Software and Systems Modeling
https://doi.org/10.1007/s10270-022-01006-z

SPECIAL SECTION PAPER

Model-based fleet deployment in the IoT–edge–cloud continuum

Hui Song[1] · Rustem Dautov[1] · Nicolas Ferry[2] · Arnor Solberg[3] · Franck Fleurey[3]

Received: 23 February 2021 / Revised: 31 January 2022 / Accepted: 28 March 2022
© The Author(s) 2022

Abstract

With the increasing computing and networking capabilities, IoT devices and edge gateways have become part of a larger IoT–edge–cloud computing continuum, where processing and storage tasks are distributed across the whole network hierarchy, not concentrated only in the cloud. At the same time, this also introduced continuous delivery practices to the development of software components for network-connected gateways and sensing/actuating nodes. These devices are placed on end users' premises and are characterized by continuously changing cyber-physical contexts, forcing software developers to maintain multiple application versions and frequently redeploy them on a distributed fleet of devices with respect to their current contexts. Doing this correctly and efficiently goes beyond manual capabilities and requires an intelligent and reliable automated solution. This paper describes a model-based approach to automatically assigning multiple software deployment plans to hundreds of edge gateways and connected IoT devices implemented in collaboration with a smart healthcare application provider. From a platform-specific model of an existing edge computing platform, we extract a platform-independent model that describes a list of target devices and a pool of available deployment plans. Next, we use constraint solving to automatically assign deployment plans to devices at once with respect to their specific contexts. The result is transformed back into the platform-specific model and includes a suitable deployment plan for each device, which is then consumed by our engine to deploy software components not only on edge gateways but also on their downstream IoT devices with constrained resources and connectivity. We validate the approach with a fleet deployment prototype integrated into a DevOps toolchain used by the partner application provider. Initial experiments demonstrate the viability of the approach and its usefulness in supporting DevOps for edge and IoT software development.

Keywords Software deployment · IoT · Model-based software engineering · Device fleet · DevOps · Constraint solving

1 Introduction

The Internet of things (IoT) is characterized by the ubiquitous presence of embedded devices and network gateways at the very edge of the network. Due to bandwidth limitations and strict latency requirements, as well as increasing computing capabilities and hardware miniaturization, more and more data processing takes place closer to the source. This process is underpinned by the emergence of *Edge Computing*, where software components are deployed on devices at the *edge* of the network, such as gateways, routers and small base stations. An edge computing application may comprise tens to thousands of distributed devices, collectively referred to as a *fleet*.

Edge application providers are essentially software developers and typically need to follow the modern DevOps practice to reduce the time between committing a change to a system and the change being placed into production

Communicated by S. Abrahão, E. Syriani, H. Sahraoui, and J. de Lara.

✉ Hui Song
Hui.Song@sintef.no

Rustem Dautov
Rustem.Dautov@sintef.no

Nicolas Ferry
nicolas.ferry@univ-cotedazur.fr

Arnor Solberg
Arnor.Solberg@tellu.no

Franck Fleurey
Franck.Fleurey@tellu.no

1 SINTEF Digital, Forskningsveien 1, 0373 Oslo, Norway

2 I3S/INRIA Kairos, Université Cote d'Azur, Sophia Antipolis, France

3 Tellu IoT AS, Lensmannslia 4, 1386 Asker, Norway

Published online: 03 May 2022

[7]. The practical challenge is how to automatically assign and deploy the updated software to the many distributed edge devices after each DevOps iteration. Unlike the centralized cloud model, where computing resources are relatively homogeneous, an edge fleet consists of distributed and heterogeneous devices, which have different cyber-physical contexts in terms of hardware capacity, network connection, user preferences, etc. [21]. Developers need to maintain multiple variants of the software to fit such different device contexts, e.g. some devices require a variant of the software optimized to the specific hardware capacity, while some others with lower network bandwidth may need a variant with low communication cost. This raises the main problem for the automatic deployment of edge computing applications, i.e. how to assign m deployment plans (each 'deployment plan' represents a unique variant of the whole software product)[1] to n devices, so that each device is assigned with a proper deployment plan that matches its context. At the same time, the whole system is required to meet its global goals, e.g. keep the software diversity in the fleet or select a designated number of devices for software preview.

The start-of-the-art deployment tools automate the installation of one application on one device, or a pre-defined set of devices, but lack the support for distributing multiple variants across a large fleet. They also do not provide sufficient automated support for updating devices with constrained resources and limited (or none) Internet connectivity [42]. Such embedded and microcontroller-enabled devices traditionally have been flashed with 'one-off' firmware not intended to be updated in the future. Until recently, they have not been seen as active contributors to the common pool of shared computing resources, which can be iteratively assigned and deployed with updated firmware. This situation has changed with the emergence of the IoT–edge–cloud computing continuum, where computing and storage tasks are distributed across all three levels. Even though the mainstream IoT/Edge fleet management platforms offer tools to keep track and maintain multiple deployments, devices, and their contexts, developers still need to manually designate which deployment goes to which device.

The lack of fleet deployment support, i.e.the lack of support for deploying many variants on the fleet as a whole instead of individual devices, has become a bottleneck for DevOps teams of IoT and edge applications. Developers have to either deploy the application on the devices one by one, or manually maintain a complicated mapping between variants and devices. This significantly increases the time and effort spent on deployment and make continuous deployment impossible even for relatively small number of variants or devices. Moreover, for many IoT applications, the set of devices is not static and can dynamically change over time—i.e. new devices may join the fleet at any time, some devices may be unavailable for some periods, or move around causing the change in their context. Such changes often require redeployment on some devices. Without automatic fleet deployment, developers will be distracted by these deployment tasks, which can be a heavy burden hindering the continuous deployment pace.

Aiming at this challenge, this paper presents an industrial research that applies model-based techniques to achieve automatic fleet deployment, by assigning multiple deployment plans to many devices in a fleet, without human interaction. The approach takes as input a platform-specific model (PSM) from a fleet management platform and transforms it into a platform-independent model (PIM) that represents deployment plans, devices and their context. After that, the approach applies constraint solving techniques to the PIM to map deployment plans to devices, guided by a set of hard and soft constraints based on the domain knowledge. The resulting PIM is then transformed back into the PSM, which will be used by the platform to actually deploy the software. In many cases, software is deployed not only to edge devices, but also to associated downstream devices such as sensors and microcontrollers. These *leaf* devices typically have neither a full-fledged operating system, nor an independent Internet connection. To automate such 'last-mile deployment'—i.e. firmware updates on IoT devices—we integrate a general modelling language for software deployment on heterogeneous devices, and the corresponding deployment engine.

We implemented the approach as a fleet deployment tool and integrated it into the DevOps toolset currently adopted by the partner Tellu, which is an eHealth application provider. An experimental scenario using the tool through a series of DevOps iterations shows that the approach is able to produce valid assignments and to provide valuable feedback for planning the subsequent development. In addition, we present a production-based use case to demonstrate how the fleet deployment tools can be used by DevOps teams. Accordingly, the contribution of this paper is threefold:

– A novel approach to automatically deploy multiple software variants to a fleet of edge and IoT devices, with respect to device contexts and global goals of the fleet.

[1] Throughout the paper, we use the terms 'deployment plan' and 'variant' interchangeably. Both terms represent the software to be deployed on a device, whereas the former represents the perspective of deployment and the latter refers to the perspective of development. That is, a software variant represents a unique way of deploying the software, including the actual code to deploy, the ways the code is built, and how it is supposed to be configured during deployment. A similar term 'version' is reserved to represent a development iteration in the software life cycle. As further discussed below, different versions of software may be deployed as different variants, but they are not the only source of emerging variants.

- A set of hard and soft constraints used to correctly assign software variants and evenly distribute them across the available matching devices (i.e. to what we refer as *even distribution*).
- A research pilot on an industrial use case, demonstrating how the proposed model-based approach with meta-modelling and constraint solving can automate key DevOps activities and increase the development productivity.

As an extension to our previous publication [51], this version extends the original scope of assigning multiple software variants to a device fleet and also addresses the deployment of the assigned software on edge devices and its associated resource-constrained IoT devices with no direct Internet connection. We combine the tools that implement these two approaches into a complete fleet deployment bundle. This way, we aim to support software assignment and deployment across the whole IoT–edge–cloud computing continuum. We practically demonstrate how the entire fleet deployment concept and the tool bundle support the DevOps practices.

The rest of the paper is organized as follows. Section 2 describes the fleet deployment problem via a motivating example. Section 3 presents the modelling languages and the engine for software deployment on one edge device and its associated downstream devices. Section 4 presents our proposed model-based approach, and Sect. 5 details the use of constraint solving for fleet assignment. Section 6 presents the tool implementation and demonstrates its viability in an established DevOps process. Section 7 introduces the technical implementation of the fleet deployment tools and a production-level use case to show how they are used by developers. Section 8 compares the approach with the existing related works, and Sect. 10 concludes the paper by discussing some limitations of the current work and the future steps.

2 Problem statement

2.1 Motivating example

Throughout this paper, we will focus on a remote patient monitoring (RPM) scenario offered by the partner eHealth service provider Tellu,[2] with whom two coauthors of this paper are affiliated. For each of their customers (typically elderly people living at their own residences), they provide a healthcare gateway—a small single-board computer similar to Raspberry Pi—together with a set of medical sensors, cameras and wearable emergency beepers. Each gateway collects measurements such as blood pressure, glucose and oxygen levels, via Bluetooth, processes and aggregates the data, and sends them to the cloud back-end. The patients and their nurses have access to the data via a web interface and a mobile App. For some patients, Tellu sends their technicians to mount the gateway on the wall, while other patients can opt for having the gateway delivered from the manufacturer and installing it themselves. Some patients choose battery-powered portable gateways to have a possibility to carry them with the essential sensors whenever they are outside their houses, e.g. for walking, taking medical examinations, or travelling. A recent feature under development by Tellu is *fall risk detection* based on image processing, which combines live gesture detection via cameras together with other real-time physiological and environmental data to continuously assess the risk of patient falling down shortly.

Since all the devices can be purchased off the shelf, Tellu is essentially a software vendor. Their main effort and focus is on developing the front-end and back-end software components, running on the IoT/Edge gateway and the cloud, respectively. The development team continuously adds new features to the software and patches issues, following a DevOps practice. During this process, they produce and simultaneously maintain multiple variants of the front-end software, which conceptually may be separated into *vertical* and *horizontal* ones. Vertically, they have *development versions* with the most recent features running on staging devices and *released versions* with mature features running on production-ready devices already used by patients. They also regularly deploy *preview* versions with new features to a small set of selected patients, in order to collect feedback from real users. Horizontally, they maintain software variants that fit gateways with different set-ups and contexts. For example, if a variant has a machine learning (ML) module for fall detection running in the back-end, then it only fits those gateways that are connected to a WiFi network, not 4G, since they need to send images to the cloud. Alternatively, the variant with the ML module running fully on the edge only fits the gateways that are mounted on the wall, since the heavy computation load implies risks of overheating the self-installed gateways (which usually do not have proper cooling/ventilation facilities). As an option, some patients may choose to buy an additional hardware accelerator (typically in the form of a pluggable USB dongle, such as Google Coral[3]), which can reduce the computation load on the gateway, but requires an application variant with a pre-compiled ML module). Moreover, as Tellu offers their services in a multi-tenant way, where some tenants (normally institutes or companies providing healthcare services) may require customized features involving new software and hardware. For example, one tenant requires a wearable alarm button, with receivers connected to the gateway, used to alert supervising nurses on sudden health deterioration and temporarily elevating access

[2] https://www.tellucloud.com/.

[3] https://coral.ai/.

rights in such situations. A corresponding software variant has to be deployed on the gateways belonging to this tenant to implement the authentication and operation of the alarm buttons. Since no variant fits every device, Tellu has to maintain multiple variants (i.e. deployment plans) and assign them to the devices according to their contexts.

In summary, Tellu maintains a fleet of about 500 gateways for their patients, and additional 10 local installations on their own premises. The gateways have different contexts in terms of network connection, mounting, hardware acceleration, etc. The development team maintains around 3–10 different variants of the front-end software. The Tellu team expects to conduct DevOps cycles on a daily basis, retiring parts or all of the active variants and introducing new ones. However, since there is *(i)* a mass redeployment after each cycle involving many or sometimes all the devices, and *(ii)* it is not fully automatic at the moment, they cannot achieve such high frequent DevOps cycles. Automatic fleet deployment becomes a bottleneck for the development team.

2.2 Fleet deployment: the concepts

Fleet deployment is a problem of automatically deploying software on many functionally similar devices (or groups of devices), while the actual software deployed may vary from one device to another, depending on the current context. Therefore, the main objective of fleet deployment is to exempt developers from the requirement to care about the specific deployment on individual devices, but instead allow them to deploy software on the fleet as a whole. To achieve this goal, we need automated tools to identify matching devices and actually deploy the software variants on these target devices.

Figure 1 illustrates the conceptual architecture of a fleet deployment framework, with three main components:

- *Fleet monitoring*, which overlooks the life cycle of all the devices and collects the context[4] for each device;
- *Fleet assignment*, which maintains active candidate deployment plans and assigns one to each device.
- *Device deployment*, which enacts the assigned deployment plan on each device, by installing software modules on the edge gateway itself and its associated IoT devices.

There are several commercial IoT/edge cloud platforms providing rich *fleet monitoring* support, such as AWS IoT Greengrass,[5] Balena Cloud,[6] and Azure IoT Edge.[7] They are typically able to maintain a list of all registered devices and perform continuous monitoring and status tracking of each device. They also provide cloud-device communication channels for implementing custom run-time status and context data collection from each device in the managed fleet. However, as opposed to fleet monitoring, device deployment and fleet assignment are still open questions for software applications across the whole IoT–edge–cloud continuum. In the following, we detail the challenges associated with device deployment and fleet assignment.[8]

2.3 Device deployment

Device deployment can be seen as an infrastructure as code (IaC) problem, extended to the context of edge computing, where target nodes are not cloud-based virtual machines, but rather heterogeneous and often resource-constrained devices. There are mature model-based IaC solutions for cloud environments, where developers provide a model specifying where to obtain software modules and how to install and configure them. The corresponding IaC engine will read the model and deploy the modules accordingly on the target nodes. Mainstream IaC solutions[9] have recently started providing support for edge devices. However, the current solutions rely on the assumption that all target resources have a functional operation system (mostly Linux) and are directly accessible by the IaC engine. This assumption is valid in the cloud, but when extended to edge and IoT devices, the heterogeneity of resources considerably increases, and many types of devices have no proper operating system and are not directly connected to the Internet.

Taking the RPM system as an example, for some users, it is required to deploy ML models on AI accelerators, or a button control logic on the microcontrollers serving as button signal receivers. These devices have neither own operating system nor an independent Internet connection. Furthermore, even though edge gateways host fully functional Linux distributions, their Internet connectivity might be hindered, i.e. when connected via WiFi, it is hidden behind the router and is not directly accessible by an IaC server at the developers' side. At the same time, since devices can leave and join the network and switch the connection method, the IP address is not easily predictable.

All these limitations result in the main challenge to be addressed to achieve proper fleet deployment—*how to provide a consistent way of deploying software components on*

[4] For example, hardware capacity or development environment. Please see Fig. 4 for more examples.

[5] https://aws.amazon.com/greengrass/.

[6] https://www.balena.io/cloud/.

[7] https://azure.microsoft.com/en-us/services/iot-edge/.

[8] It is worth noting that here we show only the edge devices (gateways), whereas in practice each of them may also be connected with a number of downstream sensor- or actuator-enabled IoT devices (e.g. blood pressure meters, alarm buttons, LCD screens).

[9] Please refer to Sect. 8 for more details.

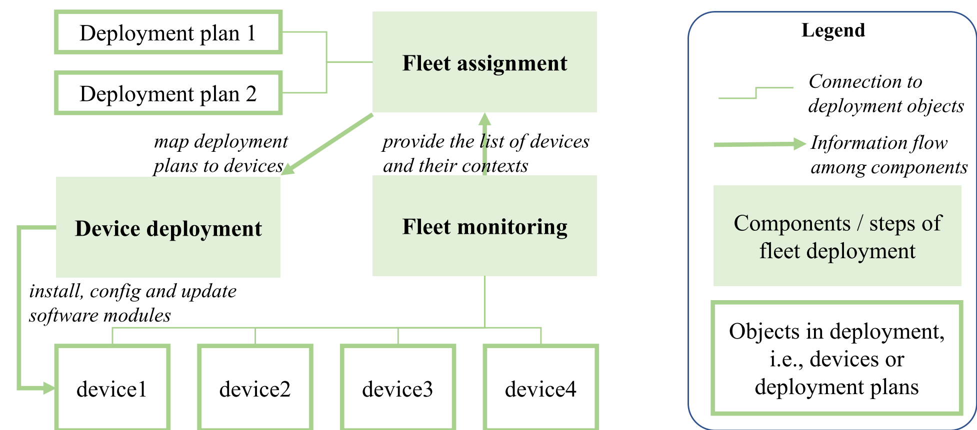


Fig. 1 Main components for managing a fleet of edge devices

across heterogeneous resources in the IoT–edge–cloud continuum?

2.4 Fleet assignment

In a global perspective, fleet assignment can be abstracted as a problem of finding the function $d : Dv \rightarrow Dp \cup \{\perp\}$ where Dv is the set of all devices and Dp is the set of active deployment plans. If no proper plan can be found for a certain device $dv \in Dv$, then $d(dv) = \perp$. This function d is the main output of fleet assignment, indicating what software should be deployed to each device. In addition, the assignment may also involve a set of functions for the configuration of deployment plans on each device: $c_x : Dv \rightarrow Vx$, which passes some additional parameters to each assigned deployment plan. For example, $c_{ie}(dv_1) = \top$ means the device dv_1 is configured in the way that a ML module learning is running on the gateway (where ie means *Intelligence on Edge*)

Fleet assignment is challenging, because it must satisfy the following requirements for all devices, at the same time:

R1. *Match hardware capacity* The software modules that are assigned to a device must match the particular device capabilities, including CPU architecture, memory volume, hardware accelerator, etc.

R2. *Fit software development pipeline* A deployment plan that is still under development can be only assigned to staging devices. At the same time, developers may want to deploy the preview version on a designated number of production-ready devices for direct feedback.

R3. *Align the resource usage* A deployment plan assigned to a device should have reasonable resource consumption, e.g. low CPU usage for a device powered by battery. Note that the same deployment plan may have different consumption on different devices, influenced by offloading, hardware acceleration, etc.

R4. *Achieve global objectives for deployment distribution* A typical objective is software diversity, i.e. even distribution of deployment plans among the devices for the sake of resilience, robustness and security.

3 Modelling language and engine for device deployment

This section presents our model-based approach for device deployment to address the fleet deployment problem. It provides a consistent way of automatic deployment across the IoT–edge–cloud continuum, covering different types of devices. The approach is called Generation and Deployment of Smart IoT Systems (GeneSIS), which includes a GeneSIS modelling language for software deployment and a deployment engine behind the language. Our main design principle behind the approach is to use the minimal set of concepts to model different types of deployment problems, while using extensible engines to support the actual deployment activities.

GeneSIS is a generic solution for the automatic deployment of IoT–edge–cloud applications. In this section, we first briefly summarize the main concepts of the language and the engine (more details can be found in our earlier publication [31]), with the help of the motivating RPM example, and then focus on the features specific to the device deployment problem in the fleet deployment context.

3.1 The GeneSIS modelling language for device deployment

The GeneSIS modelling language inspires from cloud modelling languages, which can be seen as refinements of the UML deployment language, which is generic and does not natively support cloud, edge or IoT deployment models [10] and typically needs to be refined to ultimately carry out the application provisioning [10]. More precisely, GeneSIS extends our former work on CloudMF [30] (both the language and its supporting execution engine) towards the edge and

IoT spaces, in particular enabling the deployment of software components on devices with no direct access to Internet. As a result, similar to most cloud modelling languages [9], GeneSIS follows a component-based approach and a deployment model can be regarded as a component assembly. GeneSIS is extensible, meaning that further sub-types can be added to the modelling language, and new plug-ins can be dynamically loaded into the engine.

Figure 2 shows the GeneSIS meta-model, which specifies the main concepts of the modelling language and the relations between the concepts. The complete description of the meta-model can be found in [31].

A DeploymentModel consists of GeneSISElements. They can all be associated with a list of properties in the form of key–value pairs. The two main types of GeneSISElements are Components and Links. A Component represents a reusable type of node that will compose a DeploymentModel. A Component can be a SoftwareComponent representing a piece of software to be deployed on a host (e.g. an Arduino sketch). A SoftwareComponent can be an InternalComponent meaning that it is managed by GeneSIS, or an ExternalComponent meaning that it is either managed by an external provider (e.g. an IoT middleware offered as a service) or hosted on a blackbox device (e.g. a Z-Wave transceiver). A SoftwareComponent can be associated with Resources (e.g. scripts, configuration files) adopted to manage its deployment life cycle (i.e. download, configure, install, start, and stop). An InfrastructureComponent provides hosting facilities (i.e. it provides an execution environment) to SoftwareComponents. The property needDeployer depicts that a local connection is required to deploy a SoftwareComponent on an InfrastructureComponent via a PhysicalPort (e.g. an Arduino board may only be accessible locally via serial port). This property is typically used for devices with no direct access to the Internet, which can only be reached indirectly via other devices, themselves configured by GeneSIS. Components are connected through two kinds of Ports: *(i)* a communication port represents a communication interface of a component—namely ProvidedCommunicationPort provides a feature to another component (e.g. MQTT on port 1883), while a RequiredCommunicationPort consumes a feature; *(ii)* an execution port represents the execution environment offered by a component to other components (i.e. ProvidedExecutionPort), or the required by a component (i.e. RequiredExecutionPort). As in most other cloud modelling languages [9,12,30], the rationale for using ports not only for communications but also for hostings lies in the need to specify the capabilities offered by a component and the requirement for such capabilities. Capabilities can be attached to both communication and execution ports, indicating that a component provides or requires a specific feature. They are used to validate that one component is fulfilling the requirements of another. For instance, to be deployed, a SoftwareComponent may require a specific ExecutionCapability from its host (e.g. a specific execution environment or a feature is required for the component to execute). By contrast, a provided Port may offer a Capability. For a deployment model to be valid, all the required capabilities must match a provided capability. There are two main types of Links: Hostings and Communications, connecting execution ports and communication ports, respectively. A Hosting depicts that an InternalComponent will execute on a specific host. This host can be any component, meaning that it is possible to describe the whole software stack required to run an InternalComponent. A Communication represents a communication binding between two SoftwareComponents.

Listing 1 shows an excerpt of the GeneSIS model for the RPM example. It describes the devices (infrastructure components) in a sub-system for one user, and the software (internal components) running on these devices. This part of the deployment model includes two infrastructure components, one for the gateway (main_gateway) and the other for the attached AI accelerator, such as a tensor processing unit (TPU—tpu). Both devices provide execution ports to host software components. The tpu device is the instance of an extended type named coral_usb_tpu, which has a dedicated extension to the deployment engine. One internal component, inference, represents the Docker container that collects data and executes an ML model for inference. The other component, infer_model, represents the ML model. Two links connect the internal components to the devices using ports. It is worth noting that in practice the ML model is stored on the gateway and flushed into the TPU at run-time by the inference component. Therefore, in our deployment model, the relation between infer_model and tpu is logical, but not physical. The deployment engine for the type coral_usb_tpu will be running on the gateway and will check if the tpu component is attached to the gateway, but will not physically write any code to tpu.

Fig. 2 Core concepts in the GeneSIS modelling language meta-model

Listing 1 An excerpt of the GeneSIS model for the eHealth example

```
{
  "_type": "deployment_model",
  "name": "eHealthSubSystem",
  "elements": [{
    "_type": "infrastructure/linux",
    "name": "main_gateway",
    "provided_exec_port": {"name": "ce61ee83"}
  },{
    "_type": "infrastructure/coral_usb_tpu",
    "name": "tpu",
    "attached": "ce61ee83",
    "provided_exec_port": {"name": "fac81d77"}
  },{
    "_type": "internal",
    "name": "inference",
    "resource":[{
      "_type": "docker_resource",
      "image": ...
    }],
    "required_exec_port": {"name": "70274e28"}
  },{
    "_type": "internal",
    "name": "infer_model",
    "resource":[{"_type": "ml_model_tpu", ↴
        "url":...}],
    "required_exec_port": {"name": "0e1c1ec7"}
  },{
    "_type": "hosting",
    "hosted": "70274e28", "by": "ce61ee83"
  },{
    "_type": "hosting",
    "hosted": "0e1c1ec7", "by": "fac81d77"
  }]
}
```

3.2 GeneSIS deployment engine

From a deployment model specified in GeneSIS language, the GeneSIS deployment engine is responsible for: *(i)* provisioning infrastructure resources when feasible, *(ii)* deploying the SoftwareComponents, and *(iii)* setting up communication between components.

The GeneSIS deployment engine implements the *Models@run.time* pattern for dynamic deployment with minimal impact on the running system. Models@run.time [13] leverages models as executable artefacts that can be applied to support the execution of the system. A run-time model provides abstract representations of the underlying running system, which facilitates reasoning, analysis, simulation, and adaptation. A change in the running system is automatically reflected in the model, while a modification to this model is enacted on the running system on demand. This causal connection enables the continuous evolution of the system with no strict boundaries between design- and run-time activities. Following the Models@run.time pattern, when a target model is provided to the GeneSIS engine, it is compared with the original model representing the running system. Then, the engine converts the difference between the two models into a sequence of deployment actions, such as provisioning resources and instantiating software components. Further details of these actions can be found in [31]. After

the deployment, the engine synchronizes the current GeneSIS model with the actual deployment result.

3.3 GeneSIS for device deployment

It is not always possible for the GeneSIS deployment engine to directly deploy software on all hosts. For instances, embedded devices and microcontrollers do not always have direct access to the Internet or even the necessary facilities for remote access (in such case, the access to the Internet is typically granted via a gateway). Also, in many cases for specific reasons (e.g. security), the deployment of software components can only be performed via a local connection (e.g. a physical connection via a serial port or Bluetooth). In such cases, the actual deploying of firmware on a device has to be delegated to the gateway, to which the device is locally connected.

To address this issue, GeneSIS relies on a *deployment agent*, which is generated dynamically by GeneSIS based on the artefact to be deployed and its target host. It is implemented as a Node-RED[10] application, with the following four types of nodes:

1. *Code generation nodes*, which generate the code or artefact for a target device from source code or specification languages. Code generation nodes consume as input a start compilation message to launch the compiling. Once the compilation is successfully completed, they send a generation success message that includes the location of the generated code. If the compile on start property is set to true, compilation will be triggered when the node is instantiated.
2. *Deployment configuration nodes*, which prepare the actual deployment of a software component. This typically involves in generating configuration files, such as a docker-compose file. Once this process is completed, it generates a message containing the location of configuration files together with the artefact to be deployed.
3. *Deployment nodes* which enact the deployment of a software component on a target resource. These nodes consume messages from the configuration nodes. When required, they are also in charge of removing the software they formerly deployed (e.g. killing a Docker container or deploying an empty firmware sketch).
4. *Communication nodes*, which, after deployment, are used to communicate *(i)* with the deployed software if required (e.g. sending an initialization message), and *(ii)* with the GeneSIS engine to inform about the status of the deployment.

[10] https://nodered.org/.

The GeneSIS engine generates an agent at run-time and deploys it to the target edge device (such as a gateway). The agent then acts as a broker, which generates the configurations and further deploys the software components on the resource- or network-constrained devices attached to the edge device via dedicated local communication channels such as USB cables and Bluetooth. The local communication channels are specific to the devices and hard-coded in the GeneSIS engines for the different types of devices. The GeneSIS engine itself can be deployed either on a dedicated operation machine or on the edge device. Developers can use the GeneSIS GUI to edit the deployment model and launch a deployment action. At the same time, third-party tools (such as continuous delivery pipelines) can invoke the REST APIs provided by the engine to send models and start deployment.

4 Model-based fleet assignment

We employ a model-based approach to automate the assignment part of the fleet deployment problem using metamodelling and model transformation to bridge the gap between the abstract constraint solving theory and the concrete edge computing platforms. Figure 3 illustrates the overall architecture of the approach.

From the platform, we obtain a PSM (platform-specific model) that represents the available devices and deployment plans. We transform the PSM into a PIM (platform-independent model), which represents the essential information, such as devices, deployment plans and their attributes, using a simple and standard notation, agnostic to the platform. This initial PIM, essentially representing the information from fleet monitoring, is not a complete model yet, since the mapping between devices and deployment plans is missing, together with some run-time configurations. We apply *constraint solving* on this PIM to automatically search for the missing part. The solution provided by the constraint solver is a new PIM′ with the missing part completed. We then transform PIM′ into a new PSM′, which will be finally used by the platform to install the software modules. Transforming back into PSM is necessary since it is the format that the fleet management platform understands. The concepts of PIM, PSM and model transformation comply with the classic definition in model-driven architecture [37].

The approach itself is generic across multiple edge computing platforms, i.e. for different platforms, we need to define different PSMs and the model transformation, while the PIMs and the constraint solving approach (which assigns deployment plans to devices, as further elaborated in Sect. 5) remain the same. As reported below, in the context of this paper, we use Azure IoT Edge as the target platform to implement a fleet deployment tool and demonstrate the viability of the proposed approach.

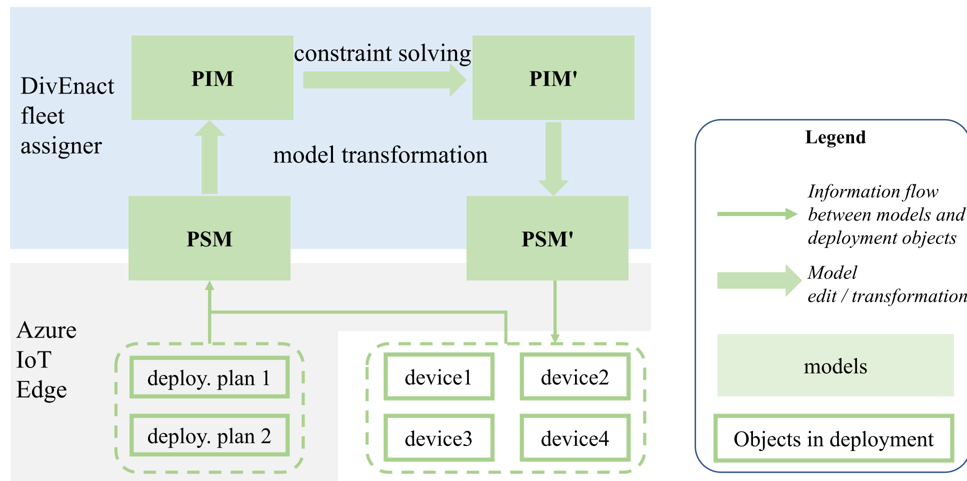


Fig. 3 Model-based fleet deployment: overall architecture

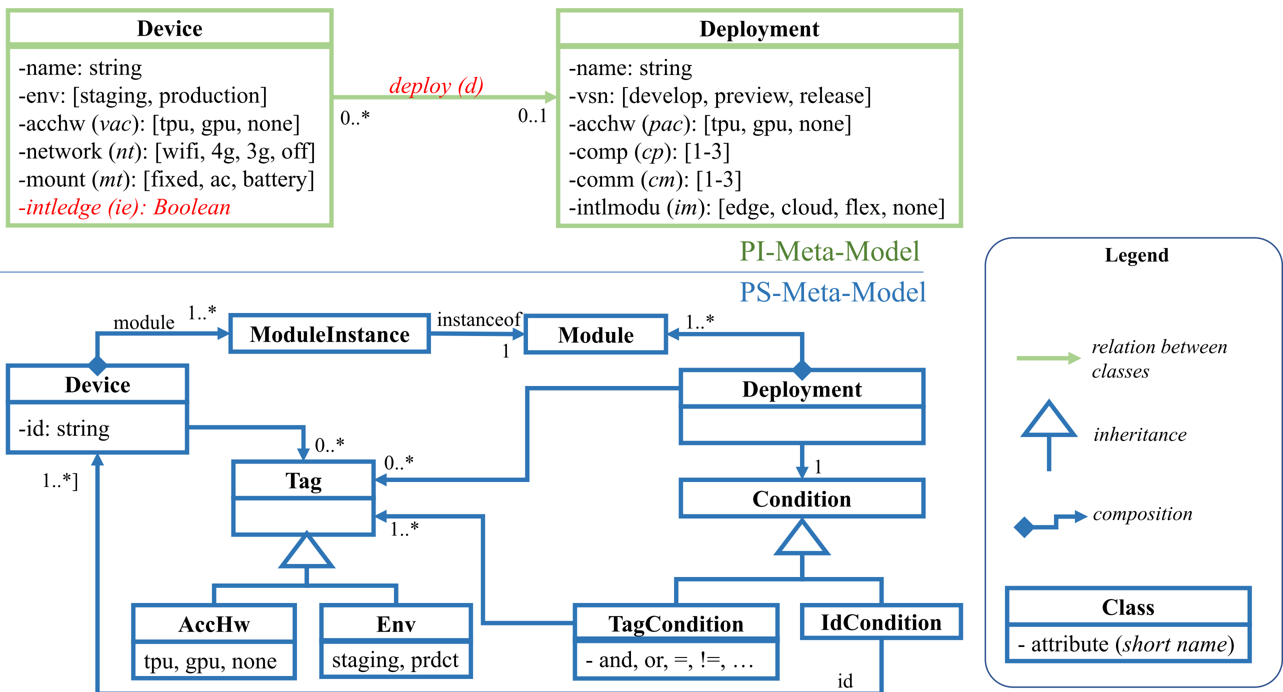


Fig. 4 The meta-models for the PIM (upper half) and PSM (lower half)

4.1 Meta-models

Figure 4 shows the meta-models we defined for the motivating example (Sect. 2.1). The platform-specific meta-model (PSMM) in the lower half of Fig. 4 reflects how the Azure IoT Edge platform manages the device fleet and deploys software modules. The platform maintains a set of Devices and a set of Deployments. Both devices and deployments are accompanied by a set of attached Tags. A tag can carry numeric, Boolean or enumerated values, which are either manually added by the application developers, or automatically collected by the platform from the environment at run-time. Each deployment contains a set of Modules. A Module is a software artefact, such as a Docker image. If a device is assigned with a deployment plan, then for each module in the plan, the platform will create one Docker container using the image. This container is represented as a ModuleInstance. The IoT Edge platform manages each device through an agent deployed on the device. The agents communicate with the hub in a 'pull' manner, i.e. reporting the context of the devices and obtain deployment commands. Therefore, in our approach, we do not need to know the IPs of any devices.

The mapping between devices and deployment plans are defined indirectly through the Conditions of deployment plans. Each plan contains a condition, defining to which devices it will apply. Azure IoT Edge supports two types of conditions. Developers can define a conditional expression using tags, which means all devices with a certain tag will be covered. Alternatively, developers can also directly refer to device IDs in the condition. In this paper, we use the ID-based conditions to represent the solving result.

The platform-independent meta-model (PIMM) in the upper half of Fig. 4) contains two simple classes, i.e. Device and Deployment. Each class contains a set of attributes, and there is one relationship, deploy, between them. What attributes are needed depends on the problem domain, and in this paper, we define them according to the RPM example. For a Device, env specifies whether it is used for staging or production; acchw specifies what hardware accelerator is used for the device; mount specifies whether the device is mounted to the wall, simply plugged to an A/C socket (installed by the patient themselves), or powered by batteries; and network specifies the network used primarily by the device. These attributes are used as input for fleet assignment. In addition, intlege tells whether the ML modules will be executed on the edge device, which is used to carry the fleet assignment result. On the deployment side, vsn specifies the development phase of the software behind this deployment; acchw specifies the type of hardware accelerator, against which the ML module was compiled; comp and comm estimate the levels of computation and communication resources required by the software; intlmodu defines where the ML model can be executed. In Fig. 4, some attribute names are followed by shortcuts, which will be used for defining constraints in the next section.

4.2 Transformations between PIM and PSM

The approach employs *forward* (from a PSM to a PIM) and *backward* (from a PIM to a PSM) transformations, main aspects of which are described below.

In the forward transformation, PSM elements of types Device and Deployment are mapped bijectively to homonymous PIM elements of devices and deployments, respectively. Tags on the PSM elements are transformed to attributes, where tag names are mapped to attribute names, and tag values to attribute values. If an element does not carry an optional tag, such as acchw, the attribute value will be assigned with a default value, such as none. Attribute intledge and reference deploy in PIM′ are left.

The backward transformation takes PIM′ and the original PSM as input and yields a new PSM′. For each Device in PIM′ (e.g. dv1), we find the homonymous Device in PSM, and propagate the values of intlege and deploy. For intlege, we create a new tag for the PSM device. For deploy, we find the corresponding deployment plan in the PSM, and append 'or id = dv1' to its IdCondition (or create a new IdCondition if there is none). The backward transformation takes the original PSM as an addition input and only complements it with the missing relations and attributes. This way, the round-trip transformation does not lose any information.

4.3 Sample models

Figure 5 illustrates sample PSM and PIM in the context of the considered example. According to the PIM, the system comprises six devices. The second device (dv2) has a TPU as an AI accelerator, connected by 4G and powered by A/C. Together with dv1, the two devices are used for staging. The other four devices are used in production. We have four deployment plans under maintenance: dp1 is still under development and supports TPU and flexible offloading. The worst-case computation and communication levels are both medium (2 out of 3). dp2 is a preview version, which hosts the ML module on the edge, and supports TPU. The worst-case computation is high, but communication requirement is low, since the analytics is on the edge. Attributes (intledge) and reference (deploy) in red do not exist in PIM, but will be added in PIM′. In the figure, we only show the partial assignment for the first two devices.

Due to the space limitation, we only show part of PSM, with one device and one deployment plan with attached attributes represented as tags. In PSM′ (the original elements in PSM as well as the additional ones marked in red in the same figure), one new tag is created, and the deploy reference is represented as IdCondition.

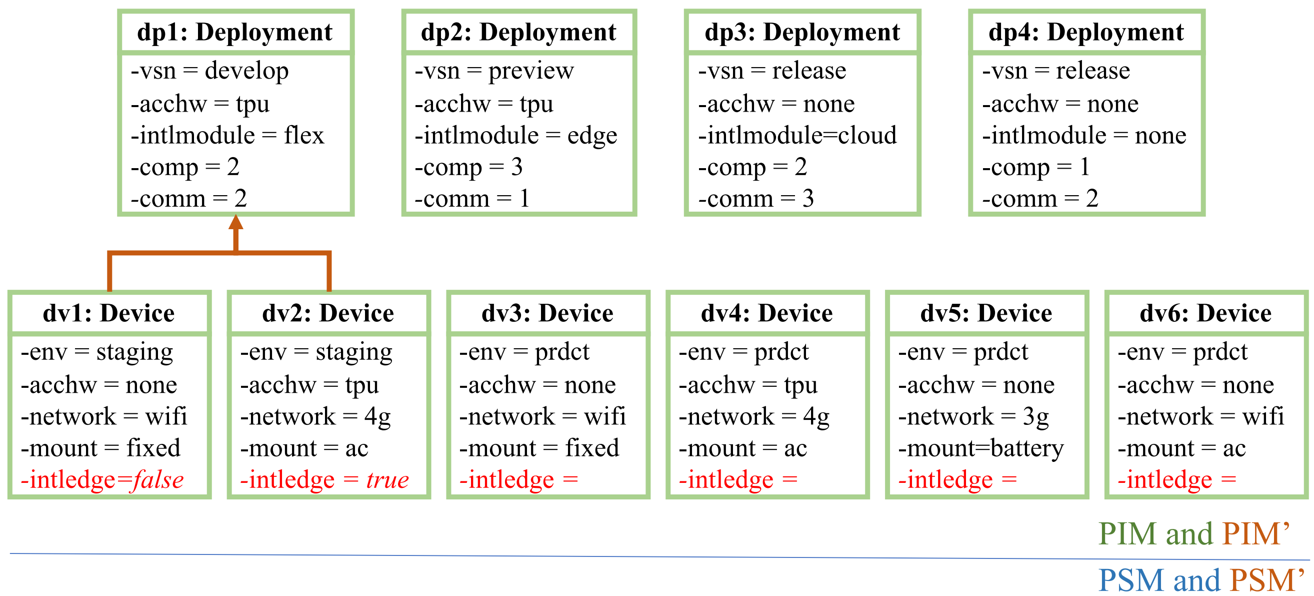
5 Fleet assignment using constraint solving

We employ SMT constraint solving and an existing implementation, Z3 solver [26], to achieve automatic fleet assignment. To do so, we transform the PIM into an SMT problem and define a set of hard and soft constraints. The solver will automatically search for the missing values in the PIM, according to the constraints.

5.1 Representing the PIM as an SMT problem

An SMT problem is a decision problem for logical formulas on a combination of background theories. It extends SAT (Boolean satisfiability theory) with a set of higher-level theories, such as first-order logic (FOL), set theory and algebra.

A basic built-in theory for SMT is the *uninterpreted function theory* [16]. An uninterpreted function is a function that is defined with only domain and co-domain, without an interpretation on how the domain is mapped to the co-domain. The building blocks of an SMT problem are a series of such unin-



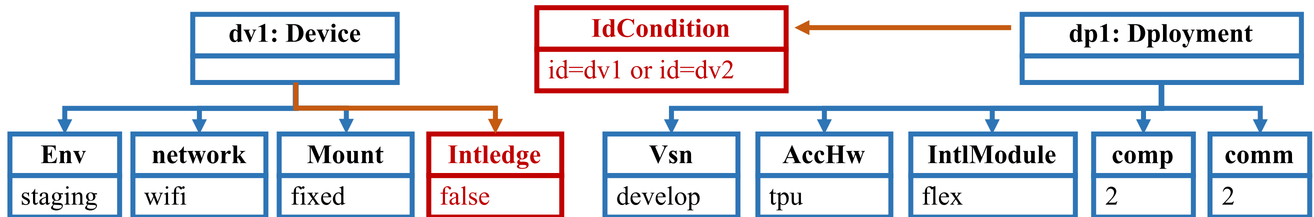


Fig. 5 Sample PIM and PSM for the motivating example

terpreted functions, on which we can define FOL formulas as constraints.

We transform a PIM into an uninterpreted function theory problem based on the following rules, explained using the sample PIM in Fig. 5.

- All devices form a set $Dv = \{dv_1, ..., dv_6\}$, and all deployment plans form a set $Dp = \{dp_1, ..., dp_4\}$.
- The deploy relation is represented by a function $d : Dv \rightarrow Dp \cup \{\perp\}$, as discussed in Sect. 2.
- Attributes in primitive types are represented as functions to the corresponding types, e.g. $cp : Dp \rightarrow \mathbb{N}$ and $ie : Dv \rightarrow \mathbb{B}$, for comp and intledge, respectively.
- The attributes whose values are within a particular set, are represented by functions with co-domains as the set of all the valid choices. For example, the version attribute is a function $vsn : Dp \rightarrow \{\text{develop, preview, release}\}$.

A constraint solver can automatically search for an interpretation for each of the uninterpreted functions that we defined above. The interpretation of function d (i.e. giving a dp for every $dv \in Dv$) and the interpretation of ie (assigning either true or false for every $dv \in Dv$) are the result for fleet assignment. The searching process will be guided by a set of constraints from (i) the current model status, i.e. the existing attribute values, (ii) the domain-specific knowledge, and (iii) global optimization objectives. In the rest of this section, we will present the three types of constraints, respectively.

5.2 Constraints representing the current model state

In the current PIM, the devices and deployment plans already have existing attribute values, which serve as the basis for fleet assignment. Therefore, we translate all the current attribute values into hard constraints. The example below shows how we generate the constraints from the current state of dp1:

$$vsn(\text{dp1}) = \text{develop} \ \wedge \ pac(\text{dp1}) = \text{tpu}$$
$$\wedge ie(\text{dp1}) = \text{flex} \ \wedge \ cp(\text{dp1}) = 2 \ \wedge \ cm(\text{dp1}) = 2 \quad (1)$$

The shortened function names are defined in the meta-model in Fig. 4. We generate the constraints automatically by iterating over all the devices and deployment plans in the PIM, reading the attribute values, and generating the equations accordingly.

5.3 Domain-specific constraints

We define a set of hard constraints based on the domain knowledge about software deployment on edge devices. These constraints are related to the requirements **R1**–**R3** in Sect. 2. To simplify the constraint definition, we first introduce 3 auxiliary functions.

$$acc : Dv \rightarrow \mathbb{B};\; vcp : Dv \rightarrow \mathbb{N};\; vcm : Dv \rightarrow \mathbb{N} \quad (2)$$

Here, acc indicates whether the hardware accelerator will be used on a device, and vcp and vcm are the actual levels of computation and communication resource consumption on the device, which is determined not only by the levels defined in the deployment plan, but also by the configuration of the device, such as whether the ML module is executed on the edge device or in the cloud (ie) and whether the accelerator is used (acc).

We start from the constraint related to the development stage (**R2**): any deployment with software under development should be only deployed on a staging device:

$$\begin{aligned} &\forall dv \in Dv, dp \in Dp : d(dv){=}dp \;\wedge\; vsn(dp){=}\mathsf{develop} \\ &\quad\Longrightarrow\; env(dv) = \mathsf{staging} label eq : 5 \end{aligned} \quad (3)$$

The next set of constraints requires that the resource consumption of a deployment plan should match the capacities of its target device (**R1** and **R3**).

The first step is to define how the actual consumption of each device is determined by the theoretically worst-case consumption defined by the developers.

$$\begin{aligned} &\forall dv \in Dv, dp \in Dp : d(dv) = dp \;\Longrightarrow \\ &\quad vcm(dv) = \begin{cases} cm(dp) - 1 & im(dp) = \text{flex} \wedge ie(dv) \\ cm(dp) & \text{otherwise} \end{cases} \\ &\quad vcp(dv) = \begin{cases} cp(dp) - 1 & im(dp) = \text{flex} \wedge \neg ie(dv) \\ cp(dp) - 1 & acc(dv) \\ cp(dp) & \text{otherwise} \end{cases} \end{aligned} \quad (4)$$

For communication, if a deployment plan allows flexible offloading of the ML model, then running the model on the edge device would reduce the requirement on communication, since the raw data will be consumed locally without being transferred to the cloud. For computation, there are two cases that the actual computation requirement on the gateway will be reduced: *(i)* When offloading is supported, and the model is not executed on the edge, the computation requirement will be reduced, since the computation task is offloaded to the cloud; *(ii)* When hardware accelerator is used, the computation requirement is reduced, since the task is offloaded to the accelerator.

The second step is to regulate when we can offload the ML module to the cloud or the hardware accelerator.

$$\begin{aligned} &acc(dv) \Longleftrightarrow ie(dv) \wedge vac(dv) = pac(dp) \neq \text{none} \\ &ie(dv) \;\Longrightarrow\; im(dp) \in \{\text{flex}, \text{edge}\} \\ &ie(dv) \Longleftarrow im(dp) = \text{edge} \end{aligned} \quad (5)$$

Offloading to the accelerator happens (i.e. $acc(dv) = \top$), *if and only if* the ML module is allocated to the device (i.e. $ie(dv)$) and the type of accelerator supported by the deployment plan match the accelerator attached to the device. The equivalence statement means that whenever it is possible, we will opt for acceleration, which is the most reasonable for our use case. The ML module is executed on the edge (i.e. $ie(dv) = \top$), only if in the deployment plan the ML module is defined as flexible or running on the edge. On the other hand, if in the deployment plan the ML module is defined to be only running on the edge, then $ie(dv)$ must be on. It is worth noting that when $im(dp) = \mathsf{flex}$, $ie(dv)$ can be either true or false, which is a decision to be made by fleet assignment.

The last set of constraints describes how the device contexts determine what level of resource assumption is allowed.

$$\begin{aligned} &\forall dv \in Dv : \\ &nw(dv) = \mathsf{4g} \;\Longrightarrow\; vcm(dv) < 3 \\ &nw(dv) = \mathsf{3g} \;\Longrightarrow\; vcm(dv) < 2 \\ &mt(dv) = \mathsf{ac} \;\Longrightarrow\; vcp(dv) < 3 \\ &mt(dv) = \mathsf{battery} \;\Longrightarrow\; cvp(dv) = vcm(dv) = 1 \end{aligned} \quad (6)$$

The rationale behind these constraints in the sample domain is threefold. First, the worse network a device has, the lower communication level it can support. Second, if the device is not mounted professionally, it should not run software modules with the highest computation requirement. Finally, if a device is powered by a battery, it should opt for the lowest computation and communication consumption.

5.4 Soft constraints to optimize the distribution

In addition to the hard constraints that must be satisfied by the assignment, we also define a set of soft constraints to guide the assignment towards optimized global distribution (**R2** and **R4**). The solver can choose to violate a soft constraint with an agreed penalty (i.e. the *weight* of the constraint) aiming at the minimal total penalty for the solving result. In our case, we used the following three groups of soft constraints.

The first group of software constraints suggests each device to be assigned with a deployment plan. For each device, we generate a soft constraint for its deployment plan to be not empty with a penalty of 50. So far, the penalties

we used for the soft constraints are based on our subjective judgement of the importance of different constraints. More thorough design and evaluation of the penalty framework is seen as one of the future work directions.

$$(d(dv_1) \neq \bot_{\text{penalty}=50}, \ \ldots, \ d(dv_6) \neq \bot)_{\text{penalty}=50} \quad (7)$$

The second group consists of only one soft constraint, which instructs the solver to select 20% of all production devices to trial the preview deployment plans (see **R2** in Sect. 2). The left side of the equation counts all the devices assigned with a preview deployment plan, while the right side counts all the devices in production and multiplies it by 0.2.

$$\begin{aligned}(&\left|\{dv \in Dv | vsn(d(dv)) = \text{preview}\}\right| \\ &= \lceil 0.2 \cdot \left|\{dv \in Dv | env(dv) = \text{production}\}\right| \rceil)_{\text{penalty}=100}\end{aligned} \quad (8)$$

The third group of soft constraints aims to evenly distribute the deployment plans in order to maintain the diversity of deployment plans in the whole group (**R4**). In the two constraints defined below, we count the number of devices assigned with each deployment plan, and suggest the number to be close to the average number of devices per deployment plan (i.e. within the scope of ±20% of the average number).

$$\begin{aligned}(&\left|\{dv \in Dv | d(dv) = dp_1\}\right| \quad > 0.8 \cdot |Dv|/|Dp|)_{\text{penalty}=20} \\ (&\left|\{dv \in Dv | d(dv) = dp_1\}\right| \quad < 1.2 \cdot |Dv|/|Dp|)_{\text{penalty}=20}\end{aligned} \quad (9)$$

This is a relatively low-cost way to achieve even distribution, compared to other complex methods, such as minimizing the total variance (i.e. min $\sum(\tau_i - \overline{\tau})^2/n$, where τ_i is number of devices for dp_i).

5.5 Implementation using the Z3 constraint solver

We use the Z3 SMT solver [26] to automatically search for a solution for the uninterpreted functions under the constraints. We define the SMT problem in Z3Py (the Python interface of Z3), feed the defined constraints into the solver as input, and extract the results.

Listing 2 demonstrates a simplified example to illustrate the whole process. We first define an enumerated type Dp to represent the set of all deployment plans (Line 1), followed by an uninterpreted function d for deploy (Line 3). The definition for Dv is similar, and is therefore omitted in this example. Function env links Device to another enumerated type representing the environment choices (Lines 4–5), and cp is a function from deployment plans to integer. After defining the functions, we instantiate a solver (Line 7), and add constraints to it. We give two examples to show how to program the constraints that we presented earlier. The first one is a hard constraint indicating that a ‘develop’ deployment plan can only be deployed on a ‘staging’ device (Lines 8–10). The second example is a group of soft constraints. For each deployment plan, we generate a constraint stating that the total number of devices assigned with it should not be more than 120% of the average number (Lines 11–13). After populating the solver with all constraints, we launch the solving procedure (Line 14), and then, as an example, check for a deployment plan assigned to device dv4.

Listing 2 Sample SMT problem in Z3Py.

```
1  Dp, dps =
2    EnumSort('Dp', ['dp1', 'dp2', 'dp3', 'dp4', 'nodp'])
3  d = Function('deploy', Dv, Dp)
4  Env, [staging, prod] = EnumSort('Env', ['staging', 'prod'])
5  env = Function('env', Dv, Env)
6  cp = Function('comp', Dp, IntSort())
7  solver = Optimize()
8  solver.add(ForAll([dv, dp], Implies(
9    And(d(dv) == dp, vsn(dp) == develop),
10   env(dv) == staging)))
11 for i in dvs:
12   solver.add_soft(Sum([If(d(j)==i, 1, 0) for j in dps])
13     < len(dv)*1.2/len(dp), 20)
14 solver.check()
15 assert dp2 == solver.model().eval(d(dv4))
```

6 Application and evaluation of fleet assignment

This section puts theory into practice by presenting how the fleet assignment approach was applied to the RPM use case of Tellu and demonstrating its effectiveness in their DevOps routine. The section evaluates the fleet deployment concept and approaches guided by the following research questions:

- **Q1** Is it feasible to apply the approach to real DevOps teams working on IoT applications?
- **Q2** Is the approach capable of generating deployment assignments that are useful for the DevOps process?
- **Q3** Is the approach beneficial to the DevOps team?
- **Q4** Is the approach scalable to real-life IoT applications?

We answer these research questions by integrating the solution into a prototype fleet management tool, applying it to the Tellu DevOps practices, and observing how the tool works in a scenario simplified from the daily DevOps routine at Tellu.

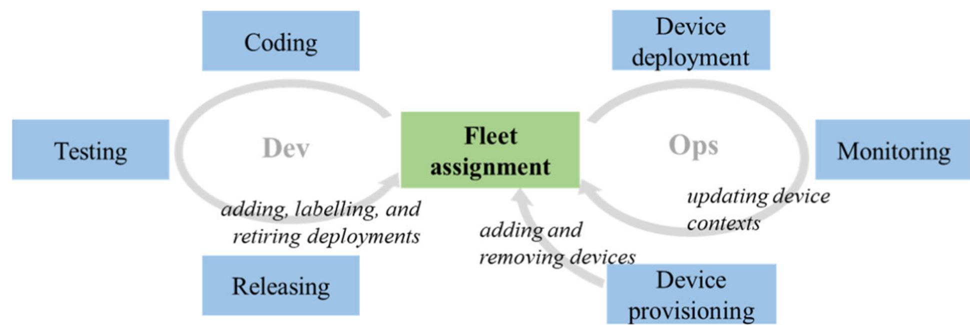


Fig. 6 The DevOps process enhanced by automatic fleet assignment. The three arrows pointing to the Fleet Assignment component indicate the three different situations within a DevOps life cycle which trigger fleet assignment

6.1 Implementation and integration

To address the question **Q1**, we implemented the proposed approach as a prototype tool named DivEnact,[11] which was integrated into the current DevOps toolset of Tellu.

Figure 6 illustrates Tellu's new DevOps pipeline. Except for the fleet assignment tool (depicted as a green box), all the other steps and associated tools are the same as the currently used in production. At present, the team is using Ansible for device deployment and Azure IoT Edge for fleet monitoring. The process is driven by the Jenkins continuous integration engine. In each *Dev* loop, after coding and testing the new changes, Tellu releases a new version of their application, and creates a number of new deployment plans which contain the new version. These deployment plans vary in terms of configurations and dependencies on other components, libraries and/or services. They label the new deployment plans with different tags, occasionally update the tags of existing deployment plans, and retire some old deployment plans. With the current tools, after each release, they need to manually update the mapping between devices and deployment plans in the form of several Ansible Inventory files, each of which records a list of IP addresses for the devices targeted by a deployment plan (which in turn is in the form of an Ansible Playbook).

The fleet assignment tool replaces this manual task. After each release, the tool will be automatically triggered, and will provide a mapping between the current devices and the active deployment plans (i.e. Ansible inventory files). The device deployment tool receives the new assignment and enforces redeployment of each individual device at a predefined timeslot (e.g. the nearest available midnight when the device is online). During run-time, the monitoring system will keep track of the device context and update the list of devices maintained by the Fleet Assignment tool. The device provisioning tool continuously adds new devices into the device list, e.g. after new patients join the service or existing patients switching to new devices. Such changes will also trigger the fleet assignment action.

The fleet assignment tool itself is implemented as a web service. The main backend service is composed of the three modules for PSM, PIM and the transformation between them, as described in Sect. 4. The PSM module invokes the external fleet management and device deployment tools through REST API, while the PIM module invokes the constraint solving logic, as presented in Sect. 5, implemented as a single Python script SMT.py. The back-end service also exposes its own REST API for other tools to check the status of deployment plans and devices, edit the labels, add additional constraints and trigger the assignment process. More details about the tool can be found in our earlier publication [23], which describes the general fleet management support without the assignment approach based on constraint solving.

6.2 Application scenarios

We evaluate the effectiveness of the approach by applying it to a DevOps scenario that simulates the development activity at Tellu. This scenario answers the research questions **Q2** and **Q3** by demonstrating that assignment generated automatically from the constraints fulfill the developers' expectation during the DevOps cycle.

The scenario involves a series of DevOps iterations, each of which yields a new version of the application, changes the status of the existing versions and/or retires some old versions. Through these iterations, the developers gradually create, test and release 6 versions of the RPM application, in order to introduce and improve the new fall detection feature with a ML module. Table 1 summarizes these versions and their resource consumption. Each version corresponds to one deployment plan.

Table 2 summarizes the automatic assignment result after each iteration. The first column lists the set of deployment plans maintained by the developers after each iteration. We use the vertical bar sign | to divide the deployment plans into three groups, according to the different development phases (or *vsn* as we named it in the model). For example, AB|C|D means deployment A and B are released to production, C is

[11] Diversity-oriented fleet deployment enactor. All source code available in the open-source repository: https://github.com/SINTEF-9012/divenact.

Table 1 Application variants after each DevOps iteration

Dep.	Summary of features
A	Base version, no ML (comm:1, comp: 1)
B	ML running on cloud (comm: 3, comp: 1)
C	ML running on gateways (comm: 1, comp: 3)
D	Flexible ML offloading (comm: 3, comp: 3)
E	Improve C by supporting accelerator (comm: 1, comp: 3)
F	Improve D, offloading + accelerator (comm: 3, comp: 3)
G	Improve E with lighter model (comm: 1, comp: 2)

under preview and D is under development. As iterations go on, more deployment plans are released and go into preview or deployment phases.

The rest of the table schematically illustrates how the proposed tool automatically assigns the deployment plans after each iteration to a fleet of 25 devices, including 4 staging and 21 production devices. The devices have different contexts in terms of physical mounting, networking, and accelerator, based on sampling with the actual devices maintained by the company. We omit the actual profile of each device, since the important part is to evaluate how the approach provides valid assignments as a whole, rather than to check if each single assignment satisfies all the constraints.

We go through the assignment results during the entire scenario, step by step, to show how the automatic fleet deployment benefits its users, i.e. the developers of edge applications. For each step, we give a brief description of the development purpose, followed by how the automatic assignment results support this purpose.

1. *Baseline deployment* In the beginning, A is the only valid deployment and has the lowest resource requirements, and therefore all devices are assigned with this deployment.
2. *First attempt with ML* Developers first implement B with the ML module running in the cloud, which requires devices with WiFi due to the high communication consumption. Two staging devices (1 and 3) were selected after iteration 1 to test B internally, but afterwards only 3 devices were picked for preview in #2 (expected to be 20%, or 4 devices) and the same 3 devices after B is released (#3), since WiFi is not widely used by patients, since most of them would skip configuring WiFi and keep using the default 4G network. The developers get the feedback that running ML only in the cloud has limited usage among their users.
3. *Migrating ML to the edge* Deployment C with ML running on the edge (tested, previewed and released through steps #2–#4) is complementary to B. It requires fixed mounted devices but not necessary WiFi. Comparing steps #2 and #3, C is deployed to devices that B was not assigned to, which is in line with the developers' expectations.
4. *Support cloud–edge offloading* The next version D allows flexible offloading. Due to the networking overheads, it has some extra cost,[12] and is therefore not a complete substitution to B and C. The developers choose to keep all the three deployment plans. They would expect an even distribution of A-D among the production devices at #4, but the assignment result has too many A's (10 devices in total), which indicates that there are still many devices that cannot run B, C or D.
5. *Utilize hardware accelerators* Deployment plans E and F add accelerator support to C and D, respectively, by adding an ML module compiled for an edge TPU. E and F still keep the original ML module, so that they can also run on devices that do not have a TPU (in that case, the computation consumption would not be lowered), This means that E can completely replace C. Therefore, when releasing F (step #6), the developers also retire C. The new assignment result does not increase the number of A's, which makes the developers hypothesize that the remaining 10 devices with A could not run any of the current deployment plans with ML.
6. *Try to retire the base version* To confirm the hypothesis, the developers attempts to retire A (#7), and, as expected, 8 production devices end up with no deployment plans, and so is one staging device (Device 4). They go on to retire B, based on an observation back at step #5 that when the assignment tool tried to recruit a sufficient number of devices to preview E, it removed B completely, which means B is replaceable by the subsequent versions. The result at step #8 also confirms this, as the number of un-deployed devices is the same. As a trial, the developers run assignment without executing the subsequent device deployment at steps #7 and #8, to avoid production devices having no deployment plans.
7. *Optimize ML module* By examining the profile of un-deployed devices, the developers realize that the computation power is the key problem, and therefore they simplify the ML module and produce G with lower computation requirement. By releasing G (at step #9, we skip the iterations for testing and previewing G, for the sake of space limitation), only four devices remained un-deployed. This can be further solved by sending accelerators to these devices, so that eventually all the devices can be equipped with the new ML feature.

In summary, the scenario demonstrates the following benefits for developers, which answers the research question **Q3**.

[12] For example, if ML is offloaded to the cloud, the computation cost is down to 2 compared to 1 in B.

Table 2 Assignment in demonstration scenarios

<table>
<tr><th>#</th><th>Deployments</th><th colspan="4">Staging devices</th><th colspan="21">Production devices</th></tr>
<tr><th></th><th></th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th><th>14</th><th>15</th><th>16</th><th>17</th><th>18</th><th>19</th><th>20</th><th>21</th><th>22</th><th>23</th><th>24</th><th>25</th></tr>
<tr><td>0</td><td>A| |</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td></tr>
<tr><td>1</td><td>A| | B</td><td>B</td><td>A</td><td>B</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td></tr>
<tr><td>2</td><td>A| B| C</td><td>C</td><td>A</td><td>B</td><td>A</td><td>A</td><td>B</td><td>A</td><td>A</td><td>A</td><td>B</td><td>A</td><td>B</td><td>A</td><td>A</td><td>B</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td></tr>
<tr><td>3</td><td>AB| C| D</td><td>D</td><td>A</td><td>B</td><td>A</td><td>A</td><td>B</td><td>A</td><td>C</td><td>C</td><td>B</td><td>A</td><td>B</td><td>C</td><td>A</td><td>B</td><td>A</td><td>C</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>C</td><td>A</td></tr>
<tr><td>4</td><td>ABC| D| E</td><td>C</td><td>E</td><td>B</td><td>A</td><td>D</td><td>D</td><td>A</td><td>D</td><td>D</td><td>B</td><td>A</td><td>B</td><td>D</td><td>c</td><td>B</td><td>A</td><td>c</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>C</td><td>A</td></tr>
<tr><td>5</td><td>ABCD | E| F</td><td>F</td><td>E</td><td>D</td><td>A</td><td>D</td><td>C</td><td>A</td><td>E</td><td>D</td><td>A</td><td>A</td><td>D</td><td>E</td><td>c</td><td>E</td><td>A</td><td>c</td><td>E</td><td>E</td><td>A</td><td>A</td><td>A</td><td>A</td><td>C</td><td>A</td></tr>
<tr><td>6</td><td>ABDEF||</td><td>D</td><td>E</td><td>B</td><td>A</td><td>D</td><td>B</td><td>A</td><td>D</td><td>F</td><td>B</td><td>A</td><td>F</td><td>F</td><td>E</td><td>B</td><td>A</td><td>E</td><td>E</td><td>F</td><td>A</td><td>A</td><td>A</td><td>A</td><td>D</td><td>A</td></tr>
<tr><td>7</td><td>BDEF | |</td><td>B</td><td>F</td><td>E</td><td>–</td><td>D</td><td>E</td><td>–</td><td>D</td><td>D</td><td>F</td><td>–</td><td>F</td><td>D</td><td>E</td><td>D</td><td>–</td><td>E</td><td>E</td><td>F</td><td>–</td><td>–</td><td>–</td><td>–</td><td>F</td><td>–</td></tr>
<tr><td>8</td><td>DEF | |</td><td>D</td><td>F</td><td>E</td><td>–</td><td>F</td><td>E</td><td>–</td><td>E</td><td>E</td><td>F</td><td>–</td><td>E</td><td>F</td><td>E</td><td>F</td><td>–</td><td>E</td><td>E</td><td>F</td><td>–</td><td>–</td><td>–</td><td>–</td><td>F</td><td>–</td></tr>
<tr><td>9</td><td>DEFG| |</td><td>D</td><td>E</td><td>F</td><td>–</td><td>D</td><td>F</td><td>G</td><td>D</td><td>D</td><td>F</td><td>G</td><td>D</td><td>E</td><td>E</td><td>D</td><td>–</td><td>E</td><td>E</td><td>F</td><td>–</td><td>G</td><td>G</td><td>G</td><td>F</td><td>–</td></tr>
</table>

- *Automation* After each iteration, developers do not need to manually select matching devices for preview and release.
- *Handling complexity* Maintaining multiple deployment plans in parallel often needs to reassign already existing plans to make space for new ones. Taking the transition from #4 to #5 as a simple example, the assigner has to change Device 12 from B to D, in order to release Device 13 to preview E. With more devices involved, such rearrangement would become too complex and challenging for manual work.
- *Testing* Running fleet assignment without the device deployment step is an efficient way to test the composition of deployments, as is shown by the attempt to retire A and B.
- *Feedback* Since the automatic assignment will guarantee that the number of devices assigned to different devices will be balanced, if a particular deployment plan is not assigned, no devices can host this deployment plan. This is valuable feedback to the developers to examine the problematic deployment plans.

6.3 Performance and scalability

As a first step, it was required to evaluate the performance of the approach to check whether it can be further used by Tellu in the current set-up. We conducted a series of experiments to test this, by selecting 5 out of the 10 compositions of deployment plans in Table 2, and trying to assign them to a fleet of varying size (from 25 to 400 devices) with generated profiles. We conducted the experiments on a MacBook Pro laptop with 3.1 GHz Intel Core i5 processor and 16GB Memory. This experiment addresses the research question **Q4**. It demonstrates that the approach is usable in the context of small- and medium-sized IoT application systems, and also reveals that scalability is still an important future work direction. Noteworthy, the experiments only reveal the time for fleet assignment. The time used for actual deployment may vary significantly depending on the size of deployment plans, available devices and connectivity.

Table 3 Time spent (in seconds) on assigning deployment plans

<table>
<tr><th>#</th><th>Dep.</th><th colspan="5">Number of devices</th></tr>
<tr><th></th><th></th><th>25</th><th>50</th><th>100</th><th>200</th><th>400</th></tr>
<tr><td>2</td><td>A|B|C</td><td>0.32</td><td>0.61</td><td>2.53</td><td>10.69</td><td>59.52</td></tr>
<tr><td>4</td><td>ABC|D|E</td><td>0.33</td><td>0.94</td><td>4.62</td><td>31.17</td><td>121.31</td></tr>
<tr><td>5</td><td>ABCD|E|F</td><td>0.77</td><td>1.91</td><td>8.44</td><td>92.52</td><td>606.63</td></tr>
<tr><td>6</td><td>ABDEF||</td><td>0.19</td><td>1.28</td><td>4.79</td><td>16.60</td><td>101.55</td></tr>
<tr><td>9</td><td>DEFG||</td><td>0.16</td><td>0.78</td><td>3.14</td><td>15.22</td><td>170.50</td></tr>
</table>

Table 3 lists the average time spent on fleet assignment after each iteration applied to different numbers of devices in a fleet. For each iteration, we start with the same 25 devices as used in the last section, run assignment for 10 times and record the average time in seconds. After that, we double the number of devices and repeat it until reaching the total number of 400 devices, which is close to the actual number of devices managed by Tellu. The time for assignment increases significantly as the number of devices or deployment plans increases. The composition of deployment plans also has an impact: Comparing #4 and #6, the one with simpler composition is marginally faster to assign. The experiments confirm the feasibility of the current approach: assigning 6 deployment plans to 400 devices in 10 minutes, albeit not perfect, is still acceptable for developers, since other automated DevOps steps, such as building, testing and device deployment takes comparable time, i.e. ranging from several minutes up to an hour.

The experiments also show the limitation in terms of scalability. Considering 1 hour as the upper limit, when the application system scales out to even more devices, we would need to divide the whole fleet into multiple sub-fleets and conduct the assignment separately. We also foresee that developers may add new attributes to the meta-model and new constraints, making it somewhat more computationally complex. In such cases, we may need to carefully design the constraints, limit the size of sub-fleets, etc. Further experiments and guidelines for performance optimization is part of the future work.

7 Model-based fleet deployment bundle for smart IoT systems

This section introduces a complete fleet deployment bundle that comprise the tools that implement the two model-based approaches for device deployment and fleet assignment, presented in Sects. 3, 4 and 5. We further extend the use case based on Tellu's production scenario to demonstrate how the fleet deployment concept and the tool bundle are used together to support the DevOps practice of applications in the IoT–edge–cloud continuum.

The implementation and application of the full fleet deployment bundle answers the following two additional research questions.

Q5. Can the global deployment assignment lead to final and concrete deployment on all individual gateways?

Q6. Is the complete fleet deployment approach beneficial to multiple stakeholders, not only the DevOps team of the IoT application?

7.1 Fleet deployment bundle: the concepts

We implemented the model-based automatic deployment approach as a prototype deployment bundle comprising two tools, i.e. GeneSIS and DivEnact, supporting automatic deployment at the level of individual devices and the fleet as a whole, respectively. The twofold bundle further demonstrates the feasibility of the fleet deployment approach, addressing the question **Q5**.

Figure 7 illustrates how the deployment bundle can be used in a typical smart IoT system. The illustrative system has several sub-systems, each of which is in charge of a particular business task, e.g. serving contents to the user, monitoring a room, etc. Such a sub-system is usually composed of at least one edge gateway and several connected IoT devices such as sensors and actuators.[13] These sub-systems form the fleet of this system, and because each sub-system includes an edge device as the main contact point, i.e. a gateway with the back-end service, we also refer to such fleet as an *edge fleet*. A fleet is normally distributed, with edge gateways (together with its associated IoT devices) serving different customers or tenants, and deployed in different physical locations. Also, the developers often maintain one or several edge devices on their own premises for testing or trial purposes.

GeneSIS supports automatic deployment within a local sub-system, e.g. the deployment on the devices located on the developers' side. In such case, the developers can directly interact with the GeneSIS engine hosted on the local edge device, and use it as the bridge to further deploy required code to the associated IoT devices. At the development phase, developers define a GeneSIS deployment model, specifying which software artefacts should be deployed on which devices. Next, at the deployment phase, the same deployment model will be provided to the GeneSIS deployment engine, running either on a local machine or on the edge device. The engine will install or update the software artefacts according to the newly received deployment model.

The situation gets more complicated and challenging when the developers want to release a new version of their application to production. In such case, they need to deploy software artefacts to all the devices on the users' sites. They cannot extend the deployment model to include every device in the fleet, because such a huge model is not maintainable, especially when the devices keep joining and exiting the fleet. DivEnact addresses this automatic deployment challenge at the fleet level. Since each user has a sub-system similar to the one at the developers' side, instead of handling every single device individually, the developers can provide the deployment models they developed at the previous phase to DivEnact. It maintains the list of all sub-systems and sends the deployment model to the devices before invoking the GeneSIS engine running on the edge gateway of the sub-system, to eventually deploy the software artefacts according to the deployment model. As already discussed, within a fleet, the sub-systems have different contexts, such as device capabilities, connectivity and user preferences and the developers need multiple variants of their software to fit different contexts. DivEnact accepts multiple deployment models representing different software variants and configurations, generated as a series of releases, and automatically assigns them to the target sub-systems.

7.2 Instantiating the fleet deployment bundle in practice

Figure 8 shows how to instantiate the deployment bundle on a fleet of IoT/Edge devices step by step.

To manage all the devices as a fleet, developers need to first create an Azure IoT Hub, which provides a cloud service to register devices and to manage their life cycles. Creating

[13] For the sake of simplicity, we do not show all the IoT devices.

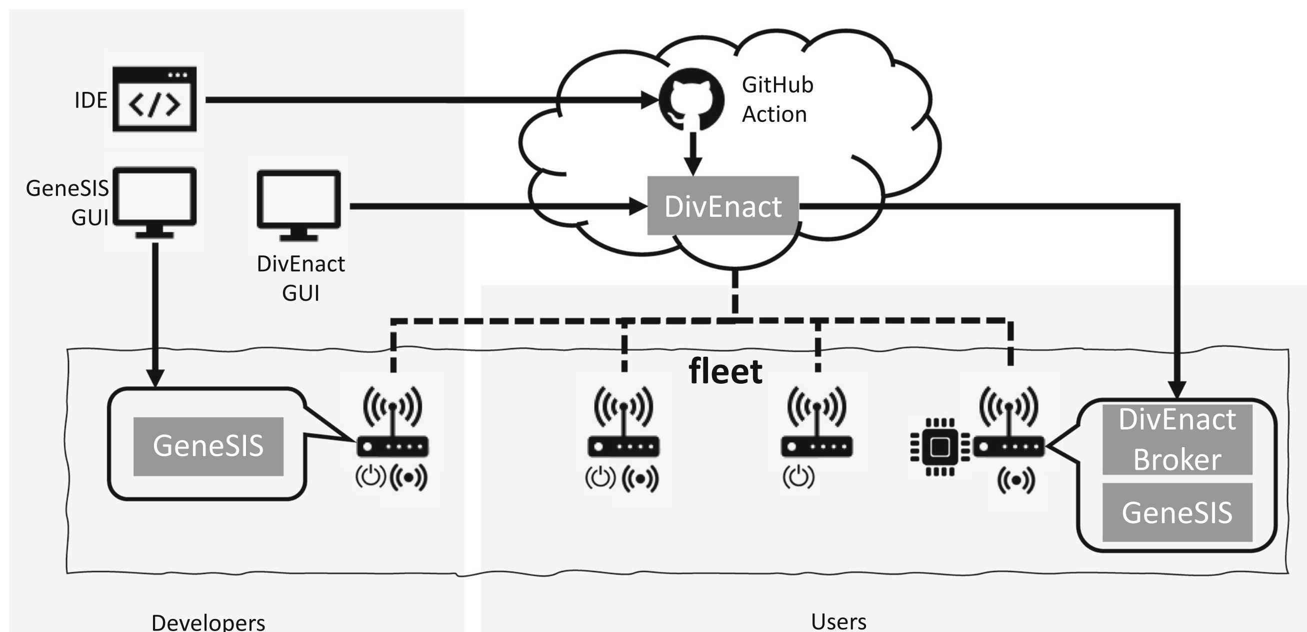


Fig. 7 Deployment bundle for a Smart IoT System. In this bundle, we use our own GeneSIS tool for device deployment (see Fig. 1) to provide a complete solution for Fleet Deployment

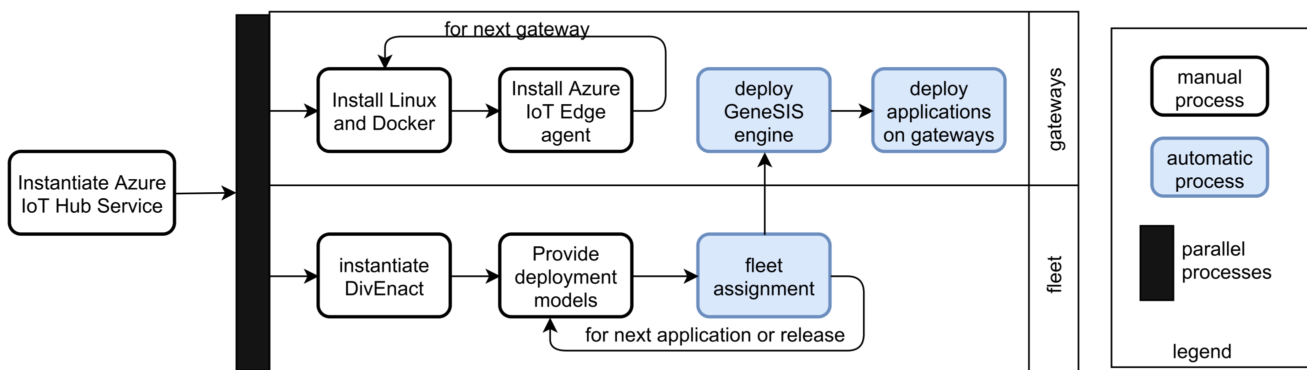


Fig. 8 Steps to instantiate the fleet deployment bundle

an IoT Hub is straight-forward, and can be done as simple as one command line, according to Azure's tutorial.[14]

After creating the IoT Hub, developers can install a DivEnact instance, either on a cloud virtual machine or on a local machine. During the installation, they need to provide a unique connection string for the newly created IoT Hub. After the installation, developers can use the DivEnact service through a web-based GUI from any devices, as long as the host machine of the DivEnact service is reachable from these devices. Through the GUI, developer can manage the fleet, which at this stage is still empty.

A gateway must go through a bootstrap step to be able to join the fleet. Starting with an empty gateway, developers need to install a Linux operating system and the docker engine. After that, they need to install Azure IoT Edge agent, which is a Linux system service plus two Docker containers. We provide a pre-defined shell script to automate this bootstrap step.[15] Every gateway has a unique identification (the device connection string), which can be obtained from the IoT Hub.

The deployment of application is done through DivEnact. Developers need to provide the deployment models, including the GeneSIS models and the DivEnact model as presented in Sects. 3.1 and 4, respectively. After receiving the model, DivEnact will automatically assign the GeneSIS models to the gateways and then instruct the gateway to install application components. If there is no GeneSIS engine installed

[14] https://docs.microsoft.com/en-us/azure/iot-edge/quickstart-linux?view=iotedge-2020-11#create-an-iot-hub.

[15] https://gitlab.com/enact/divenact/-/blob/master/edge/bootstrap/setup.sh.

on a gateway, the local deployment will start with installing it first.

The workflow in Fig. 8 contains two loops, i.e. the gateway enrolment and the continuous application deployment. The two loops happen independently, and the developers can keep deploying new applications (or new releases of applications), while new gateways can join the fleet at the same time. The two activities are normally overlooked by different stakeholders, i.e. gateway administrators and applications developers. The decoupling between the two activities ensures that the two stakeholders can work independently, which is a major advantage of fleet deployment.

7.3 Use case of the fleet deployment bundle

We introduce a second use case in the real Tellu production set-up to illustrate how the fleet deployment bundle works in the DevOps process and what it means to the involved stakeholders. For the current RPM product, Tellu has distributed 400 gateways, each of which is used by a single patient. Tellu operates all the gateways, on behalf of different customers, i.e. nursing service providers running by either private companies or local municipalities. All the gateways collectively form a large and geographically distributed fleet. The use case shows that the fleet deployment concept and approach not only benefits the application developers, but also other stakeholders such as device administrators and end users, and is useful for different stages of application development. This addresses the research question **R6**.

The use case is based on a real scenario when a new access control mechanism is introduced into the RPM fleet. The RPM service monitors the real-time status of a patient, and thus requires sufficient access control to maintain patient data privacy. Among all the available sensors, the camera is used for the most sensitive data. In the current system, the access control strategy is strict and static, and only the personal nurse can access the live video stream of the patient at specific time slots during the day. Recently, one tenant raised a new requirement for a more flexible access control strategy—i.e. upon emergency, e.g. when the patient accidentally falls down, the nurse on duty should be granted with an elevated temporary access rights to the camera to be able to remotely assess the patient's current situation. The DevOps team at Tellu decided to implement this feature gradually, starting from a basic emergency situation when the patient presses an alarm button. The button hardware includes a microcontroller board (Arduino Uno) connected to the gateway via a USB port, and a remote Bluetooth-connected button that the patient has to wear as a necklace. The software part includes: *(i)* C code running on the Arduino board to listen for remote button events, *(ii)* an updated version of gateway software to associate the button with the patient and to upload the button press events to the access control service in the cloud, and *(iii)* an upgraded access control back-end service that considers both the user role and the current emergency level.

Tellu has started to distribute the button hardware to volunteering patients, and now needs to deploy the updated software (*i* and *ii* above) to the gateways that belong to the requesting tenants and have the button hardware already connected. This is a typical fleet deployment problem—i.e. the software variant with new access control exists together with the previous version, and should only be assigned to a sub-set of the gateways depending on the tenant and hardware contexts. Such 'last-mile' deployment of the assigned software to the gateway involves software deployment on heterogeneous devices, including microcontroller boards [25].

Figure 9 shows how the fleet deployment bundle is used to realize this scenario. In the diagram, the corner-folded boxes represent the input artefacts, and the rounded rectangles are the activities. The arrows represent the causal links between activities, i.e. the target activities happens as a result of the source one. The vertical spatial relationship between activities roughly indicate their temporal relation, i.e. the activities placed higher in the diagram happens before the lower ones. We now describe the usage from three different, yet closely related perspectives—namely the DevOps team, the components inside the bundle and the end users.

From the DevOps team's perspective The development phase for this scenario yields three artefacts: *(i)* source code running on the gateways and microcontrollers, which are further packaged as a Docker image and a code package, respectively, *(ii)* GeneSIS deployment model describing how to deploy the Docker container and the code package on abstract gateways and microcontrollers, similar to the snippet in Listing 1, and *(iii)* fleet deployment models and constraints. The important constraints are:

- The new deployment should only apply to the gateways from the specific tenant;
- It should not be deployed to gateways without the button hardware.

The first two software artefacts will be committed to the Git repository, while the third one is the input for the DivEnact tool to configure the fleet assignment activity. Once the artefacts are ready, the DevOps team will launch fleet deployment action, and after that, they will keep receiving notifications about specific devices impacted and the deployment status. It is worth noting that during the whole process, the DevOps team does not need to know explicitly which concrete devices belong to the tenant, and whether and when the patients plugged the button hardware. They only view the entire fleet as a whole.

Inside the fleet deployment tools The components within the fleet deployment tools are running in two different places—cloud and gateways. The agents in the gateways keep

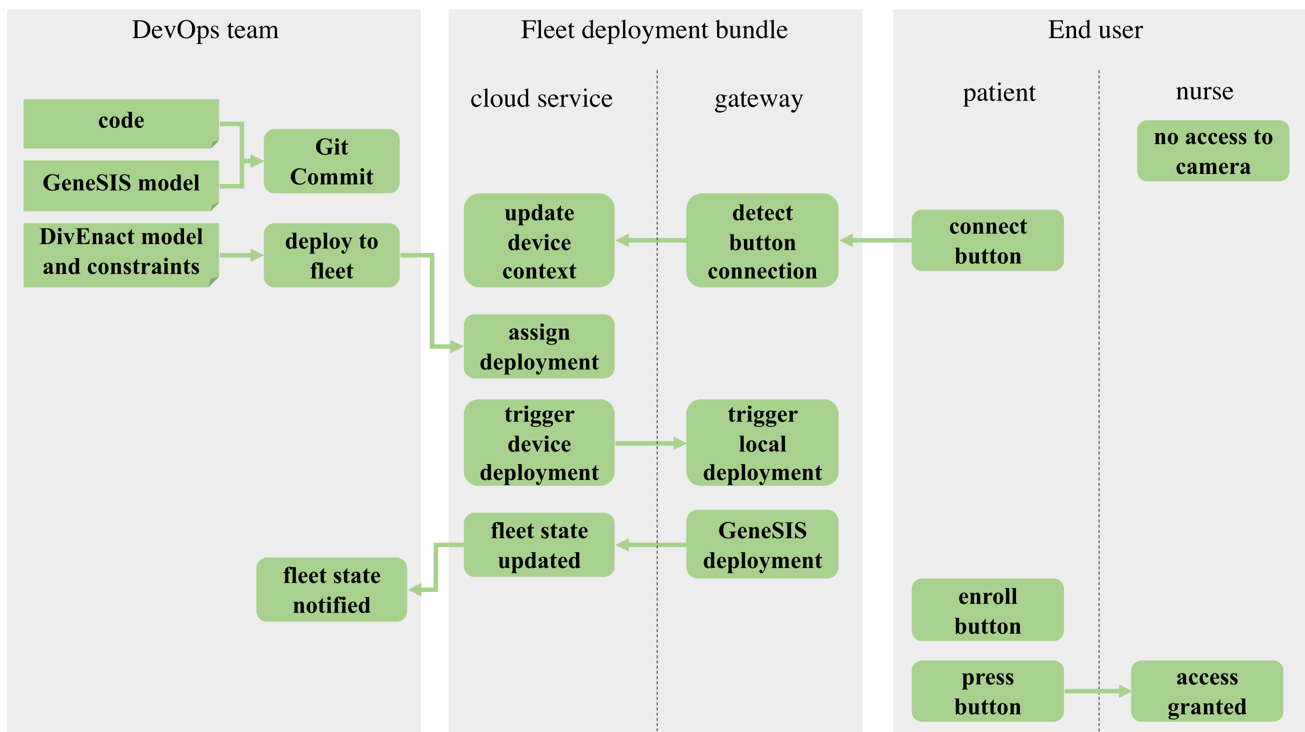


Fig. 9 Activities implementing the access control use case

detecting new hardware, and update the context of the gateway managed by the cloud service. Once fleet deployment is triggered by the DevOps team, it will first perform fleet assignment to decide which gateways should be provisioned with the new software. Next, the cloud service will instruct these gateways to trigger the deployment of the GeneSIS model within the gateway. Fleet assignment does not impact all the edge devices at the same time, and the DivEnact tool will re-evaluate the assignment in a pre-defined interval (5 minutes in this case). That is, whenever new devices join the fleet, previously inactive devices are back online, or the context of some devices changes, the tool will re-evaluate the assignment and trigger device deployment if needed. The result of the gateway deployment will be sent back to the DevOps teams as feedback.

From the end-users' perspective Two types of end-users are involved in this scenario, i.e. the patients and the nurses. From the patients' point of view, they will be informed about the new feature, and provide consent to obtain the alarm button devices. After receiving the hardware, they can plug it into the gateway at any time. When the new software is deployed on the gateway and the connected button's microcontroller, the LCD screen on the button will blink, reminding the patient to go through an enrollment process, i.e. login to their mobile app and type in the code shown on the LCD screen, to verify that the patient him-/herself owns and controls the button. After that, in case of emergency situations, they can press the button to initiate video communication with the supervising nurse.

From the nurse's point of view, the only change introduced by the newly assigned and deployed software is that some of them (i.e. the ones supervising patients who have been provisioned with an alarm button) will have access to the camera when a patient presses the emergency button—a feature that is otherwise restricted in less critical circumstances.

It is worth noting that during the entire process, the only manual work required from the developers is to define the GeneSIS and DivEnact models, as well as the constraints. All the subsequent assignment and deployment work takes place automatically. The models can be defined in the graphical and textual editors provided by our tools. To use them, developers are expected to have relevant knowledge on software deployment, edge computing and IaC mechanisms. Therefore, the target users of this approach are not traditional developers focusing only on coding, but rather DevOps engineers who are also in charge of deployment. Basic knowledge of model-driven engineering would also help them use the tool in an more convenient way, but is not mandatory. On the other hand, writing complex constraints may be challenging for developers with little background on logical programming (especially FOL). Therefore, another potential direction for future work will

investigate more intuitive interfaces for constraint specification.

8 Related work

Managing software updates across a large fleet of devices has been a key challenge for the mobile OS providers, such as Apple and Google. At present, their adopted over-the-air update mechanism implements a publish–subscribe model, where mobile devices first get notified and then fetch available updates through the centralized marketplace. Since smartphone fleets are rather homogeneous (i.e. modern smartphones are equipped with more or less the same resources and capabilities), there is no challenge of multi-criteria software assignment as such, and the compatibility check is performed only once, upon the initial installation. Furthermore, in a fleet of smartphones there are typically no global system goals, such as even distribution of components or A/B testing.

Deploying software applications on hardware platforms with respect to heterogeneous contexts is not a new problem, and has been studied for several decades now. In particular, the software communication architecture (SCA) [32] describes how properties of waveform software components are mapped to heterogeneous device characteristics, thus enabling software-defined radio (SDR) systems. Another relevant reference architecture for deploying component-based applications into heterogeneous distributed target systems is described in [44]. In particular, the proposed architecture includes the concept of *Planner*—a component responsible for matching software requirements to available platform resources and deciding whether a component is compatible with a device. These existing specifications remain implementation-agnostic and only describe the high-level concepts, whereas our approach attempts to provide a concrete solution with tool supports, for a specific use case.

Automatic deployment is a key challenge to adopt the DevOps practices. IaC tools, such as Ansible[16] [33] and Chef[17] can be used to support the installation of software modules on devices, while edge computing frameworks such as the already mentioned Azure IoT Edge and AWS IoT Greengrass provide online services to maintain device fleets throughout the whole device life cycle. Our approach is complementary to these tools, and provides the missing capability of assigning multiple deployment models to many devices in a reliable automated manner.

As far as firmware (low-level executable code deployed on microcontroller-enabled IoT devices) updates are concerned, the situation has only started to change recently. Traditionally, embedded IoT devices have been flashed with 'one-off' firmware not subject to be updated in the future. This was due to hardware and connectivity constraints, as well as security considerations, which limited consequent manipulation of embedded code once a device leaves the production line and is shipped to a customer. Delivering new firmware via an Internet-connected gateway or a smartphone was a natural fit [35,39,43]. This way, IoT devices connected to a gateway via a non-TCP/IP network interface supporting a firmware flashing protocol (e.g. Bluetooth or a serial port) can receive updates. A considerable subset of these existing approaches focuses on the security and trustworthiness of firmware updates [6,18,29,38,49,55]. In the absence of a centralized gateway for delivering firmware updates. Some recent research works also investigated how narrow-bandwidth networks can be used to enable firmware over-the-air (FOTA) of battery-powered devices [1,2]. Comparing with these approaches, our approach on device deployment targets a general deployment support to different types of devices and attempts to use edge devices as delegates to achieve automatic deployment on resource- and network-constrained devices.

Another relevant research attempt to assign firmware to relevant devices focuses on the distributed and hierarchical nature of IoT infrastructures and aims to implement software assignment in a scalable and time-efficient manner. In their works [4,20,36], to encode and query device context properties and available services, the authors employed Bloom filters—space-efficient data structures with fixed time of writing and querying. These properties make Bloom filters especially suitable for memory-constrained devices. In this approach, each device in a fleet is expected to maintain a Bloom filter representation of its current context and match it with an incoming assignment query (i.e. whether or not it matches the requested parameters), upon receiving one from an upstream gateway. Albeit extremely scalable, this approach is limited by the requirement to run queries on the constrained devices and the increased data exchange between devices and gateways. Moreover, with the distributed nature of the Bloom filter-based approaches, it is impossible to perform assignment following global fleet goals (e.g. distribution of firmware for A/B testing), since there is no global overview of the whole fleet. Finally, a Bloom filter is probabilistic data structure, where false negatives are possible when querying, which reduces the accuracy of firmware software assignment.

An approach combining the scalability and centralized assignment is described in [24]. The authors treat the challenge of allocating software components to edge and IoT devices as a generalized assignment problem and demonstrate how it can be solved using simple, yet efficient combinatorial optimization techniques. The approach is lightweight and scalable, but suffers from reduced expres-

[16] https://www.ansible.com/.

[17] https://www.chef.io/products/chef-infra/.

sivity, since it intentionally does not rely on modelling techniques.

In general, the assignment problem frequently appears in ICT scenarios, where some resources need to be allocated to available nodes, often taking into consideration various context-specific characteristics [45,46]. The research community has come up with multiple algorithms, ranging in their computational complexity, completeness, preciseness, etc. Many of these approaches treat assignment as a collection of constraints, which need to be satisfied in order to find an optimal solution in the given circumstances [3,14]. The SMT-based approaches are specifically popular and efficient due to their expressively and rich modelling language [15]. In this respect, a relevant approach that also makes use of SMT and Z3 Solver is described by Pradhan et al. [48]. The authors introduce orchestration middleware, which continuously evaluates available resources on edge nodes and redeploys software accordingly. Similar goal is pursued by Vogler et al. in [53], where authors report on a workload balancer for distributing software components on the edge. Multiple approaches specifically focus on the autonomic and wireless nature of IoT devices and contribute to energy-efficient resource allocation, where the primary criterion for software deployment is energy efficiency [56]. The main obstacle for using SMT in practice is the gap between real platforms and the mathematical model, and our approach uses model-based techniques to bridge this gap.

Our work is also related to variability management in software engineering. Research works on software product lines (SPLs) [47] seek for systematic management of software variability throughout the software life cycle. Feature models are widely used in these approaches to capture the commonality and variability of a product line [11] as a tree of product features. Developers select required features to derive a configuration of a feature model as a description of a concrete product. Many approaches use constraint solving to verify the derived configurations from various perspectives [8] or to automatically fix the inconsistencies [54]. The fleet assignment part in our work follows a similar approach to transform a high-level model into a constraint satisfaction problem to apply constraint solving. However, instead of a general feature model, our transformation works on a specific model for IoT deployment, which is essentially about mapping between two groups of elements. Our work would benefit from SPL approaches that facilitate developers in creating variant deployment plans, but this part far beyond the scope of this paper. In addition, a future extension to our work is to use feature models instead of simple tags in order to describe deployment plans and device choices, which will significantly increase the expressive power of the modelling language and thus support more complex systems. The planned direction is related to approaches on using feature models for software deployment [34].

In general, model-based techniques are often used to support DevOps. Combemale et al. [19] present an approach to use a continuum of models from design to run-time to speed up the DevOps processes in the context of cyber-physical systems. Artavc et al. [5] uses deployment models on multiple cloud environments, which is a promising way to support the smooth transition of software from testing to production environments. Our previous approaches use model-based engineering to engineer the diversity of software deployment within the entire application [41,50,52]. Looking at approaches targeted at particular application domains, Bucchiarone et al. [17] use multi-level modelling to automate the deployment of gaming systems. In [27,28], the authors apply model-driven design space exploration techniques to the automotive domain and demonstrate how different variants of embedded software are identified as more beneficial in different contexts, depending on the optimization objective and subject to multiple constraints in place. To solve this optimization problem, the authors also employ the SMT techniques and the Z3 solver implementation. As opposed to these works, our approach focuses on a particular problem in the DevOps of IoT applications, and it goes beyond merely providing models as an effective interface for developers to conduct manual work, but aims at the full automation of a particular task, i.e. fleet deployment.

9 Discussion

9.1 Threats to validity

The discussion of the proposed approach would not be complete without a critical analysis of the threats to validity. With the current size of the edge fleets managed by Tellu, it would not be realistic to perform the experiments on real physical devices, and therefore we opted for evaluating the proposed approach based on simulation, which, nevertheless, reflects the real operations and business requirements faced by the company. Besides, this way, for clarity purposes we managed to omit some trivial and non-relevant details and demonstrated what is really novel and important.

Another potential limitation of the approach is that under the current setting, developers need to clearly know upfront what exactly every context means to the assignment, in order to design the effective constraints. However, under some circumstances, the potential effect of certain characteristics on the assignment may be fuzzy or not be so evident from the beginning. For example, if the objective is to maximize the chance of receiving good feedback on a preview version, we may need to consider a series of contexts, for example, how often the patient uses the application, how actively they provide feedback, how useful the feedback has been so far, etc.

In such case, defining precise and unambiguous constraints and their effect on such context characteristics is a challenge.

We foresee two possible directions to address this issue. One possibility is to apply semantic web techniques to the meta-models, by providing a common unambiguous ontology—i.e. a shared standard vocabulary for describing device contexts. Another possible direction is to combine constraint solving with machine learning. We use constraint solving only to ensure that the assignment does not violate hard and soft constraints currently being in place. At the same time, we can also allow the tool to learn from historical records (i.e. what assignments have already been computed, and what feedback has been collected) in order to lead to a good assignment.

9.2 Lessons learned on tool development

The main output from this research work is the fleet deployment bundle comprising two open source tools.

A lesson learned for such tool-building research is to have the target users involved in the development process. Our industry partner, Tellu, has been involved in the development process from the beginning, participating in the requirement analysis, tool design, and the evaluation planning. This not only ensures the usefulness of the tools to the company, but also allows the company to prepare for the transition to the new DevOps methods based on the fleet deployment concept. From technical point of view, using widely adopted platforms available in the market (such as Azure IoT Hub in our case) helps to minimize the cost of applying the tools to the industry cases.

Our ambition behind the tool building is to create community-driven open-source tools, so that eventually the tools will be overtaken and maintained by an open-source community. This is a practical route to maximize the impact of the fleet deployment research in the industry, considering the fact that the authors are either researchers or domain-oriented application developers. Therefore, maintaining software development tools is not part of our interest. The progress towards an active open-source community around the tools is still ongoing. One possible reason for the slow growth is that the whole development process is heavily coupled to one particular company and therefore lacks the generality to attract sufficient interest from a wider community. Thus, an important future work direction is to engage other users to generalize the tools.

9.3 Generality of the approach

The fleet deployment approach is motivated and evaluated in the frame of a use case in the eHealth domain. However, the concepts, methods and tools are generic to edge computing applications in other domains, such as manufacturing and smart cities, where edge computing will be widely used. For example, in the manufacturing domain, more and more factories are employing edge devices to aggregate and pre-process the real-time data from machinery before sending them for global analysis to the server or to the cloud. Similarly, in the smart city domain [22], some data sets are processed on edge devices close to the citizens and deleted afterwards for the sake of preserving privacy. For these edge applications, it is important to keep updating the software on the edge for data processing and other emerging business logic, and different devices may need different software variants, depending not only on the device profiles but also on their cyber-physical environments, including the factory machinery that they monitor or device geo-location within a smart city. This is the type of problems that fleet deployment is targeting at.

10 Conclusion and future work

This paper described an industry-driven research, undertaken in collaboration with the edge application provider Tellu, aiming to enable the automatic assignment and deployment of software components to a fleet of edge and IoT devices. We implemented an architecture where software updates matching specific device contexts are pushed down from the cloud to intermediate edge gateways and terminal IoT devices. This way, the proposed approach supports the software assignment and deployment activities in the whole IoT–edge–cloud continuum and contributes to wider re-usability of hardware infrastructure across multiple IoT scenarios. Such fleet deployment not only accelerates the deployment of software updates for the developers, but also supports the adaptation to run-time changes. New devices joining the fleet trigger the fleet reassignment action, which will eventually deploy the required software components to the new devices without human intervention. We addressed the problem using model-based techniques including meta-modelling, transformation and constraint solving and integrated the prototype tool into the existing DevOps toolchain currently used by Tellu. An initial demonstration with simulated DevOps iterations showed that the approach is useful in the DevOps process, by generating correct assignment results and triggering actual deployment, as well as providing valuable feedback for further DevOps iterations.

As an industry-driven research, the approach is specific to the concrete problem faced by Tellu, and the early-stage application of the approach to Tellu's DevOps teams has already shown observable benefits. As an initial and informal evaluation, comparing their DevOps routines before and after applying the fleet deployment tools, Tellu developers expect to see an increase in their DevOps pace by approximately three times, with roughly five times lower effort on

integration, after the approach is fully embedded into their production processes. This will be achieved through easier delivery of devices to their patients, and a higher automation level for continuous deployment. More details can be found in a recent report by Tellu [40]. As part of an ongoing research project, future work also includes a formal empirical study to follow up the adoption of our tools by Tellu's development team, and an evaluation of the effectiveness of using the fleet deployment approach.

Acknowledgements This work is partially funded by the European Commission's H2020 Programme under Grant Agreement 780351 (ENACT), 101020416 (ERATOSTHENES) and the Norwegian Research Council's BIA-IPN Programme No. 309700 (FLEET)

Funding Open access funding provided by SINTEF.

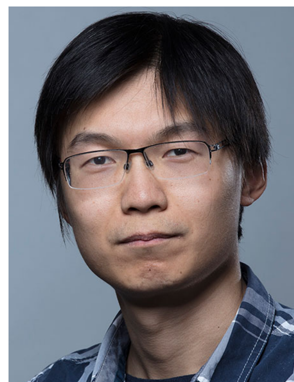
Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Abdelfadeel, K., Farrell, T., McDonald, D., Pesch, D.: How to make firmware updates over LoRaWAN possible. In: 2020 IEEE 21st International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM), pp. 16–25. IEEE (2020)
2. Anastasiou, A., Christodoulou, P., Christodoulou, K., Vassiliou, V., Zinonos, Z.: IoT device firmware update over LoRa: The blockchain solution. In: 2020 16th International Conference on Distributed Computing in Sensor Systems (DCOSS), pp. 404–411. IEEE (2020)
3. Ansótegui, C., Bofill, M., Palahí, M., Suy, J., Villaret, M.: Satisfiability modulo theories: an efficient approach for the resource-constrained project scheduling problem. In: Ninth Symposium of Abstraction, Reformulation, and Approximation (2011)
4. Antonini, M., Cirani, S., Ferrari, G., Medagliani, P., Picone, M., Veltri, L.: Lightweight multicast forwarding for service discovery in low-power IoT networks. In: 2014 22nd International Conference on Software, Telecommunications and Computer Networks (SoftCOM), pp. 133–138. IEEE (2014)
5. Artač, M., Borovšak, T., Di Nitto, E., Guerriero, M., Tamburri, D.A.: Model-driven continuous deployment for quality DevOps. In: Proceedings of the 2nd International Workshop on Quality-Aware DevOps, pp. 40–41 (2016)
6. Asokan, N., Nyman, T., Rattanavipanon, N., Sadeghi, A.R., Tsudik, G.: ASSURED: architecture for secure software update of realistic embedded devices. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **37**(11), 2290–2300 (2018)
7. Bass, L., Weber, I., Zhu, L.: DevOps: A Software Architect's Perspective. Addison-Wesley Professional (2015)
8. Benavides, D., Segura, S., Ruiz-Cortés, A.: Automated analysis of feature models 20 years later: a literature review. Inf. Syst. **35**(6), 615–636 (2010)
9. Bergmayr, A., Breitenbücher, U., Ferry, N., Rossini, A., Solberg, A., Wimmer, M., Kappel, G., Leymann, F.: A systematic review of cloud modeling languages. ACM Comput. Surv. (CSUR) **51**(1), 1–38 (2018)
10. Bergmayr, A., Breitenbücher, U., Kopp, O., Wimmer, M., Kappel, G., Leymann, F.: From architecture modeling to application provisioning for the cloud by combining UML and TOSCA. In: CLOSER (2), pp. 97–108 (2016)
11. Beuche, D., Dalgarno, M.: Software product line engineering with feature models. Overload J. **78**, 5–8 (2007)
12. Binz, T., Breitenbücher, U., Haupt, F., Kopp, O., Leymann, F., Nowak, A., Wagner, S.: Opentosca—a runtime for TOSCA-based cloud applications. In: International Conference on Service-Oriented Computing, pp. 692–695. Springer (2013)
13. Blair, G., Bencomo, N., France, R.B.: Models@ run. time. Computer **42**(10), 22–27 (2009)
14. Bofill, M., Palahí, M., Suy, J., Villaret, M.: Solving constraint satisfaction problems with SAT modulo theories. Constraints **17**(3), 273–303 (2012)
15. Bonacina, M.P., Graham-Lengrand, S., Shankar, N.: Satisfiability modulo theories and assignments. In: International Conference on Automated Deduction, pp. 42–59. Springer (2017)
16. Bryant, R.E., Lahiri, S.K., Seshia, S.A.: Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In: International Conference on Computer Aided Verification, pp. 78–92. Springer (2002)
17. Bucchiarone, A., Cicchetti, A., Marconi, A.: Exploiting multi-level modelling for designing and deploying gameful systems. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 34–44. IEEE (2019)
18. Choi, B.C., Lee, S.H., Na, J.C., Lee, J.H.: Secure firmware validation and update for consumer devices in home networking. IEEE Trans. Consum. Electron. **62**(1), 39–44 (2016)
19. Combemale, B., Wimmer, M.: Towards a model-based DevOps for cyber-physical systems. In: Software Engineering Aspects of Continuous Development (2019)
20. Dautov, R., Distefano, S.: Targeted content delivery to IoT devices using Bloom filters. In: International Conference on Ad-Hoc Networks and Wireless, pp. 39–52. Springer (2017)
21. Dautov, R., Distefano, S.: Stream processing on clustered edge devices. IEEE Transactions on Cloud Computing (2020). https://doi.org/10.1109/TCC.2020.2983402
22. Dautov, R., Distefano, S., Merlino, G., Bruneo, D., Longo, F., Puliafito, A.: Towards a global intelligent surveillance system. In: Proceedings of the 11th International Conference on Distributed Smart Cameras, pp. 119–124 (2017)
23. Dautov, R., Song, H.: Towards IoT diversity via automated fleet management. In: MDE4IoT/ModComp@MoDELS, pp. 47–54 (2019)
24. Dautov, R., Song, H., Ferry, N.: A light-weight approach to software assignment at the edge. In: 2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC), pp. 380–385. IEEE (2020)
25. Dautov, R., Song, H., Ferry, N.: Towards a sustainable IoT with last-mile software deployment. In: 2021 IEEE Symposium on Computers and Communications (ISCC), pp. 1–6. IEEE (2021)
26. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 337–340. Springer (2008)
27. Eder, J., Bahya, A., Voss, S., Ipatiov, A., Khalil, M.: From deployment to platform exploration: automatic synthesis of distributed automotive hardware architectures. In: Proceedings of the 21th

ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, pp. 438–446 (2018)
28. Eder, J., Zverlov, S., Voss, S., Khalil, M., Ipatiov, A.: Bringing DSE to life: exploring the design space of an industrial automotive use case. In: 2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 270–280. IEEE (2017)
29. Falas, S., Konstantinou, C., Michael, M.K.: A Modular End-to-End Framework for Secure Firmware Updates on Embedded Systems. arXiv preprint arXiv:2007.09071 (2020)
30. Ferry, N., Chauvel, F., Song, H., Rossini, A., Lushpenko, M., Solberg, A.: Cloudmf: model-driven management of multi-cloud applications. ACM Trans. Internet Technol. (TOIT) **18**(2), 1–24 (2018)
31. Ferry, N., Nguyen, P.H., Song, H., Rios, E., Iturbe, E., Martinez, S., Rego, A., et al.: Continuous deployment of trustworthy smart IoT systems. J. Object Technol. **19**(2), 1–23 (2020)
32. González, C.R.A., Dietrich, C.B., Reed, J.H.: Understanding the software communications architecture. IEEE Commun. Mag. **47**(9), 50–57 (2009)
33. Hall, D.: Ansible Configuration Management. Packt Publishing Ltd. (2013)
34. Hochgeschwender, N., Gherardi, L., Shakhirmardanov, A., Kraetzschmar, G.K., Brugali, D., Bruyninckx, H.: A model-based approach to software deployment in robotics. In: 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 3907–3914. IEEE (2013)
35. Jain, N., Mali, S.G., Kulkarni, S.: Infield firmware update: challenges and solutions. In: 2016 International Conference on Communication and Signal Processing (ICCSP), pp. 1232–1236. IEEE (2016)
36. Jo, H.J., Kwon, J.H., Ko, I.Y.: Distributed service discovery in mobile IoT environments using hierarchical bloom filters. In: International Conference on Web Engineering, pp. 498–514. Springer (2015)
37. Kleppe, A.G., Warmer, J.B., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley Professional (2003)
38. Lo, N.W., Hsu, S.H.: A secure IoT firmware update framework based on MQTT protocol. In: International Conference on Information Systems Architecture and Technology, pp. 187–198. Springer (2019)
39. McGrath, W., Etemadi, M., Roy, S., Hartmann, B.: Fabryq: using phones as gateways to prototype internet of things applications using web scripting. In: Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems, pp. 164–173 (2015)
40. Anton Gonzalez, M.A., et al.: Final evaluation and validation report. Technical Report, The ENACT Consortium (2021)
41. Morin, B., Høgenes, J., Song, H., Harrand, N., Baudry, B.: Engineering software diversity: a model-based approach to systematically diversify communications. In: Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, pp. 155–165 (2018)
42. Nguyen, P., Ferry, N., Erdogan, G., Song, H., Lavirotte, S., Tigli, J.Y., Solberg, A.: Advances in deployment and orchestration approaches for IoT—a systematic review. In: 2019 IEEE International Congress on Internet of Things (ICIOT), pp. 53–60. IEEE (2019)
43. Nikolov, N.: Research firmware update over the air from the cloud. In: 2018 IEEE XXVII International Scientific Conference Electronics-ET, pp. 1–4. IEEE (2018)
44. OMG: Deployment and Configuration of Component-based Distributed Applications Specification, v4.0. Technical Report, Object Management Group, Inc. (2006). https://www.omg.org/spec/DEPL/4.0/PDF
45. Öncan, T.: A survey of the generalized assignment problem and its applications. INFOR Inf. Syst. Oper. Res. **45**(3), 123–141 (2007)
46. Pentico, D.W.: Assignment problems: a golden anniversary survey. Eur. J. Oper. Res. **176**(2), 774–793 (2007)
47. Pohl, K., Böckle, G., Van Der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques, vol. 1. Springer (2005)
48. Pradhan, S., Dubey, A., Khare, S., Nannapaneni, S., Gokhale, A., Mahadevan, S., Schmidt, D.C., Lehofer, M.: Chariot: goal-driven orchestration middleware for resilient IoT systems. ACM Trans. Cyber-Phys. Syst. **2**(3), 1–37 (2018)
49. Sahlmann, K., Clemens, V., Nowak, M., Schnor, B.: MUP: simplifying secure over-the-air update with MQTT for constrained IoT devices. Sensors **21**(1), 10 (2021)
50. Song, H., Chauvel, F., Solberg, A.: Deep customization of multi-tenant SaaS using intrusive microservices. In: Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results, pp. 97–100 (2018)
51. Song, H., Dautov, R., Ferry, N., Solberg, A., Fleurey, F.: Model-based fleet deployment of edge computing applications. In: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, pp. 132–142 (2020)
52. Song, H., Elgammal, A., Nallur, V., Chauvel, F., Fleurey, F., Clarke, S.: On architectural diversity of dynamic adaptive systems. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 2, pp. 595–598. IEEE (2015)
53. Vögler, M., Schleicher, J.M., Inzinger, C., Dustdar, S.: A scalable framework for provisioning large-scale IoT deployments. ACM Trans. Internet Technol. (TOIT) **16**(2), 1–20 (2016)
54. Xiong, Y., Hubaux, A., She, S., Czarnecki, K.: Generating range fixes for software configuration. In: 2012 34th International Conference on Software Engineering (ICSE), pp. 58–68. IEEE (2012)
55. Yohan, A., Lo, N.W., Achawapong, S.: Blockchain-based firmware update framework for internet-of-things environment. In: Proceedings of the International Conference on Information and Knowledge Engineering (IKE), pp. 151–155. The Steering Committee of the World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp) (2018)
56. You, C., Huang, K., Chae, H., Kim, B.H.: Energy-efficient resource allocation for mobile-edge computation offloading. IEEE Trans. Wirel. Commun. **16**(3), 1397–1411 (2016)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Hui Song is a Senior Research Scientist with SINTEF Digital. His research interest is in software development methods and tools, and their applications on IoT, edge and cloud systems. He received his Ph.D. on computer science from Peking University, China. He has worked on many EU and national research projects related to software engineering in cloud and IoT systems and was the coordinator of the H2020 project ENACT, which investigated DevOps for IoT.

Rustem Dautov is a Research Scientist at SINTEF Digital, based in Oslo, Norway. His work focuses on applied research in the areas of the Internet of things, edge computing, and cloud computing, applying model-driven engineering and semantic web techniques. In the current position, Rustem has been involved in multiple national and international research and innovation projects, both as a researcher and as a coordinator. He received his Ph.D. degree in computer science from The University of Sheffield, UK. Before joining SINTEF, he had worked in several academic and software engineering positions in Greece and Russia. Rustem has co-authored 40+ publications and is actively serving as a reviewer for multiple international conferences and journals.

Nicolas Ferry is an Associate Professor at University Côte d'Azur. Prior, he was a Senior Research Scientist at SINTEF. He holds a Ph.D. degree from the University of Nice. His research interest includes model-driven engineering, domain- specific languages, Internet of things, cloud computing, self-adaptive systems, and dynamic adaptive systems. He has actively contributed to various national and international research projects such as the REMICS, CITI-SENSE, MC-Suite and MODAClouds EU projects and was the technical manager of the H2020 ENACT project. He has also served as a program committee member of international conferences and workshops.

Arnor Solberg is the Chief DevOps Officer at Tellu. He is part of the leader team developing the company strategy at both the technical and business level and is leading the DevOps Business Unit at Tellu. Tellu provides TeleHealth and TeleCare services. These services are provided as software as a service (SaaS). Tellu also provides a comprehensive eHealth platform for building and operating advanced eHealth services that computes distributively across the IoT, edge and cloud space. This enables Tellu to provide complete end-to-end solutions including integration with third-party systems such as electronic health records and response centres. Tellu operates in domains where information security and trustworthiness is significant. Arnor was responsible for getting Tellu certified according to the ISO 27001 (Information Security Management).

Franck Fleurey is the Chief Technical Manager at Tellu. His research interests include model-driven software engineering, embedded systems and dynamic adaptive systems. Dr. Fleurey received a Ph.D. in computer science from the University of Rennes 1.