto MQTT broker emits an *uptime* message with an ≈60 seconds periodicity using the `$SYS/broker/uptime` topic.

node does not receive a message within the expected periodicity, an *error* message is emitted with the same periodicity in the node third output; otherwise, if the message is received, an *ok* message is emitted in the second output. Using that information, the `Report by Exception` Node-RED built-in node can be used to check for a broker status change and changing, in runtime, the communication protocol in use — by swapping between two different Node-RED *flows*, one that uses MQTT communication, and other that uses HTTP.

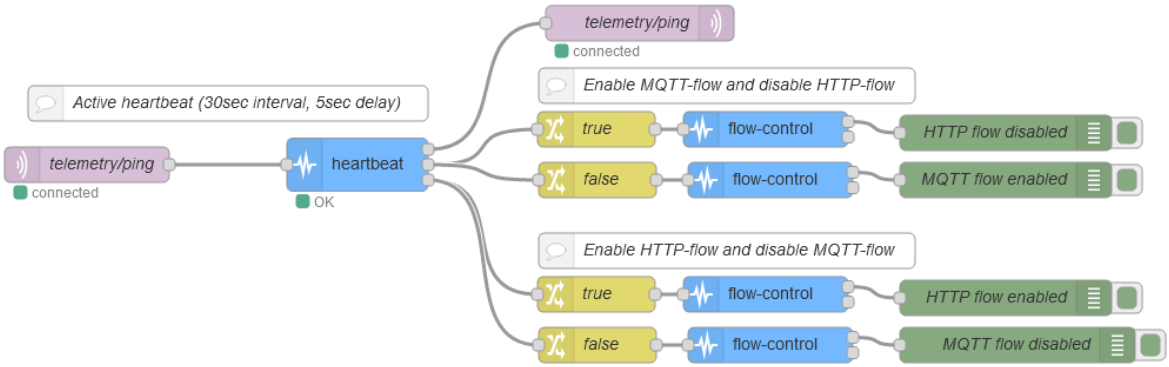


Figure 12.3: Usage of the `heartbeat` and `flow-control` nodes for detecting and healing from a broker failure, for detecting (*i.e.*, Unimpaired Connectivity or Within Reach) and healing (*i.e.*, Runtime Adaption) a potential unavailability of the broker.

As an alternative, an *active* strategy could be followed. In this case, while most the logic and implementation is similar, the first node output is used to send a *ping* message, which is then received by the same node *input*. This strategy is more suitable for publish-subscribe protocols in which a common topic can be used for routing the *ping* messages, but it is also possible to use in other protocols — depending on the proper implementation of a heartbeat mechanism by the probed system component.

Out-of-spec Sensor Scenario

SmartLab relies on the readings from different sensors so that it can act according to user-defined rules. As an example, if smoke is detected, an alarm or another notification mechanism should be triggered (and possibly trigger some contention mechanism like sprinklers). These

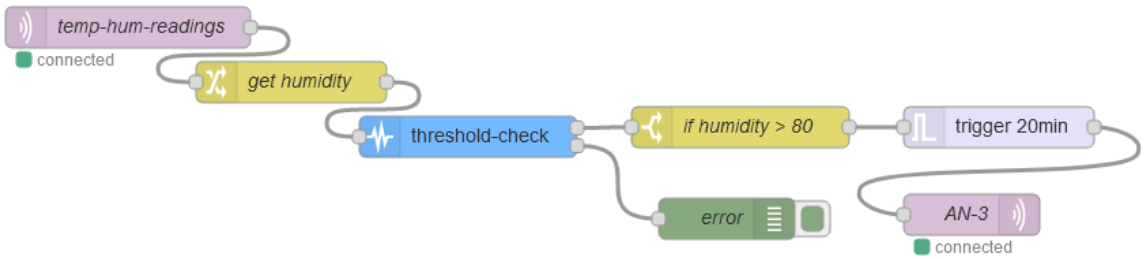


Figure 12.4: Usage of `threshold-check` in a *flow* that triggers an actuator if the humidity is above 80%, but verifies for correct sensor readings beforehand.

procedures depend on the *timeliness* and *correctness* of readings. Sensor malfunctioning can display an array of different behaviors, such as outputting *out-of-bound* or *out-of-spec* values; these can lead to wrong decisions and may end up having nefarious effects to the point of impacting the well-being of humans. Several strategies can be used separately or in combination to detect sensor malfunction. Sensors that provide periodic readings can be verified by analyzing the expected periodicity. Other errors, such as *out-of-bounds* and *out-of-spec* readings require customized verification and tailored failure conditions. Fortunately, these are usually available; *e.g.*, the DHT11 temperature/humidity sensor is capable of readings ranging from 0 °C to 50 °C, and 20% to 80% humidity. Values outside these ranges should be considered erroneous by default. In this scenario, an *isolation strategy* is followed; when an *out-of-spec* problem is detected, the readings are ignored via the `threshold-check` node. In the presence of redundant sensors, other readings may still be used by the system; otherwise, all the actuating components that depend on that sensor cease their activity (*cf.* Figure 12.4, p. 223).

Lost Action Scenario

The actuators that are deployed in the *SmartLab* depend on receiving messages to work as expected, with some of these devices supporting more than one communication protocol and having different levels of reliability (*i.e.*, the MTTF of some actuators is low, failing to comply with a request at least one time per day). Given that, in some situations, the devices are not accessible by the protocol used by default (*e.g.*, MQTT) due to connectivity disruptions, protocol *bugs*, or others, thus becoming inaccessible and eventually causing problematic side-effects (*e.g.*, cooling fan not turning on temperature increase). In other cases, the device does not trigger — a problem that could cause serious nefarious side-effects (*e.g.*, alarm not triggering). In these cases, verification can be carried after a certain amount of time, asserting if the request has been processed by the device, preferentially using an alternative data source (*e.g.*, a sensor device deployed in the surroundings).

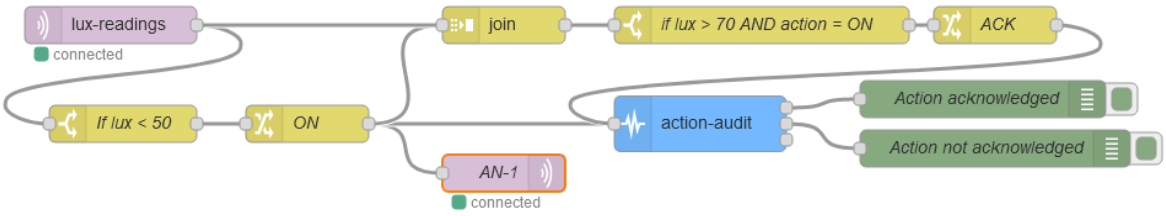


Figure 12.5: Usage of the `action-audit` to check if the lights turn on (request sent via MQTT) after a given interval, by checking if the luminosity goes above 70 lux. If the action is not acknowledged, a recovery action (*e.g.*, sending the command using another protocol or using a different actuator) can be used.

As a representative scenario consider Figure 12.5 (p. 224). After a state change request message is sent to an actuator (AN-1) via MQTT, an `action-audit` node can be used to check if the action was performed with different degrees of confidence. In this concrete scenario, a lightbulb should turn on if the luminosity of the space goes below 50 lux. Asserting if this action was performed is not easy since different factors can influence the luminosity of the space (*e.g.*, sunrise). However, having a sensor reporting the current luminosity value to allow us to have

some confidence that the action was indeed performed. If no change in the environment was detected, recovery actions could be triggered. As an example, if the light controlling device does not turn on the lights, as requested by the MQTT broker, a second request can be made to the same device, this time using HTTP.

However, even having a `action-audit` node, the *flow* incurs some essential complexity, mostly due to the limitation of Node-RED not supporting more than one input (which would make it more readable in terms of action/acknowledge). Further, we only want to consider the value of luminosity as an acknowledgment after an action was requested; thus, an additional, conditional block was required.

Having *diversity* on the IoT system allows to perform Runtime Adaption to the system, improving its capability to meet operational demands. As shown, having *things* that are capable of using different protocols allows us to adapt by dynamically switching to the most stable one given the systems' conditions (although usually incurring in a trade-off, such as the differences in energy consumption between MQTT and HTTP).

Sensor Failure Scenario

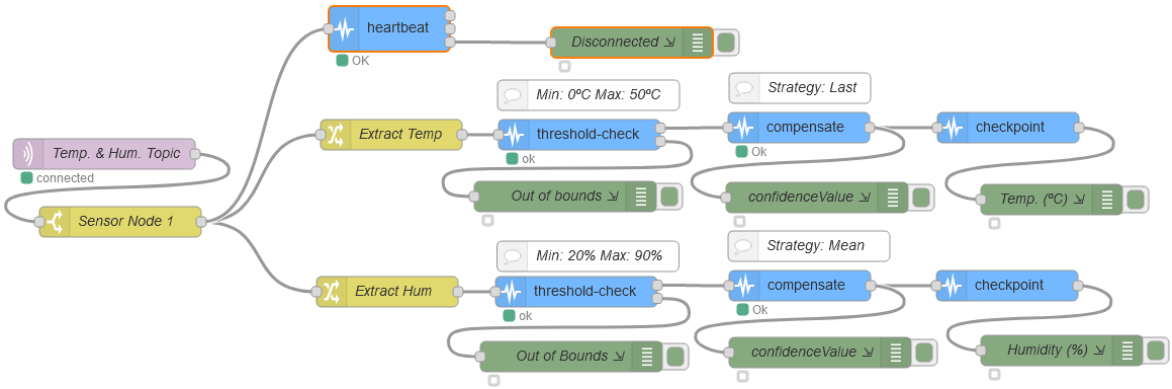


Figure 12.6: Experiment with failure of the sensing device and Compensate maintenance of health pattern, along with Heartbeat Probe.

A sensing node (*Sensor Node 1*) with a humidity and temperature sensor, connected over MQTT, producing values each 60 seconds, is connected to the Node-RED *flow* depict in Figure 12.6 (p. 225). The sensor is a DHT11, capable of reading temperatures in the range of $[0\,°\text{C}, 50\,°\text{C}]$ and humidity in the range $[20\%, 90\%]$. The configured *flow* ensures that:

1. the `threshold-check` (*cf.* Algorithm C.3, p. 341), verifies if both readings are within the expected values for the sensor, dropping values out-of-bounds;

2. the `compensate` (*cf.* Algorithm C.2, p. 341) verifies if, at some point, the sensor readings rate do not match the expected periodicity (*i.e.*, 60 seconds). If not matched, estimation is done, and a corresponding message is sent at the expected interval: (1) for the temperature, the last reading is re-sent, and (2) for humidity, the mean of the last ten readings are sent;

3. the `checkpoint` (*cf.* Algorithm C.3, p. 341) ensures that if there is any disruption that resets the Node-RED *flow* (*e.g.*, host reboot), the last message is re-sent (if within the message Time-to-live configured limit).

Also, in parallel, a passive `heartbeat` (*cf.* Algorithm C.10, p. 345) checks if the *Sensor Node 1* fails to produce any message within a given interval, triggering an error accordingly.

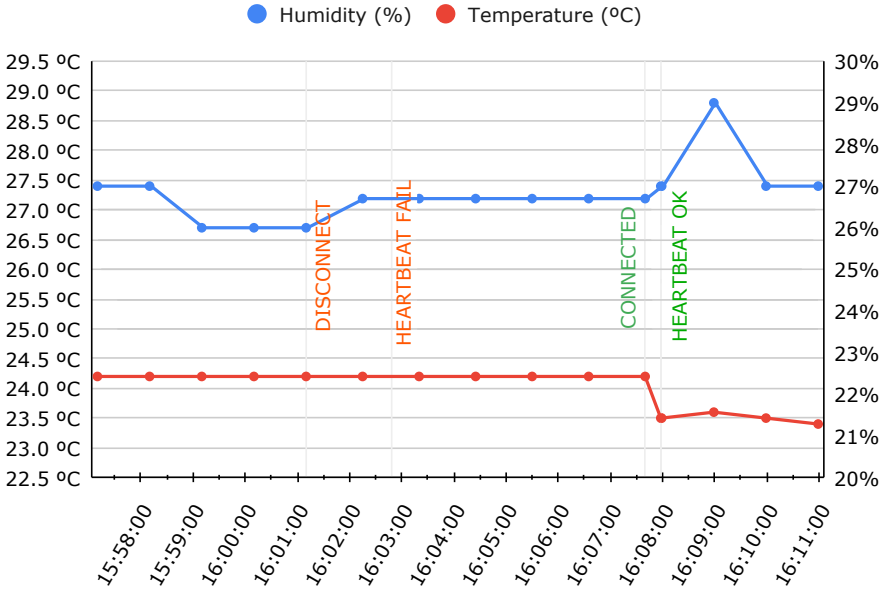


Figure 12.7: Experiment with failure of the sensing device and Compensate maintenance of health pattern, along with a Heartbeat probe.

As the first experiment, a total failure of a sensing node was replicated by disconnecting the sensor node from power at a random moment, and this can be seen in Figure 12.7 (p. 226). It can also be verified that soon after the sensor node being disconnected, the heartbeat fails. As expected, the `compensate` node triggers and compensates the missing values using the configured strategies. When the device reconnects, the `compensate` node stops producing values, and real readings are used. It is observable that when the device recovers there are two almost sequential readings, not matching the expected periodicity; this could be further managed — if required by the receiver node or device — using a `debounce` node to ensure that values are always at the same periodicity.

Load Spike Scenario

An access control device with an NFC reader is connected over MQTT to a Node-RED *flow* depict in Figure 12.8 (p. 227). The reader is placed at the entry point of the lab and ensures that every NFC card is validated using an external service, of which there are one primary host and two additional backup ones to be used in the case of exceptional usage spikes[3].

The *flow* in Figure 12.8 (p. 227) ensures that all the card validation requests happen in the quickest fashion. This is done by placing a number of self-healing *nodes* in the *flow*:

[3]For this experiment we considered a spike when more than one card is read in a 15-seconds window.

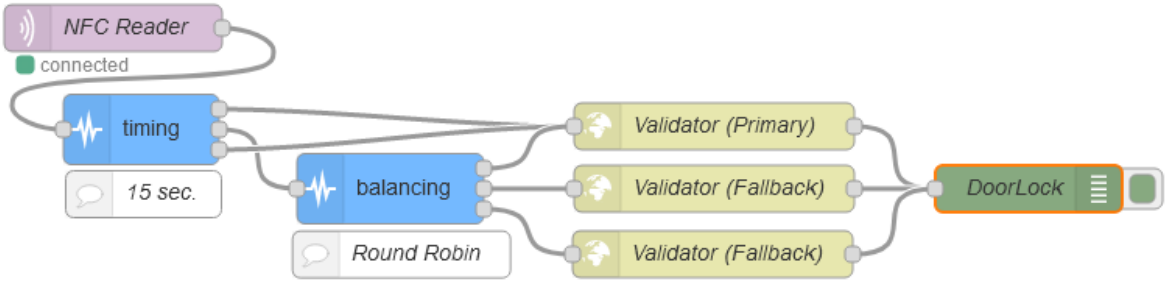


Figure 12.8: Balancing the validation of identity cards (NFC) via an external service (*e.g.*, HTTP request) when a load spike happens (increase in the number of cards swiped per unit of time).

1. the `timing-check` verifies the frequency at which the cards are being swiped in the NFC reader, categorizing (and splitting) them in accordance: *too fast*, *too slow* and *normal* (using as reference the 15-second estimated time between readings);

2. the `balancing` node, which handles readings that are coming as *too fast*, distributes them among the available *validators*, ensuring distribution in accordance to the configured strategy (*e.g.*, round-robin), thus reducing the load in the primary host.

This behavior is depicted in the marble diagram of Figure 12.9 (p. 228), replicating the behavior recorded at the testbed.

Redundancy Scenario

Although Redundancy is one of the most common patterns for fault-tolerant systems, having an implementation of it within Node-RED allows having recovery behaviors defined in the form of *flows* that go beyond simply turning on or off a whole Node-RED runtime in the traditional redundant unit fashion.

There are two instances of Node-RED running at different hosts, with a common *flow* that carries a common task: receiving sensor data from the *Sensor-Node-1* over MQTT (with the same frequency of 1 reading each minute), extract the temperature, asserting the validity of the data (`threshold-check` and `readings-watcher`) and post the data to an external service (an HTTP endpoint). The *flow* depicted in Figure 12.10 (p. 228) is deployed in both instances, running a consensus algorithm[5] to define a master in case of failure of the previously defined master instance.

Both Node-RED instances are running at all times, optimizing the mean time to recovery (MTTR) after a Node-RED instance crashes[6]. However, the *flow* is only active in one of them at any time (mutually exclusive). When the master instance crashes, an election occurs to devise a new master (which will work only until the old master instance recovers — if it recovers).

[4]The `rbe` (report-by-exception) node is part of the default Node-RED nodes and provides a way to pass a message only when it differs from the last one. In Node-RED version 2.0 this *node* is named `filter`.

[5]The consensus algorithm implemented selects as the master instance the instance running in the network machine with the highest last octet of the IP address.

[6]This is known as active-standby.

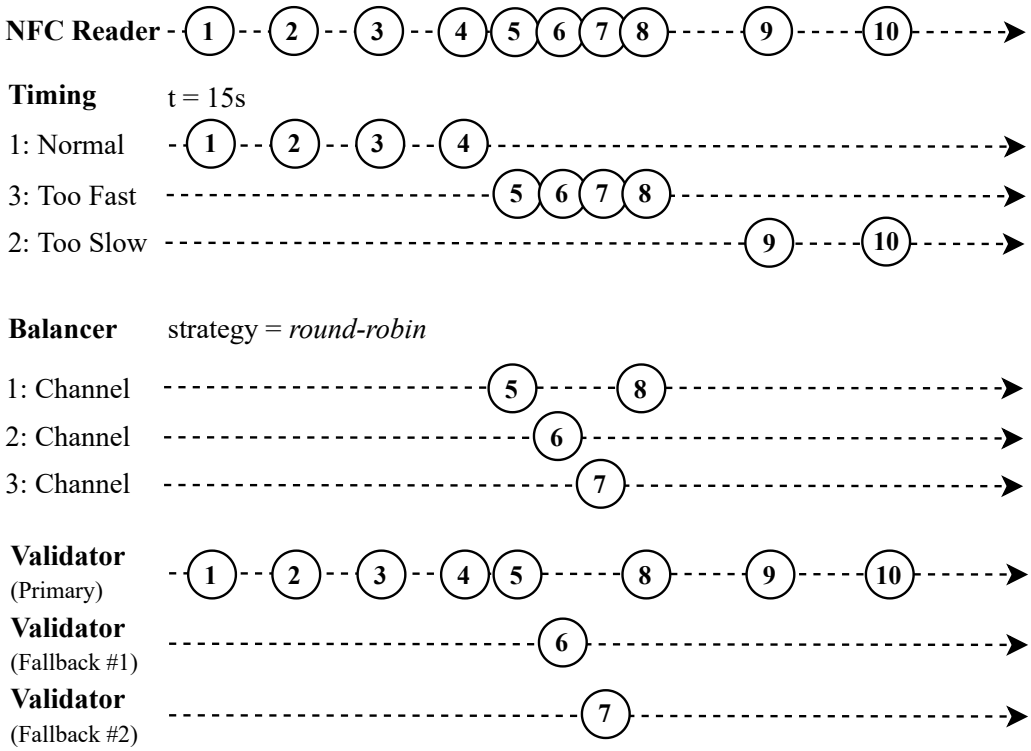


Figure 12.9: Marble diagram of the messages between the different nodes (output messages) on the *flow* of the load spike scenario, being the NFC reader the *producer* of messages and the validators the final *consumers*.

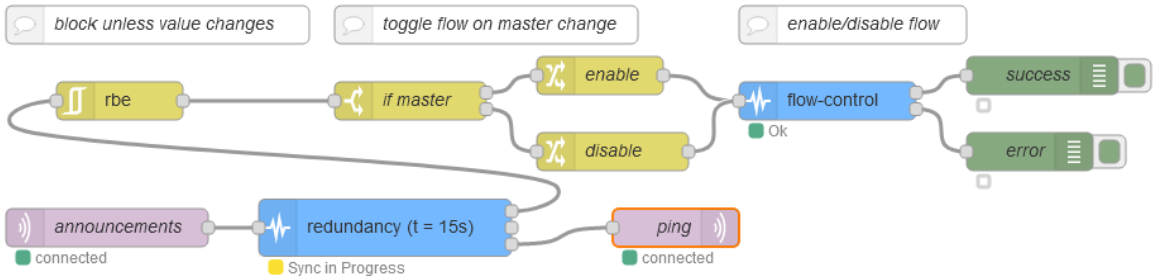


Figure 12.10: A *flow* depicting the management of two Node-RED instances (`redundancy`), and adapting the behavior of the system (`flow-control`) when the master instance changes[4]. The `redundancy` node is configured with a TIMEOUT that triggers when a redundant instance stops *pinging* for 15 seconds, triggering a new election.

In this experiment, a new election occurs every 15 seconds (configuration of the `redundancy` node). The default master instance (the one with the highest octet) was turned off multiple times (randomly). The time was measured between the disconnection of the master instance until that the sensor reading *flow* resumes on the fallback instance (a *ping* is done to an external service). A total of 10 measurements were made, and the MTTR of the system was $13.7s$ ($\sigma = 1.77s$).

As it can be seen in the Figure 12.11 (p. 229), almost all sensor readings reach the external endpoint, even with the continuous toggling of the master Node-RED instance. During a test

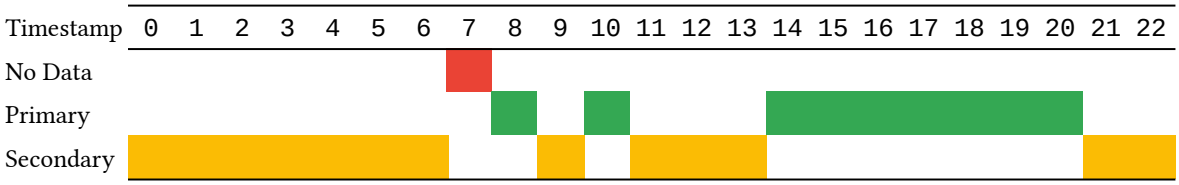


Figure 12.11: Sensor readings timeline of a 22-minute window, marking if the measurement was provided by the primary instance (master) or by the secondary instance (fallback). It is also noted when there is no data arriving in the expected time slot.

of 22 minutes (which should result in 22 sensor readings), only one reading was lost.

12.2.2 Fault-Injection Experiments

To ensure that the self-healing — or any other kind of fault-tolerance — mechanisms work as intended, they must be exercised. In the research field of fault-tolerance, fault-injection has been used as a technique to intentionally cause errors and failures in systems by introducing faults and then observing how the system behaves and recovers from them.

To carry fault-injection in IoT systems and see how the self-healing mechanism behave — *i.e.*, chaos engineering — an open-source and widely used MQTT Broker[7] was modified so that we could easily inject faults into IoT systems. The modifications enabled us to use the broker as a proxy to intercept and modify messages before being published to a specific topic and according to user-defined fault-injection rules.

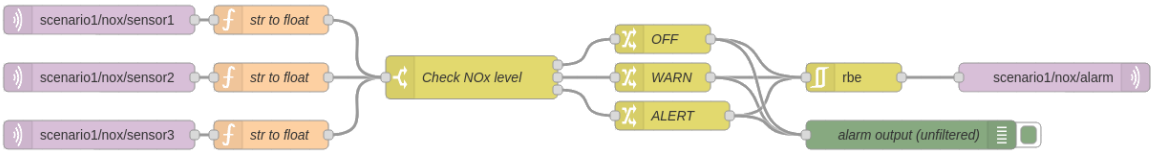


Figure 12.12: Baseline system Node-RED *flow* (**BL**). This system parses the reading received from the sensors and sends the respective alarm level as output, filtered by a report-by-exception node to ensure that values are only emitted when the current alarm level changes. The topics in this figure are set to those of Scenario 1 but are altered depending on the Scenario being tested.

For validation purposes, we defined four combinations of the System Under Testing representing all possible combinations of the system with and without self-healing or fault-injection. We called these **BL** (baseline), self-healing (**SH**), fault-injection (**FI**), and self-healing with fault-injection (**FI**×**SH**).

If the fault-injection and self-healing mechanisms are working correctly we expect that (1) the behavior of **SH** is not very different from **BL**, since no fault-injection is performed in either system and self-healing mechanisms should have low impact in a nominal system, (2) the behavior of **FI** is very different from **BL**, since the base system, without self-healing components should not be able to recover from injected faults, providing the performed fault-injection is

[7] AEDES MQTT broker, https://github.com/moscajs/aedes.

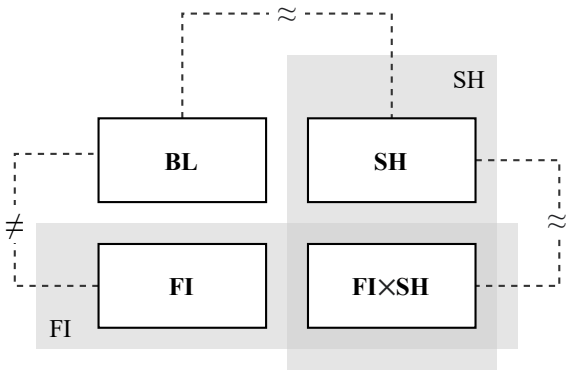


Figure 12.13: Expected similarity degree between the different combinations of the SUT.

enough to deviate from nominal operation, and (3) the behavior of **SH** is similar to that of **FI**×**SH**, showing that the self-healing mechanisms are able to bring a system with injected faults back into nominal behavior. These relations are depicted in Figure 12.13 (p. 230).

The experiments were done on a standard Linux laptop with Node-RED version 1.3.2, and the modified AEDES MQTT broker was run on NodeJS version 14.15.5. A replication package for these experiments is available (*cf.* Appendix B, p. 337).

Sensor Readings Issues (S1)

In this scenario (**S1**), we performed four experiments, and in each one, different types of fault-injection operators were applied to the sensor messages passing through the MQTT broker (*i.e.*, simulating sensor malfunctions).

A Node-RED *flow*, with self-healing mechanisms, was developed to deal with these issues (**SH**) as depicted in Figure 12.14 (p. 231). The `join` and `compensate` *nodes* are configured with a timeout of 6 seconds to have a margin of 1 second in relation to the readings' periodicity (5 seconds).

Experiment S1E1 In this experiment no fault injection was performed. Only the baseline system (**BL**) and the system with added self-healing mechanisms (**SH**) were considered. This allows us to compare the behavior of **BL** with **SH** in normal operation, and creates a base of comparison for the remaining experiments (*i.e.*, experiments with fault-injection). This also provides us a behavioral profile of **BL** when compared with **SH**, giving us insights on how self-healing mechanisms' operate with no added entropy.

We expect the SUT to remain stable during this experiment, outputting the expected alarm levels for the sensor readings' thresholds. Despite the expected similarity in behavior, it is expected that **SH**'s alarm level output will be more stable than that of **BL**. This is due to the latter not implementing any type of consensus or majority voting and instead simply using the received values directly as a stream.

Figure 12.15 (p. 232) shows the experiment results for **BL** and **SH**. Despite the alarm output (represented as shaded areas near the horizontal axis) being very similar for both experiment outputs, stability is higher for **SH**. This can be observed by the lack of fast alarm state changes for borderline values for **SH**, which occur several times for **BL** (*e.g.*, around 35 to 40 seconds

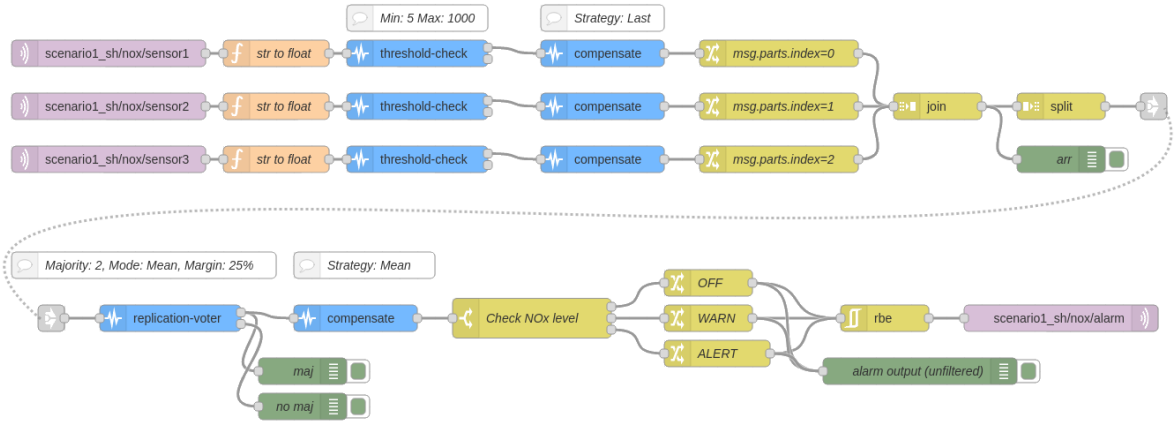


Figure 12.14: Node-RED *flow* with self-healing mechanisms to deal with issues on sensor readings (**SH**). This system expands upon **BL** by introducing self-healing capabilities via SHEN nodes. It filters extraneous messages that are outside the expected operating range, compensates for missing values after a certain timeout, joins messages so that they are considered in groups of 3, considers majority of values with a minimum consensus of 2 (with a 25% difference margin), and compensates for readings for which there is no majority with a mean of the previous readings, besides the basic functionality already implemented by **BL**.

into the experiment or around 415 to 420 seconds). The cause of this is likely to be **BL**'s lack of consensus mechanism, given that this system instead simply considers the most recent reading in order to determine the alarm state. When the three sensors report in quick succession, if the values of their readings are near the alarm level thresholds, fluctuations in the alarm level are expected (which is what happens in the examples above).

$$\mathrm{S1E1_{BL}} \cap \mathrm{S1E1_{SH}} = 97.3\% \tag{12.1}$$

The output similarity is confirmed by the alarm level overlap percentage between these two outputs, which is 97.3% (*cf.* Equation 12.1). This is also a good sanity check to confirm that the addition of self-healing capabilities to the base system does not significantly change the alarm output status, which means that comparisons for **SH** between **S1E1**, and further experiments, will be meaningful in validating self-healing recovery of injected faults.

Table 12.2: Count of alarm level state transitions for S1E1 baseline (BL) and self-healing (SH).

	BL	**SH**
Off (0)	8	4
Warn (1)	20	13
Danger (2)	11	8
Total	39	25

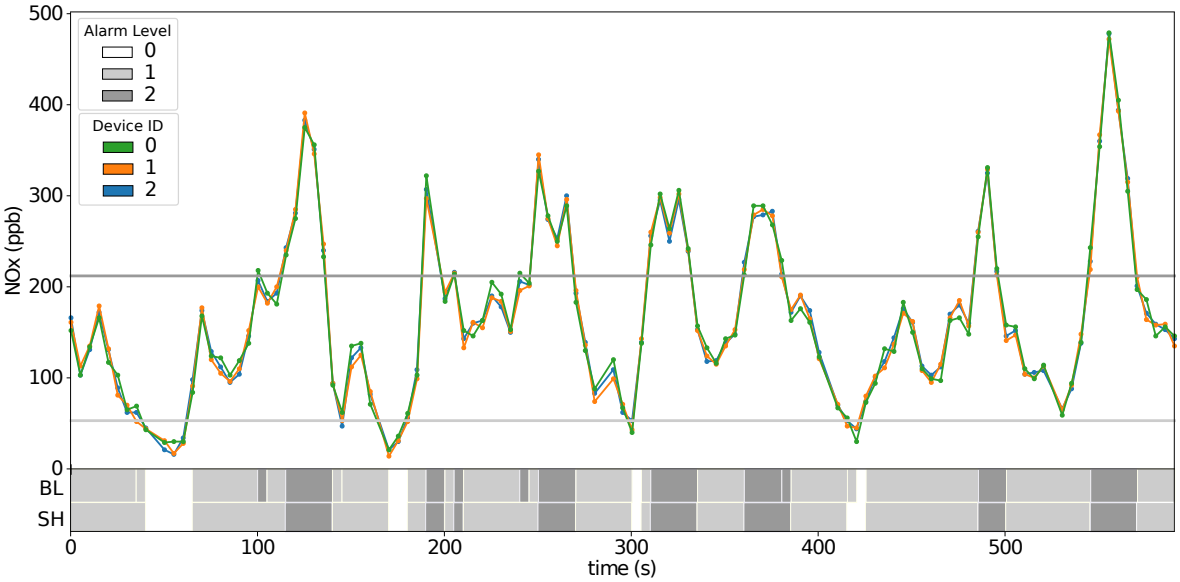


Figure 12.15: NO_x concentration and alarm status with and without self-healing for S1E1.

Table 12.2 (p. 231) supports the previous claim — that **SH** provides an improvement in system stability in comparison to **BL** — due to the lower number of alarm state transitions. Additionally, this experiment presents evidence that both **BL** and **SH** correctly implement the expected core functionality (triggering the different alarm levels for different sensor reading thresholds), given that the alarm level at a given point in time corresponds to the sensor readings' distribution along the thresholds (represented in the mentioned figures by the horizontal lines).

Experiment S1E2 Considering the baseline (**BL**) and the self-healing (**SH**) systems (which S1E1 shows to be similar in normal operation), we proceed to inject faults on both, obtaining systems **FI** (corresponding to the injection of faults in **BL**) and **FI**×**SH** (corresponding to the injection of faults in **SH**).

The fault being injected corresponds to an erroneous `Device ID 0` sensor's reading. As a result, this sensor's readings are altered to be stuck at the upper operating bound (1000 *ppb*). This experiment simulates a fault in which a sensor malfunctions by continuously emitting readings in its top operating bound.

We expect that this fault-injection will provoke an inconsistent output in **FI**, especially if the third sensor is frequently the last to emit its reading, even if only by a slight delay. Due to relying on a majority of at least two values to decide on the alarm level to emit, we expect that the self-healing mechanism will be able to deal with the faults injected in **FI**×**SH**.

Figure 12.16 (p. 233) shows the experiment results for **FI** and **FI**×**SH**. As expected, the faults injected (**FI**) disrupt the normal function of the system, resulting in constant alternation between alarm states, spending most of the experiment's time in the highest alarm level. Meanwhile, **FI**×**SH** successfully recovers from the injected faults, having a near-perfect performance in comparison to this system's output for **S1E1**.

These statements are supported by results presented in Table 12.3 (p. 233), which shows that the usage of self-healing results similar behavior when faults are injected (**FI**×**SH**) or not

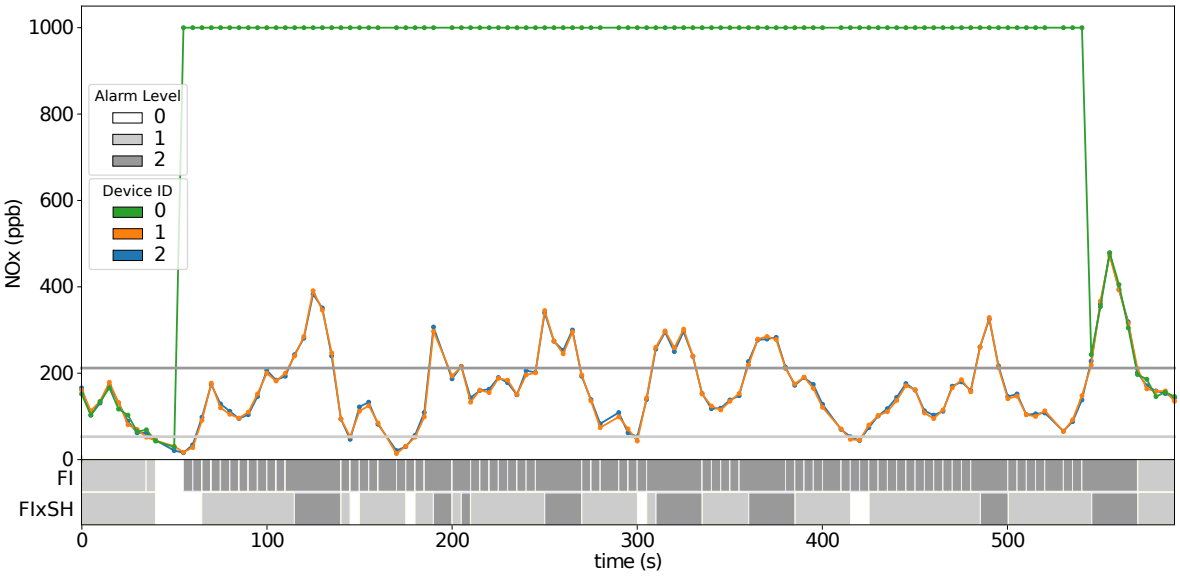


Figure 12.16: NO_x concentration and alarm status for S1E2.

Table 12.3: S1E2 overlap (%) with the base experiment S1E1.

	BL	**SH**
FI	40.0%	
FI×SH		98.1%

(**SH**), with an overlap of 98.1%. Furthermore, **FI** has a significantly lower overlap percentage of 40.0%. Table 12.4 (p. 233) also illustrate these conclusions, in which **FI** is much more unstable in comparison to **BL**, being the total number of state transitions for this experiment 148, while there were only 39 state transitions for the base experiment. Furthermore, the number of state transitions with self-healing has increased only marginally, going from 25 in the base experiment (**SH**) to 27 in this experiment (**FI×SH**).

Therefore, **S1E2** demonstrates that the original system cannot handle sensor *stuck-at* issues since the behavior of **FI** is considerably affected, which validates that the performed fault-injection was meaningful enough to disturb the system's regular operation. On the other hand, **FI×SH** can recover from the injected faults, having a remarkably similar behavior to **SH**.

Table 12.4: Alarm level state transitions for S1E2.

	FI	**FI×SH**
Off (0)	10	5
Warn (1)	68	14
Danger (2)	70	8
Total	148	27

Table 12.5: S1E3 overlap (%) with the base experiment S1E1.

	BL	**SH**
FI	76.3%	
FI×SH		97.4%

Experiment S1E3 In this experiment, we injected faults in **BL** and **SH** to obtain **FI** and **FI×SH** by multiplying 40% of the readings done by `Device ID 0` sensor by a random factor in the range $[0.2, 2.2]$, simulating *spikes* in sensor readings. The factor is randomized for each *spike* occurrence[8].

We expect that **FI** may output incorrect alarm values (in comparison to **BL**), especially when the altered values switch between alarm level thresholds. On the other hand, **FI×SH** should be able to handle the *spikes* since that, even if one of the three sensors outputs a value considerably different from the others, it will be discarded and the other two sensors' values will be considered instead — due to the usage of the `replication-voter` node.

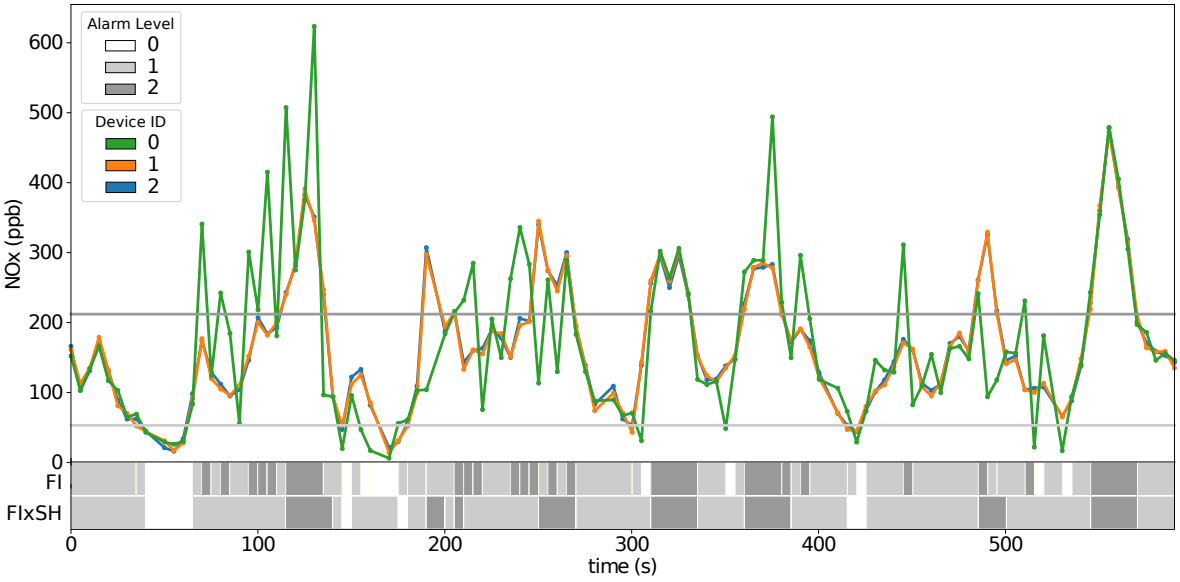


Figure 12.17: NO_x concentration and alarm status for S1E3.

Figure 12.17 (p. 234) shows the experiment results for **FI** and **FI×SH**. **FI** has had a good performance in the presence of the *spikes* (both when increasing and decreasing the read value), but there were still several situations in which the sensor reading *spike* caused the output alarm level to differ from the expected value in **BL**. **FI×SH** has held up to our expectations, handling almost all the injected faults and operating similarly to **SH**.

These statements are supported by Table 12.5 (p. 234), which shows that the self-healing mechanisms make the system perform similarly when fault-injection is (**FI×SH**) and is not applied (**SH**), with a near-perfect overlap of 97.4%, while **FI** has a lower overlap percentage of 76.3% with **BL**, showcasing the disruption provoked by the injected *spikes*.

[8]This fault is common for sensing devices when they are running out of battery [Ni+09].

Table 12.6: Alarm level state transitions for S1E3.

	FI	**FI×SH**
Off (0)	8	4
Warn (1)	26	13
Danger (2)	17	8
Total	51	25

Table 12.7: S1E4 overlap (%) with the base experiment S1E1.

	BL	**SH**
FI	99.8%	
FI×SH		98.7%

Table 12.6 (p. 235) supports the role of the self-healing mechanisms. Despite the difference not being as remarkable as that of the overlap percentages, it is of note to mention that **FI×SH** has the exact same number of alarm level state transitions of **SH** while the number of state transitions has increased for **FI** when compared with **BL**.

Despite this experiment not causing a variation as significant as that of the behavior of **FI** in **S1E2**, we were still able to observe a mismatch between the behavior of this system between the base case and this experiment, even if to a lesser extent. This shows that for the system under study, the *spikes* faults are less concerning when compared with the *stuck-at* ones injected in **S1E2**, *i.e.*, *spikes* lead to lower deviations in operation when compared to *stuck-at*. Nevertheless, due to the decline in the overlap percentage for **FI** in comparison with **BL**, we can conclude that the faults injected were significant enough to affect the system's correct functioning.

Since **FI×SH** behaved similarly to **SH**, we are able to confirm that for this experiment the presence of self-healing capabilities are beneficial for the system's correct operation, thus improving its resilience.

Experiment S1E4 In this experiment, we injected faults in `Device ID 0` sensor so that it has a 20% chance of losing messages[9]. The system does not receive any of the lost messages, as these are suppressed before leaving the middleware broker.

We expect that **FI** may report erroneous alarm values (compared to the base experiment, **S1E1**), especially when the missing values are in proximity to the alarm thresholds. **FI×SH** should be able to handle the injected faults by compensating the missing values by replaying the last message in the expected time interval.

Figure 12.18 (p. 236) shows the experiment results for **FI** and **FI×SH**. **FI** is capable of handling the loss of some readings, thus the alarm output is quite similar to **BL**. **FI×SH** is also able to handle the loss of readings, similarly having almost the same behavior as **SH**.

[9]This fault may occur when a sensor is disconnected, has an intermittent power supply, or the network is unstable [Ni+09].

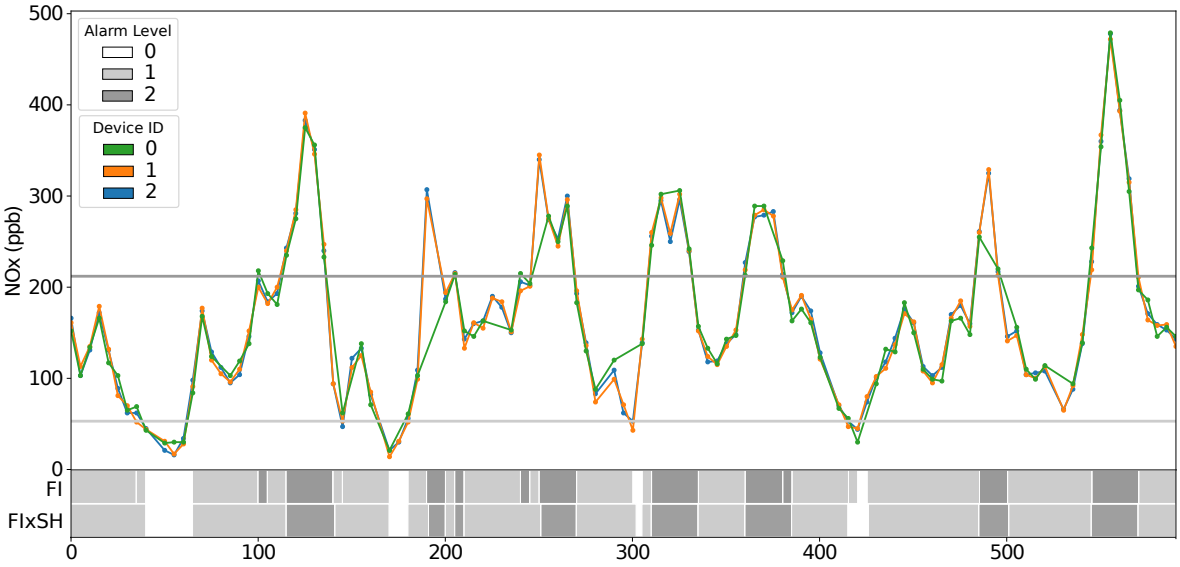


Figure 12.18: NO_x concentration and alarm status for S1E4.

Table 12.8: Alarm level state transitions for S1E4.

	FI	**FI×SH**
Off (0)	8	4
Warn (1)	20	13
Danger (2)	11	8
Total	39	25

The similarity in outputs when compared with **S1E1** is a direct result of the low probability of losing one reading. Also, since the values for different sensors in the original dataset are close to one another, even by increasing the probability of values being suppressed from one sensor would result in **FI** outputting the expected alarm values for most of the experiment's duration.

These statements are supported by Table 12.7 (p. 235), which shows that **FI** has a similar behavior to **BL**, and that **FI**×**SH** has a similar behaviour with **SH**.

The previous observations are further supported by the results presented in Table 12.8 (p. 236), which shows that the number of state transitions for both systems are identical to those that occurred in **S1E1**.

$$\mathrm{S1E1_{BL}} \cap \mathrm{S1E4_{SH}} = 98.4\% \tag{12.2}$$

This experiment did not cause a significant enough deviation from the base experiment's behavior for **FI**, which is corroborated by the high overlap percentage with **BL** (*cf.* Equation 12.2), as well as the fact that the number and type of alarm state transitions are identical to those of **S1E1** (*cf.* Table 12.8, p. 236). Thus, we conclude that the used dataset may not be the best candidate for this type of fault injection. An higher deviation in operation could possibly be observed if the fault-injection was done to more than one sensor at a time and with higher probability of losing a message.

Timing Issues (S2)

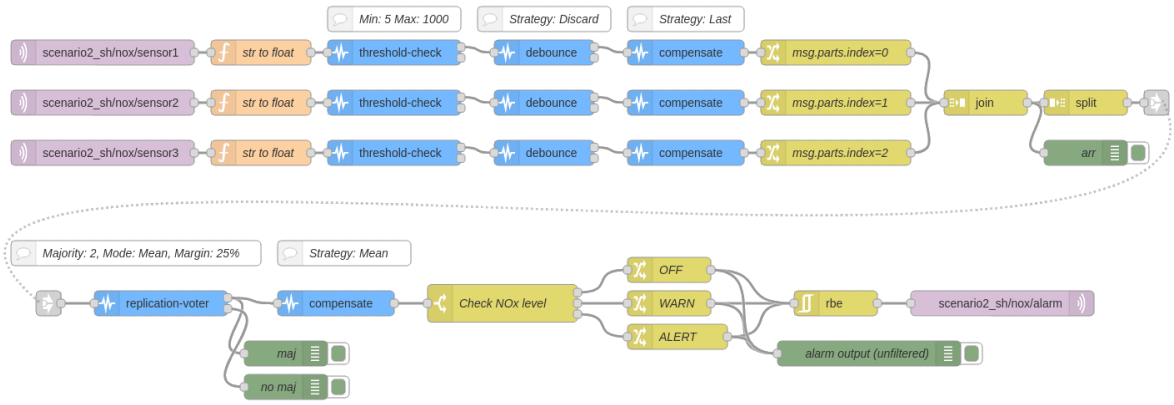


Figure 12.19: Node-RED *flow* with timing-related self-healing *nodes* (**SH**). This system expands upon **S1** by introducing `debounce` nodes which can filter out extraneous messages based on the expected timing of the system's regular messages, besides the functionality already implemented in **SH** of **S1**.

In this scenario (**S2**), the Node-RED *flow* self-healing mechanisms used in **S1** was enriched with *nodes* that detect and mitigate issues with timings (*e.g.*, readings frequency issues), as depicted in Figure 12.19 (p. 237). The `join` and `compensate` *nodes* are configured with a timeout of 6 seconds to have a margin of 1 second in relation to the readings' periodicity (5 seconds). A total of two experiments were conducted for **S2**.

Experiment S2E1 No faults are injected for this experiment. Similarly to **S1E1**, the purpose of this experiment is to confirm that the system's base functionality is correctly implemented for both **BL** and **SH**, as well as to provide a base experimental output with which to compare the behavior of the systems in following experiments.

We expect that the systems under observation remain stable during this experiment since there are no injected faults, correctly outputting the respective alarm levels for the sensor readings' thresholds. Beyond the similar alarm levels over time, it is also expected that **SH**'s alarm level output will be more stable than that of **BL**. This is due to the latter not implementing any type of consensus or majority voting and instead simply using the received values directly as a stream.

Figure 12.20 (p. 238) shows the experiment results for **BL** and **SH**. Beyond the expected similarity in both systems alert levels over time, a slightly higher stability in **SH** is observable. This is a direct result of the self-healing mechanisms that filter out alarm state changes for borderline values, which occur several times for **BL** (*e.g.*, around 35 to 40 seconds into the experiment or around 415 to 420 seconds). The cause of this is likely to be **BL**'s lack of consensus mechanism, given that the **BL** *flow* considers only the most recent reading to set the alarm state. When the three sensors report in quick succession — if the values of their readings are near the alarm level thresholds — this fluctuation is expected (which is what occurred in the examples above).

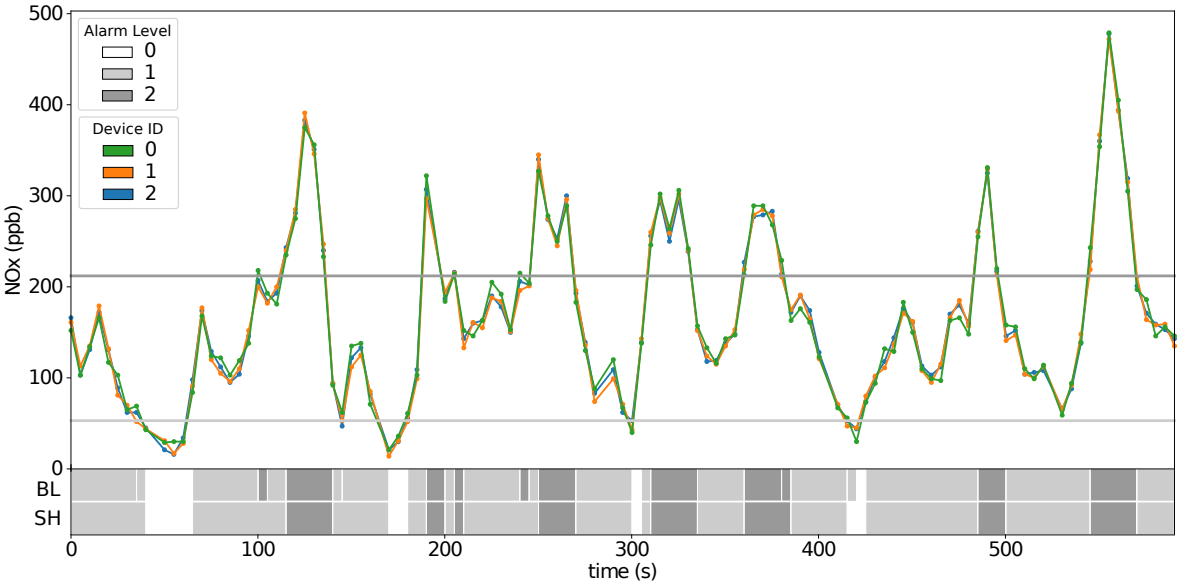


Figure 12.20: NO_x concentration and alarm status for S2E1.

$$\text{S2E1}_{\text{BL}} \cap \text{S2E1}_{\text{SH}} = 97.4\% \tag{12.3}$$

The output similarity is confirmed by the calculated alarm level overlap percentage between these two outputs, which is 97.4% (*cf.* Equation 12.3). This is also a good sanity check to confirm that the addition of self-healing capabilities to the base system does not significantly change the alarm status output, which means that comparisons for **SH**, between **S2E1** and further experiments, will be meaningful in asserting the functioning of self-healing mechanisms.

Table 12.9: Alarm level state transitions for S2E1.

	BL	**SH**
Off (0)	8	4
Warn (1)	20	13
Danger (2)	11	8
Total	39	25

Table 12.9 (p. 238) supports the previous claim that **SH** provides an improvement in system stability in comparison to **BL**, as shown by the lower number of alarm state transitions.

S2E1 also provides evidence that both **BL** and **SH** correctly implement the expected core functionality, *i.e.*, triggering the different alarm levels for different sensor reading thresholds, given that the alarm level at a given point in time corresponds to the sensor readings' distribution along the alarm thresholds (represented by the horizontal lines in Figure 12.20, p. 238).

Experiment S2E2 In this experiment, to introduce additional noise into the system, each message for `Device ID 0` is repeated after 6 seconds, as depicted in Figure 12.21 (p. 239). Since

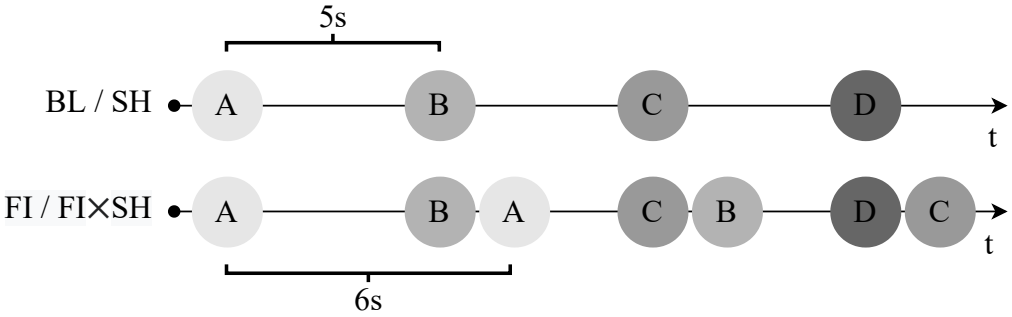


Figure 12.21: Marble diagram of messages for S2E2. The top diagram depicts the regular flow of messages, while the bottom diagram shows the messages after the fault injection done for S2E2.

the periodicity of the system's readings in regular circumstances is of 5 seconds, the repeated message will be outputted in proximity to the next reading.

We expect that **FI** will have an output that is less stable than it was for **BL** due to the injected faults. We expect this to be problematic for **FI** since it does not have any concept of message timing. On the other hand, **FI**×**SH** should be able to cope with the injected faults since the debounce *node* will filter out the additional messages that come out of the expected frequency, thus behaving similarly to **SH**.

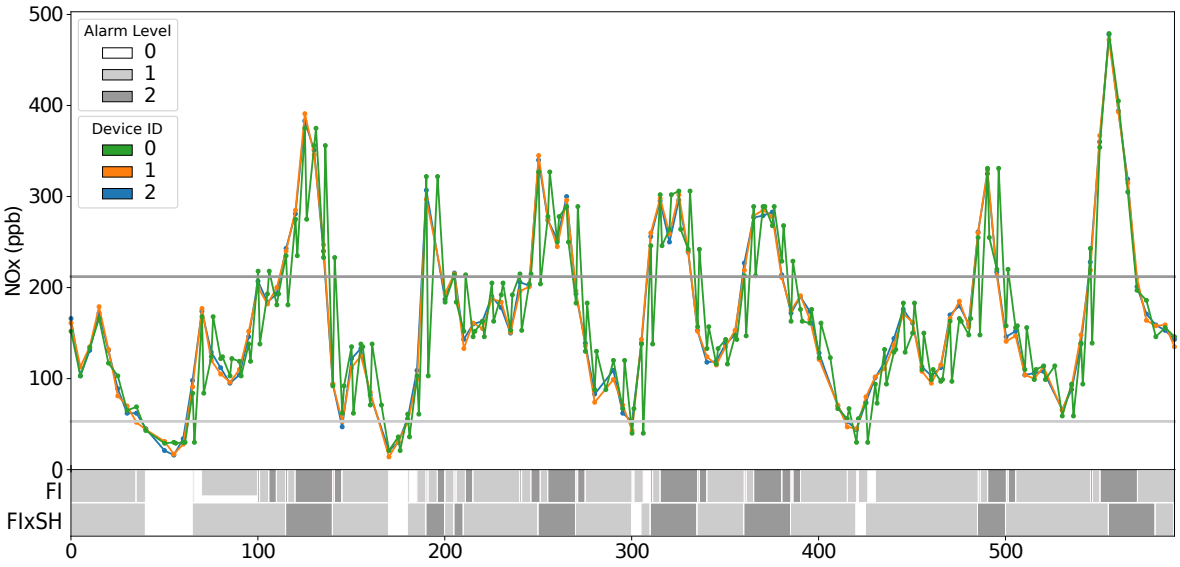


Figure 12.22: NO_x concentration and alarm status with self-healing for S2E2.

FI (*cf.* Figure 12.22, p. 239) performed significantly better than expected, despite the issues with messages near the alarm level thresholds, *i.e.*, repeating the previous message would cause the system to output the previous alarm level once again until it received the following message and went back to the expected state. This may be caused by the fact that the periodicity of the sensor readings messages is quite high and thus whenever a fault is injected, it is not *in effect* for a long duration.

Table 12.10: S2E2 overlap (%) with the base experiment S2E1.

	BL	**SH**
FI	83.4%	
FI×SH		95.7%

Table 12.11: Alarm level state transitions for S2E2.

	FI	**FI×SH**
Off (0)	12	4
Warn (1)	36	13
Danger (2)	25	8
Total	73	25

As expected, **FI×SH** was able to cope with the injected faults (*cf.* Figure 12.22, p. 239), having a near-perfect behavior in comparison to **SH**.

Table 12.10 (p. 240) shows that **SH** has a similar behavior to **FI×SH**, with an overlap of 95.7%, and that **BL** has a slightly lower overlap percentage of 83.4% to **FI**.

Despite the high overlap percentages for **FI** with **BL** (*cf.* Table 12.10, p. 240), Table 12.11 (p. 240) shows that for **S2E2**, **FI** has output nearly two times the amount of alarm level state transitions in comparison to **S2E1**.

S2E2 shows that despite the introduction of faults in **FI** the difference shown by the overlap percentage to **BL** is minimal. Despite this, **FI×SH** has better ability to cope with the injected faults, operating closer to **SH**.

FI also performs worse that **FI×SH** when taking into account the number of alarm level state transitions (*cf.* Table 12.11, p. 240). And, while it is not possible to conclude if this experiment caused enough deviation from **BL** to **FI** by taking only in consideration the overlap percentage, the difference in the count of alarm level state transition for **FI** in comparison to **BL** provides evidence that the injected faults has caused issues on the baseline system which the self-healing system is able to address.

12.3 Discussion

We showed improvements to *SmartLab* reliability and dependability both by detecting failures as they are happening and recovering or maintaining the systems' health. Node-RED does not provide any *out-of-the-box* solution for dealing with failing components, nor to dynamically change the system's behavior during runtime, which is essential to enable *self-healing*. After adding such functionalities via new nodes, users can now leverage these new capabilities. Our first example scenario shows how it becomes possible to test and recover from a SPOF (exemplified as a message broker failure). The same method could be used to deal with other SPOFs, including failures of Node-RED itself, with a `RedundancyManager` node that activates

duplicated and *inactive flows* on a different Node-RED instance (provided one is available). The second scenario shows how to isolate a system's component to ensure that its misbehaviors do not compromise the system as a whole. The last scenario shows how we can now manage several (redundant) communication protocols as an enabler of *self-healing* mechanisms and the importance of continuously asserting the actuators' outcome.

The fault-injection experiments **S1E1**, **S1E2**, **S1E3**, **S2E1**, and **S2E2** provide empirical evidence that: (1) the self-healing systems (**SH**) do not deviate too much in behavior from the baseline system (**BL**); (2) the faults injected are consequential since there is a deviation on the baseline system in comparison to the base experiment when no fault is being injected — even if the deviation is not significative per se; and (3) when the faults injected are consequential, the self-healing systems were able to recover from it, conforming with the normal service, and thus confirming that the self-healing mechanisms were being exercised and performing as expected.

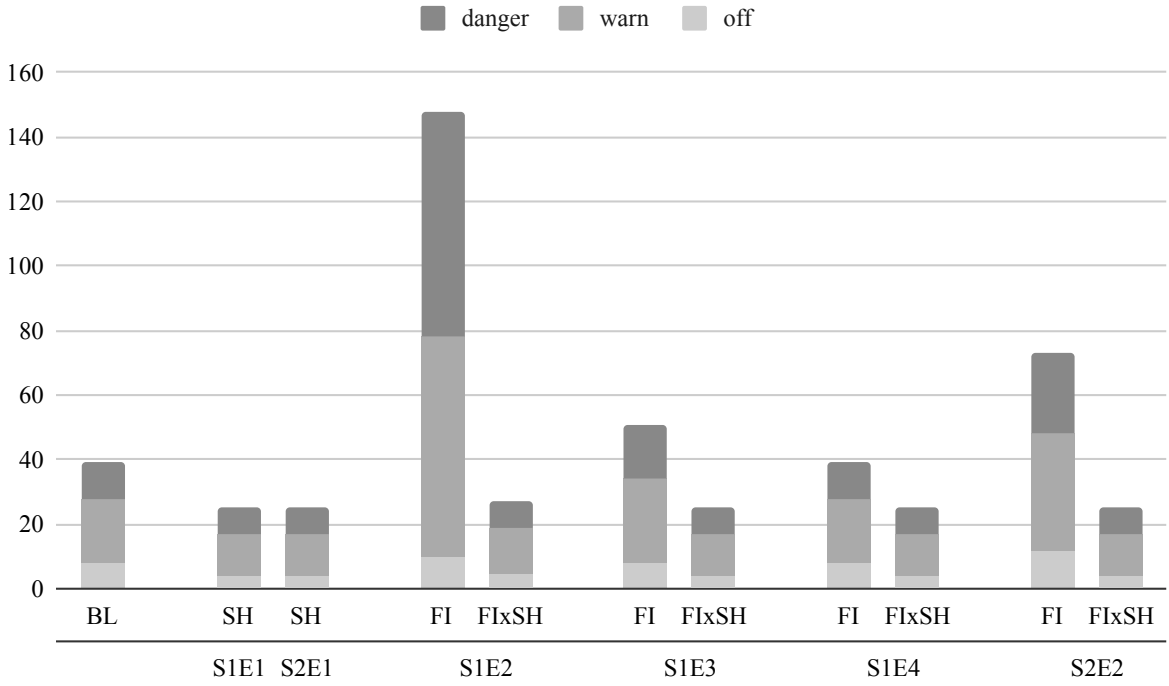


Figure 12.23: Count of alarm state transitions per experiment (the closer **FI** and **FI**×**SH** are, respectively, to **BL** and **SH**, the better).

More concretely, by analyzing the chart on Figure 12.23 (p. 241) we can see the impact that fault-injection has in a system without any fault-tolerance mechanisms versus a system with self-healing capabilities. The number of times that the alarm changes state between its three alert levels is considerably higher in all experiments, with a clear impact in the experiments **S1E2**, **S1E3**, and **S2E2**, where the number of transitions was more than two times higher than the expected number of transitions.

While the overlap percentages (*cf.* Figure 12.24, p. 242) of the different experiments do not lead to a so direct conclusion for most experiments, we can see that for **S1E2** performs considerably better when self-healing mechanisms are present.

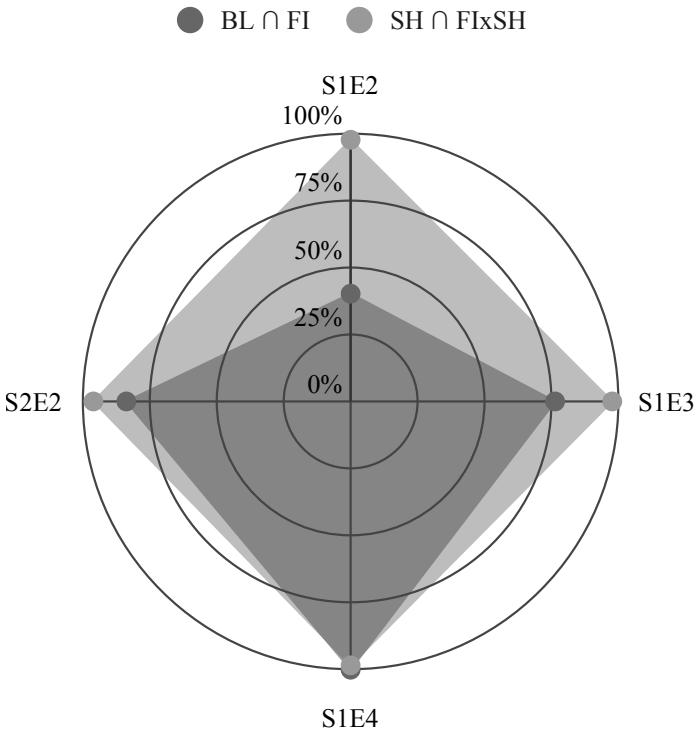


Figure 12.24: Comparison of the overlap of the different experiments with and without self-healing mechanisms to the corresponding baseline (higher percentage is better).

Additionally, in **S1E4** establishes that it is paramount for the behavior of **BL** to be *noticeably* different when faults are being injected (**FI**) in comparison to the regular operating circumstances. This factor made it so that we considered this experiment inconclusive, due to the low entropy caused in the system. Nevertheless, it showed that it is necessary to find this stark difference in expected versus observed output for the baseline system to be sure if the self-healing components are doing any work at all, since a naïve system would already be able to *recover* from most of the injected faults.

It is also noticeable that for the created scenarios, due to the used dataset, some types of fault-injection did not result in much instability. This is due to the fact that all three sensors output readings with values in close to each other. As such, if one or even two sensors fail, it is likely that a naïve system (*e.g.*, **BL**) will still perform as expected, outputting the correct alarm levels for most cases even in the presence of faults (**FI**). This is an indicator that further validation should be performed with other types of datasets and systems, as well as different types of faults.

Ensuring the dependability of software systems has been the goal of most fault-tolerance research in the past years [ALR01]. In IoT, ensuring systems are secure, reliable, and compliant is becoming a paramount concern due to the recent increase in safety-critical applications. Fault-tolerance becomes more challenging due to several factors, including, but not limited to: (1) the high heterogeneity of devices, (2) the interaction and limitations of systems deployed in a physical world, (3) the fragmentation of the field, ranging from the unusually high number of communication protocols to the different and competing standards, and (4) the intrinsic

dependability on hardware that might simply fail [Aly+19]. Moreover, in a perfect environment, every actuator should possess a monitoring sensor capable of verifying its intended end state; however, real-world cost efficiency might limit its availability to critical components.

The pervasiveness and complexity of IoT have contributed to the rise of visual programming, in particular Node-RED, as the go-to solution. Nevertheless, as it slowly permeates our lives, it becomes crucial to ensure proper functioning through self-verification and self-recovery features: *self-healing*. Although previous work attempted to tackle runtime verification and self-healing mechanisms to specific IoT systems, none was found to provide these features in a visual environment. Previous work also relies heavily on new systems (*e.g.*, rule-based monitoring services and CEP approaches) without attempting to integrate them into the existing ecosystem of tools and platforms.

Despite the current Node-RED limitations (*cf.* Chapter 5, p. 132), it was possible (up to a certain extent) to fulfill our goals mostly by using its visual notation, as seen in § 12.2 (p. 221). It should be noted that all implemented strategies fall into the *forward error recovery* category, *i.e.*, "continue from an erroneous state by making selective corrections to the system state" [JZ05]. Exploration of *backward error recovery* techniques is harder due to the dependency of system state *checkpoints*, that needs to capture a mix of device internal states, concurrent communication protocols messages, and controller state.

To further improve the self-healing capabilities of systems such as the presented *SmartLab*, devices should have extra features such as diverse communication channels (*e.g.*, Wi-Fi and ZigBee), remote management capabilities (*e.g.*, independent watchdogs that allow to gracefully restore a device), and capability announcement, which would empower dynamic usage of redundant devices. We observe these features are mostly absent from consumer-grade devices, most probably due to cost efficiency.

12.4 Summary

IoT systems are perhaps one of the most significant examples of heterogeneous architectures in existence. Different protocols, different application stacks, different integration services, and different orchestration engines all must come together in a technological solution that allows both organic growth from end-users, and dealing with security and privacy concerns at unprecedented levels. The consequence is that the system is required to keep functioning at minimal levels, even when parts of it become non-compliant, faulty, or even under attack. Requiring the end-user to address these challenges is unrealistic, as most of them are not developers. Even most system integrators cannot keep up with the pace of release devices, which seldom adhere to open standards.

We argue that an IoT system that attempts to tackle the presented challenges must be capable of *self-healing*. This is not a small feat, as most of the research being conducted in integration tools for IoT recurrently disregard failure detection and recovery. We fulfill these desiderata with SHEN, Self-Healing Extensions for Node-RED. As this popular tool lacks built-in testing and self-healing capabilities, we use it as a case-study for common failure and recovery scenarios, and (1) suggest how to leverage *meta-programming* techniques to allow self-modification of *flows* via a custom *plugin*, (2) explore common self-healing patterns and how they can be solved

by such techniques, (3) provide them as reusable *nodes* for others to incorporate in their systems, and (4) discuss which challenges remain open and which might need rethinking architectural and design decisions.

To validate our claims, we added SHEN to the existing *SmartLab*, and proceed to illustrate its behavior in different operational scenarios. We conclude that we can improve the system's reliability and dependability, both by being able to detect failing conditions, and reacting to them by self-modification of defined *flows*. Additionally, we also proceed to carry experiments with fault-injection to verify that the self-healing mechanism perform as expect when faults occur.

Fault-injection becomes paramount to ensure that the fault-tolerance mechanisms employed in a system perform as they are expected in the presence of faults. By instrumenting an MQTT broker, we enable the injection of faults at the middleware level that allows us to observe how well the subscribers deal with such faults. In this work, fault-injection allowed use to check if the self-healing mechanisms configured in the Node-RED system are sufficient to deal with pre-defined, and expected, faults. The carried experiments showcase that the self-healing extensions do, indeed, work as expected, with the injected faults causing little to none impact on the delivery of normal service.

Part IV

End-User Development

13 | Real-time Feedback in Node-RED

As the system complexity evolves, understanding what is happening (system's behavior) becomes harder. This affects most of the development tools, including visual-based one's (*cf.* Section 3.2, p. 68), reducing the ability of end-users to understand their system, thus making it challenging to create and modify existing rules while ensuring that changes do not break the expected behavior. While in this chapter, as well as in other parts of this thesis, the solution under analysis is Node-RED (*cf.* Chapter 5, p. 132), these issues are shared across the visual programming for IoT landscape.

To mitigate some of the issues that harden the development process in Node-RED, in this chapter, we propose a set of enhancements to mitigate them (or, at least, reduce their impact). We proceed to design and implement a POC (named NODERED-CAULDRON), built on top of the original Node-RED, which fulfills these enhancements. NODERED-CAULDRON allows users to check the runtime state of the system (*i.e.*, observing input/output of a given node), use debugging mechanisms like breakpoints (*i.e.*, "pause" the incoming messages of a given node, and understand each message that flows through it), and perform runtime modifications (*i.e.*, inject and change messages). To empirically assert how and how much these enhancements impact users' performance when building, evolving, and maintaining IoT systems, an experimental phase followed where 20 participants had to carry out a set of tasks in the two different Node-RED versions (original and enhanced with our POC). The overall results reveal that the added enhancements improve users' ability to develop IoT systems and ease the process of understanding how the system is behaving.

> Parts of this chapter were published in the work REAL-TIME FEEDBACK IN NODE-RED FOR IOT DEVELOPMENT: AN EMPIRICAL STUDY [Tor+20], and were partially based on the master thesis work of Diogo Torres entitled INCREASING THE FEEDBACK ON IOT DEVELOPMENT IN NODE-RED [Tor20]. The author's main contributions were on the development of the software, data curation, visualization, and writing of the published versions of the work.

13.1 Approach Overview

We conceive that there is a set of key features that when added to visual programming environments improve the development of IoT systems by reducing the development time, the

number of bugs created during development, and overall system maintainability. Consider the *flow* in Figure 13.1 (p. 247) as a motivational scenario of the current visual notations used in Node-RED.

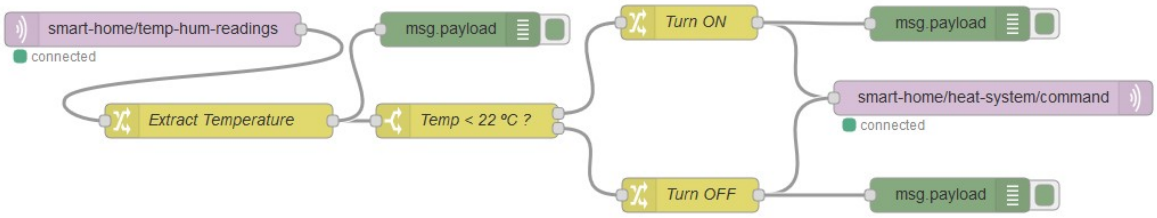


Figure 13.1: Whenever the temperature falls below 22 °C, the heating system must turn on until the temperature reaches that value.

We modified Node-RED to augment the system's observability and improve the feedback-loop between the development environment and its runtime, trying to improve the users' ability to build, evolve, and maintain IoT systems.

NODERED-CAULDRON focuses on addressing some identified missing features of Node-RED: (1) **Observability**, by providing the ability to present the information which *flows* through the nodes using different visual metaphors, (2) **Runtime Modification**, by allowing the injection of messages during runtime, and (3) **Exploration**, by enhancing the debug capabilities through breakpoints on each node without the need for re-deployments. With our approach, each node presents each input's messages; in nodes without any input, the output is presented. Thus, all the information flowing through all nodes is observable without the need to add new ones (*cf.* Figure 13.2, p. 247). Using a `Switch` node as example, we can observe the added features in detail (*cf.* Figure 13.3, p. 248).

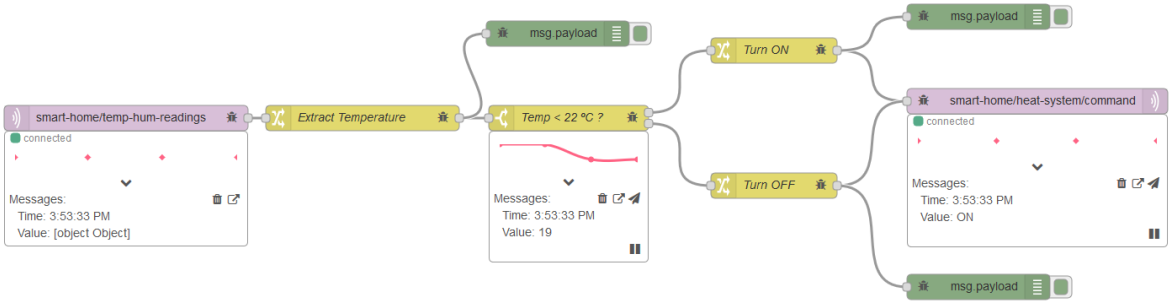


Figure 13.2: The *flow* from Figure 13.1 (p. 247) with the NODERED-CAULDRON's features, namely the message plots, and extra debug options.

Leveraging the already existing communication mechanism (between the runtime and UI), a new topic was added that allows *showing the runtime data* (*i.e.*, messages between nodes) in the UI. Using this additional communication channel, we can visualize incoming messages through two different plots (if the message's payload type is a number, it displays a line plot, otherwise, a scatter plot is displayed). These plots let the user perceive the values received or at what pace

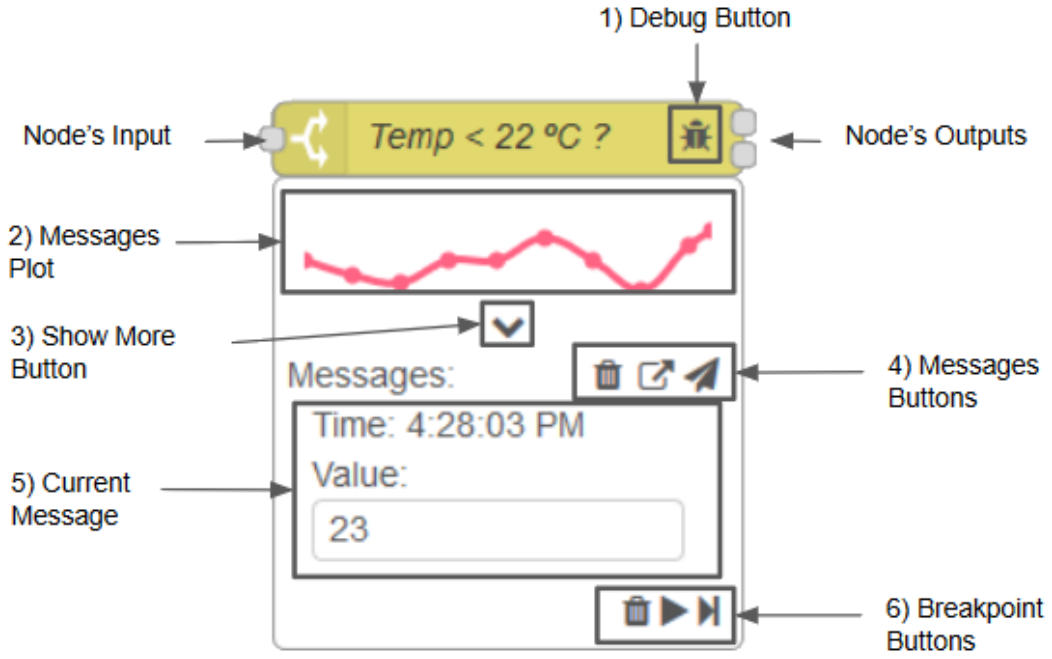


Figure 13.3: An example with a `Switch` node in NODERED-CAULDRON. On the top right, there is a *Debug Button* (1) that allows to expand/collapse the messages' plot (2) and the *Show More* button (3). This *Show More* button allows visualizing functionalities related to the messages and breakpoint system. For messages, it shows the current message to process (5) and buttons (4) to access input and output messages' history, clear this history, and injecting messages in the current node. For the breakpoint system (6), it allows pausing/starting message processing (queuing the incoming messages) and process each message at a time by using the step button. This step button also allows the modification of the current message. The trash button clears the breakpoint's queue.

they are coming. By allowing this communication to be bidirectional and applying the same strategy, we can *inject messages into the runtime* to test a specific system's behavior.

We also added extra debugging capabilities, such as breakpoints. This allows the user to *"pause"* incoming messages for a given node by queuing them. The user can also step forward one message at a time and change its payload. It is also possible to clear all the queued messages. When the node is *"unpaused"*, the queued messages are released in the "same" frequency that was received (*i.e.*, the time between the last two messages is calculated, and this value is used to set the pace). We further enhanced the `Debug` *node* with the same message's visualization capabilities of the other nodes.

13.2 Experiments and Results

Our goal is to verify if these changes impact the development process. We carried a controlled experiment to compare the performance and behavior of two developer groups [SOJ18;

KMB19]. We hypothesized that these characteristics would improve the ability of users to successfully build, evolve, and maintain IoT systems faster, easier, and with fewer errors. Specifically, we aim to answer the following study-specific research questions (SRQs):

SRQ1 *Would users with increased exposure to real-time information about the running system build and manage it faster?*

SRQ2 *Providing users with real-time feedback increases their ability to understand and change existing systems?*

SRQ3 *Is an IoT visual programming environment able to reduce human-induced errors during development by providing real-time feedback?*

13.2.1 Setup

A set of 3 experimental tasks was defined, namely: (a) debugging, (b) improving, and (c) creating an IoT system using Node-RED; hence, development experience and basic familiarity with IoT were required. A preliminary assessment of our procedure was made with two participants having distinct backgrounds: (1) a casual Node-RED user and (2) a user with no previous experience in Node-RED.

We made usage of a mix of quasi-experimental with ethnographic research. The population was split into two groups, *GA* and *GB*, with different treatments: *GA* used unmodified Node-RED, and *GB* used our tool. As there were no guarantees of equal technical expertise among groups, two Control Tasks (CT) were performed to provide basic familiarity with the tool. Following these control tasks, three Experimental Tasks (ET) were given to each group, viz. (a) debug, (b) improve, and (c) create a system from scratch. In these three tasks, *GB* was provided with additional documentation regarding the available new features. All tasks were solved in the same order, with a small-time break between them.

The study sample size was twenty participants, all of them final-year computer science students with at least basic IoT knowledge but with no Node-RED experience. To avoid participants' overload and at the same time providing a reasonable time to finish all the tasks, the duration of the experiment was set to 90 minutes, with a 25 minutes timeout per task.

All experiments were conducted in a remote environment[1]. The needed tools were hosted in a private virtual server. Video call software was used to communicate and provide access to the participant's screen. With this procedure, it was possible to observe and take notes on the participant's behavior, clarify some doubts related to the tasks, and verify if a certain outcome was correct. For both treatments we recorded: (a) the *time* taken to reach the solution; (b) the *number of deployments* made; and (c) the *number of verification requests* (*i.e.*, every time the user thought the task was finished). For *GB*, the number of clicks in each new functionality was also recorded.

A post-experimental survey was carried to assess the overall participant's experience and to collect improvement suggestions. For this, we resorted to five statements evaluated using a Likert-scale, three related to existing functionalities in NODERED-CAULDRON, and two regarding

[1] Due to the COVID-19 pandemic.

future improvements. We slightly adapted some questions to match the specificities of different treatments.

13.2.2 Tasks

To make it possible to run the experiments with equal operating conditions, a sensor/actuator simulator was developed (having a deterministic behavior) to provide real-time data (continuous flow of messages). This simulator implements mechanisms to validate the correctness of the experimental outcomes. The CTs were:

CT1 A preliminary task where Node-RED is introduced alongside the process of creating a simple *flow*. It shows how to manually inject messages in a *flow* (using the `Inject` node), parse them with custom JavaScript (using the `Function` node), and then display them in the sidebar (using the `Debug` node);

CT2 A task were data from seismometers must be used to activate an alarm, depending on the inferred earthquake's magnitude. This task introduced new nodes and logic (*e.g.*, read data from sensors, add intermediate logic, send commands to the actuators) to be used in later tasks.

The first two ETs were both based on a *smart farming* scenario where a system would automatically control a strawberry plantation inside a greenhouse. A third task focused on the development of a simple *smart home* system:

ET1 A debugging task with a set of rules. The system was capable of keeping the soil at a certain moist and temperature level. For this, the user was able to control (a) a heating system, (b) an irrigation mechanism, and (c) automatic windows. These were controlled by a humidity/temperature sensor. These rules had some bugs related to (a) erroneous conditions, (b) wrong commands sent to the actuators, and (c) mismatched field access;

ET2 An improvement task, where the user is responsible for adding a new feature to the current system, by using new devices (both sensors and actuators): (a) the status of the UV lamps should be adjusted according to weather forecasts, and (b) if the UV lamps are on, the window should be closed;

ET3 An implementation task, where the user must create a simple *smart home* system. Two different types of rules were given: (a) the lights should turn on when there is movement in the kitchen, and (b) every day at a given hour, the water heater and the coffee machine should be turned on (recurrent rule).

13.2.3 Results

We now provide an analysis of the results for both the Control and Experimental Tasks. We discarded CT1 from the analysis since it was mostly used as a sanity check to assert the correct functioning of the functionalities introduced.

Control Task

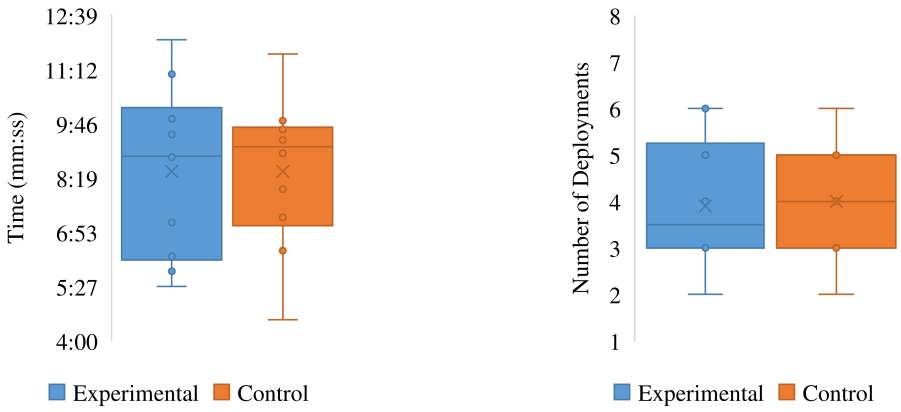
Table 13.1: Time spent and number of deploys in control task 2 (CT2).

	Grp	N	Mean	σ	Med	S-W (ρ)	Levene (ρ)	*t*-test (ρ)
Time	A	10	8:30	2:00	9:09	0.69	0.54	> 0.99
	B	10	8:30	2:15	8:54	0.61		
Deploys	A	10	4.00	1.25	4.00	0.55	0.75	0.87
	B	10	3.90	1.37	3.50	0.16		

We used CT2 to verify if there was a statistical difference between the two experimental groups by measuring the *time spent* and *number of deployments* required, as presented in Table 13.1 (p. 251).

We start with the Levene's test verifying if *both groups are from populations with equal variances*. As the obtained ρ-value is 0.54 for *time*, and 0.75 for the *number of deployments*, we cannot reject the *null hypothesis* (*i.e.*, both groups present equal variances). A Shapiro-Wilk test verifies if each of the *groups were drawn from populations with a normal distribution*. Since the resulting ρ-value is above the significance level (time: $\rho(GA) = 0.69$ and $\rho(GB) = 0.61$; deployments: $\rho(GA) = 0.55$ and $\rho(GB) = 0.16$), we also fail to reject the *null hypothesis* (*i.e.*, both groups present a normal distribution in the results). Ergo, we assume that both samples come from normally distributed populations with equal variances.

We then use a Student's *t*-test for assessing the following hypothesis related to *time*, viz. H_0: both groups needed a similar amount of *time* to complete the task, and H_1: there exists a significant difference in the average time for each group to complete the task. Concerning *deployments*, we assume H_0: both groups made a similar amount of deployments to complete the task, and H_1: there exists a difference in the average of deployments made to each group to complete the task.



(a) Time spent.

(b) Number of deploys.

Figure 13.4: Visualization of the time spent and number of deploys in control task 2 (CT2).

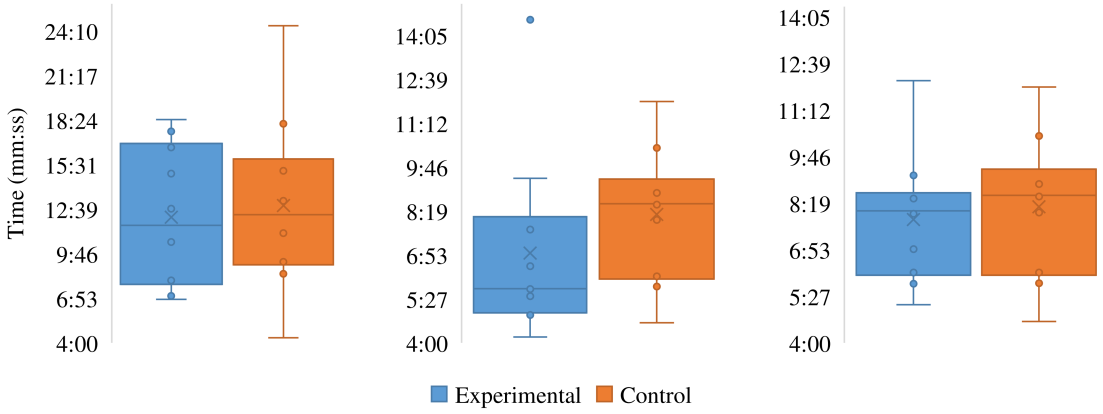
We observe that the *time spent* has a ρ-value=0.997 and the *number of deployments* has a ρ-value=0.866, failing to reject H_0, and thus be forced to consider that there is no statistical difference between the two groups, as intended (*cf.* Figure 13.4, p. 251).

Experimental Tasks

Taking into account the hypothesis as described in the *Control Tasks*, we present the results of the Experimental Tasks, together with a qualitative analysis.

Table 13.2: Time spent per experimental task (ET1–3).

Task	Grp	Mean	σ	Med	*t*-test (ρ)
ET1	A	12:53	5:34	12:17	0.75
	B	12:08	4:33	11:36	
ET2	A	8:13	2:10	8:34	0.30 (0.03*)
	B	6:57	3:05	5:47	
ET3	A	8:34	2:32	8:12	0.47
	B	7:49	1:59	8:05	



(a) Time spent in ET1. **(b)** Time spent in ET2. **(c)** Time spent in ET3.

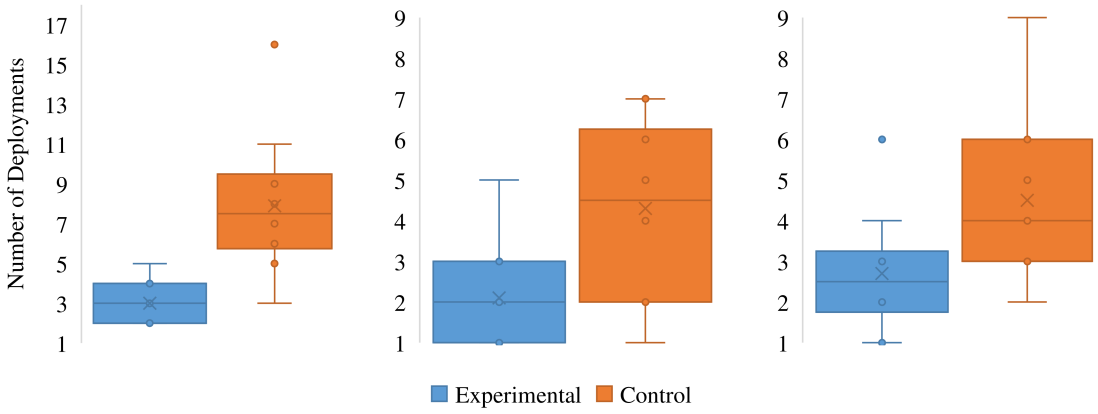
Figure 13.5: Visualization of the time spent per experimental task (ET1–3).

Time Analyzing the *time spent* for the three tasks and the results from the *t*-test (*cf.* Table 13.2, p. 252), we were initially unable to reject the *null hypothesis* for all tasks. We started by concluding there are no relevant differences between the two groups (*cf.* Figure 13.5, p. 252). However, a Grubb's test for outliers singled out one in ET2, forcing us to discard it (*cf.* Figure 13.5b, p. 252), resulting in a $\rho^\star$-value of 0.03. This allows us to conclude that the experimental group does present a statistical difference when adding new features to an existing system concerning *time*. Regarding the other tasks, we believe that they might have not captured a sufficient degree

of difficulty/complexity to evidence substantial differences and/or the sample size was insufficient. We do consistently observe a lower mean and median for all tasks in the experimental group.

Table 13.3: Number of deploys per experimental task (ET1–3).

Task	Grp	Mean	σ	Med	*t*-test (ρ)
ET1	A	7.90	3.60	7.50	$< \underline{0.01}$
	B	3.00	1.05	3.00	
ET2	A	4.30	2.11	4.50	$\underline{0.01}$
	B	2.10	1.29	2.00	
ET3	A	4.50	2.07	4.00	$\underline{0.04}$
	B	2.70	1.49	2.50	



(a) Deploys in ET1. **(b)** Deploys in ET2. **(c)** Deploys in ET3.

Figure 13.6: Visualization number of deployments per experimental task (ET1–3).

Deployments All experimental tasks present ρ-values lower than the significance level (0.05). This allows us to reject the *null hypothesis* and accept there is a significant difference in the average number of deployments made between the groups, with the experimental performing fewer attempts.

Comparing the mean and median of the *number of deployments* to reach the solution (*cf.* Table 13.3, p. 253), there is a clear tendency for the experimental group to need fewer deployments — nearly half compared to the control group. This aligns with our initial hypothesis since every time the user needs to add new debug nodes in the control group; they are forced to deploy. On the other hand, the experimental group was presented with real-time feedback, thus decreasing such need.

Verification Requests A *verification request* occurred every time a participant stated that their task were completed. The statistical analysis allows us to reject the *null hypothesis* on

Table 13.4: Number of correctness verification requests per experimental task (ET1–3).

Task	Grp	Mean	σ	Med	*t*-test (ρ)
ET1	A	1.50	0.53	1.50	0.33
	B	1.80	0.79	2.00	
ET2	A	1.50	0.53	1.50	0.05
	B	1.10	0.32	1.00	
ET3	A	1.80	0.92	1.50	0.04
	B	1.10	0.32	1.00	

both ET2 and ET3 (*cf.* Table 13.4, p. 254). Regarding the construction and evolution tasks, we conclude that there is a significant difference between groups concerning their subjective perception of task completion, as the experimental group required fewer attempts.

Behavior We observed that the experimental group, especially during ET1, changed their debugging strategy by focusing on visualizing and understanding the messages in the system instead of attempting to understand the underlying logic of each node. This was one of the most notices observations since it represents a change in the participants' behavior when approaching their tasks. This finding merits further study before any major conclusions can be drawn.

Experimental Group Feature Usage Analysis

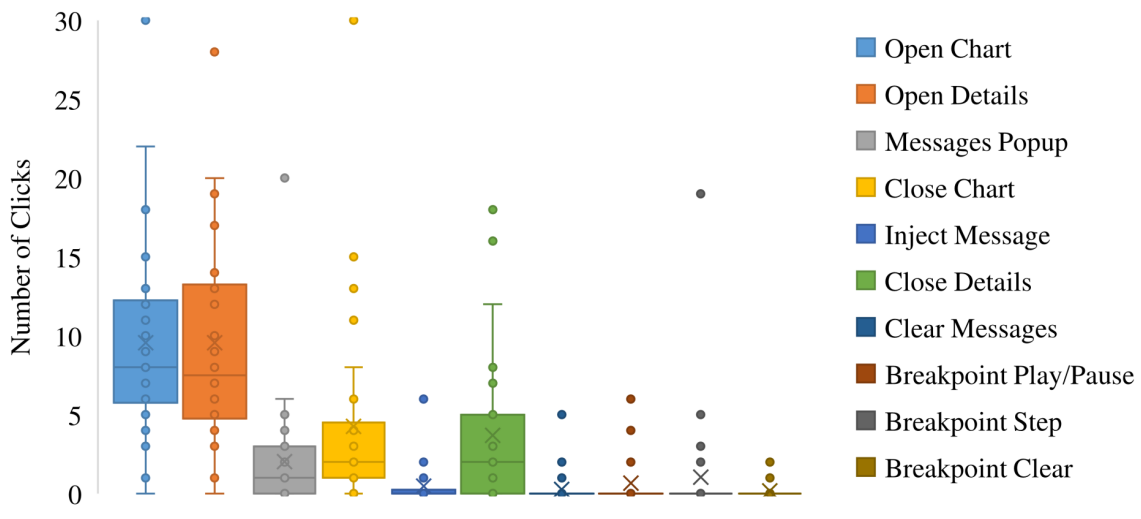
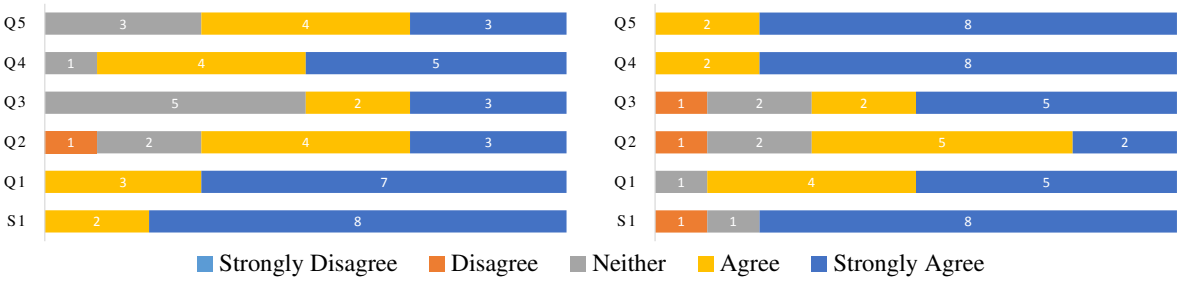


Figure 13.7: Cumulative number of clicks on NODERED-CAULDRON introduced functionalities.

After aggregating the results for each task (*cf.* Figure 13.7, p. 254), we conclude that the most used features in NODERED-CAULDRON were those related to the visualization of the messages, *i.e.*,

Table 13.5: Total number of clicks aggregated by experimental task (ET1–3).

Task	Mean	σ	Med	Min.	Max.
ET1	54.60	34.36	43.00	21	130
ET2	17.50	12.64	12.50	1	35
ET3	23.10	15.38	21.00	4	61



(a) NODERED-CAULDRON. **(b)** Node-RED.

Figure 13.8: Results of the survey post-test showcasing the subjective perception of the study participants' over the NODERED-CAULDRON introduced features and possible additional features.

(1) plot, (2) detailed message, and (3) history. In terms of usage by task (*cf.* Table 13.5, p. 255), we observe a higher mean and median for ET1, followed by ET3 and then ET2. These results were expected, since on ET1, participants spent more time in understanding the system, and consequently, the messages that flow through it. In ET2, the extra features were not used as much because the participants already understood the system and did not feel the need for a deeper exploration. ET3 was focused on constructing a new system, which results in the observed higher values as they attempted to understand the messages' flow.

Post-test Survey

To evaluate the participants' experience, we performed a post-test survey composed of six questions, one (S1) about the general satisfaction of using the solution — Node-RED vanilla and NODERED-CAULDRON —, and five concerning added or possible to add functionalities (Q1–Q5), namely, the usefulness of:

Q1 showing the input's messages on each node;

Q2 showing the plot that shows the messages;

Q3 the breakpoint system;

Q4 having typed connections between nodes;

Q5 having a highlighting mechanism of the path of a message in a *flow*.

Although only the experimental group (GB) used some new features, we also asked the control group (GA) if they *would* like to have had such features. Interestingly, we have found a proximity between the two groups (*cf.* Figure 13.8, p. 255). The highest divergence was found in Q4 and Q5, which referred to unavailable features on both groups (*i.e.*, these were not implemented in NODERED-CAULDRON). This can be explained considering that the experimental group was exposed to the experience of having real-time feedback during development and not feeling the need for these extra features. In Q3, the results were similar since, with our tool, participants ended up not using breakpoints. We conclude that most participants seem to want the functionalities described in each question. Finally, the results of S1 suggest that the experimental group had a more enjoyable experiment.

13.3 Discussion

Taking into account the experimental results presented in Section 13.2.3, we now revisit our research questions:

SRQ1. *Would users with increased exposure to real-time information about the running system build and manage it faster?* Both groups spend a similar amount of time in solving the tasks, with a statistically significant difference observed on *improving* systems. We also note that the experimental group presented consistently smaller mean and median values;

SRQ2. *Providing users with real-time feedback increases their ability to understand and change existing systems?* According to the number of deployments performed per task together with the qualitative analysis, we can conclude that in a system with higher feedback, users tend to perform fewer attempts of deployment, thus pointing that these features make the system easier to change;

SRQ3. *Is an IoT visual programming environment able to reduce human-induced errors during development by providing real-time feedback?* By analyzing the number of deployments and attempts, we see a substantial difference where users in the experimental group have less need to deploy and more confidence in their solution (*i.e.*, they required fewer attempts to achieve a successful task completion). This can be especially useful in more sensible systems, where deployments should be kept to a minimum.

Therefore, we conclude that there is significant evidence that an environment with real-time feedback and improved debug capabilities impacts the ability to build, maintain and improve IoT systems.

13.4 Summary

IoT systems and their application across application domains with different constraints and responsibilities boosted their heterogeneity and complexity at large. The tremendous gap of qualified personnel to design, develop and maintain these systems has pushed both industry and academia to create new ways to develop IoT systems that abstract the system's complexity at different degrees. One of those approaches, already used for PLCs, was visual programming.

Among those, Node-RED appeared as one of the most common solutions to develop IoT systems.

In this chapter, we describe our efforts to overcome some Node-RED development environment drawbacks by enhancing the existing Node-RED with new features that improve feedback during development and debugging capabilities. With the goal of asserting how such features would impact the development of IoT systems, a POC was developed, and an empirical evaluation followed with 20 participants. We conclude that the added enhancements improve the overall development process, with a significant reduction of the number of failed attempts to deploy the systems without fulfilling its requirements. Further, the overall system development time was lower than with the normal Node-RED.

14 | Conversational Assistant for IoT Automation

As the number of devices and interactions grows, so does the management requirements (and management complexity) of the system as a whole, as it becomes essential to understand and modify the way they (co)operate. In the literature, this capability is typically defined as *end-user programming* [Fis+04]. Once we discard trained system integrators and developers, two common approaches emerge, low-code visual programming solutions and conversational assistants [Zar18]. More complete solutions exist, such as Node-RED [Gen+17; Ray17; PC13], but they also have several limitations and shortcomings, especially for end-users without any relevant technical expertise (*cf.* Section 3.2.7, p. 87) [Sei+14]. As an example, consider a Node-RED system orchestrating a user's smart home with multiple devices. Even in situations where there are only a couple of rules defined, it can be challenging to understand why a specific event occurred due to the overwhelming data flow resulting from these. Further, just a small amount of rules can already lead to a system not possible to visualize in a single screen [JEC14]. The more rules one adds, the harder it becomes to grasp what the system can do conceptually. Part of the reason is that these solutions are built to be *imperative*, not *informative*; current solutions mostly lack meta-facilities that enable the user or the system to *query* itself.

In this chapter, we present an approach to address the problem of *managing* IoT systems using a conversational approach towards shortening the existing feature gap between assistants and visual programming. The approach is concretized in a POC named Jarvis, which was then evaluated with a user-study.

> Parts of this chapter were published in the work CONVERSATIONAL INTERFACE FOR MANAGING NON-TRIVIAL INTERNET-OF-THINGS SYSTEMS [DLF20] and subsequent work MANAGING NON-TRIVIAL INTERNET-OF-THINGS SYSTEMS WITH CONVERSATIONAL ASSISTANTS: A PROTOTYPE AND A FEASIBILITY EXPERIMENT [LDF21]. The chapter was partially based on the master thesis work of André Lago entitled EXPLORING COMPLEX EVENT MANAGEMENT IN SMART-SPACES THROUGH A CONVERSATION-BASED APPROACH [Lag18]. The author's main contributions were on the formal analysis of the study data, data curation, visualization, and writing of the published versions of the work.

14.1 Approach Overview

One of the common, sometimes complementary, alternative to visual programming are conversational assistants (also known as voice assistants). There exist a plethora of conversational assistants in the market, such as Google Assistant, Alexa, Siri, and Cortana (see [Mit18] and [LQG18] for a comparison of these tools) which are capable of answering natural language questions. Recently, these assistants have gained the ability to interact with IoT devices, with Ammari *et al*. identifying IoT as the third most common use case of voice assistants [Amm+19].

Among the most common features they provide is allowing direct interaction with sensing and actuating devices, which enables the *end-user* to *talk* to their light bulbs, thermostats, sound systems, and even third-party services. The problem with these solutions is that they are mostly composed of *simple* commands and queries directly to the smart devices (*e.g.*, "*is the baby monitor on?*", "*what is the temperature in the living room?*", or "*turn on the coffee machine*"). These limitations mean that although these assistants do provide a comfortable *interaction* with devices, a considerable gap is easily observable regarding their capabilities on *managing* a system as a whole and allowing the definition of rules for how these *smart spaces* operate. Even simple rules like "*close the windows every day at 8 pm*" or "*turn on the porch light whenever it rains*" are currently not possible unless one manually defines every single one of them as a capability via a non-conversational mechanism (other motivational examples can be found in Chapter 5, p. 123). Furthermore, most assistants are deliberately locked to specific vendor devices, thus limiting the overall experience and integration.

Although current smart assistants can be beneficial and comfortable to use, they do not have the completeness of other solutions, *e.g.*, Node-RED. Meanwhile, visual programming solutions are still far too technical for the common *end user*.

As far as we could find (*cf.* Section 3.2, p. 68) there is no solution that would simultaneously provide: (1) a non-trivial management of an IoT system, (2) be comfortable and easy to use by a non-technical audience, and (3) improve the user capability to understand how the system is functioning. By *non-trivial* we mean that it should be possible to define new rules and modify them via a conversational approach, achieving a *de facto* integration of multiple devices, not just directly interacting with its basic capabilities. The comfort would be for the user not to have to move or touch a device to get his tasks done (*i.e.*, using voice), or edit a Node-RED visual *flow*. As to understanding their system's functioning, we mean the ability to grasp *how* and *why* something is happening in their smart space. This last point, combined with the other two, would ideally allow someone to ask why something happens.

14.1.1 Conversational Interaction

We propose the development of a conversational assistant — Jarvis — dedicated to the management of IoT systems and capable of defining and managing complex system rules while providing information about the running system.

An example interaction with Jarvis by text messages on Slack can be seen in Figure 14.1 (p. 260). Jarvis provides users with several features with the aim of covering most of the interactions a user could have with physical smart spaces. The choice of these specific

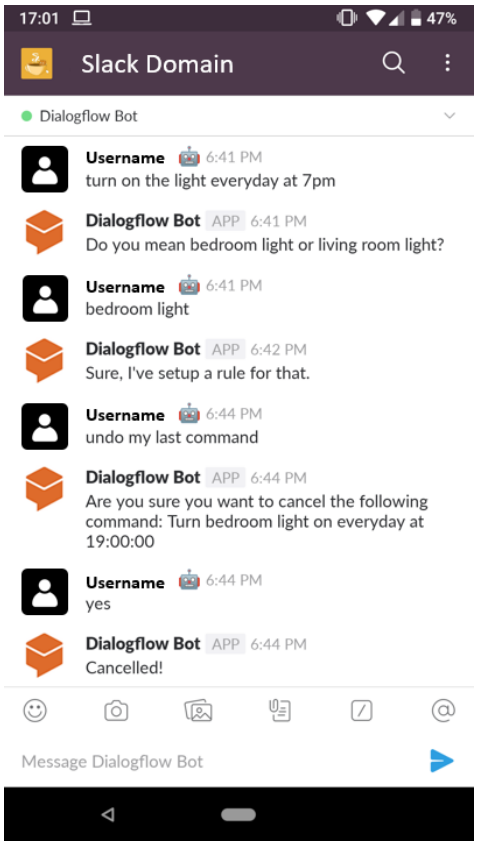


Figure 14.1: Chat with Jarvis by Slack integration.

functionalities was based on the results of the end-user survey presented in Chapter 5 (p. 123) together with the most common actions one can find in similar works and surveys [Bri+17], including those identified by [Amm+19] as main drawbacks in voice assistants. Complementarily, as far as we could find, there are little to no mentions in the literature for causality and rules queries. We have thus chosen to support the following functionalities:

Direct actions Single direct action that happens instantly, *e.g.*, *"Turn on the light"* or *"What is the current temperature of the kitchen?"*;

Delayed actions Single delayed action that happens after a certain time period, *e.g.*, *"Turn on the light tomorrow at 5 pm."*;

Repeating actions Defines a rule for an action that should be performed every day, *e.g.*, *"Turn on the light every day at 5 pm."*;

Event-triggered actions Creates an action that is performed upon a certain event, such as an activity of another device or a change of a device's status, *e.g.*, *"Turn on the light when the bedroom motion sensor is activated."*;

Causality queries Used when the user wants to know why a certain condition is true or why a certain action took place, *e.g.*, *"Why did the light turn on?"*;

Alias actions Used for the user to create an action/event that associates to a custom phrase, *e.g.*, *"Make an alias for 'party time'."* [system asks what to set the alias for] *"Turn on all lights."*;

Rules query Used to know which rules are defined for a device (allowing the modification of those rules), *e.g.*, *"What rules are defined for the living room light?"*;

Cancel command Cancel the last user command. If that was a direct action command, the action is undone, and, if the command was a rule command, the rule is canceled, *e.g.*, *"Cancel my last command."*.

Jarvis also uses *contextual awareness* to improve the user's experience and make the interaction resemble a real human-human interaction. *Contextual awareness* allows Jarvis to understand the meaning of a user query based on queries issued previously, which can be applied in many scenarios:

Device specification that is used when the device specified in an action query is unclear or ambiguous so that the user can specify the device he wants to choose.

User: *"Turn on the light."*
Jarvis: *"Do you mean the living room light or the bedroom light?"*
User: *"The bedroom light."*
Jarvis: *"Sure, light turned on."*

Updating system rules which context is used to enable following-up of the *Rules query* action, allowing to change the presented rules.

User: *"What rules are defined for the bedroom light?"*
Jarvis: *"You told me to turn the bedroom light on every day at 8 AM."*
User: *"Okay, change it to 7:50 AM."*
Jarvis: *"Sure, rule changed."*

Causality queries which context enables the user to have a dialog to understand the possible reason why something happens (instead of a single direct answer which could be not understood by the user). **User**: *"Why did the toaster turn on?"*
Jarvis: *"You told me to turn it on at 10 AM."*
User: *"Okay, change it to 9 AM."*
Jarvis: *"Sure, toaster timer was changed."*

It is noticeable that in all the examples above, the second user query would be meaningless on its own. However, it makes sense when represented along with the previous user query and Jarvis' first response. These examples showcase how *contextual awareness* can make interactions with Jarvis feel more natural, which improves the user's experience.

To ease the integration with nowadays systems and provide us with a *reproducible experimental environment*, we integrated the interface with some existing platforms, namely: Google Assistant [Goo20] and Slack [Sla20]. Integration with other services is also possible, and one can interact with Jarvis both via *voice* and *text*.

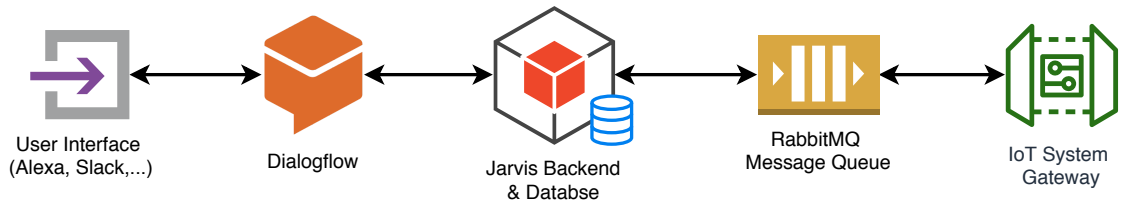


Figure 14.2: High-level view of Jarvis architectural components.

14.1.2 Implementation Details

The Figure 14.2 (p. 262) presents the high-level software components of Jarvis. Each component and corresponding techniques are explained in the following subsections.

Conversational Interface

To develop the conversational interface, we decided to opt for Dialogflow[1] as this platform provides built-in integration with multiple popular *frontends* and there exists extensive documentation for this purpose [Jan17a]. In this case, we used (1) the Slack team-communication tool (*cf.* Figure 14.1, p. 260), and (2) Google Assistant, so that both text and voice interfaces were covered. In the case of Google Assistant, the user may use any supported device paired with their account to communicate with Jarvis, following a known query prefix such as *"Hey Google, talk to Jarvis"*. Regardless of which type of interface is used, the result is converted to *strings* representing the exact user query and subsequently sent to Dialogflow's backend (thus overcoming potential challenges due to Speech Recognition), which are then analyzed using NLP techniques. Advancement of the existing NLP techniques made available by Dialogflow falls out-of-the-scope of this work.

Dialogflow Backend

Upon receiving a request, Dialogflow can either produce an automatic response or send the parsed request to a fulfillment *backend*. This component is thus responsible for parsing the incoming *strings* into a *machine understandable* format (JSON). There are a few key concepts that are leveraged in our implementation:

Entity. Things that exist in a specific IoT ecosystem can be represented by different literal strings; for example, an entity identified by `toggleable-device` may be represented by *"living room light"* or *"kitchen light"*. Additionally, entities may be represented by *other* entities. Dialogflow uses the `@` symbol (*i.e.*, `@device`) for refering to entities, and provides some system's defaults;

Intent. An intent represents certain type of user interaction. For instance, an intent named *Turn on/off device* may be represented by `turn the @device on` and `turn the @device off`. For a request such as *"turn the kitchen light on"*, Dialogflow understands that `@device` corresponds to *kitchen light* and provides that data to the fulfillment backend;

[1] Dialogflow, https://dialogflow.com/

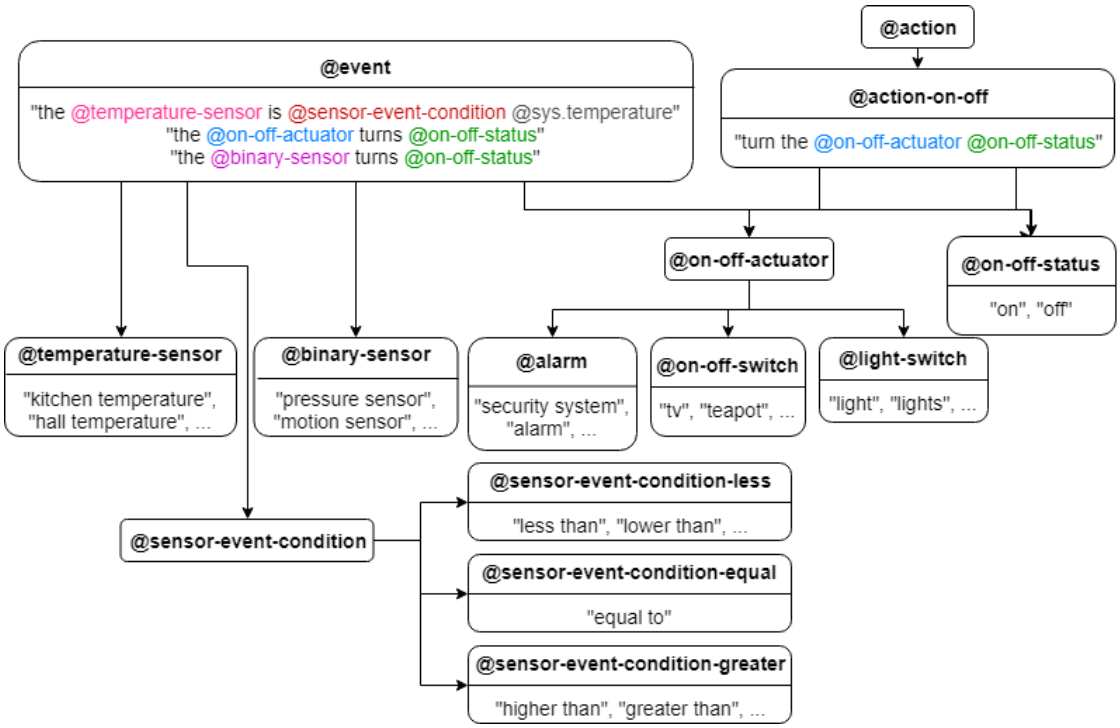


Figure 14.3: Main entities defined in Jarvis' Dialogflow project.

Context. Contexts allow intents to depend on previous requests, enabling the creation of context-aware interactions. This is foundation to support queries such as *"cancel that"* or *"change it to 8 am"*.

Multiple *intents*, *entities*, and *contexts* were defined in Jarvis and the main ones are illustrated in Figure 14.3 (p. 263). Here we provide in detail one of its *intents*:

Usage Creates an action that is performed upon a certain event, such as an activity of another device or a change of a device's status.

Definition `@action:action when @event:event`

Example *Turn the bedroom light on when the living room light turns off.*

With the above definitions, this component takes requests and builds the corresponding objects containing all actionable information to be sent to the Jarvis backend for further processing. For that, Dialogflow generates a JSON object that contains the exact user query, but also an identifier for the intent type, identifiers for the recognized entities, relevant contextual metadata and default answers (if any were specified in the Dialogflow configuration UI). This JSON is sent to the Jarvis backend via an HTTP request, to which Jarvis responds with a JSON containing the intended response along with other possible data such as contextual metadata.

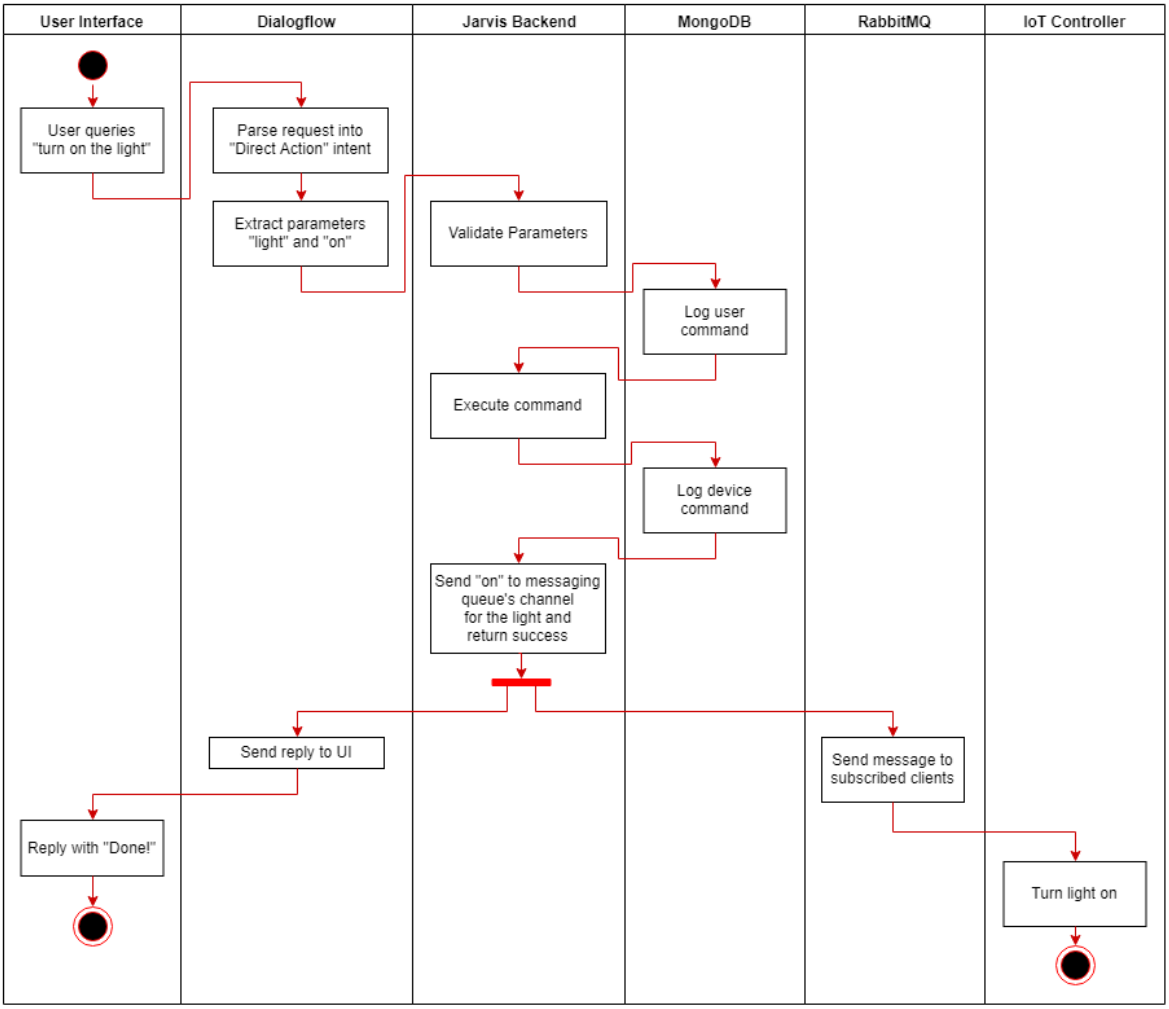


Figure 14.4: Sequence diagram for the parsing and execution of the query *turn on the light*.

Jarvis Backend

For each of the intents defined in Dialogflow, this component provides an equivalent class responsible for handling that intent, *cf.* `handler classes`. Jarvis makes use of a MEDIATOR pattern to deliver each user query to the right handler.

Each `handler class` provides the same methods to the mediator, the main of each being a `handle` method that takes in the user query as represented by Dialogflow's JSON object. Similarly, it returns the result object which should be sent to Dialogflow, containing Jarvis' response.

The `handler classes` are responsible for (a) parsing the request, (b) validating its request parameters (*e.g.*, device name or desired action), and (c) generating an appropriate response. An overview is provided in Figure 14.4 (p. 264). Should the request contain errors, an *explanatory* response is returned. When all the parameters are considered valid, but the intended device is *unclear* (*e.g.*, user wants to turn on the light; however, there is more than one light that can be the target of the command), the generated response specifically asks the user for further clarification in order to gain *context*.

In addition to the Dialogflow's JSON representation of the user query, the Jarvis backend represents user commands using the command design pattern. This provides a straightforward way to *execute*, *cancel* and *undo* mechanisms, as well as keeping a history of performed actions, which showcases its usefulness, especially in *causality queries*.

This internal representation of commands makes use of the Web Things API (*cf.* Section 2.2, p. 38), API which documents a symbolic representation of multiple devices along with their capabilities, which is useful for the Jarvis backend to be aware of a device's capabilities and features. The use of this representation enables Jarvis to know whether a specific action (*e.g.*, turning something on) applies to a particular device (*e.g.*, a light).

Interaction with IoT Devices

For the interaction with the physical IoT, we chose a simple yet functional set of technologies that would allow us to validate the functionality of the Jarvis backend. We used RabbitMQ [VMw20] as the message queue system since it supports a variety of protocols (such as AMQP, STOMP, and MQTT), allowing easy communication with devices through simple path strings (*e.g.*, `/house/kitchen`). The message queue system provided the necessary infrastructure for the Jarvis backend to communicate with the IoT devices — while being agnostic of their physical location on the network. An alternative setup could require the backend to know the IP addresses of each individual device, which would require much more maintenance if those addresses changed over time.

In order for Jarvis to know which devices exist in the system, how to communicate with them and what capabilities they have, a Device Registry [Ram+17] was set up, and such information was stored using a MongoDB [Mon20] document-based database. This database was also used to store the history of user queries and executed commands, which allows the system to provide features such as the causality queries even if it is temporarily shut down.

The direct interaction with the IoT devices was simulated using *Python* scripts that publish the devices' state changes to the message queue, as well as read instructions provided by Jarvis, applying them to the respective devices.

In the experimental setup we used in the validation of this project, the Jarvis was deployed in a Virtual Private Server (VPS) being easily accessed from any location.

14.1.3 Interaction Capabilities

Jarvis supports several interactions beyond the ones available on common voice assistants [Amm+19], allowing multi-stage conversations to describe or define automation scenarios or even find and analyze the probable cause of certain events. A complete list of the supporting queries (*i.e.*, interactions) is given in the following paragraphs with supporting examples.

Contextual Awareness

The first example of *contextual awareness* happens when the user makes a query with an unclear device. Here, Jarvis sets *contextual metadata* on the response set to Dialogflow. This metadata

is then re-sent to Jarvis by Dialogflow on the following user query, which allows Jarvis to understand interactions such as:

User: *"Turn on the light."*
Jarvis: *"Do you mean the bedroom light or the kitchen light?"*
User: *"The second one."*

Because of the *contextual metadata* set by Jarvis during the second response, when the user says *"The second one."*, Jarvis knows that the user is referring to the *"kitchen light"*, and therefore knows that it must continue the initial query and turn on that device.

In the example above, the second user query is assigned by the *mediator* to a specific *handler class* which is able to decode the contextual metadata and generate the corresponding user *command*.

Period Actions

For most intents, such as *direct actions* or *"why did something happen?"* queries, the effects are immediate. However, *period actions*, *events*, and *causality queries* require a different design approach so that they can perform actions on the backend without the need for a request to trigger them.

A *period action* is an intent that must be carried and then undone after a certain period (*e.g.*, *"turn on the light from 4 pm to 5 pm"*). In these scenarios, the Jarvis backend generates a *state machine* to differentiate between all the different action status, such as (a) nothing has executed yet (before 4 pm), (b) only the first action was executed (after 4 pm but before 5 pm), and (c) both have been executed (after 5 pm). We use a combination of *schedulers* and *threads* to guarantee proper action, and abstract all these details inside the command pattern. The same strategy applies for rules such as *"turn on the light every day at 5 pm"*, with the appropriate state machine and scheduler modifications.

In these examples, the already mentioned command representation becomes useful once again since it allows the system to manage these period actions easily. For instance, if the user wishes to change an active rule (*e.g.*, *"turn on the light from 4 pm to 6 pm"* instead of *"turn on the light from 4 pm to 5 pm"*), the Jarvis backend can cancel the active *command*, create a new instance with the updated rule and start it immediately. This update of an active *command* is itself represented as a command, which also allows the user to revert unintentional changes to other rules.

External Events

This state-machine mechanism is different for actions that are the result of external events such as *"turn on the kitchen light when the presence sensor is activated"*. These are notably different because, although direct actions and period actions depend only on the internal state of the Jarvis backend, event-bound actions are dependent on analyzing external events such as a sensor changing its state.

To implement this functionality, we leverage a *publish-subscribe* approach which orchestrates multiple unique and identifiable *message queues*. Each message queue is associated with

one or multiple devices, and it serves as a bidirectional communication layer between them and the Jarvis backend. For instance, when Jarvis wishes to change the state of a certain device, it publishes a message on the respective queue with a format that identifies the specific device to change and what that change requires. It is then the responsibility of that device's controller to read this message and perform the change. Messages published on these queues also leverage the Web Things API.

When it comes to events, communication happens in the reverse order. Each time a sensor's value changes (*e.g.*, a motion sensor is triggered or the temperature changes), that device's controller publishes a descriptive message on the message queue. The Jarvis backend then uses observers that read the message and decide whether any active command is responsible for handling it. If so, it calls a method on that command that handles the message.

This means that a user query such as *"turn on the kitchen light when the presence sensor is activated"* generates a command that knows it must handle changes to the presence sensor, such that when this happens, this command is called by the observer, causing the light to be changed accordingly.

Causality Queries

These relate to the user asking why something happened (*e.g.*, *"why did the light turn on?"*). These are a unique feature of Jarvis which are useful for users not only because they allow them to remember what are the operation rules of their system, but also because they allow users to easily change how their system works with nothing but their voice.

To implement them, we augment each command such that each command can determine whether it can cause a specific condition to be true. For instance, the command *"turn on the light when the presence sensor is activated"* knows that a possible consequence of its operation is the condition *"light is turned on"*.

With this augmentation, when the user queries Jarvis on why some condition happened, Jarvis can iterate through the log of recently executed commands and return the latest one that could have caused the queried condition, providing an informative answer (*e.g.*, *"because you asked me to turn it on at 3:56 pm"*).

However, there might exist multiple rules that may have caused the condition to be true, in which case it is not enough to blame the latest logged command. In order to expand this functionality to provide more accurate answers, we considered three different approaches:

Return the immediate possible cause This is the currently implemented approach. It is likely to provide an accurate answer in the sense that the response is always the latest action that caused the queried event. Nevertheless, this does not necessarily imply that it is the most relevant cause (*e.g.*, if multiple commands could cause the queried condition, the first of these was the one that first led to that condition).

Return the first possible cause In some scenarios, multiple rules might have been involved in the change of the current system state, and they might either be part of a "causal chain", or simply overlap in their outcome. It is debatable whether the most relevant action in the chain would be the most immediate, the root event, or anything in between.

However, in the case of overlapping, it seems that the first event to have occurred (in the sense of sequence) might be the most reasonable to blame — since it is the one that transited the state — and which was latter "reinforced" by other causes (*e.g.*, if multiple rules could have caused the light to turn on, only the first of which caused the light's state to be changed). Hence, this first rule might be the most relevant answer in some cases.

Use relevance heuristic A relevance heuristic could provide the benefits of both of the previous approaches, perhaps even providing more probable causes. In a situation where multiple rules or events could have caused the queried condition, using a heuristic could provide an answer that was more useful to the user. For instance, if both a period event and an event action could have caused the condition, a heuristic could consider the event to be a more relevant condition since it is caused by external interactions rather than the well-defined mechanisms defined by the user.

Another non-trivial scenario is where the explanation is due to a chain of interconnected rules. Here, it seems that one can (a) reply with the complete chain of events, (b) reply with the latest possible cause, or (c) engage in a *conversation* through which the user can explore the full chain of events as they deem adequate (*e.g.*, "tell me more about things that are triggered by rain"). In this work, we opted to use the earliest possible cause for the first scenario, and the latest for the second; more complex alternatives can be found in [Bra+17; Aga+18].

14.2 Experiments and Results

To understand how Jarvis compares to other systems, we established a baseline based on (1) a visual programming language, and (2) a conversational interface. Node-RED was picked among the available visual programming solution. Google Assistant was selected for the conversational interface due to its natural responses[2]. There are plenty of ways users can interact with it: (a) the standalone Google apps, (b) built-in integration with Android and Chrome OS, or (c) with standalone hardware such as the Google Home. We compare to this baseline according to two criteria: (1) the *number of different features*, and (2) their *user experience* in terms of easiness of usage and intuitiveness. For the first, we created a list of simulated scenarios to assess the ability to manage IoT systems. We then performed a feasibility experiment with users to assess the second criteria.

14.2.1 Simulated Scenarios

A total of 10 simulated tasks was performed with the goal of comparing Jarvis with two solutions available in the market: Node-RED and Google Assistant. The Table 14.1 (p. 269) summarizes the comparison of our prototype to the chosen baseline.

The (1) *one-time action* refers to a direct trigger of a device, which is possible in both voice assistants and through the Node-RED interface. The (2) *one-time action with unclear device* refers to actions like *"turn on the light"* with which Jarvis asks the user to clarify which device he

[2] The work by López *et al.* [LQG18] compares Alexa, Google Assistant, Siri, and others, and claim that although *"Siri was the most correct device (...) Google Assistant was the one with the most natural responses".*

Table 14.1: Comparison of the different interaction scenarios supported by Jarvis, Google Assistant and Node-RED.

	Jarvis	Google Assistant	Node-RED
One-time action	●	●	●
One-time action w/unclear device	●	·	·
Delayed action	●	·	●
Period action	●	·	●
Daily repeating action	●	·	●
Daily repeating period action	●	·	●
Cancel last command	●	·	·
Event rule	●	·	·
Rules defined for device	●	·	·
Causality query	●	·	·

means based through responses such as *"do you mean the bedroom or living room light?"*. Queries such as (3) *delayed action*, (4) *period action*, (5) *daily repeating action* and (6) *daily repeating period action* are possible to carry using the Jarvis assistant and with the Node-RED solution. The query (7) *cancel the last command* refers to the ability to undo the last action or rule creation by explicitly saying that, and while that is possible to be carried on Jarvis, neither Google Assistant nor Node-RED support this behavior.

In the case of an (8) *event rule*, the system must support the dynamical creation of trigger-action rules based on an event (*e.g.*, the trigger of a motion sensor or when a button is clicked), which is possible using Jarvis, but in Node-RED requires manual changes to the programmed *flows*. Query (9) *rules defined for device* refers to the user performing queries that require introspection, such as *"what rules are defined for the bedroom light?"*, which Jarvis is capable of, but this capability is not available in Google Assistant. In Node-RED, this can be accomplished up to a certain point by visual inspection of the *flows*, though it has several limitations[3]. Concerning (10) *causality query*, the solution should provide a reasonable cause for a given event, which is only possible in Jarvis.

It is observable that our prototype provides several features that are not present in either the Google Assistant or Node-RED. Both of these products do a lot more than these features. However, regarding managing smart systems, the advantage of Jarvis is evident, especially when compared to the Google Assistant given that the only type of feature it supports are *one-time direct actions* [Amm+19]. Our second conclusion is that it is possible to bring some features currently available in visual programming environments to a conversational interface; the converse (how to bring conversational features to Node-RED), eludes us.

It is essential to mention that both Node-RED and Google Assistant are systems with broader goals than just automating the management of IoT systems. Node-RED is capable of

[3]As an example of such limitation is that if more than one device is connected to the same message queue it can be considerably difficult to understand which device produced a particular outcome and thus hard to understand if a rule was trigger due to a specific device *event*.

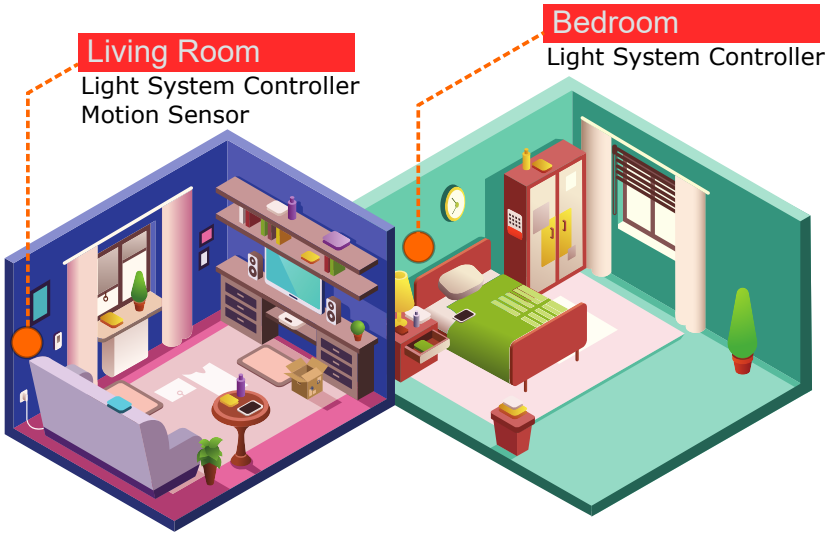


Figure 14.5: Visualization of the base space model used in the scenarios use for the feasibility experiment.

managing complex rules that connect multiple different systems. For instance, it allows users to send an automated email any time a tweet with a certain *hashtag* is published. The Google Assistant is also capable of many other features, such as listening to music or telling users about their upcoming flight reservations. Jarvis does not aim to provide any of these features, being tailored to IoT scope.

The comparison between these services and Jarvis on the limited scope of managing an IoT smart space is meant as a reinforcement of the value added by Jarvis in this limited scope, rather than downplaying the overall value and potential of the two systems used as comparisons.

14.2.2 Feasibility Experiment

In order to gain insight into how *end users* responded to a conversational approach, we performed a feasibility experiment with 17 participants. Our sample includes 14 participants without formal technological skills, with ages ranging from 18 to 51. The remaining 3 participants were students enrolled in the Masters in Informatics Engineering. We made sure that (a) all participants were familiar with the necessary technologies, such as basic usage of smartphones and the Internet, and (b) that even non-native English participants had adequate speaking and understanding skills, given that the prototype of Jarvis was implemented in the English language.

Methodology

Each participant was given 5 tasks to be completed using the same scenario with the help of Jarvis, using Google Assistant as the system interface. The scenario consisted of a *smart home* with a *living room light*, a *bedroom light* and a *living room motion sensor*, as depicted in Figure 14.5 (p. 270):

Task 0 (T0) Control task, *Turn on the living room light*;

Task 1 (T1) *Turn the living room light on in 5 minutes*;

Task 2 (T2) *Turn the living room light on when the motion sensor triggers*;

Task 3 (T3) *Check the current rules defined for the bedroom light, and then make it turn on every day at 10 pm*;

Task 4 (T4) *Find out the why the bedroom light turned on. Ask Jarvis why it happened and decide whether the answer was explanatory*.

The only instructions given to participants were that they should talk to the assistant (using the mobile phone version) in a way that feels the most natural to them to complete the task at hand. Besides the tasks, participants were also given the list of IoT devices available in the simulated smart house that they would be attempting to manage through.

Variable Identification

For each of the tasks, we collected (1) whether the participant was able to complete it, (2) the time to complete them, and (3) the number of unsuccessful queries. This count was made separately for (a) queries that were not understood by the assistant's speech recognition capabilities (*e.g.*, microphone malfunction, background noise), (b) queries where the user missed the intention or made a syntactic/semantic error (e.g., *"turn up the lighting"*), and (c) valid queries that a human could interpret, but that Jarvis was unable to.

Subjective Perception

After completing the tasks, we introduced a non-conversational alternative (Node-RED), explaining how all tasks could have been performed using that tool. We inquired the participants whether they perceived any advantages of Jarvis over such a tool and whether they would prefer Jarvis over non-conversational tools. Finally, the participants were asked if they had any suggestions to improve Jarvis and the way it handles system management.

Results

Table 14.2: Experimental results (task completion rate, task time and the number of incorrect queries), including average and standard deviation.

		Time (s)		IQ (G.A.)		IQ (User)		IQ (Jarvis)		IQ (Total)	
Task	Done(%)	$\bar{x}$	σ	$\bar{x}$	σ	$\bar{x}$	σ	$\bar{x}$	σ	$\bar{x}$	σ
T0	94%	6.41	1.12	0.24	0.56	0.12	0.33	0.24	0.56	0.59	0.87
T1	94%	7.35	1.46	0.24	0.44	0.25	0.50	0.24	0.56	0.53	0.72
T2	88%	9.94	1.20	0.35	0.70	0.35	0.61	0.53	0.8	1.24	1.15
T3	100%	19.71	1.96	0.24	0.56	0.24	0.44	0.47	0.62	0.94	0.83
T4	94%	8.65	2.32	0.29	0.47	0.29	0.59	0.12	0.33	0.71	0.85

Table 14.2 compiles the results observed during the study, each row representing a task given to the participant. Each column means:

Task Identification of the task (T0—T4);

Done Percentage of participants that completed the task successfully;

Time Time in seconds that participants took to complete the task;

IQ (G.A.) Number of occurrences of queries that were incorrect due to the Google Assistant (G.A.) not properly recognizing the user's speech;

IQ (User) Number of occurrences of queries that were incorrect due to the user not speaking a valid query;

IQ (Jarvis) Number of occurrences of queries that were incorrect due to Jarvis not recognizing a valid query;

IQ (Total) Total count of invalid queries, *i.e.*, sum of *IQ (G.A.)*, *IQ (User)* and *IQ (Jarvis)*.

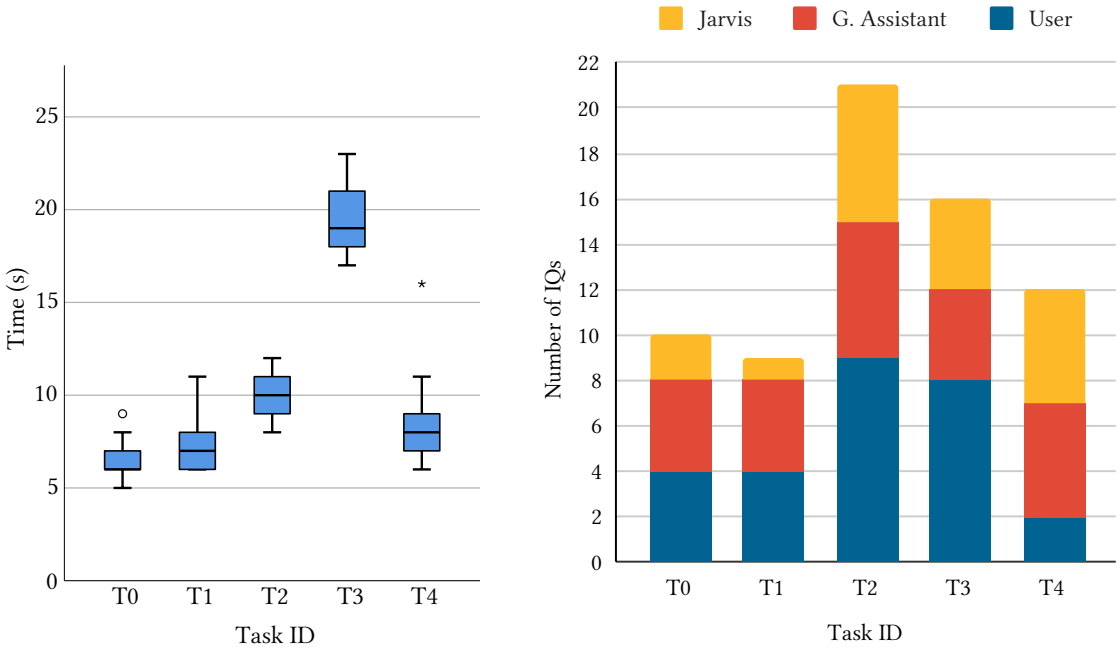
The collected results allow us to discuss how well Jarvis behaves when dealing with end-user voice entered commands. Further, it allows us to understand what is the origin of problems when commands are not understood by the approach (*i.e.*, either G.A., User, or Jarvis).

14.2.3 Results Analysis

The complexity of the queries increases from **T0** to **T3** since the queries require more words or interactions. This is reflected by the corresponding increase in task completion time, as seen in Figure 14.6a (p. 273). Some of the incorrect queries result from incorrect understanding at the (voice) assistant level, which means the speech recognition failed to translate what the participants said correctly. Although this does not have implications on the evaluation of Jarvis, it does indicate that this category of systems might be harder to use due if they are not multilingual.

Directly comparing the time needed to complete a task to what would be needed to perform it in a visual programming solution such as Node-RED is meaningless; either the task is not defined, and that would require orders of magnitude longer than what we observe here, or the task is defined and the times will be obviously similar. Similarly, we also observe a few instances of incorrect queries due to grammar mistakes or semantically meaningless, *cf. IQ (User)*, and therefore did not match the sample queries defined in Dialogflow. Nevertheless, there were grammatically incorrect user queries such as *"turn on lights"* but which still carries enough information to understand what the user's intent is.

We consider as a more serious issue the number of *valid* sentences that were considered incorrect queries by Jarvis, *cf. IQ (Jarvis)*, as it can be seen in Figure 14.6b (p. 273). These could have been caused by either a mispronunciation of a device's name or a sentence structure that is unrecognizable by the Dialogflow configuration. This possibly represents the most severe threat to our proposal, to which we will later dedicate some thoughts on how to mitigate it.



(a) Participant's task completion time per task.

(b) Number of IQs per task and their origin.

Figure 14.6: Overview of the performance of Jarvis, taking into account the task completion time and number of Incorrect Queries (IQs) of the solution.

Nonetheless, the success rate of all tasks is considerably high (always higher than 88%), which provides evidence that the system might be intuitive enough to be used without previous instruction or formation. These points were reflected by the participants' subjective perception, where they claimed Jarvis to be easy to use, intuitive, and comfortable; ultimately, these would be the deciding factors for end-users to prefer Jarvis over a non-conversational interface.

An additional observation was stated by some users pertaining Jarvis' answers, particularly those regarding causality queries (**T4**), where they claimed that if the provided response were too long, it would become harder to understand it due to the sheer increase of conveyed information. A possible solution for this problem would be to use a hybrid interface that provides both visual and audio interactions. However, there could be other approaches, such as an interactive dialogue that shortens the sentences.

In terms of subjective perception, when participants were inquired about their preference on visual programming solutions and the used voice interface, Jarvis, all of them pointed to conversational assistants as their preference, mostly due to its "ease of use", "commodity" and "accessibility". The most often referred downside was the issues with voice recognition ("margin of error that comes with voice recognition"). The participants mentioned that the main drawback of visual programming tools is the need to understand more technicalities on how the devices communicate and which actions (sensing/actuating) they can perform ("knowledge of how the hardware works"), and referred as the main advantage the large number of integrations that visual tools typically provide which lack in most conversational ones.

14.2.4 Threats to Validity

Empirical methods seem to be one of the most appropriate techniques for assessing our approach (as it involves the analysis of human-computer interaction), but it is not without liabilities that might limit the extent to which we can assess our goals. We identify the following threats:

Natural Language Capabilities where queries like *"enable the lights"* might neither be common nor semantically correct, but it still carries enough information so that a human would understand its intention. The same happens with device identification, such as when the user says *turn on the bedroom lights*, and the query fails due to the usage of the plural form. During our study, we observed many valid queries that did not work due to them not being covered by the Dialogflow configuration. This can be further addressed by creating a more extensive list of entities[4], and by training the DialogFlow model with more combinations of those entities;

Coverage error which refers to the mismatch between the *target* population and the *frame* population. In this scenario, our target population was (non-technical) end-users, while the frame population were all users who volunteered to participate;

Sampling errors are also possible, given that our sample is a small subset of the target population. Repeating the experience would necessarily cover a different sample population, and likely attain different results.

We attempt to mitigate these threats by (1) using state-of-art language understanding services, and (2) gather as many participants as possible to remove any bias in sample size and background; however, we acknowledge that experiment with different language understanding services and with a broader population would improve this work validity.

14.3 Discussion

By making a feature comparison, we can observe that Jarvis can implement many features that current conversational assistants lack, while simultaneously being more user-friendly than the available alternatives to IoT management (such as visual programming approaches). Overall Jarvis, or a similar solution, can ease and assist the process of configuring and managing IoT systems, significantly when the system in question increases in complexity, hindering the capability of end-users of understanding what is happening or which event lead to a specific outcome (and, possibly, correct the behavior). As more than one person in a typical household might use these systems, it becomes useful to understand behaviors that perhaps were defined by other members and to edit defined behaviors on-the-fly without needing to re-program the system traditionally.

[4]The basic definition of an entity is that of a list of possible values, and thus, for more coverage, it should contain several ways in which certain words can be expressed.

Although the number of functionalities that Jarvis provides and given the feasibility of such an approach for IoT configuration and management, we identify the following research directions that would improve the solution (or any similar approach):

Engaging in longer but fragmented conversations that would allow users to digest information at their own pace. This could be particularly useful when providing causality explanations since the user could iteratively explore more about the queried cause only if they wish to do so;

Support competing interactions as these can create contradictions and/or repetitions in the system. As the smart home system increases in complexity, originating by the increase of connected and interacting IoT devices (human-to-device and device-to-device) and the number of interacting people within the household, it becomes harder to avoid and mitigate overlapping rules or competing interactions. Adding specific capabilities to deal with more complex scenarios with multiple users and multiple interacting devices might reduce the complexity of dealing with such scenarios;

Support for priorities and roles as the number of individuals and parties that interact with the system increases, overriding rules can be introduced that might lead to both unintended consequences, and pose security and/or safety risk. Researching on how the system can identify which type of actions a user can request, as well as distinguishing between those that in tandem might lead to unforeseen consequences, does not seem trivial;

Exploring different causality-finding algorithms as these might provide more insightful answers. As presented, the current prototype always determines as the cause of an event the latest possible action that could have caused it; however, we believe that exploring alternatives such as heuristics that change the approach depending on the type of logged events might provide more useful answers to users;

Understanding implicit causality relations between different events. For instance, if there is a light sensor close to a light, Jarvis turning on that light could trigger a change on that sensor, which the current prototype of Jarvis would not understand as correlated events. If Jarvis were to have a more *semantic* understanding of the system, it could perceive events like these as being related, which could further improve its answers to causality queries;

Supporting addition or removal of devices to the system. Jarvis currently uses an already configured database of devices to understand the system it is managing. Adding the capability to add or remove devices to the system would make Jarvis even more useful, particularly in a scenario where it would be used by end-users in their own spaces.

Boolean operators support in user queries. For example, when defining event rules, it would be useful to use multiple conditions with Boolean ("and"/"or") operators. An example of this feature would be the query "Turn on the bedroom light if the motion sensor is

activated, and it is after 9 pm", where both conditions would have to be true to the action to be executed;

Privacy assurance most solutions, including Jarvis itself, depend on cloud-based NLP solutions to understand the user intents, which raises several concerns such as if the devices are always on (always listening), what is the history stored by the service providers (conversational logs) and how the data is managed (*e.g.*, third-party access) [Amm+19].

Being IoT one of the most common targets of conversational assistants *commands*, it becomes crucial to improve the user interaction with the devices by voice, mostly because existing solutions are limited, with the most only supporting *direct actions* [Amm+19].

14.4 Summary

In this chapter, we presented a conversational interface prototype able to carry several, and different, management tasks currently not supported by voice assistants, with capabilities that include: (1) delayed, periodic, and repeating actions, enabling users to perform queries such as *"turn on the light in 5 minutes"* and *"turn on the light every day at 8 am"*; (2) the usage of contextual awareness for more natural conversations, allowing interactions that last for multiple sentences and provide a more intuitive conversation, *e.g.*, *"what rules do I have defined for the living room light?"*; (3) event management that allows orchestration of multiples devices that might not necessarily know that each other exists, *e.g.*, *"turn on the light when the motion sensor is activated"*; and (4) causality queries, to improve the users' understanding on how the current system operates, *e.g.*, *"why did the light turn on?"*.

Causality queries, specifically, are of great relevance, given that they are not supported by either conversational or visual tools. These queries provide an advance in the level of conversational engagement with automated systems, therefore facilitating the management of smart spaces.

We conducted feasibility experiments with participants that were asked to perform specific tasks with our system. The overall high success rate shows the feasibility of our approach since the solution is intuitive enough to be used by people without significant technological knowledge. It also shows that most challenges lie in the natural language capabilities of the system, as it is hard to predict for any user queries that have the same intrinsic meaning. We thus conclude that incorporating recent *NLP* advances (that were beyond the scope of this work) would have a high impact in terms of making the system more flexible to the many ways (correct or incorrect) that users articulate the same intentions.

Some of these improvements could even be easily made by implementing adjustments to the configuration of the Dialogflow tool. As mentioned, *user intents* are defined in the tool via sample queries. Therefore, merely diversifying the set of sample queries for each user intent, which could already be done by analyzing the incorrect queries from our controlled experiments, could provide significant improvements to the system.

All the experiment participants were using Jarvis for the first time when we ran the experiment. As happens with many other kinds of products, each user's experience could benefit

from them getting to know the tool and getting more familiar with its features and capabilities. In other words, it is possible that repeated use of Jarvis would increase the user's familiarity and therefore reduce the occurrence of incorrect queries even further.

15 | Conclusions

The ubiquitous connectivity and computing vision is a close reality primarily due to widespread connectivity-enabled computing devices in everyday things, known as the Internet-of-Things.

However, developing robust IoT systems pushes state of the art and practice of both software and hardware engineers. These systems' inherent nature and characteristics and the associated challenges have been associated with a new software crisis like the one of circa 1968.

While this thesis does not attempt to create a *silver bullet* for IoT development, nor to provide a guidebook on how to avoid a *crisis*, we attempt to give insights on how to improve the current systems design, development, and evolution, towards building dependable IoT systems.

With this work, we contribute to the field of Software Engineering, specifically to the application domain of IoT, with (1) a pattern language for dependable IoT systems composed by a total of 34 patterns, along with corresponding proof of concept implementation of some of those as extensions for Node-RED, (2) novel approaches for dynamically allocating computing tasks in IoT networks to improve these systems' reliability, and (3) a set of experiments on how to improve — in terms of understanding and observability — the end-user interaction with such systems. A mix of validation approaches was used for different parts of this thesis according to which approach was most suitable for the hypothesis in question. As a direct or indirect result of this research, we have authored 29 papers for peer-reviewed venues (conferences and journals), as listed in Appendix A (p. 321).

15.1 Research Questions

Although IoT have been around in the last few years [Kev09], developing these ever-expanding systems [Tan18] is without challenges. Some have designed these challenges as the entryway for the next Software Crisis [Fit12]. Even if no crisis is imminent, there is a large consensus on the existing issues of the lifecycle of IoT systems [Al-+16; TM17; Smi17; Buj+18; Aly+19; MM21], and the recurrent reports of incidents and other problems resulting from their misoperation [ZA19; Neu20; Lan20a].

This work was driven by five main research questions directly related to our hypothesis with these issues in mind. We present our answer to each one of these questions in the following paragraphs.

RQ1 What are the unique characteristics of IoT systems that make them complex, and how does such complexity impact the end-user ability to configure their dependable systems?

As with every piece of software, in IoT systems, there are two kinds of complexity: essential and accidental [Bro86]. Regarding essential complexity, IoT, as a result of a merge of knowledge and practices of different fields, inherits most of the problems and limitations of the merged fields. Although the extensive list of essential complexity sources, we name a few that were most relevant to this work: (1) large-scale, (2) heterogeneity, (3) highly-dynamic networks, (4) end-user-centric, (5) real-world blending (*cf.* Section 2.1, p. 21). Regarding accidental complexity, different vendors and manufacturers have been rushing products to market (*i.e.*, time-to-market) while mostly disregarding best practices or standards, leading to an ever-growing technological fragmentation in which vendor-silos flourish. In addition, the recurrent *cloud-only* design has been one of the main drivers of dependability issues (*e.g.*, systems stop working without Internet connectivity).

An IoT end-user has a few solutions that allow the (end-)user to configure the system to their needs, *i.e.*, end-user development environments (*cf.* Section 3.2, p. 68). These solutions take several forms, including, but not limited to, visual programming, vendor-specific apps, or even voice assistants. These solutions leverage abstractions as a way to simplify the development process, but by doing so (1) limit what the end-user can do (or force the end-user to bypass the used abstractions [Spo04]) and (2) are arduous to use and understand as the complexity of the system increases. By posing such limitations, it becomes impracticable for the end-user to configure their systems as the number of devices, users, and automation required increases (*cf.* Chapter 5, p. 123). Thus, making such systems dependable appears as an even more significant barrier, given that end-users do not have the means to configure fallback or recovery automations — even if they wish to — using the available abstractions.

RQ2 Are there recurrent problems concerning the lifecycle of IoT systems, and what are the prevalent solutions that address them?

Guaranteeing the dependability of an IoT system is a multi-domain research venture that encompasses concerns from both hardware and software perspectives. Regarding this thesis, the focus is given on software and the recurrent problems in IoT systems which solutions can be defined and implemented in software (but faults can originate either in software or hardware components). We have identified a total of 34 patterns (problem-solution pairs) detailing recurring problems in IoT systems. Of those patterns, seven are considered *supporting patterns* (*cf.* Chapter 7, p. 151), since they address recurring problems on the design and evolution of those systems; 13 focus on problems of detecting errors on IoT systems during their operation (*cf.* Chapter 8, p. 159); and 14 present solutions to common situations on IoT system operation that either require the system to recover or, at least, to maintain its health (*cf.* Chapter 9, p. 171).

When the error detection patterns are combined with recovery and maintenance of health patterns in a system, it can behave autonomically, viz. self-heal, without (or with minimal) human intervention (*cf.* Chapter 6, p. 144).

RQ3 What can be improved concerning the IoT systems' dependability?

Several techniques and methodologies have been used to improve the systems' dependability (*e.g.*, fault-tolerance). While several of these strategies have already been adopted in the IoT domain by practitioners (RQ1), the adoption of mechanisms to distribute system load and avoid single-point-of-failure in IoT scope (as they are used in cloud computing) is only exploratory and with several pending issues. While there are several reasons for the early stages of development of such mechanisms, among them the high heterogeneity of the devices and runtimes, in this work, we provided evidence for the feasibility of having such mechanisms in the IoT scope.

By providing the system with the mechanisms dynamically allocate computational tasks (*i.e.*, serverless) while adapting to runtime constraints, we allow the system to respond to failures gracefully, *e.g.*, in case of Internet connection disruption, any service that can be run using local computational resources will continue to operate with minimal disruption (*cf.* Chapter 10, p. 184). Complementary, as the number of devices increases, it becomes challenging to manage all the available resources, automatically adapting to runtime conditions. In cloud computing, this challenge was addressed by using orchestrators, which, being aware of the available resources, can allocate the computational tasks across resources. In this work, we introduced the notion of orchestrator on top of a visual programming environment, *i.e.*, Node-RED (*cf.* Chapter 11, p. 199). The orchestrator enables users to program their visual *flows*; however, the computation of *nodes* of a given *flow* can happen in any available computational resource, thus reducing the reliance on Node-RED. This has the side-effect of improving the system dependability as computational tasks can run in any device with the required capabilities.

RQ4 How can the mechanisms identified in RQ2 be leveraged by the end-users of IoT systems?

Allowing end-users to configure their IoT systems has been a widely researched subject, with both academia and industry proposing different end-user development solutions, leveraging some abstraction to reduce the complexity of developing these systems (*cf.* Section 3.2, p. 68). Allowing an end-user to use the patterns detailed in Part II (p. 144) implies that the solution they are using (1) has the built-in mechanisms to support one or more strategies presented as solutions in the patterns and (2) leverages the same category of abstraction that the development solution already uses.

We picked Node-RED, a reference visual end-user development solution for IoT development, as a base solution for implementing the patterns. We successfully implemented 17 Node-RED *nodes*, corresponding to one or more strategies detailed as possible solutions on 19 different patterns (*cf.* Chapter 12, p. 218), thus showcasing the feasibility of adding self-healing behaviors to existing development solutions. Additional efforts were carried towards improved runtime adaptation, reducing the always-on system's dependency on Node-RED, modifying Node-RED to allocate computing tasks among available resources during runtime. Although the implementation of these strategies forced us to modify the internal functioning of Node-RED, its configuration by the end-user can still be done entirely in a visual fashion (*cf.* Chapter 11, p. 199).

RQ5 How can the end-user's ability to manage the IoT systems' lifecycle be improved without requiring specific expertise nor hindering the systems' dependability?

Aware of Node-RED's limitations in terms of runtime feedback to the end-user and ease of understanding the configured system at any given point in time, we enriched the visual abstractions used to improve the inspection of the system. The validation of these modifications with participants showcases an overall improvement of the development process with (1) a significant reduction of the number of failed attempts to deploy the systems and (2) overall reduction of development time (*cf.* Chapter 13, p. 246). Lastly, we attempt to improve the users' ability to understand the configured automations at a given time (*i.e.*, allowing to adjust the system as needed) with voice assistants, showcasing the feasibility of using such system to query the system and, in some cases, understand the causality between events (*cf.* Chapter 14, p. 258).

15.2 Hypothesis Revisited

The aforementioned Research Questions (RQs) drove this research intending to address our main hypothesis:

> **H:** *It is possible to enrich IoT-focused end-user development environments in such a way that the resulting systems have a higher dependability degree, with the lowest impact on the know-how of the (end-)users.*

The answers to those same research questions allow us to deconstruct and discuss how we have addressed the hypothesis:

It is possible to enrich IoT-focused end-user development environments… As IoT systems are mostly used by non-technical users, being these users the ones that configure the systems to their own needs (*e.g.*, creating automations, adding or removing devices and defining preferences), we picked end-user development solutions as the target of our research. Concretely, we base most of our work on Node-RED, a visual programming solution. Some considerations about the use of voice assistants are also contemplated.

…in such a way that the resulting systems have a higher dependability degree … By identifying recurrent problems of IoT systems, and proceeding to carry a systematization of available knowledge on how to tackle such problems, we defined a set of 34 patterns that can be used to improve the dependability of IoT systems. A subset of those patterns can be used in tandem to make the system self-heal. We also asserted the feasibility of using Node-RED as a visual orchestrator of the system, allowing end-users to leverage the computational resources available in a visual fashion and with autonomic adaptation to runtime conditions.

…with the lowest impact on the know-how of the (end-)users. Using Node-RED as a reference end-user development solution, we successfully implemented a subset of the patterns as extensions to Node-RED that allow users to configure self-healing behaviors, thus enabling them to enhance their systems' dependability. We also made some minimal

enhancements to the visual notations of Node-RED, improving the end-users capability to understand the running system, and asserted the feasibility of using voice assistants as a supporting tool to visual approaches to improve the users' understanding of the in-place automations and the causality of certain events.

The experimental counterparts of the contributions that sustain the presented statements provide empirical supporting evidence for the plausibility of the hypothesis. In assembling, the contributions can be used to improve the dependability of IoT systems while leveraging abstractions that do not compromise the (end-)user capability to configure, use, and evolve them.

15.3 Thesis Validation

As stated in the Chapter 5 (p. 132), a mixture of different validation methodologies was used in this work, given their suitability to the different contributions presented. As a consequence, different parts of this work present threats to the validity of specific contributions.

Nonetheless, we acknowledge a set of transversal concerns to this work that we consider as common threats to its validity, which are discussed in the following paragraphs.

One of the base concepts explored in this thesis is the one of autonomic computing, stating that IoT systems should have autonomic proprieties, viz. self-*. However, only self-healing was explored among the four key self-* proprieties. To have autonomic systems, the four proprieties– self-healing, self-configuration, self-optimization, and self-protection– must work in tandem. As an example, it is required for the end-user to configure their system proprieties and devices in a mostly manual form, *i.e.*, the system does not self-configure. Even if all the proprieties were fulfilled, some parts of the system could require manual intervention during its lifecycle, *e.g.*, replace a faulty hardware part.

Regarding the contributions around patterns, several arguments in favor of the implicit validation of patterns were presented, as the result of the systematization of widely available knowledge, describing solutions that are recurrently used for certain problems, given a set of constraints (*i.e.*, forces). However, we can only ensure that these patterns present a solution to which some developers converged but are not necessarily the best solution for a given problem, given its intricacies. We thrived on finding widely reported solutions (some of them based on strategies with empirical validation) and tried to encompass as many forces as they are crucial in IoT context. However, it is difficult to ensure that all possible drivers were correctly identified.

The system's adaptation should be transparent to the end-user, but there are parts of the proposed approach that require manual — and technical — intervention. Enabling the underlying end-user development solution (*i.e.*, Node-RED) to dynamically allocate computational tasks to available resources (*i.e.*, devices) depends on having a custom firmware running on these devices compatible with our orchestration mechanism. Having this requires to (1) have devices compatible with our firmware and (2) manually flash the devices with such firmware[1].

[1] Flashing a custom firmware can be as direct as using the existing device update features, or more troublesome, depending on using the device's hardware debug ports (*e.g.*, serial port).

As addressing this shortcoming as a whole falls out of the scope of this thesis, we ensure that by having a set of devices running our firmware, they can use this work's proposal out-of-the-box.

Different contributions were made targeting known shortcomings or issues with IoT systems, especially regarding these systems' dependability. Nonetheless, one of the goals of this thesis was to empower the end-user to leverage these contributions in their own systems to make them more dependable. While we focused on bringing these contributions to Node-RED, which is a visual programming solution, to reduce the users' need for specific technical expertise, we are aware that there is a knowledge barrier to developing systems with Node-RED that can limit our target audience. To mitigate this threat, we made improvements to Node-RED itself and further enhanced the interaction with IoT systems using voice assistants.

Finally, combining the contributions done in improving Node-RED (including the auxiliary use of voice assistants) with remaining contributions in IoT systems' dependability were only partially done. We consider this a threat as there can be limitations in the integration of the different contributions. We attempt to mitigate this by using Node-RED as the foundation to the remaining contributions; however, we did not implement (and, thus, validate) the solution as a whole.

15.4 Main Outcomes

This thesis contributes to the field of software engineering, and, in particular, to what regards Internet-of-Things systems development. We identify as the primary outcomes of this work (1) a comprehensive review of state of the art, (2) a pattern language for dependable IoT systems, (3) approaches for improving the dependability of IoT systems by both distribution of computing tasks and the addition of self-healing behaviors, and (4) enhancements to both visual programming solutions and voice assistants to improve the end-user ability to understand and evolve the running system. The contributions resulted in a total of 12 publications, and their abstracts are presented in Appendix A (p. 321). Most of these contributions have replication packages available, being presented in Appendix B (p. 337).

15.4.1 Review of the State of the Art

In this work, we revisited and summarized the core concepts behind IoT as well as other key concepts such as fault-tolerance in software systems and autonomic computing in Chapter 2 (p. 21). In Chapter 3 (p. 64), we proceed to identify what are the key challenges that IoT face during most of the parts of its lifecycle, *i.e.*, the challenges and open issues on the design, construction, and test of these systems. Additionally, given that both dependability (fault-tolerance) and autonomic computing are fundamental concepts of our approach, we delve into the literature of these fields, giving a focus on works within IoT scope. We acknowledge and discuss the recurrent problems for developing IoT systems, asked in the first part of RQ1, recognizing that there are key proprieties of IoT systems that make them not trivial to develop. The systematization of the knowledge resulting from this review, along with elicitation of pending challenges and open issues, lead to three conference publications [DFF18; Dia+18; Sil+21].

15.4.2 End-User Survey

As we consider end-users one of the main target audiences of this work, we consider it essential to understand how the end-users *use* their IoT systems, *i.e.*, smart homes. In Chapter 5 (p. 123) we present a study of a collection of 177 automation rules provided by end-users, given a *smart home*. This gave us solid information about how end-users use their systems, how they wish to automate them, and the granularity and complexity of such automations. We also found that there was an inherent intent of some users to define failsafe behaviors in their systems, showcasing both the lack of trust of 100% correct operation of the IoT systems and efforts to previse damage to their homes and wellbeing. A preliminary analysis of this survey resulted in one publication [Soa+21].

15.4.3 Patterns and Pattern Language

Having sufficient experience and knowledge about IoT systems and their nature, including the common architectures, tools, and practices, we noticed that there was a lack of the best practices to ensure the dependability of these systems leading us to RQ2. Given that fault-tolerance and reliability practices are common to both software and hardware engineering, we proceed to systematize this knowledge from an IoT perspective, given the IoT particular proprieties (*i.e.*, forces). This resulted in a total of 34 patterns, which corresponded to 3 different publications [Ram+17; DFS19; Dia+20a].

15.4.4 Autonomic and Dependable IoT

The human unmanageability of IoT systems has led researchers to identify IoT as the prime example of a system that would favor the application of autonomic computing. Allowing a system to self-adapt depends on the system implementation itself and the provided features, leading to RQ3. As an example, for a system to be capable of distributing tasks among available resources, it must be capable of decomposing the *programs* in computing tasks and orchestrating the system as needed (during runtime), taking into account capabilities and constraints. As a base work on this research venue, we study the possibility of dynamically allocating computational tasks using by defining these tasks as lambda functions (serverless), which could be allocated to different resources, both on local network or cloud, depending on runtime constraints. Given this groundwork, we move to bring such capabilities to Node-RED, allowing the visual *flows* to be decomposed into computational tasks that could be dynamically allocated to available edge devices. Finally, in other to allow the configuration of self-healing behaviors using the visual *flows* of Node-RED, we proceed to implement a total of 17 Node-RED *nodes*, which correspond to a subset of the patterns part of the self-healing pattern language. These contributions resulted in 4 publications [PDS18; Dia+20b; Sil+20; DRF21].

15.4.5 End-User Development

Empowering end-users to develop systems more tailored to their needs while ensuring the underlying system dependability depends on proper development tools and interfaces, leading

us to RQ4. Aware of the limitations of both available visual programming solutions and voice assistants for developing IoT systems, in Chapter 13 (p. 246) we improved Node-RED with new visual aids and controls that provide both visual feedback for the end-user and visual debugging mechanisms. Additionally, in Chapter 14 (p. 258) we study the use of voice assistants as a supporting tool for IoT development, introducing voice features that allow the end-user to understand the causality of certain events as well as what is the system in-place rules at a given time (allowing for fine-tuning them). These contributions resulted in a total of 2 conference publications [Tor+20; DLF20], and one journal article [LDF21].

15.5 Future Work

While the contributions presented as part of this thesis attempt to advance the current body of knowledge regarding software engineering and its application on Internet-of-Things systems, they also present several shortcomings, identify open issues in the state-of-the-art, and open new research directions. We present a few of these closely related to the work pursued in the following paragraphs.

Concerning the contributions to the patterns' literature (*cf.* Part II, p. 144), and their applicability in the IoT domain, we recognize that it would be valuable to assert their adoption and relevance in the community by distributing a survey among IoT practitioners and developers. This methodology has already been used in other research fields — *e.g.*, cloud computing [SFC21] — to empirically study the relevance of certain patterns.

Regarding the contributions to dynamically distributing and orchestrating computing tasks in IoT systems (*cf.* Chapter 10, p. 184, and Chapter 11, p. 199), we see that this work did not address several open issues. While we followed a greedy assignment process, this can have several shortcomings as the system complexity grows (*e.g.*, increasing number and heterogeneity of devices, especially with different memory constraints), and even become unsuitable (*e.g.*, not being able to satisfy the system constraints); thus optimization techniques such as the ones proposed by Skarlat *et al.* [Ska+17] could improve the current approach. Similarly, while *exploration vs. exploitation* techniques were primarily explored, their suitability for large-scale systems was not further researched. Lastly, while the system keeps re-orchestrating as changes happen (*e.g.*, devices go offline/online), constantly changing the system allocating reduces its overall availability, thus the question remains on how to keep the system balanced in terms of distribution (reducing the load in each device) while minimizing the set of changes (improving availability).

While the firmware that runs on edge devices was a requirement for carrying out this research (*cf.* Chapter 11, p. 199) allowing to allocate computational tasks on demand, there was little to no focus on improving the firmware itself. In this scope, we observe several possible improvements that can be done which can improve memory consumption and reduce the delay — *i.e.*, time from receiving to execute a task — resulting from a re-orchestration. This includes the use of alternatives beyond MicroPython, including RTOS or WebAssembly [ZB21].

Regarding the application of autonomic computing in IoT, and, more specifically, self-healing, to improve the system dependability, there is a considerable amount of patterns (as described in Chapter 9, p. 171) that were not implemented, thus are only supported by existing

literature. We recognize that most of these patterns would benefit from a reference implementation (*e.g.*, as Node-RED *nodes*). Focusing on SHEN, future work includes (1) the extension of the SHEN palette with more runtime verification and self-healing mechanisms, (2) dealing with concurrent inputs that can lead to unexpected states (*e.g.*, the system decides to turn on the lights, and the user manually turns them off), which may result in false assertions by the runtime verification mechanisms, (3) study what are the reasonable operational states that the system should converge to in the case of failure (*e.g.*, if the system has to decide between shutting down the smoke alarm or the surveillance system, which one should take prevalence?), and (4) case studies over various degrees of systems complexity and in different contexts and scales. We also only consider reactive behaviors as part of this work — trying to mitigate errors as they occur — but the predictive mechanism can play a crucial role (*cf.* Predictive Device Monitor), including the usage of machine learning and other artificial intelligence mechanisms to understand system operational patterns.

We also consider that to fulfill the view of autonomic computing supporting and articulating with other *self-** aspects is crucial; this includes *self-protection*, *self-optimization*, and *self-configuration* [GC03]. As an example, we acknowledge that auto-discovery and configuration of new devices in the system can improve the system's dependability, *e.g.*, a new mobile device can be used as a redundant sensing node while it is connected to the system network.

Lastly, while the end-user interaction with the IoT system is one of the aspects that most impacts the usage and value of these systems to the end-user (human-in-the-loop), it is also one of the aspects most disregarded by the practitioners — as it is confirmed by the fragmentation of IoT solutions and vertical silos created by their controlling applications (*cf.* Section 3.2, p. 68). We recognize that the IoT development environments can be improved (while we focused on Node-RED, the issues identified are shared among several IoT development solutions), bringing concepts common to the development solutions targeting other kinds of systems (*e.g.*, linters, static analyzers, and types system)[2]. Further, while visual notations had a notorious focus during this work, we have to ponder that other interaction mechanisms — including voice-based ones — can have an important role either as a standalone development tool or as a development supporting tool.

Considering the contributions of this work as a whole, we identify the following some pending — and mostly unaddressed — questions that can guide future research:

- How to put together all the different contributions into an all-in-one solution for IoT development? What is the scalability of such a solution, and does it indeed perform better than existing solutions?

- How to make the proposed solution more transparent to the end-user? This is a concern since that there are parts of the current approach that require some technical knowledge (*e.g.*, flash a given device with modified firmware);

- How to automatically suggest and implement modifications to the running system to improve the system's overall dependability? Should the end-user have any role in this process or be transparent?

[2] Node-RED version 2.0 introduced some of these features [OLe20].

- As the system evolves, new dependability threats can appear. How can the system continuously check for changes and assert if those changes constitute a threat, automatically identifying and addressing them (or informing the end-user about them)?

A cross-cutting question that remains is on whether the end-user should be able to perform modifications to the system that compromise its dependability or be limited by design? If so, to which degree should the system *limit* these modifications or recommend *not performing* them? As these systems embrace autonomic principles, we acknowledge that a balance between autonomic behaviors and manual adjustments is required and should be further studied.

15.6 Epilogue

As the *scope, complexity, and pervasiveness of computer-based and controlled systems continue to increase dramatically* [Pul01], Internet-of-Things appears as the prime example of such. With the promise of improving the overall quality-of-life and reshaping industry (viz. Industry 4.0), IoT is here to stay, even with its shortcomings and lingering issues, permeating through building, cities, manufacturing floors, and, even, human bodies (*e.g.*, wearables, pacemakers and other implants[3]). While the dependability of these systems is crucial (*i.e.*, in some cases categorized as safety-critical), today IoT systems are still far behind the best practices that have been used in other kinds of systems for long (*e.g.*, aerospace industry [Tor00]).

During the last four years, I have put my best efforts into understanding what has been done in this regard, deep-diving into the software (and hardware) state-of-the-art, searching for both open issues and old solutions for new problems. My research endeavors lead me as far as the first computers, telephony, and the creation of the Internet. Among the realizations of such endeavor, I took note of a few non-technical ones: (1) we, as a research community, on several occasions, fail to learn from the past, and keep *reinventing the wheel*; (2) the constant thrive for novelty foments the reuse of old ideas under different taxonomy, even without new contributions *per se*; and (3) different research fields converge to similar solutions, but collaborations (or cross-mentions) are scarce.

With this work, we attempted to mitigate these symptoms as well as we could. Instead of using only simulators and emulators to validate our ideas and contributions, we devoted time to physical build a laboratory with *do-it-yourself* devices, gathering knowledge about both the software and hardware concerning their functioning, limitations, and performance, carried several experiments with different radio[4] and wired protocols for data exchange and reverse engineered closed-source devices to understand its inner works. We expanded our literature research well beyond software engineering, reading about electronics and others while attempting to debunk re-branded concepts or dubious taxonomies. These endeavors did not add value to the thesis *per se* but allowed to have a better understanding of what are the concrete problems with IoT, and what contributions could be made to advance the current state-of-the-art of the

[3] As an example consider the CE approved Bluetooth-enabled biventricular implantable cardiac defibrillator (ICD) [Abb19].

[4] Including the successful reception of Slow-Scan Television (SSTV) images from the International Space Station (ISS) on the occasion of the 20 years of amateur radio operation on the ISS (December 24-31, 2020).

software that runs these systems which could improve their dependability while leveraging the wide-range of already available knowledge (*i.e.*, *standing on the shoulders of giants*).

Nevertheless, in practical terms, to have a thesis, concrete contributions must be made, which, in academia, typically correspond to publications. But where to publish? Do we strive for quantity or quality? Looking back and pondering, in the first years, the quantity was vital, with quality being a minimum requirement but not the main driver (at the bare minimum only CORE C [5] conference venues were considered). In the last few years, there was a shift to strive for quality (CORE A/A*), and with it came the rejections. Some of those rejections were indeed right and fair, and most of the time contained helpful feedback for our work. But, on other occasions, it was not. The key here is to keep improving, keep trying, and, eventually, get published.

At last, this is my thesis. An (extensive) summary of more than four years of research that encompassed so many other contributions that are not in the scope of this document (*cf.* Appendix A, p. 321). Completing a Ph.D. is an arduous task. Neither all ideas are good ideas, nor do all contributions have value. Making *science*, and understanding what makes for a good *hypothesis* and how to validate one properly, is not trivial. This is my best attempt, and to those who are reading this, either as inspiration or as state-of-the-art, I share some Internet wisdom[6]: *"A good dissertation is a done dissertation. A great dissertation is a published dissertation. A perfect dissertation is neither."* ■

[5] Computing Research and Education Association of Australasia (CORE) conference ranking, available at http://portal.core.edu.au/conf-ranks/.

[6] Original author unknown, shared on The Ph.D. Lab blog by Kristin Werner, https://www.thephdlab.com/.

References

[AA19] Mehmet S. Aktas and Merve Astekin. “Provenance aware runtime verification of things for self-healing Internet of Things applications”. In: *Concurrency Computation* 31.3 (2019), pp. 1–9 (see pp. 10, 55, 120, 121).

[AB19] "Topplab AB". *Noodl Reference Documentation*. [Online; accessed 2019]. "Topplab AB", 2019 (see p. 76).

[Aba19] Fardin Abad. “Safety and Security of Cyber-physical Systems”. PhD thesis. University of Illinois, 2019 (see p. 178).

[Abb19] Abbott Media. *Abbott Introduces Next-Generation Heart Rhythm Management Devices in Europe, Featuring State-of-the-Art Patient App and Bluetooth Connectivity*. 2019 (see p. 287).

[Abd+17] Fardin Abdi et al. “Application and system-level software fault tolerance through full system restarts”. In: *2017 ACM/IEEE 8th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE. 2017, pp. 197–206 (see p. 178).

[ACN10] S. Alam, M. M. R. Chowdhury, and J. Noll. “SenaaS: An event-driven sensor virtualization approach for Internet of Things cloud”. In: *2010 IEEE International Conference on Networked Embedded Systems for Enterprise Applications*. Nov. 2010, pp. 1–6 (see p. 41).

[Ada+98] Michael Adams et al. “Fault-tolerant telecommunication system patterns”. In: *The Patterns Handbook: Techniques, Strategies, and Applications, Cambridge University Press, New York* (1998), pp. 189–202 (see pp. 166, 179).

[Ade+17] Ferran Adelantado et al. “Understanding the Limits of LoRaWAN”. In: *IEEE Communications Magazine* 55.9 (2017), pp. 34–40. arXiv: 1607.08011 (see p. 32).

[Adj+15] Cedric Adjih et al. “FIT IoT-LAB: A large scale open experimental IoT testbed”. In: *IEEE 2nd World Forum on Internet of Things*. IEEE. IEEE, 2015, pp. 459–464 (see p. 98).

[ADT13] Arjun P. Athreya, Bruce DeBruhl, and Patrick Tague. “Designing for self-configuration and self-adaptation in the Internet of Things”. In: *Proceedings of the 9th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing, COLLABORATECOM 2013* (2013), pp. 585–592 (see pp. 53, 120, 121).

[Ady+04] Atul Adya et al. “Architecture and Techniques for Diagnosing Faults in IEEE 802.11 Infrastructure Networks”. In: *Proceedings of the 10th Annual International Conference on Mobile Computing and Networking*. MobiCom ’04. New York, NY, USA: Association for Computing Machinery, 2004, pp. 30–44 (see p. 163).

[Aga+18] Ioannis Agadakos et al. “Butterfly Effect: Causality from Chaos in the IoT”. In: *International Workshop on Security and Privacy for the Internet-of-Things*. Apr. 2018 (see p. 268).

[Agu+19] Ademar Aguiar et al. “Live Software Development: Tightening the Feedback Loops”. In: *Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming*. Programming ’19. Genova, Italy: ACM, 2019, 22:1–22:6 (see pp. 61, 95, 333).

[AH15a] Qazi Mamoon Ashraf and Mohamed Hadi Habaebi. “Autonomic schemes for threat mitigation in Internet of Things”. In: *Journal of Network and Computer Applications* 49 (2015), pp. 112–127 (see p. 119).

[AH15b] Qazi Mamoon Ashraf and Mohamed Hadi Habaebi. “Introducing autonomy in internet of things”. In: *14th International Conference on Applied Computer and Applied Computational Science (ACACOS’15)*. 2015 (see pp. 120, 137).

[Ahm+16] Abbas Ahmad et al. “Model-Based Testing as a Service for IoT Platforms”. In: *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications: 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part II*. Ed. by Tiziana Margaria and Bernhard Steffen. Cham: Springer International Publishing, 2016, pp. 727–742 (see p. 100).

[Ahm+21] Mohammad M Ahmadpanah et al. “SandTrap: Securing JavaScript-driven Trigger-Action Platforms”. In: *USENIX Security Symposium (USENIX Security 2021)*. 2021 (see p. 90).

[AIS77] C Alexander, S Ishikawa, and M Silverstein. *A Pattern Language*. Oxford University Press, 1977 (see pp. 8, 57, 144).

[AK14] M. Abomhara and G. M. Køien. “Security and privacy in the Internet of Things: Current status and open issues”. In: *2014 International Conference on Privacy and Security in Mobile Systems (PRISMS)*. IEEE, May 2014, pp. 1–8 (see p. 107).

[Aks+18] H. Aksu et al. “Advertising in the IoT Era: Vision and Challenges”. In: *IEEE Communications Magazine* 56.11 (Nov. 2018), pp. 138–144 (see p. 25).

[Aky+02] I.F. Akyildiz et al. “Wireless sensor networks: a survey”. In: *Computer Networks* 38.4 (2002), pp. 393–422 (see pp. 29, 36, 37).

[Al-+15] Ala Al-fuqaha et al. “Internet of Things : A Survey on Enabling”. In: *IEEE Communications Surveys Tutorials* 17.4 (2015), pp. 2347–2376 (see p. 35).

[Al-+16] S. A. Al-Qaseemi et al. “IoT architecture challenges and issues: Lack of standardization”. In: *2016 Future Technologies Conference (FTC)*. Dec. 2016, pp. 731–738 (see p. 278).

[Al-+17] S. Al-Sarawi et al. “Internet of Things (IoT) communication protocols: Review”. In: *2017 8th International Conference on Information Technology (ICIT)*. May 2017, pp. 685–690 (see p. 37).

[Al-+18] Alauddin Al-Omary et al. “Survey of hardware-based security support for IoT/CPS systems”. In: *KnE Engineering* (2018), pp. 52–70 (see p. 165).

[All20] "Connectivity Standards Alliance (formerly Zigbee Alliance)". *Matter (formerly Project CHIP)*. [Online; accessed 2021]. 2020 (see pp. 35, 37).

[ALR01] Algirdas Avizienis, Jean-Claude Laprie, and Brian Randell. “Fundamental Concepts of Dependability”. In: *Technical Report Seriesuniversity of Newcastle Upon Tyne Computing Science* 1145.010028 (2001), pp. 7–12 (see pp. 9, 46, 48, 49, 62, 242).

[Aly+19] M. Aly et al. “Is Fragmentation a Threat to the Success of the Internet of Things?” In: *IEEE Internet of Things Journal* 6.1 (Feb. 2019), pp. 472–487 (see pp. 136, 137, 243, 278).

[Ama+19] Diogo Amaral et al. “Live Software Development Environment for Java using Virtual Reality”. In: *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering — Volume 1: ENASE*. INSTICC. SciTePress, 2019, pp. 37–46 (see p. 333).

[Ama+20] Diogo Amaral et al. “Live Software Development Environment using Virtual Reality: a Prototype and Experiment”. In: *Communications in Computer and Information Science* 1172 (2020) (see p. 331).

[Amm+19] Tawfiq Ammari et al. “Music, Search, and IoT: How People (Really) Use Voice Assistants”. In: *ACM Transactions in Computer-Human Interaction* 26.3 (Apr. 2019) (see pp. 84, 85, 123, 124, 130, 131, 259, 260, 265, 269, 276).

[Amo+12] Mehdi Amoui et al. “Achieving Dynamic Adaptation via Management and Interpretation of Runtime Models”. In: *Journal of Systems and Software* 85.12 (Dec. 2012), pp. 2720–2737 (see p. 61).

[Ams+12] Marcel van Amstel et al. *Automation in Warehouse Development*. Springer, London, 2012, pp. 45–58 (see p. 59).

[Anc+18] Davide Ancona et al. “Towards runtime monitoring of node.js and its application to the internet of things”. In: *Electronic Proceedings in Theoretical Computer Science, EPTCS*. Vol. 264. 2018, pp. 27–42 (see p. 92).

[And+14] J. G. Andrews et al. "What Will 5G Be?" In: *IEEE Journal on Selected Areas in Communications* 32.6 (June 2014), pp. 1065–1082 (see p. 25).

[And+19] Pilar Andrés-Maldonado et al. "NB-IoT M2M Communications in 5G Cellular Networks". PhD thesis. Universidad de Granada, 2019 (see p. 33).

[And+21] Rossana M.C. Andrade et al. "Multifaceted infrastructure for self-adaptive IoT systems". In: *Information and Software Technology* 132.December 2020 (2021), p. 106505 (see pp. 14, 137).

[And11] Marc Andreessen. "Why software is eating the world". In: *Wall Street Journal* 20.2011 (2011), p. C2 (see p. 29).

[Ang15] Rafael Angarita. "Responsible objects: Towards self-healing internet of things applications". In: *Proceedings - IEEE International Conference on Autonomic Computing, ICAC 2015* (2015), pp. 307–312 (see pp. 10, 120, 121, 137).

[Ant+17] Manos Antonakakis et al. "Understanding the mirai botnet". In: *26th USENIX Security Symposium (USENIX Security 17)*. 2017, pp. 1093–1110 (see p. 107).

[AOK19] Liz Allen, Alison O'Connell, and Veronique Kiermer. "How can we ensure visibility and diversity in research contributions? How the Contributor Role Taxonomy (CRediT) is helping the shift from authorship to contributorship". In: *Learned Publishing* 32.1 (2019), pp. 71–74. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/leap.1210 (see p. 19).

[Ard19a] "ArduBlock". *Ardublock: A Graphical Programming Language for Arduino*. [Online; accessed 2019]. "ArduBlock", 2019 (see p. 75).

[Ard19b] "Arduino". *Arduino*. [Online; accessed 2019]. "Arduino Open Source Hardware", 2019 (see pp. 27, 70).

[AS17] Massimo Alioto and Mohsen Shahghasemi. "The Internet of Things on its edge: Trends toward its tipping point". In: *IEEE Consumer Electronics Magazine* 7.1 (2017), pp. 77–87 (see p. 30).

[ASD16] Fahed Alkhabbas, Romina Spalazzese, and Paul Davidsson. "IoT-based Systems of Systems". In: *Proceedings of the 2nd edition of Swedish Workshop on the Engineering of Systems of Systems (SWESOS 2016)* (2016) (see pp. 9, 22).

[ASD17] F. Alkhabbas, R. Spalazzese, and P. Davidsson. "Architecting Emergent Configurations in the Internet of Things". In: *2017 IEEE International Conference on Software Architecture (ICSA)*. Apr. 2017, pp. 221–224 (see p. 73).

[Aus+18] Jonas Austerjost et al. "Introducing a virtual assistant to the lab: A voice user interface for the intuitive control of laboratory instruments". In: *SLAS TECHNOLOGY: Translating Life Sciences Innovation* 23.5 (2018), pp. 476–482 (see p. 84).

[Avi+04] Algirdas Avižienis et al. "Basic concepts and taxonomy of dependable and secure computing". In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004), pp. 11–33 (see pp. 9, 46–51, 173, 176).

[Avn19] Avnet, Inc. *Don't ignore analog — it's the secret sauce of IoT*. 2019 (see p. 3).

[AWS21] AWS. *Architecture Best Practices for IoT*. [Online; accessed 2021]. 2021 (see p. 43).

[Axe07] Jan Axelson. *Serial Port Complete: The Developer's Guide*. Lakeview Research LLC, 2007 (see p. 157).

[Azu21] Azure. *Azure IoT reference architecture*. [Online; accessed 2021]. 2021 (see p. 43).

[Bai+11] Xiaoying Bai et al. "Cloud testing tools". In: *Service Oriented System Engineering (SOSE), 2011 IEEE 6th International Symposium on*. IEEE. IEEE, 2011, pp. 1–12 (see p. 97).

[Baj17] M. Bajer. "Building an IoT Data Hub with Elasticsearch, Logstash and Kibana". In: *2017 5th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*. Aug. 2017, pp. 63–68 (see p. 91).

[BBF09] Gordon Blair, Nelly Bencomo, and Robert France. "Models@run.time". In: *Computer* 42.10 (Oct. 2009), pp. 22–27 (see p. 61).

[BBS18] Iulia Bastys, Musard Balliu, and Andrei Sabelfeld. "If this then what?: Controlling flows in IoT apps". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2018, pp. 1102–1119 (see p. 77).

[BC13] Sourangsu Banerji and Rahul Singha Chowdhury. "Wi-Fi & WiMAX: A Comparative Study". In: *arXiv preprint arXiv:1302.2247* (2013) (see p. 33).

[BC15] Ranjit Bawa and Rick Clarck. *Software-defined everything*. Tech. rep. Deloitte Consulting LLP, 2015 (see pp. 3, 29).

[BCC18] Nayeon Bak, Byeong Mo Chang, and Kwanghoon Choi. "Smart Block: A Visual Programming Environment for SmartThings". In: *Proceedings - International Computer Software and Applications Conference*. Vol. 2. 2018, pp. 32–37 (see p. 199).

[BD04] Marat Boshernitsan and Michael Sean Downes. *Visual programming languages: A survey*. Tech. rep. December. University of California, Berkeley, 2004 (see p. 74).

[BD16] Rajkumar Buyya and Amir Vahid Dastjerdi. *Internet of Things: Principles and Paradigms*. Elsevier, 2016 (see pp. 4, 22–25, 41, 42, 103–106, 117).

[BE15] B. Bagula and Zenville Erasmus. "Iot emulation with cooja". In: *ICTP-IoT Workshop*. 2015 (see p. 99).

[Bec+19] Christian Becker et al. "Pervasive computing middleware: current trends and emerging challenges". In: *CCF Transactions on Pervasive Computing and Interaction* (2019), pp. 1–14 (see p. 42).

[Bei03] Boris Beizer. *Software testing techniques*. Dreamtech Press, 2003 (see p. 62).

[Bei18] Noud Beijer. "Continuous Delivery in IoT Environments". In: *International Conference on Agile Software Development*. Ed. by Agile Alliance. Agile Alliance, 2018, pp. 1–7 (see pp. 67, 151).

[Bel+03] Mariano Belaunde et al. *MDA Guide Version 1.0. 1*. Tech. rep. Object Management Group, Inc, 2003 (see p. 59).

[Bel17] Charles Bell. *MicroPython for the Internet of Things*. Springer, 2017 (see pp. 28, 201).

[Ben18] Antonio Carlos Bento. "IoT: NodeMCU 12e X Arduino Uno, Results of an experimental and comparative survey". In: *International Journal* 6.1 (2018) (see pp. 27, 28).

[Ber+13] Arun Kishore Ramakrishnan Berbers et al. "Learning Deployment Trade-offs for Self-Optimization of Internet of Things Applications". In: *10th International Conference on Autonomic Computing ({ICAC} 13)*. 2013, pp. 213–224 (see pp. 118, 119).

[BG10] Luciano Baresi and Carlo Ghezzi. "The Disappearing Boundary between Development-Time and Run-Time". In: *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*. FoSER '10. Santa Fe, New Mexico, USA: Association for Computing Machinery, 2010, pp. 17–22 (see p. 123).

[BGJ17] Terrell R. Bennett, Nicholas Gans, and Roozbeh Jafari. "Data-Driven Synchronization for Internet-of-Things Systems". In: *ACM Transactions in Embedded Computing Systems* 16.3 (Apr. 2017) (see p. 167).

[BGT16] B. Butzin, F. Golatowski, and D. Timmermann. "Microservices approach for the internet of things". In: *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*. Sept. 2016, pp. 1–6 (see p. 41).

[Bhe+21] Deva Sai Kumar Bheesetti et al. "A Complete Home Automation Strategy Using Internet of Things". In: *ICCCE 2020*. Ed. by Amit Kumar and Stefan Mozar. Singapore: Springer Singapore, 2021, pp. 363–373 (see p. 85).

[Bis19] M. Bishop. *Hypertext Transfer Protocol Version 3 (HTTP/3)*. 2019 (see p. 34).

[BL12a] M. Blackstock and R. Lea. "IoT mashups with the WoTKit". In: *2012 3rd IEEE International Conference on the Internet of Things*. Oct. 2012, pp. 159–166 (see pp. 77, 78).

[BL12b] Michael Blackstock and Rodger Lea. "WoTKit: A Lightweight Toolkit for the Web of Things". In: *Proceedings of the Third International Workshop on the Web of Things*. WOT '12. New York, NY, USA: ACM, 2012, 3:1–3:6 (see p. 78).

[BL14] Michael Blackstock and Rodger Lea. "Toward a Distributed Data Flow Platform for the Web of Things (Distributed Node-RED)". In: *Proceedings of the 5th International Workshop on Web of Things - WoT '14*. 2014, pp. 34–39 (see pp. 79, 88, 94).

[BL16] Michael Blackstock and Rodger Lea. "FRED: A Hosted Data Flow Platform for the IoT". In: *Proceedings of the 1st International Workshop on Mashups of Things and APIs*. MOTA '16. New York, NY, USA: ACM, 2016, 2:1–2:5 (see pp. 78, 87).

[Bla+10] Michael Blackstock et al. "MAGIC Broker 2: An open and extensible platform for the Internet of Things". In: *2010 Internet of Things (IOT)*. IEEE. 2010, pp. 1–8 (see p. 42).

[Blo+18] Gedare Bloom et al. "Design patterns for the industrial Internet of Things". In: *2018 14th IEEE International Workshop on Factory Communication Systems (WFCS)*. IEEE. IEEE, 2018, pp. 1–10 (see pp. 66, 120).

[BMR96] F. Bushmann, R. Meunier, and H. Rohnert. "Pattern-oriented software architecture: A system of patterns". In: *John Wiley&Sons* 1 (1996), p. 476 (see p. 57).

[BMS16] Bharat Bohora, Sunil Maharjan, and Bibek Raj Shrestha. "IoT Based Smart Home Using Blynk Framework". In: *Zerone Scholar* 1.1 (2016), pp. 26–30 (see p. 77).

[BMV17] S. Bera, S. Misra, and A. V. Vasilakos. "Software-Defined Networking for Internet of Things: A Survey". In: *IEEE Internet of Things Journal* 4.6 (Dec. 2017), pp. 1994–2008 (see p. 37).

[Boa+16] Carlo Alberto Boano et al. "Dependability for the Internet of Things: From dependable networking in harsh environments to a holistic view on dependability". English. In: *Elektrotechnik und Informationstechnik* 133.7 (Nov. 2016), pp. 304–309 (see p. 161).

[Bol20] Tiago Boldt Sousa. "Engineering Software for the Cloud: A Pattern Language". PhD thesis. Faculty of Engineering, University of Porto, 2020 (see p. 176).

[Bon08] Borzoo Bonakdarpour. "Challenges in transformation of existing real-time embedded systems to cyber-physical systems". In: *ACM SIGBED Review* 5.1 (2008), pp. 1–2 (see p. 1).

[Bor+17] Borja Bordel et al. "Cyber–physical systems: Extending pervasive sensing from control theory to the Internet of Things". In: *Pervasive and Mobile Computing* 40 (2017), pp. 156–184 (see pp. 22, 120).

[Bos+19] Stig Bosmans et al. "Testing IoT systems using a hybrid simulation based testing approach". In: *Computing* 101.7 (2019), pp. 857–872 (see p. 101).

[Bou16] Ahcène Bounceur. "CupCarbon: A New Platform for Designing and Simulating Smart-City and IoT Wireless Sensor Networks (SCI-WSN)". In: *Proceedings of the International Conference on Internet of Things and Cloud Computing* (2016), 1:1–1:1 (see p. 100).

[Bra+17] Dave Braines et al. "Conversational homes: a uniform natural language approach for collaboration among humans and devices". In: *International Journal on Advances in Intelligent Systems* 10.3/4 (2017), pp. 223–237 (see pp. 84, 268).

[BRH16] W. Bulten, A. C. V. Rossum, and W. F. G. Haselager. "Human SLAM, Indoor Localisation of Devices and Users". In: *2016 IEEE First International Conference on Internet-of-Things Design and Implementation (IoTDI)*. Apr. 2016, pp. 211–222 (see pp. 220, 340).

[BRH20] Tina Beranic, Patrik Rek, and Marjan Heričko. "Adoption and Usability of Low-Code/No-Code Development Tools". In: *Central European Conference on Information and Intelligent Systems*. Faculty of Organization and Informatics Varazdin. 2020, pp. 97–103 (see p. 14).

[Bri+17] Julia Brich et al. "Exploring End User Programming Needs in Home Automation". In: *ACM Trans. Comput.-Hum. Interact.* 24.2 (Apr. 2017) (see p. 260).

[Brö+17] Arne Bröring et al. "Enabling IoT ecosystems through platform interoperability". In: *IEEE software* 34.1 (2017), pp. 54–61 (see p. 70).

[Bro86] Frederick Phillips Jr. Brooks. "No silver bullet - Essence and Accidents of Software Engineering". In: *Proceedings of the IFIP Tenth World Computing Conference* (1986), pp. 1069–1076 (see pp. 8, 14, 132, 279).

[BS01] Jennifer Bray and Charles F Sturman. *Bluetooth 1.1: connect without cables*. Pearson Education, 2001 (see p. 31).

[BT18] Brendan Burns and Craig Tracey. *Managing Kubernetes: operating Kubernetes clusters in the real world*. O'Reilly Media, 2018 (see p. 203).

[BUA17] Marco Brambilla, Eric Umuhoza, and Roberto Acerbis. "Model-driven development of user interfaces for IoT systems via domain-specific components and patterns". In: *Journal of Internet Services and Applications* 8.1 (2017), pp. 1–21 (see p. 66).

[Buj+18] Armir Bujari et al. "Standards, Security and Business Models: Key Challenges for the IoT Scenario". In: *Mobile Networks and Applications* 23.1 (Feb. 2018), pp. 147–154 (see pp. 45, 278).

[Bur17] Miroslav Bures. "Framework for Integration Testing of IoT Solutions". In: *2017 International Conference on Computational Science and Computational Intelligence (CSCI)*. IEEE. 2017, pp. 1838–1839 (see p. 101).

[BV15] Barbara Rita Barricelli and Stefano Valtolina. "Designing for End-User Development in the Internet of Things". In: *End-User Development: 5th International Symposium, IS-EUD 2015, Madrid, Spain, May 26-29, 2015. Proceedings*. Ed. by Paloma Díaz et al. Cham: Springer International Publishing, 2015, pp. 9–24 (see p. 96).

[BV19] "Zenodys B.V." *Zenodys: Internet of things development platform*. [Online; accessed 2019]. "Zenodys B.V.", 2019 (see p. 76).

[BW16] Kyle Brown and Bobby Woolf. "Implementation Patterns for Microservices Architectures". In: *Proceedings of the 22th Conference on Pattern Languages of Programs*. 2016 (see p. 153).

[BWK13] Roland Bijvank, Wiebe Wiersema, and Christian Köppe. "Software Architecture Patterns for System Administration Support". In: *20th Conference on Pattern Languages of Programs*. PLoP '13. USA: The Hillside Group, 2013, 1:1–1:14 (see p. 153).

[CA78] Liming Chen and Algirdas Avizienis. "N-version programming: A fault-tolerance approach to reliability of software operation". In: *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8)*. Vol. 1. 1978, pp. 3–9 (see p. 166).

[Cal+11] Rodrigo N Calheiros et al. "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms". In: *Software: Practice and experience* 41.1 (2011), pp. 23–50 (see p. 100).

[Cas+11] Angelo P Castellani et al. "Web Services for the Internet of Things through CoAP and EXI". In: *2011 IEEE International Conference on Communications Workshops (ICC)*. IEEE. 2011, pp. 1–6 (see p. 42).

[CBK09] Varun Chandola, Arindam Banerjee, and Vipin Kumar. "Anomaly detection: A survey". In: *ACM computing surveys (CSUR)* 41.3 (2009), pp. 1–58 (see p. 162).

[CC20] D.R. Cacciagrano and R. Culmone. "IRON: Reliable domain specific language for programming IoT devices". In: *Internet of Things* 9 (2020), p. 100020 (see p. 74).

[CCV12] Matteo Collina, Giovanni Emanuele Corazza, and Alessandro Vanelli-Coralli. "Introducing the QEST broker: Scaling the IoT by bridging MQTT and REST". In: *2012 IEEE 23rd International Symposium on Personal, Indoor and Mobile Radio Communications-(PIMRC)*. IEEE. 2012, pp. 36–41 (see p. 42).

[CDN16] M. Clark, P. Dutta, and M. W. Newman. "Towards a Natural Language Programming Interface for Smart Homes". In: *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct*. UbiComp 16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 49–52 (see p. 85).

[CDR17] Fulvio Corno, Luigi De Russis, and Alberto Monge Roffarello. "A high-level approach towards end user development in the IoT". In: *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*. ACM. 2017, pp. 1546–1552 (see p. 77).

[Cen+16] M. Centenaro et al. "Long-range communications in unlicensed bands: the rising stars in the IoT and smart city scenarios". In: *IEEE Wireless Communications* 23.5 (Oct. 2016), pp. 60–67 (see pp. 29, 36).

[CGP08] Antonio Carzaniga, Alessandra Gorla, and Mauro Pezzè. "Self-Healing by Means of Automatic Workarounds". In: *Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-Managing Systems*. Seams '08. New York, NY, USA: Association for Computing Machinery, 2008, pp. 17–24 (see p. 173).

[CH06] Krzysztof Czarnecki and Simon Helsen. "Feature-based Survey of Model Transformation Approaches". In: *IBM Systems Journal*. Vol. 45. Feb. 2006, pp. 621–645 (see p. 60).

[CH11] Chao Chen and Sumi Helal. "A Device-Centric Approach to a Safer Internet of Things". In: *Proceedings of the 2011 International Workshop on Networking and Object Memories for the Internet of Things*. NoME-IoT '11. New York, NY, USA: Association for Computing Machinery, 2011, pp. 1–6 (see p. 161).

[Cha+12] Ioannis Chatzigiannakis et al. "True self-configuration for the IoT". In: *Proceedings of 2012 International Conference on the Internet of Things, IOT 2012* (2012), pp. 9–15 (see p. 119).

[Cha+15] Victor Charpenay et al. "An ontology design pattern for IoT device tagging systems". In: *Proceedings - 2015 5th International Conference on the Internet of Things, IoT 2015* (2015), pp. 138–145 (see p. 66).

[Cha+18] Tusher Chakraborty et al. "Fall-curve: A novel primitive for IoT Fault detection and isolation". In: *SenSys 2018 - Proceedings of the 16th Conference on Embedded Networked Sensor Systems* (2018), pp. 95–107 (see pp. 174, 178).

[Cha02] S. Chang. *Handbook of Software Engineering and Knowledge Engineering*. World Scientific Publishing Co., 2002 (see p. 58).

[Che+14] Ching Yu Chen et al. "Complex event processing for the internet of things and its applications". In: *2014 IEEE International Conference on Automation Science and Engineering (CASE)*. IEEE. 2014, pp. 1144–1149 (see pp. 41, 103).

[Che+15] Xing Chen et al. "Runtime model based approach to IoT application development". In: *Frontiers of Computer Science* 9.4 (Aug. 2015), pp. 540–553 (see p. 74).

[Che+17] Bin Cheng et al. "FogFlow: Easy Programming of IoT Services Over Cloud and Edges for Smart Cities". In: *IEEE Internet of Things Journal* PP (Aug. 2017), pp. 1–1 (see p. 81).

[Che+18a] Jiongyi Chen et al. "IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing." In: *NDSS*. 2018, (see p. 103).

[Che+18b] Bin Cheng et al. "FogFlow: Orchestrating IoT services over cloud and edges". In: *NEC Technical Journal* 13 (Nov. 2018), pp. 48–53 (see pp. 81, 82, 95).

[Che+18c] M. Chernyshev et al. "Internet of Things (IoT): Research, Simulators, and Testbeds". In: *IEEE Internet of Things Journal* 5.3 (June 2018), pp. 1637–1647 (see p. 100).

[Cho97] Timothy C. K. Chou. "Beyond Fault Tolerance". In: *IEEE Computer* 30.4 (1997), pp. 47–49 (see p. 45).

[Cic+17] Federico Ciccozzi et al. "Model-driven engineering for mission-critical iot systems". In: *IEEE software* 34.1 (2017), pp. 46–53 (see p. 74).

[Cir+15] Simone Cirani et al. "The IoT hub: A fog node for seamless management of heterogeneous connected smart objects". In: *2015 12th Annual IEEE International Conference on Sensing, Communication, and Networking-Workshops (SECON Workshops)*. IEEE. 2015, pp. 1–6 (see p. 28).

[CIT19] CITILAB. *S4A*. [Online; accessed 2019]. Fundació pel Foment de la Societat del Coneixement (CITILAB), 2019 (see p. 75).

[CK16] Y. Chen and T. Kunz. "Performance evaluation of IoT protocols under a constrained wireless access network". In: *2016 International Conference on Selected Topics in Mobile Wireless Networking (MoWNeT)*. 2016, pp. 1–7 (see pp. 35, 36).

[Cle+18] Diego Clerissi et al. "Towards an Approach for Developing and Testing Node-RED IoT Systems". In: EnSEmble 2018. Lake Buena Vista, FL, USA: Association for Computing Machinery, 2018, pp. 1–8 (see p. 94).

[Clo+09] Robert Cloutier et al. "The Concept of Reference Architectures". In: *Systems Engineering* 14.3 (2009) (see p. 42).

[Com+15] Alberto Compagno et al. "To NACK or Not to NACK? Negative Acknowledgments in Information-Centric Networking". In: *2015 24th International Conference on Computer Communication and Networks (ICCCN)* (Aug. 2015) (see p. 164).

[Com17] Computer History Museum. *Margaret Hamilton*. [Online; accessed Jun. 2018]. 2017 (see p. 6).

[com21] Home Assistant community. *Home Assistant*. [Online; accessed 2021]. 2021 (see p. 87).

[Com90] C/S2ESC - Software & Systems Engineering Standards Committee. "IEEE Standard Glossary of Software Engineering Terminology". In: *IEEE 610.12-1990 - IEEE Standard Glossary of Software Engineering Terminology* (Dec. 1990), pp. 1–84 (see p. 62).

[Cor19] "Onion Corporation". *Onion.io*. [Online; accessed 2019]. "Onion Corporation," 2019 (see p. 27).

[Cos21] Pedro Miguel Sousa da Costa. "Decentralized Real-time IoT Orchestration". MA thesis. Porto: Faculty of Engineering, University of Porto, 2021 (see pp. 199, 202).

[Cox07] Philip T. Cox. "Visual Programming Languages". In: *Wiley Encyclopedia of Computer Science and Engineering*. John Wiley & Sons, Inc., 2007 (see p. 58).

[CP16] Patricia Charlton and Stefan Poslad. "A sharable wearable maker community IoT application". In: *2016 12th International Conference on Intelligent Environments (IE)*. IEEE. 2016, pp. 16–23 (see p. 28).

[CS17] Federico Ciccozzi and Romina Spalazzese. "MDE4IoT: Supporting the Internet of Things with Model-Driven Engineering". In: *Intelligent Distributed Computing X*. Ed. by Costin Badica et al. Cham: Springer International Publishing, 2017, pp. 67–76 (see p. 73).

[CSM17] Pethuru Raj Chelliah, Harihara Subramanian, and Anupama Murali. *Architectural Patterns: Uncover essential patterns in the most indispensable realm of enterprise architecture*. Packt Publishing Ltd, 2017 (see p. 42).

[Cun+02] João Carlos Cunha et al. "Reset-driven fault tolerance". In: *European Dependable Computing Conference*. Springer. 2002, pp. 102–120 (see p. 178).

[CVD16] M. Conoscenti, A. Vetrò, and J. C. De Martin. "Blockchain for the Internet of Things: A systematic literature review". In: *2016 IEEE/ACS 13th International Conference of Computer Systems and Applications (AICCSA)*. Nov. 2016, pp. 1–6 (see p. 103).

[DAA16] Bahadir Dundar, Merve Astekin, and Mehmet S. Aktas. "A Big Data Processing Framework for Self-Healing Internet of Things Applications". In: *2016 12th International Conference on Semantics, Knowledge and Grids (SKG)*. 2016, pp. 62–68 (see p. 10).

[Dar+15] Kashif Dar et al. "A resource oriented integration architecture for the Internet of Things: A business process perspective". In: *Pervasive and Mobile Computing* 20 (2015), pp. 145–159 (see p. 37).

[Dav+16] Nigel Davies et al. "Privacy mediators: Helping IoT cross the chasm". In: *Proceedings of the 17th International Workshop on Mobile Computing Systems and Applications*. ACM. 2016, pp. 39–44 (see p. 42).

[Del+13] Flavia C. Delicato et al. "Towards an IoT Ecosystem". In: *Proceedings of the First International Workshop on Software Engineering for Systems-of-Systems*. SESoS '13. ACM, 2013, pp. 25–28 (see p. 132).

[Del+17] Daniele Dell'Aglio et al. "On a Web of Data Streams." In: *DeSemWeb ISWC*. 2017, (see p. 34).

[DeL98] David E DeLano. "Telephony Data Handling Pattern Language". In: *Pattern Languages of Program Design*. Vol. 53. 1998 (see p. 153).

[DF17] Joao Pedro Dias and Hugo Sereno Ferreira. "Automating the Extraction of Static Content and Dynamic behavior from e-Commerce Websites". In: *Procedia Computer Science* 109 (2017). 8th International Conference on Ambient Systems, Networks and Technologies, ANT-2017 and the 7th International Conference on Sustainable Energy Information Technology, SEIT 2017, 16-19 May 2017, Madeira, Portugal, pp. 297–304 (see p. 336).

[DFF18] João Pedro Dias, João Pascoal Faria, and Hugo Sereno Ferreira. "A Reactive and Model-Based Approach for Developing Internet-of-Things Systems". In: *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*. Sept. 2018, pp. 276–281 (see pp. 132, 283, 329).

[DFS19] Joao Pedro Dias, Hugo Sereno Ferreira, and Tiago Boldt Sousa. "Testing and Deployment Patterns for the Internet-of-Things". In: *Proceedings of the 24th European Conference on Pattern Languages of Programs*. EuroPLop '19. Irsee, Germany: Association for Computing Machinery, 2019 (see pp. 144, 151, 165, 284, 328).

[DGL19] Inc." "DGLogik. *IoT Application Platform DGLogik*. [Online; accessed 2019]. "DGLogik, Inc.", 2019 (see p. 76).

[DGM07] Paul M Duvall, Andrew Glover, and Steve Matyas. *Continuous integration*. Addison-Wesley Professional, 2007 (see pp. 66, 156, 157).

[Dia+18] João Pedro Dias et al. "A Brief Overview of Existing Tools for Testing the Internet-of-Things". In: *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. Apr. 2018, pp. 104–109 (see pp. 64, 283, 330).

[Dia+20a] Joao Pedro Dias et al. "A Pattern-Language for Self-Healing Internet-of-Things Systems". In: *Proceedings of the 25th European Conference on Pattern Languages of Programs*. EuroPLop '20. Irsee, Germany: Association for Computing Machinery, 2020 (see pp. 144, 159, 171, 218, 284, 326).

[Dia+20b] João Pedro Dias et al. "Visual Self-Healing Modelling for Reliable Internet-of-Things Systems". In: *Proceedings of the 20th International Conference on Computational Science (ICCS)*. Springer, 2020, pp. 27–36 (see pp. 144, 150, 159, 171, 218, 284, 327).

[Dig+19] *SERP4IoT '19: Proceedings of the 1st International Workshop on Software Engineering Research & Practices for the Internet of Things*. Montreal, Quebec, Canada: IEEE Press, 2019 (see pp. 8, 64).

[Diz+19] Jasenka Dizdarevi et al. "A Survey of Communication Protocols for Internet of Things and Related Challenges of Fog and Cloud Computing Integration". In: *ACM Computing Surveys* 51.6 (Jan. 2019), 116:1–116:29 (see p. 34).

[DLF20] João Pedro Dias, André Lago, and Hugo Sereno Ferreira. "Conversational Interface for Managing Non-Trivial Internet-of-Things Systems". In: *Proceedings of the 20th International Conference on Computational Science (ICCS)*. Springer, 2020, pp. 27–36 (see pp. 131, 258, 285, 327).

[DMF18] João Pedro Dias, Ângelo Martins, and Hugo Sereno Ferreira. "A Blockchain-based Approach for Access Control in eHealth Scenarios". In: *Journal of Information Assurance and Security* 13 (4 2018), pp. 125–136 (see p. 332).

[Doh+10] A. Dohr et al. "The Internet of Things for Ambient Assisted Living". In: *Seventh International Conference on Information Technology: New Generations* (2010), pp. 804–809 (see p. 24).

[Dom12] Mari Carmen Domingo. "An overview of the internet of underwater things". In: *Journal of Network and Computer Applications* 35.6 (2012), pp. 1879–1890 (see p. 118).

[Dou12] Charalampos Doukas. *Building Internet of Things with the ARDUINO*. CreateSpace Independent Publishing Platform, 2012 (see p. 27).

[DPB17] Flávia C. Delicato, Paulo F. Pires, and Thais Batista. "The Resource Management Challenge in IoT". In: *Resource Management for Internet of Things*. Cham: Springer International Publishing, 2017, pp. 7–18 (see pp. 103, 104).

[DPC17] João Pedro Dias, José Pedro Pinto, and José Magalhães Cruz. "A Hands-on Approach on Botnets for Behavior Exploration". In: *Proceedings of the 2nd International Conference on Internet of Things, Big Data and Security*. SCITEPRESS - Science and Technology Publications, 2017, pp. 463–469 (see pp. 107, 335, 336).

[DRF21] Joao Pedro Dias, André Restivo, and Hugo Sereno Ferreira. "Empowering Visual Internet-of-Things Mashups with Self-Healing Capabilities". In: *2021 IEEE/ACM 3rd International Workshop on Software Engineering Research Practices for the Internet of Things (SERP4IoT)*. 2021 (see pp. 144, 159, 171, 218, 284, 325).

[DRF22] João Pedro Dias, André Restivo, and Hugo Sereno Ferreira. "Designing and Constructing Internet-of-Things Systems: An Overview of the Ecosystem". In: *Internet of Things* (2022) (see pp. 64, 322, 323).

[DS18] S. Dharur and K. Swaminathan. "Efficient surveillance and monitoring using the ELK stack for IoT powered Smart Buildings". In: *2018 2nd International Conference on Inventive Systems and Control (ICISC)*. Jan. 2018, pp. 700–705 (see p. 91).

[DSM20] João Pedro Dias, Hugo Sereno Ferreira, and Ângelo Martins. "A Blockchain-Based Scheme for Access Control in e-Health Scenarios". In: *Proceedings of the Tenth International Conference on Soft Computing and Pattern Recognition (SoCPaR 2018)*. Ed. by Ana Maria Madureira et al. Cham: Springer International Publishing, 2020, pp. 238–247 (see p. 334).

[Dua+15] Y. Duan et al. "Everything as a Service (XaaS) on the Cloud: Origins, Current and Future Trends". In: *2015 IEEE 8th International Conference on Cloud Computing*. June 2015, pp. 621–628 (see p. 29).

[Dua+18] Duarte Duarte et al. "Towards a Framework for Agent-Based Simulation of User behavior in E-Commerce Context". In: *Trends in Cyber-Physical Multi-Agent Systems. The PAAMS Collection - 15th International Conference, PAAMS 2017*. Cham: Springer International Publishing, 2018, pp. 30–38 (see p. 335).

[Dua+22] Miguel Duarte et al. "Evaluation of IoT Self-healing Mechanisms using Fault-Injection in Message Brokers". In: *2022 IEEE/ACM 4th International Workshop on Software Engineering Research Practices for the Internet of Things (SERP4IoT)*. 2022 (see pp. 218, 323, 324).

[Dua21] Miguel Pereira Duarte. "MQTT Chaos Engineering for Self-Healing IoT Systems". MA thesis. Porto: Faculty of Engineering, University of Porto, 2021 (see p. 218).

[Dun+06] Adam Dunkels et al. "Protothreads: Simplifying Event-driven Programming of Memory-constrained Embedded Systems". In: *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*. SenSys '06. New York, NY, USA: ACM, 2006, pp. 29–42 (see p. 74).

[Dur+17] Caglar Durmaz et al. "Modelling Contiki-Based IoT Systems". In: *6th Symposium on Languages, Applications and Technologies (SLATE 2017)*. Ed. by Ricardo Queirós et al. Vol. 56. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 5:1–5:13 (see p. 74).

[Ecl19] Eclipse Foundation, Inc. *IoT Developer Survey 2019*. Tech. rep. Eclipse Foundation, Inc., 2019 (see p. 69).

[Edw01] Stephen H Edwards. "A framework for practical, automated black-box testing of component-based software". In: *Software Testing, Verification and Reliability* 11.2 (2001), pp. 97–111 (see p. 62).

[Ein+17] A. F. Einarsson et al. "SmartHomeML: Towards a Domain-Specific Modeling Language for Creating Smart Home Applications". In: *2017 IEEE International Congress on Internet of Things (ICIOT)*. June 2017, pp. 82–88 (see pp. 72, 73).

[Elo+14] Veli-Pekka Eloranta et al. *Designing distributed control systems: A pattern language approach*. Wiley Publishing, 2014 (see pp. 66, 164, 165, 168, 173, 177, 179–181).

[EM95] M. Erwig and B. Meyer. "Heterogeneous visual languages-integrating visual and textual programming". In: *Proceedings of Symposium on Visual Languages* (1995), pp. 318–325 (see p. 59).

[Esp19] Espressif Systems. *ESP8266 Technical Reference Manual*. Tech. rep. Shanghai, China: Espressif Systems, 2019 (see pp. 28, 201).

[esp19] esp8266.ru. *ESPlorer — Integrated Development Environment (IDE) for ESP8266 developers*. [Online; accessed 2019]. 2019 (see p. 70).

[Esp20] Espressif Systems. *ESP32 Technical Reference Manual*. Tech. rep. Shanghai, China: Espressif Systems, 2020 (see pp. 28, 201).

[Ete+15] T. Eterovic et al. "An Internet of Things Visual Domain Specific Modeling Language based on UML". In: *2015 25th International Conference on Information, Communication and Automation Technologies, ICAT 2015 - Proceedings*. Oct. 2015 (see p. 72).

[Eug+03] Patrick Th. Eugster et al. "The Many Faces of Publish/Subscribe". In: *ACM Computing Surveys* 35.2 (June 2003), pp. 114–131 (see p. 41).

[Eur09] European Economic and Social Committee. *Internet of Things — An action plan for Europe*. Tech. rep. European Commission, 2009 (see p. 4).

[EV10] J.L. Eveleens and C. Verhoef. "The rise and fall of the Chaos report figures". In: *IEEE Software* 27.1 (Jan. 2010), pp. 30–36 (see p. 6).

[Fan+14] Romano Fantacci et al. "A network architecture solution for efficient IOT WSN backhauling: challenges and opportunities". In: *IEEE Wireless Communications* 21.4 (2014), pp. 113–119 (see p. 28).

[Far+15] Muhammad Umar Farooq et al. "A critical analysis on the security concerns of internet of things (IoT)". In: *International Journal of Computer Applications* 111.7 (2015) (see p. 107).

[Fee16] Sean Feeney. *A Primer on Continuous Delivery*. [Online; accessed Jun. 2018]. Feb. 2016 (see p. 56).

[Feh+14] Christoph Fehling et al. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer, 2014 (see pp. 53, 55, 66).

[Fer+12] Nicolas Ferry et al. *Wcomp, middleware for ubiquitous computing and system focused adaptation*. 2012 (see p. 78).

[Fer11] Hugo Sereno Ferreira. "Adaptive Object-Modeling Patterns, Tools and Applications". PhD thesis. Faculty of Engineering of the University of Porto, Porto, Portugal, 2011 (see pp. 57, 61).

[FF07] Glenn A Fink and Deborah A Frincke. "Autonomic Computing: Freedom or Threat?" In: *;login:* 32.2 (2007), pp. 6–14 (see p. 5).

[FF11] Mattern Friedemann and Christian Floerkemeir. "From the Internet of Computers to the Internet of Things". In: *From Active Data Management to Event-Based Systems and More* (2011), pp. 242–259. arXiv: 9780201398298 (see pp. 4, 21).

[Fis+04] Gerhard Fischer et al. "Meta-design: a manifesto for end-user development". In: *Communications of the ACM* 47.9 (2004), pp. 33–37 (see p. 258).

[Fit12] Brian Fitzgerald. "Software crisis 2.0". In: *Computer* 45.4 (2012), pp. 89–91 (see pp. 1, 6, 7, 17, 132, 278).

[Flo+08] Christian Floerkemeier et al. *The Internet of Things: First International Conference, IOT 2008, Proceedings*. Vol. 4952. Zurich, Switzerland, Jan. 2008 (see p. 4).

[FM17] F. Fleurey and B. Morin. "ThingML: A Generative Approach to Engineer Heterogeneous and Distributed Systems". In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)* (Apr. 2017), pp. 185–188 (see p. 94).

[Foj+12] M. Fojtik et al. "Bubble Razor: An architecture-independent approach to timing-error detection and correction". In: *2012 IEEE International Solid-State Circuits Conference*. 2012, pp. 488–490 (see p. 167).

[For+13] Giancarlo Fortino et al. "An agent-based middleware for cooperating smart objects". In: *International Conference on Practical Applications of Agents and Multi-Agent Systems*. Springer. 2013, pp. 387–398 (see p. 100).

[For+14] Giancarlo Fortino et al. "Integration of agent-based and Cloud Computing for the smart objects-oriented IoT". In: *Proceedings of the 2014 IEEE 18th International Conference on Computer Supported Cooperative Work in Design, CSCWD 2014* (2014), pp. 493–498 (see p. 120).

[For+17] Giancarlo Fortino et al. "Modeling and Simulating Internet-of-Things Systems: A Hybrid Agent-Oriented Approach". In: *Computing in Science and Engineering* 19.5 (2017), pp. 68–76 (see p. 100).

[For+18] Giancarlo Fortino et al. "Agent-oriented cooperative smart objects: From IoT system design to implementation". In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 48.11 (2018), pp. 1949–1956 (see p. 120).

[For+19] Giancarlo Fortino et al. "Fluidware: An Approach Towards Adaptive and Scalable Programming of the IoT". In: *Models, Languages, and Tools for Concurrent and Distributed Programming*. Vol. 11665. Springer International Publishing, 2019, pp. 411–427 (see p. 73).

[Fou19a] "Micro:bit Educational Foundation". *Micro:bit Educational Foundation*. [Online; accessed 2019]. "Micro:bit Educational Foundation", 2019 (see p. 28).

[Fou19b] "Raspberry Pi Foundation". *Raspberry Pi*. [Online; accessed 2019]. "Raspberry Pi Foundation", 2019 (see p. 27).

[Fow02] Martin Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002 (see p. 57).

[Fra+13] Neil Fraser et al. *Blockly: A visual programming editor*. [Online; accessed 2019]. Google and the MIT Media Lab, 2013 (see p. 75).

[Fra17] B. Francis. *Web Thing API*. Tech. rep. Mozilla, 2017 (see p. 44).

[Fru+18] M. Frustaci et al. "Evaluating Critical Security Issues of the IoT World: Present and Future Challenges". In: *IEEE Internet of Things Journal* 5.4 (Aug. 2018), pp. 2483–2495 (see p. 107).

[FSM12] Hugo Sereno Ferreira, Tiago Boldt Sousa, and Angelo Martins. "Scalable Integration of Multiple Health Sensor Data for Observing Medical Patterns". In: *Cooperative Design, Visualization, and Engineering*. Ed. by Yuhua Luo. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 78–84 (see p. 176).

[Gab96] Richard P Gabriel. *Patterns of software*. Vol. 62. Oxford University Press, 1996 (see p. 57).

[Gam+95] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995, p. 334. eprint: dd (see pp. 57, 144).

[Gan16] P. Ganguly. "Selecting the right IoT cloud platform". In: *2016 International Conference on Internet of Things and Applications (IOTA)*. Jan. 2016, pp. 316–320 (see p. 165).

[Gas05] Matthew Gast. *802.11 wireless networks: the definitive guide*. " O'Reilly Media, Inc.", 2005 (see p. 33).

[GC03] Alan G. Ganek and Thomas A. Corbi. "The dawning of the autonomic computing era". In: *IBM Systems Journal* 42.1 (2003), pp. 5–18 (see pp. 4, 51–55, 286).

[Gen+17] Rosella Gennari et al. *End-User Development*. June. Springer, 2017 (see p. 258).

[GG17] Neha Garg and Ritu Garg. "Energy harvesting in IoT devices: A survey". In: *2017 International Conference on Intelligent Sustainable Systems (ICISS)*. 2017, pp. 127–131 (see p. 29).

[Ghi+17] Giuseppe Ghiani et al. "Personalization of Context-Dependent Applications Through Trigger-Action Rules". In: *ACM Transactions on Computer-Human Interaction* 24.2 (May 2017), pp. 1–33 (see pp. 77, 123).

[Gho+07] Debanjan Ghosh et al. "Self-healing systems — survey and synthesis". In: *Decision Support Systems* 42.4 (2007). Decision Support Systems in Emerging Economies, pp. 2164–2185 (see p. 54).

[Gia+15] Nam Ky Giang et al. "Developing IoT applications in the Fog: A Distributed Dataflow approach". In: *5th International Conference on the Internet of Things*. 2015, pp. 155–162 (see pp. 79, 80).

[GIM11] Dominique Guinard, Iulia Ion, and Simon Mayer. “In search of an internet of things service architecture: REST or WS-*? A developers’ perspective”. In: *International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*. Springer. 2011, pp. 326–337 (see p. 42).

[GLL18] N. K. Giang, R. Lea, and V. C. M. Leung. “Exogenous Coordination for Building Fog-Based Cyber Physical Social Computing and Networking Systems”. In: *IEEE Access* 6 (2018), pp. 31740–31749 (see pp. 79, 80, 95).

[Glo19] "ASUS Global". *Tinker Board, Single Board Computer*. [Online; accessed 2019]. "ASUS Global", 2019 (see p. 27).

[Glu+11] Alexander Gluhak et al. “A survey on facilities for experimental internet of things research”. In: *IEEE Communications Magazine* 49.11 (2011), pp. 58–67 (see pp. 97, 100).

[GND17] Alex Glikson, Stefan Nastic, and Schahram Dustdar. “Deviceless edge computing: extending serverless computing to the edge of the network”. In: *Proceedings of the 10th ACM International Systems and Storage Conference*. 2017, pp. 1–1 (see pp. 184, 198).

[Gom+17] Tiago Gomes et al. “A modeling domain-specific language for IoT-enabled operating systems”. In: *IECON 2017-43rd Annual Conference of the IEEE Industrial Electronics Society*. IEEE. 2017, pp. 3945–3950 (see p. 74).

[Gon+14] Cristian González García et al. “Midgar: Generation of heterogeneous objects interconnecting applications. A Domain Specific Language proposal for Internet of Things scenarios”. In: *Computer Networks* 64 (2014), pp. 143–158 (see p. 73).

[Goo20] Google, LLC. *Google Assistant, your own personal Google*. 2020 (see p. 261).

[Gou+17] Sotirios K. Goudos et al. *A Survey of IoT Key Enabling and Future Technologies: 5G, Mobile IoT, Sematic Web and Applications*. Vol. 97. 2. Springer US, 2017, pp. 1645–1675 (see pp. 32, 33).

[GR19] Loveleen Gaur and Ravi Ramakrishnan. *Internet of Things: Approach and Applicability in Manufacturing*. CRC Press, June 2019 (see p. 44).

[Gro19] "Autonomous Networks Research Group". *Cooja Simulator – Contiki*. [Online; accessed 2019]. "Autonomous Networks Research Group", 2019 (see p. 99).

[GT15] Padmini Gaur and Mohit P. Tahiliani. “Operating systems for IoT devices: A critical survey”. In: *Proceedings - 2015 IEEE Region 10 Symposium, TENSYMP 2015* (2015), pp. 33–36 (see p. 27).

[GTW+10] Dominique Guinard, Vlad Trifa, Erik Wilde, et al. “A resource oriented architecture for the Web of Things.” In: *IoT*. 2010, pp. 1–8 (see p. 42).

[Gub+13] Jayavardhana Gubbi et al. “Internet of Things (IoT): A vision, architectural elements, and future directions”. In: *Future Generation Computer Systems* 29.7 (2013), pp. 1645–1660. arXiv: 1207.0203 (see pp. 36, 43, 104).

[Gui16] Dominique Guinard. *The IoT needs a defrag*. [Online; accessed Mar. 2018]. Mar. 2016 (see pp. 1, 10, 13, 14, 28, 136, 156).

[Gup+17] Harshit Gupta et al. “iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments”. In: *Software: Practice and Experience* 47.9 (2017), pp. 1275–1296 (see pp. 81, 99).

[Gus+19] Marjan Gusev et al. “A deviceless edge computing approach for streaming IoT applications”. In: *IEEE Internet Computing* 23.1 (2019), pp. 37–45 (see p. 184).

[Han+14] Son N. Han et al. “DPWSim: A simulation toolkit for IoT applications using devices profile for web services”. In: *2014 IEEE World Forum on Internet of Things, WF-IoT 2014* (2014), pp. 544–547 (see p. 99).

[Han+17] David Hanes et al. *IoT Fundamentals: Networking Technologies, Protocols, and Use Cases for the Internet of Things*. Cisco Press, 2017 (see pp. 38, 39).

[Han06] Robert S. Hanmer. “Error Containment”. In: PLoP ’06 (2006) (see pp. 108–110, 113, 114, 179).

[Han07] Robert Hanmer. *Patterns for Fault Tolerant Software*. Wiley Publishing, 2007 (see pp. 9, 66, 67, 109–116, 160, 162, 164, 168, 173–179).

[Han12] Robert S. Hanmer. "Pattern Mining Patterns". In: *Proceedings of the 19th Conference on Pattern Languages of Programs*. PLoP '12. Tucson, Arizona: The Hillside Group, 2012 (see p. 144).

[Han14] Robert S. Hanmer. "Patterns for Fault Tolerant Cloud Software". In: *Proceedings of the 21st Conference on Pattern Languages of Programs*. Usa: The Hillside Group, 2014 (see pp. 159, 179).

[Har+16] Nicolas Harrand et al. "ThingML: A Language and Code Generation Framework for Heterogeneous Targets". In: *Proceedings of the ACM/IEEE 19th Int. Conference on Model Driven Engineering Languages and Systems*. MODELS '16. New York, NY, USA: ACM, 2016, pp. 125–135 (see pp. 75, 94).

[HC15] Justin Huang and Maya Cakmak. "Supporting mental model accuracy in trigger-action programming". In: *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp 2015)* (2015), pp. 215–225 (see p. 77).

[He+19] W. He et al. "When Smart Devices Are Stupid: Negative Experiences Using Home Smart Devices". In: *2019 IEEE Security and Privacy Workshops (SPW)*. 2019, pp. 150–155 (see p. 96).

[Heo+15] S. Heo et al. "IoT-MAP: IoT mashup application platform for the flexible IoT ecosystem". In: *2015 5th International Conference on the Internet of Things (IOT)*. Oct. 2015, pp. 163–170 (see p. 78).

[HKN02] Ulf Hashagen, Reinhard Keil-Slawik, and Arthur L. Norberg. *History of Computing: Software Issues*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002 (see p. 6).

[HL96] Katie Hafner and Matthew Lyon. *Where Wizards Stay Up Late: The Origins of the Internet*. 1996. arXiv: arXiv:1011.1669v3 (see p. 3).

[Hol02] Gerard J. Holzmann. "The logic of bugs". In: *ACM SIGSOFT Symposium on the Foundations of Software Engineering* (2002), pp. 81–87 (see p. 91).

[Hor01] Paul Horn. *Autonomic Computing: IBM's Perspective on the State of Information Technology*. Tech. rep. 26. [Online; accessed Nov. 2020]. IBM Corporation, 2001 (see pp. 4, 5, 10, 51, 53–55, 137).

[Hoy15] Matthew B. Hoy. "If This Then That: An Introduction to Automated Task Services". In: *Medical Reference Services Quarterly* 34.1 (2015). PMID: 25611444, pp. 98–103. eprint: https://doi.org/10.1080/02763869.2015.986796 (see p. 68).

[HTI11] Sara Hachem, Thiago Teixeira, and Valérie Issarny. "Ontologies for the Internet of Things". In: *Proceedings of the 8th Middleware Doctoral Symposium, MDS'11 of the 12th ACM/IFIP/USENIX International Middleware Conference* (2011) (see p. 14).

[HVM12] G. F. Hurlburt, J. Voas, and K. W. Miller. "The Internet of Things: A Reality Check". In: *IT Professional* 14.June (2012), pp. 56–59 (see p. 106).

[HW04] Gregor Hohpe and Bobby Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional, 2004 (see p. 57).

[IBH19] IBH SYSTEMS GmbH. *Eclipse NeoSCADA*. 2019 (see p. 70).

[IBM21] IBM. *Internet of Things architecture*. [Online; accessed 2021]. 2021 (see p. 43).

[IFT19] "IFTTT". *IFTTT helps your apps and devices work together*. "IFTTT", 2019 (see pp. 77, 86, 124).

[Ihi+20] Felicien Ihirwe et al. "Low-Code Engineering for Internet of Things: A State of Research". In: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. MODELS '20. Virtual Event, Canada: Association for Computing Machinery, 2020 (see pp. 70, 91, 93, 96, 123, 136, 200).

[IKK13] Janggwan Im, Seonghoon Kim, and Daeyoung Kim. "IoT mashup as a service: cloud-based mashup service for the Internet of things". In: *2013 IEEE International Conference on Services Computing*. IEEE. 2013, pp. 462–469 (see pp. 25, 79).

[Inc19a] Microchip Technology Inc. *Atmel Studio 7 | Microchip Technology*. [Online; accessed 2019]. 2019 (see p. 70).

[Inc19b] XOD Inc. *XOD: A visual programming language for microcontrollers*. [Online; accessed 2019]. XOD Inc., 2019 (see p. 75).

[Inc19c] "Blynk Inc." *How Blynk Works*. [Online; accessed 2019]. "Blynk Inc.", 2019 (see p. 77).

[Ior+18] Michaela Iorga et al. *Fog Computing Conceptual Model*. Tech. rep. National Institute of Standards and Technology (NIST), 2018 (see p. 39).

[IoT19a] "IoTIFY". *IoTIFY- Develop full stack IoT Application with virtual device simulation*. [Online; accessed 2019]. "IoTIFY", 2019 (see p. 98).

[IoT19b] "FIT IoT-LAB". *FIT/IoT-LAB • Very large scale open wireless sensor network testbed*. [Online; accessed 2019]. "FIT IoT-LAB", 2019 (see p. 98).

[ISO14] ISO/IEC JTC 1. *Internet of Things (IoT) - Preliminary Report*. Tech. rep. ISO, 2014 (see p. 22).

[IT19] J. Iyengar and M. Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. 2019 (see p. 34).

[Jan17a] Srini Janarthanam. *Hands-on chatbots and conversational UI development: Build chatbots and voice user interfaces with Chatfuel, Dialogflow, Microsoft Bot Framework, Twilio, and Alexa Skills*. Packt Publishing Ltd, 2017 (see p. 262).

[Jan17b] D. Janes. *IOTDB*. Tech. rep. IOTDB.org, 2017 (see pp. 44, 45).

[JEC14] Patrick Janssen, Halil Erhan, and Kian Wee Chen. "Visual Dataflow Modelling - Some thoughts on complexity". In: *Proceedings of the 32nd eCAADe Conference*. 2014 (see pp. 137, 218, 258).

[Jen+13] C. Jennings et al. *Media Types for Sensor Markup Language (SENML)*. [Online; accessed 19. May 2018]. Apr. 2013 (see p. 44).

[Jeo+18] K. E. Jeon et al. "BLE Beacons for Internet of Things Applications: Survey, Challenges, and Opportunities". In: *IEEE Internet of Things Journal* 5.2 (Apr. 2018), pp. 811–828 (see p. 31).

[JJW18] Tobias Jung, Nasser Jazdi, and Michael Weyrich. "A survey on dynamic simulation of automation systems and components in the Internet of Things". In: *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*. IEEE, Sept. 2018, pp. 1–4 (see p. 100).

[Joh89] Barry Johnson. *Design and Analysis of Fault-tolerant Digital Systems*. Addison-Wesley Professional, Feb. 1989, pp. 1–87 (see p. 10).

[JOJ21] Wha Sook Jeon, Hyun Seob Oh, and Dong Geun Jeong. "Decision of Ranging Interval for IEEE 802.15. 4z UWB Ranging Devices". In: *IEEE Internet of Things Journal* (2021) (see p. 33).

[JZ05] Weijia Jia and Wanlei Zhou. "Reliability and replication techniques". In: *Distributed Network Systems: From Concepts to Implementations* (2005), pp. 213–254 (see p. 243).

[Kan+19] Runchang Kang et al. "Minuet: Multimodal Interaction with an Internet of Things". In: *Symposium on Spatial User Interaction*. SUI '19. New Orleans, LA, USA: Association for Computing Machinery, 2019 (see p. 85).

[Kar+15] Vasileios Karagiannis et al. "A Survey on Application Layer Protocols for the Internet of Things". In: *Transaction on IoT and Cloud Computing* 3.1 (2015), pp. 11–17 (see pp. 34, 35).

[KC18] Byungseok Kang and Hyunseung Choo. "An experimental study of a reliable IoT gateway". In: *ICT Express* 4.3 (2018), pp. 130–133 (see p. 178).

[KD18] Puneet Kumar and Behnam Dezfouli. "Implementation and Analysis of QUIC for MQTT". In: *CoRR* abs/1810.07730 (2018) (see p. 34).

[Kev09] Ashton Kevin. "That 'Internet of Things' Thing". In: *RFID journal* (2009) (see pp. 21, 104, 278).

[KGC16] Megha Koshti, Sanjay Ganorkar, and L Chiari. "IoT Based Health Monitoring System by Using Raspberry Pi and ECG Signal". In: *International Journal of Innovative Research in Science, Engineering and Technology* 5.5 (2016) (see p. 27).

[Kha+19] Kasem Khalil et al. "Self-healing hardware systems: A review". In: *Microelectronics Journal* 93.August (2019), p. 104620 (see p. 54).

[KHA17] Roland Kuhn, Brian Hanafee, and Jamie Allen. *Reactive Design Patterns*. 2017, p. 360 (see pp. 108, 109, 111, 113, 114, 173, 174, 179).

[Kim20] T. Kim. "Short Research on Voice Control System Based on Artificial Intelligence Assistant". In: *2020 International Conference on Electronics, Information, and Communication (ICEIC)*. 2020, pp. 1–2 (see p. 85).

[Kis+19] R. Kishore Kodali et al. "Low Cost Smart Home Automation System using Smart Phone". In: *2019 IEEE R10 Humanitarian Technology Conference (R10-HTC)(47129)*. 2019, pp. 120–125 (see p. 84).

[KJP15] A. Krylovskiy, M. Jahn, and E. Patti. "Designing a Smart City Internet of Things Platform with Microservice Architecture". In: *2015 3rd International Conference on Future Internet of Things and Cloud*. Aug. 2015, pp. 25–30 (see p. 41).

[KK16] Ruslan Kirichek and Andrey Koucheryavy. "Internet of Things Laboratory Test Bed". In: *Wireless Communications, Networking and Applications: Proceedings of WCNA 2014*. Ed. by Qing-An Zeng. New Delhi: Springer India, 2016, pp. 485–494 (see p. 97).

[Kle+14] Robert Kleinfeld et al. "Glue.Things: A Mashup Platform for Wiring the Internet of Things with the Internet of Services". In: *Proceedings of the 5th International Workshop on Web of Things*. WoT '14. New York, NY, USA: ACM, 2014, pp. 16–21 (see p. 78).

[Kle+19] Martin Kleppmann et al. "Local-First Software: You Own Your Data, in Spite of the Cloud". In: *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2019. Athens, Greece: Association for Computing Machinery, 2019, pp. 154–178 (see p. 29).

[KM16] Maninder Jeet Kaur and Piyush Maheshwari. "Building smart cities applications using IoT and cloud-based architectures". In: *2016 International Conference on Industrial Informatics and Computer Systems (CIICS)*. IEEE. 2016, pp. 1–5 (see p. 25).

[KMB19] Barbara Kitchenham, Lech Madeyski, and Pearl Brereton. "Problems with Statistical Practice in Human-Centric Software Engineering Experiments". In: *Proceedings of the Evaluation and Assessment on Software Engineering*. EASE '19. New York, NY, USA: ACM, 2019, pp. 134–143 (see p. 249).

[Kni12] John Knight. *Fundamentals of Dependable Computing for Software Engineers*. 1st. Chapman & Hall/CRC, 2012 (see p. 47).

[Kol+17] C. Kolias et al. "DDoS in the IoT: Mirai and Other Botnets". In: *Computer* 50.7 (2017), pp. 80–84 (see p. 107).

[Koo11] Philip Koopman. *Embedded Software Testing*. Carnegie Mellon University, 2011 (see p. 97).

[Kop11] Hermann Kopetz. *Real-Time Systems*. Real-Time Systems Series. Boston, MA: Springer US, 2011, pp. 307–323. eprint: 96332259 (see p. 120).

[KP10] Christian Kohls and Stefanie Panke. "Is that true...?" In: *Proceedings of the 16th Conference on Pattern Languages of Programs - PLoP '09* January (2010), p. 1 (see pp. 140, 145, 146).

[KPG18] A Kertesz, T Pflanzner, and T Gyimothy. "A mobile IoT device simulator for IoT-Fog-Cloud systems". In: *Journal of Grid Computing* 17.3 (Oct. 2018), pp. 529–551 (see pp. 99, 100).

[Kru16] John Krumm. *Ubiquitous computing fundamentals*. Chapman and Hall/CRC, 2016 (see p. 25).

[KS18] Minhaj Ahmad Khan and Khaled Salah. "IoT security: Review, blockchain solutions, and open challenges". In: *Future Generation Computer Systems* 82 (2018), pp. 395–411 (see p. 105).

[KSP20] M. Karthikeyan, T. S. Subashini, and M. S. Prashanth. "Implementation of Home Automation Using Voice Commands". In: *Data Engineering and Communication Technology*. Ed. by K. Srujan Raju et al. Singapore: Springer Singapore, 2020, pp. 155–162 (see p. 85).

[Kub16] Thomas Kubitza. "Apps for Environments: Running Interoperable Apps in Smart Environments with the meSchup IoT Platform". In: *International Workshop on Interoperability and Open-Source Solutions*. Springer. 2016, pp. 158–172 (see p. 76).

[Küs11] Jochen Küster. *Foundations of Model-Driven Software Engineering*. 2011 (see pp. 59, 60).

[KW17] M. Kessentini and M. Wimmer. "Guest Editorial Special Issue on Computational Intelligence for Software Engineering and Services Computing". In: *IEEE Transactions on Emerging Topics in Computational Intelligence* 1.3 (June 2017), pp. 143–144 (see p. 8).

[Lag18] André Sousa Lago. "Exploring Complex Event Management in Smart-Spaces through a Conversation-Based Approach". MA thesis. Porto: Faculty of Engineering, University of Porto, 2018 (see p. 258).

[Lai+19] Xiaozheng Lai et al. "IoT Implementation of kalman filter to improve accuracy of air quality monitoring and prediction". In: *Applied Sciences* 9.9 (2019), p. 1831 (see p. 176).

[Lan+19] D. Lan et al. "Latency Analysis of Wireless Networks for Proximity Services in Smart Home and Building Automation: The Case of Thread". In: *IEEE Access* 7 (2019), pp. 4856–4867 (see p. 32).

[Lan20a] Marc Langheinrich. "Long Live the IoT". In: *IEEE Pervasive Computing* 19.2 (2020), pp. 4–7 (see p. 278).

[Lan20b] H. Langley. *The best Siri commands for controlling HomeKit and the Apple HomePod*. [Online; accessed Jan. 2020]. 2020 (see p. 85).

[Lap92] J. C. Laprie. "Dependability: Basic Concepts and Terminology". In: *Dependability: Basic Concepts and Terminology: In English, French, German, Italian and Japanese*. Ed. by J. C. Laprie. Vienna: Springer Vienna, 1992, pp. 3–245 (see p. 9).

[Lar+17] Xabier Larrucea et al. "Software Engineering for the Internet of Things". In: *IEEE Software* 34.1 (2017), pp. 24–28 (see pp. 64, 92, 93).

[Lau+91] D Lau-Kee et al. "VPL: an active, declarative visual programming system". In: *Proceedings 1991 IEEE Workshop on Visual Languages* (1991), pp. 40–46 (see p. 59).

[LB16] Huichen Lin and Neil W Bergmann. "IoT privacy and security challenges for smart home environments". In: *Information* 7.3 (2016), p. 44 (see p. 67).

[LD18] Chia-Chi Li and Behnam Dezfouli. "ProCal: A Low-Cost and Programmable Calibration Tool for IoT Devices". In: *Internet of Things – ICIOT 2018*. Ed. by Dimitrios Georgakopoulos and Liang-Jie Zhang. Cham: Springer International Publishing, 2018, pp. 88–105 (see pp. 180, 181).

[LDF21] André Sousa Lago, João Pedro Dias, and Hugo Sereno Ferreira. "Managing non-trivial internet-of-things systems with conversational assistants: A prototype and a feasibility experiment". In: *Journal of Computational Science* 51 (2021), p. 101324 (see pp. 258, 285, 323).

[Le+06] Franck Le et al. "Minerals: using data mining to detect router misconfigurations". In: *Proceedings of the 2006 SIGCOMM workshop on Mining network data*. 2006, pp. 293–298 (see p. 165).

[Lea+12] Yu Beng Leau et al. "Software development life cycle AGILE vs traditional approaches". In: *International Conference on Information and Network Technology*. Vol. 37. 1. IACSIT Press, 2012, pp. 162–167 (see p. 56).

[Leo+18] Maurizio Leotta et al. "Towards a Runtime Verification Approach for Internet of Things Systems". In: *Proceedings of the International Conference on Web Engineering*. Vol. 11153. Springer International Publishing, 2018, pp. 83–96 (see p. 121).

[LHK16] Steve Liang, Chih-Yuan Huang, and Tania Khalafbeigi. *OGC SensorThings API*. Tech. rep. Open Geospatial Consortium, 2016 (see pp. 44, 45).

[Li+17] Toby Jia-Jun Li et al. "Programming IoT Devices by Demonstration Using Mobile Apps". In: *End-User Development*. Ed. by Simone Barbosa et al. Cham: Springer International Publishing, 2017, pp. 3–17 (see p. 76).

[Li+19] Guangpu Li et al. "DFix: Automatically Fixing Timing Bugs in Distributed Systems". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Pldi 2019. New York, NY, USA: Association for Computing Machinery, 2019, pp. 994–1009 (see p. 167).

[Lin+04] Wang Linzhang et al. "Generating test cases from UML activity diagram based on gray-box method". In: *Software Engineering Conference, 2004. 11th Asia-Pacific*. IEEE. IEEE, 2004, pp. 284–291 (see p. 62).

[Lin+13] Yuanxin Lin et al. "Design and implementation of remote/short-range smart home monitoring system based on ZigBee and STM32". In: *Journal of Applied Sciences, Engineering and Technology* 5 (2013), pp. 2792–2798 (see p. 28).

[Lin+17] Jie Lin et al. "A Survey on Internet of Things: Architecture, Enabling Technologies, Security and Privacy, and Applications". In: *IEEE Internet of Things Journal* 4.5 (2017), pp. 1125–1142. arXiv: 1305.0982 (see p. 40).

[Lin+19] Yi-Bing Lin et al. "SensorTalk: An IoT device failure detection and calibration mechanism for smart farming". In: *Sensors* 19.21 (2019), p. 4788 (see p. 181).

[LL03] Philip Levis and Nelson Lee. "Tossim: A simulator for tinyos networks". In: *UC Berkeley, September* 24 (2003) (see p. 99).

[LLD18] Yongxin Liao, Eduardo De Freitas Rocha Loures, and Fernando Deschamps. "Industrial Internet of Things: A Systematic Literature Review and Insights". In: *IEEE Internet of Things Journal* 5.6 (2018), pp. 4515–4525 (see pp. 5, 22).

[Loo+12] Vilen Looga et al. "Mammoth: A massive-scale emulation platform for internet of things". In: *Cloud Computing and Intelligent Systems (CCIS), 2012 IEEE 2nd International Conference on*. Vol. 3. IEEE. Oct. 2012, pp. 1235–1239 (see p. 98).

[Lot08] Tony Loton. *Introduction to Microsoft Popfly, No Programming Required*. Lotontech Limited, 2008 (see p. 61).

[Lou+19] Pedro Lourenço et al. "CloudCity: A Live Environment for the Management of Cloud Infrastructures". In: *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering — Volume 1: ENASE*. INSTICC. SciTePress, 2019, pp. 27–36 (see pp. 333, 334).

[Lou+20] Pedro Lourenço et al. "Experimenting with Liveness in Cloud Infrastructure Management". In: *Communications in Computer and Information Science* 1172 (2020) (see p. 331).

[LQG18] Gustavo López, Luis Quesada, and Luis A. Guerrero. "Alexa vs. Siri vs. Cortana vs. Google Assistant: A Comparison of Speech-Based Natural User Interfaces". In: *Advances in Human Factors and Systems Interaction*. Ed. by Isabel L. Nunes. Cham: Springer International Publishing, 2018, pp. 241–250 (see pp. 84, 259, 268).

[Lu+17] D. Lu et al. "A Secure Microservice Framework for IoT". In: *2017 IEEE Symposium on Service-Oriented System Engineering (SOSE)*. Apr. 2017, pp. 9–18 (see p. 41).

[Luk+17] Daimler AG Lukas Reinfurt et al. "Internet of Things Security Patterns". In: *Conference Proceedings of the 24rd Conference Pattern Languages of Programs* (2017) (see pp. 66, 67).

[LY02] Kalle Lyytinen and Youngjin Yoo. "Ubiquitous computing". In: *Communications of the ACM* 45.12 (2002), pp. 63–96 (see p. 25).

[LZZ14] Guohong Li, Wenjing Zhang, and Yi Zhang. "A design of the IOT gateway for agricultural greenhouse". In: *Sensors & Transducers* 172.6 (2014), p. 75 (see p. 28).

[Maa+19] Zakaria Maamar et al. "Towards a seamless coordination of cloud and fog: illustration through the internet-of-things". In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. ACM. 2019, pp. 2008–2015 (see p. 29).

[Mag15] L Magnoni. "Modern Messaging for Distributed Sytems". In: *Journal of Physics: Conference Series* 608 (May 2015), p. 12038 (see p. 35).

[Mai+15] L. Mainetti et al. "Web of Topics: An IoT-aware model-driven designing approach". In: *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*. Dec. 2015, pp. 46–51 (see p. 73).

[Man+19] Marco Manca et al. "Supporting end-user debugging of trigger-action rules for IoT applications". In: *International Journal of Human-Computer Studies* 123 (2019), pp. 56–69 (see p. 86).

[Mar+17] Mohsen Marjani et al. "Big IoT data analytics: architecture, opportunities, and open research challenges". In: *ieee access* 5 (2017), pp. 5247–5261 (see p. 105).

[Mar+19] T. Martin et al. *Every Google Assistant command to give your Nest speaker or display*. [Online; accessed January 2020]. 2019 (see p. 85).

[Mar05] Toni Marinucci. "Characterization and Development of Distributed, Adaptive Real-Time Systems". PhD thesis. Ohio University, 2005 (see p. 170).

[Mat+20] Tiago Matias et al. "Determining Microservice Boundaries: A Case Study Using Static and Dynamic Software Analysis". In: *European Conference on Software Architecture*. Springer. 2020, pp. 315–332 (see p. 217).

[Max+07] E Michael Maximilien et al. "A domain-specific language for web apis and services mashups". In: *International Conference on Service-Oriented Computing*. Springer. 2007, pp. 13–26 (see p. 61).

[MB19] Redowan Mahmud and Rajkumar Buyya. "Modeling and Simulation of Fog and Edge Computing Environments Using iFogSim Toolkit". In: *Fog and Edge Computing*. John Wiley & Sons, Ltd, 2019. Chap. 17, pp. 433–465 (see p. 99).

[MBF02] Stephen J Mellor, Marc Balcer, and Ivar Foreword By-Jacoboson. *Executable UML: A foundation for model-driven architectures*. Addison-Wesley Longman Publishing Co., Inc., 2002 (see p. 59).

[MD97] Gerard Meszaros and Jim Doble. "Pattern Languages of Program Design". In: (1997). Ed. by Robert C. Martin, Dirk Riehle, and Frank Buschmann, pp. 529–574 (see p. 57).

[MHF17] Brice Morin, Nicolas Harrand, and Franck Fleurey. "Model-Based Software Engineering to Tame the IoT Jungle". In: *IEEE Software* 34.1 (2017), pp. 30–36 (see pp. 93, 137).

[Mi+17] Xianghang Mi et al. "An empirical characterization of IFTTT: ecosystem, usage, and performance". In: *Proceedings of the 2017 Internet Measurement Conference*. 2017, pp. 398–404 (see pp. 86, 124).

[Mic19] Microsoft. *IoT Signals – Summary of Research Learnings*. Tech. rep. Microsoft, 2019 (see pp. 5, 7, 37, 38, 136).

[Mik18] Greg Williams Mike Karlesky Mark VanderVoord. *Unity Test API*. 2018. URL: https://github.com/ThrowTheSwitch/Unity#unity-test-api (visited on 02/19/2018) (see p. 98).

[Min+16] Julien Mineraud et al. "A gap analysis of Internet-of-Things platforms". In: *Computer Communications* 89-90 (2016), pp. 5–16. arXiv: 1502.01181 (see pp. 70, 92, 199).

[Mis11] Jibitesh Mishra. *Software Engineering*. Pearson Education India, 2011 (see p. 6).

[Mit18] Martin Mitrevski. "Conversational interface challenges". In: *Developing Conversational Interfaces for iOS*. Springer, 2018, pp. 217–228 (see pp. 84, 259).

[MK16] N. Mohan and J. Kangasharju. "Edge-Fog cloud: A distributed cloud for Internet of Things computations". In: *2016 Cloudification of the Internet of Things (CIoT)*. Nov. 2016, pp. 1–6 (see p. 200).

[ML15] Morteza Mohammadi Zanjireh and Hadi Larijani. "A Survey on Centralised and Distributed Clustering Routing Algorithms for WSNs". In: *IEEE Vehicular Technology Conference*. Vol. 2015. May 2015, pp. 1–6 (see p. 37).

[MM01] Timothy F Masterson and R Mark Meyer. "Sivil : a True Visual Programming Language for Students". In: *Journal of Computing in Small Colleges* 4.May 2001 (2001), pp. 74–86 (see p. 59).

[MM18] Henry Muccini and Mahyar Tourchi Moghaddam. *IoT Architectural Styles: A Systematic Mapping Study*. Springer International Publishing, 2018, pp. 68–85 (see pp. 41, 42).

[MM21] Amir Makhshari and Ali Mesbah. "IoT Bugs and Development Challenges". In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 2021, pp. 460–472 (see p. 278).

[Moc+17] Kentaro Mochizuki et al. "Development and field experiment of wide area Wi-SUN system based on IEEE 802.15.4g". In: *2016 IEEE 3rd World Forum on Internet of Things, WF-IoT 2016* (2017), pp. 76–81 (see p. 33).

[Mon20] MongoDB, Inc. *MongoDB*. 2020 (see p. 265).

[MPM20] Jefferson Seide Molléri, Kai Petersen, and Emilia Mendes. "An empirically evaluated checklist for surveys in software engineering". In: *Information and Software Technology* 119 (2020), p. 106240 (see p. 124).

[MR16] R. Matei and A. Radovici. "Remote management system for embedded devices: Wyliodrin". In: *2016 15th RoEduNet Conference: Networking in Education and Research*. Sept. 2016, pp. 1–5 (see p. 76).

[MRG18] Armin Moin, Stephan Rössler, and Stephan Günnemann. "ThingML+: Augmenting Model-Driven Software Engineering for the Internet of Things with Machine Learning". In: *ArXiv* (2018) (see p. 74).

[Muc+18] Henry Muccini et al. "Self-adaptive IoT architectures: An emergency handling case study". In: *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*. 2018, pp. 1–6 (see p. 53).

[Mun+13] Sirajum Munir et al. "Cyber Physical System Challenges for Human-in-the-Loop Control". In: *Presented as part of the 8th International Workshop on Feedback Computing*. San Jose, CA: USENIX, 2013, pp. 777–780 (see p. 25).

[Mur04] Richard Murch. *Autonomic Computing*. IBM Press, 2004 (see pp. 56, 103).

[Mur19] Matthew Murdoch. *ArduinoUnit - ArduinoUnit is a unit testing framework for Arduino libraries*. [Online; accessed 2019]. 2019 (see p. 98).

[MZ14] K. Misura and M. Zagar. "Internet of things cloud mediator platform". In: *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. May 2014, pp. 1052–1056 (see p. 42).

[MZU16] Usama Mehboob, Qasim Zaib, and Chaudhry Usama. *Survey of IoT Communication Protocols Techniques, Applications, and Issues*. 2016 (see pp. 32–35).

[NAA+18] Mohammed Islam NAAS et al. "A Graph Partitioning-Based Heuristic for Runtime IoT Data Placement Strategies in a Fog Infrastructure". In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. SAC '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 767–774 (see p. 81).

[NAG19] Mahda Noura, Mohammed Atiquzzaman, and Martin Gaedke. "Interoperability in Internet of Things: Taxonomies and Open Challenges". In: *Mobile Networks and Applications* 24.3 (June 2019), pp. 796–809 (see pp. 37, 45).

[Nai17a] N. Naik. "Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP". In: *2017 IEEE International Systems Engineering Symposium (ISSE)*. Oct. 2017, pp. 1–7 (see pp. 34, 36, 202).

[Nai17b] Nitin Naik. "Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP". In: *2017 IEEE international systems engineering symposium (ISSE)*. IEEE. 2017, pp. 1–7 (see p. 42).

[NAS20] "NASA/JPL-Caltech". *Voyager 2 Returns to Normal Operations*. [Online; accessed 2020]. "NASA/JPL-Caltech", 2020 (see p. xx).

[NBB14] Agneta Nilsson, Jan Bosch, and Christian Berger. "Visualizing testing activities to support continuous integration: A multiple case study". In: *International Conference on Agile Software Development*. Springer. Springer, 2014, pp. 171–186 (see p. 156).

[ND18] Stefan Nastic and Schahram Dustdar. "Towards deviceless edge computing: Challenges, design aspects, and models for serverless paradigm at the edge". In: *The Essence of Software Engineering*. Springer, Cham, 2018, pp. 121–136 (see pp. 184, 185).

[Nei+14] Ricardo Neisse et al. "A model-based security toolkit for the internet of things". In: *2014 Ninth International Conference on Availability, Reliability and Security*. IEEE. 2014, pp. 78–87 (see p. 74).

[Neu09] Peter G. Neumann. "Computer-related risk futures". In: *Proceedings - Annual Computer Security Applications Conference, ACSAC* (2009), pp. 35–40 (see p. 46).

[Neu20] Peter G. Neumann. *The RISKS Digest — Forum on Risks to the Public in Computers and Related Systems*. [Online; accessed Nov. 2020]. Nov. 2020 (see pp. 12, 278).

[Ni+09] Kevin Ni et al. "Sensor network data fault types". In: *ACM Transactions on Sensor Networks* 5.3 (2009), pp. 1–29 (see pp. 162, 234, 235).

[Noo+19] Joseph Noor et al. "DDFlow: Visualized declarative programming for heterogeneous IoT networks". In: *IoTDI 2019 - Proceedings of the 2019 Internet of Things Design and Implementation*. IoTDI '19. New York, NY, USA: ACM, 2019, pp. 172–177 (see pp. 79, 80, 82, 83, 95).

[Nor16] Amy Nordrum. *Popular Internet of Things Forecast of 50 Billion Devices by 2020 Is Outdated*. Oct. 2016. URL: `https://goo.gl/EnE3td` (visited on 12/13/2017) (see p. 23).

[npm21] inc." "npm. *NPM*. [Online; accessed 2021]. "npm, inc.", 2021 (see p. 87).

[NR68] P. Naur and B. Randell. "Software Engineering: Report of a Conference Sponsored by the NATO Science Committee". In: *NATO Software Engineering Conference* October 1968 (1968), p. 231 (see p. 6).

[OLe20] Nick O'Leary. *What's next with Node-RED?* [Online; accessed 2021]. 2020 (see pp. 87, 89, 90, 94, 286).

[ONe+16] Maire ONeill et al. "Insecurity by design: Todays IoT device security problem". In: *Engineering* 2.1 (2016), pp. 48–49 (see p. 107).

[Ope19a] OpenJS Foundation. *Node-RED Community Survey*. Tech. rep. OpenJS Foundation, 2019 (see p. 135).

[Ope19b] OpenJS Foundation. *Node-RED, Flow-based programming for the Internet of Things*. [Online; accessed 2019]. 2019 (see pp. 18, 77, 87, 88, 135, 200).

[Ope21] OpenHAB Community and the OpenHAB Foundation eV. *openHAB — empowering the smart home*. [Online; accessed 2021]. 2021 (see p. 87).

[ORe16] Gerard (Cornelius Gerard) O'Regan. *Introduction to the history of computing : a computing history primer*. 2016, p. 296 (see pp. 2, 3).

[Ost02] Thomas Ostrand. "White-Box Testing". In: *Encyclopedia of Software Engineering* (2002) (see pp. 59, 62).

[Par19] "Particle". *Particle: Welcome to the Particle Docs*. [Online; accessed 2019]. "Particle", 2019 (see p. 27).

[Par21] Inc. Particle Industries. *Particle IDE*. [Online; accessed 2021]. 2021 (see pp. 70, 93).

[Pat+11] A Patel et al. "Autonomic agent-based self-managed intrusion detection and prevention system". In: *Proceedings of the South African Information Security Multi-Conference (SAISMC 2010)*. 2011, pp. 223–234 (see p. 119).

[PC13] Christian Prehofer and Luca Chiarabini. "From IoT Mashups to Model-based IoT". In: *W3C Workshop on the Web of Things* (2013) (see pp. 62, 68, 258).

[PC15] Christian Prehofer and Luca Chiarabini. "From Internet of things mashups to model-based development". In: *Proceedings - International Computer Software and Applications Conference* 3 (2015), pp. 499–504 (see pp. 77, 94, 95).

[PCB18] A. S. Prokhorov, M. A. Chudinov, and S. E. Bondarev. "Control systems software implementation using open source SCADA-system OpenSCADA". In: *2018 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*. Jan. 2018, pp. 220–222 (see p. 70).

[PD11] Harald Psaier and Schahram Dustdar. "A survey on self-healing systems: Approaches and systems". In: *Computing (Vienna/New York)* 91.1 (2011), pp. 43–73 (see pp. 4, 51, 55, 119, 121, 137, 174, 175, 218, 219).

[PDC20] Bruno Piedade, Joao Pedro Dias, and Filipe F. Correia. "An Empirical Study on Visual Programming Docker Compose Configurations". In: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. MODELS '20. Virtual Event, Canada: Association for Computing Machinery, 2020 (see p. 332).

[PDM19] D. Priest, T. Dyson, and T. Martin. *Every Alexa command to give your Amazon Echo smart speaker or display*. [Online; accessed January 2020]. 2019 (see p. 85).

[PDR18] José Pedro Pinto, João Pedro Dias, and R. J. F. Rossetti. "Growing Smart Cities on an Open-Data-Centric Cyber-Physical Platform". In: *2018 IEEE International Smart Cities Conference (ISC2)*. Sept. 2018, pp. 1–6 (see pp. 334, 335).

[PDS18] D. Pinto, João Pedro Dias, and Hugo Sereno Ferreira. "Dynamic Allocation of Serverless Functions in IoT Environments". In: *2018 IEEE 16th International Conference on Embedded and Ubiquitous Computing (EUC)*. Oct. 2018, pp. 1–8 (see pp. 184, 284, 329).

[PDS20] Guilherme Vieira Pinto, João Pedro Dias, and Hugo Sereno Ferreira. "Blockchain-Based PKI for Crowdsourced IoT Sensor Information". In: *Proceedings of the Tenth International Conference on Soft Computing and Pattern Recognition (SoCPaR 2018)*. Ed. by Ana Maria Madureira et al. Cham: Springer International Publishing, 2020, pp. 248–257 (see p. 328).

[Pen20] Sheng-lung Peng. *Principles of Internet of Things (IoT) Ecosystem: Insight Paradigm*. Ed. by Sheng-Lung Peng, Souvik Pal, and Lianfen Huang. Vol. 174. Intelligent Systems Reference Library. Cham: Springer International Publishing, 2020 (see p. 28).

[Per+12] Charith Perera et al. "Ca4iot: Context awareness for internet of things". In: *2012 IEEE International Conference on Green Computing and Communications*. IEEE. 2012, pp. 775–782 (see p. 104).

[Pet02] Marian Petre. "Mental imagery, visualisation tools and team work". In: *Proceedings of the Second Program Visualization Workshop*. June 2002, pp. 3–14 (see p. 94).

[Pfl+16] T. Pflanzner et al. "MobIoTSim: Towards a mobile IoT device simulator". In: *Proceedings - 2016 4th International Conference on Future Internet of Things and Cloud Workshops, W-FiCloud 2016* (2016), pp. 21–27 (see p. 99).

[Pin18] Duarte Manuel Ribeiro Pinto. "Serverless architectural design for IoT systems". MA thesis. Porto: Faculty of Engineering, University of Porto, 2018 (see p. 184).

[PIS17] P. Patel, M. Intizar Ali, and A. Sheth. "On Using the Intelligent Edge for IoT Analytics". In: *IEEE Intelligent Systems* 32.5 (2017), pp. 64–69 (see p. 199).

[PJ21] Vaishali S Phalake and Shashank D Joshi. "Low Code Development Platform for Digital Transformation". In: *Information and Communication Technology for Competitive Strategies (ICTCS 2020)*. Springer, 2021, pp. 689–697 (see p. 14).

[PK16] Tamás Pflanzner and Attila Kertész. "A survey of IoT cloud providers". In: *2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE. 2016, pp. 730–735 (see p. 25).

[PK19] Alexander Power and Gerald Kotonya. "Providing Fault Tolerance via Complex Event Processing and Machine Learning for IoT Systems". In: *Proceedings of the 9th International Conference on the Internet of Things*. IoT 2019. Bilbao, Spain: Association for Computing Machinery, 2019 (see p. 162).

[PKB15] Vamsikrishna Patchava, Hari Babu Kandala, and P Ravi Babu. "A smart home automation technique with raspberry pi using iot". In: *2015 International Conference on Smart Sensors and Systems (IC-SSS)*. IEEE. 2015, pp. 1–4 (see p. 27).

[Pla19] "PlatformIO". *PlatformIO: An open source ecosystem for IoT development*. [Online; accessed 2019]. "PlatformIO", 2019 (see pp. 69, 92, 98).

[PLF18] Pedro Martins Pontes, Bruno Lima, and João Pascoal Faria. "Izinto: a pattern-based IoT testing framework". In: *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*. ACM. 2018, pp. 125–131 (see pp. 103, 160, 162, 219).

[PQW10] Ahmed Patel, Qais Qassim, and Christopher Wills. "A survey of intrusion detection and prevention systems". In: *Information Management & Computer Security* (2010) (see p. 119).

[Pre+09] Mirko Presser et al. "The SENSEI project: Integrating the physical world with the digital world of the network of the future". In: *IEEE Communications Magazine* 47.4 (2009), pp. 1–4 (see p. 43).

[Pre+16] Alexandros Preventis et al. "IoT-A and FIWARE: Bridging the Barriers between the Cloud and IoT Systems Design and Implementation." In: *CLOSER (2)*. 2016, pp. 146–153 (see pp. 42, 43).

[Pre01] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. 5th. McGraw-Hill Higher Education, 2001 (see p. 49).

[Pro+99] Stacy J. Prowell et al. *Cleanroom Software Engineering: Technology and Process*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999 (see p. 6).

[Pro17] Open Web Application Security Project. *Tester IoT Security Guidance*. 2017. URL: https://www.owasp.org/index.php/OWASP_Internet_of_Things_Project (visited on 12/27/2017) (see pp. 102, 107).

[Pro19] "AT&T Intellectual Property". *AT&T Flow – Design and build solutions for the Internet of Things*. [Online; accessed 2019]. "AT&T Intellectual Property", 2019 (see p. 76).

[Pru07] Mark Pruett. *Yahoo! pipes*. O'Reilly, 2007 (see p. 61).

[Pu11] Calton Pu. "A World of Opportunities: CPS, IOT, and Beyond". In: *Proceedings of the 5th ACM International Conference on Distributed Event-Based System*. DEBS '11. New York, New York, USA: Association for Computing Machinery, 2011, pp. 229–230 (see p. 1).

[Pul01] Laura L. Pullum. *Software Fault Tolerance - Techniques and Implementation*. 2001, p. 343 (see p. 287).

[Pyc21] Pycom. *Pymakr — IDE plugin for MicroPython*. [Online; accessed 2021]. 2021 (see pp. 70, 93).

[Qan+16] Soheil Qanbari et al. "IoT design patterns: Computational constructs to design, build and engineer edge applications". In: *Proceedings - 2016 IEEE 1st International Conference on Internet-of-Things Design and Implementation, IoTDI 2016* (2016), pp. 277–282 (see pp. 66, 67, 157).

[Qin+14] Zhijing Qin et al. "A software defined networking architecture for the internet-of-things". In: *2014 IEEE network operations and management symposium (NOMS)*. IEEE. 2014, pp. 1–9 (see p. 37).

[Qin+16] Yongrui Qin et al. "When things matter: A survey on data-centric internet of things". In: *Journal of Network and Computer Applications* 64 (2016), pp. 137–153 (see p. 105).

[Qin+19] Zhu Qingyi et al. "Applications of Distributed Ledger Technologies to the Internet of Things: A Survey". In: *ACM Computing Surveys* 52 (Nov. 2019), pp. 1–34 (see p. 103).

[Rah+17] A. Rahmati et al. "IFTTT vs. Zapier: A Comparative Study of Trigger-Action Programming Frameworks". In: *ArXiv* abs/1709.02788 (2017) (see p. 68).

[Raj+18] B Rajesh et al. "A study on onion omega 2 plus IOT device in weather application". In: *Journal of Advanced Research in Dynamical and Control Systems* 10.3 Special Issue (2018), pp. 196–200 (see p. 27).

[Ram+17] Antonio Ramadas, Gil Domingues, Joao Pedro Dias, Ademar Aguiar, and Hugo Sereno Ferreira. "Patterns for Things that Fail". In: *Proceedings of the 24th Conference on Pattern Languages of Programs*. PLoP '17. ACM - Association for Computing Machinery, 2017 (see pp. 121, 144, 151, 156, 167, 204, 265, 284, 330).

[Ran+17] L Paul Jasmine Rani et al. "Dynamic traffic management system using infrared (IR) and Internet of Things (IoT)". In: *2017 Third International Conference on Science Technology Engineering & Management (ICONSTEM)*. IEEE. 2017, pp. 353–357 (see p. 34).

[Rat+19] D. Ratasich et al. "A Roadmap Toward the Resilient Internet of Things for Cyber-Physical Systems". In: *IEEE Access* 7 (2019), pp. 13260–13283 (see p. 12).

[Raw+18] Reza Rawassizadeh et al. "NoCloud: Exploring Network Disconnection through On-Device Data Analysis". In: *IEEE Pervasive Computing* 17 (Mar. 2018) (see pp. 184, 199).

[Ray16] Partha Pratim Ray. "A survey of IoT cloud platforms". In: *Future Computing and Informatics Journal* 1.1 (2016), pp. 35–46 (see pp. 25, 26).

[Ray17] Partha Pratim Ray. "A Survey on Visual Programming Languages in Internet of Things". In: *Scientific Programming* 2017 (2017), pp. 1–6 (see pp. 31, 35, 44, 59, 75, 76, 91, 93, 136, 258).

[Ray18] P.P. Ray. "A survey on Internet of Things architectures". In: *Journal of King Saud University - Computer and Information Sciences* 30.3 (2018), pp. 291–319 (see pp. 29, 37).

[Rei+16] Lukas Reinfurt et al. "Internet of things patterns". In: *21st European Conference on Pattern Languages of Programs - EuroPlop '16* (2016), pp. 1–21 (see pp. 66, 67, 119, 144, 154, 168, 174, 180).

[Rei+17a] Lukas Reinfurt et al. "Internet of Things Patterns for Device Bootstrapping and Registration". In: *Proceedings of the 22nd European Conference on Pattern Languages of Programs*. ACM. ACM, 2017, p. 15 (see pp. 66, 67, 144, 147).

[Rei+17b] Lukas Reinfurt et al. "Internet of things patterns for devices". In: *Ninth international Conferences on Pervasive Patterns and Applications (PATTERNS) 2017*. EuroPlop16. New York, NY, USA: Xpert Publishing Services (XPS), 2017, pp. 117–126 (see pp. 66, 67, 144).

[Rei+17c] Lukas Reinfurt et al. "Internet of Things Patterns for Devices: Powering, Operating, and Sensing". In: *International Journal on Advances in Internet Technology* (2017), pp. 106–123 (see pp. 66, 67, 144).

[Rei13] Stefan Reichhard. *UPnP and DPWS*. Tech. rep. Technical University of Vienna, 2013 (see p. 35).

[Res+09] Mitchel Resnick et al. "Scratch: Programming for all." In: *Communications of ACM* 52.11 (2009), pp. 60–67 (see p. 75).

[Rey+13] Carlos Rey-Moreno et al. "Experiences, challenges and lessons from rolling out a rural WiFi mesh network". In: *Proceedings of the 3rd ACM Symposium on Computing for Development*. 2013, pp. 1–10 (see p. 33).

[Ric04] Richard Murch. *Autonomic Computing*. Prentice Hall PTR, 2004, p. 336 (see pp. 51–53).

[Ric17] Chris Richardson. *A pattern language for microservices.(2017)*. 2017 (see pp. 41, 66).

[Ric18] Chris Richardson. *Microservices Patterns: With examples in Java*. Manning, 2018 (see p. 66).

[Rie+01] Dirk Riehle et al. "The architecture of a UML virtual machine". In: *International Conference on Object Oriented Programming Systems Languages and Applications (OOSPLA)* (2001), pp. 327–341 (see p. 60).

[Ris98] Linda Rising. *The patterns handbook: Techniques, strategies, and applications*. Cambridge University Press, 1998 (see p. 156).

[RK15] Jari Rauhamäki and Seppo Kuikka. "Patterns for Control System Safety". In: *Proceedings of the 18th European Conference on Pattern Languages of Program*. EuroPLoP '13. New York, NY, USA: Association for Computing Machinery, 2015 (see pp. 160, 177, 179).

[RM12] Andreas Rümpel and Klaus Meißner. "Requirements-driven quality modeling and evaluation in web mashups". In: *2012 Eighth International Conference on the Quality of Information and Communications Technology*. IEEE. 2012, pp. 319–322 (see p. 61).

[Ros+20] Thomas Rosenstatter et al. "REMIND: A Framework for the Resilient Design of Automotive Systems". In: *2020 IEEE Secure Development (SecDev)*. 2020, pp. 81–95 (see p. 117).

[RR12] K. Ravindran and M. Rabby. "Protocol-level reconfigurations for infusion of resilience in distributed network services". In: *2012 IEEE Network Operations and Management Symposium*. 2012, pp. 1207–1213 (see p. 173).

[RS17] A. Rajalakshmi and H. Shahnasser. "Internet of Things using Node-Red and alexa". In: *2017 17th International Symposium on Communications and Information Technologies (ISCIT)* (Sept. 2017), pp. 1–4 (see p. 85).

[RT17] Biljana L. Risteska Stojkoska and Kire V. Trivodaliev. "A review of Internet of Things for smart home: Challenges and solutions". In: *Journal of Cleaner Production* 140 (2017), pp. 1454–1464 (see pp. 105, 106).

[RTS10] Leah Muthoni Riungu, Ossi Taipale, and Kari Smolander. "Research issues for software testing in the cloud". In: *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*. IEEE. IEEE, 2010, pp. 557–564 (see p. 97).

[Rue18] Jim Ruehlin. *Continuous delivery and iterative development for Internet of Things projects*. 2018 (see p. 67).

[SA16] Rohini Shete and Sushma Agrawal. "IoT based urban climate monitoring using Raspberry Pi". In: *2016 International Conference on Communication and Signal Processing (ICCSP)*. IEEE. 2016, pp. 2008–2012 (see p. 27).

[SAA16] Abhimanyu Singh, Pankhuri Aggarwal, and Rahul Arora. "IoT based waste collection system using infrared sensors". In: *2016 5th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO)*. IEEE. 2016, pp. 505–509 (see p. 34).

[SAA21] Aymen J Salman, Mohammed Al-Jawad, and Wisam Al Tameemi. "Domain-Specific Languages for IoT: Challenges and Opportunities". In: *IOP Conference Series: Materials Science and Engineering*. Vol. 1067. 1. IOP Publishing. 2021, p. 012133 (see p. 74).

[Sae+21] Raghdah Saemaldahr et al. "Reference Architectures for the IoT: A Survey". In: *Innovative Systems for Intelligent Health Informatics*. Ed. by Faisal Saeed, Fathey Mohammed, and Abdulaziz Al-Nahari. Cham: Springer International Publishing, 2021, pp. 635–646 (see pp. 42, 43).

[Sal+15] A. Salihbegovic et al. "Design of a domain specific language and IDE for Internet of things applications". In: *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. 2015, pp. 996–1001 (see p. 73).

[Sam16] Tariq Samad. "Control Systems and the Internet of Things [Technical Activities]". In: *IEEE Control Systems* 36.1 (2016), pp. 13–16 (see p. 55).

[San+14] Luis Sanchez et al. "SmartSantander: IoT experimentation over a smart city testbed". In: *Computer Networks* 61 (2014), pp. 217–238 (see p. 98).

[Sar02] Titos Saridakis. "A System of Patterns for Fault Tolerance". In: *Proceedings of 2002 EuroPLoP Conference*. 2002 (see pp. 160, 164, 166, 173, 179).

[SAZ17] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices". In: *IEEE Access* 5.Ci (2017), pp. 3909–3943 (see p. 66).

[SB08] N. Vinod Sarma and Srinivas Rao Bhagavatula. "Freeway Patterns for SOA Systems". In: *Proceedings of the 15th Conference on Pattern Languages of Programs*. PLoP '08. New York, NY, USA: ACM, 2008, 6:1–6:10 (see pp. 152, 154).

[SBC20] Jan Seeger, Arne Bröring, and Georg Carle. "Optimally Self-Healing IoT Choreographies". In: *ACM Transactions on Internet Technology* 20.3 (July 2020) (see pp. 10, 121).

[SCF15] Tiago Boldt Sousa, Filipe Figueiredo Correia, and Hugo Sereno Ferreira. "Patterns for Software Orchestration on the Cloud". In: *Proceedings of the 22nd Conference on Pattern Languages of Programs*. PLoP '15. USA: The Hillside Group, 2015, 17:1–17:12 (see pp. 66, 67, 157).

[Sch02] C.R. Schoenberger. *The internet of things*. [Online; accessed Nov. 2020]. 2002 (see p. 21).

[Sch06] Douglas C Schmidt. "Model-Driven Engineering". In: *IEEE Computer* 39.2 (2006), pp. 25–31 (see p. 59).

[Scu18] Padraig Scully. *The Top 10 IoT Segments in 2018 – based on 1,600 real IoT projects*. [Online; accessed 15 May 2018]. 2018 (see p. 24).

[Sei+14] Ronny Seiger et al. "Modelling Complex and Flexible Processes for Smart Cyber-physical Environments". In: *Journal of Computational Science* 10 (Aug. 2014) (see p. 258).

[Sei+19] Ronny Seiger et al. "Toward a framework for self-adaptive workflows in cyber-physical systems". In: *Software and Systems Modeling* 18.2 (2019), pp. 1117–1134 (see p. 53).

[Sei17] Niels Seidel. "Empirical Evaluation Methods for Pattern Languages: Sketches, Classification, and Network Analysis". In: EuroPLoP '17. Irsee, Germany: Association for Computing Machinery, 2017 (see pp. 144, 146).

[Sem20] "Nordic Semiconductor". *Nordic Semiconductor*. [Online; accessed 2020]. "Nordic Semiconductor", 2020 (see p. 28).

[Sen+19] Joanna Sendorek et al. "FogFlow - Computation Organization for Heterogeneous Fog Computing Environments". In: *Computational Science – ICCS 2019*. Cham: Springer International Publishing, 2019, pp. 634–647 (see pp. 80, 81, 95).

[Seo+17] Jihye Seo et al. "Optimally Self-Healing IoT Choreographies". In: *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS* Part F1271.1 (2017), pp. 91–104. arXiv: 1907.04611 (see p. 119).

[Ser+18] Y. Seralathan et al. "IoT security vulnerability: A case study of a Web camera". In: *2018 20th International Conference on Advanced Communication Technology (ICACT)*. IEEE, Feb. 2018, pp. 172–177 (see p. 107).

[SF10] Shivanshu K Singh and Mohamed E Fayad. "The AnyCorrectiveAction stable design pattern". In: *Proceedings of the 17th Conference on Pattern Languages of Programs*. ACM. ACM, 2010, p. 24 (see p. 155).

[SFC21] Tiago Sousa, Hugo Sereno Ferreira, and Filipe Figueiredo Correia. "A Survey on the Adoption of Patterns for Engineering Software for the Cloud". In: *IEEE Transactions on Software Engineering* (2021), pp. 1–1 (see p. 285).

[SFZ17] Claudio Savaglio, Giancarlo Fortino, and Mengchu Zhou. "Towards interoperable, cognitive and autonomic IoT systems: An agent-based approach". In: *2016 IEEE 3rd World Forum on Internet of Things, WF-IoT 2016* (2017), pp. 58–63 (see p. 120).

[SHA17] Ronny Seiger, Stefan Herrmann, and Uwe Abmann. "Self-Healing for Distributed Workflows in the Internet of Things". In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. 2017, pp. 72–79 (see p. 10).

[Sic+15] S. Sicari et al. "Security, privacy and trust in Internet of things: The road ahead". In: *Computer Networks* 76 (2015), pp. 146–164. arXiv: 1404.7799 (see pp. 105, 106).

[Sil+20] Margarida Silva et al. "Visually-defined Real-Time Orchestration of IoT Systems". In: *Proceedings of the 17th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*. MOBIQUITOUS 2020. Online: Association for Computing Machinery, 2020 (see pp. 94, 199, 284, 325, 326).

[Sil+21] Margarida Silva et al. "A Review on Visual Programming for Distributed Computation in IoT". In: *Proceedings of the 21st International Conference on Computational Science (ICCS)*. Springer, 2021 (see pp. 64, 283, 324).

[Sil20a] Ana Margarida Oliveira Pinheiro da Silva. "Orchestration for Automatic Decentralization in Visually-defined IoT". MA thesis. Porto: Faculty of Engineering, University of Porto, 2020 (see p. 199).

[Sil20b] David Silver. *Lecture 9: Exploration and Exploitation*. 2020 (see pp. 188, 190, 191).

[Sim19] Inc." "SimpleSoft. *SimpleSoft's IoT Simulator for CoAP, MQTT, MQTT-SN, HTTP/REST sensors and gateways*. [Online; accessed 2019]. "SimpleSoft, Inc.", 2019 (see p. 100).

[SIN19] "Stanford Information Networks Group (SING)". *TOSSIM – TinyOS*. [Online; accessed 2019]. "Stanford Information Networks Group (SING)", 2019 (see p. 99).

[SK03] S. Sendall and W. Kozaczynski. "Model transformation: the heart and soul of model-driven software development". In: *IEEE Software* 20.5 (Sept. 2003), pp. 42–45 (see p. 94).

[SK09] James Scott and Rick Kazman. *Realizing and Refining Architectural Tactics : Availability*. Tech. rep. August. Software Engineering Institute, 2009 (see pp. 167, 205).

[SK17] Kiran Jot Singh and Divneet Singh Kapoor. "Create Your Own Internet of Things: A survey of IoT platforms." In: *IEEE Consumer Electronics Magazine* 6.2 (2017), pp. 57–68 (see p. 26).

[Ska+17] Olena Skarlat et al. "Towards QoS-Aware Fog Service Placement". In: *1st IEEE International Conference on Fog and Edge Computing, ICFEC 2017, Madrid, Spain, May 14-15, 2017*. IEEE Computer Society, 2017, pp. 89–96 (see p. 285).

[Sla20] Slack Technologies, Inc. *Slack: Where work happens*. 2020 (see p. 261).

[SLM17] Long Sun, Yan Li, and Raheel Ahmed Memon. "An open IoT framework based on microservices architecture". In: *China Communications* 14.2 (2017), pp. 154–162 (see p. 41).

[SM17] Dipa Soni and Ashwin Makwana. "A survey on MQTT: a protocol of internet of things (IoT)". In: *International Conference On Telecommunication, Power Analysis And Computing Techniques (ICTPACT-2017)*. 2017, (see pp. 34, 202).

[SM19] C. J. Sutherland and B. MacDonald. "RoboLang: A Simple Domain Specific Language to Script Robot Interactions". In: *2019 16th International Conference on Ubiquitous Robots (UR)*. 2019, pp. 265–270 (see p. 72).

[SMG19] Fabio Sartori, Riccardo Melen, and Fabio Giudici. "IoT Data Validation Using Spatial and Temporal Correlations". In: *Research Conference on Metadata and Semantics Research*. Springer. 2019, pp. 77–89 (see p. 176).

[Smi17] Sean Smith. *The Internet of Risky Things*. O'Reilly Media, Inc., 2017 (see pp. xx, 4, 5, 8, 10, 14, 278).

[SMM19] Eman Shaikh, Iman Mohiuddin, and Ayisha Manzoor. "Internet of Things (IoT): Security and Privacy Threats". In: *2019 2nd International Conference on Computer Applications Information Security (ICCAIS)*. 2019, pp. 1–6 (see p. 29).

[SN15] M. Sneps-Sneppe and D. Namiot. "On web-based domain-specific language for Internet of Things". In: *2015 7th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT)*. 2015, pp. 287–292 (see p. 74).

[SNS15] Sandra Scott-Hayward, Sriram Natarajan, and Sakir Sezer. "A survey of security in software defined networks". In: *IEEE Communications Surveys & Tutorials* 18.1 (2015), pp. 623–654 (see p. 165).

[Soa+21] Danny Soares et al. "Programming IoT-spaces: A User-Survey on Home Automation Rules". In: *Proceedings of the 21st International Conference on Computational Science (ICCS)*. Springer, 2021 (see pp. 123, 284, 324, 325).

[Soa20] Danny Almeida Soares. "Model-to-Model Mapping of Semi-Structured Specifications to Visual Programming Languages". MA thesis. Porto: Faculty of Engineering, University of Porto, 2020 (see pp. 123, 124).

[SOE18] Cagatay Sonmez, Atay Ozgovde, and Cem Ersoy. "EdgeCloudSim: An environment for performance evaluation of edge computing systems". In: *Transactions on Emerging Telecommunications Technologies* 29.11 (2018), e3493 (see p. 100).

[SOJ18] Adrian Santos, Markku Oivo, and Natalia Juristo. "Moving Beyond the Mean: Analyzing Variance in Software Engineering Experiments". In: *Product-Focused Software Process Improvement*. Ed. by Marco Kuhrmann et al. Springer, 2018, pp. 167–181 (see p. 248).

[Som10] Ian Sommerville. *Software Engineering*. 9th. USA: Addison-Wesley Publishing Company, 2010 (see pp. 58, 65, 68).

[Sot+14] Stelios Sotiriadis et al. "Towards simulating the internet of things". In: *IEEE 28th International Conference on Advanced Information Networking and Applications Workshops* (2014), pp. 444–448 (see p. 99).

[Sou+18] Tiago Boldt Sousa et al. "Engineering Software for the Cloud: External Monitoring and Failure Injection". In: *Proceedings of the 23rd European Conference on Pattern Languages of Programs*. 2018, pp. 1–8 (see p. 170).

[Spi17] Diomidis Spinellis. "Software-Engineering the Internet of Things". In: *IEEE Software* 34.1 (2017), pp. 4–6 (see p. 8).

[Spo04] Joel Spolsky. "The law of leaky abstractions". In: *Joel on Software*. Springer, 2004, pp. 197–202 (see pp. 93, 134, 279).

[ST13] Andy Stanford-Clark and Hong Linh Truong. "Mqtt for sensor networks (mqtt-sn) protocol specification". In: *International business machines (IBM) Corporation version* 1 (2013) (see p. 34).

[Sta14] John A Stankovic. "Research directions for the internet of things". In: *IEEE Internet of Things Journal* 1.1 (2014), pp. 3–9 (see p. 25).

[Sta15] Standish Group International. *The Chaos Report*. Tech. rep. Standish Group International, 2015 (see pp. 6, 132).

[sta22] statista. *Global digital population as of January 2021*. Tech. rep. statista, 2022 (see p. 4).

[STB18] Andreas Seitz, Felix Thiele, and Bernd Bruegge. "Fogxy: An Architectural Pattern for Fog Computing". In: *Proceedings of the 23rd European Conference on Pattern Languages of Programs*. Vol. 1. 1. ACM. ACM, 2018, p. 33 (see pp. 40, 41, 66, 67).

[STH18] Eugene Siow, Thanassis Tiropanis, and Wendy Hall. "Analytics for the Internet of Things: A Survey". In: *ACM Computing Surveys (CSUR)* 51.4 (2018) (see p. 105).

[STM19] "STMicroelectronics". *STM32 32-bit Arm Cortex MCUs*. [Online; accessed 2019]. "STMicroelectronics", 2019 (see p. 28).

[Sun+10] H. Sundmaeker et al. *Vision and Challenges for Realising the Internet of Things*. Tech. rep. European Commission, 2010 (see p. 23).

[SYB04] Anthony Sulistio, Chee Shin Yeo, and Rajkumar Buyya. "A taxonomy of computer-based simulations and its mapping to parallel and distributed systems simulation tools". In: *Software: Practice and Experience* 34.7 (2004), pp. 653–673 (see p. 98).

[Szy+17] T. Szydlo et al. "Flow-Based Programming for IoT Leveraging Fog Computing". In: *2017 IEEE 26th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*. June 2017, pp. 74–79 (see p. 77).

[Tan02] Andrew S Tanenbaum. *Computer networks*. Pearson Education India, 2002 (see p. 7).

[Tan18] H. Tankovska. *Number of internet of things (IoT) connected devices worldwide in 2018, 2025 and 2030*. Tech. rep. Data by Statista. Strategy Analytics, 2018 (see pp. 5, 22, 278).

[TC16] Kleanthis Thramboulidis and Foivos Christoulakis. "UML4IoT–A UML-based approach to exploit IoT in cyber-physical manufacturing systems". In: *Computers in Industry* 82 (2016), pp. 259–272 (see p. 72).

[Tei+11] Thiago Teixeira et al. "Service oriented middleware for the internet of things: a perspective". In: *European Conference on a Service-Based Internet*. Springer. 2011, pp. 220–229 (see p. 41).

[Ter16] Doug Terry. "Toward a New Approach to IoT Fault Tolerance". In: *Computer* 49.8 (2016), pp. 80–83 (see pp. 160, 173, 175, 178, 179).

[TGC17] V. Trifa, D. Guinard, and D. Carrera. *Web of Things*. Tech. rep. EVRYTHNG, 2017 (see p. 44).

[THP93] Walter F. Tichy, Nico Habermann, and Lutz Prechelt. "Summary of the Dagstuhl Workshop on Future Directions in Software Engineering". In: *SIGSOFT Software Engineering Notes* 18.1 (Jan. 1993), pp. 35–48 (see p. 140).

[Tig+09] Jean-Yves Tigli et al. "WComp middleware for ubiquitous computing: Aspects and composite event-based Web services". In: *Annals of Telecommunications* 64.3 (2009), pp. 197–214 (see p. 78).

[Tka+18] Rafał Tkaczyk et al. "Cataloging design patterns for internet of things artifact integration". In: *2018 IEEE International Conference on Communications Workshops, ICC Workshops 2018 - Proceedings* (2018), pp. 1–6 (see p. 66).

[TM17] Antero Taivalsaari and Tommi Mikkonen. "A Roadmap to the Programmable World: Software Challenges in the IoT Era". In: *IEEE Software* 34.1 (2017), pp. 72–80 (see pp. 8, 56, 93, 278).

[TMD19] Mohammad Tahir, Qazi Mamoon Ashraf, and Mohammad Dabbagh. "Towards enabling autonomic computing in IoT ecosystem". In: *Proceedings - IEEE 17th International Conference on Dependable, Autonomic and Secure Computing, IEEE 17th International Conference on Pervasive Intelligence and Computing, IEEE 5th International Conference on Cloud and Big Data Computing, 4th Cyber Scienc* (2019), pp. 646–651 (see pp. 53, 118, 119).

[Toa18] Ray Toal. *Software Architecture*. Loyola Marymount University, 2018 (see p. 38).

[Too+08] Y. Toor et al. "Vehicle Ad Hoc networks: applications and related technical issues". In: *IEEE Communications Surveys Tutorials* 10.3 (Mar. 2008), pp. 74–88 (see p. 33).

[Tor+20] D. Torres et al. "Real-time Feedback in Node-RED for IoT Development: An Empirical Study". In: *2020 IEEE/ACM 24th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. 2020, pp. 1–8 (see pp. 94, 246, 285, 326).

[Tor00] Wilfredo Torres-Pomales. "Software Fault Tolerance: A Tutorial". In: October (2000). NASA / TM-200-210616, p. 55 (see pp. 45, 47, 49, 160, 166, 169, 287).

[Tor20] Diogo Luis Rey Torres. "Increasing the feedback on IoT development in Node-RED". MA thesis. Porto: Faculty of Engineering, University of Porto, 2020 (see p. 246).

[TP-19] TP-Link Technologies Co., Ltd. *Smart Home Router SR20*. 2019 (see p. 28).

[TTJ18] Dave Thaler, Hannes Tschofenig, and Jaime Jimenez. *Report from the Internet of Things (IoT) Semantic Interoperability (IOTSI) Workshop 2016*. Tech. rep. Internet Engineering Task Force (IETF), 2018 (see p. 45).

[Tut03] Walter HW Tuttlebee. *Software defined radio: enabling technologies*. John Wiley & Sons, 2003 (see p. 3).

[TVH14] Dobroslav Tsonev, Stefan Videv, and Harald Haas. "Light fidelity (Li-Fi): towards all-optical networking". In: *Broadband Access Communication Technologies VIII*. Vol. 9007. International Society for Optics and Photonics. 2014, p. 900702 (see p. 34).

[TVM19] G. Tanganelli, C. Vallati, and E. Mingozzi. "Rapid Prototyping of IoT Solutions: A Developer's Perspective". In: *IEEE Internet Computing* 23.4 (July 2019), pp. 43–52 (see p. 28).

[UHM11] Dieter Uckelmann, Mark Harrison, and Florian Michahelles. *Architecting the internet of things*. Springer Science & Business Media, 2011 (see p. 27).

[UK18a] Itorobong S. Udoh and Gerald Kotonya. "Developing IoT applications: challenges and frameworks". In: *IET Cyber-Physical Systems: Theory & Applications* 3.2 (2018), pp. 65–72 (see pp. 14, 37).

[UK18b] Onoriode Uviase and Gerald Kotonya. "IoT Architectural Framework: Connection and Integration Framework for IoT Systems". In: *Electronic Proceedings in Theoretical Computer Science* 264 (2018), pp. 1–17 (see p. 41).

[Uki+16] Arijit Ukil et al. "IoT healthcare analytics: The importance of anomaly detection". In: *2016 IEEE 30th international conference on advanced information networking and applications (AINA)*. IEEE. 2016, pp. 994–997 (see p. 179).

[Ur+14] B. Ur et al. "Practical trigger-action programming in the smart home". In: *Conference on Human Factors in Computing Systems - Proceedings* (Apr. 2014) (see pp. 86, 124).

[Vas+19] D. R. Vasconcelos et al. "Cloud, Fog, or Mist in IoT? That is the qestion". In: *ACM Transactions on Internet Technology* 19.2 (2019) (see pp. 38, 39).

[Vaz+12] Mabel Vazquez-Briseno et al. "Using RFID/NFC and QR-code in mobile phones to link the physical and the digital world". In: *Interactive Multimedia*. IntechOpen, 2012, (see p. 31).

[Ver+11] Ovidiu Vermesan et al. "Internet of things strategic research roadmap". In: *Internet of things-global technological and societal trends* 1.2011 (2011), pp. 9–52 (see pp. 105, 120, 137).

[VH08] András Varga and Rudolf Hornig. "An overview of the OMNeT++ simulation environment". In: *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*. ICST (Institute for Computer Sciences, Social-Informatics and … 2008, p. 60 (see p. 100).

[Vie+19] Kleber Vieira et al. "Autonomic intrusion detection and response using big data". In: *IEEE Systems Journal* 14.2 (2019), pp. 1984–1991 (see p. 119).

[VMw20] VMware, Inc. *RabbitMQ*. 2020 (see p. 265).

[Was+19] Hironori Washizaki et al. "Landscape of iot patterns". In: *Proceedings - 2019 IEEE/ACM 1st International Workshop on Software Engineering Research and Practices for the Internet of Things, SERP4IoT 2019* (2019), pp. 57–60 (see p. 65).

[Was+20] Hironori Washizaki et al. "Landscape of Architecture and Design Patterns for IoT Systems". In: *IEEE Internet of Things Journal* 7.10 (2020), pp. 10091–10101 (see p. 65).

[WE16] Michael Weyrich and Christof Ebert. "Reference architectures for the internet of things". In: *IEEE Software* 33.1 (2016), pp. 112–116 (see pp. 42, 43).

[Web09] Rolf H. Weber. "Internet of things – Need for a new legal environment?" English. In: *Computer Law & Security Review: The International Journal of Technology Law and Practice* 25.6 (2009), pp. 522–527 (see p. 107).

[Wei+15] Bruce D. Weinberg et al. "Internet of Things: Convenience vs. privacy and secrecy". In: *Business Horizons* 58.6 (2015). SPECIAL ISSUE: THE MAGIC OF SECRETS, pp. 615–624 (see p. 107).

[Wei02] Mark Weiser. "The computer for the 21st century". In: *IEEE pervasive computing* 1.1 (2002), pp. 19–25 (see p. 4).

[Wen18] Lilian Weng. *The Multi-Armed Bandit Problem and Its Solutions*. 2018 (see pp. 188–190).

[Wiz+11] Theresa Wizemann et al. "Trustworthy medical device software". In: *Public Health Effectiveness of the FDA 510 (k) Clearance Process: Measuring Postmarket Performance and Other Select Topics: Workshop Report*. National Academies Press (US), 2011 (see p. 151).

[WLA06] León Welicki, Juan Manuel Cueva Lovelle, and Luis Joyanes Aguilar. "Meta-Specification and Cataloging of Software Patterns with Domain Specific Languages and Adaptive Object Models." In: *EuroPLoP*. ACM, 2006, pp. 359–392 (see p. 61).

[Won+21] Jason Wong et al. *Magic Quadrant for Enterprise Low-Code Application Platforms*. [Online; accessed Nov. 2021]. Sept. 2021 (see pp. 14, 137).

[WSJ15] R. Want, B. N. Schilit, and S. Jenson. "Enabling the Internet of Things". In: *Computer* 48.1 (2015), pp. 28–35 (see p. 199).

[XHL14] L. D. Xu, W. He, and S. Li. "Internet of Things in Industries: A Survey". In: *IEEE Transactions on Industrial Informatics* 10.4 (Nov. 2014), pp. 2233–2243 (see p. 37).

[XP18] Wen Xi and Evan W. Patton. "A Blocks-Based Approach to Internet of Things in MIT App Inventor". In: *BLOCKS+ 2018 workshop, part of the SPLASH 2018*. ACM, 2018, (see p. 75).

[Yan+12] Xue Yang et al. "A multi-layer security model for internet of things". In: *Internet of things*. Springer, 2012, pp. 388–393 (see p. 74).

[Yan+15] Fan Yang et al. "uPnP: Plug and Play Peripherals for the Internet of Things". In: *10th European Conference on Computer Systems*. EuroSys '15. New York, NY, USA: ACM, 2015, 25:1–25:14 (see p. 152).

[Yaq+17] I. Yaqoob et al. "Internet of Things Architecture: Recent Advances, Taxonomy, Requirements, and Open Challenges". In: *IEEE Wireless Communications* 24.3 (June 2017), pp. 10–16 (see p. 43).

[YF03] M Bani Younis and G Frey. "Formalization of Existing PLC Programs: A Survey". In: *Computational Engineering in Systems Applications* 70 (2003) (see p. 93).

[YSD15] Lina Yao, Quan Z. Sheng, and Schahram Dustdar. "Web-based management of the internet of things". In: *IEEE Internet Computing* 19.4 (2015), pp. 60–67 (see pp. 77, 79).

[ZA19] Syed Rameem Zahra and Mohammad Ahsan Chishti. "RansomWare and Internet of Things: A New Security Nightmare". In: *2019 9th International Conference on Cloud Computing, Data Science Engineering (Confluence)*. 2019, pp. 551–555 (see p. 278).

[Zai+18] T. A. Zaitoun et al. "Evaluation and Enhancement of the EdgeCloudSim using Poisson Interarrival time and Load capacity". In: *2018 8th International Conference on Computer Science and Information Technology (CSIT)*. July 2018, pp. 7–12 (see p. 100).

[Zam+19] Franco Zambonelli et al. "Towards Adaptive Flow Programming for the IoT: The Fluidware Approach". In: *2019 IEEE International Conference on Pervasive Computing and Communications Workshops, PerCom Workshops 2019* (2019), pp. 549–554 (see p. 73).

[Zam17] Franco Zambonelli. "Key abstractions for IoT-oriented software engineering". In: *IEEE Software* 34.1 (2017), pp. 38–45 (see p. 64).

[Zar18] Andrzej Zarzycki. "Strategies for the integration of smart technologies into buildings and construction assemblies". In: *Proceedings of eCAADe 2018 Conference*. 2018, pp. 631–640 (see p. 258).

[ZB21] Koen Zandberg and Emmanuel Baccelli. "Femto-Containers: DevOps on Microcontrollers with Lightweight Virtualization & Isolation for IoT Software Modules". working paper or preprint. June 2021 (see p. 285).

[ZDC14a] Y. Zhang, L. Duan, and J. L. Chen. "Event-Driven SOA for IoT Services". In: *2014 IEEE International Conference on Services Computing*. June 2014, pp. 629–636 (see p. 41).

[ZDC14b] Yang Zhang, Li Duan, and Jun Liang Chen. "Event-driven soa for iot services". In: *2014 IEEE International Conference on Services Computing*. IEEE. 2014, pp. 629–636 (see p. 41).

[ZEB18] T Zoranovic, V Erceg, and I Berkovic. "IoT project in agriculture". In: *Proceedings of the 8th International conference on applied internet and information technologies*. Vol. 8. "St Kliment Ohridski" University-Bitola, Faculty of Information and … 2018, pp. 17–21 (see p. 27).

[Zen+17] Xuezhi Zeng et al. "IOTSim: A simulator for analysing IoT applications". In: *Journal of Systems Architecture* 72 (2017), pp. 93–107 (see p. 99).

[Zha+14] Z. K. Zhang et al. "IoT Security: Ongoing Challenges and Research Opportunities". In: *2014 IEEE 7th International Conference on Service-Oriented Computing and Applications*. Nov. 2014, pp. 230–234 (see p. 97).

[Zho+15] S. Zhou et al. "Supporting Service Adaptation in Fault Tolerant Internet of Things". In: *2015 IEEE 8th International Conference on Service-Oriented Computing and Applications (SOCA)*. Oct. 2015, pp. 65–72 (see pp. 167, 176).

[Zig18] Alliance Zigbee. *JupiterMesh*. [Online; accessed May 2018]. May 2018 (see p. 32).

[ZW98] M.V. Zelkowitz and D.R. Wallace. "Experimental models for validating technology". In: *Computer* 31.5 (May 1998), pp. 23–31 (see p. 140).

Appendices

A | Publications

Parts of this dissertation — including ideas, approaches, results, tables, and figures — have been published in peer-reviewed journals and conference proceedings. The author published 24 peer-reviewed conference papers and 5 peer-reviewed journal articles. The author also supervised 6 students pursuing their Master's degree in Informatics and Computing Engineering, from Faculdade de Engenharia da Universidade do Porto (FEUP). The Table A.1 (p. 321) identifies these supervisions.

Table A.1: Co-supervised Master's thesis at FEUP.

Title	Author	Year
MQTT Chaos Engineering for Self-Healing IoT Systems	Miguel Duarte	2021
Visual Programming Language for Orchestration with Docker	Bruno Piedade	2020
Implementing a Multi-Approach Debugging of Industrial IoT Workflows	Andreia Rodrigues	2019
Dynamic Allocation of Serverless Functions in IoT Environments	Duarte Pinto	2018
Blockchain as a PKI for Ownership Control of IoT Devices	Guilherme Pinto	2018
Interoperability In Software Applications For Smart Cities: Towards A Reference Architecture	José Pedro Pinto	2017

The published works during this thesis work gathered a total of 385 citations, which resulted in a h-index of 13, and an i10-index of 14[1]. Parts of this thesis and parallel research were published in 20 different venues/journals. A summary of these venues and journals is given in Table A.2 (p. 322). As a reference, Scimago was considered for journal ranking and CORE for conference ranking[2] — ranking at the time of the most recent submission in each of the given venues/journals. The next sections detail the publications resulting — authors, venues, and abstracts — from this research and others from the author.

[1] Using Google Scholar as a reference metric at the time of writing, in accordance to https://scholar.google.com/citations?user=NYavJ60AAAAJ.

[2] In the case of workshops the CORE ranking corresponding to the main venue at the time of publication is considered.

Table A.2: Summary of publications per venue, with corresponding ranking. A total of 10 papers were published in Q1/A/A* venues, 6 in CORE B venues, 5 in Q4/C venues, and the 8 remaining in unranked venues.

Journal (#4)	Scimago	#
Journal of Computational Science	Q1	1
Communications in Computer and Information Science	Q4	2
Internet of Things: Engineering Cyber Physical Human Systems	—	1
Journal of Information Assurance and Security	—	1
Conference Venue (#13)	CORE	#
Int. Conference on Mobile and Ubiquitous Systems (MobiQuitous)	A	1
Int. Conference on Computational Science (ICCS)	A	4
Int. Symposium on Distributed Simulation and Real Time Applications (DS-RT)	B	1
Conference on Pattern Languages of Programs (PLoP)	B	1
Int. Conf. on Evaluation of Novel Software Approaches to Software Engineering (ENASE)	B	2
Int. Conference on Embedded and Ubiquitous Computing (EUC)	C	1
Int. Conference on the Quality of Information and Communications Technology (QUATIC)	C	1
Int. Conference on Practical Applications of Agents and Multi-Agent Systems (PAAMS)	C	1
European Conference on Pattern Languages of Programs (EuroPLoP)	B	2
Int. Conference on Information Assurance and Security (IAS)	—	2
Int. Smart Cities Conference (ISC2)	—	1
Int. Conference on Ambient Systems, Networks and Technologies (ANT)	—	1
Int. Conference on Internet of Things, Big Data and Security (IoTBDS)	—	1
Conference Venue — Workshop (#4)	CORE	#
Int. Conference on Software Engineering (ICSE)	A*	2
Int. Conference on Model Driven Engineering Languages and Systems (MODELS)	A	1
Int. Conference on Software Testing, Validation and Verification (ICST)	A	1
Int. Conference on the Art, Science, and Engineering of Programming (‹Programming›)	—	1

A.1 Publications Resulting from this Research

Journal Articles

A.1.1 Designing and Constructing Internet-of-Things Systems: An Overview of the Ecosystem

João Pedro Dias, André Restivo, and Hugo Sereno Ferreira. "Designing and Constructing Internet-of-Things Systems: An Overview of the Ecosystem". In: *Internet of Things* (2022)

Abstract: The current complexity of IoT systems and devices is a barrier to reach a healthy ecosystem, mainly due to technological fragmentation and inherent heterogeneity. Meanwhile, the field has scarcely adopted any engineering practices currently employed in other types of large-scale systems. Although many researchers and practitioners are aware of the current state of affairs and strive to address these problems, compromises have been hard to reach,

making them settle for sub-optimal solutions. This paper surveys the current state of the art in designing and constructing IoT systems from the software engineering perspective, without overlooking hardware concerns, revealing current trends and research directions [DRF22].

A.1.2 Managing Non-trivial Internet-of-Things Systems with Conversational Assistants: A Prototype and a Feasibility Experiment

> André Sousa Lago, João Pedro Dias, and Hugo Sereno Ferreira. "Managing non-trivial internet-of-things systems with conversational assistants: A prototype and a feasibility experiment". In: *Journal of Computational Science* 51 (2021), p. 101324

Abstract: Internet-of-Things has reshaped the way people interact with their surroundings and automatize the once manual actions. In a smart home, controlling the Internet-connected lights is as simple as speaking to a nearby conversational assistant. However, specifying interaction rules, such as making the lamp turn on at specific times or when someone enters the space is not a straightforward task. The complexity of doing such increases as the number and variety of devices increases, along with the number of household members. Thus, managing such systems becomes a problem, including finding out why something has happened. This issue lead to the birth of several low-code development solutions that allow users to define rules to their systems, at the cost of discarding the easiness and accessibility of voice interaction. In this paper we extend the previous published work on Jarvis, a conversational interface to manage IoT systems that attempts to address these issues by allowing users to specify time-based rules, use contextual awareness for more natural interactions, provide event management and support causality queries. A proof-of-concept is presented, detailing its architecture and natural language processing capabilities. A feasibility experiment was carried with mostly non-technical participants, providing evidence that Jarvis is intuitive enough to be used by common end-users, with participants showcasing an overall preference by conversational assistants over visual low-code solutions [LDF21].

Conference Publications

A.1.3 Evaluation of IoT Self-healing Mechanisms using Fault-Injection in Message Brokers

> Miguel Duarte, João Pedro Dias, André Restivo, and Hugo Sereno Ferreira. "Evaluation of IoT Self-healing Mechanisms using Fault-Injection in Message Brokers". In: *2022 IEEE/ACM 4th International Workshop on Software Engineering Research Practices for the Internet of Things (SERP4IoT)*. 2022

Abstract: The widespread use of Internet-of-Things (IoT) across different application domains leads to an increased concern regarding their dependability, especially as the number of potentially mission-critical systems becomes considerable. Fault-tolerance has been used to reduce the impact of faults in systems, and their adoption in IoT is becoming a necessity. This work

focuses on how to exercise fault-tolerance mechanisms by deliberately provoking its malfunction. We start by describing a proof-of-concept fault-injection add-on to a commonly used publish/subscribe broker. We then present several experiments mimicking real-world IoT scenarios, focusing on injecting faults in systems with (and without) active self-healing mechanisms and comparing their behavior to the baseline without faults. We observe evidence that fault-injection can be used to (a) exercise in-place fault-tolerance apparatus, and (b) detect when these mechanisms are not performing nominally, providing insights into enhancing in-place fault-tolerance techniques [Dua+22].

A.1.4 A Review on Visual Programming for Distributed Computation in IoT

> Margarida Silva, João Pedro Dias, André Restivo, and Hugo Sereno Ferreira. "A Review on Visual Programming for Distributed Computation in IoT". in: *Proceedings of the 21st International Conference on Computational Science (ICCS)*. Springer, 2021

Abstract: Internet-of-Things (IoT) systems are considered one of the most notable examples of complex, large-scale systems. Some authors have proposed visual programming (VP) approaches to address part of their inherent complexity. However, in most of these approaches, the orchestration of devices and system components is still dependent on a centralized unit, preventing higher degrees of dependability. In this work, we perform a systematic literature review (SLR) of the current approaches that provide visual and decentralized orchestration to define and operate IoT systems, reflecting upon a total of 29 proposals. We provide an in-depth discussion of these works and find out that only four of them attempt to tackle this issue as a whole, although still leaving a set of open research challenges. Finally, we argue that addressing these challenges could make IoT systems more fault-tolerant, with an impact on their dependability, performance, and scalability [Sil+21].

A.1.5 Programming IoT-Spaces: A User-Survey on Home Automation Rules

> Danny Soares, João Pedro Dias, André Restivo, and Hugo Sereno Ferreira. "Programming IoT-spaces: A User-Survey on Home Automation Rules". In: *Proceedings of the 21st International Conference on Computational Science (ICCS)*. Springer, 2021

Abstract: The Internet-of-Things (IoT) has transformed everyday manual tasks into digital and automatable ones, giving way to the birth of several end-user development solutions that attempt to ease the task of configuring and automating IoT systems without requiring prior technical knowledge. While some studies reflect on the automation rules that end-users choose to program into their spaces, they are limited by the number of devices and possible rules that the tool under study supports. There is a lack of systematic research on (1) the automation rules that users wish to configure on their homes, (2) the different ways users state their intents,

and (3) the complexity of the rules themselves—without the limitations imposed by specific IoT devices systems and end-user development tools. This paper surveyed twenty participants about home automation rules given a standard house model and device's list, without limiting their creativity and resulting automation complexity. We analyzed and systematized the collected 177 scenarios into seven different interaction categories, representing the most common smart home interactions [Soa+21].

A.1.6 Empowering Visual Internet-of-Things Mashups with Self-Healing Capabilities

> Joao Pedro Dias, André Restivo, and Hugo Sereno Ferreira. "Empowering Visual Internet-of-Things Mashups with Self-Healing Capabilities". In: *2021 IEEE/ACM 3rd International Workshop on Software Engineering Research Practices for the Internet of Things (SERP4IoT)*. 2021

Abstract: Internet-of-Things (IoT) systems have spread among different application domains, from home automation to industrial manufacturing processes. The rushed development by competing vendors to meet the market demand of IoT solutions, the lack of interoperability standards, and the overall lack of a defined set of best practices have resulted in a highly complex, heterogeneous, and frangible ecosystem. Several works have been pushing towards visual programming solutions to abstract the underlying complexity and help humans reason about it. As these solutions begin to meet widespread adoption, their building blocks usually do not consider reliability issues. Node-RED, being one of the most popular tools, also lacks such mechanisms, either built-in or via extensions. In this work we present SHEN (Self-Healing Extensions for Node-RED) which provides 17 nodes that collectively enable the implementation of self-healing strategies within this visual framework. We proceed to demonstrate the feasibility and effectiveness of the approach using real devices and fault injection techniques [DRF21].

A.1.7 Visually-defined Real-Time Orchestration of IoT Systems

> Margarida Silva, João Pedro Dias, André Restivo, and Hugo Sereno Ferreira. "Visually-defined Real-Time Orchestration of IoT Systems". In: *Proceedings of the 17th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*. MOBIQUITOUS 2020. Online: Association for Computing Machinery, 2020

Abstract: In this work, we propose a method for extending Node-RED to allow the automatic decomposition and partitioning of the system towards higher decentralization. We provide a custom firmware for constrained devices to expose their resources, as well as new nodes and modifications in the Node-RED engine that allow automatic orchestration of tasks. The firmware is responsible for low-level management of health and capabilities, as well as executing MicroPython scripts on demand. Node-RED then takes advantage of this firmware by (1) providing a device registry allowing devices to announce themselves, (2) generating MicroPython code from dynamic analysis of flow and nodes, and (3) automatically (re-)assigning nodes

to devices based on pre-specified properties and priorities. A mechanism to automatically detect abnormal runtime conditions and provide dynamic self-adaptation was also explored. Our solution was tested using synthetic home automation scenarios, where several experiments were conducted with both virtual and physical devices. We then exhaustively measured each scenario to allow further understanding of our proposal and how it impacts the system's resiliency, efficiency, and elasticity [Sil+20].

A.1.8 A Pattern-Language for Self-Healing Internet-of-Things Systems

> Joao Pedro Dias, Tiago Boldt Sousa, André Restivo, and Hugo Sereno Ferreira. "A Pattern-Language for Self-Healing Internet-of-Things Systems". In: *Proceedings of the 25th European Conference on Pattern Languages of Programs*. EuroPLop '20. Irsee, Germany: Association for Computing Machinery, 2020

Abstract: Internet-of-Things systems are assemblies of highly-distributed and heterogeneous parts that, in orchestration, work to provide valuable services to end-users in many scenarios. These systems depend on the correct operation of sensors, actuators, and third-party services, and the failure of a single one can hinder the proper functioning of the whole system, making error detection and recovery of paramount importance, but often overlooked. By drawing inspiration from other research areas, such as cloud, embedded, and mission-critical systems, we present a set of patterns for self-healing IoT systems. We discuss how their implementation can improve system reliability by providing error detection, error recovery, and health mechanisms maintenance [Dia+20a].

A.1.9 Real-time Feedback in Node-RED for IoT Development: An Empirical Study

> D. Torres, J. P. Dias, A. Restivo, and H. S. Ferreira. "Real-time Feedback in Node-RED for IoT Development: An Empirical Study". In: *2020 IEEE/ACM 24th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. 2020, pp. 1–8

Abstract: The continuous spreading of the Internet-of-Things across application domains, aided by the continuous growth on the number of devices and systems that are Internet-connected, created both a rise in the complexity of these systems and made noticeable a lack of human resources with the expertise to design, develop and maintain them. Recent works try to mitigate these issues by creating solutions that abstract the complexity of the systems, such as using visual programming languages. Node-RED, as one of the most common solutions for the visual development IoT systems, stills has several limitations, such as the lack of observability and inadequate debugging mechanisms. In this work, we address some of these limitations by enhancing Node-RED with new features that improve the user's system development, debugging, and understanding tasks. We proceed to empirically evaluate the impact of these enhancements, concluding that, overall, such enhancements reduce the development time and the number of failed attempts to deploy the system [Tor+20].

A.1.10 Visual Self-Healing Modelling for Reliable Internet-of-Things Systems

João Pedro Dias, Bruno Lima, João Pascoal Faria, André Restivo, and Hugo Sereno Ferreira. "Visual Self-Healing Modelling for Reliable Internet-of-Things Systems". In: *Proceedings of the 20th International Conference on Computational Science (ICCS)*. Springer, 2020, pp. 27–36

Abstract: Internet-of-Things systems are comprised of highly heterogeneous architectures, where different protocols, application stacks, integration services, and orchestration engines co-exist. As they permeate our everyday lives, more of them become safety-critical, increasing the need for making them testable and fault-tolerant, with minimal human intervention. In this paper, we present a set of self-healing extensions for Node-RED, a popular visual programming solution for IoT systems. These extensions add runtime verification mechanisms and self-healing capabilities via new reusable nodes, some of them leveraging meta-programming techniques. With them, we were able to implement self-modification of flows, empowering the system with self-monitoring and self-testing capabilities, that search for malfunctions, and take subsequent actions towards the maintenance of health and recovery. We tested these mechanisms on a set of scenarios using a live physical setup that we called SmartLab. Our results indicate that this approach can improve a system's reliability and dependability, both by being able to detect failing conditions, as well as reacting to them by self-modifying flows, or triggering countermeasures [Dia+20b].

A.1.11 Conversational Interface for Managing Non-trivial Internet-of-Things Systems

João Pedro Dias, André Lago, and Hugo Sereno Ferreira. "Conversational Interface for Managing Non-Trivial Internet-of-Things Systems". In: *Proceedings of the 20th International Conference on Computational Science (ICCS)*. Springer, 2020, pp. 27–36

Abstract: Internet-of-Things has reshaped the way people interact with their surroundings. In a smart home, controlling the lights is as simple as speaking to a conversational assistant since everything is now Internet-connected. But despite their pervasiveness, most of the existing IoT systems provide limited out-of-the-box customization capabilities. Several solutions try to attain this issue leveraging end-user programming features that allow users to define rules to their systems, at the cost of discarding the easiness of voice interaction. However, as the number of devices increases, along with the number of household members, the complexity of managing such systems becomes a problem, including finding out why something has happened. In this work we present Jarvis, a conversational interface to manage IoT systems that attempts to address these issues by allowing users to specify time-based rules, use contextual awareness for more natural interactions, provide event management and support causality queries. A proof-of-concept was used to carry out a quasi-experiment with non-technical participants that provides evidence that such approach is intuitive enough to be used by common end-users [DLF20].

A.1.12 Testing and Deployment Patterns for the Internet-of-Things

> Joao Pedro Dias, Hugo Sereno Ferreira, and Tiago Boldt Sousa. "Testing and Deployment Patterns for the Internet-of-Things". In: *Proceedings of the 24th European Conference on Pattern Languages of Programs*. EuroPLop '19. Irsee, Germany: Association for Computing Machinery, 2019

Abstract: As with every software, Internet-of-Things (IoT) systems have their own life-cycle, from conception to construction, deployment, and operation. However, the testing requirements from these systems are slightly different due to their inherent coupling with hardware and human factors. Hence, the procedure of delivering new software versions in a continuous integration/delivery fashion must be adopted. Based on existent solutions (and inspired in other closely-related domains), we describe two common strategies that developers can use for testing IoT systems, (1) Testbed and (2) Simulation-based Testing, as well as one recurrent solution for its deployment (3) Middleman Update [DFS19].

A.1.13 Blockchain-based PKI for Crowdsourced IoT Sensor Information

> Guilherme Vieira Pinto, João Pedro Dias, and Hugo Sereno Ferreira. "Blockchain-Based PKI for Crowdsourced IoT Sensor Information". In: *Proceedings of the Tenth International Conference on Soft Computing and Pattern Recognition (SoCPaR 2018)*. Ed. by Ana Maria Madureira, Ajith Abraham, Niketa Gandhi, Catarina Silva, and Mário Antunes. Cham: Springer International Publishing, 2020, pp. 248–257

Abstract: The Internet of Things is progressively getting broader, evolving its scope while creating new markets and adding more to the existing ones. However, both generation and analysis of large amounts of data, which are integral to this concept, may require the proper protection and privacy-awareness of some sensitive information. In order to control the access to this data, allowing devices to verify the reliability of their own interactions with other endpoints of the network is a crucial step to ensure this required safeness. Through the implementation of a blockchain-based Public Key Infrastructure connected to the Keybase platform, it is possible to achieve a simple protocol that binds devices' public keys to their owner accounts, which are respectively supported by identity proofs. The records of this blockchain represent digital signatures performed by this Keybase users on their respective devices' public keys, claiming their ownership. Resorting to this distributed and decentralized PKI, any device is able to autonomously verify the entity in control of a certain node of the network and prevent future interactions with unverified parties [PDS20].

A.1.14 Dynamic Allocation of Serverless Functions in IoT Environments

> D. Pinto, João Pedro Dias, and Hugo Sereno Ferreira. "Dynamic Allocation of Serverless Functions in IoT Environments". In: *2018 IEEE 16th International Conference on Embedded and Ubiquitous Computing (EUC)*. Oct. 2018, pp. 1–8

Abstract: The IoT area has grown significantly in the last few years and is expected to reach a gigantic amount of 50 billion devices by 2020. The appearance of serverless architectures, specifically highlighting FaaS, raises the question of the suitability of using them in IoT environments. Combining IoT with a serverless architectural design can effective when trying to make use of local processing power that exists in a local network of IoT devices and creating a fog layer that leverages computational capabilities that are closer to the end-user. In this approach, which is placed between the device and the serverless function, when a device requests for the execution of a serverless function will decide based on previous metrics of execution if the serverless function should be executed locally, in the fog layer of a local network of IoT devices, or if it should be executed remotely, in one of the available cloud servers. Therefore, this approach allows dynamically allocating functions to the most suitable layer [PDS18].

A.1.15 A Reactive and Model-based Approach for Developing Internet-of-Things Systems

> João Pedro Dias, João Pascoal Faria, and Hugo Sereno Ferreira. "A Reactive and Model-Based Approach for Developing Internet-of-Things Systems". In: *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*. Sept. 2018, pp. 276–281

Abstract: Software has a longstanding association with a state of crisis considering its success rate. The explosion of Internet-connected devices - Internet-of-Things - adds to the complexity of software systems. The particular characteristics of these systems, such as its large-scale and heterogeneity, pose increasingly new challenges. Model-based approaches have been widely used as a mechanism to abstract low-level programming details and processes. By using such approaches, and leveraging concepts as reactive design, visual notations, and live programming, we believe to be able to reduce the complexity of creating, operate/monitor and evolve such systems. The main objective of this Ph.D. is to delve into the software engineering practices for developing IoT systems and systems of systems, exploiting models as a suitable abstraction, expecting to reduce the complexity of managing most of the software development lifecycle that targets IoT systems and to develop the prototype that will aid on the validation of such approach [DFF18].

A.1.16 A Brief Overview of Existing Tools for Testing the Internet-of-Things

> João Pedro Dias, F. Couto, A. C. R. Paiva, and Hugo Sereno Ferreira. "A Brief Overview of Existing Tools for Testing the Internet-of-Things". In: *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. Apr. 2018, pp. 104–109

Abstract: Systems are error-prone. Big systems have lots of errors. The Internet-of-Things poses us one of the biggest and widespread systems, where errors directly impact people's lives. Testing and validating is how one deals with errors; but testing and validating a planetary-scale, heterogeneous, and evergrowing ecosystem has its own challenges and idiosyncrasies. As of today, the solutions available for testing these systems are insufficient and fragmentary. In this paper we provide an overview on test approaches, tools and methodologies for the Internet-of-Things, its software and its devices. Our conclusion is that we are still lagging behind on the best practices and lessons learned from the Software Engineering community in the past decades [Dia+18].

A.1.17 Patterns for Things that Fail

> Antonio Ramadas, Gil Domingues, Joao Pedro Dias, Ademar Aguiar, and Hugo Sereno Ferreira. "Patterns for Things that Fail". In: *Proceedings of the 24th Conference on Pattern Languages of Programs*. PLoP '17. ACM - Association for Computing Machinery, 2017

Abstract: Internet of Things is a paradigm that empowers the Internet-connected heterogeneous devices alongside with their capabilities to sense the physical world and act on it. Internet of Things has a wide range of application in a variety of areas and contexts, such as smart spaces like smart homes and smart cities. These systems have to deal with device-related problems, such as heterogeneity, fault-tolerance, privacy and security. This paper addresses some patterns about some recurring problems when designing and implementing Internet of Things systems. More concretely, the patterns address how to deal with highly-changeable, error-prone and failing devices and their networks. Device Registry demonstrates how to deal with devices that can change over time. Device Raw Data Collector explains how to check the health of heterogeneous devices by a constant collection of raw device data. Device Error Data Supervisor shows how to leverage the continuous data-flow coming from devices to enable error detection and, consequently, processing and handling those errors. Lastly, Predictive Device Monitor show how to pro-actively predict devices operation behavior enabling maintenance actions [Ram+17].

A.2 Other Publications from the Author

Journal Articles

A.2.1 Experimenting with Liveness in Cloud Infrastructure Management

> Pedro Lourenço, João Pedro Dias, Ademar Aguiar, Hugo Sereno Ferreira, and André Restivo. "Experimenting with Liveness in Cloud Infrastructure Management". In: *Communications in Computer and Information Science* 1172 (2020)

Abstract: Cloud computing has been playing a significant role in the provisioning of services over the Internet since its birth. However, developers still face several challenges limiting its full potential. The difficulties are mostly due to the large, ever-growing, and ever-changing catalog of services offered by cloud providers. As a consequence, developers must deal with different cloud services in their systems; each managed almost individually and continually growing in complexity. This heterogeneity may limit the view developers have over their system architectures and make the task of managing these resources more complex. This work explores the use of liveness as a way to shorten the feedback loop between developers and their systems in an interactive and immersive way, as they develop and integrate cloud-based systems. The designed approach allows real-time visualization of cloud infrastructures using a visual city metaphor. To assert the viability of this approach, the authors conceived a proof-of-concept and carried on experiments with developers to assess its feasibility [Lou+20].

A.2.2 Live Software Development Environment using Virtual Reality: a Prototype and Experiment

> Diogo Amaral, Gil Domingues, João Pedro Dias, Hugo Sereno Ferreira, Ademar Aguiar, Rui Nóbrega, and Filipe Figueiredo Correia. "Live Software Development Environment using Virtual Reality: a Prototype and Experiment". In: *Communications in Computer and Information Science* 1172 (2020)

Abstract: Successful software systems tend to grow considerably, ending up suffering from essential complexity, and very hard to understand as a whole. Software visualization techniques have been explored as one approach to ease software understanding. This work presents a novel approach and environment for software development that explores the use of liveness and virtual reality (VR) as a way to shorten the feedback loop between developers and their software systems in an interactive and immersive way. As a proof-of-concept, the authors developed a prototype that uses a visual city metaphor and allows developers to visit and dive into the system, in a live way. To assess the usability and viability of the approach, the authors carried on experiments to evaluate the effectiveness of the approach, and how to best support a live approach for software development [Ama+20].

A.2.3 A Blockchain-based Approach for Access Control in eHealth Scenarios

> João Pedro Dias, Ângelo Martins, and Hugo Sereno Ferreira. "A Blockchain-based Approach for Access Control in eHealth Scenarios". In: *Journal of Information Assurance and Security* 13 (4 2018), pp. 125–136

Access control is a crucial part of a system's secu-rity, restricting what actions users can perform on resources.Therefore, access control is a core component when dealing witheHealth data and resources, discriminating which is availablefor a certain party. We consider that current systems that at-tempt to assure the share of policies between facilities are proneto system's and network's faults and do not assure the integri-ty of policies life-cycle. By approaching this problem with ablockchain where the operations are stored as transactions, wecan ensure that the different facilities have knowledge about allthe parts that can act over the eHealth resources while main-taining integrity, auditability, and authenticity [DMF18].

Conference Publications

A.2.4 An empirical study on visual programming docker compose configurations

> Bruno Piedade, Joao Pedro Dias, and Filipe F. Correia. "An Empirical Study on Visual Programming Docker Compose Configurations". In: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. MODELS '20. Virtual Event, Canada: Association for Computing Machinery, 2020

Abstract: Infrastructure-as-Code tools, such as Docker and Docker Compose, play a crucial role in the development and orchestration of cloud-native and at-scale software. However, as IaC relies mostly on the development of text-only specifications, these are prone to misconfigurations and hard to debug. Several works suggest the use of models as a way to abstract their complexity, and some point to the use of visual metaphors. Yet, few empirical studies exist in this domain. We propose a visual programming notation and environment for specifying Docker Compose configurations and proceed to empirically validate its merits when compared with the standard text-only specification. The goal of this work is to produce evidence of the impact that visual approaches may have on the development of IaC. We observe that the use of our solution reduced the development time and error proneness, primarily for configurations definition activities. We also observed a preference for the approach in terms of ease of use, a positive sentiment of its usefulness and intention to use [PDC20].

A.2.5 Live Software Development: Tightening the Feedback Loops

> Ademar Aguiar, André Restivo, Filipe Figueiredo Correia, Hugo Sereno Ferreira, and Joao Pedro Dias. "Live Software Development: Tightening the Feedback Loops". In: *Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming*. Programming '19. Genova, Italy: ACM, 2019, 22:1–22:6

Abstract: Live Programming is an idea pioneered by programming environments from the earliest days of computing, such as those for Lisp and Smalltalk. One thing they had in common is liveness: an always accessible evaluation and nearly instantaneous feedback, usually focused on coding activities. In this paper, we argue for Live Software Development (LiveSD), bringing liveness to software development activities beyond coding, to make software easier to visualize, simpler to understand, and faster to evolve. Multiple challenges may vary with the activity and application domain. Research on this topic needs to consider the more important liveness gaps in software development, which representations and abstractions better support developers, and which tools are needed to support it [Agu+19].

A.2.6 Live Software Development Environment for Java using Virtual Reality

> Diogo Amaral, Gil Domingues, João Pedro Dias, Hugo Sereno Ferreira, Ademar Aguiar, and Rui Nóbrega. "Live Software Development Environment for Java using Virtual Reality". In: *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering — Volume 1: ENASE*. INSTICC. SciTePress, 2019, pp. 37–46

Abstract: Any software system that has a considerable growing number of features will suffer from essential complexity, which makes the understanding of the software artifacts increasingly costly and time-consuming. A common approach for reducing the software understanding complexity is to use software visualizations techniques. There are already several approaches for visualizing software, as well as for extracting the information needed for those visualizations. This paper presents a novel approach to tackle the software complexity, delving into the common approaches for extracting information about software artifacts and common software visualization metaphors, allowing users to dive into the software system in a live way using virtual reality (VR). Experiments were carried out in order to validate the correct extraction of metadata from the software artifact and the corresponding VR visualization. With this work, we intend to present a starting point towards a Live Software Development approach [Ama+19].

A.2.7 CloudCity: A Live Environment for the Management of Cloud Infrastructures

> Pedro Lourenço, João Pedro Dias, Ademar Aguiar, and Hugo Sereno Ferreira. "CloudCity: A Live Environment for the Management of Cloud Infrastructures". In: *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering — Volume 1: ENASE*. INSTICC. SciTePress, 2019, pp. 27–36

Abstract: Cloud computing has emerged as the de facto approach for providing services over the Internet. Although having increased popularity, challenges arise in the management of such environments, especially when the cloud service providers are constantly evolving their services and technology stack in order to maintain position in a demanding market. This usually leads to a combination of different services, each one managed individually, not providing a big picture of the architecture. In essence, the end state will be too many resources under management in an overwhelming heterogeneous environment. An infrastructure that has considerable growth will not be able to avoid its increasing complexity. Thus, this papers introduces liveness as an attempt to increase the feedback-loop to the developer in the management of cloud architectures. This aims to ease the process of developing and integrating cloud-based systems, by giving the possibility to understand the system and manage it in an inte ractive and immersive experience, thus perceiving how the infrastructure reacts to change. This approach allows the real-time visualization of a cloud infrastructure composed of a set of Amazon Web Services resources, using visual city metaphors [Lou+19].

A.2.8 A Blockchain-based Scheme for Access Control in e-Health Scenarios

> João Pedro Dias, Hugo Sereno Ferreira, and Ângelo Martins. "A Blockchain-Based Scheme for Access Control in e-Health Scenarios". In: *Proceedings of the Tenth International Conference on Soft Computing and Pattern Recognition (SoCPaR 2018)*. Ed. by Ana Maria Madureira, Ajith Abraham, Niketa Gandhi, Catarina Silva, and Mário Antunes. Cham: Springer International Publishing, 2020, pp. 238–247

Abstract: Access control is a crucial part of a system's security, restricting what actions users can perform on resources. Therefore, access control is a core component when dealing with e-Health data and resources, discriminating which is available for a certain party. We consider that current systems that attempt to assure the share of policies between facilities are mostly centralized, being prone to system's and network's faults and do not assure the integrity of policies lifecycle. Using a blockchain as store system for access policies we are able to ensure that the different entities have knowledge about the policies in place while maintaining a record of all permission requests, thus assuring integrity, auditability and authenticity [DSM20].

A.2.9 Growing Smart Cities on an Open-Data-Centric Cyber-Physical Platform

> José Pedro Pinto, João Pedro Dias, and R. J. F. Rossetti. "Growing Smart Cities on an Open-Data-Centric Cyber-Physical Platform". In: *2018 IEEE International Smart Cities Conference (ISC2)*. Sept. 2018, pp. 1–6

Abstract: Considering an environment that consists of several services, applications and platforms, each present entity produces a certain amount of data. With so many sources of data,

there are a number of things bound to exist: different formats of information, redundancy and no consistent standards of information. In environments as these, the collaboration between different entities creates an opportunity for innovation, where data interoperability allows for the re-use of information, the possibility of different services taking advantage of other third-party sources and the development of new businesses from existing information. This, however, is only possible if there is some sort of interoperability between the data, a way for it to be transmitted from entity to entity, always with the possibility of creating value with Its manipulation and consumption. This paper exposes the work done in the development of a platform focused on data, looking into its forms of representation and how to solve the problems caused by the ever existing necessity of data interoperability between systems. The possibility for maintaining and creating Open Data Ecosystems is also analysed in the scope of the proposed platform [PDR18].

A.2.10 Towards a Framework for Agent-based Simulation of User behavior in E-Commerce Context

> Duarte Duarte, Hugo Sereno Ferreira, João Pedro Dias, and Zafeiris Kokkinogenis. "Towards a Framework for Agent-Based Simulation of User behavior in E-Commerce Context". In: *Trends in Cyber-Physical Multi-Agent Systems. The PAAMS Collection - 15th International Conference, PAAMS 2017*. Cham: Springer International Publishing, 2018, pp. 30–38

Abstract: In order to increase sales and profits, it is common that e-commerce website owners resort to several marketing and advertising techniques, attempting to influence user actions. Summarizing and analysing user behavior is a complex task since it is hard to extrapolate patterns that never occurred before and the causality aspects of the system are not usually taken into consideration. There has been studies about characterizing user behavior and interactions in e-commerce websites that could be used to improve this process. This paper presents an agent-based framework for simulating models of user behavior created through data mining processes within an e-commerce context. The purpose of framework is to study the reaction of user to stimuli that influence their actions while navigating the website. Furthermore a scalability analysis is performed on a case-study [Dua+18].

A.2.11 A Hands-on Approach on Botnets for Behavior Exploration

> João Pedro Dias, José Pedro Pinto, and José Magalhães Cruz. "A Hands-on Approach on Botnets for Behavior Exploration". In: *Proceedings of the 2nd International Conference on Internet of Things, Big Data and Security*. SCITEPRESS - Science and Technology Publications, 2017, pp. 463–469

Abstract: A botnet consists of a network of computers that run a special software that allows a third-party to remotely control them. This characteristic presents a major issue regarding

security in the Internet. Although common malicious software infect the network with almost immediate visible consequences, there are cases where that software acts stealthy without direct visible effects on the host machine. This is the normal case of botnets. However, not always the bot software is created and used for illicit purposes. There is a need for further exploring the concepts behind botnets and network security. For this purpose, this paper presents and discusses an educational tool that consists of an open-source botnet software kit with built-in functionalities. The tool enables anyone with some computer technical knowledge, to experiment and find out how botnets work and can be changed and adapted to a variety of useful applications, such as introducing and exemplifying security and distributed systems' concepts [DPC17].

A.2.12 Automating the Extraction of Static Content and Dynamic behavior from e-Commerce Websites

> Joao Pedro Dias and Hugo Sereno Ferreira. "Automating the Extraction of Static Content and Dynamic behavior from e-Commerce Websites". In: *Procedia Computer Science* 109 (2017). 8th International Conference on Ambient Systems, Networks and Technologies, ANT-2017 and the 7th International Conference on Sustainable Energy Information Technology, SEIT 2017, 16-19 May 2017, Madeira, Portugal, pp. 297–304

Abstract: E-commerce website owners rely heavily on analysing and summarising the behavior of costumers, making efforts to influence user actions and optimize success metrics. Machine learning and data mining techniques have been applied in this field, greatly influencing the Internet marketing activities. When faced with a new e-commerce website, the data scientist starts a process of collecting real-time and historical data about it, analysing and transforming this data in order to get a grasp into the website and its users. Data scientists commonly resort to tracking domain-specific events, requiring code modification of the web pages. This paper proposes an alternative approach to retrieve information from a given e-commerce website, collecting data from the site's structure, retrieving semantic information in predefined locations and analysing user's access logs, thus enabling the development of accurate models for predicting users' future behavior. This is accomplished by the application of a web mining process, comprehending the site's structure, content and usage in a pipeline, resulting in a web graph of the website, complemented with a categorization of each page and the website's archetypical user profiles [DF17].

B | Replication Packages

A replication package containing both the source code and experimental materials for several parts of this thesis is available as open-source as listed.

B.1 Home Automation Survey

Replication package of the home automation survey carried in min-2020 with 20 participants, as discussed in Chapter 5 (p. 123). It contains the raw collected data of the 177 submitted scenarios, along with the list of devices, house 2D/3D models, and processed data (*e.g.*, isometric representation of the house).

Zenodo https://doi.org/10.5281/zenodo.4531395

B.2 Self-Healing Extensions for Node-RED

A collection of nodes (extensions) for making Node-RED more resilient by adding self-healing capabilities. It contains the source code of 17 nodes, along with supporting documentation. The corresponding work is presented and discussed in Chapter 12 (p. 218).

GitHub https://github.com/jpdias/node-red-contrib-self-healing

NPM https://www.npmjs.com/package/node-red-contrib-self-healing

Zenodo https://doi.org/10.5281/zenodo.4448164

B.3 Real-Time Feedback on Node-RED

Capacitating Agile Users with Live Debugging Resources On Node-RED (CAULDRON) is a modified version of Node-RED, as discussed in Chapter 13 (p. 246). The package contains both the source code and supporting documentation.

GitHub https://github.com/SIGNEXT/Node-RED-CAULDRON

Zenodo http://doi.org/10.5281/zenodo.5724488

B.4 Jarvis Voice Assistant

Package with the source code and supporting documentation for setting up and running Jarvis, a chat-bot (conversational assistant) for interaction with Internet-of-Things systems. The corresponding work is presented and discussed in Chapter 14 (p. 258).

GitHub https://github.com/andrelago13/jarvis/

Zenodo http://doi.org/10.5281/zenodo.3741953

B.5 Node-RED Distributed Orchestration

Decentralized Computation extensions for Node-RED is a set of *nodes* that allow to distribute flows *nodes* across available computational resources in the network. It is accompanied by an Espressif ESP-compatible MicroPython firmware which is compatible with the orchestration procedure and is able to run the generated Python scripts. The corresponding work is presented and discussed in Chapter 11 (p. 199).

GitHub https://github.com/SIGNEXT/node-red-contrib-decentralized-computation

NPM https://www.npmjs.com/package/node-red-contrib-decentralized-computation

Zenodo https://doi.org/10.5281/zenodo.5724435

B.6 Serverless for IoT Tasks

Replication package for serverless computing across tiers of an IoT system. It contains the source code, supporting documentation, and usage examples. The corresponding work is presented and discussed in Chapter 10 (p. 184).

GitHub https://github.com/jpdias/serverless-iot/

Zenodo https://doi.org/10.5281/zenodo.5724372

B.7 Fault-injection on Publisher/Subscriber IoT Systems

Replication package for fault-injection on publisher/subscriber IoT systems. It contains the source code, supporting documentation, usage examples and experimental data. The corresponding work is presented and discussed in Chapter 12 (p. 218).

GitHub https://github.com/SIGNEXT/instrumentable-aedes

Zenodo https://doi.org/10.5281/zenodo.5724429

C | Self-Healing Algorithms

This appendix contains some extra algorithms that correspond to implementations of the self-healing patterns presented in Chapter 6 (p. 144).

Algorithm C.1: Pseudo-code for the `kalman-filter` node.

Input : sensorReading
Output : sensorReading
/ The present algorithm is a simplification of the Kalman Filter which assumes that the state vector equals 1, control vector equals 0 and measurement vector equals 1 as per the work of Bulten* et al. *[BRH16]. */*

```
1  init
2      config: {
3          R: ℝ>0,                                  // noise power desirable
4          Q: ℝ>0,                                  // noise power estimated
5      }
6      covariance: ℝ,
7      measurement: ℝ
8  onInput
9      if not measurement then
10         covariance ← Q
11         measurement ← sensorReading
12     else
13         uncertainty ← covariance + R
14         kalmanGain ← uncertainty / (uncertainty+Q)
15         measurement ← kalmanGain × sensorReading
16         covariance ← uncertainty - (kalmanGain × uncertainty)
17         return measurement
```

Algorithm C.2: Pseudo-code for the `compensate` node.

```
Input  : reading
Output : reading
1  init
2      config: {
3          historySize: ℤ_{>1}
4          interval: ℝ_{>0}
5          strategy ← s ∈ {avg, max, min, last, nil, …}
6      }
7      timer ← newTimer(config.interval)
8      sensorHistory ← [ ]
9  onInput
10     if |sensorHistory| ≥ config.historySize then
11         delete(sensorHistory_0)
12     sensorHistory ← sensorHistory ++ reading
13     timer.start()                                  // (Re)start timer
14     return reading
15 onTimeout
16     reading ← config.strategy(readingHist)
17     if |sensorHistory| ≥ config.historySize then
18         delete(sensorHistory_0)
19     sensorHistory ← sensorHistory ++ reading
20     timer.start()
21     return reading
```

Algorithm C.3: Pseudo-code for the `threshold-check` node.

```
Input  : reading
Output : ⟨reading, error⟩                  // egress ignores '_' messages
1  init
2      config: {
3          lowThreshold: ℝ
4          highThreshold: ℝ
5          inv lowThreshold ≤ highThreshold
6      }
7  onInput
8      if reading ∈ [config.lowThreshold, config.highThreshold] then
9          return ⟨reading, _⟩
10     else
11         return ⟨_, error⟩
```

Algorithm C.4: Pseudo-code for the `replication-voter` node with timeout.

Input : sensorReading
Output : ⟨sensorReading, error⟩

```
1  init
2      config { minConsensus: ℤ_{>0}}
3      timer ← newTimer(interval: ℝ_{>0})
4      sensorReads ← [ ]
5  onInput
6      sensorReads ← sensorReads ++ sensorReading
7      if not timer.started() then
8          timer.start()
9  onTimeout
10     h(x) ← Σ_{v∈sensorReads}[v = x]                    // occurrences of x
11     ⟨majority, voters⟩ ← ⟨arg max_x h(x), max_x h(x)⟩
12     sensorReads ← [ ]
13     if voters ≥ config.minConsensus then
14         return ⟨majority, NIL⟩
15     else
16         return ⟨NIL, error⟩
```

Algorithm C.5: Pseudo-code for the `replication-voter` node without timeout.

Input : sensorReading
Output : ⟨sensorReading, error⟩

```
1  init
2      config: {
3          numberOfReadings: ℤ_{>0}
4          minConsensus: ℤ_{>0}
5      }
6      sensorReads ← [ ]
7  onInput
8      sensorReads ← sensorReads ++ sensorReading
9      if config.numberOfReadings == |sensorReads| then
10         h(x) ← Σ_{v∈sensorReads}[v = x]                // occurrences of x
11         ⟨majority, voters⟩ ← ⟨arg max_x h(x), max_x h(x)⟩
12         sensorReads ← [ ]
13         if voters ≥ config.minConsensus then
14             return ⟨majority, NIL⟩
15         else
16             return ⟨NIL, error⟩
```

Algorithm C.6: Pseudo-code for the `flow-control` node.

Input : ⟨flowID, statusBool⟩
Output : ⟨success, error⟩

```
1  init
2      config: {
3          ip: ipAddress
4          port: [1 . . . 65535]
5      }
6      httpObj ← ⟨config.ip, config.port⟩                    // can be localhost
7      flowConfig: JSONObject
8  onInput
9      flowConfig ← httpObj.get("/flow/flowID")
10     flowConfig.disabled ← not statusBool
11     httpCode ← httpObj.put("/flow/flowID", flowConfig)
12     if httpCode = "200 OK" then
13         return ⟨success, Nil⟩
14     else
15         return ⟨Nil, error⟩
```

Algorithm C.7: Pseudo-code for the `device-registry` node.

Input : _
Output : deviceList
/ Maintains a registry of all devices in the network with a last seen timestamp.* `Obs` *are observations with extra info if available,* e.g.*, manufacturer. */*

```
1  init
2      devices: {id → ⟨ip, lastSeen, obs⟩}
3  onInput
4      lastSeen ← time.now()
5      id ← md5(ip, obs)
6      devices ← devices ++ id → ⟨ip, lastSeen, obs⟩
7      return devices
```

Algorithm C.8: Pseudo-code for the `redundancy-manager` node.

```
   Input  : pingMsg
   Output : ⟨pingMsg, hosts, selfIsMaster⟩
1  init
2      config: {pingInterval: ℤ>0
3               timeout: ℝ>0}
4      myIP ← OS.getIpAddress()
5      timer ← newTimer(config.pingInterval)
6      selfIsMaster ← FALSE
7      hostRegistry: {
8          host: ipAddress → {
9                          master: boolean,
10                         lastSeen: timestamp
11             }
12     }
13     broadcast ← {master ← selfIsMaster,                  // first host announcement
14                  hostIP ← myIP }
15     return ⟨broadcast, _, _ ⟩
16 onInput
17     if not pingMsg.hostIP = myIP then
18         hostRegistry ∪ { pingMsg.hostIP → {
19                         master ← pingMsg.master,
20                         lastSeen ← time.now() } }
21     if not timer.started() then
22         timer.start()
23 onTimeout
24     hosts ← hostRegistry.keys() if |hosts| = 0 and not selfIsMaster then
25         selfIsMaster ← TRUE
26     else
27         forall host in hosts do
28             if hostRegistry_host.lastSeen - time.now() ≥ config.timeout then
29                 hostRegistry \ host
30                 if hostRegistry[host].master then
31                     hostIp ← getIpWithMaxLastOctect(hosts)
32                     selfIsMaster ← hostIp = myIp
33     pingMsg ← {
34                 master ← selfIsMaster,                   // host alive message
35                 hostIP ← myIP }
36     return ⟨pingMsg, hosts, selfIsMaster ⟩
```

Algorithm C.9: Pseudo-code for the `checkpoint` node.

```
Input  : message
Output : message
1  init
2      config: {timeToLive: ℝ>0}
3      store: {timestamp, lastMessage}
4      timestamp ← store.timestamp or NIL
5      lastMessage ← store.lastMessage or NIL
6      if lastMessage ≠ NIL then
7          aliveTime ← time.now() − store.timestamp
8          if aliveTime ≤ config.timeToLive then
9              retained ← lastMessage
10             lastMessage ← NIL
11             return retained
12 onInput
13     store.timestamp ← time.now()                    // store is persistent
14     store.lastMessage ← message
15     return message
```

Algorithm C.10: Pseudo-code for the `heartbeat` node.

```
Input  : message
Output : ⟨ping, ok, error⟩                  // egress ignores '_' messages
1  init
2      config: {
3          ping: message
4          ok: message
5          error: message
6          mode ← m ∈ { passive, active }
7          timeout: ℝ>0
8      }
9      timer ← newTimer(config.timeout)
10 onInput
11     timer.restart()
12     if config.mode = passive then
13         return ⟨_, config.ok, _⟩
14     if config.mode = active then
15         return ⟨config.ping, config.ok, _⟩
16 onTimeout
17     timer.restart()
18     return ⟨_, _, config.error⟩
```

Algorithm C.11: Pseudo-code for the `action-audit` node.

```
Input  : message ∈ { action, acknowledgment }
Output : ⟨ok, timeout, error⟩                          // egress ignores '_' messages
/* Acknowledging an actuator correct operation depends on recieving a confirmation from an
   independent source, such as a sensor. The action is confirmed when an acknowledgment message
   arrives within the defined timeout window. Assumes that acknowledgments occur in the same
   order as actions.                                                                  */
1  init
2      config: {
3          timeout: ℝ_{>0}
4      }
5      actionQueue: queue⟨message, timer⟩
6  onInput
7      if message is action then
8          timer ← newTimer(config.timeout)
9          actionQueue.push({message, timer})
10     if message is acknowledgment then
11         if |actionQueue| = 0 then
12             return ⟨_, _, error⟩
13         else
14             actionQueue.dequeue()
15             return ⟨ok, _, _⟩
16 onTimeout
17     message ← timedOutMessage
18     actionQueue.remove(message)
19     return ⟨_, timeout, _⟩
```

Algorithm C.12: Pseudo-code for the debounce node.

```
Input  : message
Output : ⟨message, delayed⟩
1  init
2      config: {
3          minInterval: ℝ>1,                    // minimal interval between messages (sec.)
4          strategy: s ∈ { discard, first, last, allByOrder },
5      }
6      timer ← newTimer(interval: ℝ>0)
7      lastMsgTimestamp: timestamp
8      pendingMessages: message[ ]
9  onInput
10     currentTimestamp ← time.now()
11     if (currentTimestamp - lastMsgTimestamp ≥ minInterval and |pendingMessages| = 0) or not
         lastMsgTimestamp then
12         lastMsgTimestamp ← currentTimestamp
13         return ⟨message, _⟩
14     if currentTimestamp − lastMsgTimestamp ≤ minInterval then
15         if config.strategy ≠ discard then
16             if not timer then
17                 timer ← newTimer(config.minInterval)
18             pendingMessages ← pendingMessages ++ message
19         return ⟨_, message⟩
20 onTimeout
21     if config.strategy = allByOrder then
22         message ← pendingMessages0
23         delete(pendingMessages0)
24         if |pendingMessages| > 0 then
25             timer ← newTimer(config.minInterval)
26         return ⟨message, _⟩
27     if config.strategy = last then
28         message ← pendingMessages.pop()
29         delete(pendingMessages)
30         return ⟨message, _⟩
31     if config.strategy = first then
32         message ← pendingMessages0
33         delete(pendingMessages)
34         return ⟨message, _⟩
```

Algorithm C.13: Pseudo-code for the `http-aware` node.

```
Input  : _
Output : ⟨ip, port⟩[ ]
1  init
2      config: {
3          ipRange: ipRange,                                  // e.g., 192.168.0.124
4          ports: p[ ] ⊂ {80,443,8080},
5          scanInterval: ℝ>0
6      }
7      timer ← newTimer(config.scanInterval)
8      services: ⟨ip, port⟩[ ]
9  onInput
10     timer.stop()
11     services ← [ ]
12     forall ip in ipRange do
13         forall port in ports do
14             httpObj ← ⟨ip, port⟩
15             httpCode ← httpObj.get("/")
16             if httpCode = "200 OK" then
17                 servicesFound.push(⟨ip, port⟩)
18     timer.restart()
19     return ⟨services⟩
20 onTimeout
21     trigger onInput
```

Algorithm C.14: Pseudo-code for the `resource-monitor` node.

```
Input  : message
Output : ⟨processorAlert, memoryAlert, storageAlert, batteryAlert⟩
1  init
2      config: {
3          maxProcessorUsage: {x : 0 ≤ x ≤ 100},
4          maxMemoryUsage: {x : 0 ≤ x ≤ 100},
5          maxStorageUsage: {x : 0 ≤ x ≤ 100},
6          minBattery: {x : 0 ≤ x ≤ 100},
7      }
8  onInput
9      processorAlert ← message.processor > maxProcessorUsage
10     memoryAlert ← message.memory > maxMemoryUsage
11     storageAlert ← message.storage > maxStorageUsage
12     minBattery ← message.battery < minBattery
13     return ⟨processorAlert, memoryAlert, storageAlert, batteryAlert⟩
```

Algorithm C.15: Pseudo-code for the `balancing` node.

Input : message
Output : $\langle \text{out}_0, \ldots, \text{out}_{\text{config.numberOfOutputs}} \rangle$

```
1  init
2      config:{
3          numberOfOutputs: ℝ>1,
4          strategy: s ∈ { roundRobin, weightedRoundRobin, random},
5          weights: ℝ>1[ ]
6      }
7      lastOutputUsed ← NIL
8      inv |weights| = config.numberOfOutputs
9  onInput
10     if strategy = random then
11         outIndex = random(1,config.numberOfOutputs)
12         return ⟨ …, message_outIndex, …⟩
13     if strategy = roundRobin then
14         outIndex = (lastOutputUsed + 1) mod config.numberOfOutputs
15         lastOutputUsed = outIndex
16         return ⟨ …, message_outIndex, …⟩
17     if strategy = weightedRoundRobin then
18         outIndex =
             sortOutputsByWeights(weights)_((lastOutputUsed+1) mod config.numberOfOutputs
             lastOutputUsed = outIndex
19         return ⟨ …, message_outIndex, …⟩
```

Algorithm C.16: Pseudo-code for the `timing-check` node.

Input : message
Output : ⟨normal,tooSlow,tooFast⟩

```
1  init
2      config: {
3          messageInterval: ℝ>0,
4          margin: ℝ>0,
5          historySize: ℝ>0,
6      }
7      msgHistory: [ ]
8      lastMsgTimestamp: ℝ>0
9      inv |msgHistory| ≤ config.historySize
10 onInput
11     message.timestamp ← time.now()
12     if |msgHistory| ≥ config.historySize then
13         delete(msgHistory_0)
14     msgHistory ← msgHistory ++ message
15     if not lastMsgTimestamp then
16         return ⟨message, _, _⟩
17     else
18         averageInterval ← (Σ_{i=1}^{config.historySize} msgHistory_i.timestamp) / config.historySize
19         if averageInterval > messageInterval + margin then
20             return ⟨_, message, _⟩
21         if averageInterval < messageInterval - margin then
22             return ⟨_, _, message⟩
23         return ⟨message, _, _⟩
```

Algorithm C.17: Pseudo-code for the `network-aware` node.

```
Input  : _
Output : devices
1  init
2      config: {
3          ipRange: ipRange,                                  // e.g., 192.168.0.124
4          scanInterval: ℝ_{>0}
5      }
6      timer ← newTimer(config.scanInterval)
7      devices ← []: {
8          ip: ipAddress,
9          manufacturer: string,
10         timestamp: timestamp
11     }
12 onInput
13     devices ← []
14     timer.stop()
15     arpTable ← OS.getArpTable()
16     forall entry in arpTable do
17         deviceEntry ← {
18             ip: entry.ip,
19             manufacturer: manufacturerFromMAC(entry.mac),
20             timestamp: time.now()
21         }
22         devices ← devices ++ deviceEntry
23     timer.restart()
24     return devices
25 onTimeout
26     trigger onInput
```

Algorithm C.18: Pseudo-code for the `readings-watcher` node.

Input : sensorReading
Output : ⟨okReading,errorReading⟩

```
1  init
2      config: {
3          minChange: {x : x ∈ ℝ0 ∧ x ≥ 0}
4          maxChange: {x : x ∈ ℝ0 ∧ x ≥ 0}
5          stuckCounter: {x : x ∈ ℝ0 ∧ x ≥ 0}
6      }
7      sensorHistory ← [ ]
8      inv |sensorHistory| = stuckCounter
9  onInput
10     if |sensorHistory| = 0 then
11         sensorHistory ← sensorHistory ++ sensorReading
12         return ⟨sensorReading, _⟩
13     change ← |sensorHistory0-sensorReading|
14     if minChange and change ≤ minChange then
15         return ⟨_, sensorReading⟩
16     if maxChange and change ≥ maxChange then
17         return ⟨_, sensorReading⟩
18     if stuckCounter and |sensorHistory| = stuckCounter then
19         if ∨ x, y ∈ sensorHistory | x = y then
20             return ⟨_, sensorReading⟩
21         if |sensorHistory| ≥ config.historySize then
22             delete(sensorHistory0)
23         sensorHistory ← sensorHistory ++ sensorReading
24     return ⟨sensorReading, _⟩
```