FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Exploring Complex Event Management in Smart-Spaces through a Conversation-Based Approach

André Sousa Lago

Master in Informatics and Computing Engineering

Supervisor: Hugo Sereno Ferreira, PhD

Second Supervisor: Claudio Cherubino

June 22, 2018

Exploring Complex Event Management in Smart-Spaces through a Conversation-Based Approach

André Sousa Lago

Master in Informatics and Computing Engineering

June 22, 2018

Abstract

The Internet of Things, or IoT, refers to the networked connection of everyday objects which is often equipped with a collective sense of intelligence. IoT enables the creation of customized intelligent systems that can be installed virtually anywhere and improve the interaction between humans and their surrounding spaces.

One of the most common applications of IoT are smart homes. These are regular homes equipped with sensors and other devices that enable users to create automation mechanisms or interact with their own home remotely. Some examples of smart home applications are remote door locks, security cameras and alarm systems, pressure and motion sensors or even smart switches.

To provide smart homes with even more intelligence, the world's tech giants have developed smart assistants that aim to make the interaction with these IoT systems more seamless. Products like Siri or the Google Assistant enable users to turn on lights or their alarm system with nothing but their voice, which makes smart homes truly smart and even more appealing.

However, these assistants are incomplete as they are incapable of fully managing IoT systems using only voice commands. They are usually limited to simple commands such as "turn on the light" not being able to easily setup complex actions such as systems rules or repeating commands.

As an alternative to these assistants there are visual programming tools (VPPs) like IBM's Node-RED that enable users to fully manage any IoT system creating complex rules like turning on lights when a tweet with #led is received simply by dragging and connecting blocks in a web app. However, these tools are lacking in making the management seamless and effortless as they require users to sit in front of a computer to setup the system and their visual UI can be hard to interpret when the system complexity increases.

The main goal of this project is to develop a bot that is capable of fully managing an IoT system, being able to interact with the system and manage its rules. This means that the supported interactions should range from simple commands like "turn on the toaster" to "when it is raining outside, close the windows". Additionally, users should be able to understand how the system is setup by asking questions like "why did the light turn on?".

This project's success can prove that IoT assistants can evolve into more complex systems, able to provide more complex management features than what is currently possible. Simultaneously, a more complete bot like the one being propose can help more people to start using IoT in their homes, bringing more automation and comfort to their lives.

Resumo

A Internet das Coisas, ou IoT, refere-se à conexão interligada de objetos do dia-a-dia que é frequentemente equipada com um sentido coletivo de inteligência. O IoT permite a criação de espaços inteligentes personalizados que podem ser instalados em praticamente qualquer lugar e melhoram a interação entre os seres humanos e os espaços que os rodeiam.

Uma das aplicações mais comuns do IoT são as casas inteligentes. Estas são casas normais equipadas com sensores e outros dispositivos que permitem aos utilizadores a criação de mecanismos automatizados ou a interação remota com a sua própria casa. Alguns exemplos de aplicações de casas inteligentes são fechaduras de portas, câmaras de segurança ou sistemas de alarme com acesso remoto, sensores de pressão ou movimento ou mesmo interruptores inteligentes.

Para fornecer ainda mais utilidade a estas casas inteligentes, as gigantes empresas tecnológicas mundiais têm desenvolvido assistentes inteligentes cujo objetivo é tornar a interação entre humanos e sistemas de IoT ainda mais fácil. Produtos como a Siri ou o Google Assistant permitem que os seus utilizadores liguem luzes ou sistemas de alarme apenas através da sua voz, tornando as casas verdadeiramente inteligentes e apelativas.

Contudo, estes assistentes são incompletos na medida em que são incapazes de gerir a totalidade de um sistema de IoT apenas com comandos de voz. Por norma, eles estão limitados a comandos simples como "liga a luz" não sendo capazes de gerir ações mais complexas como regras para o sistema ou comandos repetitivos.

Como alternativa a estes assistentes, existem ferramentas de programação visual como o Node-RED da IBM que permitem aos seus utilizadores a completa gestão de um sistema de IoT pela criação de regras complexas como ligar luzes quando receber um tweet com #led simplesmente por arrastar e conectar blocos numa aplicação web. Contudo, estas ferramentas falham em tornar a gestão fácil exigindo que os seus utilizadores se sentem em frente a um computador para criar o sistema e tendo uma interface visual que se pode tornar difícil de interpretar com o aumento da complexidade do sistema associado.

O principal objetivo deste projeto é o de desenvolver um bot capaz de gerir de forma completa um sistema de IoT, sendo capaz de interagir com este e gerir as suas regras. Assim, as interações devem abranger desde frases simples como "liga a torradeira" a "quando estiver a chover, fecha as janelas". Adicionalmente, os utilizadores devem ser capazes de compreender o funcionamento do sistema fazendo perguntas como "porque é que a luz se ligou?".

O sucesso deste projeto pode provar que assistentes de IoT podem evoluir para sistemas mais completos capazes de gerir de forma mais completa os sistemas a que se associam. Para além disso, um bot mais capaz pode fazer com que mais pessoas comecem a utilizar IoT, tornando as suas vidas mais autónomas e confortáveis.

Acknowledgements

First, I thank my supervisor Hugo Sereno Ferreira for guiding me through the process of defining and developing this project. His knowledge and ideas were a huge contribution to the success of the project.

I also thank my second supervisor, Claudio Cherubino, for the availability to provide his opinions and experiences to someone he didn't know while being over 8000km away.

To my family, I give the biggest of gratitudes for their never ending love and support through not only this dissertation but also my entire life. I am who I am because of them, and nothing would be the same without them. A special "thank you" goes to my brother, who has been the closest friend and definitely the most important person in my live.

Last but not least, I thank my friends, "O Grupo das Boleias", for many years of amazing friendship, moments and experiences, and for the many years to come.

André Sousa Lago

"I can accept failure, everyone fails at something.
But I can't accept not trying."

Michael Jordan

Contents

List of Figures

List of Tables

LIST OF TABLES

Abbreviations

API	Application Programming Interface
EPC	Electronic Product Code
GPIO	General Purpose Input/Output
IPA	Intelligent Personal Assistant
IoT	Internet of Things
ISA	International Society of Automation
JSON	JavaScript Object Notation
ML	Machine Learning
NLP	Natural Language Processing
QPM	Queries Per Minute
REST	Representational State Transfer
RFID	Radio Frequency Identification
SR	Speech Recognition
TS	Text Synthesis
TTS	Text-to-Speech
UI	User Interface
URL	Uniform Resource Locator
VPP	Visual Programming Platform
WAR	Web Application Archive
Wi-Fi	Wireless Internet
W3C	World Wide Web Consortium
WWW	*World Wide Web*

Chapter 1

Introduction

This chapter introduces the scope of this project, as well as the problems it aims to solve. Section 1.1 describes the framing of the project in terms of the technology fields it is based on. Section 1.2 presents the main problem tackled by this project, as well as some of its causes. Then, section 1.3 discusses the main forces driving this project as well as what it proposes to deliver, setting the bar for what should be expected from it. Finally, section 1.4 explains how the rest of this document is structured, and what content should be expected in it.

1.1 Context

The Internet of Things, or IoT, is the networked connection of everyday objects, which is often equipped with a collective sense of intelligence [XYWV12]. The integration of such objects creates a huge range of distributed systems that are able to interact with the environment and human beings around them in a lot of different ways.

The flexibility of IoT systems has enabled their use across many different product areas and markets, including smart homes, smart cities, healthcare, transportation, retail, wearables, agriculture and industry [Mou17].

IoT is a booming technological market, and Gartner predicts that 11.2 billion devices will be connected in 2018, a number that is also predicted to almost double over the following 2 years, becoming 20.4 billion devices by 2020 [vdM17]. The Boston Consulting Group also estimates that by 2020 companies will spend 250 billion Euros in IoT on top of what they already spend on other technologies [HRS+17]. This means that not only more people will be using IoT, but also that it will be present in a lot of different environments and situations. This represents a unique opportunity for IoT to evolve as a facilitator on people's lives. After all, having intelligently connected devices around us should help us make our day to day lives easier.

Even though the boom of IoT is quite recent, there are some older fields of software engineering that are also playing a big role in IoT's evolution: Natural Language Processing (NLP), Speech Recognition (SR) and Text-to-Speech (TTS). Combined, these technologies allow for conversational interaction between humans and technology systems, with the aim of making human-computer interaction closer to human-human interaction. Because of that, companies are able to create intelligent personal assistants, or IPAs, that aim to assist people in their daily activities [SRC+16].

Some examples of these assistants are the Google Assistant[1], Amazon Alexa[2] or Apple's Siri[3]. These aim to help users complete simple daily tasks like finding a recipe, calculating the distance to the moon or converting Euros to Dollars without a single touch on a device. Plus, these assistants have the ability to communicate with certain IoT systems, which makes them a great tool for making smart homes.

1.2 Problem Definition

IoT devices tend to be divided into several ecosystems, which are often associated with one of the smart assistants developed by the technology giants (Google Assistant, Amazon Alexa and Apple Siri). On its own, installing and coordinating products from just one of these ecosystems in a closed space is relatively easy. However, by doing so the range of usable devices is limited, and therefore the possibilities of what the system as a whole can do are reduced.

Using platforms like Node-RED[4], it is possible to set up a customized IoT environment, but doing so can demand a lot of work, especially for people that are not familiar with programming or software development. For example, IBM's tutorial[5] for creating a customized smart home requires the users to write code, use wires and lastly sit in front of a computer to create the rules that define the system's behavior.

This means that, for a regular user, the choices are usually restricted to the tech giants' ecosystems, and because of that some users may not be able to fully experience the advantages of IoT simply because its setup and management is too hard.

1.3 Motivation and Goals

The main goal of this project is to make the management of IoT systems easier so that not only current users can increase their comfort with such systems, but also so that it is more seamless for new users to start using smart spaces.

To achieve that, the goal is to develop an intelligent conversational bot that is able to completely manage an IoT system in a seamless way. The purpose of this bot is to enable the complete

[1] https://assistant.google.com/
[2] https://developer.amazon.com/alexa
[3] https://www.apple.com/ios/siri/
[4] https://nodered.org/
[5] https://www.ibm.com/developerworks/library/iot-lp101-get-started-develop-iot-home-automation/index.html

management of a smart space without the need of using a computer to visualize the system's setup or change system rules.

Users should be able to interact with the IoT system in several different ways, with different degrees of complexity:

- Execute a direct action on a device:
  - "Turn on the kitchen light"
  - "What is the living room temperature?"
- Create a delayed task:
  - "In five minutes, turn on the kitchen light"
- Create a repeating task:
  - "Every day at 11pm turn on the alarm"
  - "Close the windows when it's raining"
- Understand how the system works by talking to the bot:
  - "Why did the kitchen light turn on?"
- Edit a task by having a conversation with the bot:
  - User - "When did I tell you to turn on the kitchen light?"
    Bot - "Every day at 5pm"
    User - "Make that 6pm"

From a user's standpoint, the final result of the project should be usable through usual conversational means that already exist, being the text interface of apps such as Slack or Facebook Messenger, or through natural language voice interfaces such as the Google Assistant or Amazon Alexa. This is possible because assistants like the Google Assistant[6] and Amazon Alexa[7], among others, provide APIs for developers to create custom integrations with them. It should also be possible to interact with the bot with text in case of noisy environments.

If this goal is achieved, the result should enable less instructed people to create their own IoT systems without the hassle of doing the setup by hand.

1.4 Report Structure

Chapter 2 describes the state of the art surrounding this project's scope. Throughout it, the history and current status of the technological and economical fields related to this project's objective are presented as a way to introduce the issues this project aims to resolve. Chapter 3 focuses on those issues, describing the proposed solution and its expected impact. Along with that, the techniques used to measure this project's success are also presented. Chapter 4 focuses on the work that has been done, describing how the solution was developed and how several techniques

[6] https://developers.google.com/actions/building-your-apps
[7] https://developer.amazon.com/alexa-skills-kit

were put together in order to create the results that are presented afterwards. Chapter 5 analyzes the evaluation process that was made to validate the results of the project, going through how the proposed features work from a usage standpoint and how they were made easier by this project. Finally, Chapter 6 reflects on the actual success of the project, presenting its main difficulties and future goals.

Chapter 2

State of the Art

This chapter extensively details the context of this project in the form of its state of the art. At first, Section 2.1 reports the definition and history of automation as a field of technology and the economic market. As a consequence of this movement comes the Internet of Things, described in Section 2.2, yet another technological movement with major implications on what this project is about. IoT's history and current applications will be debated as a way to introduce the scope of this project. Being smart spaces one of the most popular IoT applications, the products and platforms used to interact with them are described in Section 2.3. Finally, since chatbots will be shown to be extremely relevant in the field of IoT and this project's scope, some tools used to develop these bots are described in Section 2.4.

2.1 Automation

The International Society of Automation (ISA) defines automation as "the technique of making an apparatus, a process, or a system operate automatically" or, in their words, "the creation and application of technology to monitor and control the production and delivery of products and services" [ISA10].

The very definition of laziness has led humans to automate tasks for thousands of years in an attempt to make their lives easier. Even something as simple as creating a structure that allows cattle to be fed on demand instead of forcing the farmer to deliver food can be seen as a way to automate a process. Guarnieri documents the oldest form of automation as the water clock, dating back to 285-222 BC [Gua10], which uses water movement to keep track of time therefore doing it without any human interaction.

In the recent past, large scale forms of automation were seen the transformation of manual and demanding tasks into automatic processes through the use of developing technologies. A clear example of this was the automation of phone call switching which led to the extinction of telephone call operators starting in 1892 [Jer34]. Even earlier than that, machines started to be used in the industry to replace manual labour that previously heavily depended in human dexterity and skill. As early as in the 1840s, automated machines were used in factories so that unexperienced laborers and boys became more able to operate them [MR69]. As this last example demonstrates, automation can be particularly useful for manufacturing and the industry, which was one of its most popular applications back in the 18th century partly due to the industrial revolution felt at the time.

Despite automation's problems with initial costs or even the ethical concerns such as loss of jobs to machines, it simultaneously presents us with unquestionable advantages that range from increased productivity and consistency to scalability and replication.

The advantages mentioned above led to the transition of automation from the manufacturing industry to many other interesting use cases where it is helpful. Use cases of automation can be seen in waste management [McF17], video surveillance [Dr.], retail [Pet17] and mining [Oli17], among many others.

These examples demonstrate how services or production processes can be improved by replacing older systems or even people by automatic machines that outperform their predecessors in many ways. While these can make human jobs and tasks more effective, they disregard an important place we spend a lot of time on - our homes.

Based on a study from the American Bureau of Labor Statistics, in 2016 american working adults spent at least over 8 hours per day at home (if time spent on household activities and sleep is added up) [bur17]. This means that the average American worker spends over a third of his time in the same physical space, which in turn should make that space a particular subject of increased comfort for its occupants.

This is where a new type of automation comes in - **domotics**. This term refers to the automation of mechanisms or processes in a home, which usually aim to make its inhabitants' lives easier and more comfortable. The first widespread applications of domotics are mostly utensils that nowadays are considered simple and "mandatory" in modern houses: washing machines, refrigerators, air conditioning and many others. All of these are systems that aim to automate certain specific and isolated tasks, which makes them extremely useful in our daily lives. Even though domotics wasn't the first focus of automation, it is still a market that is experiencing a fast growth, documented by an ABI Research study that indicates 1.5 Million home automation systems having been installed in the United States during 2012 [abi12].

Especially in the early days of domotics, a characteristic of most applications was that they were isolated tasks being automated by machines. In other words, these systems were able to automate specific tasks without communicating with each other or doing any sort of functionality that spread beyond what they were programmed to do. All of the examples mentioned above, from air conditioning systems to refrigerators, were very good at one limited set of tasks, but could not cooperate or present any relevant level of intelligence. More recently, with the advent of the Internet of Things, that panorama started to change.

Figure 2.1: RFID label

2.2 Internet of Things

The Internet of Things is a complex concept that has evolved a lot over the years, both in terms of applications but also market size. Throughout this section, we will present a brief history of IoT along with some of its main applications, in order to better understand the framing of this dissertation.

2.2.1 History

Even though IoT is a concept that saw a huge growth in the past few years, it is not a recent concept at all. In fact, it is almost as old as the concept of Internet itself as applications as simple as a toaster that could be turned on over the Internet were made in the year after the global Internet was born, in 1989 [SDPA14]. However, the term "Internet of Things" only became popular after being used as the title for a presentation made by Kevin Ashton, cofounder of the Auto-ID Center at the Massachusetts Institute of Technology in 1999 [Kev09].

Only one year later, LG announced the world's first consumer Internet refrigerator, which was the first widely available consumer product that incorporated the concept of IoT by connecting a household appliance to the Internet [FS16].

In 2003, news organizations like The Guardian started reporting advances in Radio Frequency Identification (RFID) and Electronic Product Code (EPC) that would turn out to be very relevant for the growth of IoT [Sea03]. With these technologies, it was possible to create RFID labels that can be placed in all sorts of objects, from a tin can to a bottle of water, to make them instantly identifiable with an appropriate reader. There are a lot of different RFID labels, and one of them is presented in Figure 2.1.

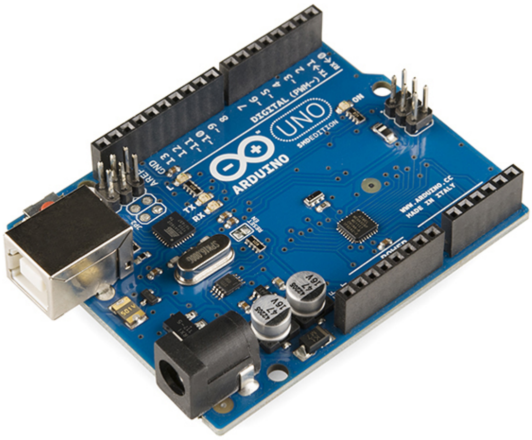

Figure 2.2: Arduino board

Marking the increased recognition of IoT, the first European IoT Conference was held in 2008[1] , which along with the birth of IPv6 [Dee98] and IoT's steady growth allowed for the number of Internet connected devices to increase.

The new millennium also brought devices like Arduino (2005[2]) and the Raspberry Pi (2012 [Den13]) to the general public, which made it cheap to have small, connected computer in all sorts of spaces, from houses to offices.

Arduino[3] is an open-sourced hardware platform that allows users to program electronic boards to read inputs and perform certain outputs. Because of Arduino boards' low cost, they are widely used for homemade IoT systems, as they can be used to turn on lights, publish online activity or move an object. Figure 2.2 displays an Arduino board.

On a similar concept, the Raspberry Pi Foundation[4] aims to provide a low cost fully functional computer that can motivate people to learn more about computers, but also make them more accessible when money is a limitation. Similarly to Arduino boards, the Raspberry Pi computers are very popular for homemade IoT systems for being cheap and very capable. Another advantage of the Raspberry Pi is that it has a GPIO port that allows for customized connections with all sorts of electronic appliances. Figure 2.3 shows what a Raspberry Pi looks like.

The result of combining Internet connectivity, product identification, network identity and cheap electronic hardware was a huge increase in connected devices around the globe, with IEEE owned online publication Electronics360 estimating that 20 Billion devices were connected to the

[1] http://www.the-internet-of-things.org/iot2008/
[2] https://www.arduino.cc/en/Main/AboutUs
[3] https://www.arduino.cc/en/Guide/Introduction
[4] https://www.raspberrypi.org/about/

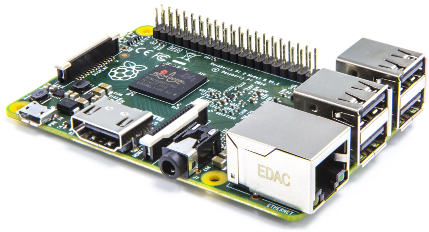

Figure 2.3: Raspberry Pi computer

Internet during 2017 [Bro17]. Similar studies by tech consultant Gartner indicate similar numbers for the overall growth of IoT [vdM17] [RvdM14].

2.2.2 IoT Applications

This boom in worldwide connected devices has led to a lot of different applications of these technologies across countries and product areas. Although being a relatively small sample, the examples below demonstrate different use cases of IoT when combined with multiple technologies and markets [CXL+14, LL15, XHL14].

Smart Homes are the IoT application of domotics. While domotics usually refers to individual systems that perform isolated tasks automatically, smart homes usually refer to a set of connected sensors and electronics that allow for a house to be more autonomous. Some smart houses include appliances such as fridges that remind users when a certain item is about to run out, self-regulating temperature systems or self-locking door and window mechanisms. Perhaps more importantly, many of these devices can be controlled or monitored remotely which provides users with a greater sense of control of their appliances.

One of the most popular companies in this market is Nest[5], an Alphabet company that develops smart thermostats, doorbells, cameras and alarm systems. Nest products can be connected to a network, which allows for their monitoring and control, even when the user is not at home. For example, the Nest Secure[6] system can be controlled via a mobile app, through which users can check the lock status on doors or toggle the alarm system.

Wearables are devices that are worn like clothes, accompanying human beings in their regular activities. Some examples of wearable devices are smartwatches, step counters or smart glasses. With the sizes of processors and electronic boards shrinking, the capabilities of these devices have increased, and such can be seen in the growth of this industry segment which was predicted to surpass 4 billion dollars in 2017 by Forbes [Mar16].

One example of a wearable device is the Apple Watch[7] (displayed in Figure 2.5), a smartwatch that is able to send text messages, play music, count steps and much more.

[5] https://nest.com
[6] https://nest.com/alarm-system/overview/
[7] https://www.apple.com/watch/

Figure 2.4: NEST product family

Smart Cities are a concept similar to smart homes, where the same technology is applied in the context of a public space. These usually aim towards simplifying urban life, or making it more environment-friendly. The most common use cases in this segment are smart parking spaces, smart waste management systems or smart street lighting.

One of the smart-city companies is Telensa[8], that develops both smart street lighting and smart parking. These products provide not only wireless control of street lighting and energy saving techniques, but also parking monitoring systems to reduce congestion.

Retail can also be an interesting use case for IoT as it can benefit both customers and store managers. In these cases, IoT can not only help customers instantly know whether a certain product is in stock or not, but also help the manager determine when to order a certain product based on its current shelf stock.

A good example of Retail IoT is IBM's TRIRIGA[9] facility management system [10], which performs facility planning and energy-saving techniques based on IoT devices and sensors.

Healthcare is yet another field where IoT can be very beneficial, as it can help doctors remotely keep track of a patient's live status, or receive an alert when a problem is detected with a patient. An article by IBM even alerts that even though there are a lot of problems around IoT in healthcare, especially due to data privacy, it can help reduce healthcare costs or improve the outcome of treatments [Pat17].

An example of a real healthcare IoT product is Eversense's portable connected glucose monitoring systems[11].

Finally, **customized smart spaces** are a logical consequence of the growth of IoT and its associated products, as it became possible for almost everyone to create a customized IoT experience based on products and hardware available. In any IoT system, the essential items are the physical devices that are connected by the system and interact with the environment. In this document these are called **leaf devices**.

[8] https://www.telensa.com/
[9] https://www.ibm.com/us-en/marketplace/ibm-tririga
[10] https://www.ibm.com/internet-of-things/iot-solutions/facilities-management/
[11] https://ous.eversensediabetes.com/products/eversense-sensor/

Figure 2.5: Apple Watch

The first step to create a customized smart space is to acquire the leaf devices, depending on the intended use for the system. Some of the most common leaf devices are:

- **Temperature sensors:** used to monitor the temperature of a room or object
- **Pressure sensors:** used to sense pressure in objects like chairs, windows or doors
- **Luminosity sensors:** used to determine the intensity of light in a specific area
- **Motion sensors:** used to detect movement in a space, usually using infrared radiation technology
- **Remote switches:** used to remotely switch lights or electronics on or off
- **Cameras:** used to remotely view a recording of a certain room or area
- **Lock mechanisms:** used to lock or unlock doors, windows or gates
- **Speakers and displays:** used to output audio or visual information via a remote input source (e.g. stream a YouTube video from a phone to the living room TV)
- **Thermostats:** used to remotely control the temperature in a room

Although some of these devices have controllers of their own that can be programmed or controlled in a certain way, it is also possible to acquire **middleware devices** that connect to the leaf devices, and therefore are able to read and modify their current status.

Arduino boards and Raspberry Pi computers, which were mentioned above, are often used as middleware devices due to their setup simplicity and low cost. For example, a Raspberry Pi's GPIO[12] ports can be simultaneously connected to a luminosity sensor and a light switch. That way, not only the luminosity value of the sensor can be sent to a remote server via Wi-Fi, but also the light switch can be turned on when the luminosity drops below a certain level. Arduino boards can also be used to increase the capacity of simpler devices such as sensors or actuators. For example, an Arduino board can be connected to a sensor providing it with a connection to a local network, so that the sensor's status can be accessed remotely.

[12]General Purpose Input/Output, multi-purpose ports that can be programmed for different inputs and outputs

Once the devices are acquired, it is necessary to connect them to a network and manage their behavior. To achieve this, a common technique is to use a **supervisor device**, a middleware device that acts as a supervisor for all the other devices. A computer or Raspberry Pi are common supervisor devices as these usually require a bit more power than alternatives like Arduino boards can offer. Once the supervisor device is ready, platforms like Node-RED[13] or Home Assistant[14] can be installed to facilitate the management of the system as a whole. These **management platforms** are described thoroughly below.

If there is the need to store data from the system, there are plenty **data storage** solutions that can be applied to lightweight systems such as these. MongoDB[15] and InfluxDB[16] are popular choices especially for simple local instances of storage. Another option is Apache Cassandra[17], a tool driven towards scalable distributed databases which can be useful with large amounts of data. Although it might not look like it, IoT systems can generate a lot of data depending on the granularity desired for devices like sensors. If, for instance, the value of a sensor is read and stored 10 times per second and there are 1000 sensors in the system, it is clear how fast the need for storage can increase.

The final piece in making a complete smart space is a **monitoring tool**, such as Kibana[18] or Grafana[19]. These allow for not only the monitoring of the system device's values (values of sensors, position of actuators, . . .), but also its usage (middleware device processor usage, battery levels, . . .). With such tools, all this monitoring can be done remotely with a friendly web interface that displays all that information in a friendly way.

As a practical example, let's picture a user that wants to have a luminosity sensor and a temperature sensor in his room, and an actuator that can open and close the window. With these, the user wants to have a dashboard where he can consult the history of the room's luminosity as well as the status (open/closed) of the window. Finally, the user wants the window to be shut if the temperature drops below a certain level. To achieve this functionality, all the user needs is to buy the actual sensors, the actuator and a Raspberry Pi. Then, the sensors and the actuator are connected to the Pi, which is then given an installation of Node-RED, MongoDB and Grafana. With Node-RED, the user can make the window close when the temperature drops, and also direct the values of the luminosity sensor to the MongoDB storage system. Finally, the user configures the Grafana dashboard and makes it accessible from outside of his local network so that he can see what's going on when he is not at home.

With all the components mentioned in this section, it is possible for users to create a fully customized smart space with personalized sensors, actuators, management platforms and even data storage mechanisms. This is a great advantage of IoT, especially for tech-savvy users, as it allows for virtually anything to be automated or remotely controlled.

---

[13] https://nodered.org/
[14] https://home-assistant.io/
[15] https://www.mongodb.com/
[16] https://www.influxdata.com/
[17] http://cassandra.apache.org/
[18] https://www.elastic.co/products/kibana
[19] https://grafana.com/

2.3 Interaction with Smart Spaces

As mentioned in the previous section, the growth and evolution of IoT has led to the appearance of many forms of smart spaces. The wide variety of products and platforms developed for such applications has also led to the use of many ways to interact with these systems. The goal of this section is to describe the many ways and tools used to interact with smart spaces, as well as the technologies they require.

2.3.1 Visual Programming Platforms

Visual programing platforms, or VPPs, are tools that are usually installed in supervisor devices in IoT systems so that they can access all the devices and components in such systems. Because of that, these platforms can offer users with extensive and friendly UIs through which the user can visualize the state of the system, customize its behavior or even configure the system's devices themselves. Some VPPs even offer integrations with third parties such as Twitter or Instagram, so that their APIs can be used as part of the system's behavioral rules.

Let's take a look at some of the most popular VPPs: Node-RED and Home Assistant.

Node-RED[20] is a tool developed by IBM's Emerging Technology Services team in 2013, and is now a part of the JS Foundation. Node-RED follows a flow-based programming approach to the management of an IoT system, providing its users with a Node.js-based web application through which they can create, edit and delete system rules and connections in an interface that displays rules and connections as a flow of information, events or actions.

In Node-RED, system devices are represented as colourful nodes that possess specific properties and actions, which can be interconnected with other nodes. Similarly, conditions, actions and events are also represented as nodes, which makes it easy to understand how elements can be connected in the platform.

Being based in Node.js, all that is required to use Node-RED is to have it running in a supervisor device in the system, such as a Raspberry Pi, and then the platform can be accessed through any regular web browser.

As an example, Figure 2.6 will be analysed as it represents a simple application of Node-RED to manage an IoT system.

In the example of Figure 2.6, the user has defined a flow where built-in Node-RED nodes are connected in a flow of information.

In this case, the flow starts with a Twitter node that triggers the following node in the flow whenever a tweet with the hashtag "#led" is detected in the public feed or the user's private tweet feed. When such a tweet is detected, it is sent to the next node in the flow.

The following element, which is the trigger node, is configured so that, when triggered, it relays a signal with the value "1" to the next node in the flow for 1 second, and then relays the signal "0" once that time has passed.

[20] https://nodered.org/

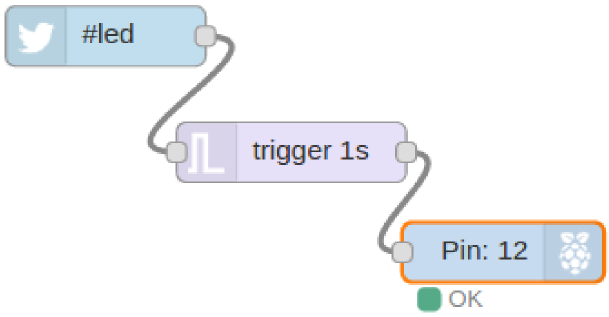

Figure 2.6: Simple setup of a Node-RED managed system

The last element is a Raspberry Pi GPIO node, which controls the input and output of GPIO ports of a Pi. In this case, when it receives the signal “1” it turns on the GPIO port 12 on the Pi, turning it back off when it receives the signal “0”.

If, for example, an LED is connected to the GPIO port 12 of the Pi, then it will be lit for 1 second every time a tweet with the hashtag “#led” is detected.

The same way it is possible to connect Node-RED to Twitter, it is also possible to integrate with a lot of other components ranging from Fitbit wristbands to weather APIs or traffic monitoring systems. Connecting all of these with IoT devices can help make a smart space extremely efficient and intelligent given how many complex integrations can be made. With Node-RED, it is possible to turn on the coffee-machine whenever the user wakes up, turn on the heating system when the user is returning home after work, turn on the A/C if the temperature rises above a certain level, and a lot of other examples. There is virtually an infinite amount of combinations that can be made, and Node-RED’s growing community[21] contributes greatly to those integrations.

Another very useful application of Node-RED is to connect data sources to database systems. With integrations like MongoDB or InfluxDB, it is possible to direct virtually any data source to a data storage system, which is important for keeping track of records, events and other data. This can be used to store anything from tweets to sensor values, and it can be done with either a local, remote or even distributed database.

Node-RED can also integrate with messaging systems such as MQTT[22] or RabbitMQ[23]. These systems are used to convey messages between different software applications, which can even be running in separate machines or networks. As an example, MQTT is quite popular for IoT applications due to its lightweight nature and ease of use. With it, it is possible for applications subscribe to message topics which are identified by paths (e.g. “home/room/light”), which other applications can publish messages to. MQTT then handles the distribution of such messages according to user-defined policies. When integrated with Node-RED, it becomes possible to notify

[21] https://nodered.org/about/community/
[22] http://mqtt.org/
[23] https://www.rabbitmq.com/

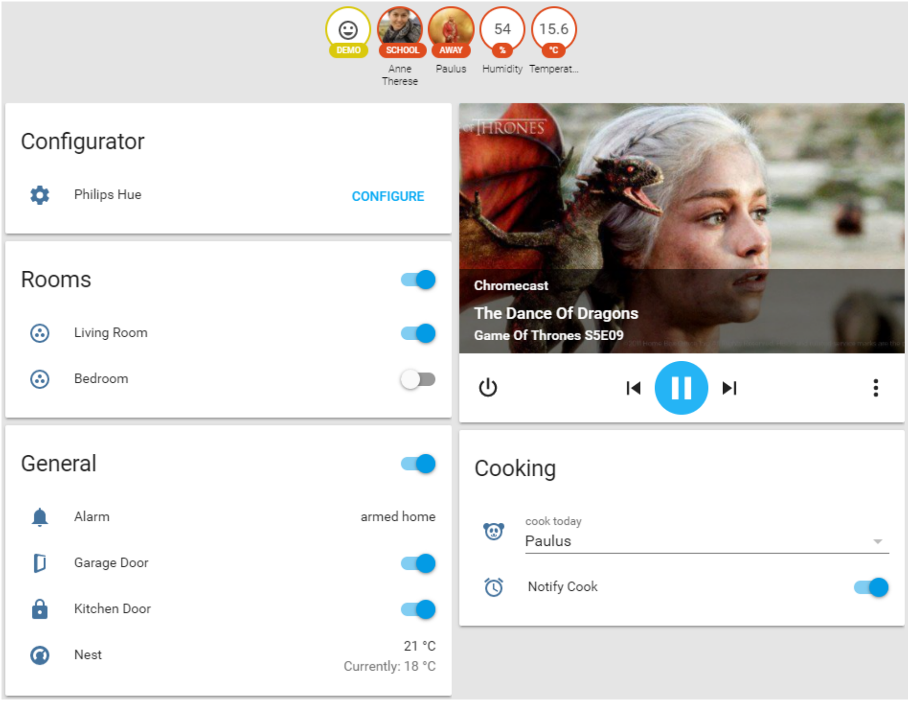

Figure 2.7: Home Assistant's live demo UI

a certain topic in MQTT that the value of a certain sensor has changed, so that other systems and applications can act accordingly.

Another useful integration of Node-RED are dashboard frameworks such as Grafana[24], Kibana[25] or Freeboard[26]. With these integrations, it is possible to convey messages and system data to dashboard applications that may even be hosted remotely, so that the user is given a friendly interface through which he can visualize the status of the system live. With these dashboards it is possible to visualize the current status or value of sensors, the history of a room's temperature, and much more. Again, these are especially useful for customized smart spaces because they allow for remote monitoring of the whole system.

Home Assistant[27] follows a slightly different approach to the design of a VPP, but it offers a powerful and simple way to interact with an established IoT system.

With a very simple and intuitive UI, which is accessible via a web or mobile app, Home Assistant is able to connect to a lot of different types of components of typical IoT systems (NEST, MQTT, IFTTT, Arduino, ...) so that the user can not only visualize the state of his system's devices, but also change their behavior and even define operational rules. Let's take a look at an example from Home Assistant's live demo[28], displayed by Figure 2.7:

[24] https://grafana.com/
[25] https://www.elastic.co/products/kibana
[26] https://freeboard.io/
[27] https://home-assistant.io/
[28] https://home-assistant.io/demo/

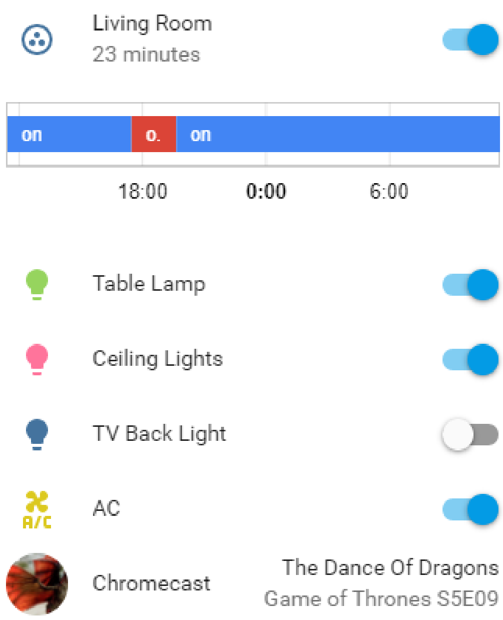

Figure 2.8: Home Assistant's UI for a room's controls (living room)

In the example of Figure 2.7, it is possible to view toggles that represent actuators of the systems (switches, sensors, ...). When a division like "Living Room" is clicked, we can see the panel below which displays all the actuators in that division, as well as their state and a toggle to change their behavior. This panel is displayed by Figure 2.8.

The example also displays the integration with the NEST thermostat, so that the user can visualize not only its current temperature but also its target temperature. Clicking on that item of the UI brings up a chart that shows the evolution of the temperature detected by the thermostat over time.

Automation is also visible in Home Assistant's example, where it is indicated that today's cook ("Paulus") will be notified that it is his time to cook. Other examples of automation through this app are also describe in Home Assistant's documentation[29], which explains how the app can be used to turn on and off lights when a rain sensor detects the presence of rain.

Although its UI may be simpler or slightly better than Node-RED's UI, Home Automation falls a bit short when it comes to the complexity of its rules. Node-RED's more complex UI ends up giving the user more freedom to make custom system rules, and the rules that this platform allows are far more complex, in big part due to its flow programming design. Home Automation might be considered prettier or simpler, but that also takes it some of the power that its alternative can offer. Interestingly enough, it is possible to use both platforms simultaneously, which can make it easier to access Node-RED's information through Home Assistant's pretty UI.

[29] https://home-assistant.io/cookbook/automation_for_rainy_days/

As the examples above demonstrate, VPPs can be extremely useful for users of custom smart spaces due to the flexibility and power they provide. However, they possess several disadvantages that make them somewhat hard to use, especially for users that are not entirely comfortable with understanding how certain technologies work.

Picture a Node-RED system, embedded in a user's home, with multiple devices. Even if there are only a couple of dozens of rules defined, it can be extremely difficult to understand why a certain event took place due to the overwhelming flow that results from them. A single dozen of rules can already result in a Node-RED page that you need to scroll to completely visualize, let alone a couple dozen! The more rules you add to the system, the harder it becomes to look at Node-RED's interface and understand what the system is able to do, in part because it is impossible to make any other form of "query" to the platform besides looking at it.

Another major disadvantage of this sort of platforms is that they require the user to sit in front of a computer to set up the system, even if it is for simple tasks. For example, if a user is sitting in his couch, far away from his computer, and thinks that it would be great to have his light turn on as soon as it gets dark outside, he would need to actually get up and go to the computer when there can possibly be a lot of IoT devices around him that he could interact with. Again, this can make these platforms hard or boring to use as it may require a lot of time to perform simple tasks such as the one described.

To solve some of these issues, there are plenty of chatbots and assistants that can help IoT users. These, are described in Section 2.3.3.

2.3.2 Technologies for Natural Language Interaction

There are three main technologies required to have a conversational experience with a virtual system: speech recognition, natural language processing and text synthesis.

Speech Recognition (SR) is the technology that converts the audio input of a person's voice into a text string that represents exactly what the person said. SR is a relatively old field of study, even going back to 1930 [JR04], but it still improves thanks to pushes from a lot of the world's largest companies. According to TechRadar, Nuance Communications' Dragon[30], Google's Cloud API[31] and Microsoft's Windows[32] provide some of the best user SR products, and some of them are even available for developers to use in their own applications [All17].

Natural Language Processing (NLP) is the extraction of meaning from a text that represents something a person said or wrote, allowing computers to understand human language [Cho03]. NLP is a very complex field of technology, especially because human communication itself can be very complex. Depending on location, language, background and context, humans have a lot of different ways for saying something, and this makes developing NLP algorithms a lot harder. However, there is a lot of work done on NLP that has enabled the development of a wide range of tools, that perform different types of NLP, and with different results. Currently, API's like Google

[30] https://www.nuance.com/dragon/dragon-accessories.html
[31] https://cloud.google.com/speech/
[32] https://www.microsoft.com/en-us/windows/cortana

Cloud[33] or Dialogflow[34] (previously API.AI) give developers the tools they need to understand user input, extracting information like the user's intent or previous conversation context only from a user's text input.

Text Synthesis (TS) or Text-to-Speech (TTS) technologies do the opposite of what SR and NLP do, converting an expression a computer wants to express to a user into an artificial voice, enabling computers to speak like humans do. TS can be seen in a lot of different products and operating systems. One of the first operating systems to have TS out of the box was Apple's Macintosh, already being able to "speak" in 1984 [Her84]. Since then, TS has been featured in multiple other systems like Microsoft's Windows, Google's Android, as well as other minor appliances in vehicle audio systems and other products.

Together, SR, NLP and TS allow for realistic conversational experiences with computers, as they allow for bi-directional comprehension of information and intent.

2.3.3 Chatbots and Intelligent Assistants

With a different approach from VPPs, chatbots and intelligent assistants aim to enable a conversational interaction between humans and IoT systems. What this means is that the ultimate goal of these applications is to have humans talking to machines as if they were human too, as that way the communication is more natural.

When it comes to products and interaction with smart spaces, the technologies used for Speech Recognition, Natural Language Processing and Text Synthesis are merged together into the smart assistants developed by some of the world's largest companies: Google's Assistant[35], Apple's Siri[36], Amazon's Alexa[37], Microsoft's Cortana[38], Samsung's Bixby[39], among others. These are powerful, often cloud-based, software solutions that allow users to either talk or write to algorithms that understands the intent in user phrases.

Chatbots have existed for a while now, and their use has ranged from online companion bots like CleverBot[40] to Pizza Hut customer assistance bots[41] that talk to users while their pizza is being made and delivered. However, these applications were limited not only in the dialog complexity but also its actionable capabilities.

Most of the modern intelligent assistants allow for not only text interactions, but also voice-exclusive interactions, which means the user doesn't need to touch any physical device to interact with it. All that's required is an active microphone, so that the user can talk to it and expect a certain response. This has been one of the main focus features of these assistants, due to the flexibility and ease of use it provides for users.

[33] https://cloud.google.com/natural-language/
[34] https://dialogflow.com/
[35] https://assistant.google.com/
[36] https://www.apple.com/ios/siri/
[37] https://developer.amazon.com/alexa
[38] https://www.microsoft.com/en-us/windows/cortana
[39] http://www.samsung.com/pt/apps/bixby/
[40] http://www.cleverbot.com/
[41] https://botlist.co/bots/pizza-hut

A lot of the world's largest tech companies have focused and invested in developing their own smart assistant, each with his own gimmicks and tricks, but in the end they are all fairly similar: they answer questions from information they find in the web, execute simple tasks on demand and interact with third-party services. Some of the tasks these bots can do based on user queries are getting answers from the internet (with questions like "How tall is the Portuguese President?"), scheduling events (with queries like "Remind me to turn off the oven in 5 minutes") or creating grocery lists (with queries like "Add bacon to my grocery list").

One important and complex feature of most of these bots is conversational awareness - the ability for the bots to connect multiple separate phrases to extract a common line of thought or information. Let's take a look at the following example:

User: "What is the capital city of Portugal?"

Bot: "The capital of Portugal is Lisbon."

User: "What is its population?"

Bot: "The population of Lisbon is 506 892 inhabitants."

In the example, the first question is a relatively easy question for most of the aforementioned bots, as all it requires is to lookup the "capital city" attribute of an entity named "Portugal". Based on the knowledge graphs available online, along with information sources like Wikipedia, this information is quite easy to obtain. However, the second question has a complicated detail that makes it harder to understand, as the user wants to know the "population attribute" of an entity named "its". Obviously, "its" is not a valid entity, so the question, on its own, makes no sense. However, most of the bots mentioned are now capable of interconnecting user queries, and in examples like the one above they are able to understand that "its" relates to an entity from a previous query, and therefore the bot can understand that it refers to "Portugal" and answers the question correctly.

Conversational awareness is an important progress in conversational technologies, as it is a step towards making them more humanlike. Human conversation is embedded with contextual and conversational details that are natural for us, but extremely hard for algorithms and computers that are not yet capable of representing and analysing the world the same way that we do.

However limited they may be, the most recent conversational technologies available can already provide incredible experiences for their users, and they surely lead towards a more automated and intelligent world where humans and machines interact more closely than ever before.

There are a lot of very good assistants currently available to the public, but some of the most popular examples of are these:

- **Google Assistant:** the Assistant is embedded in a lot of Google related products, from Android smartphones, to Pixel laptops and even its own Google Home hardware. It aims to provide information to the user even before he asks for it, giving the user personalized recommendations based on the traffic to work, calendar events, and other information. It also integrates with external services such as mobile networks to provide hands-free cell phone calls to its users.

- **Apple's Siri:** Siri also comes integrated with all of Apple's mobile device range (from iPhones to iWatches), providing users with a lot of hands free features they wouldn't normally be able to perform. A useful integration of Siri is with Apple's own smart home products, Homekit, enabling users to interact with their home in a hands-free way.
- **Amazon's Alexa:** Alexa can be reached through mobile apps or through Amazon's own hardware, the Amazon Echo. With these interfaces, users can talk to Alexa to perform tasks such as searching the web or Alexa's connected systems. Being integrated with the Amazon marketplace, Alexa is even able to perform hands-free purchases with queries like "Order [item] from Prime Now"[42].

Currently, tasks that involve searching the internet for information and facts are considered trivial for assistants as advanced as these are, and pretty much all of them can offer the same completion rate for queries like "what is the capital city of Portugal?" or "what does the weather for tomorrow look like?".

Most of these assistants also offer the conversational awareness mentioned above. This means that not only they know which real entity "Portugal" refers to, but also they understand when a user chains queries that relate to each other in a certain way. The improvements in the conversational and contextual awareness of these assistants make them progressively sound more human and natural, which is definitely a step in the right direction.

Let's take a deeper look into the Google Assistant, as it represents most of what the other assistants can do and has features that are interesting in the context of this document[43].

The Google Assistant is a mostly cloud-based smart assistant. In Google's own words, "Ask it questions. Tell it to do things. It's your own personal Google, always ready to help".

There are plenty of ways users can interact with the Google Assistant, either through the standalone Google apps[44], built-in integrations with Android (6.0+) and Chrome OS (also known as Chromium OS[45]) or with standalone hardware, such as the Google Home and Google Home Mini[46]. Through these interfaces, it is possible to ask the Assistant questions, ask it to do things and interact with other associated products.

With this, comes one of the interesting use cases of the Assistant - interacting with smart spaces. With the Assistant it is possible to talk to smart space devices such as NEST thermostats[47] or alarms[48], as well as other third-party services light Philips Hue[49] bulbs or IFTTT[50] queue systems, among others[51].

[42] https://www.amazon.com/gp/help/customer/display.html?nodeId=201807210

[43] All of the Google Assistant features mentioned in this section were tested using the Google app (version 8.5.15.21.arm64) on a Google Pixel phone with the Android P Beta (May 5, 2018 security patch)

[44] https://play.google.com/store/apps/details?id=com.google.android.googlequicksearchbox&hl=pt_PT

[45] https://www.chromium.org/chromium-os

[46] home.google.com

[47] https://nest.com/thermostats/nest-learning-thermostat/overview/

[48] https://nest.com/alarm-system/overview/

[49] https://www2.meethue.com/en-us

[50] https://ifttt.com/

[51] https://support.google.com/googlehome/table/7401014

The problem with the Assistant's approach is that these interactions are quite simple, and are mostly directly associated with direct commands or queries to the smart devices. All the Assistant can do with these devices is perform queries like "is the baby monitor on?" or "what is the temperature in the living room?", or execute direct actions such as "turn on the coffee machine"[52].

What this means is that although intelligent assistants like the Google Assistant, Siri and other can make it much more comfortable to interact with smart spaces because they remove the need of touching a physical device, they do not allow you to define rules for how the spaces operate. Saying things like "everyday at 3pm close the windows" or "when it is raining turn on the porch light" won't work on these assistants unless you manually define every single one of them.

Overall, it is easy to understand that although current smart assistants can be very helpful and comfortable to use, they don't yet have the complexity and completeness that other IoT management systems like Node-RED possess. Also, some of them are limited to a specific range of vendor devices, so there is always a limitation to the customization they can offer.

2.3.4 IoT Communication Protocols

When it comes to IoT, there are many protocol definitions that are essential for systems to work, and these can fit into the many layers they are made of. These protocols can be dedicated for IoT, but they can also be general application protocols that happen to be usable in these systems.For example, IoT systems may use the IPv6 protocol for the infrastructure layer, Bluetooth in the communication layer or MQTT for data protocols.

As mentioned in Section 2.3.1, physical IoT devices can communicate with each other using message queue systems like RabbitMQ or MQTT. These use path formats to describe devices, topics or other endpoints of the system, therefore creating an organized structure for the communication system. Another way of describing this path structure is to compare it to a REST API as the paths used for these are very similar.

However, without a common structure or protocol defined for these messaging structures, each individual IoT system would be using an ad-hoc solution that could fit its purpose but lead to a reduced flexibility of the system. While it is not essential, having the same organizational structure among IoT systems makes the development of new applications simpler and more structured, removing the need of re-implementing the communication structure wheel.

In an attempt to solve this issue, Mozilla is currently developing a Web Thing API[53] which is expected to be submitted for approval to the W3C[54], a community that brings together companies and experts to develop Web standards. The goal of Mozilla's Web Thing API is to describe a data model and API that could be used in the context of IoT describe physical devices in a JSON format. This API could then be used together with the communication platforms like MQTT

[52] https://store.google.com/us/product/google_home_learn?hl=en-US
[53] https://iot.mozilla.org/wot/
[54] https://www.w3.org/

and RabbitMQ to create a more standardized way of establishing communication structures and protocols for IoT systems.

While the document is still being prepared for submission, it's principles for communication structures and device identification are already interesting and useful for developing projects. The description of some concepts described by Mozilla's API can be found below.

The code snippet in Listing 2.1 describes a sample "web thing", a device that interacts with the environment by reading and/or changing its status.

The JSON code in Listing 2.1 can fully describe the status and abilities of a "web thing", which can be useful for example for knowing what sort of actions a certain device is capable of or to read its current status.

In the example, "properties" can be used to define attributes of a device, which should represent its status. On the other hand, "actions" define functions the device is able to perform, which may be activated remotely. The events the device is capable of emitting are stated under "events", which can be used for monitoring certain changes in the environment. Finally, "links" describes the URLs for accessing the previously mentioned properties of the device, which can be used through communication tools like RabbitMQ.

Besides the description of a device, Mozilla's API also describes the REST APIs that can be used to access such properties of the devices, as well as what device types may be used in this definition (e.g. "onOffSwitch" or "binarySensor").

2.4 Chatbot Development Platforms

Chatbots, like virtually any software application, can be developed from scratch by anyone with the required software development skills. This development can even be made easier with the inclusion of open-source of free to use APIs and libraries that tackle implicit problems of chatbots like speech recognition or natural language processing.

However, software reuse has been proven to be an effective way of improving the whole process of software development [Gri97], and chatbots are not an exception. In fact, there are multiple platforms for chatbot development like Botsify[55] or Chatfuel[56]. These platforms are aimed towards providing bots with text interfaces so that they can be used for example for customer assistance bots in Facebook Messenger. The main downside of these platforms is that the bots are too scripted, lacking the abilities to perform more complex dialogs or understanding the existence of entities in phrases. Also, these platforms lack the voice interaction that makes most chatbots more usable and user friendly.

An alternative to these solutions is Dialogflow[57] (previously known as API.AI). DialogFLow is a product and company owned by Google that provide a platform for developers to create conversational interfaces that use AI to interact with their users. Dialogflow's goal is to use machine

[55] https://botsify.com/
[56] https://chatfuel.com/
[57] https://dialogflow.com/

```
{
  "name":"WoT Pi",
  "type": "thing",
  "description": "A WoT-connected Raspberry Pi",
  "properties": {
    "temperature": {
      "type": "number",
      "unit": "celsius",
      "description": "An ambient temperature sensor",
      "href": "/things/pi/properties/temperature"
    },
    "humidity": {
      "type": "number",
      "unit": "percent",
      "href": "/things/pi/properties/humidity"
    },
    "led": {
      "type": "boolean",
      "description": "A red LED",
      "href": "/things/pi/properties/led"
    }
  },
  "actions": {
    "reboot": {
      "description": "Reboot the device"
    }
  },
  "events": {
    "reboot": {
      "description": "Going down for reboot"
    }
  },
  "links": {
    "properties": "/thing/pi/properties",
    "actions": "/things/pi/actions",
    "events": "/things/pi/events",
    "websocket": "wss://mywebthingserver.com/things/pi"
  }
}
```


Listing 2.1: Mozilla Web Thing API's sample of a "web thing" which can be an IoT device learning to understand the intent, entities and meaning of a user's phrase based on examples given by the developer, changing the bot's responses based on the input.

One big advantage of Dialogflow is that provides a simple way to integrate the bots with many popular products like Facebook Messenger, Slack or the Google Assistant, enabling both text and voice interactions with the created bots. What this means is that the developers can focus only on the conversation aspect of the bot, disregarding the interface between it and the users.

To create a chatbot in Dialogflow, the developer must specify intents which represent a user query. For example, to define the intent of a user saying "hello", all that's needed is to create an intent that accepts "hello" as a user input. Then, the developer defines output responses for the intent such as "good day". Once this is done, Dialogflow uses machine learning to generate phrases that may be similar to "hello" such as "greeting" or "hi" so that the bot becomes able to reply to a more diverse range of queries.

Another useful feature of Dialogflow is the ability to define entities. These are structures that define variable parts of user phrases which may have different values with different meanings. Also, the entities' values may be useful if the bot's response varies according to those. For example, in the intent "pass me a pear", "pear" may be replaced by an entity so that other fruits can be used in its place. In that case, the developer could create the entity "@fruit" and define it as having the values "pear", "banana" and "strawberry". That way, the bot would accept more user inputs and be able to deal with those fruits differently.

In Dialogflow's web UI it is possible to define these intents and entities and all of the processing occurs in Dialogflow's backends. However, it is also possible to make queries to the bots be sent to external APIs so that they may be processed differently. This can be extremely useful to create interactions between a Dialogflow bot and other software systems like smart spaces.

Let's take a look at a possible integration between Dialogflow and a smart space. To enable the Dialogflow bot to respond to a query such as "what is the temperature in the living room", it is possible to use the webhooks provided by Dialogflow to send that exact request to the smart space's API so that the temperature sensor's value is appended to the bot's response. When this is done, the bot sends a JSON object to the smart space's API that indicates not only the user's input (including the entities used in it, the bot's confidence in it;s analysis and other relevant data) but also it's suggested outputs. Then, the API may create a customized response that includes the actual sensor data so that the bot provides a real up-to-date response.

All of the characteristics mentioned above make Dialogflow a great platform for chatbot development as it is able to support complex speech features but also many integrations that can be important to spread the use of a certain bot.

2.5 Summary

Over the years, IoT has become a very relevant concept in today's technology fields. However, the capabilities of the management tools for IoT systems still lack in certain types of features that could help more people to use IoT in their homes.

On one hand, the visual programming platforms like Node-RED and Home Assistant offer a good deal of complexity for the management of a system, including the possibility of defining complex rules and interactions between systems. However, they lack the simplicity and comfort of use that could make them more appealing, especially for users that are not very comfortable with technology.

On the other hand, there are the smart assistants that not only try to maximize the user's comfort by providing voice interactions, but also are already capable of performing multiple integrations and different commands. These assistants are also great for users not so comfortable with technology because they require no technical knowledge. However, as advanced as they may be, they still lack the understanding and complexity required for performing a more complete management of smart spaces like defining operational rules, conditional triggers or interoperability of devices.

Adding to that, recent studies [Ras14] show that users are dissatisfied with the inflexibility and complicated interfaces of IoT systems, claiming that there should be easier and more comfortable ways to interact with them. A study participant's complaint stated that "I don't want to work through a menu just to turn off the lights". [Ran03]

All these problems boil down to the lack of a simple interface for users to manage IoT systems. More people could be creating their own smart spaces if interacting with them was made easier. With the current state of conversational technologies it seems that we don't yet take full advantage of what these can do when it comes to interacting with smart spaces and making them even more useful.

Chapter 3

Problem Statement

The goal of this section is to describe exactly what the problems with the current solutions for IoT management are and what the proposed solution for them is. Then, the relevant features of the solution are presented along with the means for evaluating this project's success and impact.

3.1 Current Issues

As presented in Chapter 2, the current solutions available in the market offer great alternatives for the management of smart spaces, but none of them seems complete as a whole. This is because none of the presented tools simultaneously has these features:

- **Complex Management:** the ability to perform a wide range of tasks, including direct actions, delayed actions, conditional actions or device interoperability.
- **Comfort and ease of use:** the possibility to manage the IoT system with the minnimum possible effort. The maximum comfort would be for the user not to have to move or touch a device in order to get his tasks done, as can happen with voice assistants.
- **Understanding system's functioning:** the ability to understand how the configure system works or why something was done. For example, with Node-RED, this is only possible by looking at all the configured rules to figure out which one could have caused somethign to happen. Ideally, all that should be needed is to ask the system why something happen and it should do that search for the user.

3.2 Proposal

The goal of this project is to develop a conversational bot dedicated to the management of smart spaces that is capable of defining and managing complex system rules, called **Jarvis**.

Jarvis's abilities reach across different levels of operational complexity, ranging from direct one-time actions (e.g. *"turn on the light*) to repeating conditional actions (e.g. *"when it is raining,* *close the windows"*). Besides that, Jarvis should also let the user easily change or understand the status of the system, through queries like *"why did the toaster turn on?"*. In that latter case, Jarvis should also possess conversational awareness that allows for chained commands. This particular feature is demonstrated by the following dialogue:

User: “Why did the toaster turn on?”

Jarvis: "You told me to turn it on at 8 AM.”

User: “Okay, change it to 7:50 AM.”

Jarvis: “Sure, toaster timer was changed.”

In the example above, the second query of the user wouldn’t make sense on its own, however it does make sense as a follow-up to the previous interactions. This can not only be extremely user but also facilitate the user’s experience since it avoids repeating information that was already mentioned in the conversation.

To make the bot easy to integrate with current systems, its interface will be made through existing platforms like the Google Assistant, Amazon Alexa, Facebook Messenger, Slack, among others. This range of integrations give the bot the ability to interact with users via voice or text queries.

3.3 Assumptions

Due to the scope of this dissertation as well as its specific focus of creating a powerful IoT management tool through a conversational bot, there are many aspects of a real chatbot that will not be stressed as they are not essencial to the project’s scope.

Smart spaces are often characterized by a great heterogeneity of devices, communication protocols, feature sets, among other characteristics. These can make the construction of the system, as well as its operation, much more complex and difficult. These characteristics will not be considered for this project. Instead, a well-defined set of device types, communication protocols and features will be defined at the start and used throughout the project since that heterogeneity is a high priority.

When it comes to exposing smart spaces to the Internet for remote access, there are a lot of security and privacy issues that on their own present huge problems in the domain of IoT and smart spaces. In fact, there are multiple cited articles that study and address this concern with IoT’s progression and growth [DKJG17]. These are serious problems that, if not properly addressed, can lead to the spread of private personal information on the web or even illegal access to devices and controllers in a private space. However, because remote access is important for this project and security and privacy are enormous problems on their own, these aspects will not be taken into consideration for the development of this project. Instead, the assumption will be made that the developed system is perfectly secure and private.

Finally, this project assumes that the third-party technologies used for the conversational analysis and device interaction will remain open for use, as they are essential for the project to work

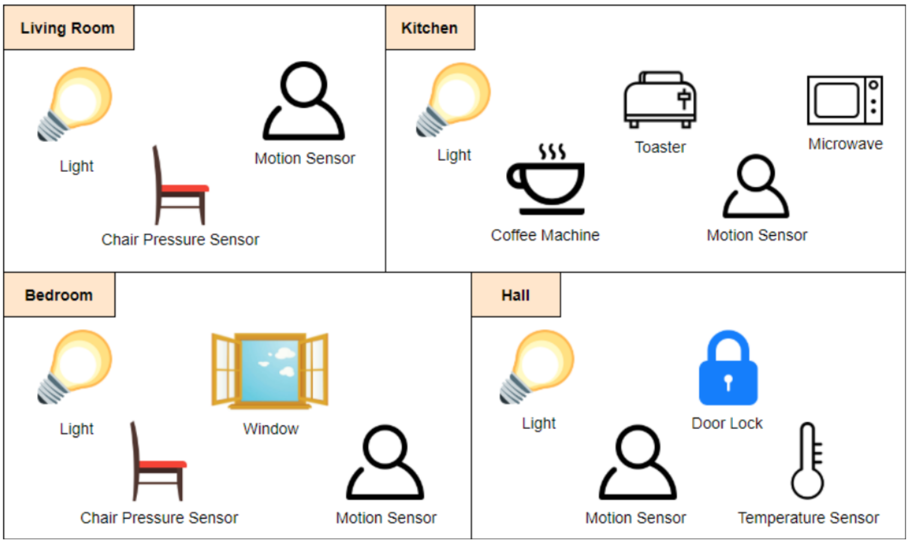

Figure 3.1: Example of a smart home with a wide range of IoT devices

and also represent big problems on their own. This is relevant for the purpose of replicating this project since its functioning depends on the external technologies used.

3.4 Desiderata

This section describes the features that we aim to tackle this project. Some of these features are new, but others already exist in current products. However, their development along this project is essential in proving that complex management of IoT spaces is possible and better with a smart voice assistant.

To visualize how the solution would be applied, we illustrate through Figure 3.1 a sample smart home that includes different sorts of devices, ranging from on-off switches to multilevel sensors (temperature sensors). The goal is that this sample house includes a wide range of the sort of IoT devices found in the majority of smart homes, so that it is possible to understand the expected behavior and advantages of this project.

To better discuss the expected behavior, each of the following subsections explains one of the expected use cases of the solution. All of the examples are based on the house model of Figure 3.1.

3.4.1 Use Case 1: one-time action query

One-time action on a device that happens instantly and does not repeat unless the user specifically commands it. This is the simplest type of command supported by the system and the basic capability of any IoT interaction tool.

Examples:

- *"Turn on the kitchen light."*
- *"Open the bedroom window."*

- *"Set the hall temperature to 20 degrees."*

3.4.2 Use Case 2: one-time action query with uncertainty of device

Similar to Use Case 1, but in this case the device specification is not clear to the system. In these cases, the system should query the user to clarify which device he intends to choose, if any. This is also an instance of contextually-aware conversation as the user's response is only meaningful as a follow-up to a previous request, which the bot should remember and understand.

Example:

User: *"Turn on the light."*

Jarvis: *"Do you mean the living room light, kitchen light, bedroom light or hall light?"*

User: *"The living room light."* or *"The first."*

Jarvis: *"Sure, I've turned on the living room light."*

An important note about the example in this subsection is that this device specification technique is also possible in most of the following scenarios, anytime it is required to specify a device with which the system should interact. However, since the course of the dialogue would be very similar, this technique will not be demonstrated for any of the following use cases.

3.4.3 Use Case 3: delayed one-time action query

One-time action that instead of being executed at the time of the user query is delayed for a certain period of time ("in 30 minutes") or to a certain time ("at 10pm") or date ("tomorrow at 10am") specified by the user. In other words, this use case is an application of Use Case 1 with a specified delay.

Examples:

- *"Turn on the living room light in 30 minutes."*
- *"Turn on the kitchen light at 10pm."*
- *"Turn on the kitchen light tomorrow at noon."*

3.4.4 Use Case 4: delayed period action query

Certain actions that can be performed in IoT devices can be reversed. For example, if a switch is turned on it can be later turned off and if a thermostat is set to a certain temperature its original temperature can later be restored.

Because this property can be useful at times, this use case documents an example where a certain condition is made true for a period of time defined by the user. This can be useful for example if the user wants to keep a light turned on during the time he has breakfast or keep the windows closed during the night.

Examples:

- *"Turn on the bedroom light from 8am to 10am."*
- *"Open the bedroom window during the afternoon."*

- *"Turn on the hall light from tomorrow 10am to 11am."*

3.4.5 Use Case 5: daily repeating action query

This use case represents a direct action, as was seen on Use Case 1, that is executed every day at a certain time. This is the first example of a rule query, which means that it is automated indefinitely until the user modifies or cancels it. This is called a **rule** of the system as it will remain active as long as the system does.

Examples:

- *"Turn on the bedroom light everyday at 10am."*
- *"Close the bedroom window everyday at night."*

3.4.6 Use Case 6: daily repeating period query

This use case is very similar to Use Case 5 with the nuance of being a period action, which means it is only active for a time period defined by the user.

Examples:

- *"Turn on the bedroom light everyday from 8am to 10am."*
- *"Open the bedroom window everyday during the afternoon."*

3.4.7 Use Case 7: cancel last command query

This command is used to cancel a user's last command, regardless of the kind of command it is. If it is a direct action, that action is undone. If the command is the definition of a rule, that rule is cancelled.

Since this command can cause system rules to be changed it uses contextual awareness to confirm what command the user wants to cancel.

Example:

User: *"Cancel my last command."*

Jarvis: *"Are you sure you want to cancel the action: 'turn on the living room light from 10am to 11am'?"*

User: *"Yes."*

Jarvis: *"Sure, the command was cancelled."*

3.4.8 Use Case 8: event rule query

Creates a rule that executes a direct rule (similar to those found in Use Case 1) when a certain event occurs. This event is an event declared by one of the IoT devices known by the system, as documented by Mozilla's Web of Things protocol described in Section 2.3.4.

This command allows devices to operate based on the interactions of other devices which is the first step to create behavorial bridges between different devices that don't necessarily need to be phisically connected to one another.

Examples:

- *"Turn on the kitchen light if the kitchen motion sensor is activated."*
- *"Turn on the living room light if the living room chair pressure sensor is activated."*

3.4.9 Use Case 9: rules defined for device query

In order to understand how the system is setup it is essencial that the user can know what rules are defined for a certain device. This command allows the user to learn how the system is setup by asking what rules are defined for a device. As well be seen in Use Case 10, this will be very useful to modify or cancel specific system rules.

Example:

User: *"What rules are defined for the living room light?"*

Jarvis: *"You told me to turn on the living room light everyday at 10am."*

3.4.10 Use Case 10: change device rule query

The dialogue mentioned in Use Case 9 allows the user to know what rules are defined for a certain device. As a follow-up to the result of that query, the user may modify or cancel rules without the need to specify exactly which rule he wants to cancel because the bot has the contextual awareness required to know what the user means. With this follow-up command, rules can quickly be modified or cancelled.

Example:

User: *"What rules are defined for the living room light?"*

Jarvis: *"You told me to turn on the living room light everyday at 10am."*

User: *"Change that to when the living room motion sensor is activated.", "Change that to everyday at 9:30am."* or *"Cancel that rule."*

Jarvis: *"Sure, the rule was changed."* or *"Sure, the rule was cancelled."*

3.4.11 Use Case 11: causality query (single possible answer)

One of the most important features that the project provides is to query the system about the reason why a certain event happened or a certain condition is true. This can make it much easier for a user to understand how the system works. In the current voice assistants this feature is not available, and in VPPs this can only be done manually by checking every system rule to understand which rules may cause the consequence to be true.

The main problem with causality queries is that they may have complex answers or, in some case, multiple possible answers, which may make it hard for the system to decide which answer to give in order not to provide an answer that is too long for the user to understand.

Another issue that causality queries face is related with un-documented cause-consequence pairs, i.e. when the cause of a certain action is indirectly related to something else that happened in the system. One real world example of this would be how closing electric blinds could cause a room to get darker, therefore triggering some event on a luminosity sensor.

The different possible outcomes of causality queries are discussed by use cases 11 through 15, along with their expected behaviors and possible approaches.

In this command we explore a simple causality query with a clear and simple answer. In this case, there is exactly one system rule that causes the consequence queried by the user and the result is that same rule.

Previously defined rules:

- When bedroom motion sensor is activated, turn on bedroom light.

Example:

[Bedroom motion sensor is activated]

[Bedroom light turns on]

User: *"Why did the bedroom light turn on?"*

Jarvis: *"You told me to turn on the bedroom light when the bedroom motion sensor is activated."*

3.4.12 Use Case 12: causality query (parallel possible causes)

In this case, when the user queries the bot for the cause of a certain condition, there are multiple rules that could have caused that condition to be true. Therefore, there are two possible approaches to this issue. On one hand, the bot can return the earliest interaction that caused the queried condition. On the other hand, there might have been another following interaction that is more meaningful to the user as to why the condition is true. In that case, the bot could either reply with all possible causes of the condition, or use an heuristic to choose the cause that is more relevant to the user (e.g. in some causes, interactions that are caused by the system's rules rather than direct queries by the user are more relevant as an answer to this sort of query).

Previously defined rules:

- When bedroom motion sensor is activated, turn on bedroom light.
- When it's 7pm, turn on bedroom light.

Example (answer with earliest possible cause):

[(6:50pm) Bedroom motion sensor is activated]

[Bedroom light turns on]

User: (7:10pm) *"Why did the bedroom light turn on?"*

Jarvis: *"You told me to turn on the bedroom light when the bedroom motion sensor is activated."*

3.4.13 Use Case 13: causality query (documented chained causality, single answer)

In this example there is a set of rules that chain into each other, i.e., the consequence of one rule is the cause for another rule. Because of that, if a rule is triggered that has other rules leading up to it, the answer to why that rule's consequence happened might not be clear since although the rule's cause is the most direct cause to that consequence, there may have been other earlier interactions that lead up to it. From the standpoint of the user, it might be useful to not only know what was the exact rule that caused the consequence, but also the other rules that led to it.

This particular use case explores the possibility of the bot either returning the latest possible rule that caused the consequence or the complete list of events that led to the consequence.

Previously defined rules:

- When kitchen motion sensor is activated, turn on kitchen light.
- When kitchen light is turned on, turn on kitchen coffee machine.

Example (answer with latest possible cause):

[Kitchen motion sensor turns on]

[Kitchen light turns on]

[Kitchen coffee machine turns on]

User: *"Why did the kitchen coffee machine turn on?"*

Jarvis: *"You told me to turn on the kitchen coffee machine when the kitchen light turns on." or "The kitchen motion sensor turned on, which caused the kitchen light to turn on, which caused the kitchen coffee machine to turn on."*

3.4.14 Use Case 14: causality query (documented chained causality, contextual conversation)

As an alternative to what is done in Use Case 13, this scenario explores the possibility of the bot engaging in a contextual conversational with the user. In this case, the bot at first returns the most immediate rule that caused the consequence queried by the user, but then the user may ask more about what happened, in which case the bot responds with the complete list of events that led up to that consequence.

Previously defined rules:

- When kitchen motion sensor is activated, turn on kitchen light.
- When kitchen light is turned on, turn on kitchen coffee machine.

Example:

[Kitchen motion sensor turns on]

[Kitchen light turns on]

[Kitchen coffee machine turns on]

User: *"Why did the kitchen coffee machine turn on?"*

Jarvis: *"You told me to turn on the kitchen coffee machine when the kitchen light turns on."*

User: *"And why did that happen?"*

Jarvis: *"The kitchen motion sensor turned on, which caused the kitchen light to turn on, which caused the kitchen coffee machine to turn on."*

3.4.15 Use Case 15: causality query (undocumented chained causality)

Unlike the previous causality query scenarios, in this example the set of events that leads up to a certain consequence is not documented by system rules. In other words, the chain between a certain pair of rules is explicit, being an indirect consequence of the first rule.

As demonstrated by the example below, the consequence of the first rule is the kitchen window opening. Because of the increased wind caused by this action, movement of objects in the kitchen may cause the kitchen motion sensor to be activated, therefore activating the second rule. In this example, the first rule may be considered as the cause for the final consequence, and the system should notify the user of that.

This use case is intended as a point of discussion as to how the system should behave in this sort of scenarios, and the conclusions of that discussion are presented further in this document.

Previously defined rules:

- When the kitchen temperature rises, open the kitchen window.
- When the kitchen motion sensor is activated, turn on the kitchen light.

Example:

[Temperature rises]

[Kitchen window opens]

[Wind causes objects in the kitchen to move]

[Kitchen motion sensor is activated]

[Kitchen light turns on]

User: *"Why did the kitchen light turn on?"*

Jarvis: *"You told me to turn on the kitchen light when the kitchen motion sensor is turned on."*

User: *"And why did that happen?"*

Jarvis: *"It might be because the kitchen window was opened because the temperature rose."*

3.4.16 Use Case 16: alias definition query

In certain cases, the correct physical actions or conditions on devices are not the most intuitive to use in a natural language interaction. For example, if a user says "open the kitchen windows when it gets hot", the system will not understand because "hot" is not a condition that it understands as it is not a valid state for a temperature sensor.

To help with that, the user may create aliases for both actions or events that are then used in any of the other commands. Aliases can be created manually by the user, or automatically when an unknown command is given. The two examples below demonstrate the creation and usage of aliases:

Example 1 (manual creation):

User: *"Create an alias for when it gets hot."*

Jarvis: *"What do you want that alias to represent?"*

User: *"The living room temperature is above 20 degrees."*

Jarvis: *"Sure, I've created an alias for when it gets hot."*

(...)

User: *"Open the kitchen windows when it gets hot."*

Jarvis: *"Sure, the rule has been created."*

Example 2 (automatic creation):

User: *"Open the kitchen windows when it gets hot."*

Jarvis: *"I'm not sure what 'when it gets hot' means. Do you wish to create an alias for that?"*

User: *"Yes, create an alias for when the living room temperature is above 20 degrees."*

Jarvis: *"Sure, I've created an alias and setup your rule."*

Aliases are intended as a stretch goal for this project and therefore should be seen as a proof of concept of a sort of feature that can deeply increase the comfort of use of voice assistants for IoT management.

3.5 Research Questions

Based on the use cases mentioned above, we establish that the research questions of this dissertation are the following:

1. **"How can delayed, period and repeating actions be implemented?":** we aim to understand how features demonstrated by use cases 3 through 6 can be implemented in a conversational bot. These are simple actions that can prove to be very helpful in a voice assistant.

2. **"Can we use contextual awareness for IoT management?":** with examples such as use cases 7 and 10 we believe that contextual awareness in a conversational assistant can be of great help for the management of an IoT system. Contextual awareness can prevent the user from having to repeat certain commands or phrases, increasing the system's comfort of use.

3. **"Can a conversational assistant handle event management?":** this question focuses on allowing interaction between different IoT devices through the use of a conversational IoT assistant, as demonstrated by use case 8. With events, it becomes possible to have separate devices depend on each other's actions, which increases the complexity and capabilities of the system as a whole.

4. **"How can a conversational bot solve causality queries?":** use cases 11 through 16 showcase the many different ways a simple "*How did that happen?*" query can be answered. Through this question we aim to explore different techniques on how to answer this sort of queries.

3.6 Validation

As stated above, the main purpose of this project is not only to provide users with management features that aren't yet available, but also to make the access to these features easier and simpler.

With that in mind, the successful implementation of this project will me measured in two ways:

On one hand, the project should be able to fulfill the use cases mentioned in Section 3.4. These features are the main focus point of the project and the defining characteristic that makes it innovative and useful. Therefore, each use case that is not fulfilled represents an opportunity that was not properly explored, being a failure for the project. This list of use cases represents an objective way to compare the project with the tools that already exist in terms of capability and handled complexity.

On the other hand, a user study is the best way to evaluate the comfort of use of the project. The main goal of the study is to understand if it is easier to perform the complex management use cases using this project when compared to the other tools.

Together, both this measurements provide valuable insights on the supported complexity of this project as well as its advantages in terms of simplicity and ease of use.

To further validate the success and quality of the project, as well as its scientific significance, there will be attempts to submit related papers or articles to conferences such as Interact[1], CASE[2], IEEE RO-MAN[3] and ICARCV[4].

3.7 Summary of Contributions

The expected contributions of this project reflect the answers to the research questions mentioned in Section 3.5. Therefore, the contributions of this project all relate to the capabilities of the Jarvis chatbot that are innovative and new in the domain of IoT management chatbots, which are:

1. **Complex management commands such as delayed, period and repeating actions:** as current chatbots are limited to simple direct queries such as "*turn on the light*", Jarvis presents more complex actions such as delayed or repeating actions.

2. **"Take advantage of contextual awareness for quicker management":** if a user has to repeat commands to manage an IoT system, that task becomes repetitive. With the use of contextual awareness, queries become interconnected and therefore, with Jarvis, commands such as "*change that to 5pm*" work because the system understand they refer to previous queries.

3. **"Events for device interoperability":** with current IoT chatbots it is impossible for a thermostat to operate with a light because they are unaware of each other's existence. Jarvis

[1] http://www.interact2017.org/
[2] http://case2018.org/
[3] http://ro-man2018.org/
[4] http://icarcv.org/2016

supports the existence of events, a mechanism through which a device's actions can depend on another device without them ever being connected. This is demonstrated by queries such as "*turn on the light when the chair pressure sensor is on*".

4. **"Answer queries about causality":** neither current chatbots nor visual programming platforms can tell the user why something happened. Chatbots simply don't support it, and VPPs require the user to look through all the configured rules in order to find the right one. With Jarvis, all that's required is to ask "*why did the light turn on*" for it to explain the rule that caused that condition.

3.8 Conclusions

The purpose of this project is not to reimplement features that are already available in current chatbots such as answering questions about the weather or celebrities. Instead, the focus is to specialize in IoT management features that are not currently available and that can make IoT easier to use, especially by users that are not familiar with technologies.

Through using a conversational interface, this project provides new and more complete ways to manage an IoT system while retaining the comfort of natural language interactions. The possibility of creating delayed or repeating rules, managing a system through events and the support for causality queries make Jarvis a more complete solution for IoT management.

Chapter 4

Implementation

This chapter describes the entire solution developed for the problems mentioned in Chapter 3.

At first, the high-level overview of the developed solution is described, briefly going through the technologies and methodologies chosen to make the entire system work.

Then, each of the software components is thoroughly analyzed so that the project's architecture and algorithms are clear and help to understand how the whole system is able to deliver the proposed features. At this point, the software's architecture and organization goes through a critical process to understand where it works well, where it doesn't and where it could be improved for real usage.

At the end, an example of how this project was tested and simulated will be presented so that it is easy for anyone to replicate the project's results with the given source code. This is an open source project, that is available in its GitHub page[1].

4.1 Overview of a Chatbot

When it comes to software components, there are a lot of ways to implement a regular chatbot and each implementation may be very different in a lot of ways. However, since Dialogflow is mentioned in many news sources as a reference for a top chatbot framework [Dav17] [Tec17], it will be used in this document as a reference high-level structure for a chatbot.

Google Developer Expert Joe Birch wrote a great article on Medium that thoroughly describes how a Dialogflow agent is generally structured (at the time of the writing of the article, Dialogflow was still called API.AI) [Bir17]. The article explains a lot of key concepts of Dialogflow as well as how to use them in the development of a bot, but it also explains how the fulfillment flow in a bot works. That flow is shown in Figure 4.1.

[1] https://github.com/andrelago13/jarvis

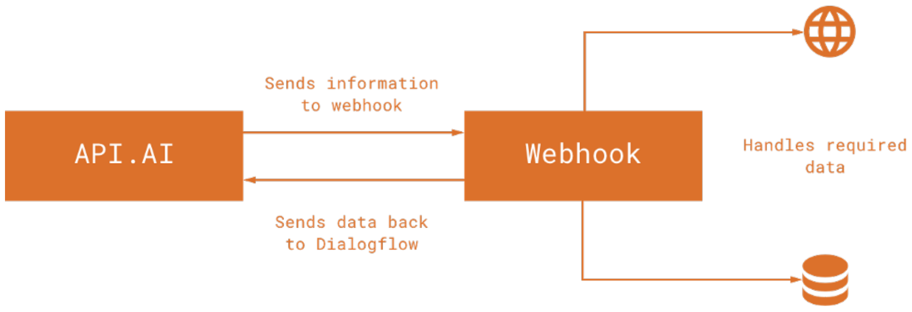

Figure 4.1: Joe Birch's visualization of the Dialogflow fulfillment

Birch's diagram shows how Dialogflow's parsed information can be sent to a fulfillment webhook. The backend that receives this data may do whatever it is supposed to do, be it store the data or interact with other systems based on the data.

When it comes to IoT bots, like most software applications, there may be several possible architectures for the system as a whole which depend a lot on the type of use and available resources for the system. Figure 4.2 illustrates a possible architecture for a system intended to be used as a driver for this project: a small number of IoT devices and a backend that receives under 1 QPM on average, and around 20 QPM at peak usage.

- **User Interface (UI) ~** the UI represents any tool the user may use to interact with the chatbot. When this is a text interface (Slack, Facebook Messenger, . . .), the only responsibility of the UI is to send the text written by the user to the Language Processing Backend and then present the backend's response to the user. In other cases, such as the Google Assistant or Alexa, the UI is also capable of voice interaction. In these cases, the device has Speech Recognition capabilities, being able to translate the user's audio to a string which is then sent to the backend. The backend's string response is then converted to audio through Text Synthesization processes to make the interaction sound like a human.

- **Dialogflow ~** the Dialogflow system is used to process the user query strings and understand their intent. Once the intent is determined, a fulfillment request is sent to the Language Processing Backend so that the proper IoT interactions may take place. The response from the backend is then sent to the UI that initially sent the request to Dialogflow.

- **Language Processing Backend ~** the language backend is responsible for making sure the user query is correct and then perform the required actions. For example, if the query is "turn on the living room light" the backend should communicate that action to the Messaging Queue and return a message of success. In the opposite case, if the query is "turn on the windows" the backend should understand this is an invalid query and return an appropriate error message. The backend sees the messaging queue as a single point of contact to the IoT System Controllers, which means it does not need to be concerned about the topology or specific protocols of each controller.

- **Messaging Queue ~** the messaging queue is not essencial to make this type of system work,

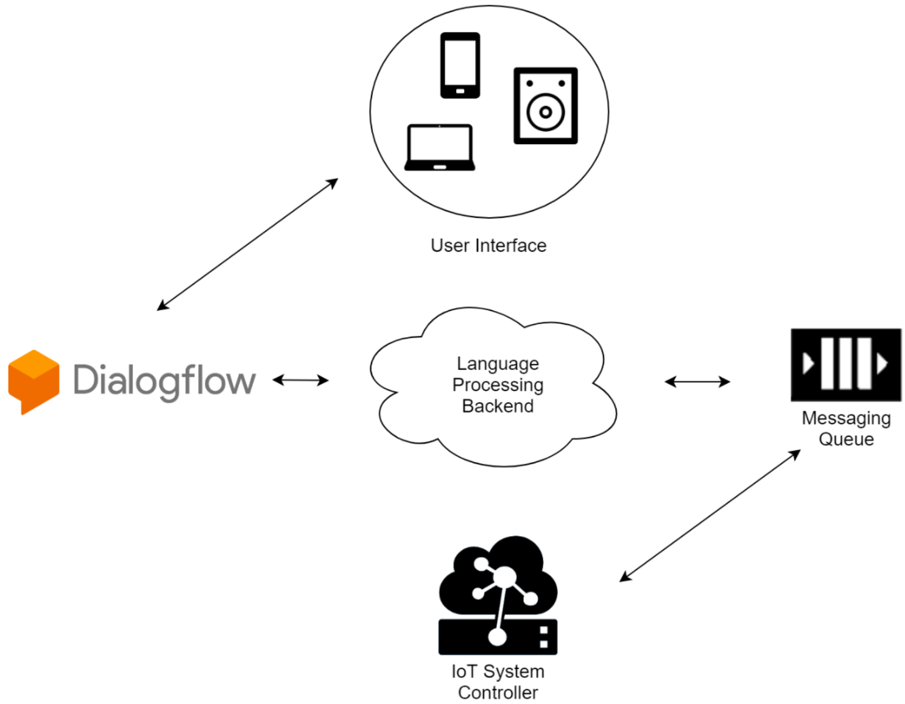

Figure 4.2: Overview of a chatbot suitable for this project

but it helps in making the IoT Controller layer more scalable and immutable to the language backend. In sum, the messaging queue is a system know by both the language backend and all the IoT System Controllers. In it, there are messaging channels for each IoT device's property or action to which the clients publish or subscribe, depending on the intended use. For example, the backend publishes on a device's action channel to make it perform a certain action, but the controller might publish to a sensor's property channel to notify recipients of a change in the sensor's value. Because of the messaging queue, the communication with IoT devices for the language backend regardless of whether there is a single controller or multiple controllers spread across different network locations.

- **IoT System Controllers ~** these are networked devices that write or receive messages from the messaging queue respective to the IoT devices they control. These operate with low-level technologies that interact with the actual IoT devices to perform any action or query.

This architecture is fitting for this project's scope as it fits Dialogflow's model and is able to support the desired scale under which the project will be tested. However, the core part of the entire system is the fulfillment backend, which could easily be scaled for larger systems with minor changes to the data storage and IoT communication modules.

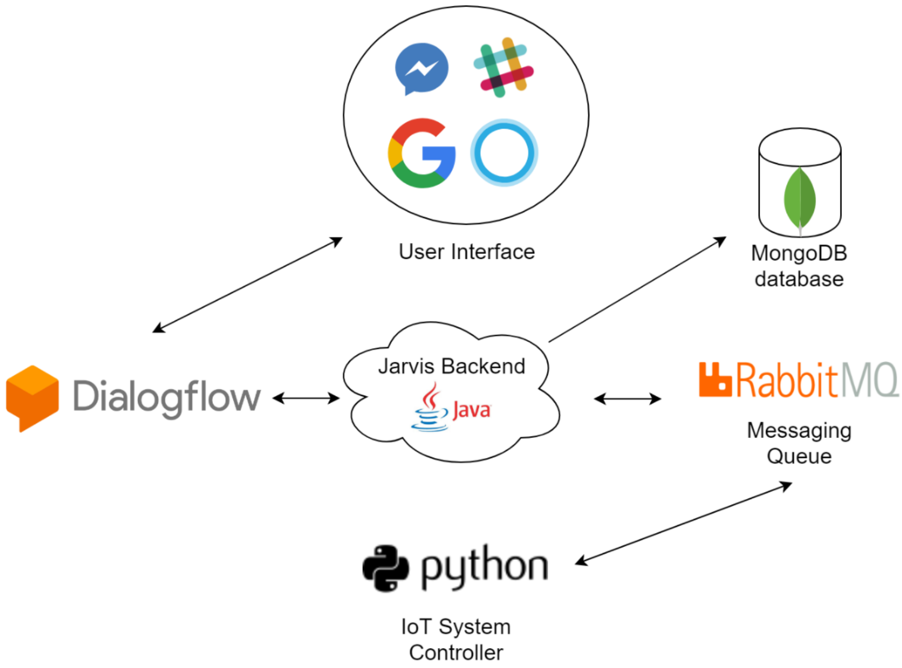

Figure 4.3: Jarvis overall architectural components

4.2 Solution Overview

As mentioned in Section 3.2, the main purpose of the developed chatbot is to not only make it easier to use features of already available consumer products, but also to enable new functionalities that aren't yet available.

Figure 4.3 illustrates the chosen architecture for Jarvis. It follows the same overall structure shown in Figure 4.2, specifying which technologies were used for each component.

When it comes to the NLP engine, Dialogflow was chosen due to the fact that not only is it mentioned in several articles and lists for top bot frameworks [Dav17] [Tec17], but also because it integrates with multiple UIs. Dialogflow has built-in integrations [2] with UIs such as the Google Assistant, Slack or Facebook Messenger, which makes it easy to support both text and voice interactions with users.

Another interesting feature of Dialogflow is ML expansion[3] (Machine Learning expansion). This means that the provided sample user queries are used with ML models to generate similar user queries, therefore increasing the range of accepted queries with similar structure and words.

Together, these characteristics make Dialogflow very useful in creating the top layer of Jarvis' architecture since it provides both the UI and NLP engine.

The result of the processing done by Dialogflow's engine is then sent to the Jarvis Backend for further processing. This backend component is a Java EE[4] server that provides a simple REST API

[2] https://dialogflow.com/docs/integrations/
[3] https://dialogflow.com/docs/training
[4] http://www.oracle.com/technetwork/java/javaee/overview/index.html

to which Dialogflow sends its requests. This backend then processes the parsed input generated by DialogFlow in order to understand what is the user's intent and how it should act upon that request. If the request requires actions to be made to the IoT system, the backend sends a message to a messaging system that will then communicate with the IoT devices. On the other hand, if the user request requires a database or system query, then the backend merely retrieves the information it requires so that it can provide the user with a response. In the scope of this project, this backend can be considered a single point of failure of the system. However, because it is a stateless service, it could easily be scaled into a more reliable system.

To persist the data relative to both the IoT devices and the user actions, the backend uses a MongoDB[5] database. This was the chosen technology because it is the most popular document store database system [6] and, because of having a key-document structure, is a great fit for storing JSON objects that represent the IoT devices (as seen on Section 2.3.4).

The messaging queue acts as an intermediate system to provide communication between the Jarvis backend and the IoT controllers. For this component, RabbitMQ[7] was the chosen technology because it uses scalable techniques for creating a distributed messaging system. Also, RabbitMQ uses communication channels [8] to distribute messages from publishers to consumers, which is ideal for the type of exchanges that are described in the sections below. This queue is a simple yet very important part of the system because it allows the backend to be unaware of the location or architecture of the IoT controllers. Instead, the backend communicates to a single system that may be connected to one or many controllers. Although it was not used for this project, this wouldn't be considered a single point of failure of the system because it is also possible to create a distributed RabbitMQ broker[9].

On the final end of the entire system are the IoT System Controllers. These are light Python scripts that must run on networked devices (e.g. Raspberry Pis) that are connected via hardware to the actual IoT devices in order to control them or be able to read their status. These controllers are clients of the messaging queue so that they read the actions published by the backend and publish the changes in the status of the devices.

Combined, these technologies allow Jarvis to perform the use cases mentioned in Section 3.4, and each of them is thoroughly detailed and explained in the following sections.

4.3 Component Overview

Throughout the following sub-sections each of the software components pictured in Figure 4.3 are thoroughly described. By the end of this section, each of the components should be individually understood as the assembly of the entire system is detailed in the following section.

[5] https://www.mongodb.com/
[6] According to DB-Engines' ranking of almost 300 systems: https://db-engines.com/en/ranking
[7] https://www.rabbitmq.com/
[8] https://www.rabbitmq.com/tutorials/amqp-concepts.html
[9] https://www.rabbitmq.com/distributed.html

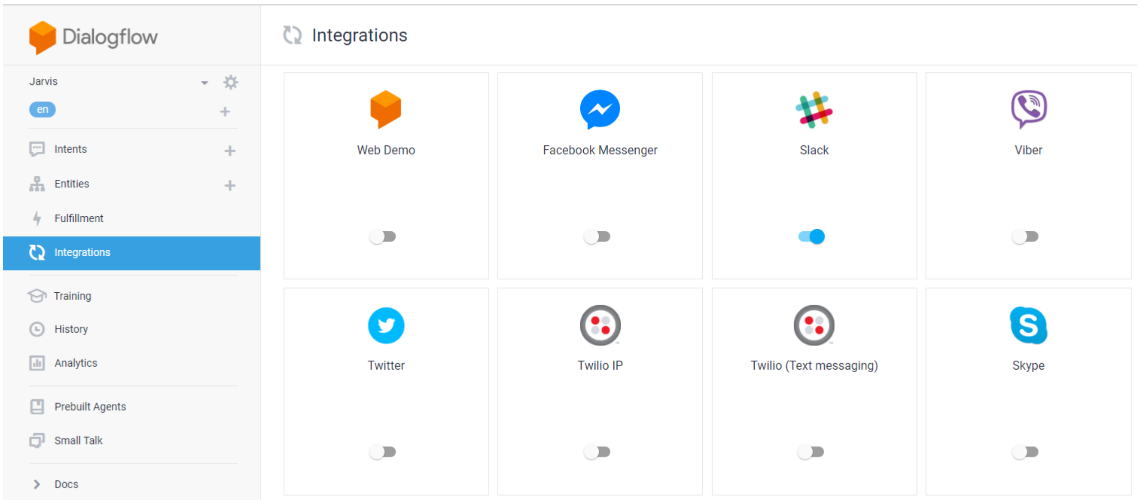

Figure 4.4: Slack integration enabled in Dialogflow's integrations panel

4.3.1 User Interface

The user interface is the means through which the user interacts with Jarvis. Due to the use of DialogFlow, there is no need to create specific interfaces for Jarvis as this platform provides built-in integration with multiple popular interfaces. In this case, the main interfaces used were Slack and the Google Assistant so that both text and voice interfaces were covered.

Slack[10] is a very powerful team-communication tool used worldwide for team collaboration and efficient communication. Slack provides direct communication with individuals or groups of individuals, but also channeled conversations. In this case, users may create public or private messaging channels in which they can write themed content for other members of the channels to read.

Slack's API[11] allows developers to create bots that talk to users in a natural language fashion. This is already used by many applications available online[12]. Dialogflow uses this same API[13] to create Slack bots through which Dialogflow bots communicate. This integration allows users to create bots with Dialogflow that are then easily integrated with Slack with the click of a button.

To set up the Slack integration two steps are required. First, through DialogFlow's integrations panel, shown in Figure 4.4, the user must enable the integration toggle. Then, a web page allows the user to specify a Slack domain where the bot should be made available.

Once this integration is setup, it is possible to chat with the bot via Slack's UI the same way a user would chat with another user. Figure 4.5 illustrates how a conversation with the bot would look like using Slack's Android app[14].

[10] https://slack.com/
[11] https://api.slack.com/
[12] https://slack.com/apps/category/At0MQP5BEF-bots
[13] https://dialogflow.com/docs/integrations/slack
[14] Slack App version 2.59.0 running on a Google Pixel phone with Android P Beta

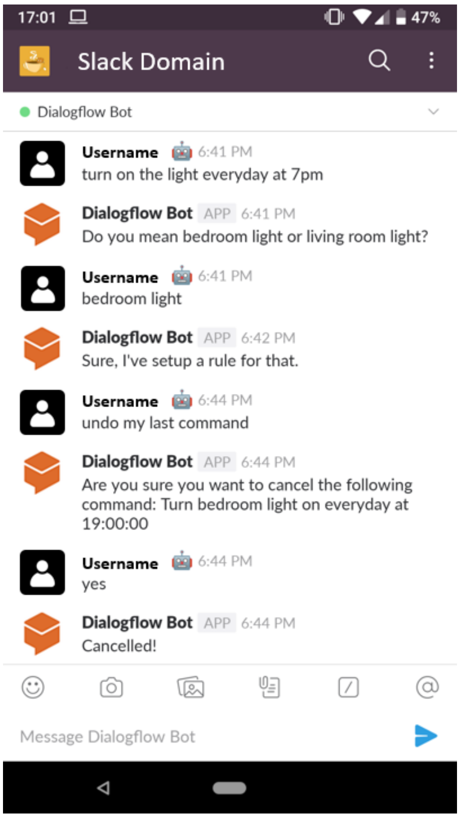

Figure 4.5: Chat with Jarvis bot using Dialogflow's Slack integration (Android app version 2.61.0)

Slack's UI was also used for debugging the project. Through the Slack API developer console it is possible to generate URLs to which a system may send POST requests with messages that are then displayed in a specific channel of a specific Slack domain. This tool allowed Jarvis debug messages to be sent to a Slack channel so that interested parties may be notified every time an instance of the Jarvis backend is launched, when an unexpected exception happens or if there is a problem with the database connection.

The integration with the Google Assistant is also made easily due to Dialogflow's built-in integration system. In fact, this integration is already enabled by default and is even prepared for testing live changes made to the Dialogflow bot. To use it, all the user needs to do is click the "Google Assistant" section of the Dialogflow integrations panel already shown in Figure 4.4 and then the popup of Figure 4.6 will be shown. In this popup, the "test" button takes the user to a simulator where he can simulate the interactions as if they were being made in a device with the Google Assistant, which includes text, voice and gestures. The "manage assistant app" button in that same popup allows the user to make the necessary configurations to take the bot into production.

Once this configuration is complete, the user may use other Google Assistant enabled devices with his account to communicate with the bot, only needing to start with a query such as *"Hey Google, talk to Jarvis"*. Using the simulator mentioned above, a chat with the Assistant looks like

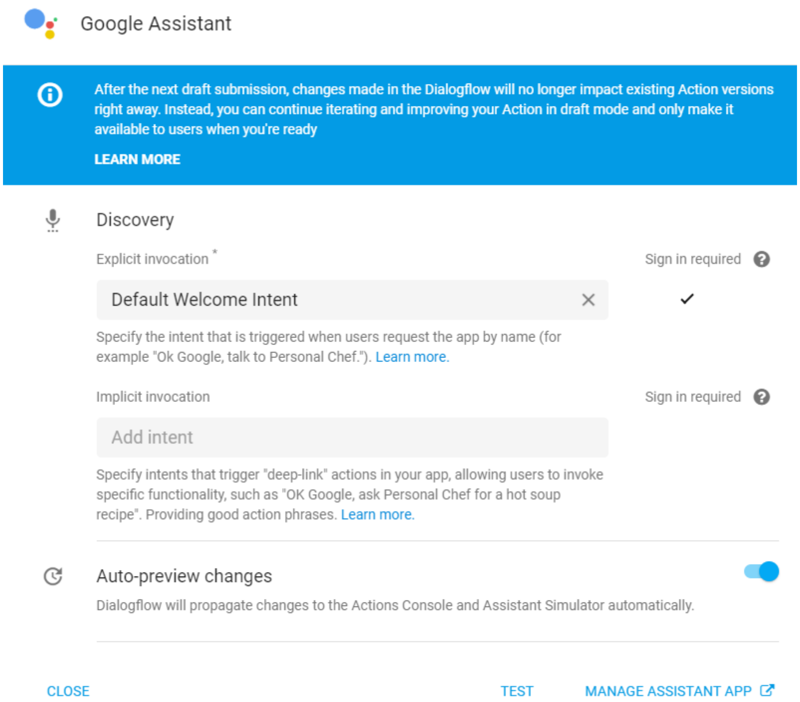

Figure 4.6: Google Assistant configuration popup in Dialogflow's integrations panel

what's shown in Figure 4.7. This UI is used on smartphones even if the interaction is made using the voice, so that the user knows what the system understood. On the right side of the figure the simulator shows the user queries, followed by the bot's responses on the left. On the bottom there is a panel to type text queries or enable the microphone to speak voice queries.

Regardless of which of these interfaces is used, the strings representing the exact user query are sent from the interface to DialogFlow's backend. Because Dialogflow is being used, there is no need to implement any Speech Recognition techniques because they are already implemented. This means that Dialogflow's backend receives raw strings that represent the user queries, which are then analyzed using Natural Language Processing techniques.

4.3.2 Dialogflow Backend

This system receives the user queries from the different user interfaces and parses them into machine understandable code.

Upon receiving a request, there are two things DialogFlow can do: either respond with an automatic response or send the parsed request to a fulfillment backend (in this case, the Jarvis Backend) which will then process the request and return the desired response.

There are a few key concepts that are important to understand in Dialogflow in order to make it operate the best possible way. All the information below can be found in the official Dialogflow

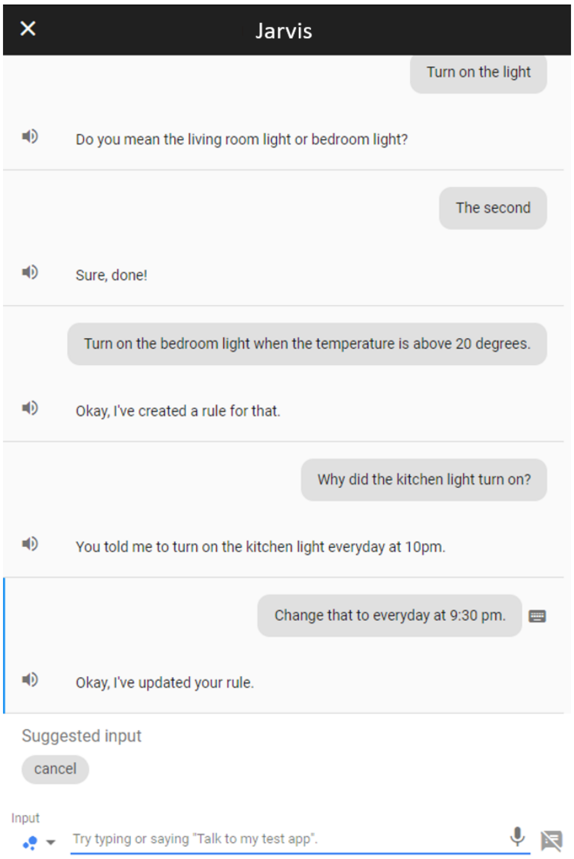

Figure 4.7: Chat with Jarvis using Google Assistant's online simulator

documentation[15].

- **Fulfillment Backend:** a secondary user-created server that provides a REST API for Dialogflow to send processed messages in order to receive the intended response to the user. This backend receives the result of the processing done by Dialogflow, being able to perform other actions that Dialogflow can't perform (e.g. interact with IoT devices).
- **Request:** plain string that represents a user query. An example would be "turn on the living room light".
- **Entity:** symbol that can be represented by different literal strings. As an example, there may be an entity called "toggleable-device" which may be represented by "living room light" or "kitchen light". Additionally, entities may be represented by other entities, which means that an entity "device" could be represented by the entity "toggleable-device" which then may be represented by certain strings. Entities are represented in DialogFlow with the use of the "@" symbol ("@device"). The purpose of entities is to act as arguments of a request, as will be seen further in this section.
- **System entity:** a system entity is a regular entity that instead of being manually defined

[15] https://dialogflow.com/docs/getting-started/basics

by the user is already defined by the system. There are multiple useful system entities to represent values such as time, temperature, color or currency.

- **Intent:** represents a set of example requests that may contain multiple entities. Intents are defined by the user and, ideally, each represents one type of query by the user. As an example, an intent named *"Turn on/off device"* may be represented by the requests *"turn the @device on"* and *"turn the @device off"*. In this case, if the request is *"turn the kitchen light on"*, the DialogFlow engine will understand that *"@device"* is represented by *"kitchen light"* and provide that information to the fulfillment backend. Additionally, in the example *"on"* and *"off"* could also be represented by an entity *"@on-off-status"*. Also note that using a backend for fulfillment is optional and defined manually for each intent.

- **Context:** contexts are used to allow intents or requests to depend on previous instances, enabling for the creation of context-aware interactions. A context is defined by a simple string (e.g. "device-choice") and can be used as input and/or output for an intent. A context has a lifespan which determines for how many consecutive requests it will be active unless otherwise specified by the fulfillment backend. If an intent has a context as output, the following requests will carry that context in their parameters. On the other hand, if an intent has a context as input, a request will only be parsed into that intent if it carries that context in its parameters (e.g. if a previous intent had that same context as output). Contexts may also carry parameters that help in processing a follow-up request. The parameters are a simple list of key-value pairs where the key must always be a string. Whether or not a context is the output of an intent can either be pre-defined in DialogFlow or defined by the fulfillment backend. An example of a use of a context would be if a user query is *"Who is LeBron James?"*. In this case, the system replies with *"LeBron James is a professional basketball player"* and stores that information in the context parameters. That way, if the following request is "How tall is he?" the system can use the context parameters to know that *"he"* represents *"LeBron James"* and therefore can answer the request properly.

When it comes to entities and intents, DialogFlow provides an optional useful feature called Machine Learning (ML) expansion. If the user enables this feature, DialogFlow will use ML to look for string values that are similar to those defined by the user while still keeping the same meaning. For example, if this feature is enabled, if the user defined *"turn on the light"* as an intent, DialogFlow could recognize *"turn the light on"* as an instance of the same intent.

In the case of Jarvis, there are multiple intents, entities and contexts defined to make the entire system work. Figure 4.8 illustrates the main entities defined for Jarvis in Dialogflow. As it is shown, entities may be used by other entities and can also contain text. In the case of *"@action-on-off"*, the entity is defined by a text similar to a user request and contains two other entities.

Multiple intents were defined in order to represent the multiple supported user queries and make use of the entities shown in Figure 4.8. Below is a list of the main intents defined as well as their relevant details. Note that when entities are used in intents, each entity must be given an alias name that is used for that entity's representation in the JSON that is sent to the fulfillment

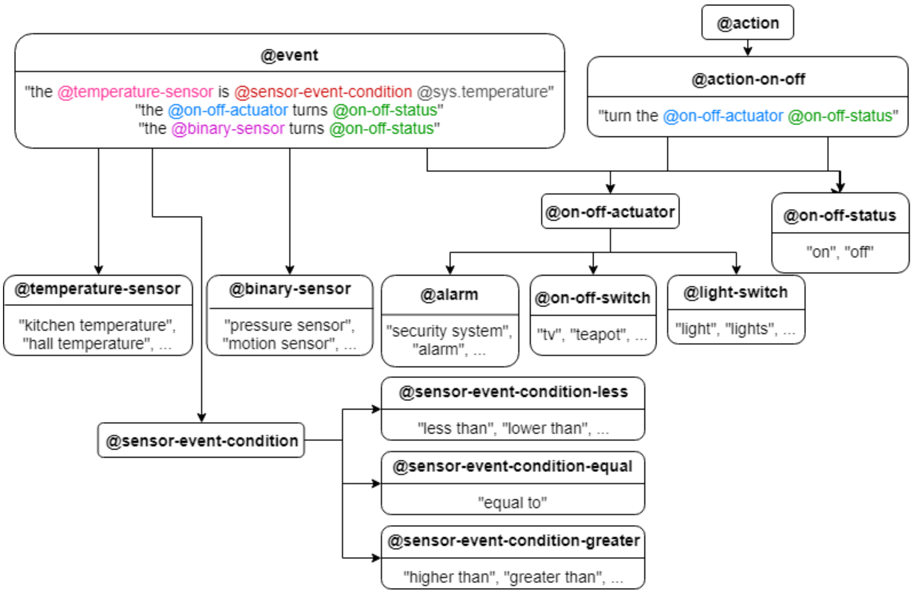

Figure 4.8: Main entities defined in Jarvis' Dialogflow project

backend. Because of that, if an intent is defined by *"@entity:alias"*, *"entity"* is the entity's name and *"alias"* is the name that should be used in the JSON (as is seen further in this section).

- **Direct Action**
  - **Usage:** single direct action that happens instantly. The example JSON provided below is an example of when the specified device is ambiguous (there may be several "light" devices), and therefore contains a context that allows the follow-up request to specify the device.
  - **Definition:**
    * "@action:action"
  - **Possible output context:**
    * *"confirm-thing-choice"* (if specified device is amiguous)
  - **Example:**
    * *"Turn on the light."*
  - **Generated JSON:** Appendix A, Listing A.1
- **Delayed Action**
  - **Usage:** single delayed action that happens after a certain time period.
  - **Definition:**
    * "@time:time @action:action"
    * "@action:action @time:time"
  - **Example:**

    - *"Turn the light on tomorrow at 5pm"*
  - **Generated JSON:** Appendix A, Listing A.2
- **Confirm Thing Choice**
  - **Usage:** 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. An example of the system's response previous to this query would be *"Do you mean the living room light or bedroom light?"*. Note that because this intent has an input context, it may only be used if that context was enabled by a previous query. Also note that the example provided in the appendix does not have a context because it is intentionally reset by the fulfillment backend as the action can now be considered complete.
  - **Definition:**
    - "@thing:thing"
    - "the @sys.ordinal:ordinal"
  - **Input context:** *"confirm-thing-choice"*
  - **Examples:**
    - *"The third."*
    - *"The bedroom light."*
  - **Generated JSON:** Appendix A, Listing A.3
- **Repeating Intent**
  - **Usage:** used for defining a rule for an action that should be performed everyday.
  - **Definition:**
    - "@action:action @periodtime:periodtime"
  - **Example:**
    - *"Turn on the light everyday at 5pm"*
  - **Generated JSON:** Appendix A, Listing A.4
- **Event Intent**
  - **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 on the bedroom light when the living room light turns off"*
  - **Generated JSON:** Appendix A, Listing A.5
- **Why did something happen?**

- **Usage:** used when the user wants to know why a certain condition is true or why a certain action took place.
- **Definition:**
  * "Why did @action-past:action-past ?"
  * "Why has @action-past:action-past ?"
- **Example:**
  * *"Why did the living room light turn on?"*
- **Generated JSON:** Appendix A, Listing A.6

• **Alias Intent**
- **Usage:** used for the user to create an action/event that associates to a custom phrase.
- **Definition:**
  * "make an alias for @sys.any:alias"
  * "create an alias for @sys.any:alias"
- **Example:**
  * *"Create an alias for turn up the heat"*
- **Generated JSON:** Appendix A, Listing A.7

• **Alias Intent | Set Alias Type**
- **Usage:** as the follow-up request to the Alias Intent request, this is used to confirm what the alias being created should represent.
- **Definition:**
  * "execute @action:action"
  * "do @action:action"
  * "when @event:event"
- **Example:**
  * *"Execute turn on the light"*
- **Generated JSON:** Appendix A, Listing A.8

• **Rules Defined**
- **Usage:** used for the user to know which rules are defined for a device, and possibly change them.
- **Definition:**
  * "which rules have i defined for the @thing:thing"
  * "what rules have i defined for the @thing:thing"
  * "tell me the rules defined for the @thing:thing"

    * "tell me the rules for the @thing:thing"
  - **Example:**
    * *"What rules are defined for the living room light?"*
  - **Generated JSON:** Appendix A, Listing A.9

- **Rules Defined | Change Single Rule**
  - **Usage:** possible follow up to the query from the Rules Defined intent. With this query, the user may change a rule that is active for a certain device by specifying the change on the rule.
  - **Definition:**
    * "make that when @event:event"
    * "change that to when @event:event"
  - **Example:**
    * "Change that to when the bedroom light turns off."
  - **Generated JSON:** Appendix A, Listing A.10

- **Cancel Command**
  - **Usage:** used to cancel the last user command. If that was a direct action command, the action is undone. If the command was a rule command, the rule is cancelled. Note that this cancellation requires confirmation with the follow up request in the Confirm Cancel intent.
  - **Definition:**
    * "Cancel my last command"
  - **Output context:**
    * *"confirm-cancel-event"*
  - **Example:**
    * *"Cancel my last command."*
  - **Generated JSON:** Appendix A, Listing A.11

- **Confirm Cancel**
  - **Usage:** used to confirm the cancelled command mentioned in the Cancel Command intent.
  - **Definition:**
    * "Sure"
    * "Yes"
  - **Input context:**

        * *"confirm-cancel-event"*

  - **Example:**
        * *"Yes"*

  - **Generated JSON:** Appendix A, Listing A.12

The meaning of each field in the JSON objects generated by Dialogflow is described in the official documentation [16], but below is a summarized explanation of each of them:

- **'result':** contains the result of processing the query's entities. In this field it is possible to find the exact user request ("*resolvedQuery*") as well as the present entities along with their values ("*parameters*").
- **'contexts':** array of contexts carried as output from previous requests.
- **'metadata':** this field contains information relative to the intent the DialogFlow engined associated the query to. Both "*intentId*" and "*intentName*" are useful for the Jarvis backend so that it knows what to do with the query parameters. In other words, the metadata specifies the intended action or query that the backend should perform.
- **'fulfillment':** specifies a list of "*messages*" that the DialogFlow engine provides as default answers to the query. These are usually overridden by the Jarvis backend to match the query's result.

All the intents above have fulfillment enabled, which means that the JSON objects listed in Appendix A are sent to the Jarvis backend which responds with the response it wants the UI to present to the user. The response must follow the format specified by Dialogflow's documentation[17]. Note that after the project began, Dialogflow has released a newer version of its API[18] (V2) but this project uses the older version (V1).

4.3.3 Jarvis Backend

The Jarvis backend is a JavaEE[19] application that provides a REST API to which Dialogflow requests are sent via a POST URL to compute the replies to user queries as well as perform the required actions on IoT devices.

For each of the intents mentioned in Section 4.3.2 the backend has an equivalent class that is responsible for parsing the request and generating a response. Upon receiving a request, each of the intent classes is responsible for validating the request parameters, to make sure that not only all required parameters are present (e.g. device names or desired action) but also that the specified devices, if any, is clear and unique. If the request has errors, an appropriate and explanatory response should be returned. If the parameters are valid but the intended device is unclear (e.g. user wants to turn on the "light" but there is a "living room light" and a "bedroom light"), the

[16] https://dialogflow.com/docs/fulfillment
[17] https://dialogflow.com/docs/fulfillment
[18] https://dialogflow.com/docs/reference/v2-agent-setup
[19] http://www.oracle.com/technetwork/java/javaee/overview/index.html

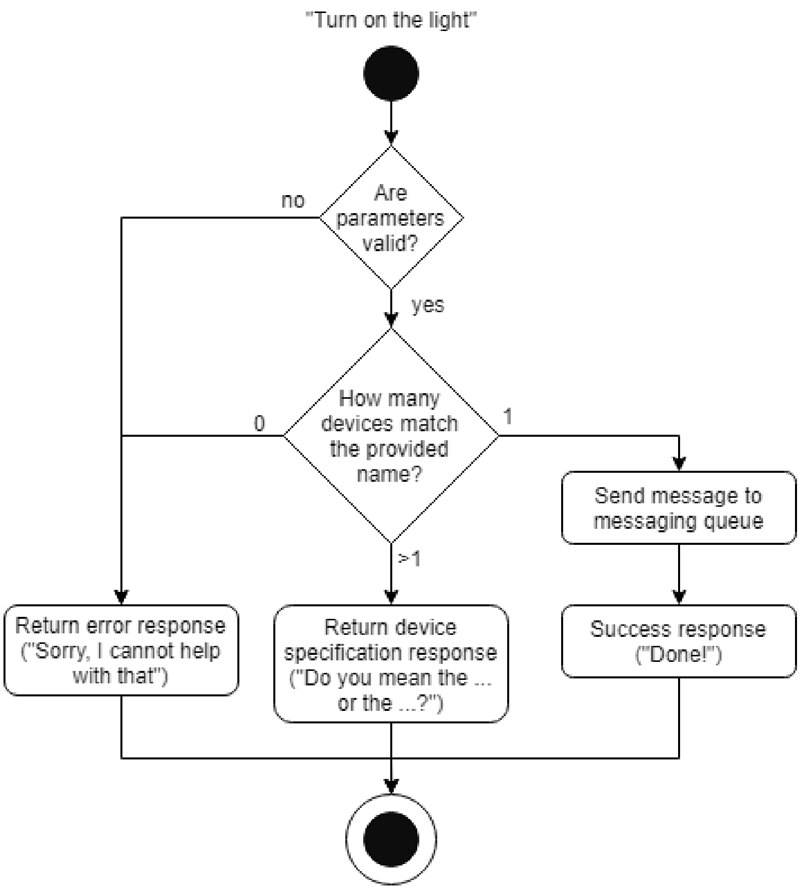

Figure 4.9: Sample activity diagram for the parsing of an intent

device specification context should be added and the response should ask the user to specify the desired device.

Figure 4.9 demonstrates a sample activity flow for one of the intent classes, in this case, given the query "*Turn on the light.*".

To perform the logs of commands and user commands, all actions that can be performed are represented as commands, therefore using the Command[20] design pattern. Each command has "execute" and "undo", implemented in a way that calling "undo" after "execute" should return the system to its original state. This pattern can be used to represent both user commands and device commands. The backend also contains an engine that is responsible for calling the execute and undo methods of the commands, so that their logging is centralized in a single class.

This engine is a crucial part of the system as it centralizes the main methods related to the functioning of the whole system, such as command execution, logging and scheduling or database initialization and population.

With most intents, such as direct actions or "why did something happen?" queries, the effects are immediate and require no delayed activity from the backend, except for period actions, rules

[20] http://w3sdesign.com/

and events. These require a special engineering so that they can perform actions on the backend without the need of a request to trigger them.

As mentioned in Section 4.3.2, a period action is an action that must be done and then undone after a certain period of time. This can happen as a result of a query such as "*Turn on the light from 4pm to 5pm*". As a result of this sort of query, a command is generated. This command includes an implementation of a state machine in which the state tells whether the first action (4pm) was not executed yet, the first action was executed but the second (5pm) wasn't or both have been executed. Therefore, when the command is executed, it schedules a Java thread to be run at 4pm, at which time the command is executed again, changing its state and executing the 4pm action. At this point, a second thread is scheduled for 5pm to execute the second action and move the command to its final state. The use of the Command pattern is useful as it abstracts from the engine the notion of direct or delayed commands.

The same concept is used for system rules, which are used for queries such as "*Turn on the light everyday at 5pm*". In this case, at first, a thread is scheduled for the following "5pm" to execute the command for the first time. From then on, everytime the command is executed it schedules a new thread for the following 5pm, but never leaves this state of rescheduling. The only way to cancel this action is to call the "undo" method which cancels the currently scheduled thread.

This thread scheduling mechanism does not work for events because there is not a known time at which these are executed. Events are a result of queries such as "*Turn on the light when the sensor is activated*". In this case, it is necessary for the system to listen to the messaging queues of the device actions and states. As will be mentioned in Section 4.3.5, each device has messaging queues that relate to its actions and state changes. Messages are published in the latter by the device controller everytime its state is changed, which means that listening to those queues allows a listener to be notified of state changes on the device.

On the backend, it wouldn't make sense to create a new listener for every event that uses a certain condition as that would put more load on the messaging queue layer of the backend. Instead, a different mechanism is used. Upon startup of the backend, a listener is created for each event queue of each device. These are simple classes that are responsible for receiving messages published in those queues and notifying the Jarvis backend engine that those messages were received.

The engine then uses the Observer[21] design pattern to allow commands to add observers to these messages. For example, the query "*Turn on the light when the sensor is activated*" would add an observer that would look for messages on the sensor's event queue with the value "on". Because of that, the engine would notify this observer of all messages on that queue leading to the event being executed. The advantage of this technique is that it allows the engine to be a centralized point through which all messages go through and with which all observers must be registered.

[21] `http://w3sdesign.com/`

Other patterns are used across the solution such as Builder (for creating IoT device classes), Singleton (for database and messaging queue networked connections) and Adapter (for abstracting the networked interactions with the messaging queue for the IoT devices).

One of the main features of this project are the causality queries, the requests through which the user asks why something happened.

To implement this, the command classes also have a method that determines whether or not they could have caused a certain condition to be true. Because of that, a simple solution to this problem is to, given the queried condition, look through the log of commands and user commands to understand what could have caused that condition.

However, this approach does not work for all the scenarios mentioned in Section 3.4. For example, Use Case 12 demonstrates a case where, at the moment of the query, multiple rules may have caused the condition to be true. In these cases, it is not enough to return the latest logged command that could cause the queried condition. Instead, there are two possible approaches:

- **Return earliest possible cause:** if multiple rules/events could have caused the queried condition to be true, choose the one that happened first as that was the cause for the condition to be true in the first place.

- **Use an heuristic to calculate most relevant cause:** instead of assuming the best answer is the earliest cause, use an heuristic to evaluate all the possible causes in terms of relevance and choose the most relevant. This calculation can use the nature of the cause (caused by user vs. caused by rule), order by which causes happened, among others.

In the context of this project the implemented solution was to use the earliest possible cause, but it is important to mention that the alternative would be just as valid and possibly more useful to the user.

It is also possible that a chain of interconnected rules caused the queried condition to be true, as is demosntrated by use cases 13 and 14. In this case, it is possible to either reply with the complete chain of events, the latest possible cause (Use Case 13) or engage in a conversation through which the user can explore the full chain of events chronologically (Use Case 14).

In the context of this project the solution uses the approach of returning the latest possible cause. However, with the mindset of creating a helpful and fully capable bot, the best approach would be to engage in a conversation with the user, not only because it would allow the user to obtain more information at his own pace but also because it would be possible to make changes to the rules as he goes. For example, if the user is having that conversation and he decides to change one of the rules, he can just interrupt the conversation to modify the rule. With the current approach, that is only possible if the answer contains a single rule or cause to the condition which in reality is not always the case.

All the techniques mentioned above allow the backend to fulfill all the intents that were defined in Dialogflow and therefore allow for the management of an IoT system. To use these features in a production environment, the entire backend system is compiled into a Web Application Resource file (*.war*) which can then be deployed in a regular server equipped with a webserver such as Glassfish[22].

4.3.4 Database

The database has a simple setup yet is very useful for the Jarvis backend to operate. It is a MongoDB[23] document store database that runs on Mongo's official Docker image[24] for easy setup and deployment.

The main purpose of the database is for the backend to store the list of devices and their capabilities in the same format specified by Mozilla's Web Thing API, mentioned in Section 2.3.4. This is essencial in the context of this project since there is no discovery system that dinamically provides a list of available devices as they are added or removed from the overall system. The objects that represent the IoT devices are stored in a collection named "*things*".

The database is also used to keep a log of user commands (collection "*userCommands*") and device commands (collection "*commands*"). User commands represent instructions given by the user, while device commands are actual commands executed on devices. For example, if a user says "turn on the light from 5pm to 6pm" this generates a user command and two device commands (for 5pm and 6pm). The examples for user commands and device commands are available in Appendix B in listings B.1 and B.2, respectively.

Finally, the database is used to store a log of the events that have been triggered. These are the events relative to system rules defined by user requests. These are stored in collection "*eventHistory*", as can be seen in Listing B.3 from Appendix B

4.3.5 Messaging Queue

The messaging queue acts as a middle-man for the communication between the Jarvis backend and the IoT controllers. This allows for the backend to be completely unaware of how many controllers there are, where they are located, how to reach them or how to talk to them. Instead, it has a single point of communication to know, the messaging queue, and can be abstracted from everything else.

This is especially useful if there are multiple IoT controllers that may be in different locations. Without the messaging queue, if the backend needed to interact with a device it would not only have to know where each of the controllers were but also which devices were connected to each of them. This does not scale well since having a change in the system's topology would require a change in the backend software. Instead, by using the queue the backend communicates only with it, giving the controllers the responsibility of subscribing to the correct channels in the queue.

[22] `https://javaee.github.io/glassfish/`
[23] `https://www.mongodb.com/`
[24] `https://hub.docker.com/_/mongo/`

The queue is implemented with a simple RabbitMQ[25] Docker image[26], which allows for a quick and easy setup in any server equipped with Docker. RabbitMQ is a messaging framework that supports several protocols such as AMQP, STOMP and MQTT. Through it, clients can subscribe or publish to messaging channels defined by a URL-like name (e.g. "/house/kitchen") that can have different properties in regard to how they operate, i.e. if messages are persisted, if they are broadcast to all subscribers or delivered to only a single subscriber, among others.

Once the queue is running, both the Jarvis backend and IoT controllers can connect to it with the configured credentials and IP address.

As mentioned in Section 2.3.4, IoT devices can present URLs for actions (URLs to which sent requests trigger some sort of action on the device), events (URLs from which other devices can read messages that convey the information that certain event happened) or properties (URLs from which other devices can read property changes on the device). Based on thes edifferent URLs, the interactions may be bi-directional by using RabbitMQ:

- **Actions:** if the backend intends to perform an action on an IoT device, it should write the action's value of that device's action URL (e.g. write "on" to "*/house/living room/living room light*"). On the other hand, that device's IoT controller should subscribe to that same URL and act accordingly to the messages it receives there.
- **Events:** if the backend intents to be notified of changes on a device's property (e.g. a sensor's value), it should listen to messages from the device's property URL (e.g. "*/house/living room/temperature sensor/temperature*") to receive the changes in the format of messages. On the other hand, that sensor's IoT controller should be responsible for publishing that event (in this case, property value change) on that same URL.

4.3.6 IoT System Controllers

Since this component is not under the scope of the project, as was mentioned above in Section 3.3, its architecture or technologies are not relevant for the purpose of evaluating this project. However, since it was important in the project's development, it will be described in this sub-section.

The IoT controllers are middleware devices that send and receive messages from the messaging queue and deliver them to the appropriate devices. For the purpose of testing this project, each controller is represented by a RaspberryPi that runs a Python script to write and read messages to/from the messaging queue and interact with the IoT devices accordingly.

There are two things that these controllers do. On one hand, they perform actions on the devices: if a controller is responsible for turning on/off the kitchen light, it will subscribe to the "*/house/kitchen/kitchen light*" queue and turn on or off that light if it receives the messages "on" or "off", respectively. On the other hand, controllers can notify the queue of changes to device status if, for example, a binary sensor or a multivalued sensor changes its value. An example

[25] https://www.rabbitmq.com/
[26] https://docs.docker.com/samples/library/rabbitmq/

of this would be a controller for a kitchen temperature sensor publishing messages to "*/house/kitchen/kitchen temperature sensor*" publishing the message "21" if the sensor's value changed to 21 degrees.

```
import RPi.GPIO as GPIO
import time
import pika

pin_1 = 18
pin_2 = 17
host = 'andrelago.eu'
username = 'rabbitmq'
password = 'rabbitmq'
bedroom_queue = '/house/bedroom_light/actions'
living_room_queue = '/house/living_room_light/actions'

def initGPIO ():
   GPIO.setmode(GPIO.BCM)
   GPIO.setwarnings(False)
   GPIO.setup(pin_1,GPIO.OUT)
   GPIO.setup(pin_2,GPIO.OUT)

def ledOn_1 ():
   GPIO.output(pin_1, GPIO.HIGH)
   print "LED 1 on"

def ledOff_1 ():
   GPIO.output(pin_1, GPIO.LOW)
   print "LED 1 off"

def ledOn_2 ():
   GPIO.output(pin_2, GPIO.HIGH)
   print "LED 2 on"

def ledOff_2 ():
   GPIO.output(pin_2, GPIO.LOW)
   print "LED 2 off"

initGPIO()

```

```python
connection = pika.BlockingConnection(pika.ConnectionParameters(host, credentials=pika.PlainCredentials(username=username, password=password)))
channel = connection.channel()
channel.queue_declare(queue=bedroom_queue)
channel.queue_declare(queue=living_room_queue)

###################################
######### RECEIVE MESSAGE ##########
###################################

def callback_bedroom(ch, method, properties, body):
    print(" [x] Received %r" % body)
    if body == 'on':
        ledOn_1()
    elif body == 'off':
        ledOff_1()
    else:
        print("Unrecognized message")

def callback_living_room(ch, method, properties, body):
    print(" [x] Received %r" % body)
    if body == 'on':
        ledOn_2()
    elif body == 'off':
        ledOff_2()
    else:
        print("Unrecognized message")

channel.basic_consume(callback_bedroom, queue=bedroom_queue, no_ack=True)
channel.basic_consume(callback_living_room, queue=living_room_queue, no_ack=True)
print(' [*] Waiting for messages. To exit press CTRL+C')
channel.start_consuming()
```

Listing 4.1: Python code used by one of the Jarvis IoT controllers

This controller is used to control two lights, living room light and bedroom light. At the beginning of the code, the pins controlling each of them are defined, as well as the variables that allow the connection to the messaging queue. Then, after defining the methods to turn on or off the lights as well as initiating the connection to the queue, two consumer callbacks are defined for *callback_bedroom* and *callback_living_room*. These callbacks are later setup to be called when messages are received from both the living room and bedroom light actions URLs (defined at the beginning of the file). When messages with the value "on" are received, the callbacks turn on the respective LED, and the opposite is done if the message is "off".

Therefore, with this setup, if the Jarvis backend writes on/off messages to the action queues of either the bedroom light or living room light, this controller will perform that action on the actual hardware devices. Note that the backend does not communicate directly with the controller, which means that if it was intended to move one of the lights to a separate controller there would be no need to perform changes on the backend.

4.4 Assembling the Solution

By assembling the entire solution it is possible to obtain a voice or text chatbot that manages a diverse and possibly distributed smart space with natural language commands.

The User Interface is enabled by the integration of Dialogflow, as explained in Section 4.3.2.

The Jarvis backend, database and messaging queue must be able to be connected to each other and the IoT controllers must also be able to reach the messaging queue, which means some sort of networked server is required. In the case of this project, the backend, database and messaging queue are setup in a DigitalOcean[27] server with domain associated with it (in this case, *andrelago.eu*).

Listing 4.2 presents a sample *docker-compose.yml* file that allows the backend, database and messaging queue components of the system to be deployed in a server equipped with Docker[28] with a single bash command.

```yaml
version: '3'

services:
  backend:
    container_name: "jarvis_backend"
    image: oracle/glassfish:5.0
    ports:
     - "3001:8080"
     - "4848:4848"
    volumes:
     - ./jarvis-backend:/jarvis-backend
     - ./jarvis-backend/out/artifacts/jarvis_backend_war:/
       glassfish5/glassfish/domains/domain1/autodeploy
```

[27] https://www.digitalocean.com/
[28] https://www.docker.com/

```
    entrypoint: asadmin start-domain --verbose
    links:
      - db
      - "rabbit:rabbit"
    depends_on:
      - db
      - rabbit

  db:
    image: mongo:3.2
    container_name: "mongodb"
    environment:
      - MONGO_DATA_DIR=/data/db
      - MONGO_LOG_DIR=/dev/null
      - MONGO_INITDB_ROOT_USERNAME=${MONGO_USR}
      - MONGO_INITDB_ROOT_PASSWORD=${MONGO_PWD}
    volumes:
      - db-data:/data/db
    ports:
      - 27017:27017
    command: mongod --smallfiles --logpath=/dev/null --auth

  rabbit:
    image: "rabbitmq:3-management"
    hostname: "rabbit1"
    environment:
      RABBITMQ_DEFAULT_USER: ${RABBIT_USR}
      RABBITMQ_DEFAULT_PASS: ${RABBIT_PWD}
    ports:
      - "15672:15672"
      - "5672:5672"
    volumes:
      - "./rabbitmq/enabled_plugins:/etc/rabbitmq/enabled_plugins"

volumes:
  db-data:
```

Listing 4.2: Example docker-compose.yml file which deploys several Jarvis components

Docker Compose[29] is a tool for configuring and running multiple Docker applications that may be networked or depend on each other. Docker containers are virtual machines that may be configured to run different operating systems and have installed different applications or technologies. The syntax of Docker Compose files is thoroughly explained in its official documentation[30], but below is a brief explanation of the values and content shown in Listing 4.2.

The file declares the services "backend", "db" and "rabbit" which respectively represent the Jarvis backend, the Mongo database and the RabbitMQ. The backend uses the Docker image for the Glassfish[31] web server. The file also exposes the required ports for the server to be exposed to the web and declares it to be networked with the remaining services. Finally, the file declares that upon launch the Jarvis ".war" file must be copied into the container and executed at the start. This service ensures that the Jarvis JavaEE server is properly launched.

The following service, "db", configures the Mongo database. The file specifies the database credentials and exposes the required ports.

Finally, the file declares the "rabbit" service which configures the messaging queue with the appropriate credentials and ports.

Because the server is configured with a public domain, all the services mentioned above can be accessed through that same domain.

Because of this Docker Compose file, all that is required to launch the backend, database and messaging queue is to download the project code onto the desired server and run "*docker-compose up*" at the root of the project.

Once this is done, it is also required to add the server's URL on the Dialogflow settings panel so that Dialogflow is able to reach the fulfillment backend.

To complete the setup of the project it is necessary to setup the IoT controllers. Because they were developed as Python scripts, all that is necessary is to run the scripts on the background of a device such as a RaspberryPi that is connected to the internet.

Figure 4.10 illustrates what the complete flow of a query is in the assembled system.

Once the query is entered or spoken into the User Interface, it is sent to the Dialogflow backend for Natural Language Processing. There, the query is associated with an intent and its parameters (device name, intended action, ...) are parsed into a JSON object that contains all the information in the query in a language understandable by an application. That JSON is sent to a fulfillment backend, the Jarvis backend, where the parameters of the query are fully validated. If the parameters are valid, the user command is logged into the database and a device command is generated. That device command is also logged and then executed, to actually perform the desired action on the physical device(s). Simultaneously, the result is returned to Dialogflow in the form of a JSON that contains the desired response to be provided to the UI. Dialogflow then sends this reply to the UI where it is spoken or shown to the user.

[29] https://docs.docker.com/compose/
[30] https://docs.docker.com/compose/compose-file/
[31] https://hub.docker.com/r/oracle/glassfish/

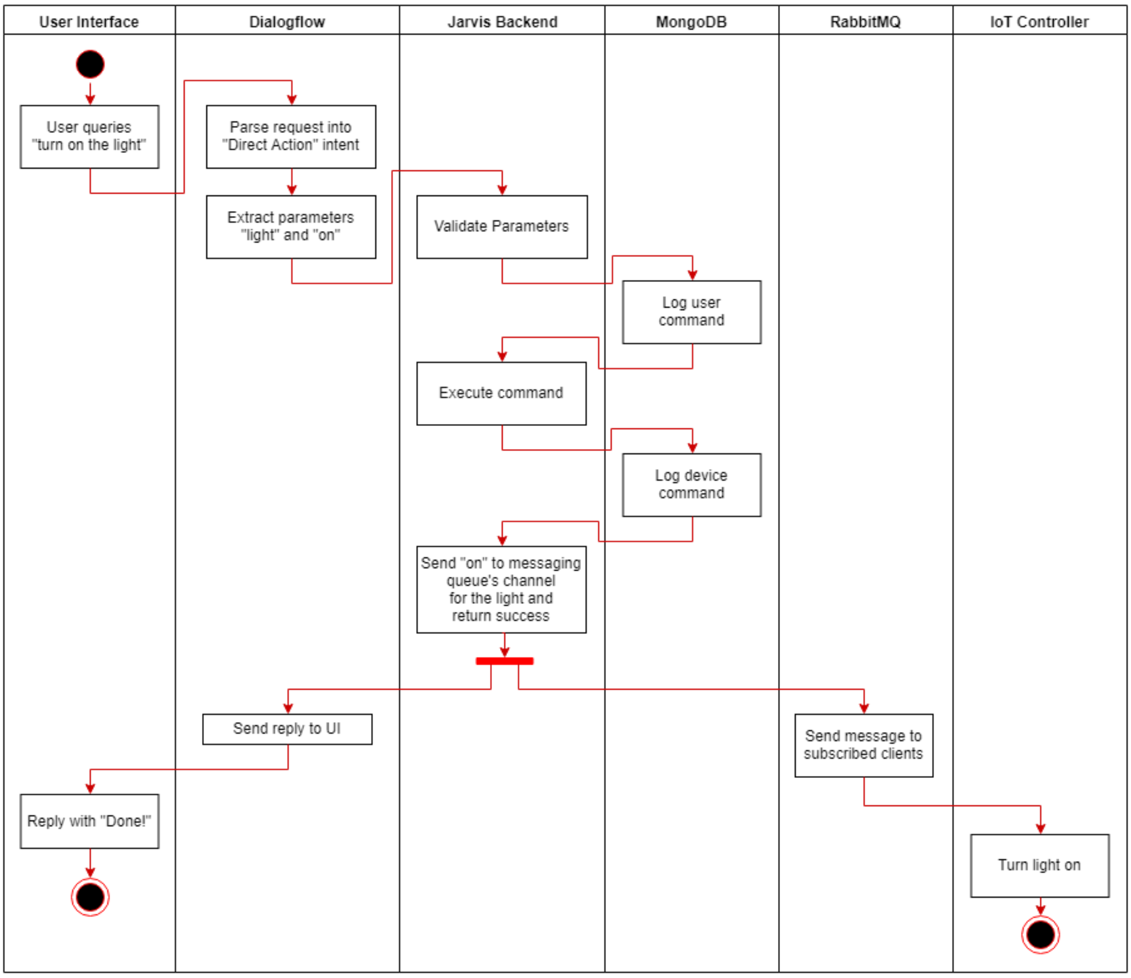

Figure 4.10: Sequence diagram for a sample query in the assembled system

4.5 Conclusions

Jarvis takes advantages of technologies and techniques that already exist to provide users with a complete IoT management experience. From a software standpoint, many of Jarvis' components could be easily scalable due to the use of technologies such as RabbitMQ or MongoDB that can be easily distributed as was detailed in the previous sections.

It becomes important to understand exactly how well the system works and how it delivers its proposed features. Chapter 5 describes the methods used to evaluate the obtained results and presents the solutions drawn from the testing of the system.

Chapter 5

Validation

This section demonstrates how the solution implemented for this project accomplishes the features proposed in Chapter 3.

At first a critical overview of how many of the proposed features were delivered will be made, with a reflexion on how well they work and how advantageous they might be when compared to the tools that are already available on the market. This is an objective measure over the technical progress made in terms of creating a more capable voice assistant in the domain of IoT management.

Then, the description of the user study made will be presented. The goal of the study is to understand whether Jarvis makes it more easy and comfortable to manage an IoT system and whether it could replace the tools and techniques currently being used.

5.1 Simulated Scenarios

The simulated scenarios are an execution of the features proposed in the use cases described in Section 3.4. Their purpose is to objectively evaluate how the features were accomplished using a test scenario that simulates a real IoT environment.

To simulate this scenario, all that is required is to have an assembled solution as mentioned in Section 4.4. In this case, the IoT controller is a single RaspberryPi device that is connected to two LED's that represent the living room and bedroom lights. The RaspberryPi is also connected to a binary sensor (in this case, a button) which represents a living room motion sensor. Note that although this was chosen as the intended use for the button, it could also represent other kinds of binary sensors such as pressure sensors.

Note that other kinds of devices such as temperature sensors or thermostats could have been used for a more diverse representation of an IoT system, but they were kept out of the this evaluation method because their use is similar to the devices chosen and the specified assembled solution is enough to stress the features being tested.

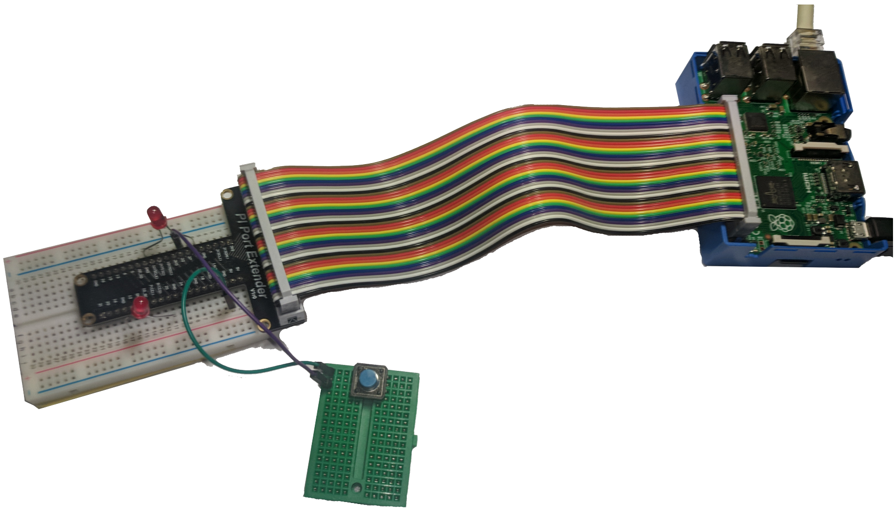

Figure 5.1: Assembled RaspberryPi with 2 LEDs and 1 button

Figure 5.1 displays the assembled RaspberryPi with the connected LED's and button. The RaspberryPi is also connected to the internet via an ethernet cable. This is required so that the Python controller can communicate with the Rabbit messaging queue. The Python controller is a simple script very similar to the one previously described in Listing 4.1.

The chosen User Interface for this test was Slack, but any other interface such as the Google Assistant would perform the same way due to Dialogflow's built-in integrations.

Below, all the performed test scenarios are described, along with the use cases from Section 3.4 they exercise. For each of the scenarios, the performed steps are presented along with the expected and obtained results, as well as a brief comparison to some of the currently available tools and products. In this case, Node-RED was chosen to represent Visual Programming Platforms and the Google Assistant was chosen to represent voice assistants. At the end, a comparison table will be presented to provide a more compact comparison of Jarvis, Node-RED and the Google Assistant in terms of provided features.

5.1.1 Scenario 1 - one-time action

- **Conversation:**
  - **User**: *"Turn on the living room light."*
  - **Jarvis**: *"Done!"*
- **Expected behavior:** Living room LED turns on.
- **Comparison to other tools:**
  - **Google Assistant:** performs the same task with very similar queries, but is limited to vendor devices that are support by the Assistant[1].

[1] https://assistant.google.com/intl/en_us/resources/#partners

  - **Node-RED:** can perform the same task with UI buttons that the user can press to trigger certain actions.
- **Fulfilled use cases:** Use Case 1.

5.1.2 Scenario 2 - one-time action with uncertainty of device

- **Conversation:**
  - **User**: *"Turn on the light."*
  - **Jarvis**: *"Do you mean the living room light or the bedroom light?"*
  - **User**: *"The first."*
  - **Jarvis**: *"Okay, done!"*
- **Expected behavior:** Living room LED turns on.
- **Comparison to other tools:**
  - **Google Assistant:** feature not supported.
  - **Node-RED:** the rules in Node-RED are device-specific and therefore this type of feature is not supported.
- **Fulfilled use cases:** Use Case 2.
- **Notes:** although this feature has been exemplified only for Use Case 2 it is also possible in all other use cases where there is only one specified device.

5.1.3 Scenario 3 - delayed one-time action

- **Conversation:**
  - **User**: *"Turn on the living room light in 5 minutes."*
  - **Jarvis**: *"Sure."*
- **Expected behavior:** Living room LED turns on 5 minutes after the initial query.
- **Comparison to other tools:**
  - **Google Assistant:** feature not supported.
  - **Node-RED:** can perform the same task with a UI button that is connected to a timer block that generates the intended delay.
- **Fulfilled use cases:** Use Case 3.

5.1.4 Scenario 4 - delayed period action

- **Conversation:**
  - **User**: *"Turn on the living room light from 4pm to 5pm."*
  - **Jarvis**: *"Sure."*

- **Expected behavior:** Living room LED is turned on at the first instance of 4pm after the initial query, and is turned of at the following 5pm.
- **Comparison to other tools:**
  - **Google Assistant:** feature not supported.
  - **Node-RED:** can perform the same task with a UI button that is connected to timers that change the light status at the desired times.
- **Fulfilled use cases:** Use Case 4.

5.1.5 Scenario 5 - daily repeating action

- **Conversation:**
  - **User**: *"Turn on the bedroom light everyday at 9am."*
  - **Jarvis**: *"Okay, will do."*
- **Expected behavior:** Bedroom LED turns on at 9am everyday.
- **Comparison to other tools:**
  - **Google Assistant:** feature not supported.
  - **Node-RED:** can perform the same task with an automated timer that triggers the action to turn on the light at the desired time every day.
- **Fulfilled use cases:** Use Case 5.

5.1.6 Scenario 6 - daily repeating period action

- **Conversation:**
  - **User**: *"Turn on the bedroom light everyday from 8am to 10am."*
  - **Jarvis**: *"Will do."*
- **Expected behavior:** Bedroom LED turns on at 8am and turns off at 10am every day.
- **Comparison to other tools:**
  - **Google Assistant:** feature not supported.
  - **Node-RED:** can perform the same task with automated timers that trigger the actions to turn on or off the light at the desired times every day.
- **Fulfilled use cases:** Use Case 6.

5.1.7 Scenario 7 - cancel last command

- **Conversation:**
  - **User**: *"Cancel my last command."*

  - **Jarvis**: *"Are you sure you want to cancel the command 'turn on the bedroom light at 9pm'?"*
  - **User**: *"Yes."*
  - **Jarvis**: *"Okay, cancelled."*
- **Expected behavior:** Bedroom LED does not turn on at 9pm.
- **Comparison to other tools:**
  - **Google Assistant:** feature not supported for IoT management commands, although it is supported for other sorts of actions such as restaurant bookings.
  - **Node-RED:** with Node-RED the user defines a model of rules that is then deployed into a host device. Because of that, cancelling is not possible not only because the UI does not provide an 'undo' feature but also because it is not possible to re-deploy a previous instance of the rule model.
- **Fulfilled use cases:** Use Case 7.
- **Notes:** this scenario is valid under the assumption that the command "*Turn on the bedroom light at 9pm*" was sucessfully executed beforehand.

5.1.8 Scenario 8 - event rule

- **Conversation:**
  - **User**: *"Turn on the living room light when the living room motion sensor is activated."*
  - **Jarvis**: *"Will do."*
- **Expected behavior:** Living room LED turns on when button is pressed, simulating sensor activation.
- **Comparison to other tools:**
  - **Google Assistant:** feature not supported.
  - **Node-RED:** can perform the same task by connecting the block representing the sensor to the one representing the light, configuring the messages to be sent appropriately. Note that to change the rules it would be necessary to manually change the model in Node-RED's UI and then deploy the new model.
- **Fulfilled use cases:** Use Case 8.

5.1.9 Scenario 9 - rules defined for device

- **Conversation:**
  - **User**: *"What rules are defined for the bedroom light?"*
  - **Jarvis**: *"You told me to turn on the bedroom light everyday at 8am."*
  - **User**: *"Change that to 8:30am."*

  - **Jarvis**: *"Okay, I've changed that rule."*
- **Expected behavior:** Bedroom LED turns on everyday at 8:30am.
- **Comparison to other tools:**
  - **Google Assistant:** feature not supported.
  - **Node-RED:** it is not possible to query Node-RED for all the rules defined for a device. However, it is possible to look through all the rules and check which ones are related to a device and then change them.
- **Fulfilled use cases:** Use Case 9, Use Case 10.
- **Notes:** this scenario is valid under the assumption that the command "*Turn on the bedroom light everyday at 8am*" was sucessfully executed beforehand.

5.1.10 Scenario 10 - causality query

- **Conversation:**
  - **User**: *"Why did the bedroom light turn on?"*
  - **Jarvis**: *"You told me to turn on the bedroom light when the living room light is turned on."*
- **Expected behavior:** the answer should be correct under the assumption that a rule to turn on the bedroom light when the living room light turns on exists and that light was turned on.
- **Comparison to other tools:**
  - **Google Assistant:** feature not supported.
  - **Node-RED:** it is not possible to make such a query in Node-RED. However, technically it is possible to obtain the same information by looking at all the rules defined in Node-RED's model and understanding how they could have caused the condition to be true.
- **Fulfilled use cases:** Use Case 11, Use Case 12, Use Case 13, Use Case 14, Use Case 15.
- **Notes:** this behavior would be expected even if a set of chained rules caused the light to turn on (e.g. there was also a rule to turn on the living room light if the living room motion sensor was activated).

5.1.11 Test Results

Table 5.1 displays a brief comparison on how Jarvis, the Google Assistant and Node-RED fulfill the different test scenarios defined above. Green checkmarks denote scenarios that the tool in that column can fulfill, while red crosses represent the opposite.

As shown in the picture, Jarvis clearly outperforms the Google Assitant when it comes to the management features represented through the test scenarios. This means that Jarvis is capable of a more complete set of IoT management tasks which in turn provide users with more flexibility to

Scenario	Jarvis	Google Assistant	Node-RED
1	✓	✓	✓
2	✓	✗	✗
3	✓	✗	✓
4	✓	✗	✓
5	✓	✗	✓
6	✓	✗	✓
7	✓	✗	✗
8	✓	✗	✗
9	✓	✗	✗
10	✓	✗	✗

Table 5.1: Simulated scenarios fulfillment in Jarvis, Google Assistant and Node-RED

manage the system. The Assistant is limited to direct actions on devices which lacks far behind the diverse range of features provided by Jarvis.

It is important to note that the Google Assistant does not specialize in IoT management, instead being a more generic chatbot that can aid users with a lot of different tasks, many of which have nothing to do with IoT.

The difference is not so big with Node-RED. Although many cells have red crosses, the notes on the scenarios above demonstrate that the same functionality can be achieved with workarounds that still allow users to obtain the information they desire or perform the tasks they want.

However, as will be seen in Section 5.2, the main advantage of Jarvis in all of the scenarios is the increased comfort it provides. Regardless of the feature, Node-RED requires users to use a computer in order to interact with the system, while Jarvis can be used by talking to a phone while sitting on a couch without even touching the phone[2].

In sum, the test scenarios demonstrate that, for the sort of features they simulate, Jarvis is able to provide a wider range of possibilities for users to manage their IoT systems. Although this might not necessarily be enough to make Jarvis better, it proves that Jarvis exhibits progress made in terms of using natural language interfaces to better manage an IoT system.

[2]This is possible with, for example, the Google Assistant, since interactions are automatic based on trigger phrases such as "Ok Google" (https://support.google.com/assistant/answer/7394306).

5.2 User Study

Although the simulated scenarios described in Section 5.1 evaluate Jarvis regarding its feature completion and diversity, it does not evaluate whether it is or isn't easier and better to use when compared to other tools such as the Google Assistant or Node-RED.

To do such evaluation, this user study was performed to measure how fast users can get tasks done using Jarvis and whether they prefer it over a non-conversational interface. It is important to note that this study does not perform a direct comparison to the Google Assistant since it is the same behavior-wise but less complete in terms of IoT management features.

5.2.1 Study Description

The study was done with 17 participants with ages ranging from 18 to 51. The main goal on obtaining the test participants was to obtain a significant age range but also to include mostly people without a background in technology to understand whether the language capabilities of Jarvis are enough for the users to interact with it. Because of that, 14 of the participants did not have a background in software development although they were familiar with different technologies such as smartphones and the internet.

Each participant was given a set of 5 tasks (1 control task and 4 study tasks) that they were supposed to get done with the help of Jarvis, using the Google Assistant as the user interface. The only instructions given were that they must talk to the phone in the way that feels the most natural to them in order to complete the tasks at hand.

Besides the set of tasks, participants were given the list of IoT devices available in the simulated smart house they would be attempting to manage through Jarvis. To increase the diversity and reduce the bias of the study, two different sets of devices and tasks were created, so that different smart house topologies were tested. The participants were assigned one of the test sets randomly.

For each of the tasks, the study administrator would take note of whether each participant was able to complete each of the tasks, the time it took to complete the task and the count of unsuccessful queries. This count was made separately for queries that were not understood by the Assistant's speech recognition capabilities, for queries the user mispronounced and for queries that were correct but Jarvis was unable to process. After completing the tasks, the administrator would explain to participants that an alternative to Jarvis to perform the tasks at hand was a non-conversational visual interface such as Node-RED, giving an example of how the same tasks would be performed using that tool. After that, the administrator would ask participants what they believe are the advantages of Jarvis over such a tool, if they find any, and whether they would prefer Jarvis over any non-conversational tool. Finally, the participants would be asked what they think could be improved about Jarvis and the way it handles smart space management.

It is important to note that participants could have been required to use Node-RED to perform the same tasks in order to compare the execution times of both. However, this was not done because Node-RED would require slightly more training and instructions to participants as it is not as intuitive as a conversational interface that users can simply talk to without being given any sort of previous instructions.

As an example, this is one of the task sets given to participants (the other set was very similar):

- **Available Devices:** living room light, bedroom light and living room motion sensor.
- **Task 0:** turn on the living room light.
  - **Note:** this is a control task used to get the participant familiarized with the interface.
- **Task 1:** make the living room light turn on in 5 minutes.
- **Task 2:** make the living room light turn on when the living room motion sensor is turned on.
- **Task 3:** check the current rules defined for the bedroom light and make it turn on everyday at 10pm.
- **Task 4:** the bedroom light turned on but you don't know why. Ask Jarvis why it happened and decide whether the answer was explanatory.

5.2.2 Results

Table 5.2 compiles the results observed during the study. Each row represents one of the tasks given to participants. The numbers displayed by the table are analyzed further in Section 5.2.3 and the meaning of each column is as follows:

- **"Task:"** number of the task (0-4) and the task set number in parenthesis (1/2).
- **"Done:"** percentage of participants that were able to complete the task successfully.
- **"Time:"** average ("*Avg*") and median ("*Med*") time, in seconds, that participants took to complete the task.
- **"IQ (Ast):"** average ("*Avg*") and median ("*Med*") times that queries were incorrect due to the Google Assistant not properly recognizing the user's speech (due to microphone malfunction, background noise, ...).
- **"IQ (User):"** average ("*Avg*") and median ("*Med*") times that queries were incorrect due to the user not speaking a valid query (e.g. saying "*Turn up the lighting*").
- **"IQ (Jvs):"** average ("*Avg*") and median ("*Med*") times that queries were incorrect due to Jarvis not recognizing a valid query.
- **"IQ:"** average ("*Avg*") and median ("*Med*") times queries were incorrect (sum of "*IQ (Ast)*", "*IQ (User)*" and "*IQ (Jvs)*").

In terms of the questions asked to participants regarding the performance of Jarvis and its comparison to visual programming tools, all participants mentioned that in a real life scenario they would prefer a conversational assistant such as Jarvis due to its greater "ease of use" and "commodity". Most pariticipants considered that the main disadvantage of visual tools to be the need of a physical device such as a computer as well as a "knowledge of how the hardware works". Some participants also mentioned that although Jarvis allows for complex management features

		Time		IQ (Ast)		IQ (User)		IQ (Jvs)		IQ	
Task	Done	Avg	Med	Avg	Med	Avg	Med	Avg	Med	Avg	Med
0 (1)	94%	6.4	6	0.13	0	0.25	0	0.13	0	0.5	0
1 (1)	94%	7.1	7	0.38	0	0.5	0.5	0	0	0.5	0.5
2 (1)	88%	10	10	0.75	0.5	0.63	0.5	0.25	0	1	1
3 (1)	100%	20	19.5	0.13	0	0.13	0	0.75	1	1	1
4 (1)	94%	9	8	0.25	0	0.38	0	0	0	0.63	0
0 (2)	100%	6.4	6	0.33	0	0	0	0.33	0	0.67	0
1 (2)	94%	7.6	7	0.11	0	0	0	0.44	0	0.56	0
2 (2)	100%	9.9	10	0	0	0.11	0	0.78	1	0.89	1
3 (2)	88%	19.44	19	0.33	0	0.33	0	0.22	0	0.89	1
4 (2)	100%	8.33	8	0.33	0	0.22	0	0.22	0	0.78	1

Table 5.2: User Study results (task completion rate, task time and incorrect queries).

not supported by visual tools, these also have certain tools that Jarvis doesn't such as integrations with Twitter[3] or IFTTT[4]. Finally, participants also remarked that although Jarvis has an increased "accessibility" due to its conversational interface, there is a "margin of error that comes with voice recognition" that can cause a worse user experience.

5.2.3 Analysis

In terms of the spoken queries as well as their response, the complexity increases from task 0 to task 3 since the queries require more words or interactions. This is reflected by the task time that also increases from task 0 to task 3 in both the average and median values and in both task sets. Other than that, it is hard to use the task times as means of comparison to visual tools since no comparison study was done. However, since using a visual tool such as Node-RED involves moving to a computer, opening the platform, dragging blocks and then configuring their options, we consider that in any case the overall task would take longer in a visual tool than it does with Jarvis.

The numbers related to incorrect queries also bring some interesting conclusions about the system's performance. The table shows that there were some incorrect queries for the Assistant which means that the Assistant's speech recognition failed to correctly translate what the participants said. However, this was mostly due to the participant's pronounciation since none of the speakers had English as their main language. This does not have implications to the evaluation of Jarvis, but it does indicate that this sort of systems might be harder to use if they do not support the user's native language.

It can also be noted that there were a few instances of incorrect queries that were the user's fault ("*IQ (User)*"). These were cases where user queries were gramatically incorrect and therefore did not match the sample queries defined in Dialogflow. For example, a query like "*Turn on lights*" is not a considered the user's fault since the correct one would be "*Turn on the lights*", however,

[3] https://twitter.com/
[4] https://ifttt.com/

it still carries enough information to understand what the user's intent is. More reflexion on these cases will be done below in Section 5.2.4.

There is a significant number of incorrect queries due to Jarvis ("*IQ (Jvs)*"), meaning that the user's query was valid but its meaning was not understood by Jarvis. This can be caused by either a mispronounciation of a device's name (e.g. saying "*Turn on the living room light**s***" instead of "*Turn on the living room light*") or a sentence structure that is valid but is not recognized due to the sample queries inserted into Dialogflow. This possibly represents the most serious downside of this project, but its severity and possible solutions are discussed below in Section 5.2.4.

It is also important to note that despite the numbers for incorrect queries, the success rate of all tasks is very high, which indicates that the system is intuitive enough to be used without previous instruction or formation. This proves that not only the features work on the smart space management side, but also that the conversational interface is well designed and works as expected.

All of the reflexions made above are reflected by the participants' answers to the questions made, since they reiterate that Jarvis is easy to use and provides "accessibility" and "commodity". This is very important since it is the deciding factor for participants to prefer Jarvis over a non-conversational interface, which they still consider to have some advantages over Jarvis.

5.2.4 Validation Threats

The study results present some threats to the validation of this project along with its goals, described in Chapter 3.

One one hand, there are the incorrect queries that are due to the user either having a bad pronounciation or providing incorrect sentences. For example, a query like "*Turn up lights*" is not a valid query since that would be "*Turn on the lights*", however, it still carries enough information to understand what the user's intent is. As it stands, Jarvis does not support this sort of queries. However, including gramatically incorrect queries in Dialogflow's system would make Jarvis more accept these partially incorrect queries, making it more flexible to these scenarios and therefore improve the user experience.

On the other hand, there are queries that are valid but fail because Jarvis fails to recognize them. This is concerning because if the user feels that a query didn't work because his pronounciation or grammar was bad he repeats the query and does not blame the system. However, if the user feels he said what was supposed to be said but the system did not understand it properly, he loses motivation to use the system and feels like it doesn't work, which is bad and goes against the goals of Jarvis. The positive remark is that in most cases the flaw was due to the user mispronouncing the device name, e.g. saying "*Turn on the living room light**s***" instead of "*Turn on the living room light*". This could easily be fixed by adapting the way Jarvis detects devices. Currently, it looks for devices that match the exact same name that was provided in the user query. If, instead, Jarvis looks for similar names that are not an exact match, the number of incorrect queries due to Jarvis can be significantly reduced, therefore minimizing this problem and improving the system's functioning.

There were also the cases that failed due to Jarvis due to a sentence structure that was not included in the sample queries configured in Dialogflow. This happened because the sample queries configured in Dialogflow were not enough to cover all the different ways that the same command can be given, resulting in an increased error rate for queries. For example, in the intent "Rules Defined" defined in Section 4.3.2, a query like "*What are the rules for the bedroom light?*" would be considered invalid since none of the sample queries is similar to it, despite it being a completely valid query to ask for the rules for a device. To fix this problem it is important to detect other ways of providing the same commands already represented by the intents that are already defined, for example by looking at the failed queries from the user study, to increase the range of commands that Jarvis can parse. Doing this reduces the error rate of queries, which improves the system's flexibility even further.

5.3 Conclusions

The simulated scenarios demonstrate that Jarvis is a rather complete system in terms of smart space management features, clearly outperforming current conversational assistants. The wide variety of management features that were made possible with a natural language interface make the system very capable and able to perform complex tasks with nothing but voice commands.

The user study helped to clarify some of the problems of Jarvis and its embedded natural language interfacing capabilities, while simultaneously reinforcing the idea that a conversational interface can be better for smart space management due to its increased comfort as long as it still presents comlex management features.

The study showed that the participants prefer a conversational interface over a visual interface due to its greater ease of use. Simultaneously, the experiences of participants in the study showed that the main problem with Jarvis might be its reduced flexibility to partially incorrect queries. While this is a clear problem for the system's user experience, it is a problem that can easily be fixed in the future work of Jarvis and that can significantly improve its ease of use.

Overall, we are satisfied with the evaluation of this project since it has proved that it is capable of the management features it aimed to support and the natural language interface problems brought up by the user study have simple solutions that shall be addressed in the future work described further in Section 6.4.

Chapter 6

Conclusions and Future Work

This chapter is a reflexion on the entire work done for this project, gathering what was learned with it. At first, we will present the main difficulties felt with the project, then moving to the main contributions and conclusions collected from this dissertation. At the end of the chapter, we look at possible future work on this project that could increase its scientific contributions and make it a better system.

6.1 Main Difficulties

One of the main challenges of developing an IoT management tool is that there are already many different product ecosystems in place, mainly the Apple, Amazon and Google ecosystems. The problem with this is that there are no common standards for communication with devices or other common functionalities which in turn makes it hard to create a "one fits all" solution, i.e. a solution that simultaneously works with all ecosystems. An example of this is the Dialogflow platform, which doesn't work with the Apple ecosystem although it is a very popular integration platform.

Another adversity with IoT and smart spaces when it comes to developing management tools is the huge diversity of devices available. As was described in Section 2.2.2, there are many different IoT devices that range from remote light switches to smart refrigerators, and this brings extra complexity for management tools, especially for the conversational ones. In the case of Jarvis, this diversity means not only considering the many different device types when specifying the user queries, but also supporting the interactions with this devices. This problem is made even worse due to the fact that there is no common communication protocol in place for IoT devices. Despite of Mozilla's attempt at a Web Thing API[1] described in Section 2.3.4, the lack of a common interface for IoT devices means that most hardware interactions are "hard coded" for each type of device, giving these systems less flexibility.

[1] https://iot.mozilla.org/wot/

In the case of a conversational assistant such as Jarvis, many of the problems faced were related to the very nature of a natural language interface. When designing the system and defining sample user requests, it is often hard to imagine the different ways the same command can be given by different people, which was detected during the user study described in Section 5.2. The user study also helps minimize this problem since, once new queries are discovered it is very easy to add them to Dialogflow's system, therefore improving the experience for future users.

Another slight difficulty of using third-party platforms such as Dialogflow for NLP tasks is that it becomes hard to control the latency of the system as a whole. Looking at it from a user's point of view it is clear that having a high latency can cause problems to the user experience because it makes the conversation less fluid or natural. With a completely specialized tool it is easier to make performance optimizations in tune with the platform's goal, but with a generic external tool that is just not possible.

6.2 Main Contributions

As was mentioned in Chapter 3, the main contributions of Jarvis are focused on new IoT management features for conversational assistants. Because of that, the main contributions of this dissertation are the following:

- **Delayed, period and repeating actions:** the ability for users to perform queries such as "*Turn on the light in 5 minutes*" or "*Turn on the light everyday at 8am*".
- **Use contextual awareness for more natural conversations:** the ability to have conversations that last for multiple sentences to provide a more intuitive converstation. This is started by queries such as "*What rules do I have defined for the living room light?*".
- **Event management:** this is used to create interactions and dependencies between devices that might not necessarily know that each other exists. This is used for queries such as "*Turn on the light when the motion sensor is activated*".
- **Causality queries:** the ability to understand how the system operates simply by asking it, with queries such as "*Why did the light turn on?*".

A paper summarizing this project's contributions can be found in [LF18].

6.3 Conclusions

The main conclusion of this project is that it is possible to use a conversational interface to provide a useful management tool for smart spaces. As was mentioned in Section 2.3.3, current conversational bots already have the capacity to interact with IoT devices, so an integration of the features of Jarvis would make them more complete and useful to users.

At the same time, through stages of the development such as the user study mentioned in Section 5.2 it becomes clear that natural language is a tricky part of the process. The user study showed how hard it is to predict the many user queries that have the same intrinsic meaning, and this problem would only be increased if multiple languages were targeted. This wouldn't have implications on the complexity of tasks that the bot does, but considering multiple languages would bring additional problems not only for defining user queries but also for establishing the representation of entities such as devices.

Another conclusion drawn from the use of a conversational interface is that there are additional problems when the responses to the user are two long. Comments on the user study state that when the responses provided by the system are too long they may get harder to understand since there is more information to be conveyed. A possible solution for this problem would be to use a hybrid interface that provides both visual and audio interactions, but there could be other approaches such as an interactive dialogue that shortens sentences.

We believe that the contribution of Jarvis is clear, as it outperforms current conversational assistants in terms of supported features while simultaneously being easier to use for users (as seen by the user study). Additionally, throughout the document the many research questions defined in Section 3.5 were answered. Even if some of them could be answered differently, the project presents a functioning solution for all of them that could be used in real user products, therefore making it easier for more people to start using IoT in their homes.

6.4 Future Work

As mentioned in Section 3.4.16, alias commands were seen as a stretch goal for this project due to the time constraints of the dissertation. However, Section 4.3.2 showcases some of the work done to implement alias commands such as the query definitions in Dialogflow. Although some work was done towards creating alias commands, it still needs refinement in order to provide a complete feature and therefore we believe this would be useful for the future of this project.

Section 3.4.15 mentions causality queries for undocumented relations of events. This was not possible in the span of the project mostly because it involves serious considerations on how the induction of relationships between events can be done. One possible approach to this would be considering events that happened close to each other to possibly be related, but this could not be a sufficient criteria in many scenarios. Therefore, we believe that a further study of event relations that are not explicit to be an important future step for Jarvis and for conversational smart space management tools.

As was mentioned in this project's assumptions listed in Section 3.3, the addition or removal of devices was not considered for the development of this project. However, it would obviously be important for a user product in order to allow users to dinamically add or remove devices to their smart spaces. This would be very useful for the evolution of Jarvis and another big step towards making conversational systems able to fully manage smart spaces.

As was discussed in Section 5.2, the user study that was made revealed that the system's overall flexibility could be improved by increasing the sample requests in Dialogflow since many of the failed queries in the study carried enough information for them to be understood. Therefore, improving the selection of sample requests to include partially invalid queries or rephrasing the current ones can improve the system's perfomance by reducing its error rate and therefore improving the user experience.

A useful future feature for Jarvis would be a "*What can you do?*" query that prompts the system to explain to a user what it can do. This can be used to create an interactive dialogue that can be used as a tutorial for a beginner user which in turn makes the system even easier to use. We believe this would be a very important feature for users that are less acquainted with technology, since they learn the types of queries and supported features more intuitively.

Finally, we believe a major feature that could be made available in conversational assistants would be complex boolean logic. For example, when defining event rules, it would be useful to use multiple conditions with the "and" or "or" boolean operators. An example of this feature would be the query "*Turn on the bedroom light if the motion sensor is activated and it's after 9pm*", where both conditions would have to be true in order to the action to be executed. This would provide conversational assistants with increased complexity and usefulness, which in turn would make them even more powerful tools for users.

Overall, considering Jarvis' goal of making IoT easier to use for users that are not very comfortable with technology, many of the future work tasks mentioned in this section aim to make interactions even more intuitive and easier to use, without requiring a technical knowledge of how the devices operate.

References

[abi12] 1.5 Million Home Automation Systems Installed in the US This Year. ABI Research, available at https://www.abiresearch.com/press/15-million-home-automation-systems-installed-in-th/, 2012.

[All17] Darren Allan. The best voice recognition software of 2017. *TechRadar*, 2017.

[Bir17] Joe Birch. Exploring Dialogflow: Understanding Agent Interaction. Medium, retrieved from https://medium.com/@hitherejoe/exploring-dialogflow-understanding-agent-interaction-8f3323e3b738, October 2017.

[Bro17] Peter Brown. 20 Billion Connected Internet of Things Devices in 2017, IHS Markit Says. Electronics360, retrieved from http://electronics360.globalspec.com/article/8032/20-billion-connected-internet-of-things-devices-in-2017-ihs-markit-says, 2017.

[bur17] Average hours per day spent in selected activities by sex and day. Bureau of Labor Statistics, retrieved from https://www.bls.gov/charts/american-time-use/activity-by-sex.htm, 2017.

[Cho03] Gobinda G. Chowdhury. Natural language processing. *Annual Review of Information Science and Technology*, 37(1), jan 2003.

[CXL+14] Shanzhi Chen, Hui Xu, Dake Liu, Bo Hu, and Hucheng Wang. A vision of IoT: Applications, challenges, and opportunities with China Perspective, Aug 2014.

[Dav17] Olga Davydova. 25 Chatbot Platforms: A Comparative Table. Medium, retrieved from https://chatbotsjournal.com/25-chatbot-platforms-a-comparative-table-aeefc932eaff, May 2017.

[Dee98] Stephen E Deering. Internet protocol, version 6 (ipv6) specification. 1998.

[Den13] Andrew K Dennis. *Raspberry Pi home automation with Arduino*. Packt Publishing Ltd, 2013.

[DKJG17] Ali Dorri, Salil S. Kanhere, Raja Jurdak, and Praveen Gauravaram. Blockchain for IoT security and privacy: The case study of a smart home. In *2017 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 618–623. IEEE, Mar 2017.

[Dr.] Dr. Brian M. Pierce. Autonomous Real-time Ground Ubiquitous Surveillance - Infrared (ARGUS-IR). DARPA, retrieved from https://www.darpa.mil/program/autonomous-real-time-ground-ubiquitous-surveillance-infrared.

[FS16] A. Florea and V. Sgârciu. Smart refrigerator : A next generation refrigerator connected to the IoT. In *Proceedings of the 8th International Conference on Electronics, Computers and Artificial Intelligence, ECAI 2016*, pages 1–6. IEEE, Jun 2016.

[Gri97] M.L. Griss. Software reuse architecture, process, and organization for business success. In *Proceedings of the Eighth Israeli Conference on Computer Systems and Software Engineering*. IEEE Comput. Soc, 1997.

[Gua10] Massimo Guarnieri. The roots of automation before mechatronics. In *IEEE Industrial Electronics Magazine*, volume 4, pages 42–43, Jun 2010.

[Her84] Andy Hertzfeld. It Sure Is Great To Get Out Of That Bag! *Folklore*, 1984.

[HRS+17] Nicolas Hunke, Michael Rüßmann, Florian Schmieg, Akash Bhatia, and Ninpun Kalra. Winning in IoT: It's All About the Business Processes. Boston Consulting Group, retrieved from https://www.bcgperspectives.com/content/articles/hardware-software-energy-environment-winning-in-iot-all-about-winning-processes/, 2017.

[ISA10] ISA. What is Automation? - ISA. Retrieved from https://www.isa.org/about-isa/what-is-automation/, 2010.

[Jer34] Harry Jerome. Mechanization in Industry. *National Bureau of Economic Research*, page 158, 1934.

[JR04] B H Juang and Lawrence R Rabiner. Automatic Speech Recognition – A Brief History of the Technology Development. 2004.

[Kev09] Kevin Ashton. That 'Internet of Things' Thing. RFID Journal, retrieved from http://www.rfidjournal.com/articles/view?4986, 2009.

[LF18] André Sousa Lago and Hugo Sereno Ferreira. Conversation-based complex event management in smart-spaces. arXiv, retrieved from https://arxiv.org/abs/1807.07047, 2018.

[LL15] In Lee and Kyoochun Lee. The Internet of Things (IoT): Applications, investments, and challenges for enterprises. *Business Horizons*, 58(4):431–440, Jul 2015.

[Mar16] Bernard Marr. 15 noteworthy facts about Wearables in 2016. Forbes, retrieved from https://www.forbes.com/sites/bernardmarr/2016/03/18/15-mind-boggling-facts-about-wearables-in-2016/{#}79ad0f5e2732http://www.forbes.com/sites/bernardmarr/2016/03/18/15-mind-boggling-facts-about-wearables-in-2016/{#}389fed3f4773, 2016.

[McF17] Matt McFarland. Why Volvo's self-driving garbage truck spends most of its time in reverse. CNN Money, retrieved from http://money.cnn.com/2017/05/18/technology/volvo-garbage-truck/index.html, May 2017.

[Mou17] Rahul Mourya. IoT applications spanning across industries. IBM Blog, available at https://www.ibm.com/blogs/internet-of-things/iot-applications-industries/, Apr 2017.

[MR69] Albert Edward Musson and Eric Robinson. *Science and technology in the industrial revolution*. Manchester University Press, 1969.

[Oli17] Oliver Balch. Automated mining will cost jobs and tax income: it's time for governments to act. The Guardian, retrieved from https://www.theguardian.com/sustainable-business/2017/jan/20/autonomous-mining-will-cost-jobs-and-tax-income-its-time-for-governments-to-act, 2017.

[Pat17] Kevin Patel. 6 Benefits of Internet of Things for Hospitals and Healthcare. IBM IoT Blog, retrieved from https://www.ibm.com/blogs/internet-of-things/6-benefits-of-iot-for-healthcare/, 2017.

[Pet17] Hayley Peterson. How automation will impact the retail industry. Business Insider, retrieved from http://www.businessinsider.com/how-automation-will-impact-the-retail-industry-2017-5, 2017.

[Ran03] Dave Randall. Living inside a smart home: A case study. In *Inside the smart home*, pages 227–246. Springer, 2003.

[Ras14] Katharina Rasch. An unsupervised recommender system for smart homes. *Journal of Ambient Intelligence and Smart Environments*, 6:21–37, 2014.

[RvdM14] Janessa Rivera and Rob van der Meulen. Gartner Says 4.9 Billion Connected "Things" Will Be in Use in 2015. *Gartner - Newsroom*, pages 9–10, 2014.

[SDPA14] P. Suresh, J. Vijay Daniel, V. Parthasarathy, and R. H. Aswathy. A state of the art review on the Internet of Things (IoT) history, technology and fields of deployment. In *2014 International Conference on Science Engineering and Management Research (ICSEMR)*. IEEE, Nov 2014.

[Sea03] Sean Dodson. The internet of things. The Guardian, retrieved from https://www.theguardian.com/technology/2003/oct/09/shopping.newmedia, 2003.

[SRC+16] Joao Santos, Joel J. P. C. Rodrigues, Joao Casal, Kashif Saleem, and Victor Denisov. Intelligent Personal Assistants Based on Internet of Things Approaches. *IEEE Systems Journal*, pages 1–10, 2016.

[Tec17] Maruti Techlabs. Exploring Dialogflow: Understanding Agent Interaction. Medium, retrieved from https://chatbotslife.com/which-are-the-best-on-site-chatbot-frameworks-3dbf5157fb57, April 2017.

[vdM17] Rob van der Meulen. Gartner Press Release, Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016, 2017.

[XHL14] Li Da Xu, Wu He, and Shancang Li. Internet of things in industries: A survey. IEEE Transactions on Industrial Informatics, Nov 2014.

[XYWV12] Feng Xia, Laurence T Yang, Lizhe Wang, and Alexey Vinel. Internet of Things. *INTERNATIONAL JOURNAL OF COMMUNICATION SYSTEMS Int. J. Commun. Syst. Int. J. Commun. Syst*, 25(25), 2012.

REFERENCES

Appendix A

Sample Intent JSON Representations

This appendix includes sample JSON representations of intents defined in Jarvis' Dialogflow project.

A.1 Direct Action Intent


```
{
  "id": "5615a93a-a265-4d27-841c-9cec1f19e6b2",
  "timestamp": "2018-06-09T16:17:21.698Z",
  "lang": "en",
  "result": {
    "source": "agent",
    "resolvedQuery": "turn on the light",
    "action": "",
    "actionIncomplete": false,
    "parameters": {
      "action": {
        "action-onoff": {
          "status": "on",
          "actuator": {
            "light-switch": "light"
          }
        }
      }
    },
    "contexts": [
      {
        "name": "confirm-thing-choice",
```

```
        "parameters": {
          "request": "{\"result\":{\"actionIncomplete\":false,\"
             score\":1,\"metadata\":{\"intentName\":\"Direct
             Action\",\"intentId\":\"8b4ac42a-6b56-43a0-9fbb-
             ca5da1e7d6d1\",\"webhookUsed\":\"true\",\"
             webhookForSlotFillingUsed\":\"false\"},\"
             resolvedQuery\":\"turn on the light\",\"speech
             \":\"\",\"action\":\"\",\"source\":\"agent\",\"
             contexts\":[],\"fulfillment\":{\"speech\":\"Sorry, I
             was not able to help.\",\"messages\":[{\"speech\":\"
             Sorry, I was not able to help.\",\"type\":0}]},\"
             parameters\":{\"action\":{\"action-onoff\":{\"
             actuator\":{\"light-switch\":\"light\"},\"status\":\"
             on\"}}}},\"id\":\"5615a93a-a265-4d27-841c-9
             cec1f19e6b2\",\"sessionId\":\"de3a07c8-6a90-bb4a-341d
             -d694317dac53\",\"lang\":\"en\",\"timestamp
             \":\"2018-06-09T16:17:21.698Z\",\"status\":{\"code
             \":200,\"errorType\":\"success\"}}",
          "choices": [
            "bedroom light",
            "living room light"
          ],
          "intent": "OnOffSubIntent"
        },
        "lifespan": 1
      }
    ],
    "metadata": {
      "intentId": "8b4ac42a-6b56-43a0-9fbb-ca5da1e7d6d1",
      "webhookUsed": "true",
      "webhookForSlotFillingUsed": "false",
      "webhookResponseTime": 5129,
      "intentName": "Direct Action"
    },
    "fulfillment": {
      "messages": [
        {
          "speech": "Do you mean bedroom light or living room
             light?",
          "type": 0
```

```
        }
      ]
    },
    "score": 1
  },
  "status": {
    "code": 200,
    "errorType": "success"
  },
  "sessionId": "de3a07c8-6a90-bb4a-341d-d694317dac53"
}
```


Listing A.1: JSON object generated by Dialogflow for the intent "Direct Action"

A.2 Delayed Action Intent


```
{
  "id": "64f0ac94-4e63-4011-a526-f4dfe63bc730",
  "timestamp": "2018-03-07T15:53:22.624Z",
  "lang": "en",
  "result": {
    "source": "agent",
    "resolvedQuery": "turn the light on tomorrow at 5pm",
    "action": "",
    "actionIncomplete": false,
    "parameters": {
      "time": {
        "date-time": "2018-04-09T19:52:00Z"
      },
      "action": {
        "action-onoff": {
          "actuator": {
            "light-switch": "bedroom light"
          },
          "status": "on"
        }
      }
    },
```

```
      "contexts": [],
      "metadata": {
        "intentId": "36e77c05-5ada-404d-8de5-cf7a7c1f2f96",
        "webhookUsed": "false",
        "webhookForSlotFillingUsed": "false",
        "intentName": "Delayed Action"
      },
      "fulfillment": {
        "speech": "Delayed? Sure",
        "messages": [
          {
            "type": 0,
            "speech": "Delayed? Sure"
          }
        ]
      },
      "score": 1
    },
    "status": {
      "code": 200,
      "errorType": "success",
      "webhookTimedOut": false
    },
    "sessionId": "8e854d8c-db7b-487d-a248-f8b9eeca3ad1"
  }
```


Listing A.2: JSON object generated by Dialogflow for the intent "Delayed Action"

A.3 Confirm Thing Choice Intent


```
{
  "id": "8f12acf6-7868-4aa9-bfac-d90bd48bfcab",
  "timestamp": "2018-06-09T16:28:41.284Z",
  "lang": "en",
  "result": {
    "source": "agent",
    "resolvedQuery": "the first",
    "action": "",
```

```
    "actionIncomplete": false,
    "parameters": {
      "thing": "",
      "ordinal": 1
    },
    "contexts": [],
    "metadata": {
      "intentId": "5f3f7eb8-c2dc-42db-9f59-38a9bf0c3a52",
      "webhookUsed": "true",
      "webhookForSlotFillingUsed": "false",
      "webhookResponseTime": 2363,
      "intentName": "Confirm Thing Choice"
    },
    "fulfillment": {
      "messages": [
        {
          "speech": "Done!",
          "type": 0
        }
      ]
    },
    "score": 1
  },
  "status": {
    "code": 200,
    "errorType": "success"
  },
  "sessionId": "de3a07c8-6a90-bb4a-341d-d694317dac53"
}
```


Listing A.3: JSON object generated by Dialogflow for the intent "Confirm Thing Choice"

A.4 Repeating Intent


```
{
  "id": "d0616836-4331-4b23-895d-182f9aedb94c",
  "timestamp": "2018-03-19T12:43:38.313Z",
  "lang": "en",
```

```
    "result": {
      "source": "agent",
      "resolvedQuery": "turn on the light everyday at 5pm",
      "action": "",
      "actionIncomplete": false,
      "parameters": {
        "action": {
          "action-onoff": {
            "status": "on",
            "actuator": {
              "light-switch": "light"
            }
          }
        },
        "periodtime": {
          "time": "17:00:00"
        }
      },
      "contexts": [],
      "metadata": {
        "intentId": "977ef8d4-1096-4733-b724-82bad2931fc8",
        "webhookUsed": "false",
        "webhookForSlotFillingUsed": "false",
        "intentName": "Repeating Intent"
      },
      "fulfillment": {
        "speech": "Well, okay",
        "messages": [
          {
            "type": 0,
            "speech": "Well, okay"
          }
        ]
      },
      "score": 1
    },
    "status": {
      "code": 200,
      "errorType": "success",
      "webhookTimedOut": false
```

```
    },
    "sessionId": "7c3e8cc7-b596-4dcf-9709-a4f4d618e8b5",
    "alternativeResultsFromKnowledgeService": {}
}
```


Listing A.4: JSON object generated by Dialogflow for the intent "Repeating Intent"

A.5 Event Intent


```
{
  "id": "bfdbc989-e113-4fd6-b7b7-956282d16031",
  "timestamp": "2018-03-29T14:28:57.939Z",
  "lang": "en",
  "result": {
    "source": "agent",
    "resolvedQuery": "turn on the bedroom light when the living room light turns off",
    "action": "",
    "actionIncomplete": false,
    "parameters": {
      "action": {
        "action-onoff": {
          "status": "on",
          "actuator": {
            "light-switch": "bedroom light"
          }
        }
      },
      "event": {
        "on-off-actuator": {
          "light-switch": "living room light"
        },
        "on-off-status": "off"
      }
    },
    "contexts": [],
    "metadata": {
      "intentId": "d234e808-2fd7-420f-9aef-1852750f8bc6",
```

```
        "webhookUsed": "false",
        "webhookForSlotFillingUsed": "false",
        "intentName": "Event Intent"
      },
      "fulfillment": {
        "speech": "Sorry, I was unable to help.",
        "messages": [
          {
            "type": 0,
            "speech": "Sorry, I was unable to help."
          }
        ]
      },
      "score": 1
    },
    "status": {
      "code": 200,
      "errorType": "success",
      "webhookTimedOut": false
    },
    "sessionId": "41d7872b-eb38-4e5f-84aa-03f7db824748"
  }
```


Listing A.5: JSON object generated by Dialogflow for the intent "Event Intent"

A.6 Why Did Something Happen Intent


```
  {
    "id": "49b3c4e3-9317-499e-9464-bdaec4b04b85",
    "timestamp": "2018-04-07T13:41:07.296Z",
    "lang": "en",
    "result": {
      "source": "agent",
      "resolvedQuery": "why did the living room light turn on",
      "action": "",
      "actionIncomplete": false,
      "parameters": {
        "action-past": {
```

```
          "action-past-onoff": {
            "actuator": {
              "light-switch": "bedroom light"
            },
            "status": "on"
          }
        }
      },
      "contexts": [],
      "metadata": {
        "intentId": "a835dba1-f963-4b82-a41f-c8dbf6eb8c19",
        "webhookUsed": "false",
        "webhookForSlotFillingUsed": "false",
        "intentName": "Why did something happen?"
      },
      "fulfillment": {
        "speech": "I'm sorry but I was not able to help.",
        "messages": [
          {
            "type": 0,
            "speech": "I'm sorry but I was not able to help."
          }
        ]
      },
      "score": 1
    },
    "status": {
      "code": 200,
      "errorType": "success",
      "webhookTimedOut": false
    },
    "sessionId": "8290b4d1-4f06-4fdc-a40e-d01736c58ad7"
  }
```


Listing A.6: JSON object generated by Dialogflow for the intent "Why did something happen?"

A.7 Alias Intent

```
{
  "id": "0ff9ee90-3cba-4067-a49e-bf2b68748eae",
  "timestamp": "2018-05-02T16:20:13.249Z",
  "lang": "en",
  "result": {
    "source": "agent",
    "resolvedQuery": "create an alias for turn up the heat",
    "action": "",
    "actionIncomplete": false,
    "parameters": {
      "alias": "turn up the heat"
    },
    "contexts": [],
    "metadata": {
      "intentId": "b9038171-11bb-4024-91b4-cad41fdcc9e6",
      "webhookUsed": "true",
      "webhookForSlotFillingUsed": "false",
      "webhookResponseTime": 8499,
      "intentName": "Alias Intent"
    },
    "fulfillment": {
      "speech": "Sorry, I was not able to help.",
      "messages": [
        {
          "type": 0,
          "speech": "Sorry, I was not able to help."
        }
      ]
    },
    "score": 1
  },
  "status": {
    "code": 206,
    "errorType": "partial_content",
    "errorDetails": "Webhook call failed. Error: 503 Service
       Unavailable"
  },
  "sessionId": "857547d2-9e74-43b0-af54-c95459b486a4",
  "isStackdriverLoggingEnabled": false
}
```

Listing A.7: JSON object generated by Dialogflow for the intent "Alias Intent"

A.8 Set Alias Type Intent


```
{
  "id": "4fe0e2d7-67e1-4873-a178-d19ad4d8d3fe",
  "timestamp": "2018-05-02T16:24:37.74Z",
  "lang": "en",
  "result": {
    "source": "agent",
    "resolvedQuery": "execute turn on the light",
    "action": "",
    "actionIncomplete": false,
    "parameters": {
      "event": "",
      "action": {
        "action-onoff": {
          "status": "on",
          "actuator": {
            "light-switch": "light"
          }
        }
      }
    },
    "contexts": [
      {
        "lifespan": 1,
        "name": "alias-creation-context",
        "parameters": {
          "alias": ""
        }
      }
    ],
    "metadata": {
      "intentId": "99637bce-5049-4fc4-92f6-da12110059ed",
      "webhookUsed": "true",
      "webhookForSlotFillingUsed": "false",
```

```
      "webhookResponseTime": 2946,
      "intentName": "Alias Intent | Set Alias Type"
    },
    "fulfillment": {
      "speech": "Sorry, I was not able to help.",
      "messages": [
        {
          "type": 0,
          "speech": "Sorry, I was not able to help."
        }
      ]
    },
    "score": 1
  },
  "status": {
    "code": 206,
    "errorType": "partial_content",
    "errorDetails": "Webhook call failed. Error: 503 Service
       Unavailable"
  },
  "sessionId": "857547d2-9e74-43b0-af54-c95459b486a4",
  "isStackdriverLoggingEnabled": false
}
```


Listing A.8: JSON object generated by Dialogflow for the intent "Alias Intent | Set Alias Type"

A.9 Rules Defined Intent


```
{
  "id": "1abcefa2-d854-4840-b063-9a6b0aff30d2",
  "timestamp": "2018-04-09T15:50:52.161Z",
  "lang": "en",
  "result": {
    "source": "agent",
    "resolvedQuery": "what rules are defined for the living room
       light",
    "action": "",
    "actionIncomplete": false,
```

```
      "parameters": {
        "thing": {
          "light-switch": "living room light"
        }
      },
      "contexts": [],
      "metadata": {
        "intentId": "51d173a2-2efd-4641-befa-b06dc692f155",
        "webhookUsed": "false",
        "webhookForSlotFillingUsed": "false",
        "intentName": "Rules Defined"
      },
      "fulfillment": {
        "speech": "Sorry, I was not able to help with that.",
        "messages": [
          {
            "type": 0,
            "speech": "Sorry, I was not able to help with that."
          }
        ]
      },
      "score": 0.8299999833106995
    },
    "status": {
      "code": 200,
      "errorType": "success",
      "webhookTimedOut": false
    },
    "sessionId": "8290b4d1-4f06-4fdc-a40e-d01736c58ad7"
  }
```


Listing A.9: JSON object generated by Dialogflow for the intent "Rules Defined"

A.10 Change Single Rule Intent


```
  {
    "id": "3412015b-1f94-4a2a-acaf-991e000486b1",
    "timestamp": "2018-04-11T23:42:46.52Z",
```

```
    "lang": "en",
    "result": {
      "source": "agent",
      "resolvedQuery": "change that to when the living room light is
          on",
      "action": "",
      "actionIncomplete": false,
      "parameters": {
        "event": {
          "on-off-actuator": {
            "light-switch": "living room light"
          },
          "on-off-status": "on"
        }
      },
      "contexts": [
        {
          "lifespan": 1,
          "name": "edit-single-rule",
          "parameters": {
            "event": "{\"id\":\"2144309538794832901\",\"tag\":\"
               booleanEventHandler\",\"value\":false,\"consumer
               \":{\"event\":{\"ON_OFF\":{\"unit\":\"boolean\",\"
               description\":\"Whether light is on\",\"href\":\"/
               house/living room light/events/ON_OFF\",\"type\":\"
               boolean\"}},\"thing\":{\"name\":\"living room light
               \",\"description\":\"On/Off light switch\",\"links
               \":{\"actions\":\"/house/living room light/actions
               \",\"properties\":\"/house/living room light/
               properties\",\"events\":\"/house/living room light/
               events\"},\"type\":\"onOffLight\",\"properties\":{\"
               status\":{\"description\":\"Describes current state
               of the switch (true=on)\",\"type\":\"boolean\"}},\"
               events\":{\"ON_OFF\":{\"unit\":\"boolean\",\"
               description\":\"Whether light is on\",\"href\":\"/
               house/living room light/events/ON_OFF\",\"type\":\"
               boolean\"}}}},\"command\":{\"id
               \":6840089621934373932,\"type\":\"onOffCommand\",\"
               thing\":\"bedroom light\",\"status\":true}}"
          }
```

```
        }
      ],
      "metadata": {
        "intentId": "9798598e-cb3c-4bcc-bb68-3721a1ec3b23",
        "webhookUsed": "false",
        "webhookForSlotFillingUsed": "false",
        "intentName": "Rules Defined | Change Single Rule"
      },
      "fulfillment": {
        "speech": "",
        "messages": [
          {
            "type": 0,
            "speech": ""
          }
        ]
      },
      "score": 1
    },
    "status": {
      "code": 200,
      "errorType": "success",
      "webhookTimedOut": false
    },
    "sessionId": "a4bb052c-5238-4a73-a7f9-95595bffa42f"
  }
```


Listing A.10: JSON object generated by Dialogflow for the intent "Rules Defined | Change Single Rule"

A.11 Cancel Command Intent


```
  {
    "id": "fbcf0b39-a211-4f6a-a51c-fddfdf8ca8d3",
    "timestamp": "2018-04-02T16:03:55.586Z",
    "lang": "en",
    "result": {
      "source": "agent",
```

```
    "resolvedQuery": "cancel my last command",
    "action": "",
    "actionIncomplete": false,
    "parameters": {},
    "contexts": [
      {
        "name": "confirm-cancel-event",
        "parameters": {},
        "lifespan": 1
      }
    ],
    "metadata": {
      "intentId": "62544394-9d76-42da-a40b-6a6cda34066b",
      "webhookUsed": "true",
      "webhookForSlotFillingUsed": "false",
      "webhookResponseTime": 2284,
      "intentName": "Cancel Command"
    },
    "fulfillment": {
      "messages": [
        {
          "speech": "Sorry, I found no command to cancel.",
          "type": 0
        }
      ]
    },
    "score": 1
  },
  "status": {
    "code": 200,
    "errorType": "success",
    "webhookTimedOut": false
  },
  "sessionId": "41d7872b-eb38-4e5f-84aa-03f7db824748"
}
```


Listing A.11: JSON object generated by Dialogflow for the intent "Cancel Command"

A.12 Confirm Cancel Intent

```json
{
  "id": "70caa64e-9a81-4d73-9115-c939a12b0392",
  "timestamp": "2018-04-02T20:05:32.543Z",
  "lang": "en",
  "result": {
    "source": "agent",
    "resolvedQuery": "yes",
    "action": "",
    "actionIncomplete": false,
    "parameters": {},
    "contexts": [
      {
        "name": "confirm-cancel-event",
        "parameters": {
          "commandId": "6053189115902158217"
        },
        "lifespan": 1
      }
    ],
    "metadata": {
      "intentId": "145e7d3b-9105-4bfb-8c6a-78c3e5c2783b",
      "webhookUsed": "true",
      "webhookForSlotFillingUsed": "false",
      "webhookResponseTime": 522,
      "intentName": "Confirm Cancel"
    },
    "fulfillment": {
      "messages": [
        {
          "speech": "Sorry, I was not able to cancel that event.",
          "type": 0
        }
      ]
    },
    "score": 1
  },
  "status": {
    "code": 200,
    "errorType": "success",
```

```
    "webhookTimedOut": false
  },
  "sessionId": "41d7872b-eb38-4e5f-84aa-03f7db824748"
}
```


Listing A.12: JSON object generated by Dialogflow for the intent "Confirm Cancel"

Appendix B

Sample Jarvis MongoDB Documents

This appendix includes examples of objects stored in Jarvis' Mongo database for commands, event history and user commands.

B.1 Commands

```json
{
  "_id":"5ae9e3ae55d6b72c3cf71984",
  "undo":false,
  "success":true,
  "commandText":"[OnOffCommand] Executed device on (living room light)",
  "command":{
    "id":"614569302563896791",
    "type":"onOffCommand",
    "thing":"living room light",
    "status":true
  },
  "timestamp":"1525277614116"
}
```

Listing B.1: JSON stored in the database representing a command

B.2 User Commands

```json
{
```

```
  "_id":"5ae9e39755d6b72c3cf71982",
  "undo":false,
  "success":true,
  "commandText":"[Event] do [OnOffCommand] Executed device on (
     living room light) when living room motion sensor is
     activated",
  "command":{
     "id":"3210187144565540456",
     "type":"eventCommand",
     "event":{
        "on-off-status":"on",
        "binary_sensor":"living room motion sensor"
     },
     "command":{
        "id":"6145693025638967911",
        "type":"onOffCommand",
        "thing":"living room light",
        "status":true
     }
  },
  "timestamp":"1525277591322"
}
```


Listing B.2: JSON stored in the database representing a user command

B.3 Event History


```
{
  "_id":"5ae9e3ae55d6b72c3cf71986",
  "event":{
     "condition":true,
     "id":"6942623486896140466",
     "tag":"binaryTriggerEventHandler",
     "consumer":{
        "event":{
           "TRIGGER":{
              "unit":"boolean",
              "description":"If sensor is activated",
```

```
               "href":"/house/living room motion sensor/events/trigger",
               "type":"trigger"
            }
         },
         "thing":{
            "name":"living room motion sensor",
            "description":"Binary Sensor",
            "links":{
               "actions":"/house/living room motion sensor/actions",
               "properties":"/house/living room motion sensor/properties",
               "events":"/house/living room motion sensor/events"
            },
            "type":"binarySensor",
            "properties":{
               "value":{
                  "description":"Is triggered (\"on\") if the sensor is activated",
                  "href":"/house/living room motion sensor/properties/value",
                  "type":"boolean"
               }
            },
            "events":{
               "TRIGGER":{
                  "unit":"boolean",
                  "description":"If sensor is activated",
                  "href":"/house/living room motion sensor/events/trigger",
                  "type":"trigger"
               }
            }
         }
      },
      "command":{
         "id":"6145693025638967 91",
         "type":"onOffCommand",
         "thing":"living room light",
```

```
            "status":true
        }
    },
    "timestamp":"1525277614852"
}
```


Listing B.3: JSON stored in the database representing an event