Die approbierte Originalversion dieser Dissertation ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

The approved original version of this thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Programming, Provisioning and Governing IoT Cloud Systems

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der Technischen Wissenschaften

eingereicht von

Dipl.-Ing. Stefan Nastic, BSc.
Matrikelnummer 0527493

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ. Prof. Dr. Schahram Dustdar

Zweitbetreuung: Priv.-Doz. Dr. Hong-Linh Truong

Diese Dissertation haben begutachtet:

Univ. Prof. Dr. Schahram Dustdar

Univ. Prof. Dr. Uwe Zdun

Wien, 15. April 2016

Stefan Nastic

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Programming, Provisioning and Governing IoT Cloud Systems

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Dipl.-Ing. Stefan Nastic, BSc.
Registration Number 0527493

to the Faculty of Informatics

at the TU Wien

Advisor: Univ. Prof. Dr. Schahram Dustdar

Second advisor: Priv.-Doz. Dr. Hong-Linh Truong

The dissertation has been reviewed by:

Univ. Prof. Dr. Schahram Dustdar

Univ. Prof. Dr. Uwe Zdun

Vienna, 15th April, 2016

Stefan Nastic

I dedicate this thesis to my mother and my father.

Erklärung zur Verfassung der Arbeit

Dipl.-Ing. Stefan Nastic, BSc.
Rüdigergasse 6/6, 1050 Vienna, Austria

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 15. April 2016

Stefan Nastic

Acknowledgements

This PhD thesis marks the end of almost a decade long journey of my academic education in Computer Science. In spite many ups and downs, it was most of all a fun journey that largely shaped me into the man I am today – determined, strong in my beliefs, but open minded to new challenges and for the rest of my life an explorer. The work I have done would never be possible without a profound support of many people and now, looking back, I realize that this is not an achievement of a single person, but a result of caring and supporting environment of people who have all contributed in their own ways.

First and foremost, I would like to express my gratitude to Prof. Schahram Dustdar, my mentor and my friend, who patiently guided me throughout the entire thesis. I am thankful to Dr. Hong-Linh Truong for the endless discussions and an enjoyable collaboration. I want to thank Porf. Frank Leymann for reviewing and helping improve this thesis. Also, I am most grateful to Prof. Uwe Zdun for serving as the examiner of the thesis. Moreover, I am grateful to my coworkers from the Distributed Systems Group for providing a highly productive and pleasant working environment. I would also like to express my gratitude to my colleges from Pacific Controls Cloud Computing Lab, for giving me the opportunity to apply my research in cutting edge industrial environment, as well as the fellow researchers from H2020 U-Test project, for the inspired discussions and wonderful collaboration.

Special thanks are reserved for my friends, with whom I have shared so many enjoyable and memorable moments, for always finding a way to help me see the brighter side and for making me a better person. Finally and most importantly my deepest and sincerest thanks are dedicated to my family for their wholehearted love, endless support and everything they have done for me that cannot be put into words. My success will always be your success!

Stefan Nastic
Vienna, April 2016

The work conducted during the course of this thesis has been funded by Pacific Controls Cloud Computing Lab (PC3L), which is a joint lab between Pacific Controls L.L.C., Dubai and the Distributed Systems Group, TU Wien and; by EU Horizon 2020 Programme U-Test project, under grant agreement 645463.

Kurzfassung

In den letzten Jahren konvergieren Cloud Computing und das Internet der Dinge (IoT) immer stärker und schaffen große, geographisch verteilte Systeme. Solche IoT Cloud Systeme haben die Verbreitung von verschiedenen Anwendungen gefördert. Der dringende Bedarf an Volumen, Geschwindigkeit, die Vielfalt von IoT-Daten und schnellere Übertragung von geschäftskritischen Entscheidungen an den Rand der Infrastruktur zu ermöglichen waren die haupt Antriebe dafür. Die Vorteile von IoT Cloud sind zweifellos erkennbar. Jedoch, werden sie von einer Reihe von Herausforderungen begleitet. Eine davon ist das Programmieren von IoT Cloud Anwendungen. Gründe dafür sind die Verarbeitung großer Menge von IoT-Daten, die großen Anzahl von Domainabhängigen IoT Steuerungen und die komplexen Abhängigkeiten zwischen der Anwendungslogik und der Eigenschaften neuer Infrastrukturen. In IoT Cloud Systemen ist eine traditionelle Provisionierung aufgrund der Dynamic, Heterogenität, Umfangs und geographischer Verteilung von IoT Cloud kaum möglich ist. Bei der traditionellen Provisionierung wird implizit davon ausgegangen, dass das IoT-Gerät vor Ort verfügbar ist und eine manuelle Interaktion möglich. Schließlich ist IoT Cloud ein integraler Bestandteil der bestehenden Geschäftsmodelle und Wegbereiter für neue Geschäftsmöglichkeiten. Dies erfordert eine systematische Vorgehensweise in IoT Cloud Governance, die bisher unterentwickelt blieb.

Diese Dissertation entwickelt ein reiches Ökosystem, das neue Modelle, Frameworks und Werkzeuge umfasst, die eine erleichterte Programmierung, Provisionierung und Governance von IoT Cloud Systemen ermöglichen. Zuerst wird ein umfassendes Programmierungsframework eingeführt, dass die Programmierung auf ein deutlich höheres Abstraktion-Niveau hebt und eine einfachere und intuitivere Entwicklung von IoT Cloud Anwendungen ermöglicht. Andererseits, stellt das Framework ein flexibles Programmier-Modell, dass speziell auf ressourcenbeschränkte IoT-Geräte zugeschnitten ist und zur Unterstützung von Domain Experte vorgesehen ist. Zweitens werden ein Provisionierungsmodell und Middleware eingeführt, die unter anderem folgendes anbieten: i) Mechanismen für die Ressourcen Abstraktion und anwendungsspezifische Anpassungen; ii) Unterstützung für die automatisierte Provisionierung von IoT Cloud Ressourcen, Anwendungskomponenten und Konfigurationsmodelle, in einer logisch-zentralisierten Art und Weise, durch Middleware verwaltete APIs; iii) Flexible Provisionierungs-Modelle, die auf Nachfrage Verbrauch von beiden IoT und Cloud Ressourcen unterstützen. Schlußendlich stellt diese Dissertation GovOps vor – ein neuartiges Governance-Modell für die IoT Cloud Systemen. Dieses ermöglicht die nahtlose Ausrichtung von High-Level

Governance-Zielen mit ausführbaren Operations-Prozessen und unterstützt GovOps Manager: i) Zeitkonsistente Governance-Prozesse in die IoT Cloud Systemen zu implementieren; ii) Unsicherheit- und Elastizität-bewusste Govenrnacestrategien zu entwicklen. Die vorgesttelten Ansätze wurden evaluiert basierend auf der realen Welt Fahrzeugflotte- und Gebäudemanagementsysteme.

Abstract

Over the recent years, cloud computing and the Internet of Things (IoT) have been converging ever stronger, sparking creation of large-scale, geographically distributed systems. Such IoT Cloud systems have fostered proliferation of various applications, driven by an urgent need to respond to volume, velocity and variety of IoT data, but also to enable timely propagation of business-crucial decisions to the edge of the infrastructure. The benefits of IoT Cloud are undoubtable, but they are also accompanied with a number of challenges. Programming IoT Cloud applications is challenging due to the need to handle large volumes of IoT data in a nontrivial manner, plethora of domain-dependent IoT controls and, inherently complex dependencies between application business logic and novel infrastructure features. Because of dynamicity, heterogeneity, scale and geographical distribution of IoT Cloud, traditional provisioning approaches, which implicitly assume on-site presence or manual interactions with IoT devices are hardly feasible in this novel landscape. Finally, IoT Cloud systems are becoming an integral part of existing business models and key enabler for new business opportunities. This calls for systematic approach to IoT Cloud governance, which to date remains largely underdeveloped.

This doctoral thesis contributes a rich ecosystem comprising novel models, frameworks and tools, intended to facilitate programming, provisioning and governing IoT Cloud systems. First, a comprehensive programming framework is introduced, which raises the level of programming abstraction, enabling easier and more intuitive development of cloud-centric IoT Cloud applications. On the other side, it provides a flexible programming model, specifically tailored for resource-constrained IoT devices, which is intended to support domain expert developers. Second, a provisioning model and middleware are introduced, which among other, offer: i) Light-weight mechanisms for resource abstraction and application-specific customizations; ii) Support for automated provisioning of IoT Cloud resources, application components and configuration models in a logically centralized manner through middleware managed APIs and; iii) Flexible provisioning models, supporting on-demand consumption of both IoT and Cloud resources. Finally, the thesis introduces GovOps – a novel governance model for IoT Cloud systems. It enables seamless alignment of high-level governance objectives with executable operations processes and supports GovOps managers to: i) Implement time-consistent governance processes for IoT Cloud systems and; ii) Develop uncertainty- and elasticity-aware governance strategies. The proposed approaches have been evaluated based on real-world Fleet- and Building Management Systems.

Contents

List of Figures

List of Listings

List of Publications

Parts of the work presented in this dissertation were published in the following publications. For a full publication list of the author please refer to his website[1].

1. Stefan Nastic, Hong-Linh Truong, and Schahram Dustdar. SDG-Pro: A Programming Framework for Software-Defined IoT Cloud Gateways. Springer, Journal of Internet Services and Applications 2015, 6:21.

2. Stefan Nastic, Georgiana Copil, Hong-Linh Truong, and Schahram Dustdar. Governing Elastic IoT Cloud Systems under Uncertainty. The 7th International Conference on Cloud Computing Technology and Science (CloudCom 2015), 30 November - 3 December, 2015, Vancouver, Canada.

3. Stefan Nastic, Michael Vögler, Christian Inzinger, Hong-Linh Truong, and Schahram Dustdar. rtGovOps: A Runtime Framework for Governance in Large-scale Software-defined IoT Cloud Systems. The 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering, Mar 30 – Apr 3, 2015, San Francisco, CA, USA. (Selected among the 3 best papers.)

4. Stefan Nastic, Christian Inzinger, Hong-Linh Truong, and Schahram Dustdar. GovOps: The Missing Link for Governance in Software-defined IoT Cloud Systems. The 10th International Workshop on Engineering Service Oriented Applications (WESOA'14) in conjunction with ICSOC 2014, 3. November 2014, Paris, France.

5. Stefan Nastic, Sanjin Sehic, Le-Duc Hung, Hong-Linh Truong, and Schahram Dustdar. Provisioning Software-defined IoT Cloud Systems. The 2nd International Conference on Future Internet of Things and Cloud (FiCloud-2014), August 27-29, 2014, Barcelona, Spain.

6. Stefan Nastic, Sanjin Sehic, Michael Vögler, Hong-Linh Truong, and Schahram Dustdar. PatRICIA - A Novel Programming Model for IoT Applications on Cloud Platforms. International Conference on Service Oriented Computing and Applications (SOCA 2013), December 16-18, 2013, Hawaii, USA.

[1]http://dsg.tuwien.ac.at/staff/snastic

7. Michael Vögler, Johannes M. Schleicher, Christian Inzinger, Stefan Nastic, Sanjin Sehic, and Schahram Dustdar. LEONORE – Large-Scale Provisioning of Resource-Constrained IoT Deployments. The 9th IEEE International Symposium on Service-Oriented System Engineering (SOSE 2015), March 30 - April 3, 2015, San Francisco Bay, USA.

8. Stefan Nastic, Hong-Linh Truong, and Schahram Dustdar. A Middleware Infrastructure for Utility-based Provisioning of IoT Cloud Systems. The First IEEE/ACM Symposium on Edge Computing, October 27-28, 2016, Washington DC, USA. (In review).

9. Stefan Nastic, Hong-Linh Truong, and Schahram Dustdar. Data and Control Points: A Programming Model for Resource-constrained IoT Cloud Edge Devices. IEEE International Conference on Systems, Man, and Cybernetics (SMC 2016), October 9-12, 2016, Budapest, Hungary. (In review).

CHAPTER 1

Introduction

Recent advances in the *Internet of Things (IoT)* have provided solid foundations for a global infrastructure of networked physical entities, able to monitor and control their physical status and the surrounding environment, as well as to expose themselves via data streams and services over the Internet [82]. Over the last decade, cloud computing has put itself forward as one of the most important paradigms for delivering and consuming digital resources [6, 90], mainly due to utility-driven, on-demand nature of cloud offerings, which allow customers to elastically provision the exact type and amount of resources needed for a given task at a given time.

Over the recent years, cloud computing and the IoT have been converging ever stronger, sparking creation of very large-scale, geographically distributed systems. These emerging systems are generally categorized as *IoT Cloud systems*[1]. Such systems extend the traditional cloud computing systems beyond the data centers and cloud services to include a variety of edge IoT devices such as sensors and sensory gateways. On the one side, these systems utilize the IoT infrastructure resources to deliver novel value-added services, which leverage data from different sensor devices and enable timely propagation of decisions, crucial for business operation, to the edge of the infrastructure. On the other side, IoT Cloud systems utilize cloud's theoretically unlimited resources, e.g., compute and storage, to enhance traditionally resource-constrained IoT devices as well as to support the rising demand for complex analytics of large volumes of data generated by the IoT. Moreover, IoT Cloud systems represent an important landmark in a paradigm shift from single-purpose, closed Machine-to-Machine (M2M) systems to an open ecosystem, in which numerous stakeholders interact in order to deliver and consume large pools of IoT Cloud resources and capabilities as well as to develop diverse IoT Cloud applications. Therefore, the novel IoT Cloud systems denote a convergence of IoT and Cloud beyond

[1]In this thesis we will mostly use the term IoT Cloud systems/applications. However, depending on geographical region, particular research focus or even type of organization, this term is synonymously used with Cyber-physical Systems (CPS), Mobile Cloud systems and Cloud of Things.

on a mere technological level, e.g., computation offloading [29, 85] or data staging [42, 3], thus calling for development of novel engineering, management and business models that are specifically tailored for the new landscape. However, this fully fledged convergence can only be achieved by looking beyond traditional IoT or Cloud approaches, to also consider advanced engineering and operations approaches such as software-defined [87] and DevOps [40], which offer a novel perspective on programming, provisioning and governing the IoT Cloud systems.

The emerging IoT Cloud systems enable the Internet to connect previously unreachable areas of production chains, public sector and other spheres of our society. They create a whole new set of services, augmented by leveraging new data and capabilities provided by the IoT. Such systems offer numerous advantages and create new opportunities to the variety of the involved stakeholders in terms of optimizing key tasks of their existing business process as well as allowing for new types of business models and cross-domain applications. However, due to diversity, complexity and scale of IoT Cloud systems, elasticity requirements regarding novel IoT Cloud resources, demand for fine-grained consumption and customization of the IoT Cloud infrastructure, as well as the need to handle large volumes of data in a nontrivial manner, the IoT Cloud systems create a plethora of challenges for system designers, developers and operations managers. To date, the development of IoT cloud systems and applications mostly involves composing device-level services into complex control sequences and data processing schemes, which makes the development process a highly challenging task and potentially limits the programming scale of such applications on IoT Cloud platforms. Concrete abstractions, programming models and runtime mechanisms, which enable efficient, more intuitive and scalable development of IoT Cloud applications remain largely underdeveloped. Currently in such systems, domain-specific application requirements drive the design of all system components and determine most technical choices ranging from hardware devices to application behaviors. This typically results in vertically closed systems in which hardware, network, middleware and applications are tightly coupled together and in the best case scenario devices or services, provided by different stakeholders are integrated through proprietaries APIs, connectors and protocols. Consequently application development process can hardly be opened to the large number of application designers and developers inhabiting the IoT Cloud ecosystem. Furthermore, until now the IoT Cloud infrastructure resource have been mostly provided as coarse-grained, rigid packages. The infrastructure components and software libraries are specifically tailored for the problem at hand and do not allow for flexible customization and provisioning of the individual resource components or runtime topologies. This "siloed approach" inherently hinders self-service, utility-oriented consumption of the IoT resources at a finer granularity levels, thus calling for a systematic approach to enable provisioning and governance of IoT cloud resources and capabilities. What is more, due to variety of involved stakeholder, as well as dynamicity, heterogeneity and geographical distribution of IoT Cloud, traditional provisioning and governance approaches are hardly feasible in practice. This is mostly because they implicitly make assumptions, such as physical on-site presence, manual logging into devices, understanding device's specificities, etc., which are difficult, if not impossible, to achieve in IoT Cloud systems. In spite of this, models, techniques and tools, which provide programmatic, conceptually centralized view on system provisioning and runtime governance are largely missing.

1.1 Problem Statement

The emerging IoT Cloud systems create numerous opportunities for a multitude of diverse stakeholders. However, such opportunities are accompanied with the aforementioned challenges, which open a broad field of research problems to be addressed. To narrow down and clearly define the scope of the thesis, this section describes a detailed problem statement. Firstly, in order to clarify the problem domain and its context, Section 1.1.1 outlines the most relevant concepts that are the main research subject of this thesis and gives a more detailed description of IoT Cloud systems. Secondly, Section 1.1.2 elicits key research questions, addressed in this thesis.

1.1.1 Problem context

The main aim of this work is to provide an ecosystem (comprising novel models, frameworks, tools and middleware) in order to facilitate the key tasks of application lifecycle, namely *development (programming), provisioning and governance, in the context of emerging IoT Cloud systems.* In this thesis, we define end-to-end provisioning in a broader sense [149], particularity involving a set of activities performed by developers and operations managers to prepare (infrastructure) resources and system/application artifacts, bringing an application to a state where it is usable for the end user, but also to guarantee continuous enforcement of such sate, e.g., in case of baseline deviations. Software governance serves as an umbrella term, encompassing a variety of management and controlling activities, risk management and regulatory or legal compliance measures imposed by high-level business objectives. In this thesis, we mainly focus on one particular subset of software governance, namely operational runtime governance (similar to [133, 70]), which deals with mapping the business objectives to concrete operational processes and enforcing such objectives through the operational processes during application runtime. At this point, it is also worth mentioning that testing, although one of the crucial tasks in the software development lifecycle, is out of the scope of this thesis, but it is a substantial part of our current research effort in the field of IoT Cloud systems.

Although there have been many attempts to define IoT Cloud systems [3, 160, 46, 134], there is still no general and commonly accepted definition of IoT Cloud systems neither in research nor in industry communities. In order to reach a common view on the problem domain, in the following IoT Cloud systems are described in more detail. Figure 1.1 shows a high-level architecture of IoT Cloud systems, that is adopted in this thesis, and depicts the key stakeholders. The bottom layer represents the Physical infrastructure, which comprises a variety of edge devices (e.g., sensors and gateways), network elements (routers and switches) and large data centers. In reality, the physical infrastructure is not flat and follows a hierarchical structure, where sensors and actuators are connected to the data centers via gateways, which are intermediary nodes that mediate the communication, but also provide (constrained) computational and storage resources. The communication between the Edge and the data centers is realized over heterogeneous networks which include wired, wireless and cellular communication channels. Moreover, contrary to the Cloud, IoT Cloud infrastructure is highly-decentralized and distributed among multiple geographical regions and organizations. As opposed to traditional cyber-physical and IoT systems, one of the distinguishing features of IoT Cloud systems is the Infrastructure virtualization layer. A number of existing approaches deal with the Edge devices virtualization and management, exposing them to the upper layers on different levels of abstraction. Most prominent approaches are centered around the Unikernels and kernel-supported virtualization such as Linux Containers (LXCs) or focus on higher-level data integration. The latter usually rely on semantic techniques and ontologies to mediate communication with heterogeneous edge-devices and integrate multitude of data formats, exposing them as event streams to the upper layers. The Middleware platform is the key part and in general its main responsibility is to provide a uniform representation of the underlying (virtual) infrastructure resources, enable management of such resources and accommodate applications with a runtime environment. This layer needs to provide mechanisms and tools for infrastructure provisioning, managing configuration models, deployment of application artifacts, as well as for governing the IoT Cloud applications and resources during the runtime. Finally, the Applications

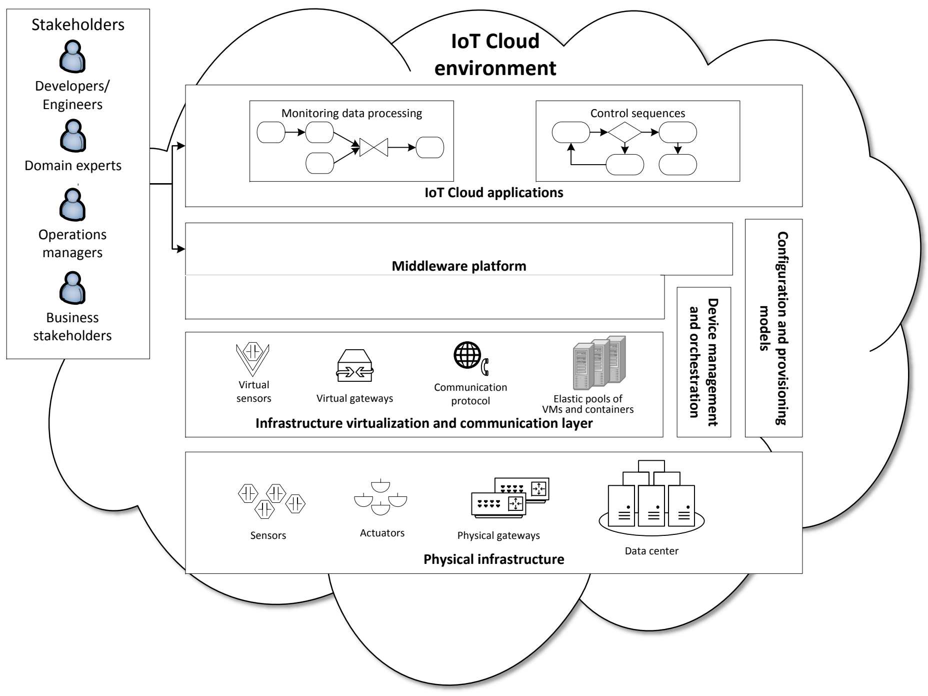


Figure 1.1: Overview of IoT Cloud Systems Architecture.

layer should provide programming models, capable to cater to different needs of the involved stakeholders (e.g., developers and domain experts), which is crucial in order to enable seamless development of IoT Cloud applications. With respect to the component presented in Figure 1.1, the work presented in this thesis mainly focuses on the *Middleware platform* and the development support for *IoT Cloud applications*.

1.1.2 Main research questions

The challenges identified in the beginning of this chapter serve as general motivation for the research conducted during the course of this thesis. Specifically, this work in detail addresses the following key research questions:

Q1: *What is a suitable programming model and methodology for developing novel IoT Cloud applications in an efficient, scalable and generic manner?*

The recent emergence of IoT Cloud systems has fostered proliferation of various application types mainly driven by urgent need to respond to volume, velocity and variety of data generated by IoT Cloud. In this thesis we mainly focus on one particular, yet general enough type of applications, i.e., *reactive IoT Cloud applications*[2]. In a very broad sense, such applications can be characterized by receiving (monitoring) information, e.g., sensory data, and as a response performing a sequence of (control) actions, e.g., on the physical entities (cf. Figure 1.1). However, independent of the particular application type, the aforementioned properties of IoT Cloud significantly impact application development process. Firstly, the development context of IoT cloud applications has grown beyond writing custom business logic components (e.g., services) that can be executed anywhere and requires considering capabilities and properties of the involved IoT devices. The main reasons for this are complex and strong dependence of the application business logic on specific capabilities of the underlying devices, novel resource features that need to be considered such as device location, and heterogeneity of the utilized IoT Cloud resources. In spite of this, the IoT cloud applications need to be generic, in the sense that their business logic needs to be expressed independently of the low-level device properties. Secondly, IoT Cloud applications execute in very dynamic, heterogeneous environments and interact with hundreds or thousands of both physical and virtual (Cloud) entities. Therefore, handling the data delivered by these entities, and controlling such entities in a scalable manner is another challenge for developers of IoT Cloud applications. This is mainly because they need to be able to dynamically identify the scope of application's actions, depending on the task at hand, which is in contrast to the majority of contemporary approaches that require explicit interactions with individual devices. Thirdly, due to variety of involved stakeholders (e.g., domain experts, high-level business logic developers and operations managers), who are usually dispersed across different teams or organizations, a suitable programming model should provide different logical views on the application development process, while retaining

[2]In this thesis the term IoT Cloud applications is used to refer to reactive IoT Cloud applications, unless otherwise stated.

a uniform view on the application artifacts. Unfortunately, developers currently lack suitable programming abstractions to deal with such concerns in a unified manner, from early stages of application development.

Q2: *Which provisioning models, techniques and tools can be applied to enable on-demand, self-service provisioning of IoT Cloud resources at fine granularity?*

When facing large-scale systems with heterogeneous, dynamic and geographically distributed resource pool, efficacy of provisioning models, mechanisms and tools plays a crucial role. With the rise of cloud computing, we have witnessed numerous benefits of self-service, utility-oriented resource consumption models, in terms of more flexible and cheaper IT operations [90, 23]. Since cloud is a one of the key constituents of IoT Cloud, it is natural to expect that these flagship properties of cloud computing are inherited by IoT Cloud as well. Unfortunately, this is still not the case and as a consequence, in many situation we are facing IoT and Cloud systems, which are rather task-specific hybrids, instead of native IoT Cloud systems. Currently in such systems, domain-specific application requirements drive the design of all system components and determine most technical choices, ranging from hardware components to application behavior, thus the IoT Cloud infrastructure is still consumed as a rigid, single-purpose "utility". Also from technical perspective, cloud computing has provided many advances in both infrastructure, platform and application provisioning (stipulating development of novel provisioning approaches, mechanisms and tools), but due to inherently different nature of IoT Cloud (cf. Section 1.1.1) many of these concepts cannot be simply "reused" for IoT Cloud systems. Over the last years, this has lead to a large body of work in both academia and industry [134, 160, 3, 43], which has provided solid foundations (especially on the lower-levels, e.g., hardware virtualization or network management) for IoT Cloud provisioning. However many challenges, both conceptual and engineering, still remain. One of the main concerns a is genuine lack of provisioning models, specifically tailored to treat both Edge and Cloud resources in a unified manner and enable uniform interactions with large pool of IoT Cloud resources, e.g., via well-defined APIs. To date, the IoT Cloud infrastructure resources are mostly provided as coarse-grained, rigid packages. The infrastructure components and software libraries are specifically tailored for the problem-at-hand and do not allow for flexible customization of individual resource components or runtime topologies. This requires rethinking existing support for representing infrastructure resources, managing their configuration and deployment models as well as composing low-level resource components into usable infrastructures, capable to support novel business logic requirements. The issue is further exacerbated due to strong dependencies of the business logic on infrastructure resource capabilities, as discussed above, that prevent consuming IoT Cloud infrastructure as traditionally generic compute or storage resources. Unfortunately, this is only one part of the problem, because existing provisioning support, in terms of available tools and frameworks, discriminates against inherent properties of IoT Cloud infrastructures such as heterogeneity, geographical distribution, and the sheer scale of such infrastructures. System integrators and operations managers have to rely on provisional solutions, which require combining multitude of provisioning techniques such as manual, script- and service-based provisioning. Many of these approaches blindly assume physical on-site presence or manual logging into edge devices, making them hardly feasible in practice. Therefore, suitable provisioning and deployment framework needs to provide elastic and scalable provisioning solutions, which can automate provisioning tasks to a large extent, while taking into account resource-constrained nature of the edge devices. The provisioning tools need to provide a logically centralized view on IoT Cloud infrastructure, which is crucial for realizing consistent infrastructure base-line across the entire IoT Cloud and for coping with highly distributed nature of such infrastructures.

Q3: *Which models, techniques and tools are required to achieve structured and systematic IoT Cloud governance?*

Wide and ever-stronger growing application area of IoT Cloud systems, e.g., in the context of smart cities, has lead to stronger interplay and entanglement among variety of diverse stakeholders (both business and technical). Various domains increasingly rely on IoT Cloud resources and capabilities to optimize their key business tasks, improve efficiency of processes and quality of life. As a consequence, IoT Cloud systems are becoming an integral part of many existing business models and a key enabler for new business opportunities and cross-domain applications. This has spurred numerous governance initiatives and approaches that deal with safety issues, legal, compliance, and data privacy concerns [44, 51, 152], mainly due to their potential impact on the multitude of the involved stakeholders. Unfortunately, vast majority of contemporary approaches draw a hard line between high-level governance objectives (that mainly concern business stakeholders) and operations processes, which concern technical stakeholders such as operations managers that need to implement concrete operations processes, conforming to the high-level governance objectives. Therefore, at the moment there is a wide gap between different stakeholders involved in governing IoT Cloud systems, increasing the risk of lost requirements or causing over-regulated systems, potentially incurring higher operation costs or limiting business opportunities. For example, according to Gartner [57]: "Through 2015, 80% of outages impacting mission-critical services will be caused by people and process issues...". Moreover, current approaches in IoT governance usually addresses the *Internet* part of the IoT, e.g, in the context of the Future Internet services[3], while operations processes mostly deal with *Things* (e.g, [32]) as additional resources that need to be operated. Governance objectives (law, compliance, etc.) are not easily mapped to operations processes (e.g., configuring sensory data streams or adding/removing devices) and, in practice, bridging the gap between governance and operations management of IoT cloud systems poses a significant challenge for the involved stakeholders, mainly due to the large number of such stakeholders, novel requirements and regulations for shared IoT Cloud resources and their geographical distribution, usually across different areas of jurisdiction. What is more, even with perfectly aligned governance objectives, designing and realizing

[3]http://ec.europa.eu/digital-agenda/en/internet-things

operational governance processes [133, 70], posses a significant challenge. Due to dynamicity, heterogeneity, geographical distribution, and the large scale of IoT cloud systems, traditional approaches to realize even basic operational governance processes are hardly feasible in practice. This is mostly because such approaches implicitly make assumptions such as traditional interaction patterns with novel resource types and perfect understanding of underlying infrastructure, at the same time neglecting numerous uncertainties inherently present in the IoT Cloud systems, caused by novel interactions of IoT elements, network elements, cloud resources and humans. Therefore, due to a lack of systematic approaches for operational governance in IoT Cloud systems, operations managers currently have to rely on ad-hoc solutions to deal with the characteristics and complexity of IoT cloud systems when performing operational governance processes. Moreover, supporting models, tools and mechanisms for runtime governance of IoT cloud systems remain largely undeveloped, thus putting much of the burden on operations managers to perform operational governance processes in practice.

1.2 Scientific Contributions

The main goal of this thesis is to respond to the previously identified research challenges by presenting the most important results of our research in the emerging field of IoT Cloud systems. Generally, the conducted research has lead to development of novel models, techniques and components of a general framework for developing, provisioning and governing IoT Cloud systems and applications. Concretely, the main contributions of this thesis include:

C1: *Programming models, middleware and methodology for developing IoT Cloud applications*

The first contribution of this thesis deals with conceptualizing and implementing programming models and middleware, specifically tailored to cater to the novel requirements and challenges imposed by IoT Cloud applications. Due to significant network, compute and storage requirements, the IoT Cloud applications are bound to utilize all the available infrastructure resources ranging from large data centers to the constrained IoT devices, e.g., gateways. Thus, the main driving force behind this contribution was to provide a set of programming abstractions and runtime mechanisms that enable more efficient, scalable and intuitive development of generic IoT Cloud applications, which seamlessly utilize both the Edge and the Cloud. Moreover, to account for the variety of involved stakeholders and the complexity of software stack, ranging from embedded devices development to high-level IoT Cloud applications, our approach is designed in such manner to provide multiple logical views on the application development process, while retaining a uniform view (in code) on the produced application artifacts. More specifically, this contribution comprises: a) *A programming model and a runtime for IoT Cloud gateways and domain libraries (Data* *and Control Points);* b) *A higher level programming model and a middleware for IoT Cloud applications (Intents and IntentScopes);* c) *A unifying programming framework for IoT Cloud systems, based on everything-as-code paradigm.* This contribution has been originally published in [106, 107, 109].

The first part of this contribution introduces support for programming IoT Cloud gateways, which reside at the edge. This generally involves developing common monitor and control tasks. For example, monitor tasks can include processing, correlation and local filtering of sensory data streams. These tasks are main constituents of domain libraries that form the cornerstone for building higher-level (cloud) services, as we describe subsequently. To enable development of such monitor and control tasks, we propose two main abstractions, which are exposed to developers, namely Data- and Control Points. Generally, they represent low-level data and control channels to the sensors/actuators in an abstract manner and mediate the communication with the connected devices (e.g., digital, serial or IP-based), by implementing the necessary communication protocols, e.g., I^2C, CAN or SOX/DASP. Compared to numerous other approaches that address similar issues [5, 16, 28, 1, 41, 66, 69, 131], with the Data- and Control Points we put a special focus on enabling customization, flexible configuration and virtually exclusive access to underlying device channels, which are still not fully addressed in the literature. To this end, the supporting middleware provides mechanisms which act as multiplexers of the data and control channels, thus enabling the device services to have their own view of and define custom configurations for such channels, e.g., sensor poll rates or data stream filters. Finally, by providing an illusion of an exclusive access to the underlying devices, our middleware supports execution of multiple applications within a single IoT gateway. The second part of this contribution mainly addresses development and execution of the cloud counterpart of IoT Cloud applications, e.g., cloud services. Introduced programming model defines high-level programming constructs and operators, which raise the level of programming abstraction, enabling developers to implement IoT Cloud applications without worrying about the specifics of diverse underlying edge devices. The core of this programming model revolves around the notion of Intent and IntentScope. Intents are high-level representation of the aforementioned monitor and control tasks. They allow the developers to communicate to the supporting middleware what needs to be done, instead of worrying how the underlying devices will perform a specific task, effectively shielding the developers from the complexity of IoT controls and complex data processing schemes. By supporting dynamic binding of the intents and tasks, the underlying middleware supports development of loosely coupled applications that are independent of the specific task implementation and guarantees stability (e.g., backward compatibility) of developed applications. The IntentScope is an abstraction, which represents a group of physical entities that share some common properties, e.g., context. Our programming model introduces a number of operators, which enable the developers to define/refine IntentScopes. IntentScopes allow for dynamically delimiting the scope of applications actions, without manually referencing individual physical entities. This is one of the key preconditions to achieve development of generic IoT

Cloud applications in a scalable manner. Whereas the Data- and Control Points are mainly intended for domain experts, this programming model offers a different logical view on the application development process and is mainly meant to be used by the developers of higher-level business logic. However, these benefits come at a cost, i.e., at this level we trade flexibility and expressiveness for a scalable, more intuitive and efficient programming of IoT Cloud applications. Finally, as the third part of this contribution, we have developed a comprehensive programming framework for IoT Cloud applications. The main goal of this framework is to address the growth (in scope and complexity) of the application development context, by providing a unified, programmatic view for the entire development process (everything as code). To this end, besides enabling application business logic development, the framework introduces additional support for programmatic application provisioning and governance. The framework encapsulates most important aspects of IoT Cloud applications provisioning and governance, exposing them to the developers in terms of uniform APIs and light-weight provisioning and governance DSLs. By providing a systematic and structured support for everything-as-code paradigm our framework makes the entire application development process more traceable and easily auditable, but also enables exploiting proven and well-known technologies, e.g., source control or configuration management systems, during the entire application lifecycle. It is important to mention that this contribution mainly focuses on application-level support for programmatic provisioning and governance, but it utilizes the provisioning and governance models for IoT Cloud, which are subject of other contributions of this thesis. We describe them in more detail subsequently. The proposed programing models and middleware have been validated and evaluated on a set of real-life applications, in domains of building management system and vehicle management system[4].

C2: *Provisioning model and a middleware infrastructure for provisioning IoT Cloud systems.*

In the second contribution of this thesis the focus is shifted from application-level support to IoT Cloud Middleware Platform, in order to address the lack of adequate conceptual and tooling support for IoT Cloud provisioning. The conceptual model (cf. Figure 1.1) of IoT Cloud systems is based on a layered architecture, which assumes a middleware platform to expose the underlying infrastructure in a unified manner, which can be customized and configured to support execution of application business logic. However, this is still not the case, mainly because in IoT Cloud systems, in addition to Cloud, Edge devices (e.g., gateways) are also first-class execution environments, which are still largely underutilized and mainly integrated with the Cloud in an ad-hoc fashion. This contribution is mainly driven by a stringent need: To enable refactoring of the underlying infrastructure into finer-grained resource components whose behavior can be defined in software; To provide conceptually unified representation of both Edge

[4]The applications and their requirements are derived from a case study that was conducted in collaboration with our industry partners within P3CL lab. The main outcomes of the case study are presented in Section 2.

and Cloud resources; As well as to enable automated and scalable management of IoT Cloud infrastructures and their configuration models in a logically centralized fashion. To this end, we build on existing hardware virtualization approaches such as OS-level virtualization and kernel-supported virtualization, and infrastructure automation solutions, i.e., Infrastructure-as-Code (IaC), and explore how advance engineering approaches, most notably software-defined principles, can be adapted and applied in the context of IoT Cloud. The main parts of this contribution include: a) *A unified provisioning model and framework support for logically centralized provisioning of IoT Cloud systems;* b) *A middleware infrastructure for utility-based provisioning of IoT Cloud systems.* This contribution has been originally published in [105, 146, 108].

The first part of this contribution introduces a conceptual model of software-defined IoT Cloud systems and a preliminary implementation of supporting provisioning framework. The core concept of the model model are software-defined IoT units, i.e., within our model the IoT Cloud resources (e.g., virtual sensors), their runtime environments (e.g., gateways) and capabilities (e.g., communication protocols or data point controllers) are described as software-defined IoT units. Such units are used to encapsulate the IoT Cloud resources and abstract their provisioning in software. To this end, the software-defined IoT units expose well-defined APIs and they can be composed at different levels, creating virtualized runtime infrastructures for IoT Cloud applications. To technically realize our unit model we introduce a concept of unit prototypes. The unit prototypes are hosted in the IoT Cloud and enriched with provisioning capabilities (delivered by framework's provisioning agents), that allow them to be dynamically configured, interconnected, deployed and controlled. In our work, we do not introduce novel virtualization solutions, but rely on proven technologies, namely kernel-supported virtualization and inversion of control concepts to provide generic unit prototypes. The unit prototypes along with a number of example software-defined IoT units are implemented and provided as stock components within the provisioning framework. The framework also provides mechanisms to support automated composition of the software-defined IoT units and centralized management of infrastructure configuration models, thus enabling flexible customizations and simplifying the provisioning of IoT Cloud infrastructures. The second part of this contribution conceptually extends and technically refines our provisioning model and framework, by introducing a middleware infrastructure for provisioning IoT Cloud systems. Its main objective is to provide a comprehensive support for scalable, multi-level provisioning of IoT Cloud systems, in order to support execution of provisioning processes that are based on the aforementioned provisioning model. The middleware comprises a cloud-based provisioning controller and edge-based provisioning agents and deamons. The controller architecture and the provisioning mechanisms are specifically tailored to account for the large-scale of IoT Cloud, but also for the resource-constrained nature of Edge devices. To this end, the middleware controller supports elastically scalable execution of the provisioning processes and the light-weight provisioning agents and deamons enable virtualizing compute resources of edge devices, as well as provide support execution of local installation and configuration directives that are specified within the provisioning processes. The main mechanisms of the middleware include: i) A light-weight mechanism for resource abstraction (software-defined gateway), which allow for application-specific customizations of IoT Cloud resources. ii) Support for automated provisioning and management of infrastructure resources, application components and configuration models; in a uniform, logically centralized manner through middleware-managed APIs; iii) Extensible and flexible provisioning models, which support on-demand consumption of the Edge-device resources. The main advantage of our middleware is reflected in facilitating on-demand, self-service resource consumption by providing flexible provisioning models and support for uniform, logically centralized provisioning of Edge devices, application artifacts and their configuration models. Our middleware also enables application-specific customization of Edge devices through software-defined gateways and well-defined APIs, while preserving the benefits of proven virtualization techniques. We experimentally show that our middleware enables scalable execution of provisioning tasks across relatively large IoT Cloud resource pool and at the same time its overhead in terms of resource consumption is suitable for resource-constrained devices. Generally, the second contribution of this thesis lays a cornerstone towards realizing our vision of utility-based provisioning of IoT Cloud systems. To validate and evaluate our approach we have developed an IoT Cloud infrastructure and installed it in our department. It comprises 15 physical gateways (based on Raspberry Pi[5]) and around 4000 virtualized gateways, deployed in our local OpenStack[6] cloud. The cloud-based part of testbed relies on Linux Containers (LXC), which are used to virtualize and mimic physical gateways based on a snapshot of a real-world gateway, developed by our industry partners.

C3: *GovOps model and a runtime framework for automated governance of IoT Cloud systems.*

The third contribution of this thesis introduces GovOps (Governance and Operations) – a novel methodology and framework for governing IoT Cloud systems. GovOps' primary focus is on enabling design, implementation and execution of operational governance processes ([133, 70]), which represent a subset of general IoT Cloud governance, and deal with enforcing high-level governance objectives through operations processes at runtime. The main incentive for introducing GovOps is to bring business stakeholders and operations managers closer together, in order to make a step forward in bridging the gap between governance objectives (e.g., standards and regulations) and operations processes. Conceptually, the main objective of GovOps is twofold. On the one side it aims to enable seamless integration and alignment of high-level governance objectives and strategies with executable operations processes, by incorporating the key aspects of both high-level governance and operations management from early designing stages. On the other side, it strives to support performing operational governance processes for IoT Cloud systems in such manner they are feasible in practice, considering the previously-described challenges. To this end, three main parts constitute this contribution:

[5]https://www.raspberrypi.org/
[6]https://www.openstack.org/

a) *A GovOps methodology and a reference model for operational governance processes;* b) *A GovOps runtime framework for operational governance processes;* and c) *An uncertainty extension for GovOps framework and a DSL for developing uncertainty- and elasticity-aware GovOps processes.* This contribution has been originally published in [104, 110, 103]

As the first part of this contribution we introduce a methodology for developing GovOps strategies, its main design principles, concepts and roles. Conceptually, GovOps reference model builds on the previously introduced software-defined IoT units and introduces fundamental elements for developing operational governance processes (GovOps processes). Within GovOps model, the main building blocks of the GovOps processes are governance capabilities. They represent operations which can be applied on IoT Cloud resources, e.g., query current version of a software, dynamically change communication protocol, and spin-up a virtual gateway. These operations manipulate IoT Cloud resources in order to put IoT Cloud system into a specific (target) state, that conforms with governance objectives. Furthermore, we outline the main steps of GovOps design process and introduce a novel role, GovOps manager, who is responsible to guide and manage designing the GovOps processes, because in practice it is very difficult, risky, and ultimately very costly to adhere to the traditional organizational silos that strictly separate business stakeholders from operations managers. The second part of this contribution introduces a runtime framework for implementing and executing the GovOps processes. The main aim of the GovOps framework is to provide logically centralized point of governance, i.e., to enable conceptually centralized interaction with an IoT Cloud system and provide a unified view on the system's governance capabilities through framework's governance controller, while retaining fine-grained control over the IoT Cloud resources (e.g., gateways) through the governance capabilities. Further, by introducing light-weight governance agents, our framework strives to provide a higher degree of autonomy to the underlying devices and enable automated execution of GovOps processes, making them easily enforceable and repeatable across large-scale resource pool. Generally, the framework supports GovOps managers to handle two main tasks. First, the GovOps framework enables dynamic, on-demand injection of capabilities into IoT cloud resources, and supports coordinating the dynamic profiles of these resources at runtime. Second, our framework allows for runtime management of governance capabilities throughout their entire lifecycle that, among other things, includes remote capability invocation and managing dynamic APIs, which are exposed to users. By providing suitable runtime mechanism, GovOps framework supports the key steps of the aforementioned tasks to be performed transparently to GovOps managers and the GovOps processes. The only thing that such processes observe are API calls and the corresponding responses. At this point it is important to mention that, from technical perspective, GovOps does not make any assumptions about the implementation of the GovOps processes, in the sense that such processes can be realized as business processes (e.g., using BPMN), via a Domain Specific Language (DSL), or even as dedicated governance applications or services. In addition, GovOps does not impose any constraints on formalizing high-level governance objectives. To this end, there is an abundance of governance models and accountability frameworks, such as the 3P [128], CMMI [2] or COBIT [63], that conceptually complement GovOps to support managing governance objectives and coordinating decision making processes. Finally, the third part of the contribution introduces an uncertainty extension for the GovOps framework. To this end, we define a DSL to support development of uncertainty- and elasticity-aware GovOps processes, together with supporting runtime mechanisms for uncertainty mitigation. To achieve elasticity-awareness of GovOps processes, the introduced DSL exposes typical elasticity controls at the level of GovOps processes. For this purpose, we have refined and extended rSYBL [34], which is a cloud-based, muliti-level elasticity controller and integrated it with the GovOps controller. The presented DSL supports uncertainty-awareness in GovOps processes by providing language constructs that enable configuring and parameterizing such processes with uncertainty information. Inspired by the well-established fault, error, failure classification [10] and the general belief model [122], we have developed an uncertainty taxonomy for IoT Cloud systems. Based on this taxonomy we have derived a set of uncertainty families, which are used to systematically classify uncertainties, analyze their effects on typical GovOps processes and enable deriving requirements, actions and configuration models needed for streamlining the uncertainty management. Finally, to approach the intrinsic bootstraping problem in the uncertainty management, we have provided comprehensive runtime mitigation mechanisms for two most relevant uncertainty families, related to data quality (incomplete and missing data) and actuation dependability aspects of GovOps processes. The GovOps approach and its feasibility to govern large-scale IoT Cloud systems is evaluated on the aforementioned testbed that was developed as a part of the second contribution. In the experiments we have used the real-life applications, developed for the purposes of the first contribution (C1), together with their governance objectives and requirements derived in the aforementioned case study[4].

1.3 Organization of the Thesis

This dissertation includes the contributions from original research papers that were published during the author's doctoral studies. The individual contributions have been re-worked, extended and presented in a unified context.

The reminder of the thesis is organized as follows: Chapter 2 presents the background information and a case study, which introduces two motivating scenarios used throughout the thesis. Chapters 3 to 9 introduce the main contributions of this thesis. These chapters are further divided into three parts. Part I (chapters 3-5) deals with programming IoT Cloud systems. Chapter 3 introduces a high-level programming model and a runtime for developing cloud-centric IoT Cloud applications. Chapter 4 introduces a programming model and a runtime for resource-constrained Edge devices, e.g., gateways. In Chapter 5, we introduce a unifying programming framework for IoT Cloud systems, based on everything-as-code paradigm. Part II (Chapter 6 and Chapter 7) deals with provisioning IoT Cloud systems. Chapter 6 introduces a unified provisioning model (based on software-defined principles) and a framework support for logically centralized provisioning of IoT Cloud systems. In Chapter 7, we introduce a middleware infrastructure for utility-based provisioning of IoT Cloud systems. Part III (Chapter 8 and Chapter 9) deals with governing IoT Cloud systems. Chapter 8 introduces GovOps – a methodology and a reference model for operational governance processes in IoT Cloud systems. The same chapter also also presents a runtime GovOps framework for implementing and executing operational governance processes in large-scale IoT Cloud systems. In Chapter 9, we introduce an uncertainty extension for GovOps, which provides a declarative policy language and a runtime for developing uncertainty- and elasticity-aware governance processes. Finally, Chapter 11 concludes the thesis with a reflection on the main research questions and an outlook of the future research.

CHAPTER 2

Case Study & Background

2.1 Case Study Scenarios

To better motivate our work and ground the previously-described research questions (cf. Chapter 1) in realistic IoT Cloud systems, we conducted a detailed case study in Pacific Controls Cloud Computing Lab[1] (PC^3L). The case study was carried on in a tight collaboration with our industry partners[2] and its main aim was to provide a detailed analysis of the major development, operations and governance tasks in real life IoT Cloud systems and applications. Further, the case study has elicited the main requirements for such tasks and identified concrete research challenges that currently hinder realizing those tasks in practice.

This section presents two scenarios in the context of IoT Cloud that were the main outcomes of our case study. These scenarios are referred to throughout the thesis where they are discussed in more detail. Subsequently, we give a brief overview of the main features and requirements of two real life systems: Fleet Management System (FMS) and Building Management System (BMS), which are the core of our scenarios. In the subsequent chapters, we analyze these systems from different perspectives, depending on a concrete research challenges that are emphasized and addressed in the respective chapters, as well as highlight design and implementation details of the relevant (sub)systems and applications[3].

2.1.1 Fleet Management System (FMS)

Fleet Management System (FMS) is a real life IoT Cloud system responsible for managing fleets of zero-emission, electric vehicles deployed worldwide (cf. Figure 2.1). For our

[1]http://pcccl.infosys.tuwien.ac.at/
[2]http://pacificcontrols.net/
[3]Some of the proprietary algorithms are only described on a high-level, to protect business information.

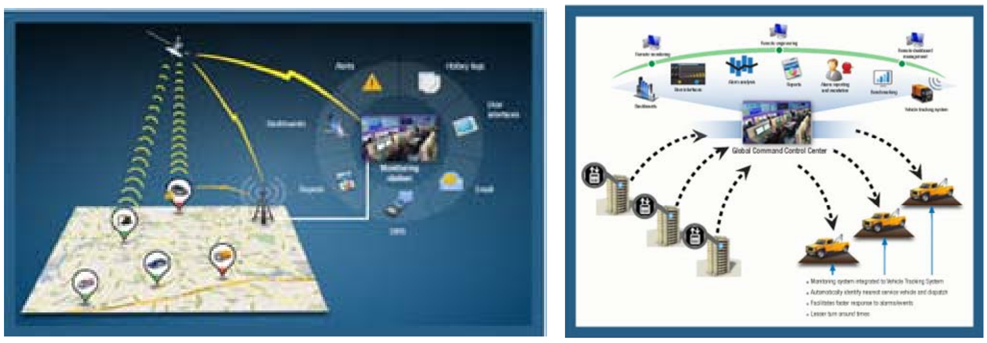


Figure 2.1: The Fleet Management System[4].

discussion the most important functionality of the FMS is management of the electric vehicles on different golf courses. In general, the FMS supports the involved stakeholders (described in the following) to remotely manage the fleet vehicles dispersed among numerous golf courses in different parts of the world in order to optimize tasks, crucial for their respective business processes.

Electric vehicle information systems consist of on-board gateways and software platforms that are tightly coupled with the vehicles and their usage scenario – mainly as golf cars. The on-board gateway interfaces the underlying vehicle assets via different protocols such as LIN[5]), CAN[6] (engine) or OBC[7] (charger) and provides execution runtime for features such as: a) vehicle maintenance (fault history, battery health, crash history, and engine diagnostics), b) vehicle tracking (position, driving history, and geo-fencing), c) vehicle info (charging status, odometer, serial number, and service notification), d) set-up (club-specific information, maps, and fleet information). For legacy cars that are not equipped with such gateways, a device acting as a CAN-IP bridge is used (e.g, Teltonika FM5300[8]). In this case FMS hosts virtual gateways on the cloud that execute the aforementioned services on behalf of the vehicles.

These vehicles communicate with the Cloud via 3G or Wi-Fi networks (via local hot spots) to exchange telematic and diagnostic data. On the Cloud, FMS provides different applications and services to manage this data, as well as to respond to changes in vehicle operation and environment in terms of performing control actions on the physical assets (e.g., vehicles) or generating notifications in near realtime. Examples of cloud services and applications include: a) Near realtime vehicle status: location, driving direction, speed, vehicle fault alarms; b) Remote diagnostics: equipment status, battery

[4]Figure 2.1 and Figure 2.2 are obtained for our industry partner's website. URL:http://www.pacificcontrols.net/.

[5]http://cs-group.de/fileadmin/media/Documents/LIN_Specification_Package_2.2A.pdf

[6]http://kvaser.com/software/7330130980914/V1/can2spec.pdf

[7]http://gptechgroup.com/pdf/OBC-353.pdf

[8]http://teltonika.lt/en/pages/view/?id=1024

health and timely maintenance reminders; c) Remote control: overriding on-board vehicle control system in case of emergency or theft, e.g. to prevent a car to leave a golf-course; d) Batch configuration and software updates: remotely configure course maps, course information and software feature set for a fleet, applying software updates, and installing new value-added services to a large number of cars; e) Fleet upkeep: service history and fleet usage patterns.

The FMS is currently used by the following three types of stakeholders: vehicle manufacturers, distributors, and golf course managers. These stakeholders have different business models. For example, when a manufacturer only leases vehicles to customers, they are interested in the status and upkeep of the complete fleet, will perform regular maintenance, as well as monitor crashes and battery health. Golf course managers are mostly interested in vehicle security to prevent misuse and ensure safety on the golf course (e.g., using geofencing features). In general, the stakeholders rely on the FMS and its services to optimize their respective business tasks. Therefore, they also have different requirements in terms of applications, resource provisioning and general system governance requirements. We discuss the FMS in more detail in Chapter 3, Chapter 5 and Chapter 8.

2.1.2 Building Management System (BMS)

Building Management System (BMS) is an IoT Cloud control system for buildings, which enables remote monitoring and controlling of buildings' mechanical and electrical assets and equipment such as HVAC, lighting, elevators, pluming and fire alarm systems (cf. Figure 2.2). In general, it connects the buildings' assets to a cloud-based platform, which provides applications for centralized management of such assets. Some of the core features of the BMS include managing the environment temperature, CO_2 emission and humidity within a building, as well as optimizing the building's energy consumption and handling predictive maintenance. For example, the climate control services are responsible to control the production of heating and cooling, managing air distribution systems throughout the building, and locally controlling the air mixture to achieve the desired environment temperature. Country to the FMS, the BMS is less dynamic and has a smaller degree of geographical distribution. In spite this, it is a large scale systems that supports operating several thousands buildings.

The BMS relies on hierarchical IoT Cloud infrastructure in which the building sensors are connected to gateways (intermediary nodes installed throughout the buildings) via wired or wireless communication channels. These gateways are mainly based on Sedona[9] or Niagara^{AX}[10] Edge devices and they provide constrained execution environments for light-weight BMS device services. The upstream communication with the cloud platform

[9]`http://sedonadev.org/`
[10]`http://niagaraax.com/`

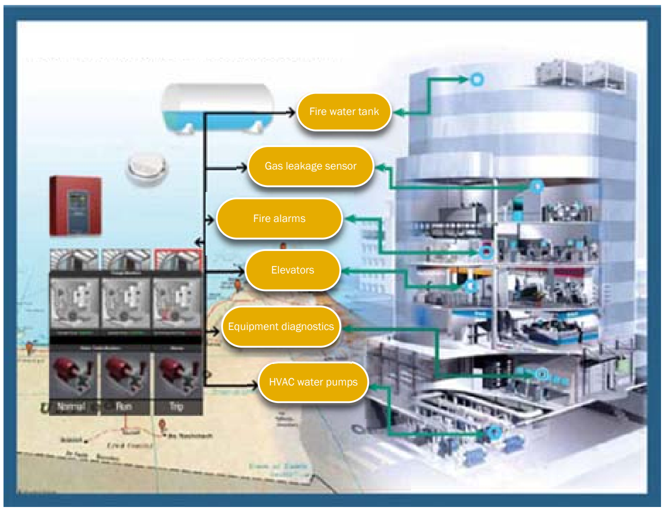


Figure 2.2: The Building Management System.

is realized via ModBus[11], BACnet[12], CoAP[13] or MQTT[14] protocols and the collected sensory data is either polled periodically from the gateways (during routine checks) and analyzed by cloud services off-line or the gateways push the data (alarms) to the cloud, e.g., in case of emergency such as fire.

For safety-critical services, e.g., alarm handling, timely processing of the events and the availability of the BMS play a crucial role. While for such safety-critical services real-time delivery and processing is essential, for services such as HVAC controller, cost reduction is more important. Due to the multiplicity of the involved stakeholders such as building managers, building owners, residents and civil service authorities, the BMS needs to allow for flexible runtime customizations (in terms of resources provisioning and governance) in order to exactly meet the stakeholder's functional or compliance requirements. This largely depends on the problem at hand and availability or accessibility of the assets, as well as desired system's non-functional properties. We discuss the BMS in more detail in Chapter 6 and Chapter 9.

2.2 Background

In this section, we provide an overview of the well-established models, concepts and technologies, which were used as groundwork for the work presented in this thesis. We mainly focus on the most important background research and industrial advances in the areas of cloud computing, Internet of Things, software-defined systems and DevOps.

[11]http://modbus.org/specs.php
[12]http://bacnet.org/Bibliography/EC-9-97/EC-9-97.html
[13]http://tools.ietf.org/html/rfc7252
[14]http://mqtt.org/documentation

Internet of Thing

The term Internet of Things (IoT) has been originally coined by Kevin Ashton in 1999 to refer to a global network of Internet-connected objects attached with a Radio Frequency Identification (RFID) chips. Since then the definition and application field of IoT has significantly expanded to offer connectivity of devices and services that goes beyond traditional machine-to-machine (M2M) [68] and covers variety of applications and domains, including smart homes [75, 26], smart healthcare [46, 21], smart transportation [65, 86], smart grids [159, 20], and eventually smart cities [161, 67, 102]. The Global Standards Initiative on Internet of Things (IoT-GSI) gives a definition of IoT: *"... as a global infrastructure for the information society, enabling advanced services by interconnecting (physical and virtual) things based on existing and evolving interoperable information and communication technologies."* [73]

Recent advances in the IoT have provided solid foundations for a global infrastructure of networked physical entities, able to monitor and control their physical status and the surrounding environment, as well as to expose themselves via data streams and services over the Internet [82]. Such vast IoT infrastructure enables novel applications and systems to utilize IoT resources in order to deliver novel value-added services, which leverage data from different sensor devices or enable timely propagation of decisions, crucial for business operation to the edge of the infrastructure, resulting in improved efficiency, accuracy and economic benefit [144, 145, 96, 31].

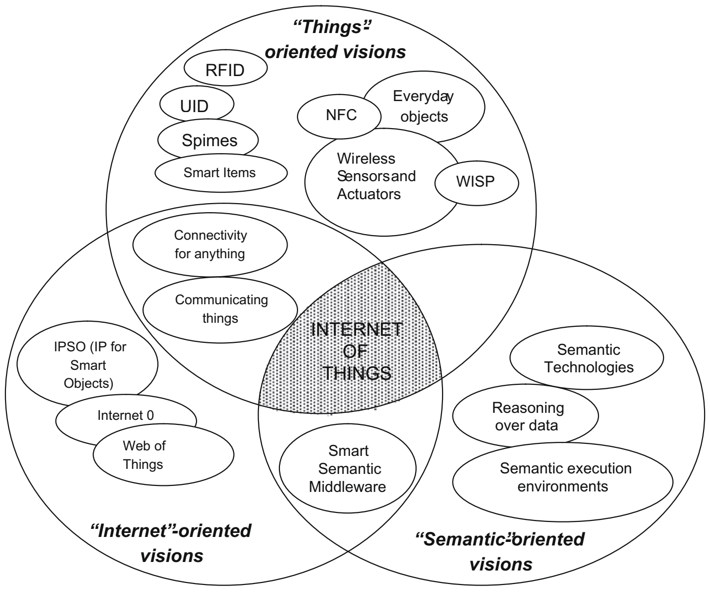

"Things"-oriented visions
RFID
UID
Spimes
Smart Items
NFC
Everyday objects
Wireless Sensors and Actuators
WISP
Connectivity for anything
Communicating things
INTERNET OF THINGS
IPSO (IP for Smart Objects)
Internet 0
Web of Things
Smart Semantic Middleware
Semantic Technologies
Reasoning over data
Semantic execution environments
"Internet"-oriented visions
"Semantic"-oriented visions

Figure 2.3: Different visions and approaches to the Internet of Things (taken from [9]).

Wide application area and numerous benefits of IoT have spurred various research and industrial efforts exploring the IoT. This has resulted in a number of different approaches and visions to realize systems which utilize the capabilities and data provided by the IoT (cf. Figure 2.3). For example, emphasizing on network aspects of IoT different approaches have provided a wide variety of communication protocols, which support diverse communication patterns (e.g., message-based, REST-like, mash-up and opportunistic networks) among the connected things, usually optimized for a specific problem or application domain. Further, thing-centric approaches address some of the crucial issues such as uniquely identifying the connected IoT devices and abstracting the things, e.g., via SOA-based techniques. Although mainly industry-driven, a number of approaches sharing the things-centric vision have developed different IoT gateways. Such gateways are an integral part of the global IoT infrastructure and are mainly used to connect the low-level devices (sensors and actuators) to the Internet, but also to provide (constrained) processing and storage resources for IoT applications. Mainly focusing on exploiting the Big Data, generated by the IoT, a large body of work has recently emerged. It has resulted in a variety of approaches to mediate the heterogeneous data formats, analyze and reason over the IoT data, e.g., by utilizing semantic-based techniques and large-scale data processing frameworks.

The work presented in this thesis does not adopt any of the proposed visions per se, instead it relies and builds on different approaches/enablers (models, protocols and frameworks) to address common problems related to programming, provisioning and governance of systems/applications that utilize the IoT.

Cloud Computing

Over the last decade, cloud computing has put itself forward as one of the most important paradigms for delivering and consuming digital resources [6, 90], mainly due to utility-driven, on-demand nature of cloud offerings, which allow customers to elastically provision the exact type and amount of resources needed for a given task at a given time. The National Institute of Standards and Technology (NIST) defines cloud computing as follows: *"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."* [121].

One of the most important features of cloud computing is reflected in its support for utility-based delivery and consumption of digital resources. Therefore, cloud computing generally refers to a broader (business) model instead of a concrete technology stack supporting this paradigm. In spite of this, over the recent years a general, high-level view on the cloud computing has emerged, involving three core "layers": Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS), as shown in Figure 2.4.

The IaaS is the lowest of the aforementioned layers and it offers the most flexible usage of Cloud resources. Generally, it enables dynamic provisioning of compute, storage and network resources. One of the first and still predominant public offerings of IaaS is

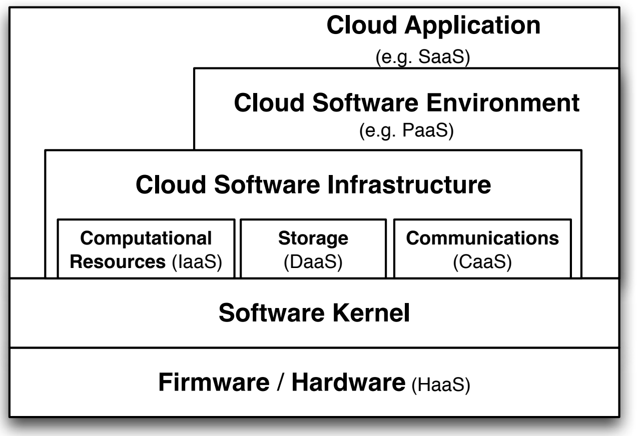

Cloud Application
(e.g. SaaS)
Cloud Software Environment
(e.g. PaaS)
Cloud Software Infrastructure
Computational Resources (IaaS)
Storage (DaaS)
Communications (CaaS)
Software Kernel
Firmware / Hardware (HaaS)

Figure 2.4: High-level overview of cloud computing. (taken from [158]).

Amazon's Elastic Compute Cloud[15]. There is also a number of open source IaaS solutions such as OpenStack[16] which can be for example utilized as private IaaS. Independent of the usage model, virtualization technologies such as Xen [15], VMWare ESX [147] or KVM [79], play the crucial role, by supporting multiple VM's to concurrently execute on a single host, transparent to the users and applications. This is achieved by isolating each VM's kernel, filesystem, network stack, as well as other OS parts. Recently more light-weight kernel-supported vitalization solutions have been receiving a lot of attention. This solutions (usually called containers) share the same kernel and filesystem, but offer isolated namespaces as well as limit and isolate resource usage such as CPU and memory. Popular container "execution environments" include Linux Containers (LXC), Virtuozzo and Linux V-Server. Also a number of container management and orchestration solutions exist such as Docker and Cloud Foundry Warden to name just a few. The PaaS builds on IaaS and it usually offers a higher-level support such as middleware, application execution environments, application monitoring facilities and platform customizations as services. Some of the currently popular PaaS offerings are Salesforce[17] and Heroku[18]. Finally, the SaaS provides applications and services to the end users. Examples of SaaS include Googe Apps and Netflix. For the work presented in this thesis IaaS techniques and solutions play the most important role.

Software-defined Environments and DevOps

Traditionally, in a narrow sense software defined is a term used to refer to software-defined networking (SDN). SDN is a networking approach intended to support network administrators to abstract and manage the network elements (e.g, switches and routers)

[15]http://aws.amazon.com/ec2/
[16]https://www.openstack.org/
[17]http://www.salesforce.com/
[18]http://www.heroku.com/

through high-level functions such as APIs [112]. General, idea behind the SDN is to separate the (possibly centralized) decision logic describing where the data packets are sent (control plane) from the actual mechanisms that deliver (forward) the data packets (data plane) [80, 78, 76, 81]. Most popular current implementations of SDN protocols are OpenFlow [97], ForCES [45] and OpenDaylight Platform[19].

Recently different approaches have emerged exploiting and extending software defined concepts to facilitate utilization and management of the pooled sets of shared resources, e.g., software-defined storage [140] and software-defined data centers [37]. The work presented in this thesis mainly exploits the general idea of software-defined environments aiming to enable refactoring the large-scale IoT Cloud resource pools and enable their provisioning and governance programmatically in software through well-defined APIs. To this end, in Part II of this thesis we elicit concrete design principles of software-defined IoT Cloud systems together with the main technical enablers to enable and support such principles.

The aforementioned tendency to abstract the infrastructure resources and enable their programmatic management is not only addressed in the cloud computing and software defined circles. DevOps is a relatively novel practice which aims to apply general principles of software engineering in order to facilitate the development of automation logic for provisioning, deploying, configuring, and upgrading system's infrastructure resources [48, 130, 99, 127]. DevOps, however, mainly focuses on providing tools and frameworks which would enable easier management of infrastructure components and configuration models. Some of the currently most popular DevOps tools revolve abound the notion of Infrastructure-as-Code (IaC) and include Chef[20], Puppet[21]. Finally, DevOps is more than a set of automation tools and it is sometimes reffed to as DevOps culture that aims to bring development and operations teams closer together through increased collaboration, shared responsibility and no silos approach [99, 127]. These concepts server as foundations for our governance approach in IoT Cloud systems, which is presented in Part III of this thesis.

[19]https://www.opendaylight.org/
[20]http://www.chef.io/chef/
[21]http://puppetlabs.com/

Part I

Programming IoT Cloud Systems

Preface

Recent emergence of IoT Cloud systems has fostered proliferation of various applications mainly driven by urgent need to respond to volume, velocity and variety of data generated by IoT Cloud, but also to enable timely propagation of actuation decisions, crucial for business operation, to the Edge of the infrastructure. However, due to the diversity, heterogeneity and scale of IoT Cloud systems, the need to handle large volumes of IoT data in a nontrivial manner, and the plethora of domain-dependent IoT controls, programming IoT Cloud applications remains a great challenge. The issue is further exacerbated due to inherently complex dependencies between application business logic and the underlying IoT Cloud infrastructure capabilities, which currently need to be explicitly considered during application development. Unfortunately, most of the contemporary solutions focus on low-level data and device integration, mainly aiming to support device-level development and domain experts. Such approaches provide only rudimentary support to account for diverse requirements coming from a variety of involved developer roles, leading to provisional solutions and heterogeneous application artifacts, which are "stitched up" into IoT Cloud applications.

The first part of this thesis aims to respond to the first main research question: "What is a suitable programming model and methodology for developing novel IoT Cloud applications in an efficient, scalable and generic manner?". The main driving force behind the contributions presented in this part of the thesis is to provide a set of programming abstractions and runtime mechanisms that enable more efficient, scalable and intuitive development of generic IoT Cloud applications, which seamlessly utilize both the Edge and the Cloud. Moreover, to account for the variety of involved stakeholders and the complexity of software stack, ranging from embedded devices development to high-level IoT Cloud application artifacts, our approach is designed in such manner to provide multiple logical views on the application development process, while retaining a uniform view (in code) on the produced application artifacts. To this end, in Chapter 3, we introduce PatRICIA – a novel programming model, which defines high-level programming constructs (MonitorIntents, ControlIntents and IntentScopes) that raise the level of programming abstraction, enabling developers to implement IoT Cloud applications without worrying about the specifics of diverse underlying Edge devices. Chapter 4, introduces Data- and Control-Points, which provide a different logical view on application development process and are mainly intended to support domain experts in developing common monitor and control tasks for IoT Cloud gateways, which reside at the Edge. Finally, in Chapter 5, we introduce SDG-Pro, which is a unifying programming framework for IoT Cloud, based on everything-as-code paradigm. The SDG-Pro framework combines Intents with Data- and Control-Points to provide a uniform support for application business logic development. In order to enable everything as code, it provides additional development support for programmatic provisioning and governance of IoT cloud systems, unifying it with the support of application business logic.

CHAPTER 3

A High-Level Programming Model for Cloud-centric IoT Cloud Applications

Advances in the Internet of Things have provided a global infrastructure of networked physical entities, able to monitor and control their physical status and surrounding environment, as well as to expose themselves via data streams and services over the network [9, 82, 96]. Various enterprise systems, e.g., smart building management system [119] and smart healthcare[60], utilize IoT applications to optimize key tasks of their business processes. Recently, cloud computing has become the key enabler for large-scale IoT systems. Researchers (e.g., [134, 64, 160]) recognize the benefits of exploiting cloud computing for IoT systems, as it could offer better solutions to support IoT applications in terms of device virtualization, provisioning of virtual sensors and actuators, and providing suitable execution infrastructure for resource-intensive IoT applications. However, to enable the development of IoT Cloud applications, we need high-level abstractions and mechanisms, which support scalable and efficient programming with diverse device services and raw data streams on cloud platforms. There are several approaches, which rely on service-oriented computing (SOC) principles to abstract device data- and actuation-points, e.g., [62]. They deal with heterogeneous devices by enabling direct, service-based access to devices and provide mechanisms for service discovery, provisioning and management. However, their support is usually restricted to the device-level services. Recently, several interesting attempts to apply cloud computing technologies in large-scale IoT systems have emerged, e.g., [160, 134]. They mostly focus on data and device integration by utilizing cloud infrastructure and virtualizing individual sensors and actuators as services in the cloud. Although these approaches help simplify the development of the IoT Cloud applications, the development process mostly involves composing these device-level services into admissible control sequences or data processing schemes, which makes the development of such applications highly complex, and potentially limits the programming scale. Concrete abstractions and mechanisms, which enable efficient, more intuitive and scalable development of IoT Cloud applications still remain underdeveloped.

In this chapter we present PatRICIA – a novel programming model for IoT applications on cloud platforms. PatRICIA defines high-level programming constructs (*Intents and IntentScopes*) and operators, which encapsulate domain-specific knowledge (domain model and behavior) and raise the level of programming abstraction, enabling developers to implement IoT Cloud applications without worrying about the complexity of low-level device services and raw sensory data streams. We demonstrate, based on a real-life applications, how our programming model enables easier, efficient and more intuitive development of IoT Cloud applications.

The remainder of the chapter is organized as follows. In Section 3.1, we present the motivating scenario and the key research challenges. Section 3.2 outlines the PatRICIA framework and Section 3.3 presents the programming model. Section 3.4 describes the implementation and evaluation of our system. Finally, Section 3.5 presents final remarks and concludes the chapter.

3.1 Motivation and Research Challenges

3.1.1 Scenario

Let us consider our real-life FMS scenario, described in Chapter 2, form an application perspective, i.e., in this chapter we analyze FMS applications, mainly focusing on different aspects of such applications in terms of required development support to realize them in practice.

In general, the FMS applications are characterized by a reactive behavior. They receive some (*monitoring*) information, e.g., a change in vehicles' operation and, as a response, perform (*control*) actions on the vehicles. Figure 3.1 visualizes general, high-level architecture of FMS IoT Cloud applications. We notice that applications define custom *business logic*, but can utilize similar *monitor and control tasks* and need to apply them on dynamically defined *scopes* to manage the entities in a scalable manner. Therefore, these tasks can be modeled and represented, in such a manner to enable their reuse across different IoT Cloud applications. Typically, these applications monitor vehicle's status like maintenance and fault history, battery health, engine status, location, tire pressure and so forth. For example, one of FMS applications needs to determine if a vehicle is consuming more energy compared to other vehicles in the fleet, i.e., to detect high *energy fault*. To this end, we can utilize power- and odometer from fleet vehicles and correlate this information in order to detect the fault. Further, our applications react to changes, e.g., in vehicle operation, by taking appropriate actions. For example, if an *energy fault* is detected the application can decide to *notify* the driver and the golf course manager via available devices, e.g., a smartphone or to put the vehicle in a *reduced*

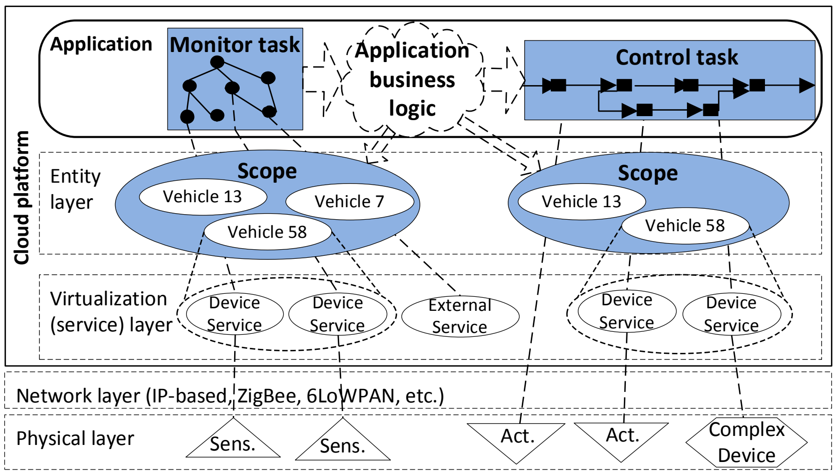


Figure 3.1: Example FMS IoT Cloud application.

energy mode, by setting speed, RPM and transmission limits, and wait for the vehicle to return to the base where it can be further examined.

To perform the above-mentioned tasks, among other things, the FMS applications require complex and expensive analytics and have high demand on storage and communication resources. Because these applications connect to and deal with a large number of vehicles, which are distributed across different golf courses, they must be able to handle vast amounts of data efficiently and need to have a global view of the distributed fleet. To support these requirements, it is natural to execute these applications in the cloud, as it has capabilities to connect, provide access, and a unified global view of the geographically distributed fleet. The applications are envisioned to run continuously, but they can be elasticaly scaled down in off-peek times, e.g., during the night, when most of the vehicles remain dormant. In this case, the elastic nature of the cloud can provide advantages in terms of cost reductions and greener IoT computing, e.g., because of reduced energy consumption. Due to the multiplicity of the involved stakeholders with diverse requirements and business models (cf. Chapter 2), FMS applications need to support different and customizable usage experiences. Also here the cloud computing is essential, as it potentially offers new, possibly cross- domain, application opportunities and enables flexible business and usage models. Therefore, in this context, the cloud plays a crucial role, as existing enterprise-specific platforms are hardly capable to meet all of these requirements.

3.1.2 Research Challenges

We show that most of the current approaches (cf. Chapter 10) that support the development of IoT applications deal with device and data integration and focus mostly on the *Device virtualization layer* or the layers below (cf. Figure 3.1), thus, application developers have to deal with much of the complexity, diversity and scale of IoT Cloud applications. We identify several challenges regarding the development of such applications:

- *RC1* – Developers need to work with raw sensory data streams and write complex queries and event processing schemes (*monitor tasks*), from scratch. This largely requires them, among other things, to be knowledgeable about individual device specifics, data formats and communication protocols. However, such assumptions are hard to meet in the face of large-scale systems comprising variety of heterogeneous Edge devices.
- *RC2* – A developer needs a good knowledge about diverse low-level device services and implications of invoking these atomic services, to be able to establish correct dependencies between them and compose them into admissible control sequences (*control tasks*).
- *RC3* – The IoT Cloud applications execute in very dynamic environments and interact with hundreds or thousands of physical entities. Therefore, monitoring and controlling these entities in a scalable manner is another challenge for developers of IoT Cloud applications, because these applications need to be able to dynamically identify *the scope* of their actions, depending on the task-at-hand.
- *RC4* – Finally, due to dynamicity of environments, diversity of devices, ad hoc requirements of diverse stakeholders, and hardware or network failures, developing security-, privacy-, safety-, cost- and quality-aware IoT Cloud applications is a very challenging task without adequate runtime mechanisms to support it.

3.2 Design Requirements and Overview of PatRICIA Framework

3.2.1 Main Design Requirements

Contemporary cloud techniques, e.g., virtualization, elastic scaling, resource and tenant management play a crucial role in highly dynamic and heterogeneous IoT systems. Although, they enable us to virtualize and connect vast amounts of devices, provide a unified view on IoT infrastructure and offer theoretically unlimited processing and storage capabilities, we still need to reduce the complexity and enable scalable development of IoT applications. Therefore, this requires rethinking the existing application development and execution models.

The main aim of the **PatRICIA** *(**PR**ogramming **I**ntent-based **C**loud-scale **I**oT **A**pplications)* framework is to define an ecosystem, which provides an end-to-end solution for IoT Cloud applications. This includes providing a programming model and development tools, as well as cloud-based application execution platform. The core idea of the

PatRICIA is to enable development of value-added IoT applications, which are executed and provisioned on cloud platforms but leverage data from different sensor devices and enable timely propagation of decisions, crucial for business operation, to the Edge of the infrastructure.

Programming IoT Cloud applications requires different skills and backgrounds, e.g., working with low-level hardware, developing enterprise applications and having knowledge about the domain of interest. Therefore, our framework needs to provide different *logical views* on the development process and enable different developer roles to coherently encapsulate their expertise and focus their development effort. We noticed that most of the tasks performed by the applications are generic, in the sense, they capture knowledge and industrial best practices in the domain. Therefore, they need to be represented as generic components that can be easily reused. To enable working with these generic tasks in a *cloud-scale manner*, we need to enable automatic task instantiation for developer-defined scopes, e.g., a golf course or the fleet. To keep our programming model stable and easily extensible, we need to enable late runtime binding of the tasks, i.e., decouple task representation from the implementation of its behavior.

Different runtime mechanisms are needed to enable the applications to adapt to changes in quality and costs but also to guarantee safety and security for both users and devices. Further, we need to provide code-distribution techniques, which will allow IoT Cloud applications to utilize the edge of the infrastructure (e.g., gateways) as the additional resources, e.g., processing and storage. Application deployment and infrastructure administration need to become fully automated, due to the scale of IoT systems and because domain-specific verticals are increasingly becoming refactored, enabling development of cross-domain IoT Cloud applications. Therefore, the major requirements for our framework include:

1. Providing a programming model to raise the level of programming abstraction by decoupling domain knowledge implementation from its representation and usage.

2. Providing a cloud-based application execution environment and the supporting runtime mechanisms.

3. Providing development tools, such as testing and staging environments to fully support application development lifecycle.

4. Policy-based automation to enable development of security-, privacy-, safety-, cost- and quality-aware applications.

In this chapter, we focus primarily on the requirements 1 and 2 and present the design and evaluation of the components to support them.

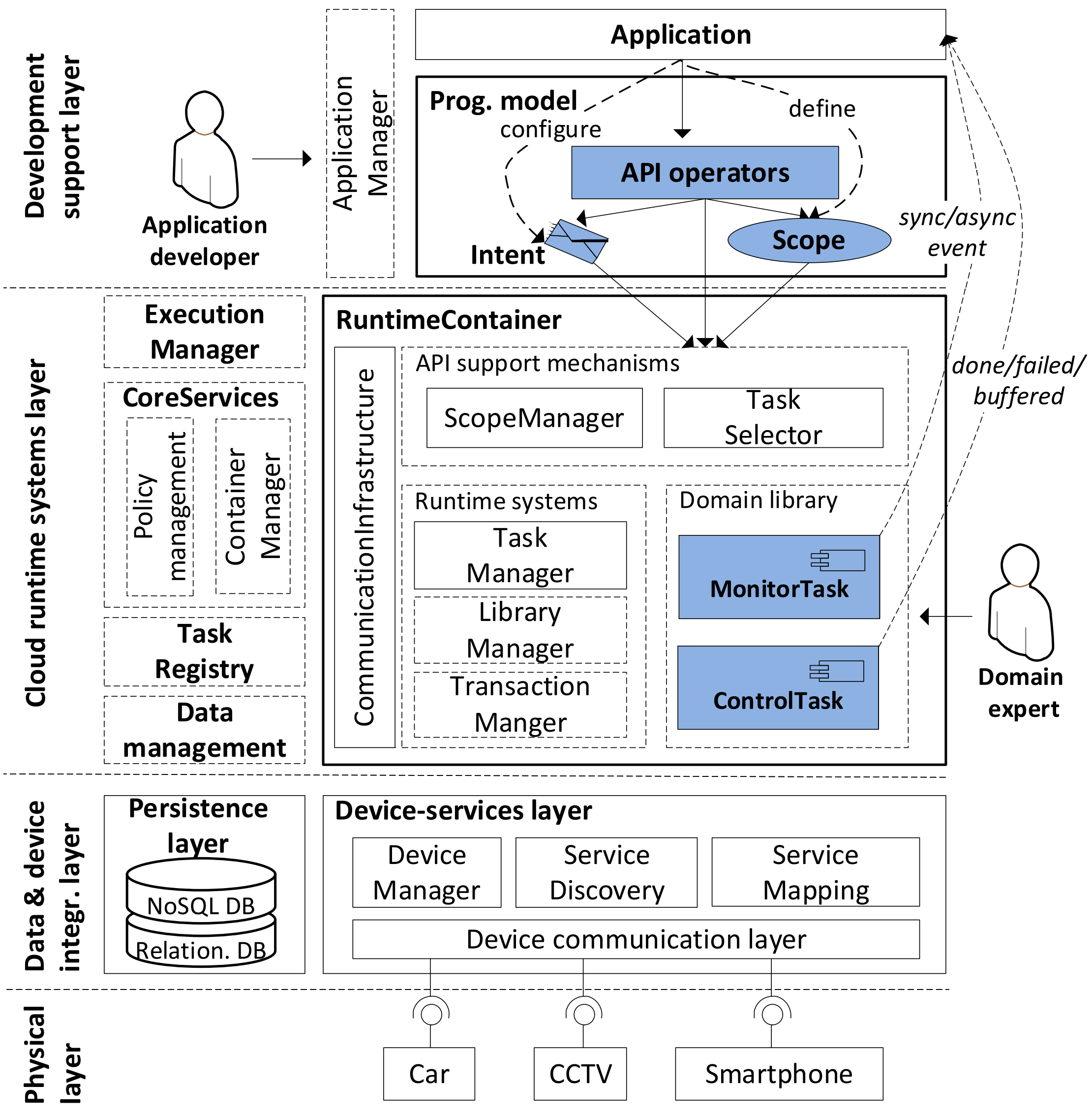


Figure 3.2: PatRICIA architecture overview.

3.2.2 PatRICIA Architecture and Multiple Logical Views on Application Development Process

Figure 3.2 gives an overview of PatRICIA's architecture. Our framework architecture is based on SOA design principles which enable flexible, adaptable and evolvable architecture. The modularity of the framework enables extending the current prototype with future concepts and allows flexible configuring and scaling of individual components atop cloud infrastructure.

Development support layer contains development tools, which are needed to support application development lifecycle and enable provisioning of IoT Cloud applications. The programming model is the most important component of this layer as it enables development of IoT Cloud applications. *ApplicationManager* is responsible for application configuration, deployment and licensing. Also, this layer integrates a testing environment for the IoT Cloud applications.

Cloud runtime systems layer includes the *RuntimeContainer*, which implements supporting mechanisms for our programming model and provides an execution environment for the applications. The *ExecutionManager* is responsible to monitor applications and the *RuntimeContainer*, and provide mechanisms to elastically scale them during runtime. The *CoreServices* contain a variety of runtime mechanisms and services which are needed by the PatRICIA framework. For example, the *ContainerManager* is used to configure the *RuntimeContainer* and deploy different policies. Further, the *Policy management* component provides mechanisms to specify and enforce these policies, e.g., costs, privacy, security and safety. The *TaskRegistry* is used to store task templates and their metadata. The *Data management* component provides storage, manipulation and analytics mechanisms for the sensory data. It also provides a data quality assurance service, accessible to the runtime container to perform data quality checks.

Data and device integration layer includes *Device-services layer*, which is the IoT device virtualization and management layer of the PatRICIA framework and it underpins monitor and control tasks. It contains *Device communication layer* that implements different connectors, which encapsulate device-specific APIs, communication protocol and enable device-cloud connectivity. The *ServiceMapping* component wraps a physical device and exposes it as a service which defines push, poll, pub and sub methods, providing a communication interface with the devices. *DeviceManager* is responsible for device management, e.g., to detect newly connected devices. The *ServiceDiscovery* component enables discovering and registering device-services. Finally, *Persistence layer* contains NoSQL and relational database, which are used to store the sensory data and other information, needed by the PatRICIA framework.

At this point it is important to note that the PatRICIA framework defines *two logical views* on the development process of IoT Cloud applications. Conceptually, it provides support for both domain expert and high-level application developer roles. In the rest of this chapter, however, we mostly focus on the support provided to high-level application developers (Figure 3.2 top-left), by introducing a programming model, a cloud-based application execution environment and prototype implementation of a vehicle management domain library. Domain experts (Figure 3.2 right-hand side) use Data- and Control-Points (an extension of the PatRICIA framework presented in Chapter 4) to define a domain model and common monitor and control tasks, which form *the domain libraries.*

3.3 Intent-based Programing Model for IoT Cloud Applications

3.3.1 Main Programming Constructs and Operators

Our programming model defines constructs and operators, used by developers, to write IoT Cloud applications. They enable the developers to work with predefined control and monitor tasks that are provided in the domain library. A control task is any permissible sequence of actuating steps which can be used to control physical devices. Further, a monitor task includes processing, correlation and analysis of sensory data streams to provide meaningful information about the state changes of the underlying environment. The domain libraries are discussed in more detail in Chapter 4.

At application level, PatRICIA provides explicit representation of these tasks via *Intent*s, i.e., developers write *Intents* to configure and invoke the tasks. When a task is invoked, it is automatically instantiated for the supplied *IntentScope*. Developers use *IntentScope* to delimit the range of an *Intent*. For example, a developer might want to code the expression: "stop all vehicles on hole 1". In this case, "stop" is the desired *Intent*, which needs to be applied on a *IntentScope* that encompasses all vehicles with the location property "hole 1". In our programming model *Intent* and *IntentScope* are first-class entities. This means that they can be stored in variables, used in expressions and passed as parameters to functions. Generally, they enable developers to work with common, predefined concepts, without worrying about their implementation in the underlying environment.

IntentScope

IntentScope is an abstraction, which represents a group of physical entities (e.g., vehicles), which share some common properties, e.g., context. More precisely, it is a set of software entities on the cloud platform, which virtualize corresponding physical entities. Thus, *IntentScope*s are determined on the cloud platform, but they enable developers to dynamically delimit physical entities on which an *Intent* will have an effect. In reality there are infinitely many scopes, which can be defined by the applications and can include hundreds of diverse, geographically distributed vehicles. Therefore, we provide mechanisms to dynamically define and work with *IntentScope*s on the cloud platform.

To define an *IntentScope* developers specify properties, which need to be satisfied by the physical entities to be included in the scope. For example, *IntentScope*s can be defined based on a behavior, e.g., "all vehicles exceeding speed limit", a state ("all vehicles with low battery") or a static feature ("all vehicles with a price over ..."). To enable *IntentScope* bootstrapping, we provide a special type of *IntentScope*, which is called *GlobalScope*. It defines the maximal scope for an application and usually contains all physical entities administered by a stakeholder at the given time. Therefore, it is reasonable to assume that the *GlobalScope* is slow-changing over time and it can be configured by a user, e.g., a golf course manager. In PatRICIA the *GlobalScope* is represented as a global variable, which can be directly referenced by an application. Contrary, the minimal *IntentScope*, which can be referenced by an application is a single entity. Our programming model allows *IntentScopes* to be defined explicitly and implicitly. To explicitly define an *IntentScope*, a developer can manually add the entities to the scope by specifying their Ids. Implicit definition of the scope is usually performed by recursively pruning the *GobalScope* and/or combining two or more *IntentScopes*.

Formally, we use the well-known set theory to define *IntentScope* as a finite, countable set of entities (set elements). The *GlobalScope* represents the universal set, denoted as S^{max}, therefore, $\forall S(S \subseteq S^{max})$, where S is an *IntentScope*, must hold. Further, for each entity E in the system general membership relation $\forall E(E \in S | S \subseteq S^{max})$, must hold. The S_{min} is a unit set which contains a single entity. Further, empty IntentScope ($\emptyset$) is not defined in our programming model, thus applying an *Intent* on it results in an error. Finally, a necessary condition for an *IntentScope* to be valid is: *IntentScope* is valid iff it is a set S, such that $S \subseteq S^{max} \wedge S \neq \emptyset$ holds. Equation 3.1 to 3.7 show how to define or refine an *IntentScope*.

$$S = S_{min} \tag{3.1}$$
$$S = S^{max} \tag{3.2}$$
$$S = \subseteq_{cond} A,\ where\ A\ is\ a\ valid\ IntentScope \tag{3.3}$$
$$S = A \cup B,\ where\ A\ and\ B\ are\ valid\ IntentScopes \tag{3.4}$$
$$S = A \cap B,\ where\ A\ and\ B\ are\ valid\ IntentScopes \tag{3.5}$$
$$S = A \setminus B,\ where\ A\ and\ B\ are\ valid\ IntentScopes \tag{3.6}$$
$$\tag{3.7}$$

The most interesting operator is $\subseteq_{cond} A$. It is used to find a subset (S) of a set A, which satisfies some condition, i.e., $E \in S \mid E \in A \wedge cond(E) = True$. In this context *cond* can be $True$ or $False$ depending if an entity satisfies specified property, e.g., if it displays "EnergyFault". To define the *cond* developers can use any monitor task defined in our domain library and need to provide a parametrized condition expression, e.g, "EnergyFault==True". Further, we provide the common operations on sets, i.e., $\cup, \cap$ and $\setminus$, which have their traditional meaning.

By introducing *IntentSope*s at the application level, we enable development of IoT Cloud applications in a scalable manner by shielding the developers from directly referencing the vast number of diverse physical entities and enabling them to dynamically delimit the range of *Intent*s. Therefore, IntentScopes address the *RC3*.

Intent

Intent is a data structure that describes a specific task which can be performed in a physical environment. In reality, *Intent*s are processed and executed on the cloud platform, but enable monitoring and controlling of the physical environments. Based on the information contained in an *Intent*, a suitable task is dynamically *selected, instantiated*

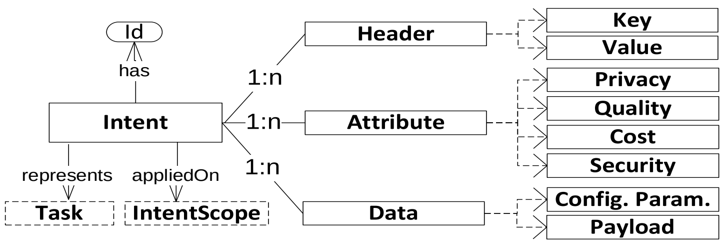


Figure 3.3: Intent structure.

and executed on the cloud platform. Our framework translates the *Intent* into a sequence of actuation or data processing steps and maps them on the underlying physical devices (cf. Section 3.3.3). Depending on the task's nature, we distinguish two different types of *Intent*s: *ControlIntent* and *MonitorIntent*. *ControlIntent*s enable applications to provision, operate and manage the low-level components. They abstract the underlying devices and provide a high-level representation of their functionality. *MonitorIntent*s are used by applications to subscribe for events from the underlying environment and to obtain and provision devices' context.

Figure 3.3 shows the *Intent* structure and its most relevant parts. Each *Intent* contains an id, used to correlate invocation response with it or apply additional actions on it. Additionally, it contains a set of headers, which specify meta information needed to process the *Intent* and bind it with a suitable task during the runtime. Among other things, headers carry intent's name and a reference to an *IntentScope*. Further, an *Intent* can contain a set of attributes, which provide information, such as costs, quality, privacy or security requirements. They describe the *Intent* to more detail and are used by the runtime to select the best matching task instance in case there are multiple implementations supporting the *Intent*. Finally, *Intent* can contain data, which is used to configure the tasks and devices or supply additional payload, e.g., a notification message.

To perform an IoT control or to subscribe for relevant events, developers only need to define and configure *Intent*s. This allow them to communicate to the system what needs to be done, instead of worrying how the underlying devices will perform the specific task. Additionally, by supporting dynamic binding of the tasks, we enable development of loosely coupled applications that are independent of the specific task implementation and guaranty stability (e.g., backward compatibility) and enable extensibility of our programming model. Therefore, *Intent*s shield the developers from the complexity of IoT controls and complex data processing, as well as from the diversity of IoT devices and physical environments, addressing research challenges *RC1& RC2* .

Coupling Intents with IntentScopes

To enable runtime coupling of *Intent*s and *IntentScope*s we need to fully define a validity of *IntentScope*s. First, we examine applicability of an *Intent* on S_{min} (see Section 3.3.1). Obviously, this comes down to applying the *Intent* on an entity. Therefore,

$apply(I, S_{min}) = apply(I, E)|E \in S_{min}$, where I is an *Intent*. Further, $apply(I, E)$ is true if an *Intent* can be instantiated for the entity and it is determined by the system at runtime, by examining the *mapping and filter* conditions (see Section 3.3.3). Therefore, we can apply an *Intent* on S_{min} iff we can apply it on the entity E. Further, because each set S_n can be defined as union of unit sets (S^i_{min}), $S_n = \bigcup_1^n S^i_{min}$, we observe applying an *Intent* can be defined recursively, i.e, $apply(I, S_n) \equiv \bigwedge_1^n apply(I, S^i_{min})$. Therefore, we can apply an *Intent* on an *IntentScope* if we can apply it on its all subsets.

```java
Intent eFault = Intent.newMIntent("EnergyFault");
//monitor whole fleet
eFault.setScope(IntentScope.getGlobal());
notify(energyFault,this);//invoke task
//callback function called on event arrival
public void onEvent(Event e){//perform some action}
```

Listing 3.1: Example usage of MonitorIntent and GlobalScope.

```java
//Define IntentScope with eFault (defined in the listing above)
IntentScope cs = delimit(IntentScope.getGlobal(),Cond.isTrue(eFault));
//Define and configure Intent
Intent eCons = Intent.newCIntent("ReduceEnergy");
eCons.setScope(cs);//set intent scope
eCons.set("speed").value("5");
eCons.set("RPM").value("1100");
send(eCons); //invoke task
```

Listing 3.2: Example usage of ControlIntent and custom IntentScope.

Now we can show concrete examples of *Intent* and *IntentScope*. Listing 3.1 depicts a MonitorIntent used to monitor energy consumption and detect potential "EnergyFault" for each vehicle in the fleet. Listing 3.2 gives an example of *ControlIntent* usage. It shows how to define an *IntentScope* for all vehicles displaying "EnergyFault" and sends *"ReduceEnergy" ControlIntent* to all of them to set the speed limit to 5km/h and to limit engine to 1100 rpm.

Intent operators

Since Intent is a passive data structure, we need to provide developers with operators to work with the *Intent*s. These operators encapsulate mechanisms to *select, instantiate and execute* underlying tasks, based on the input *Intent*. Consequently, instead of dealing with the individual tasks, a developer is presented with a unified interfaces to communicate with the runtime systems. To this end, PatRICIA APIs are divided into three categories: *core, system and utility operators*. In the following, we only present the core operators, shown in Listing 3.3.

The `send` primitive is used to communicate and execute a *ControlIntent*. It accepts the *ControlIntent* as an argument and returns *done* if the *ControlIntent* was executed

```
send(in ci:ControlIntent,out r:Result)

notify(in mi:MonitorIntent,in o:CallBackObj)

poll(in mi:MonitorIntent,out el:List<Event>)

delimit(in s:IntentScope,in c:Cond,out so:IntentScope)
```

Listing 3.3: Core Intent API operators.

successfully, *failed* if it is currently impossible to execute the *ControlIntent* and *buffered* if the underlying device is currently busy. When the `send` operator is invoked the container first selects suitable tasks to execute the *ControlIntent* by using intent headers. The task list is further filtered, based on intent attributes, e.g., quality requirements. Here, we use best-effort to find the best matching task implementation. Further, the selected task is configured with *Intent*'s configuration parameters and a payload. Subsequently, the task is instantiated for each entity and finally executed (see Section 3.3.3).

The core operators `notify` and `poll` are used to support working with the *MonitorIntents*. The operator `notify` is used by an application to subscribe for events, which are asynchronously delivered to the application. It requires two arguments: a *MonitorIntent*, used to match the appropriate monitor task and a reference to a *callback object*, which gets notified when a new instance of an event becomes available. The `poll` is used to synchronously check the status of the environment, i.e., it will block application's main thread if the required event is currently unavailable. It also requires a *MonitorIntent* as an argument and returns an event (or null) as a result.

The `delimit` operator is API equivalent of $\subseteq_{cond}$, defined in Section 3.3.1. It is used to define an *IntentScope* with entities, which satisfy a certain condition. Usually, when an application wants to determine the *IntentScope*, it will start by invoking `delimit` on the *GlobalScope* and further refine it by recursively applying this operator and/or using other operators defined in Section 3.3.1.

3.3.2 Application Structure and Lifecycle

Figure 3.4 depicts a simplified UML diagram showing the structure of our applications. *Application* structure is defined via `onCreate`, `onAppStart` and `onAppExit` methods, which represent hooks used by the runtime to manage applications lifecycle. The `onCreate` method represents application entry point. It provides the application with runtime *Context*, which provides global information about the application environment. After initialization completion, the container invokes `onAppStart`, which contains application's business logic and from this point on the application is ready to use programing model constructs. Finally, before the application ends, the container invokes `onAppExit` method. This method is used to perform "house keeping", e.g., close any open connections and release any direct references to tasks. From developer's point of view applications are single-threaded and only the main thread is visible to the developer.

Therefore, the container takes the responsibility to spawn and manage new threads for each task in the system. This significantly eases application development and enables clear separation of application and task lifecycle management.

Task (see Figure 3.4) is defined as an abstract class, which captures general concepts of *MonitorTask*/*ControlTask* and represents the main building block for domain libraries. The *Task* contains metadata, used by the container to bind an *Intent* with the *Task* at runtime. Metadata contains *Filter*s, *Mapping*s, *Config* and *Attributes*. The *Filter* specifies a list of *Intents*, supported by the task. The *Mapping* provides information about supported entity types, e.g., vehicle family. The *Config* contains a default configuration of the task. Finally, *Attributes* provide additional information about the task, e.g., provided data quality. Further, *Task*s also have a lifecycle, which is managed by the runtime container. To this end, *Tasks* provide hooks which are used to initialize (`onCreate`), activate (`onStart`) and stop (`onStop`) the task. The `onCreate` contains custom code to initialize a *Task*. The `onStart`, contains processing logic or a sequence of actuation steps. This is the core part of each *Task*, as it contains the specific domain logic, which is executed when an *Intent* invokes the *Task*. We don't define how *Task*s implement the domain logic. For example, monitoring tasks can utilize an event processing framework to implement data processing, but conceptually *Task*s are technology independent. When `onStart` exits, one of the communication methods (`onProcessingDone` or `onActuationDone`) is invoked in order to send response to the application. Finally,

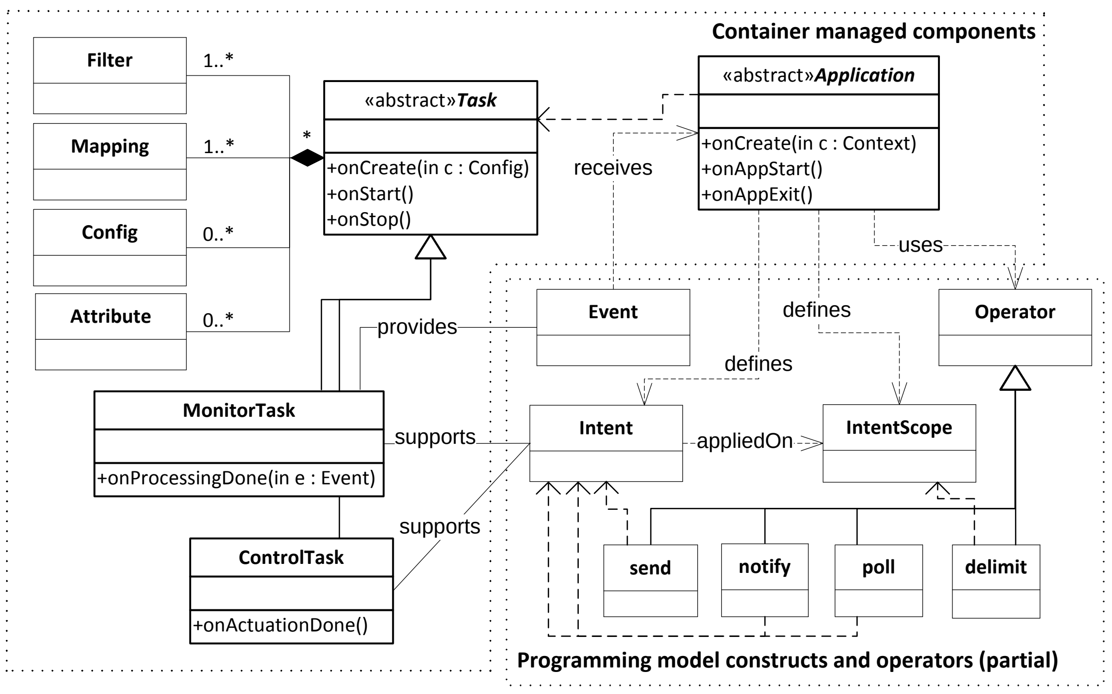


Figure 3.4: Simplified UML diagram of application structure.

when a *Task* instance is no longer needed the container invokes the `onStop` method.

3.3.3 Runtime Support

The *RuntimeContainer* (see Figure 3.2) provides an execution environment for the IoT Cloud applications. To this end, it implements mechanisms to manage *Task*s and their lifecycle and to dynamically bind *Intent*s with *Task*s. Further, it provides interfaces to register new *Task*s and mediates the communication between the applications and domain libraries.

Intent selection, instantiation and execution

When an application submits a new *Intent*, the *RuntimeContainer* first routes it to the *TaskSelector*, which matches intent *headers* with *Task*'s *filters* and *mappings* to find *Task*s which provide the *Intent* implementation. Afterwards, the *TaskSelector* reads the required (*Intent*) and promised (*Task*) *attributes* and compares them to find the best matching task. Attributes are represented as feature vectors and a multi-dimensional utility function, based on the Hamming distance, is used to perform the matching. Further, *TaskSelector* requests a *Task* instance, by providing its description to the *ScopeManager*, which checks the validity of the coupling and if it is valid, forwards it to the *TaskManager* or otherwise marks the *Intent* as failed. The *TaskManager* instantiates the *Task* via reflections and configures it with intent's *data*. Finally, it triggers the `onCreate` method on all task instances, to execute any custom initialization code and `onStart` to trigger the execution of the task logic. In Chapter 4, we provide more detailed information on tasks creation and their runtime management.

Intent delivery mechanism

*Intent*s are used to invoke *Task*s. They are processed and executed on the cloud platform, but interact with underlying devices. To perform an *Intent*, PatRICIA instantiates the corresponding *Task* for an entity. It is the responsibility of *Task*s to map the *Intent* to the low-level device services. To this end, in the current prototype they utilize *Devices-services layer* and implement required monitor/control logic.

The CommunicationInfrastructure (cf. Figure 3.2) is a communication backbone of our *RuntimeContainer*. It is used to transport *Intent*s and events between applications and the *Task*s. To enable loose coupling between the applications and the domain library tasks it follows pub/sub paradigm. Furthermore, because *Intent*s contain information needed to process and route them, the *CommunicationInfrastructure* must be able to understand *Intent* headers. Therefore, the communication is performed via a partial content-based pub/sub model.

For example, currently our domain library provides implementations of *Task*s per vehicle family. Each vehicle has a unique Id, which is used to define a messaging topic. All the communication between our library's *Task*s and the vehicle is performed via this topic. The *ServiceMapping* component provides the communication interface and *Device communication layer* provides the required *connector* (a message broker) to communicate with the physical environment. Naturally, as our *Task*s are technology independent, domain libraries can use other mechanisms (*connectors*) to map the *Task*s on the underlying devices.

Runtime coupling of Intents with IntentScopes

ScopeManager implements the *IntentScope* API. It defines a global reference to the *GlobalScope* and implements operators to work with scopes. *GlobalScope* is a singleton, which is initialized with all devices found in the tenant's database. For storing device meta information PatRICIA uses relational databases in the *Persistence layer*. To determine temporary changes in the *GlobalScope*, e.g., devices gone offline, *ScopeManager* communicates with the *DeviceManager*, which implements the Last Will and Testament (LWT) pattern to detect device failure and adapt the *GlobalScope* accordingly.

To support the `delimit` operator *ScopeManager* provides functionality to evaluate the condition expressions and initiates *selection, instantiation and execution* of a *MonitorIntent*, to obtain the value of the specified property (see Section 3.3.1). Finally, it provides runtime checks to apply an *Intent* on *IntentScope*. To do this, when an *IntentScope* is added to an *Intent*, the *ScopeManager* resolves the scope and checks if there are suitable *Task*s to support this *Intent* for each entity in the scope by comparing task *filters and mappings*, with intent *headers* (name, Id, entities types, etc.).

3.4 Evaluation & Prototype Implementation

3.4.1 Prototype Implementation

The PatRICIA framework is implemented in the Java programming language and it is based on WSO2 Stratos[155], which is an open source, full-fledged PaaS solution stack that provides many customizable services, such as identity management, monitoring, logging and multi-tenancy support, necessary for the PatRICIA implementation. The *Persistence layer* relies on a MySQL database to store relational data, e.g., device meta information, and key/value storage (Apache Cassandra [4]) for storing NoSQL sensory data. The PatRICIA chooses how to store the data, depending on its nature. For example, user data, device meta information, etc. is relational and usually requires immediate consistency, thus relational database is used. Contrary, our sensory data is write-intensive and eventual consistency is sufficient most of the times, because the analysis is mostly performed off-line, e.g., by submitting map/reduce jobs. The *CommunicationInfrastructure* in our *RuntimeContainer* is based on Mosquitto MQTT-Broker[1] and it is used to mediate the communication between the applications and domain library *Task*s. This allows PatRICIA to leverage existing, proven technologies, which provide content-based pub/sub communication between the components, decoupling

[1]http://mosquitto.org

them and making our programming model higly extensible. The communication is topic-based and the *TaskSelector* uses message selectors to route *Intent*s/events between the applications and the *Task*s. We use message properties to model Intent headers and realize the content-based communication. Currently, at the application level we support XML and JSON (attribute/value) representation of the *Intents*. To enable *IntentScope* bootstrapping *ScopeManager* defines a global singleton reference to the *GlobalScope*. It is initialized by querying MySQL database and updated when the *DeviceManager* receives an update from the broker.

Prototype implementation of the *Domain library* contains a set of *Tasks*, which support *Intent*s used to develop our applications. The task implementations are out of scope of this chapter and they are presented in Chapter 4. Library tasks rely on the *Device communication layer* to communicate with the vehicles. It contains its own message broker and *connectors* (one per vehicle family) to mediate the communication. Library tasks communicate with the vehicles over MQTT topic, identified via the vehicle Id. MQTT [114] is a lightweight M2M pub/sub messaging transport, which is a standard for communication with the IoT devices. To implement the *connector*s we used Protocol Buffers [58]. The communication protocol with the vehicles defines a set of vehicle control and status messages, which are marshaled and transported between the vehicles and the applications by the *Device-services layer*. To this end, each vehicle gateway hosts a MQTT client, which translates the protocol messages for a specific task.

3.4.2 Evaluation

Example application implementation

We now show how PatRICIA's programming model is used to implement the real-world IoT Cloud application (see Section 3.1) and use traditional programming model evaluation criteria to validate its feasibility. The complete source code of the application is shown in Listing 3.4. Considering *readability and simplicity*, we notice that a developer uses intuitive high-level abstractions (Intent and IntentScope) to write IoT applications, instead of dealing with low-level device-services. Further PatRICIA also provides improvements regarding *reusability* and more *efficient development.* For example, a developer can easily code monitoring of specific fleet vehicles, which fulfill some criteria (lines 7-11). Although, we limit the *expressiveness* to a certain extent, a developer can still easily and intuitively express many common behaviors of could-scale IoT applications (lines 16-19). Finally, *extensibility* of our programming model is guaranteed by deferring the Intent-Task binding to the runtime. This enables adding new concepts (Intents and Tasks) to the model without modifying the existing ones and at the same time guaranties the *backward compatibility* of the applications. Therefore, PatRICIA reduces the *complexity* and enables developers to cope with the *diversity* and the *scale* of the IoT Cloud applications.

```
public class ExampleApplication extends Application{
 private Container cont;
 public void onCreate(Context c){
  this.cont = c.getContainerRef();
 }
public void onAppStart(){
 IntentScope s = cont.delimit(IntentScope.getGlobal(),
   Cond.greaterThan("price", "5000"));
 Intent eFault = Intent.newMIntent("EnergyFault");
 eFault.setScope(s);
 cont.notify(eFault, this);//sub. to event
 IntentScope controlS = cont.delimit(s,Cond.isTrue(eFault));
 performIntent(controlS);
}
private void performIntent(IntentScope ts){
 //define Intent and use default configuration
 Intent eCons = Intent.newCIntent("ReduceEnergy");
 eCons.setScope(ts);//set task scope
 cont.send(eCons); //send to all vehicles in ts
}
public void onEvent(Event e){
 performIntent(IntentScope.create(e.getEntityId()));
}
public void onAppExit(){//nothing to do here}
}
```

Listing 3.4: Example IoT Cloud application.

Experiment results

In order to evaluate how our PatRICIA framework behaves in a large-scale setup (hundreds of Edge devices), we created a virtualized IoT Cloud testbed based on CoreOS[2]. In our testbed we use Docker containers to virtualize and mimic physical gateways in the cloud. These containers are based on a snapshot of a real-world gateway, developed by our industry partners. Additionally, we have implemented a simple MonitorTask that relies on simulated (real-life replayed) sensory readings to perform rudimentary data processing in the virtualized gateways and deliver the events to a cloud-based application.

For the subsequent experiments we deployed a CoreOS cluster on our local OpenStack cloud. The cluster consists of 4 CoreOS 444.4.0 VMs (with 4 VCPUs and 7GB of RAM), each running approximately 150 Docker containers. The MonitorTask is preinstalled in the containers. The PatRICIA framework and the cloud-based application are deployed on an Ubuntu 14.04 VM (with 2VCPUs and 3GB of RAM).

Figure 3.5 show the performance of PatRICIA's intent delivery mechanism (cf. Section 3.3.3), based on 5 repetitions of the experiment. It is worth mentioning that the execution times shown in the figure represent the full lifecycle of an Intent execution, i.e., Intent, IntentScope and MonitorTask instantiation, Intent delivery to the devices(tasks), as well as execution time of the task and results delivery. Figure 3.5 illustrates a scatter plot of the individual experiment runs for different sizes of the IntentScope, which ranged

[2]http://coreos.com/

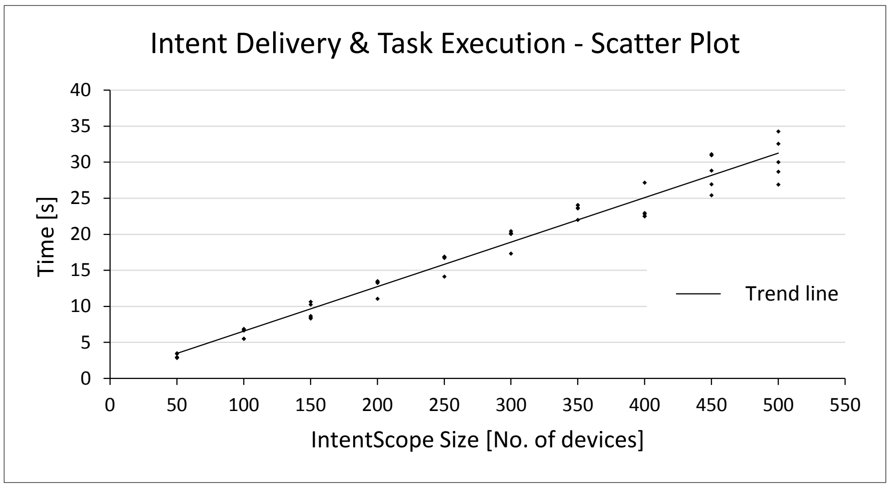


Figure 3.5: Intent delivery time for different IntentScope sizes.

from 50 to 500 gateways. The main purpose of the experiments was to show the scalability of our framework's core runtime services. To this end, the figure also plots a trend line, extrapolated from the individual test runs, based on linear regression. We observe that the intent matching algorithm and the delivery mechanism scale linearly with the size of IntentScopes, which indicates eventually consistent execution of the Intents for relatively large number of devices. Finally, it is important to note that the actual duration of Intent execution largely depends on the nature of the underlying task (i.e., problem-at-hand) and the overhead of PatRICIA's runtime was statistically negligible.

3.5 Conclusion

In this chapter, we introduced the PatRICIA framework for programming IoT Cloud applications. We discussed how our framework offers two logical views on the development process of IoT Cloud applications, in order to provide programming support for multiple developer roles. In this chapter we mainly focused on the development support provided to high-level application developers. To this end, we presented PatRICIA's main programming abstractions: *Intents* and *IntentScopes*, together with a set of runtime mechanisms to support such high-level developers in dealing with the complexity and diversity of IoT Cloud systems, as well as to enable development of IoT applications in a cloud-scale manner. The set of proposed concepts is not exhaustive, but is sufficient to express many common behaviors of IoT Cloud applications. However, this additional level of abstraction comes at cost and as discussed PatRICIA's programming model trades flexibility for a scalable, more intuitive and efficient programming of the IoT Cloud applications. Since this behavior might be a limitation for some application developers (e.g, domain experts), in Chapter 4, we introduce an extension to the PatRICIA framework that offers more flexibility.

CHAPTER 4

A Programming Model for Resource-constrained IoT Cloud Edge Devices

In Chapter 3, we introduced a high-level programming model for IoT Cloud applications. We discussed the suitability of PatRICIA programming model for developers of cloud-centric application business logic, but also its limitations in terms of reduced flexibility and expressiveness. Moreover, we mentioned that PatRICIA offers two distinct views on the application development process. However, in Chapter 3 we mostly focused on two main programming abstractions (Intents and IntentScopes), tacitly assuming that the Intent counterparts, i.e., monitor and control tasks are readily available. Therefore, the main motivation for the work presented in this chapter is to provide a suitable development support (withstanding the previous limitations, but providing a suitable level of abstraction) for domain expert developers, who are mostly concerned with developing light-weight, low-level applications and components that execute in Edge devices. We refer to such components monitor and control tasks.

Recent advances in Edge computing have resulted in numerous approaches in terms of programming frameworks and middleware for developing application business logic suitable for resource-constrained IoT devices such as sensory gateways. However, such approaches mainly focus on hiding the heterogeneity of data sources (e.g., sensors) [28, 1, 41, 66, 69, 131], defining data processing schemes [28, 131], and dealing with mobility [28, 66, 41], privacy [69] and scalability [66, 131]. In spite these and other approaches that address similar issues, e.g., on communication protocols level [55] or by utilizing SOA principles [39, 62], enabling virtually exclusive access to the underlying devices, e.g., field bus sensors and supporting flexible application-specific customizations for such devices are still not fully addressed in the literature. This inherently prevents utilizing the Edge devices as generic execution environments that can be potentially shared by multiple IoT Cloud applications. Still in large-scale IoT Cloud systems, leveraging the computational resources of the Edge devices is especially important, as their currently untapped processing capabilities can be used to optimize IoT Cloud applications by making edge devices first-class execution environments, i.e., by moving parts of application logic away from cloud platforms towards the edge of IoT Cloud infrastructure.

In this chapter we introduce Data- and Control Points programming model – an extension of the PatRICIA framework (cf. Chapter 3) that provides a set of programming abstractions for developing common monitor and control tasks. The Data- and Control Points represent low-level channels to the sensors/actuators in an abstract manner and mediate the communication with the connected devices, e.g., digital, serial or IP-based. The supporting framework provides mechanisms which act as multiplexers of the data and control channels, thus enabling the device services to have their own view of and define custom configurations for such channels, e.g., sensor poll rates, data units or data stream filters. By providing an illusion of an exclusive access to the underlying devices, our framework supports execution of multiple applications within a single Edge device.

The remainder of the chapter is structured as follows: In Section 4.1 we outline the framework architecture; Section 4.2 introduces the Data and Control Points and presents the main concepts of the programming model; In Section 4.3, we describe the main runtime mechanisms of the edge-devices framework. Finally, Section 4.5 concludes this chapter.

4.1 Overview of DRACO Framework

The main aim of DRACO (*Data And Control pOints*) framework is to facilitate the development of common monitor and control tasks in IoT Cloud systems. These task are the main building blocks of edge-device applications/services and the main constituents of reusable domain libraries. Generally, such libraries form the cornerstone for building higher-level cloud-centric IoT Cloud applications. To support domain experts in developing such domain libraries, comprising reusable monitor and control tasks, our framework provides a programming model and a set of runtime mechanisms, which constitute Edge device runtime. To this end, DRACO offers support to allow for virtually exclusive access to the underlying devices and enables flexible customizations of such devices. This means that the applications can have their "own view" of and precisely define the behavior of low-level devices. In Section 4.2, we describe DRACO's application-level support in more detail.

Figure 4.1 shows a high-level overview of the DRACO framework. In general, our framework follows a layered architecture and runs inside resource constrained Edge device, enabling local execution of device-level applications. In a broader sense it acts as an interlayer between low-level devices such as sensors and actuators and the high-level services which are executed on cloud platforms. The main layers of DRACO framework include: i) *Edge device middleware layer*, ii) *Application layer* and iii) *Cloud connectivity layer*. In the following we briefly describe each of these layers.

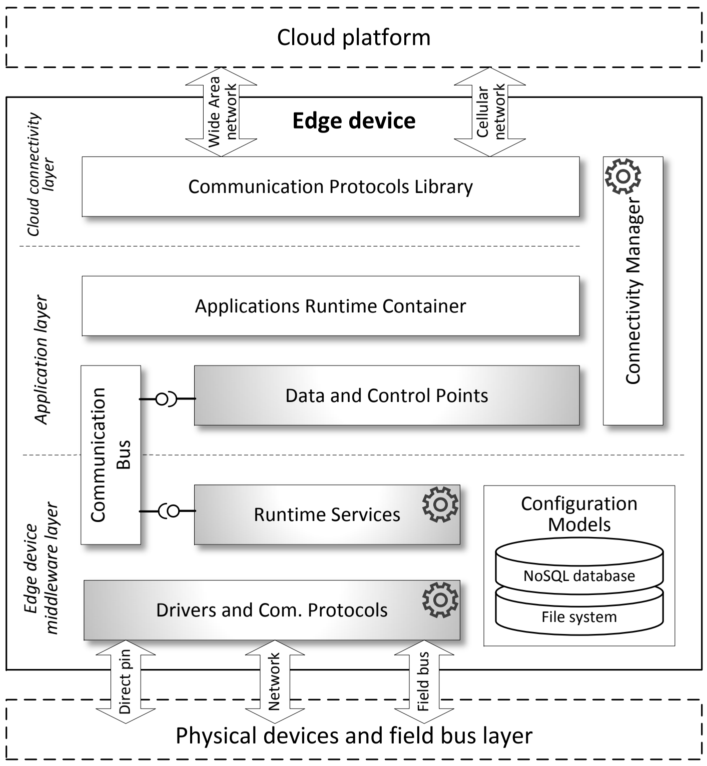


Figure 4.1: DRACO high-level architecture overview.

Starting from bottom up, the *Edge device middleware layer* is generally responsible to mediate communication with the underlying physical devices, maintain configuration models and provide and execution runtime for the monitor and control tasks. To enable communication with the physical devices this layer provides *Drivers and Com. Protocols* component. Its main responsibility is to provide the supporting driver implementations, which enable direct communication with the devices, e.g., via general purpose IO (GPIO) pins, field bus communications over protocols such as I^2C or ModBus, or communication over IP-based networks. Th Edge device middleware layer also provides *Configuration Models* repositories such as light-weight NoSQL database. The configuration models are stored locally in the device and among other things they specify how the underlying devices are connected. For example, in case of direct pin connection such models contain meta-data such as pin class (e.g., analog in), name and hardware-related data, e.g., multiplexer addresses or value correction constants. Finally, the *Runtime Services* constitute the tasks execution runtime and provide the sporting runtime mechanisms for Data and Control Points. This component is discussed in more detail in Section 4.3.

The next layer is the *Application layer* and its main purpose is to provide application development and execution support. The crucial part of this layer is the *Data and Control Points* component. It provides concrete implementation our programming model's abstractions and APIs, which are used by domain expert developers, as we describe in more detail subsequently. Further, the *Applications Runtime Container*, provides an execution runtime for the Edge-device applications. It is important to mention that the DRACO framework does not impose many limitations regarding the application model. For example, such applications can be based on OSGi container or even stand alone applications. In the current prototype, we rely on a striped-down JVM (based on Oracle Compact Profiles) to run the programming model, thus the framework only requires the application runtime to be JVM compatible.

The *Connectivity Manager* is a cross cutting component (between the *Application layer* and *Cloud connectivity layer*), which provides a flexible mechanisms for the Edge-device applications and services to communicate with the cloud. The Connectivity Manager relies on the *Communication Protocols Library* to offer a set of higher-level communication protocols such as CoAP or MQTT to the applications. It supports the applications to dynamically configure and utilize the available protocols, without having to dela with the low-level implementation details. In this chapter we do not discuss the *Cloud connectivity layer* to more detail, since it is out of scope of DRACO framework, but we provide a more detailed discussion about it in Chapter 7. In the remainder of this chapter, we mainly focus on describing Data and Control Points, which is the programming model provided by the DRACO framework, together with its major supporting runtime mechanisms.

4.2 Data and Control Points: A Programming Model for Edge Devices

The main purpose of the Data and Control points is to support domain expert developers to implement the light-weight monitor and control tasks, which are executed in edge devices. Generally, a control task is any permissible sequence of actuating steps which can be used to control physical devices, via the actuators they expose. Further, a monitor task includes processing, correlation and analysis of sensory data streams to provide meaningful information about the state changes of the underlying devices or the surrounding environment.

In our framework we envision two distinct usage patterns for the Data and Control Points and the aforementioned tasks. First, they can be used to develop “stand alone” edge-device applications, which do not necessarily depend on the cloud. An example of such application would be a logging application that reads sensory data and stores it locally at device side, e.g., in a light-weight database. In this case, the tasks behave similarly to application services. Second, they can be used to develop domain libraries. In this context a domain library contains a set of reusable tasks that are responsible to encapsulate domain-specific knowledge, most notably domain model and common behaviors, in a reusable manner. For example, a building automation expert developer could develop such a library to facilitate development of higher-level functionality for building management systems.

4.2.1 Main programming abstractions and application model

Data and Control Points represent and enable management of data and control channels (e.g., device drivers) to the low-level sensors/actuators in an abstract manner. Generally, they mediate the communication with the connected devices (e.g., digital, serial or IP-based), enable application-specific customizations of the channels and also implement communication protocols for the connected devices, e.g., Modbus, CAN or I^2C.

Figure 4.2 shows a simplified UML diagram of the main components of our programming model. From the figure we notice that the EdgeApplication contains multiple Tasks. Further such tasks can have multiple DCPoints associated with them. The DCPoint is an abstract class which provides main operators and lifecycle management hooks for the Data and Control Points. Both DataPoints and ControlPoints inherit from this component and encapsulate the specialized behavior for reading sensory data (DataPoints) and preforming the actuations (ControlPoints). In general, DCPoints allow the developers to perform concurrent reads and writs, regardless of whether the low-level drivers support sequential or concurrent reads and writes. This means, in case of DataPoints, it is possible for multiple applications to read (receive updates from) the same sensor simultaneously, by configuring and instantiating their own DataPoints. In this way the applications have an impression of exclusive usage of the available devices. Another important feature of DCPoints is that they enable developers to configure custom behavior of underlying devices. For this purpose each DCPoint can have a ConfigurationModel associated with it. For example, an application can configure sensor poll rates, activate a low-pass filter for an analog sensory input or configure unit and type of data instances in the stream. However, there are physical limitation, which need to be considered, such as a sensor might sample data at a different rate then specified in the ConfigurationModel for the DataPoint abstracting the sensor. The most important case is when the poll interval specified by an application is shorter than sensor's minimum interval. In this case the corresponding DataPoint issues a warning to the application (e.g., not supported configuration), but it resends the last available reading, given the configuration, until a new fresh reading is available. This enables developers to handle such situation dynamically, while allowing the applications to run without runtime interrupts.

The most important concept supporting the DCPoints are the VirtualBuffers, which are provided and managed by our framework. Our framework supports M to N mappings between the DCPoints and the VrtualBuffers. In general, such buffers enable virtualized access to and custom configurations of underlying sensors and actuators. They act as multiplexers of the data and control channels, thus enabling the device applications to have their own view of and define custom configurations for such channels. To this end, the VitualBuffers wrap the DeviceDrivers and share a common behavior with them, inherited through the Component Interface. For example, they can be initialized, shutdown and released. Both buffers and drivers lifecycle are managed by the VirtualBuffersManager.

The DeviceDrivers Package contains a set of driver implementations. For readability purposes, in the figure we only show the component for I^2C protocol, but each implemen-

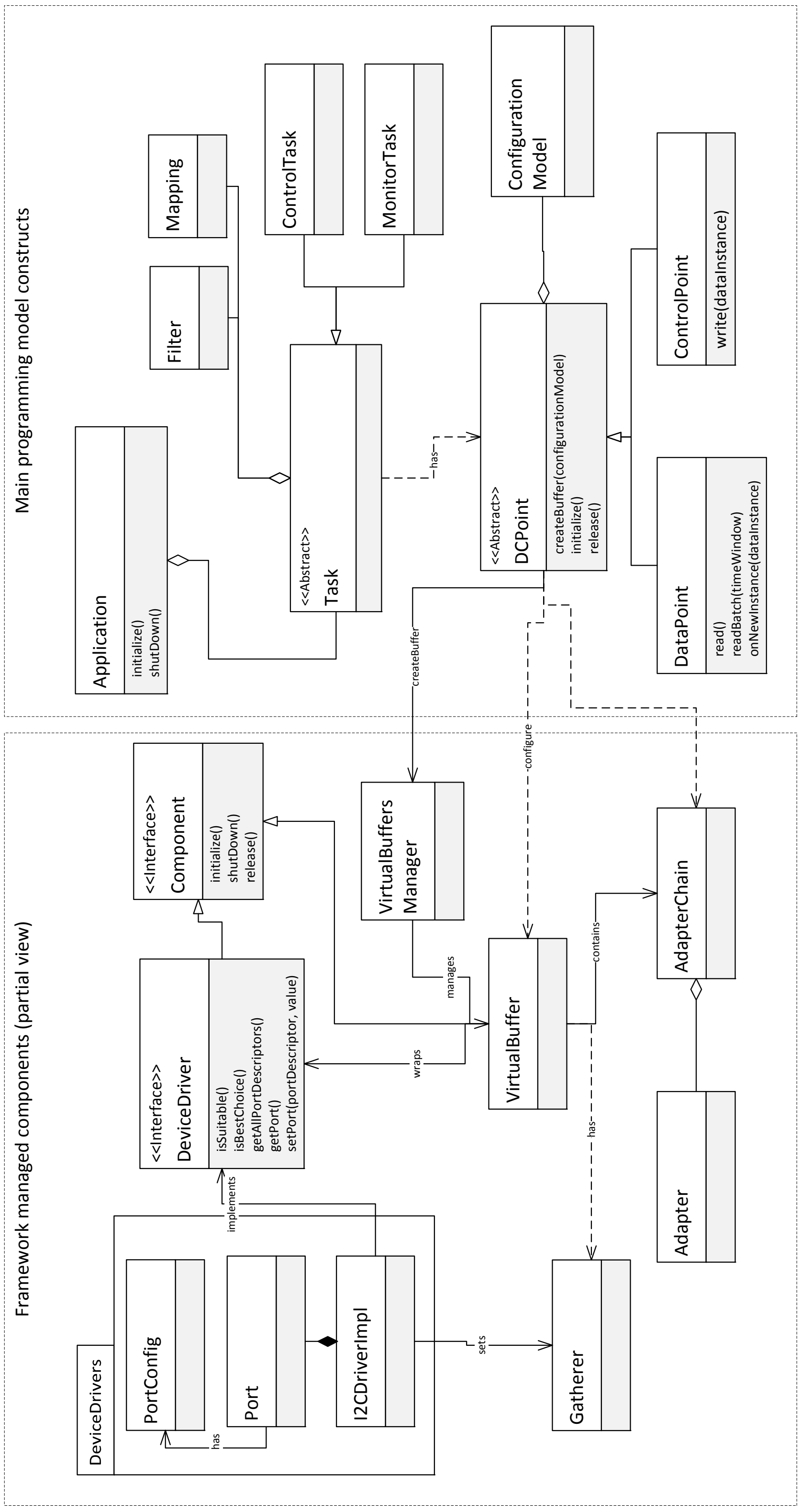


Figure 4.2: Simplified UML diagram of Data and Control Points.

tation follows similar principle. It contains a set of Ports, which is a framework internal representation of devices attached to the bus. Such Ports are dynamically instantiated by the VirtualBuffersManager at device bootup during driver initialization phase, based on the provided PortConfig. At the moment, PortConfig is specified as a JSON file that contains the meta-data such as port class (e.g., analog in), name and hardware-related data, e.g., multiplexer address or value correction constants. A limitation of the current prototype implementation is that it does not support dynamic device reconfiguration, meaning that if low-level configurations change the framework runtime servicers must be restarted.

Moreover, a virtual buffer references a set of Gatherers and can contain an optional AdapterChain. Generally, a gatherer is a higher level representation of a port. For example, in case of a sensing device (DataPoint) the gatherer represents the most resent value of the hardware interface. The principle for ControlPoints is similar to the one for the sensing devices. The only difference is that in case of actuation request the gatherers act as serializes instead of observers. However, due to possible conflicts, things are more complex here. At the moment we only provide a rudimentary priority-based conflict resolutions where different ControlPoints can have different priorities. To support application-specific configurations such as sensor poll rate, filters or scalers, each virtual buffer can have an AdapterChain. Adapter chains reference different Adapters, which are specified and parametrized via DCPoint's ConfigurationModel. For example, a raw sensing value is passed through such adapter chain before being delivered to a DataPoint. However, as discussed above, the adapters cannot account for physical limitations such as sensor resolution or refresh rate.

4.2.2 Application data model

Besides supporting development of monitor and control tasks the Data and Control Point enable the domain expert developers to define a custom application data model. Figure 4.3 depicts a simplified UML data model of the DCPoints. It can be seen as a meta model that enables applications to define a custom data (domain) model. This is especially important for defining groups of DCPoints that represent some logical entity in the physical environment. For example, this model can be used to describe a complex device, which contains multiple sensors or an application-specific domain model entity, e.g., room. To this end, the DataInstance acts as a wrapper of a sensory reading (value) and enriches it with additional information such as timestamp. Moreover, the DataType enables defining custom data instances types. It extends the built-in Java types and provides additional information about the data instance such as its unit (e.g., Kelvin). In this context, the most important feature provided by our framework is the support for complex data types and complex data instances. A complex type is represented as a record, hence it consists of named fields that have again have a type associated with them. Similarly, a complex data instance is a combination of simple data instances and it additionally provides a processing hook, which allows the developers to specify additional filters or aggregations of the data instances. For example, it could contain a functionality to compute an average of the requested sensory readings. However, in such cases, it is developer's responsibility to deal with the instances compatibility, e.g., their units. More importantly our framework provides support for synchronizing the individual readings within a complex data instance. Therefore, developers only need to declare a complex instance and the framework takes care of collecting the relevant readings from multiple streams and delivering the complex instance to the application when it is fully initialized or updated.

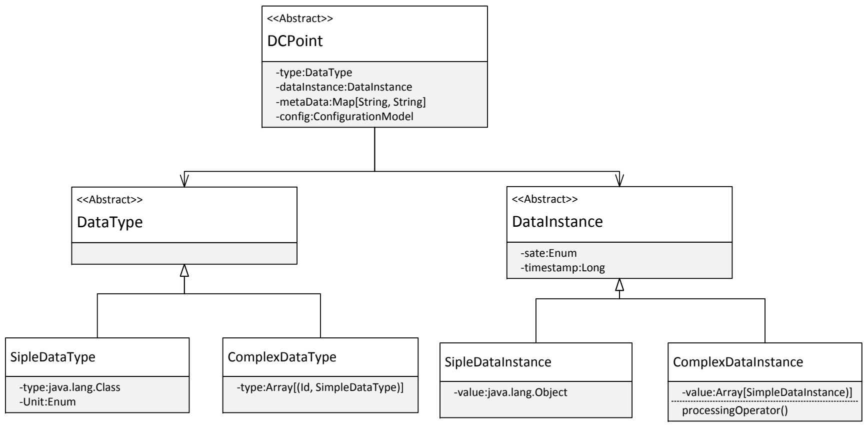


Figure 4.3: Simplified UML data model.

4.2.3 Application-level programming constructs

Listing 4.1, gives a general example of how developers define a DataPoint. It shows a data point with one stream of simple data instances that represent, e.g., vehicle's tire speed, based on the required sensor properties. By default the data points are configured to asynchronously push the data to the applications at a specific rate, which can be

```
DataPoint dataPoint = new DataPoint();

// Query available buffers
Collection<BufferDescription> availableBuffers
        = dataPoint.queryBuffers(new SensorProps(...));

// Assign the buffers to the data point
dataPoint.assign(availableBuffers.get(0));
dataPoint.setPollRate(300);
dataPoint.addCallback(this);

// Event handler
void onNewInstance(DataInstance di){
  ...
}
```

Listing 4.1: A DataPoint with callback handler.

configured as shown in the example. The application defines a call-back handler, which contains some data processing logic, e.g., based of complex event processing techniques. Additionally, the data and control points offer a `read` operator that can be used to sequentially (or in batch) read a set of instances from a stream, e.g., in order to perform more complex stream processing operations.

```
 1 //Configure a custom data channel
 2 BufferConfig bc = new BufferConfig("voltage_in");
 3 bc.setClass(BufferClass.SENSOR);
 4 bc.getAdapterChain().add(new ScalingAdapter(0.0,100.0,10.0));
 5 bc.getAdapterChain().add(new LowpassFilter(0.30));
 6 BufferManager.create("lowpass-scaled", bc);

 7 //Define diagnostics model
 8 DataPoint diagnostics =  DataPoint.newComplexInst("lowpass-scaled","voltage_in");
 9 DataInstance di = diagnostics.read();
10 //Log the diagnostics data locally
11 ...
```

Listing 4.2: Custom configuration of DataPoints.

Listing 4.2 shows a more complex example of a custom data point, together with a simple diagnostics data model. The diagnostic data contains raw engine voltage readings and scaled voltage readings with low-pass filter, e.g., possibly indicating that something is taking the power away from the motor. The listing shows how to define a custom (partial) configuration for the data point. In this case, we define a scaling adapter and a filter, which are added to data point's adapter chain, as shown in lines 2-6. After creating a custom data point (virtual sensor) (line 8) application can treat this sensor as any other sensor. Finally, the example shows how to synchronously read a data instance from the newly created virtual sensor. Storing the data is omitted for readability purposes.

4.3 Main Runtime Mechanisms of the DRACO Framework

In this section, we present the most important mechanisms provided by the DRACO's RuntimeServices (cf. Figure 4.1) in order to provice runtime support for the Data and Control Points. We mainly focus on describing the DCPoints and VirtualBuffers lifecycle, as well as on discussing the main information flow within the DRACO framework.

4.3.1 Lifecycle of Data and Control Points

In the DRACO framework both the DCPoints and the VirtualBuffers have clearly defined lifecycles, which are entangled and mainly share a common behavior. For example, when a DCPoint is created it is associated with one or more VirtualBuffers, as shown in Figure 4.2 and if this DCPoint is released the corresponding buffers are also automatically released by the framework (given that no other DCPoint is referencing them). The main difference in lifecycle management of these components is that the DCPoints are mainly managed by applications (cf. Figure 4.2 right-hand side), while the DRACO framework manages the VirtualsBuffers and their dependencies (cf. Figure 4.2 left-hand side).

Figure 4.4 illustrates the main phases of the VirtualBuffers lifecycle. Depending on the configuration a VirtualBufer is initialized either when a device is connected or when explicitly requested by a DCPoint. The initialization phase includes allocating a buffer instance, creating a corresponding Gatherer and performing configuration directives, specified in the DCPoint configuration model. The latter usually includes creating an adapter chain with corresponding scalers, filters and adapters. This part is automatically handled by the RuntimeServices and it happens transparently to the applications. After the initialization is complete the buffer transitions to the Ready state.

Generally, when in the Ready state VirtualBuffers are discoverable and can be queried by the applications via the DCPoint APIs. Also at this point the corresponding Gatherer is automatically started by the framework and it starts gathering the data from the underlying device or in case of a ControlPoint it is ready to receive serialization requests. After a DCPoint obtains a reference to the buffer it can decide to start it (e.g., to open a data stream) or the buffer is automatically started by the framework if the callback object is provided.

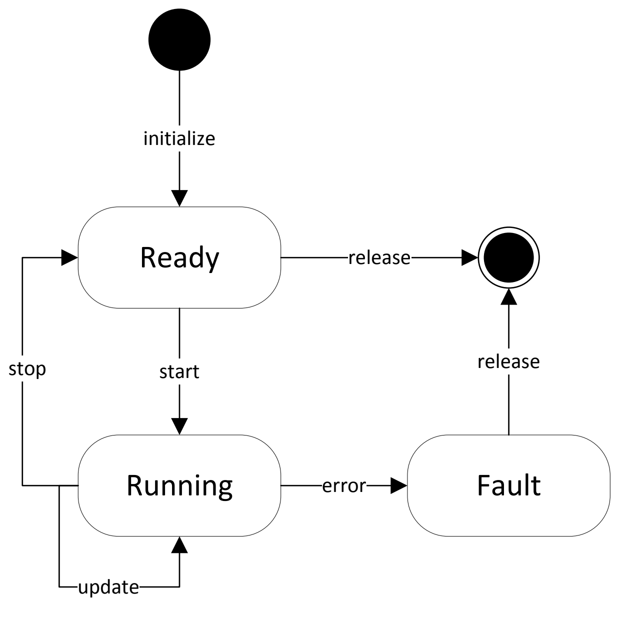


Figure 4.4: VirtualBuffers lifecycle overview.

After a successful start both the VirtualBuffer and the DCPoint are in the Running state. In this state the DCPoint receives periodic updates from the underlying buffer or it explicitly reads the buffer state via the read operator, i.e., sets a new state via the write operator. Finally, there are two ways to release a DCPoint/VirtualBuffer. An a pplication can manually stop and release them after it has finished using the DCPoint or in case an error occurs, e.g, device disconnected, the VirtualBuffer is moved to a Fault state. When it the Fault state the buffer notifies the DCPoint about the error after which it is automatically released by the framework.

4.3.2 Main Information flow

Figure 4.5 shows the main steps of the information flow within the DRACO framework. For readability purposes, the figure only illustrates the information flow of a sensory reading (DataPoint), but the framework behaves in a similar fashion for the ControlPoins, with a main difference that the direction of the information flow is reversed.

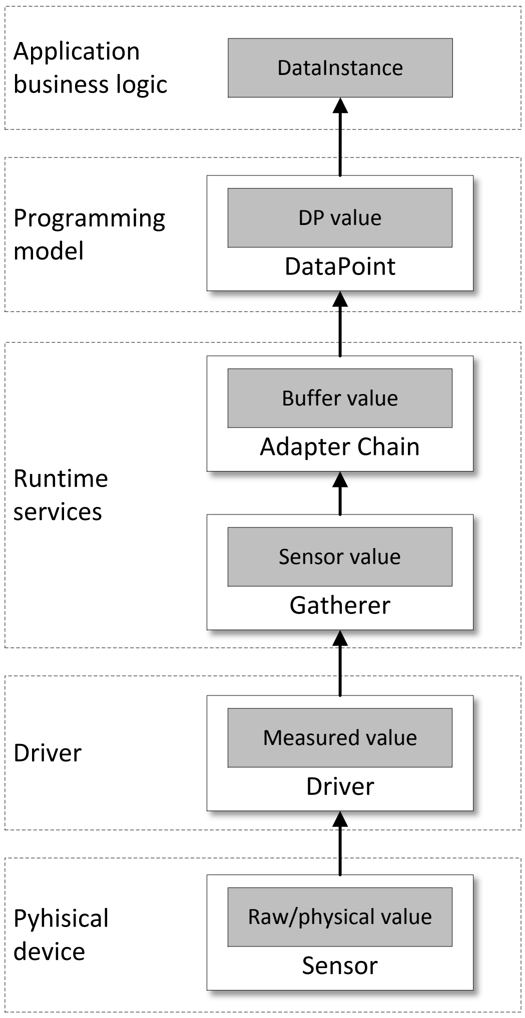


Figure 4.5: Information flow of a sensory reading.

When an application requests a sensory reading, initially the *raw physical value*, e.g.

temperature is measured by a sensor that is connected to the Edge device, e.g., via a field-bus (e.g., RS422, CAN, etc). The raw value enters the framework through a driver. The driver handles the protocol on the field-bus and acquires the *measured value* from the sensor. In most cases the measured value is a linear function that is applied on the raw value, i.e., on a stream of bits and it scales the raw value between 0% and100% of the measurement range (provided in the PortConfiguration, cf. Figure 4.2). Next, the measured value has to be formatted to a common framework internal format, which is independent of the used driver. This is performed by the Gatherer and by default it formats the sensor value as a double. At this stage the sensor reading is formatted as a *sensor value*, which is universal understood by the framework. In the next steps the sensor value is propagated and processed through the AdapterChain, i.e., the framework applies application-specific adapters and filters on it. This transforms the sensor value resulting in a *buffer value* that is delivered to a DCPoint. Finally, the DCPoints creates a DataInstance with the required format, unit and type (cf. Figure 4.3) and delivers it to the calling application.

4.4 Evaluation & Prototype Implementation

4.4.1 Prototype Implementation

The current prototype is implemented in Java programming language (based on Java SE Embedded). The framework is designed to run on a stripped-down JVM and for this purpose we have created a lightweight compact profile JVM runtime specifically tailored for constrained devices. The complete source code and supplement materials providing more details about current framework implementation are publicly available in Git Hub[1].

4.4.2 Experiments

Test bed Gateway and Experiments Setup

In order to evaluate our DRACO framework, we built a test physical gateway (cf. Figure 4.6). The getaway is based on Raspberry Pi 2, with ARMv7 CUP and 1Gb of RAM. They run Raspbian Linux 8 (based on Debian “Jessi”) on Linux Kernel 4.1. Further it contains a serial I2C bus system, which is used to attach the test sensors (5 digital inputs and 6 analog inputs, which are used to simulate changes in sensor readings) and actuators (8 light-emitting diodes (LED)).

In this chapter we mainly focus on evaluating our framework regarding its resources usage in order to validate its suitability for resource-constrained Edge devices. To this end we have developed several edge applications, which are also available in the aforementioned Git Hub repository. The qualitative evaluation of our programming model is discussed in detail in Chapter 5.

[1]http://github.com/tuwiendsg/SoftwareDefinedGateways

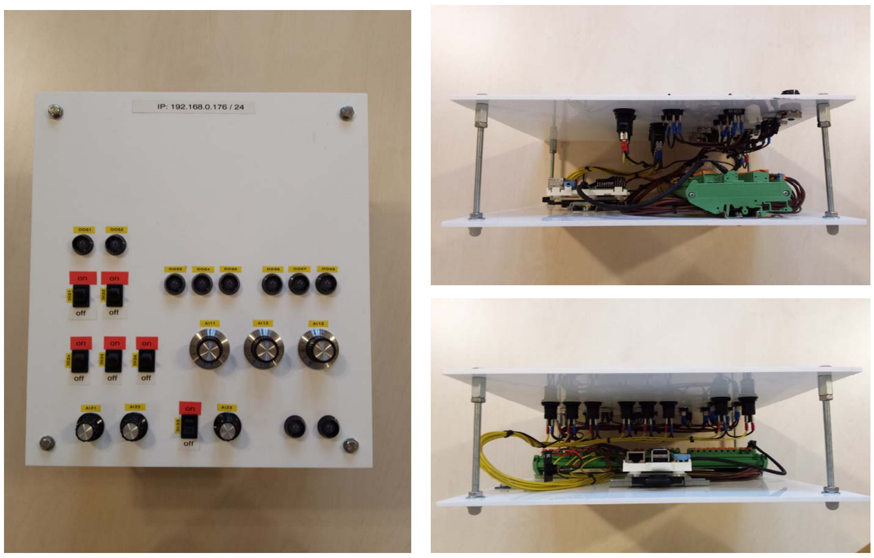


Figure 4.6: Testbed gateway for Data and Control Points.

4.4.3 Experiments Results

For the evaluation purposes we have developed two applications. First application (LogApp) runs inside the test bed gateway, collecting all the sensory inputs (both analog and digital), logging them locally and displaying the changes in sensors readings on stdout. It defines several tenths of the Data Points, which have different configurations such as scaling adapters and filters for the connected sensors. It also logs the raw sensory readings. Second application (ActApp) is also running in the gateway and its main purpose is to demonstrate different actuations, based on the changes in sensory readings. For this purpose it creates several Data Points (actuation triggers) and also several Control Points, which are responsible to perform actuations, i.e., in this case turning on/off the LEDs.

Figure 4.7 and Figure 4.8 show memory and CPU usage of the LogApp. Initially (Figure 4.7) we notice that the DRACO framework consumes below 5% of the CPU when no applications are running. The first spike in CPU consumption happens when the LogApp application is started. The reason for this is that at this point the application instantiates its Data Points and requests the framework to allocate the corresponding VirtualBuffers, AdapterChains, etc. This is also reflected in Figure 4.8, where we observe an increase in RAM of around $1Mb$. After this point in time the application is running (processing and logging the changes in the sensory readings). These changes are simulated by manually adjusting the analog inputs, i.e., by alternating the digital switches. In general, the application is mostly consuming less then 10% of CPU and its memory usage is fairly stable (with only minimal increase mainly due to created data instances). The smaller spikes in CPU usage represent noticeable changes in sensory readings (e.g.,

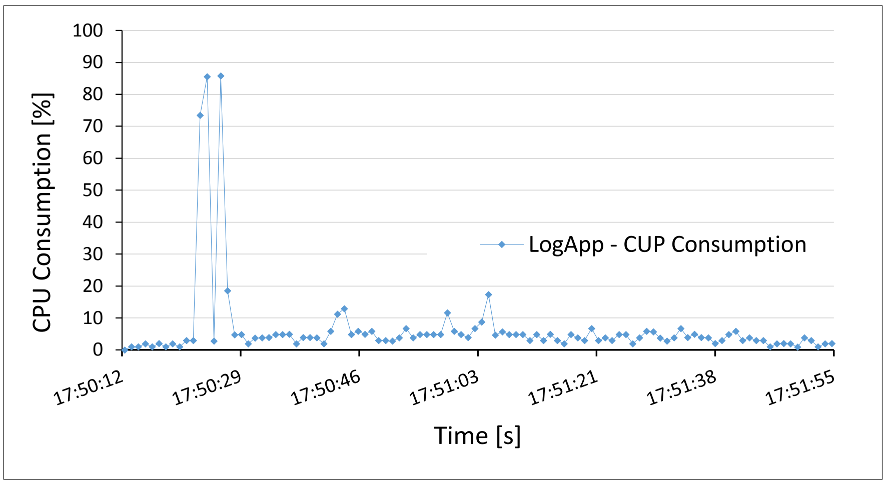


Figure 4.7: CUP consumption of the example logging application (LogApp).

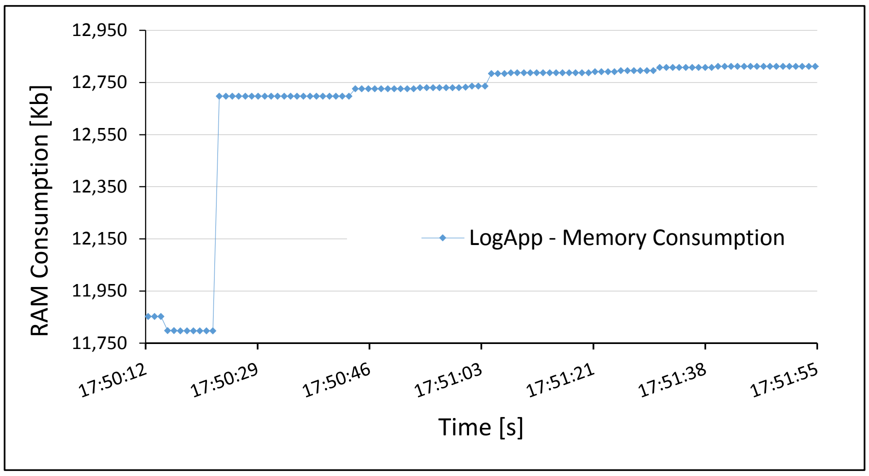


Figure 4.8: Memory usage of the example logging application (LogApp).

several knobs are rotated). However, even when all the knobs are affected, effectively forcing all the buffers to perform their individual data processing actions, the CPU usage remains below 20%. Moreover, the increased CPU usage is temporary and both the application and the framework quickly return to normal resource usage. Similar things can be observed in Figure 4.8, as the memory usage during the observation time remains below 13Mb. Therefore, we notice that even for applications which use relatively large number of Data Points the resource consumption can be seen as acceptable for Edge devices.

Similar results can be observed in Figure 4.9 and Figure 4.10, where we show the performance of the ActApp, which besides the Data Points also utilizes the Control

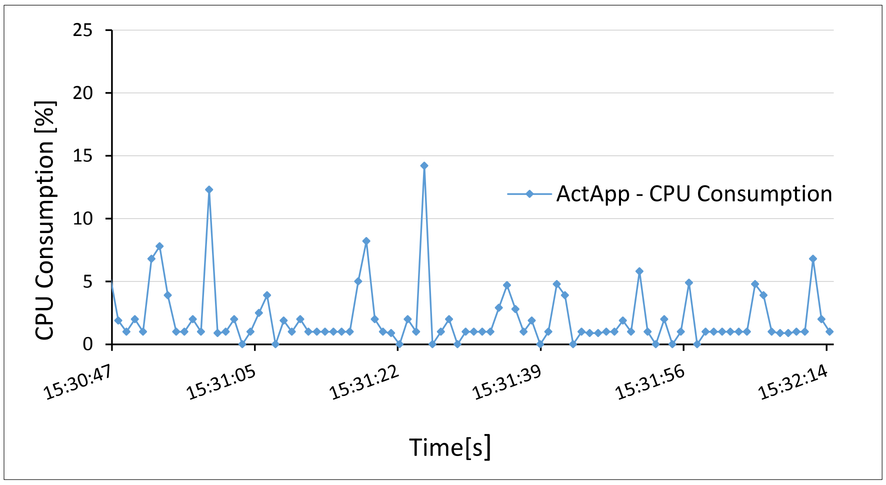


Figure 4.9: CUP consumption of the example actuation application (ActApp).

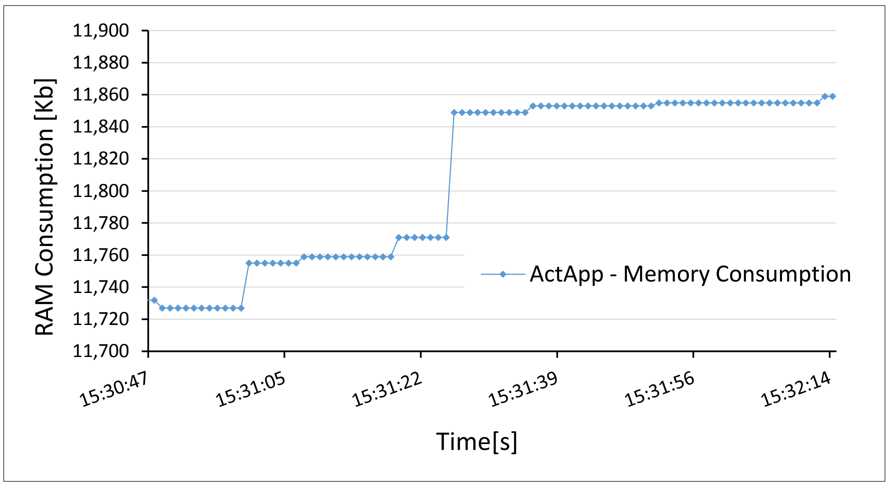


Figure 4.10: Memory usage of the example actuation application (ActApp).

Points. The main differences are reflected in the overall smaller memory consumption (below $12Mb$) and slightly higher CPU spikes. The main reason for the former is that ActApp instantiate smaller amount of Data Points. The latter is mainly due to the fact that the changes in sensory readings trigger actuations, which also require some processing to be done by the framework, such as serializing the Control Points instances. Also here the changes in sensor inputs were manually simulated by the test sensor knobs and switches. We notice that the general behavior of the Control Points can be seen as satisfactory.

Finally, it is worth noticing that for the both experiments the memory and CPU usage was measured on the process level (i.e., entire JVM). Also, albeit small, in both cases we notice a constant increase in memory usage. This is generally not a desired behavior (e.g., since it can indicate a memory leak). In this case, however, the reason for such behavior is that the figures do not show the garbage collection of old data instances. Additionally, when an application exits it releases all its resources. This can be seen in the first couple of seconds in Figure 4.8, where we stopped another application before starting the LogApp.

4.5 Conclusion

In this chapter, we introduced Data and Control Points, a programming model for resource-constrained Edge devices. The main aim of the presented framework is to facilitate development of light-weight, edge-centric applications as well as domain-specific libraries that contain reusable, generic monitor and control tasks and domain models. We discussed how our programming model complements the high-level programming abstractions (cf. Chapter 3), by providing development support for domain expert developers. We presented the main features of the supporting DRACO framework: providing a virtually exclusive access to the connected sensors and actuator; enabling application-specific view on such devices; and supporting flexible customizations of the low-level sensing and actuating channels. Finally, we demonstrated feasibility of our proof-of-concept prototype and suitability of its runtime mechanism for the Edge devices, in terms of optimized and relatively small resource consumption.

CHAPTER 5

A Unifying Programming Framework and Methodology for Everything-as-Code in IoT Cloud Systems

Emerging IoT Cloud systems extend the traditional cloud computing systems beyond the data centers and cloud services to include a variety of edge IoT devices such as sensors and sensory gateways. Such systems utilize the IoT infrastructure resources to deliver novel value-added services, which leverage data from different sensor devices or enable timely propagation of decisions, crucial for business operation, to the edge of the infrastructure. On the other side, IoT Cloud systems utilize cloud's theoretically unlimited resources, e.g., compute and storage, to enhance traditionally resource constrained IoT devices.

In order to enable development of IoT Cloud systems, existing research and industry have produced numerous infrastructure, platform and software services [160, 134, 29, 139, 39, 129]. These advances set a cornerstone for proliferation of (unified) IoT Cloud platforms and infrastructures, which offer a myriad of IoT Cloud capabilities and resources. One of the promising approaches to facilitate development of IoT Cloud applications are software-defined IoT Cloud systems. As thoroughly discussed in Part II of this thesis, we introduce a conceptual model for software-defined IoT Cloud and supporting middleware, in order to facilitate utility-oriented delivery of the IoT Cloud resources and enable automated and logically centralized provisioning of the geographically distributed IoT Cloud infrastructure. Generally, the software-defined IoT Cloud systems abstract from low-level resources (e.g., hardware) and enable their programmatic management through well-defined APIs. They allow for refactoring the underlying IoT Cloud infrastructure into finer-grained resource components whose functionality can be (re)defined in software, e.g.,

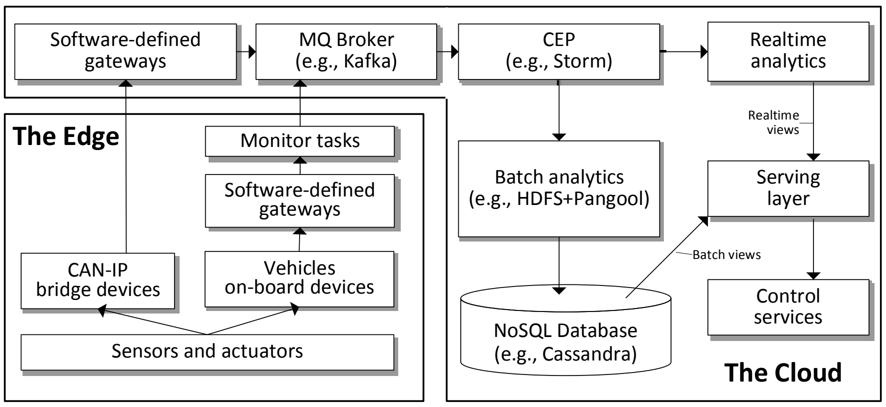


Figure 5.1: Overview of FMS architecture and deployment.

applications, thus enabling more efficient resource utilization and simplifying management of the IoT Cloud systems. However, most of the contemporary approaches dealing with IoT Cloud are intended for platform/infrastructure providers and operations managers. Therefore, from the developer's perspective there is a lack of structured, holistic approach to support development of the IoT Cloud systems and applications.

In this chapter we introduce SDG-Pro – a novel programing framework for software-defined IoT Cloud systems, as the final part of the first contribution of this thesis. It provides a unified, programmatic view for the entire development process (*everything as code*) of IoT Cloud applications, thus making it more traceable and auditable. This chapter substantially extends and refines our previous work presented in Chapter 3 and Chapter 4. The SDG-Pro framework builds on the previously introduced concepts (Intents, IntentScopes, Data- and Control Points) and extends them by introducing comprehensive programming support for unified development, provisioning and governance of IoT Cloud applications. This chapter mainly focuses on application-level support, i.e., programmatic provisioning and governance of IoT Cloud applications. Comprehensive provisioning and governance support for IoT Cloud systems are thoroughly discussed respectively in Part II and Part III of this thesis.

The remainder of the chapter is structured as follows. In Section 5.1, we describe a motivating scenario and main research challenges; Section 5.2 gives an overview of the development methodology and outlines the design of the SDG-Pro framework; Section 5.3 presents the framework's programming model; In Section 5.4, we outline SDP-Pro's runtime support; Section 5.5 presents our experiments; Finally, Section 5.6 concludes the chapter.

5.1 Motivation and Research Challenges

5.1.1 Scenario

Let us consider our real-life FMS scenario (cf. Chapter 2) form a different perspective. In Chapter 3, we approached the FMS exclusively from application perspective, i.e., we mainly focused on concrete examples of FMS applications, tacitly assuming that the main application elements (e.g., monitor tasks) are deployed and readily available in the Edge devices. In this chapter, we adopt a more general view on the FMS, approaching it from a perspective of a holistic system.

Figure 5.1 gives a high-level overview of the common elements in FMS's architecture and deployment. For readability purposes, we only show the common propagation of the sensory data within the FMS (depicted with arrows) and omit control actions. The FMS runs atop a complex IoT Cloud infrastructure, which includes a variety of IoT Cloud resources. It is a large-scale, geographically distributed system that has nontrivial deployment topologies. The FMS deployment topologies span across the entire IoT Cloud infrastructure, i.e., from large data centers to the edge of the network, resulting in complex dependencies among the business logic services, but also between such services and the underlying infrastructure. Therefore, during application development, developers need to consider numerous infrastructure resources and their properties such as availability of sensors, devices ownership and their location.

The FMS applications perform a variety of analytics and are mostly characterized by a reactive behavior. They receive (monitoring) data, e.g., a change in vehicles' operation and, as a response, perform (control) actions. Such monitoring and control tasks (cf. Chapter 4) are executing in heterogeneous, dynamic FMS environment and interact with many geographically distributed vehicles and their low-level capabilities, e.g., engine control points. Further, FMS applications have different requirements regarding cloud communication protocols. For example, the fault alarms need to be pushed to the services, e.g, via MQ Telemetry Transport (MQTT) and vehicle's diagnostics should be synchronously accessed via RESTful protocols such as Constrained Application Protocol (CoAP) or Simple Measurement and Actuation Profile (sMAP). Therefore, from the developers perspective such tasks and capabilities need to be decoupled from the underlying physical infrastructure, but also easily specified, provisioned and managed programmatically (as code), without worrying about the complexity of low-level device services, communication channels and raw sensory data streams.

The currently limited development support regarding the FMS requirements and features (as discussed in Chapter 10), renders the development of its applications a complex task. Consequently, system designers and application developers face numerous challenges to develop IoT Cloud applications.

5.1.2 Research challenges

RC-1 – The development context of IoT Cloud applications has grown beyond writing custom business logic (e.g., services) components to also considering the involved IoT devices (e.g., their capabilities) as well as the deployment and provisioning of such services across the IoT Cloud infrastructure. The main reasons for this are complex and strong dependence of the business logic on the underlying devices (and their specific capabilities), novel (resource) features that need to be considered, such as device location and the heterogeneity of the utilized IoT Cloud resources. Unfortunately, developers currently lack suitable programming abstractions to deal with such concerns in a unified manner, from early stages of development.

RC-2 – IoT Cloud applications execute in very dynamic, heterogeneous environments and interact with hundreds or thousands of physical entities. Therefore, monitoring and controlling these entities in a scalable manner is another challenge for developers of IoT applications, mainly because they need to dynamically identify the scope of application's actions, depending on the task at hand, but also express its business logic independently of the low-level device capabilities.

RC-3 – The IoT Cloud applications mostly rely on common physical infrastructure. However, IoT Cloud infrastructure resources are mostly provided as coarse-grained, rigid packages. The infrastructure components and software libraries are specifically tailored for the problem at hand and do not allow for flexible customization and provisioning of individual resource components or runtime topologies. This inherently hinders self-service, utility-oriented delivery and consumption of IoT Cloud resources at fine granularity levels.

RC-4 – Due to dynamicity, heterogeneity and geographical distribution of IoT Cloud, traditional provisioning and governance approaches are hardly feasible in practice. This is mostly because they implicitly make assumptions, such as physical on-site presence, manual logging into devices, understanding device's specificities, etc., which are difficult, if not impossible, to achieve in IoT Cloud systems. In spite of this, techniques and abstractions, which provide a programmatic, conceptually centralized view on system provisioning and runtime governance are largely missing.

In the rest of the chapter, we introduce our SDG-Pro framework and focus on describing and evaluating its unifying programming model for IoT Cloud applications.

5.2 The SDG-Pro Framework and Development Methodology for IoT Cloud Applications

The main aim of our SDG-Pro (*Software-Defined Gateways Programing framework*)[1] is to provide programming support for IoT Cloud application developers, which offers a

[1]Software-defined gateways are thoroughly discussed in Part II of this thesis.

set of adequate programming abstractions to facilitate overcoming the aforementioned challenges. To this end, SDG-Pro, enables expressing application's provisioning, governance and business logic programmatically, in a uniform manner. By raising the level of programming abstraction, SDG-Pro reduces the complexity of application development, makes the development process traceable and auditable and improves efficacy and scalability of application development.

SDG-Pro's programming model is designed to enforce the main design principles of software-defined IoT Cloud systems (presented in Part II of this thesis) at application level, from the early development stages. It adopts development methodology we proposed in [71], extending it to provide programming abstractions that facilitate entire development lifecycle of the essential application artifacts.

5.2.1 Main Design Principles and Development Methodology for IoT Cloud Applications

Software-defined IoT Cloud systems are thoroughly discussed in Part II of this thesis. In the following, we give a brief overview of such systems and introduce their main concepts, relevant for the discussion in this chapter. Generally, software-defined IoT Cloud systems comprise a set of resource components, provided by IoT Cloud infrastructure, which can be provisioned and governed at runtime. Such resources (e.g., sensory data streams), their runtime environments (e.g., gateways) and capabilities (e.g., communication protocols and data point controllers) are described as software-defined IoT units. The software-defined gateways (cf. Figure 5.1) are a special type of such units and they are the main building blocks of IoT Cloud infrastructure, e.g., similar to VMs in cloud computing. In our conceptual design of software-defined IoT Cloud systems, such gateways abstract resource provisioning and governance through well-defined APIs and they can be composed at different levels, creating virtual runtime topologies for IoT Cloud applications. This enables opening up the traditional infrastructure silos and moving one step higher in the abstraction, i.e., effectively making applications independent of the underlying rigid infrastructure. The main design principles of software-defined IoT systems include:

Everything as code – All the concerns, i.e., application business logic, but also IoT Cloud resources provisioning and runtime governance, should be expressed programmatically in a unified manner, as a part of the application's logic (code).

Scalable development – The programming abstractions exposed to the developers need to support scalable application development, i.e., shield the developers from dealing with the concerns such as manually referencing individual devices or managing the low-level data and control channels.

API Encapsulation – IoT Cloud resources and capabilities are encapsulated in well-defined APIs, to provide a uniform view on accessing functionality and configurations of IoT Cloud infrastructure.

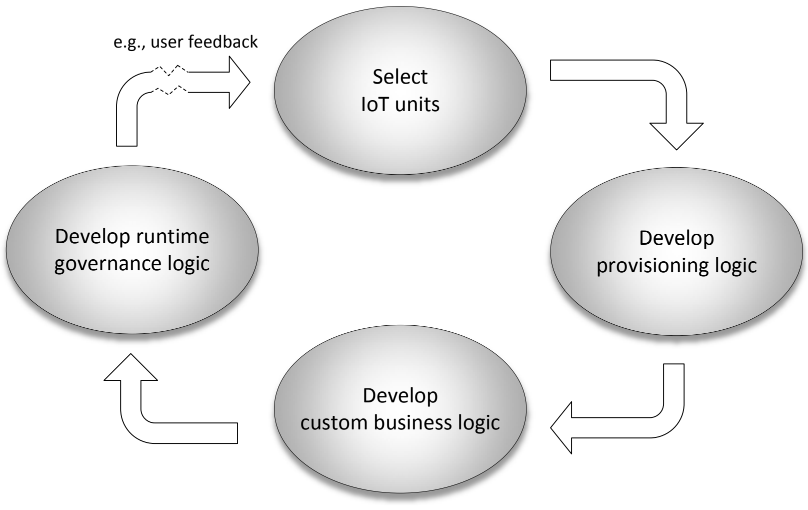

e.g., user feedback
Select IoT units
Develop provisioning logic
Develop custom business logic
Develop runtime governance logic

Figure 5.2: Most important steps in development methodology for software-defined IoT Cloud systems (partial view).

Declarative provisioning – The units are specified declaratively and their functionality is defined programmatically in software, using well-defined API and available, familiar software libraries.

Central point of operation – Enable conceptually centralized (API) interaction with the software-defined IoT Cloud system to allow for a unified view on the system's operations and governance capabilities (available at runtime), without worrying about low-level infrastructure details.

Automation – Main provisioning and governance processes need to be automated in order to enable dynamic, on-demand configuring and operating the software-defined IoT Cloud systems, without manually interacting with IoT devices (e.g., logging in the gateways).

As proposed in [71], building IoT Cloud systems includes creating and/or selecting suitable software-defined IoT units, provisioning and composing more complex units and building custom business logic components. This (iterative) development process is structured along four main phases (cf. Figure 5.2): i) Initially, developers need to design and implement the software-defined IoT units or obtain them form a third-party, e.g., in a market-like fashion. Among other things the IoT units support execution of the light-weight device services (monitor and control tasks), i.e., from the software engineering perspective they encapsulate such tasks, comprising domain libraries; ii) Next, the developers need to design and provision the required application topologies. This process includes implementing the dependencies among the business logic services, but also between such services and the underlying infrastructure; iii) Building custom business logic components mainly involves developing device services and implementing reactive

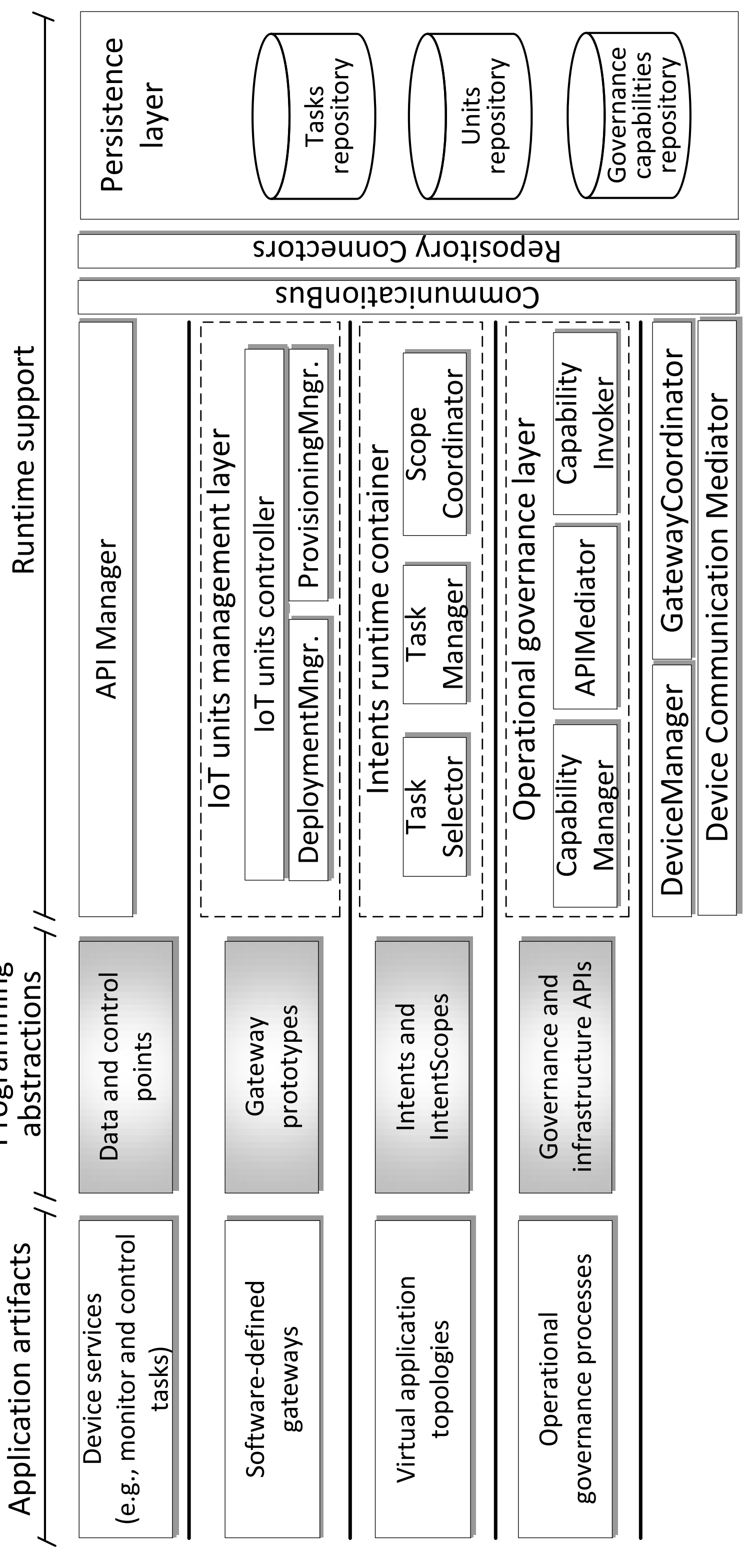


Figure 5.3: Overview of SDG-Pro's architecture and main artifacts of IoT Cloud applications.

business logic (e.g., cloud services) around the device services; iv) Finally, the developers implement operational governance logic for managing the IoT units during application runtime. The operational governance is discussed in Part III of this thesis. In this chapter we mainly focus on the application-level support for operational governance processes.

5.2.2 SDG-Pro architecture overview

Generally, the SDG-Pro framework is distributed across the clouds, i.e., large data centers, and “small" IoT devices, e.g., physical gateways or cloudlets. It is designed based on the microservices architecture, which among other things enables evolvable and fault-tolerant system, while allowing for flexible management and scaling of individual components. Figure 5.3 gives a high-level overview of SDG-Pro’s architecture and main IoT Cloud application artifacts. These artifacts can be seen as executables produced by the aforementioned development methodology. To support the development of such artifacts our framework provides a set of programming abstractions (depicted as gray components in Figure 5.3 and described later in Section 5.3) and runtime support mechanisms (Section 5.4).

The runtime mechanisms are part of the SDG-Pro’s Runtime support (cf. Figure 5.3), which underpins the programming abstractions exposed to the developers, i.e., provides an execution environment for IoT Cloud applications. It takes over a set of responsibilities such as placement of the software-defined gateways, their runtime migration and elasticity control, infrastructure topology management and application scope coordination. By doing so, it does most of “heavy lifting" on behalf of the applications, thus supporting the developers to easier cope with the diversity and geographical distribution of the IoT Cloud and enabling better utilization of the numerous edge devices.

Internally, our framework’s runtime support comprises several microservices, which can be grouped into: APIManager, IoT units management layer, Intents runtime container and Operational governance layer. The *APIManager* exposes governance capabilities and the low-level data and control channels from the IoT Cloud infrastructure to the applications via well-defined APIs and handles all API calls from such applications. The *IoT units management layer* provides mechanisms and agents to support instantiating, provisioning and deploying software-defined gateways programmatically and on-demand. The *Intents runtime container* is responsible to handle incoming application requests (Intents) and select, instantiate and invoke device services (tasks), based on the information provided in the intents. It also enables applications to dynamically delimit the scopes of their actions, by providing support for IntentScopes resolution. Finally, the *Operational governance layer* supports execution of the governance processes by enabling remote invocation of governance capabilities and mapping of API calls on underlying devices via governance agents. The SDG-Pro’s runtime support is discussed in more details in Section 5.4. In continuation, we first present SDG-Pro’s programming model.

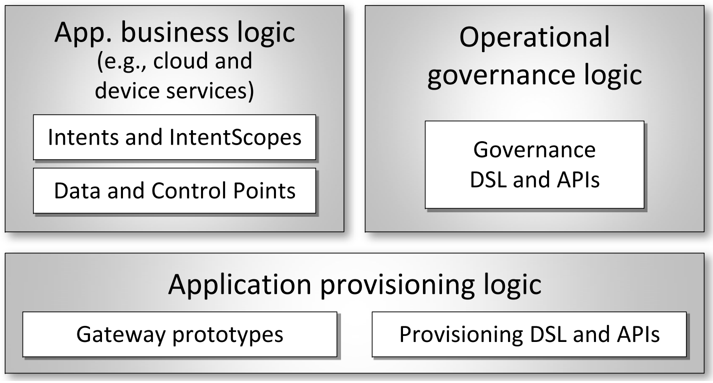

App. business logic
(e.g., cloud and device services)
Intents and IntentScopes
Data and Control Points
Operational governance logic
Governance DSL and APIs
Application provisioning logic
Gateway prototypes
Provisioning DSL and APIs

Figure 5.4: High-level overview of IoT Cloud application structure.

5.3 SDG-Pro's Programming Model

5.3.1 Structure of IoT Cloud applications

The main purpose of our programming model is to provide a programmatic view on the whole application ecosystem, i.e., the full stack from the infrastructure to software components and services. The main principle behind our programming model is *everything as code*. This includes providing support for writing IoT Cloud applications' business logic, as well as representing the underlying infrastructure components (e.g., gateways) at the application level and enabling developers to programmatically determine their deployment and provisioning. Generally, this principle denotes that all the concerns, i.e., application business logic, but also resource (i.e., software-defined gateways) management of an IoT Cloud application, should be expressed as a part of the application's logic (code) in a unified manner. Among other things, this includes representing the underlying infrastructure components (e.g., gateways) at the application level and enabling their provisioning and runtime governance through well-defined APIs. Figure 5.4 shows a component diagram with the logical structure of IoT Cloud applications. The main components of such application include: custom business logic components (e.g., cloud services and device services); resource provisioning and deployment logic (custom or stock component provisioning); and operational governance logic.

5.3.2 Programming support for business logic services

In SDG-Pro we distinguish between two types of business logic services: *device-level services and cloud services*. The *device-level services* are executed in IoT devices and implement control and monitor tasks. For example, a monitoring task includes processing, correlation and analysis of sensory data streams. To support task development, the SDG-Pro framework provides *Data and Control points* (cf. Chapter 4). The *cloud services* usually define virtual service topologies by referencing the aforementioned tasks. At the application level, we provide explicit representation of these tasks via *Intents*, i.e., developers write *Intents* to dynamically configure and invoke the tasks (cf. Chapter 3). Further, developers use *IntentScope*s to delimit the range of an *Intent*. For example, a developer might want to code the expression: “stop all vehicles on golf course X". In this case, “stop" is the desired *Intent*, which needs to be applied on an *IntentScope* that encompasses all vehicles with the location property “golf course X".

Since these concepts are described in detail in Chapter 3 and Chapter 4, as a reminder, in this section we only show example usage of the main programming abstractions provided by SDG-Pro to support development of business logic services in a uniform manner. Listings 5.1 and 5.2 show example usage of Intents and IntentScopes. Listing 5.3, gives a general example of how to define a data point. It shows a data point with one stream of simple data instances that represent, e.g., vehicle’s tire speed, based on the required sensor properties.

```
Intent eFault = Intent.newMIntent("EnergyFault");
//monitor whole fleet
eFault.setScope(IntentScope.getGlobal());
notify(eFault,this);//invoke task
//callback function called on event arrival
public void onEvent(Event e){//perform some action}
```

Listing 5.1: Example MonitorIntent.

```
IntentScope cs = delimit(IntentScope.getGlobal(),
  Cond.isTrue(eFault)); //eFault defined in Listing1
Intent eCons = Intent.newCIntent("ReduceEnergy");
eCons.setScope(cs);//set intent scope
eCons.set("speed").value("5");
eCons.set("RPM").value("1100");
send(eCons); //invoke task
```

Listing 5.2: Example ControlIntent.

```
DataPoint dataPoint = new DataPoint();
// Query available buffers
Collection<BufferDescription> availableBuffers
    = dataPoint.queryBuffers(new SensorProps(...));
// Assign the buffers to the data point
dataPoint.assign(availableBuffers.get(0));
dataPoint.setPollRate(5);
```

Listing 5.3: Example usage of data points.

5.3.3 Programmatic application provisioning

The most important abstraction for provisioning IoT Cloud applications is the software-defined gateway. In our programming model software-defined gateways are treated as

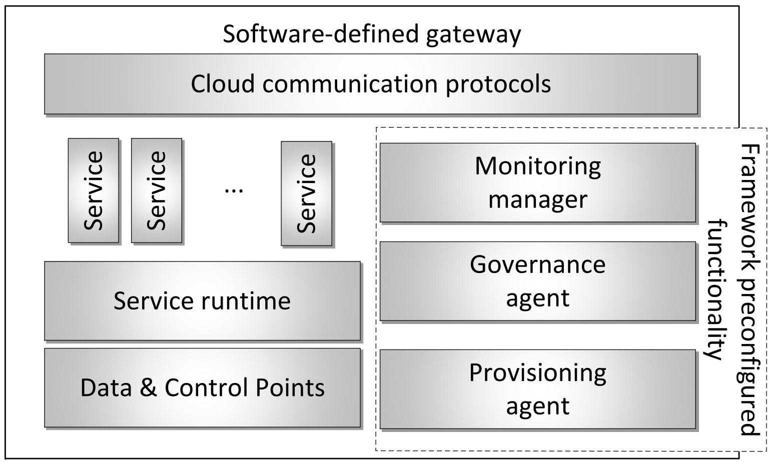


Figure 5.5: Overview of software-defined gateway structure.

first-class citizens. This allows the developers to specify, manipulate and manage the IoT Cloud infrastructure resources and application artifacts programmatically from within the application logic.

Generally, provisioning part of the application logic is used to programmatically specify the infrastructure and artifact dependencies, i.e., the state of the infrastructure required by the business logic services to execute correctly. To this end, our framework supports the developers to perform two main tasks. First, the developers can programmatically define the software-defined gateways and specify their internal structure, i.e., they customize it with application-specific artifacts and services. Second, our framework supports the developers to deploy such gateways atop IoT Cloud (e.g., data centers or physical IoT gateways) form within the application logic. Therefore, provisioning logic is specified in software enabling the infrastructure dependencies and requirements to be defined dynamically, in code.

Figure 5.5 shows the typical structure of a software-defined gateway. We notice two important properties of software-defined gateways. First, to technically realize software-defined gateways SDG-Pro offers gateway prototypes. These are resource containers, used to bootstrap more complex, higher-level gateway functionality. Generally, they are hosted atop IoT Cloud and enriched with functional, provisioning and governance capabilities, which are exposed via well-defined APIs. Currently, our framework supports software-defined gateways based on kernel supported virtualization, but virtualization choices do not pose any limitations, because, by utilizing the well-defined API, our gateway prototypes can be dynamically configured, provisioned, interconnected, deployed, and controlled. Second, developers use the software-defined gateways to programmatically provision and deploy required application services, but also to configure an execution environment for such services. Therefore, by utilizing the provisioning APIs, developers can customize the software-defined gateways to exactly meet the application's functional requirements. For example (Figure 5.5), they can dynamically configure a specific cloud communication protocol, e.g., CoAP or MQTT, select services runtime, e.g., Sedona VM or configure data and control points, e.g., based on Modbus.

In order to provision a software-defined gateway, initially the developers need to specify the software defined gateway prototypes. Listing 5.4 illustrates how the gateway prototypes are programmatically defined with our framework. In this example, a software-defined gateway is created from a gateway prototype, based on BusyBox. In the background the framework creates a Linux container and installs the provisioning and governance agents on it (more details about this process are given in Section 5.4). In general, the agents expose the provisioning APIs, which are activated and available at that point. Afterwards, a developer provides a configuration model for the gateway. In this example the gateway is configured to be deployed on a specific host by setting the “host address". In case it is not set the framework uses the deployment class to determine the gateway placement. Finally, the developer specifies the class that contains the internal provisioning logic.

```
//Define and configure a gateway
SDGateway gateway
      = UnitsController.create(GType.BUSYBOX);
gateway.setId("gateway-X");
gateway.setHost("http://host_address");
gateway.setMetaData(Deployment.class);
gateway.addConfigClass(Provisioning.class);
```

Listing 5.4: Example of software-defined gateway prototype.

Listing 5.5 illustrates our framework's support for dynamic provisioning of such gateways. The gateway provisioning logic contains the directives to internally provision the gateway, e.g., to install and configure device services, cloud communication libraries and data and control points. To this end, developers can use the framework's provisioning support, which comprises infrastructure provisioning APIs (provided by the provisioning agents) and a provisioning DSL containing a number of functions that facilitate provisioning of the software-defined gateways. In this example, we show how to provision a gateway with Sedona runtime.

```
String dest = ".../G2021/svm";
provisioner.CreateDirIfMissing(dest);
provisioner.CopyToDir("sedona-vm-1.2.28/svm", dest);
provisioner.setPermissions(dest, "a+x");
```

Listing 5.5: Example of provisioning DSL and APIs usage.

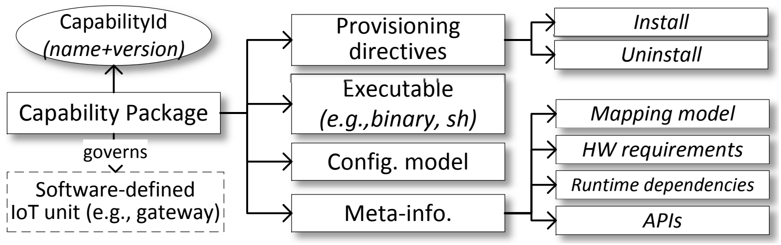


Figure 5.6: Governance capability package structure.

5.3.4 Programmatic application governance

After an application is provisioned and deployed, a new set of runtime concerns emerges, e.g., dynamically reconfiguring sensor update rates or elastically scaling software-defined gateways. In order to address such concerns, application developers implement operational governance processes (cf. Figure 5.4).

In Chapter 8 we introduce a general approach for runtime operational governance in software-defined IoT Cloud systems. In this chapter we mainly deal with operational governance processes that manipulate the states of IoT Cloud applications, at runtime. Such processes can be seen as a sequence of operations, which perform runtime state transitions from a current state to some desired target state (e.g., that satisfies some non-functional properties, enforces compliance, or exactly meets custom requirements). The core abstraction behind the operational governance business logic are *governance capabilities*. Generally, the governance capabilities represent the main building blocks of operational governance processes and they are usually executed in IoT devices. The governance capabilities encapsulate governance operations which can be applied on deployed software-defined gateways, e.g., to query the current version of a service, change a communication protocol, or spin up a virtual gateway. Such capabilities are described via well-defined APIs and are usually provided by domain experts who develop the IoT units. The framework enables such capabilities to be dynamically added to the system and supports managing their APIs. Generally, we do not make any assumptions about concrete capability implementations. However, the framework requires them to be packaged as shown in Figure 5.6.

To enable programmatic operational governance (implementing application governance processes) our framework offers a governance DSL and managed governance APIs. They are used by application developers to develop application governance logic, i.e., operational governance processes that install, deploy, manage and invoke the governance capabilities. One way to define the operational governance processes is to provide a sequence of governance API calls. Listing 5.6, shows examples of operational governance APIs exposed by our framework. In addition, SDG-pro provides a higher-level support to develop application governance processes in terms of a governance DSL. Listing 5.7 and Listing 5.8 respectively show how to use two main constructs of the application governance DSL, namely how to define a GovernanceScope and how to define a simple

GovernancePolicy. The governance policies are discussed in more detail in Chapter 9.

```
/* General case of capability invocation.       */
/deviceId/{capabilityId}/{methodName}/{arguments}?
      arg1={first-argument}&arg2={second-argument}&...

/* Data points capability invocation example.   */
deviceId/DPcapa/setPollRate/arguments&rate=5s/deviceId/DPcapa/list

/* Capabilities manager examples.               */
/deviceId/cManager/capabilities/list
/deviceId/cManager/{capabilityId}/stop
```

Listing 5.6: Examples of operational governance APIs.

```
G:GOVERNANCE_SCOPE
   query:= location=buildingX & type=JACE-545
   CONSIDERING_UNCERTAINTY:
    missing_data = "location<='?',type<='*'" AND
    selection_strategy = optimistic AND
    use_cache = false
```

Listing 5.7: Example governance scope.

```
S:STRATEGY CASE Fulfilled(CND1):
   setUpdateRate(5s) FOR G //see listing above
   CONSIDERING_UNCERTAINTY:
     run_in_isolation = true AND
     keep_alive = 5min AND
     degree_parallelism = 200 AND
     tolerate_fault_percentage  = 20% AND
     fallback_count = 2 AND
     time_to_next_fallback = 500ms
```

Listing 5.8: Example governance policy.

5.4 SDG-Pro's Runtime Mechanisms

The SDG-Pro framework provides a set of runtime mechanisms that underpin the programming abstractions (Section 5.3) and support application execution atop the IoT Cloud. Generally, application execution includes instantiating, provisioning and deploying software-defined gateways; dynamically loading device services atop the gateways; instantiating virtual application topologies (with Intents and IntentScopes); and triggering execution of operational governance processes (on-demand, depending on the business logic). Next, we discuss the design and implementation of the most important SDG-Pro's runtime mechanisms in more detail.

5.4.1 Instantiating, provisioning and deploying software-defined gateways

Currently SDG-Pro supports a version of software-defined gateways (prototypes), which is based on Linux Containers (LXC). When a developer instantiates a gateway prototype (e.g., as shown in Listing 5.4), the *IoT units controller* (cf. Figure 5.3) performs three main tasks. First, it creates an instance of LXC and installs the *provisioning and governance agents* in the container. Second, the provisioning agent[2] executes the provisioning directives, supplied in a provisioning script (e.g., Listing 5.5). Finally, the *IoT units controller* deploys the gateway instance in IoT Cloud.

Firstly, to instantiate a software-defined gateway our framework relies on Docker[3], i.e., more specifically on Docker deamon that offers a remote API for programmatic container creation. To bootstrap the instantiation, SDG-Pro provides a custom base image, which we developed atop a BusyBox user land on a stripped-down Linux distribution. In SDG-Pro, the *DeviceManager* is based on the Docker remote API, but it provides additional support for configuring and managing containers such as specifying the custom meta information (e.g., location) to provide more control over the software-defined gateways at the application level. As the last part of gateway instantiation, SDG-Pro installs its provisioning and governance agents that support execution of the subsequent phases.

Provisioning a software-defined gateway includes configuring, deploying and installing different artifacts such as device services, libraries (e.g., cloud communication protocols) and other binaries atop the newly created gateway instance. In the first step of the provisioning process, the *ProvisioningManager* creates artifacts image. In essence, it is a (compressed) set of component binaries and provisioning scripts. Next, the *DeploymentManager* places the image in the update queue. The provisioning agent periodically inspects the queue for new updates and when it is available the agent polls the image in the gateway (container) in a lazy manner. Additionally, SDG-Pro allows the components to be asynchronously pushed to the gateways, similarly to eager object initialization. Finally, the agent interprets provisioning scripts, i.e., performs a local installation of the binaries and executes any custom configuration directives.

Lastly, the SDG-Pro framework selects an IoT Cloud node, i.e., an edge device or a cloud VM, and deploys the gateway instance on it. The main component responsible for gateways (containers) allocation and deployment is the *GatewayCoordinator*. In the current prototype, the GatewayCoordinator is built based on fleet and etcd. The fleet is a distributed init system that provides the functionality to manage a cluster of host devices, e.g., the IoT Cloud nodes. The etcd is a distributed key/value store that supports managing shared configurations among such nodes. In order to allocate a gateway, i.e., select the best matching node in the IoT Cloud, the GatewayCoordinator compares the available gateway attributes (e.g., location, ownership, node type, etc.) with the

[2]The provisioning agent is implemented as a light-weight service, based on Oracle Compact Profile1 JVM

[3]https://docker.com/

meta data of the available IoT Cloud nodes. The gateway's meta data is obtained from a developer-specified configuration model. The nodes' meta data is provided by the *DeviceManager* and it is mostly maintained manually, e.g., by system administrators. At the moment, we only provide a rudimentary support for gateway allocation, i.e., SDG-Pro only considers static node properties. In the future, we plan to address this issue by including support for dynamic properties such as available bandwidth and providing support for runtime migration (reallocation) of software-defined gateways. Finally, after a node is selected, the GatewayCoordinator invokes the fleet to deploy the gateway on that node.

5.4.2 Intent-based invocation and IntentScope resolution

In the SDG-Pro framework, the communication among the main application components is performed via Intents. Generally, it follows a partial content-based publish/subscribe model and in the current prototype it is based on the Apache ActiveMQ JMS broker.

When an application submits a new Intent to SDG-Pro's *RuntimeContainer*, it first routes the Intent to the *TaskSelector*, which matches intent headers with device services (task) filters to find suitable services that match the Intent. Afterwards, the TaskSelector reads the Intent attributes and compares them with the task filters to find the best matching task. The attributes are represented as feature vectors and a multi-dimensional utility function, based on the Hamming distance, is used to perform the matching. Afterwards, the TaskSelector requests a service instance, by providing its description to the the TaskManager. It checks the validity of the mapping and, if it is valid, invokes the corresponding service. If no service is available the Intent is marked as failed and the invoker is notified.

In a more general case, when an Intent gets invoked on an IntentScope, the aforementioned invocation process remains the same, with the only difference that our framework performs all steps on a complete IntentScope, in parallel, instead on an individual gateway. To this end, the *ScopeCoordinator* provides dynamic resolution of the IntentScopes. The IntentSope specifications are implemented as composite predicates which reference device meta information and profile attributes. The predicates are applied on the GlobalScope (Section 5.3.2), filtering out all resources that do not match the provided attribute conditions. The ScopeCoordinator uses the resulting set of resources to initiate the Intent mapping and invocation. The ScopeCoordinator is also responsible to provide support for gathering results delivered by the invoked device services. This is needed since the scopes are resolved in parallel and the results are asynchronously delivered by the software-defined gateways.

5.4.3 Invocation of runtime governance capabilities

As shown in Section 5.3.4, application developers define operational governance logic as a sequence of API calls to the governance capabilities. The *APIManager* is responsible to mediate (map) these invocations to the underlying infrastructure, i.e., the software-defined gateways. To this end it relies on the *CapabilityManager*, which is a cloud-based service and the governance agent, which is a light-weight HTTP deamon, preinstalled in software-defined gateway prototypes.

When an API request is submitted by an application, SDG-Pro performs following steps: it registers the capability, maps the API call, executes the capability, and returns the result. First, the APIManager registers the API call with the corresponding capability. This involves querying the capability repository to obtain its meta-information (such as expected arguments), as well as building a dynamic mapping model, which includes capability ID, a reference to a runtime environment (e.g., Linux shell), input parameters, the result type, and further configuration directives. The CapabilityManager forwards the model to the gateways (i.e. the governance agent) and caches this information for subsequent invocations. During future interactions, the framework acts as transparent proxy, since subsequent steps are handled by the underlying gateways. In the next step, the governance agent needs to perform a mapping between the API call and the underlying capability. By default, it assumes that capabilities follow the traditional Unix interaction model, i.e., that all arguments and configurations (e.g., flags) are provided via the standard input stream (stdin) and output is produced to standard output (stdout) or standard error (stderr) streams. This means, if not specified otherwise in the mapping model, the framework will try to invoke the capability by its ID and will forward the provided arguments to its stdin. For capabilities that require custom invocation, e.g., property files, policies, or specific environment settings, the framework requires a custom mapping model. This model is used in the subsequent steps to correctly perform the API call. Finally, the governance agent invokes the governance capability and as soon as the capability completes it collects and wraps the result. Currently, the framework provides means to wrap results as JSON objects for standard data types and it relies on the mapping model to determine the appropriate return type. However, this can be extended to support generic behavior, e.g., with Google Protocol Buffers.

5.5 Evaluation

5.5.1 Evaluation methodology

In this section we present a functional evaluation of the chapter's main contribution – the SDG-Pro's programming model for IoT Cloud applications. To validate SDG-Pro's programming model we follow evaluation design guidelines provided in [100]. The main objective of our qualitative analysis is twofold. First, to show that SDG-pro facilitates dealing with the challenges of designing and developing IoT Cloud applications (RC1-RC4), we demonstrate how our programming model enforces the main design principles of IoT Cloud systems, as justified in Section 5.2.1. Second, in order to show that SDG-Pro enables easier, efficient and more intuitive development of IoT Cloud applications, we compare it against traditional programming model evaluation criteria that include: *readability, code simplicity, reusability, expressiveness* and *functional extensibility.*

5.5.2 Examples of FMS applications and services

To demonstrate the most important concepts and features of SDG-Pro's programming model, we present a set of real-life applications from our FMS system (Section 5.1). This example suite is designed to cover typical interactions and requirements of IoT Cloud applications, such as realtime monitoring and data analytics, remote actuation and control, autonomous device tasks and offline data analytics, in order to show the *completeness* of SDG-Pro's programming model regarding its support w.r.t. the real-life requirements. The example applications are developed and deployed atop a virtualized IoT Cloud testbed, based on CoreOS. In our testbed we simulate and mimic physical gateways in the cloud. The gateways are based on a snapshot of a real-world gateway, developed by our industry partners. The testbed is deployed on our local OpenStack cloud and it consists of 7 CoreOS 444.4.0 VMs, each running 150 LXCs, thus simulating approximately 1000 vehicles.

Example 1 – Energy consumption tracking

The FMS needs to monitor high-value vehicles' energy consumption in (near) real-time. In case any energy fault is detected, it must notify a golf course manager and put the vehicles in a reduced energy mode.

```
1   //select high-value vehicles
2   IntentScope s =
3    cont.delimit(IntentScope.getGlobal(),
4     Cond.greaterThan("price", "5000"));
5   Intent eFault = Intent.newMIntent("EnergyFault");
6   eFault.setScope(s);
7   cont.notify(eFault, this);//sub. to event
8  ...
9  public void onEvent(Event e){
10  IntentScope ts = IntentScope.create(e.getEntityId();
11  Intent eCons = Intent.newCIntent("ReduceEnergy");
12  eCons.setScope(ts);//set task scope
13  cont.send(eCons); //send to all vehicles in ts
14 }
```

Listing 5.9: Remote monitoring of fleet's energy consumption.

The most important part of this application in shown in Listing 5.9. To implement the monitoring behavior, developers only need to define an IntentScope (lines 2-4), in which they declare properties (e.g., metadata) that need to be satisfied by monitored vehicles, define a MonitorIntent and assign the desired scope to it (lines 5-7). Similarly, to implement a remote control behavior developers only need to define a ControlIntent (lines 12-14). In this example, it is natural to use asynchronous communication (of sensory data), thus a developer uses SDG-Pro's `notify` directive (line 8), to subscribe for the state changes in the environment.

This example demonstrates how easy it is to implement a real-time remote monitoring behavior. By introducing IntentSopes at the application level, SDG-Pro shields the developers from directly referencing the vast number of diverse physical entities and enables them to delimit the range of their actions on a higher abstraction level. Similarly, to perform an IoT control action or to subscribe for relevant events, developers only need to define and configure the corresponding Intents. This allows them to communicate to the system what needs to be done, instead of worrying how the underlying devices will perform the specific task.

Example 2 – Scheduled maintenance check

The FMS performs daily checks of the fleet's health. This is done mainly during the night, when the vehicles reside dormant in the club house, within the Wi-Fi range. The application reads the diagnostic data, gathered during the day, and analyzes them offline.

```
1  Intent localCon = Intent.newMIntent("ConnType");
2  localCon.setScope(IntentScope.getGlobal());
3  IntentScope ds = container.
4   delimit(Cond.eq("WLAN", localCon));
5  ds.addObserver(this);

7 public void update(Observable obs, Object arg){
8  Intent di = Intent.newMIntent("DiagnosticsLogger");
9  di.setScope((IntentScope)obs);
10 List<Event> data = container.poll(diagnostics);
11 // send data to an analytics framework
12 }
```

Listing 5.10: Scheduled maintenance check.

To implement such behavior, application first needs to determine that a vehicle is connected to a local network. This is achieved by defining an active IntentScope, as shown in Listing 5.10, lines 1-4. Second, the application needs to gather vehicles' diagnostic data and store them, e.g., in a local database. To synchronously poll the vehicle data, a developer simply defines a MonotorIntent and uses the `poll` directive (lines 8-10).

This example, demonstrates several important points. First, since MonitorIntents can be used to define an IntentScope, SDG-Pro enables developers to dynamically (e.g., based on environment or context changes) determine application behavior. Second, since IntentScopes are observable, developers can specify complex conditions that will trigger an execution of the business logic, without having to write complicated queries and event processing schemes. Finally, it is worth noticing that SDG-Pro does not provide support for the data analytics. However, we have shown that with little effort, by using intuitive concepts, an offline analytics application can obtain the required data, which can then be analyzed with data analytics frameworks, e.g., MapReduce.

Example 3 – Diagnostics data logging

This application periodically pools the data from the variety of vehicle's sensors e.g., engine status, battery status, transmission, etc. and stores them locally for later analysis (e.g., see Example 2).

```
 1 //Create custom sensor from physical channel
 2 BufferConf bc = new BufferConf("voltage_in");
 3 bc.setClass(BufferClass.SENSOR);
 4 bc.getAdapterChain().add(
 5   new ScalingAdapter(0.0,100.0,10.0));
 6 bc.getAdapterChain().add(new LowpassFilter(0.30));
 7 BufferManager.create("lowpass-scaled", bc);
 8 //Define diagnostics model
 9 DataPoint diagnostics =
10   new ComplexDataPoint("lowpass-scaled","voltage_in");
11 DataInstance di = diagnostics.read();
12 //log the diagnostics data di
```

Listing 5.11: Logging diagnostics data locally.

Listing 5.11 shows a partial diagnostics data model. The diagnostic data contains raw engine voltage readings and scaled voltage readings with low-pass filter, e.g., possibly indicating that something is taking the power away from the motor. To develop a custom sensor, developers only need to create a virtual buffer (referencing the base channel, e.g., raw voltage readings) and configure its adapter chain, as shown in lines 2-6. After creating a custom virtual sensor (line 7) application can treat this sensor as any other sensor. Consequently, a data model can then be easily defined with Data Points, as sown in lines 9-11. Storing the data is omitted for readability purposes.

Essentially, this example shows how our framework transparently virtualizes access to the same voltage sensor. This demonstrates two important features of the data and control points. First, since the SDG-Pro provides (virtually) exclusive access to the sensors (i.e., buffers act as multiplexers), developers can define custom configurations for the data streams, effectively creating an application-specific view of the sensors. An important consequence is that multiple applications can easily share the infrastructure, retaining a custom view of it. Second, since Data and Control points support developers to interact with underlying devices in a unified manner, i.e., independent of the communication protocols or the input channel types, applications can define their (arbitrarily complex) data models by only specifying the required data points. These can be seen as volatile fields in traditional data model entities.

Example 4 – Energy fault detection

To detect vehicles over consuming battery an FMS service relies on powermeter, odometer and temperature sensors that are available in the vehicles and uses a custom algorithm to detect potential energy faults.

In Listing 5.12 we show a code snippet from the corresponding FMS service. Developers create two data points. The `dp1` combines the battery status and odometer readings and it asynchronously delivers the sensory readings to the service. The `dp2` queries the available temperature data channels, based on their meta data and aggregates

```
1 DataPoint dp1 = DataPoint.
2     create( "battery", "odometer" );
3 DataPoint dp2 = new DataPoint();
4 //Since we have multiple temperature sensors
5 //we query them via the meta data
6 Collection<BufferDescription> tempBuffers =
7    dataPoint.queryBuffers(
8       new SensorProps("*temperature*"));
9 dp2.assign(tempBuffers);
10 ...
11  //invoke energy fault detection algorithm
```

Listing 5.12: Device service for energy fault detection.

the temperature readings from the available thermometers (lines 6-9). Among other things, the energy fault detection algorithm uses these data points and Complex Event Processing (CEP) techniques to determine potential energy faults, but its implementation is omitted in accordance with our nondisclosure agreement.

We notice that application obtains the temperature readings without directly referencing any physical sensor. Instead it generically queries the sensors' meta data. Further, since SDG-Pro takes care of synchronizing the sensors' readings, e.g., among the temperature sensors, developers can focus on custom data processing steps (algorithm). This is a crucial requirement to be able to develop portable applications, which do not directly depend on the physical infrastructure.

Example 5 – Provisioning and deploying application runtime environment

In order to execute an application/service (see Examples 1-4), developers need to provision a software-defined gateway and deploy it atop IoT Cloud.

```
1 /* Snippet from Provisioning.java*/
2 //install JVM Compact Profile 1
3 String dest = ".../G2021/jvm";
4 provisioner.CreateDirIfMissing(dest);
5 provisioner.CopyToDir("jvm-profile1-1.8.0/*",dest);
6 provisioner.setPermissions(dest, "a+x");
7 ...
8 /* Snippet from Gateways.java*/
9 SDGateway gateway
10     = UnitsController.create(GType.BUSYBOX);
11 gateway.addConfigClass(Provisioning.class);
12 UnitsController.startParallel(gateway,
13      IntentScope.getGlobal().asResource());
```

Listing 5.13: Creating a software-defined gateway.

Listing 5.13, shows how to programmatically add Java Compact Profile runtime to a gateway and how to deploy instances of that gateway atop the vehicles' on-board devices. In lines 3-6 we show how developers can use the provisioning API to specify which custom resources are required in the gateway prototype. Further, this example show the most important parts related to gateways deployment, i.e., gateway instantiation from Docker-based Busybox prototype (lines 9,10), associating the configuration model with the prototype (line 11) and multiple deployment (lines 12,13).

This example shows a part of general SDG-Pro's provisioning API. We notice that our framework provides a generic API which can be used to declaratively configure different types of resources. This essentially enables developers to programmatically deal with complex IoT Cloud infrastructure and its dependencies, i.e., the desired configuration baseline is specified locally and once for multiple application instances. SDG-Pro provides a unified view on defining and manipulating the infrastructure through software-defined gateways, but also offers a fine-grained access and control of the gateways configuration (e.g., container's base image).

Example 6 – Configuring application dependencies programmatically

The FMS applications have different dependencies and requirements e.g., regarding communication protocols. To guarantee correct application behavior, developers (or operations managers) need to correctly configure such infrastructure dependencies.

```
1  //install Modbus
2  provisioner.addDCPointResource("modbus/Modbus.sab");
3  provisioner.addDCPointResource("modbus/Modbus.sax");
4  provisioner.addDCPointResource("modbus/Kits.scode");
5  provisioner.addDCPointResource("modbus/Kits.xml");
6  //install MQTT client
7  RemoteLibrary mqttClient
8      = provisioner.getFromURL(
9         "http://..../mqtt-client-0.0.1.jar");
10 provisioner.installComProto(mqttClient.getBinary());
11 ...
```

Listing 5.14: Configuring application dependencies.

Listing 5.14 shows excerpt of typical FMS protocols configuration. Lines 2-6 show how developers can to configure Modbus device protocol (used by Data and Control Points) and MQTT cloud connectivity protocol (lines 7-10), e.g., used by MonitorIntents.

The most important thing to notice here is that SDG-Pro provides software-defined gateway specific provisioning APIs. This shows that our abstractions are designed in such manner to inherently support programmatic provisioning, by exposing well-defined API and providing runtime mechanisms which transparently enable inversion of control and late (re)binding of the dependencies. Also standard provisioning operations such as fetching a remote resource can be combined with specific provisioning APIs, as shown in lines 7-9. The most important consequence is that developers can design generic application business logic and transparently declare the desired infrastructure dependencies programmatically, e.g., in a separate application module.

Example 7 – Emergency governance process

In case of an emergency situation the FMS needs to increase the monitoring frequency of vehicles' sensors.

```
1 Iterator<Vehicle> vehicles.iterator();
2 //for each vehicle on the golf course
3 List<DataPoint> dPoints = HTTPClient
4     .invoke(".../APIManager/mapper/"
5        +vehicles.next().getId()+"/DPcapa/list");
6 for (DataPoint dp : dPoints) {
7   HTTPClient.invoke(".../DPcapa/"
8    +"setPollRate/args?rate=10s&id="+dp.getId()");
9 }
```

Listing 5.15: Example emergency operational governance process.

To satisfy this cross-cutting compliance requirement, developers need to develop an operational governance process [104, 110]. Listing 5.15 shows a code snippet form such emergency governance process. The most important part of the process is shown in lines 7-8, which show how a developer can use governance API to dynamically manipulate the edge of the infrastructure, in this case change the sensor update rate.

SDG-Pro takes over the responsibility of invoking individual governance capabilities (e.g., per vehicle), effectively shielding the developers from low-level infrastructure details. The most important consequence of having such governance API is that the governance logic can be specified programmatically and maintained locally. Also governance processes are completely separated from the business logic, thus the core business logic is not polluted with cross-cutting governance concerns. In addition, since at the application level the infrastructure is perceived as a set of capabilities exposed through the governance API, the developers do not have to worry about geographical distribution, heterogeneity or scale of the IoT Cloud infrastructure nor directly deal with individual devices.

5.5.3 Discussion

As shown on a set of real-life examples, our SDG-Pro framework enables addressing most of development concerns at application code level (*everything as code*). This provides advantages such as having a uniform view on the entire development process, which makes it easily traceable and auditable, but also enables exploiting proven and well-known technologies, e.g., source control or configuration management systems, during the entire application lifecycle. Moreover, it gives full control to developers and makes IoT Cloud applications less infrastructure-dependent.

We have shown how SDG-Pro provides *API encapsulation* of the most important aspects related to gateway provisioning and governance. A key advantage of this approach is that developers do not need to explicitly worry about the underlying infrastructure. Rather, they perceive the complex and heterogeneous IoT Cloud infrastructure as a set of uniform APIs that enable programmatic management of such infrastructure. Our SDG-Pro framework supports the developers to *declaratively provision* IoT Cloud systems and to *automate* most of the provisioning process. This improves general readability and maintainability of the provisioning logic and simplifies the provisioning process. Additionally, by encoding the provisioning directives as part of application's source code, our framework makes the provisioning process easily repeatable. This reduces the potential errors, but more importantly enables continuous, automated enforcement of the configuration base line. Regarding the governance processes, by providing a logically *centralized point of operation* of IoT Cloud infrastructure, SDG-Pro supports developers to easily define desired states and runtime behavior of IoT Cloud systems, but also enables automated enforcement of governance processes, which is crucial to realize (time) consistent governance strategies across the entire IoT Cloud system.

We also notice a number of limitations of our approach. From the technical perspective, at the moment SDG-Pro offers a rudimentary mechanism for gateway allocation, which only considers static properties when deploying the software-defined gateways. Additionally, although IoT Cloud systems include many mobile and unstable devices, the current prototype provides a limited support regarding the dependability concerns. However, optimization of gateway allocation and addressing the dependability issues related to device mobility are subject of our future work. Furthermore, the set of proposed programming concepts is not exhaustive and especially the provisioning and governance APIs are in an active state of development and refinement. However, as we have shown on a set of real-life examples, SDG-Pro offers programming support sufficient to express many common behaviors of IoT Cloud applications. Although our programming model has many important traits such as readability and simplicity, as well as facilitates writing reusable and portable application logic, in SDG-Pro's programming model, we trade flexibility and expressiveness for more intuitive and efficient programming of the IoT Cloud applications. Finally, although developers utilize the well-known Java programming language, SDG-Pro introduces a number of new concepts that require an initial learning effort. However, by explicitly enforcing main design principles of software-defined IoT Cloud systems, we believe that in the long-run our framework can reduce development time, potential errors and eventually the costs of application development.

5.6 Conclusion

In this chapter, we introduced the SDG-Pro framework for developing software-defined IoT Cloud applications in a uniform manner. We presented SDG-Pro's programming model for IoT Cloud applications, which is designed to enforce the main principles of software-defined IoT Cloud systems. We discussed that by enforcing such principles on the application level, our framework enables easier, efficient and more intuitive application development. Besides supporting business logic development, SDG-Pro introduces additional support and provides a unified programmatic view on the entire development process (*everything as code*). We illustrated how our framework encapsulates most important aspects of IoT Cloud provisioning and governance, exposing them to the developers in terms of uniform APIs and light-weight provisioning and governance DSLs. We discussed on a set of real-life applications, that by providing a systematic and structured support for everything-as-code paradigm our framework makes the entire application development process more traceable and easily auditable, but also enables exploiting proven and well-known technologies, e.g., source control or configuration management systems, during the entire application lifecycle.

Finally, the comprehensive provisioning and governance models are subject of other contributions of this thesis, which are respectively discussed in more detail in Part II and Part III of the thesis.

Part II

Provisioning IoT Cloud Systems

Preface

Over the recent years, cloud computing and the Internet of Things have been converging ever stronger, sparking creation of very large-scale, geographically distributed systems. When facing such large-scale systems with heterogeneous, dynamic and geographically distributed resource pool, efficacy of provisioning models, mechanisms and tools plays a crucial role. With the rise of cloud computing, we have witnessed numerous benefits of self-service, utility-oriented provisioning models, in terms of more flexible and cheaper IT operations. Since cloud is a one of the key constituents of IoT Cloud, it would be natural to expect that these flagship properties of cloud computing would be inherited by IoT Cloud as well. Unfortunately, this is still not the case and as a consequence system integrators and operations managers have to rely on provisional solutions, which require combining multitude of provisioning techniques such as manual, script- and service-based provisioning. This requires rethinking existing support for representing infrastructure resources, managing their configuration and deployment models as well as composing low-level resource components into usable infrastructures, capable to support novel application business logic requirements.

The main objective of the second part of this thesis is to respond to one of the main research questions formulated in Chapter 1, namely: "Which provisioning models, techniques and tools can be applied to enable on-demand, self-service provisioning of IoT Cloud resources at fine granularity?". The contributions presented in this part of thesis are mainly driven by a stringent need: To enable refactoring of the underlying infrastructure into finer-grained resource components whose behavior can be defined in software; To provide conceptually unified representation of both Edge and Cloud resources; As well as to enable automated and scalable management of IoT Cloud infrastructures and their configuration models in a logically centralized fashion. To this end, in Chapter 6 we introduce a conceptual model and layout a road map towards utility-based provisioning of IoT Cloud systems. The main building blocks of our provisioning model are software-defined IoT units. Our model conceptualizes the software-defined IoT units and elicit their main design principles together with a road map to develop corresponding technical enablers. Chapter 6 also introduces a preliminary prototype implementation of a provisioning framework, which provides support (enablers) for automating the main aspect of provisioning processes (units composition) and enables centrally managed configuration models. Chapter 7 continues our line of research towards utility-based provisioning, by introducing a middleware infrastructure, which provides a comprehensive support for multi-level provisioning of IoT Cloud systems. The main features of our middleware include: i) Support for automated provisioning and management of infrastructure resources, application components and configuration models in a uniform, logically centralized manner through middleware-managed APIs; ii) Extensible and flexible provisioning models, which support self-service, on-demand consumption of Edge-device resources; iii) A generic, light-weight resource abstraction mechanism, which allows for application-specific customization of IoT Cloud resources with well-defined APIs.

CHAPTER 6

Provisioning Software-defined IoT Cloud Systems

Cloud computing technologies have been intensively exploited in development and management of the large-scale IoT systems, e.g., in [134, 64, 160], because theoretically, cloud offers unlimited storage, compute and network capabilities to integrate diverse types of IoT devices and provide an elastic runtime infrastructure for IoT systems. Self-service, utility-oriented model of cloud computing can potentially offer fine-grained IoT resources in a pay-as-you-go manner, reducing upfront costs and possibly creating cross-domain application opportunities and enabling new business and usage models of the IoT cloud systems. However, most of the contemporary approaches dealing with IoT cloud systems largely focus on data and device integration by utilizing cloud computing techniques to virtualize physical sensors and actuators. Although, there are approaches providing support for provisioning and management of the virtual IoT infrastructure (e.g, [160, 134, 42]), the convergence of IoT and cloud computing is still at an early stage. System designers and operations managers face numerous challenges to realize large-scale IoT cloud systems in practice, mainly because these systems impose diverse requirements in terms of granularity and flexibility of IoT resources consumption, custom provisioning of IoT capabilities such as communication protocols, elasticity concerns, and runtime governance. For example, modern large-scale IoT cloud systems heavily rely on the cloud and virtualized IoT resources and capabilities (e.g., to support complex, computationally expensive analytics), thus these resources need to be accessed, configured and operated in a unified manner, with a central point of management. Further, the IoT systems are envisioned to run continuously, but they can be elastically scaled in/down in off-peek times, e.g., when a demand for certain data sources reduces. Due to the multiplicity of the involved stakeholders with diverse requirements and business models, the modern IoT cloud systems increasingly need to support different and customizable usage experiences. Therefore, to utilize the benefits of cloud computing, IoT cloud systems need to support virtualization of IoT resources and IoT capabilities (e.g., gateways, sensors, data streams and communication protocols), but also enable: i) encapsulating them in a well-defined API, at different levels of abstraction, ii) centrally managing configuration models and automatically propagating them to the edge of infrastructure, iii) automated provisioning of IoT resources and IoT capabilities.

In this chapter, we introduce the concept of *software-defined IoT units* – a novel approach to IoT cloud computing that encapsulates fine-grained IoT resources and IoT capabilities in a well-defined API in order to provide a unified view on accessing, configuring and operating IoT cloud systems. Our software-defined IoT units are the fundamental building blocks of software-defined IoT cloud systems. They enable consuming IoT resources at a fine granularity, allow for policy-based configuration of IoT capabilities and runtime operation of software-defined IoT cloud systems. We present a preliminary implementation of a framework for dynamic, on-demand provisioning of the software-defined IoT cloud systems. By automating main aspect of provisioning processes and supporting centrally managed configuration models, our framework simplifies provisioning of such systems and enables flexible runtime customizations.

The rest of this chapter is structured as follows: Section 6.1 presents a motivating scenario and research challenges; Section 6.2 describes main principles and our conceptual model of software-defined IoT systems; Section 6.3 outlines main provisioning techniques for software-defined IoT systems; Section 6.4 introduces design and implementation of our prototype, followed by its experimental evaluation; Finally, Section 6.5 concludes the chapter.

6.1 Motivation

In this chapter, we analyze the two main use cases (FMS and BMS) derived from the case study, presented in Chapter 2. This section approaches the two real-life systems form the perspective of operations management. It illustrates tasks that need to be performed to provision such systems and derives concrete research challenges, which the operations managers currently face to provision large-scale, geographically-distributed IoT Cloud systems.

6.1.1 Scenarios

Provisioning FMS

As mentioned in Chapter 2 the FMS is an IoT cloud system comprising vehicle's on-board gateways, network and the cloud infrastructure. The main features provided by the on-board device include: a) vehicle maintenance (fault history, battery health, crash history, and engine diagnostics), b) vehicle tracking (position, driving history, and geo-fencing), c) vehicle info (charging status, odometer, serial number, and service notification), d) set-up (club-specific information, maps, and fleet information). Vehicles communicate with the cloud via 3G, GPRS or Wi-Fi network to exchange telematic and diagnostic data. On the cloud we host different FM subsystems and services to manage the data. For example: a) *Realtime vehicle status*: location, driving direction, speed, vehicle fault alarms; b) *Remote diagnostics*: equipment status, battery health and timely maintenance reminders; c) *Remote control*: overriding on-board vehicle control system in case of emergency; d) *Fleet management*: service history and fleet usage patterns. In the following we highlight some of the FMS features, which need to be considered during system provisioning:

- The FMS subsystems and services are hosted in the cloud and heavily rely on the virtualized IoT resources, e.g., vehicle gateways and their capabilities. Therefore, we need to enable encapsulating and accessing IoT resources and IoT capabilities via uniform APIs.
- The FMS has different requirements regarding communication protocols. The fault alarms and events need to be pushed to the services (e.g, via MQ Telemetry Transport (MQTT) [114]), when needed vehicle's diagnostics should be synchronously accessed via RESTfull protocols such as CoAP [55]or sMAP [38]. The remote control system requires a dedicated, secure point-to-point connection. Configuring these capabilities should be decoupled from the underlying physical infrastructure, in order to allow dynamic, fine-grained customization.
- The FMS spans multiple, geographically distributed cloud instances and IoT devices that comprise FM's virtual runtime topologies. These topologies abstract a portion of the IoT cloud infrastructure, e.g., needed by specific subsystem, thus they should support flexible configuring to allow for on-demand provisioning.
- The FMS involves growing number of stakeholders. Therefore, we need to accommodate the scale and geographical distribution of the current FMS offering as well as support projected growth and future customization requirements.

6.1.2 Provisioning BMS

In general, to provision BMS operations managers perform two distinct tasks. The initial deployment and staging of devices on the one hand, and updates with varying frequency and priorities on the other hand. In our scenario the BMS provider is responsible for managing several hundreds of buildings with a variety of tenants. The managed buildings are equipped with variety of Edge devices ranging from sensors to detect smoke and heat, to elevator and door controls, to complex cooling and heating systems. They rely on gateways, which provide constrained execution environments with limited processing, storage, and memory resources to execute the device firmware and simple routines. Gateways enable the basic bundling and management of a wide variety of connected entities. Due to the current market situation and the existing lack of standards in this novel field, there exists a huge heterogeneity in terms of software environments when it comes to these gateways. Initially all these devices need to be staged with the necessary capabilities to enable their basic functionality. The connected sensors need to be supported, the latest firmware needs to be installed and they need to be integrated into a specific deployment structure. This is followed by long term evolution in terms of general maintenance, changing deployments, shifting capabilities as well as updating the software environment or firmware. The second kind of updates revolve around security patches and hot fixes that need to be deployed very fast in order to ensure that the whole infrastructure stays operational. These updates are time critical since delays can cause severe security problems in the whole infrastructure. Similarly to FMS we outline the following distinct requirements in the context of BMS:

- Gateways participating in an IoT infrastructure are resource-constrained in terms of their processing, memory, and storage capabilities.
- Our scenario deals with large-scale deployments comprising thousands of gateways with a wide variety of different supported execution environments.
- Requirements of these gateways change over time, which makes updates necessary. These updates can either be non-time-critical or time-critical, like security updates.
- In order to sustain operations all updates need to be efficient and fast, and, therefore, have to be performed during system runtime, without interrupting its operation, i.e., down time.

6.1.3 Research Challenges

The limited support for fine-grained provisioning at higher levels leads to tightly coupled, problem specific IoT infrastructure components, which require difficult and tedious provisioning and configuration management tasks on multiple levels. This inherently makes provisioning and runtime operation of IoT cloud systems a complex task. Consequentially, system designers and operations managers face numerous challenges to provision and operate large-scale IoT cloud systems such as the FMS or BMS.

RC1 – The IoT cloud services and subsystems provide different functionality or analytics, but they mostly rely on common physical IoT infrastructure. However, to date the IoT infrastructure resources have been mostly provided as coarse grained, rigid packages, in the sense that the IoT systems, e.g., the infrastructure components and software libraries are specifically tailored for the problem at hand and do not allow for flexible customization and provisioning of the individual resource components or the runtime topologies.

RC2 – *Elasticity*, although one of the fundamental traits of the traditional cloud computing, has not yet received enough attention in IoT cloud systems. Elasticity is a principle to provision the required resources dynamically and on demand, enabling applications to respond to varying load patterns by adjusting the amount of provisioned resources to exactly match their current needs, thus minimizing resources over- provisioning and allowing for better utilization of the available resources [47]. However, IoT cloud systems are usually not tailored to incorporate elasticity aspects. For example, new types of resources, e.g., data streams, delivered by IoT infrastructure are still not provided elastically in IoT cloud systems. Opportunistic exploitation of constrained resources, inherent to many IoT cloud systems further intensifies the need to provision the required resources on-demand or as they become available. These challenges prevent current IoT systems from fully utilizing the benefits cloud's elastic nature has to offer and call for new approaches to incorporate the elasticity capabilities in the IoT cloud systems.

RC3 – Dependability is a general measure of dynamic system properties, such as availability, reliability, fault resilience and maintainability. Cloud computing supports developing and operating dependable large-scale systems atop commodity infrastructure, by offering an abundance of virtualized resources, providing replicated storage, enabling distributed computation with different availability zones and diverse, redundant network links among the system components. However, the challenges to build and *operate dependable large-scale IoT cloud systems* are significantly aggravated because in such systems the cloud, network and embedded devices are converging, thus creating very large-scale hyper-distributed systems, which impose new concerns that are inherently elusive with traditional operations approaches.

RC4 – Due to dynamicity, heterogeneity, geographical distribution and the sheer scale of IoT cloud, traditional management and provisioning approaches are hardly feasible in practice. This is mostly because they implicitly make assumptions such as physical on-site presence, manually logging into devices, understanding device's specifics, etc., which are difficult, if not impossible, to achieve in IoT cloud systems. Thus, novel techniques, which will provide an unified and conceptually centralized view on system's configuration management are needed.

Therefore, we need novel models and techniques to provision and operate the IoT cloud systems, at runtime. Some of the obvious requirements to make this feasible in the very large-scale, geographically distributed setup are: (i) We need tools which will automate development, provisioning and operations (DevOps) processes; (ii) Supporting mechanisms need to be late-bound and dynamically configurable, e.g., via policies; (iii) Configuration models need to be centrally managed and automatically propagated to the edge of the infrastructure; (iv) Processes such as configuration models enforcement and deployment need to be flexibly repeatable with little effort as possible.

6.2 Main Building Blocks of Software-defined IoT Systems

6.2.1 Design Principles of Software-Defined IoT Cloud Systems

Generally, software-defined denotes a principle of abstracting the low-level components, e.g., hardware, and enabling their provisioning and management through a well-defined API [87]. This enables refactoring the underlying infrastructure into finer-grained resource components whose functionality can be defined in software after they have been deployed.

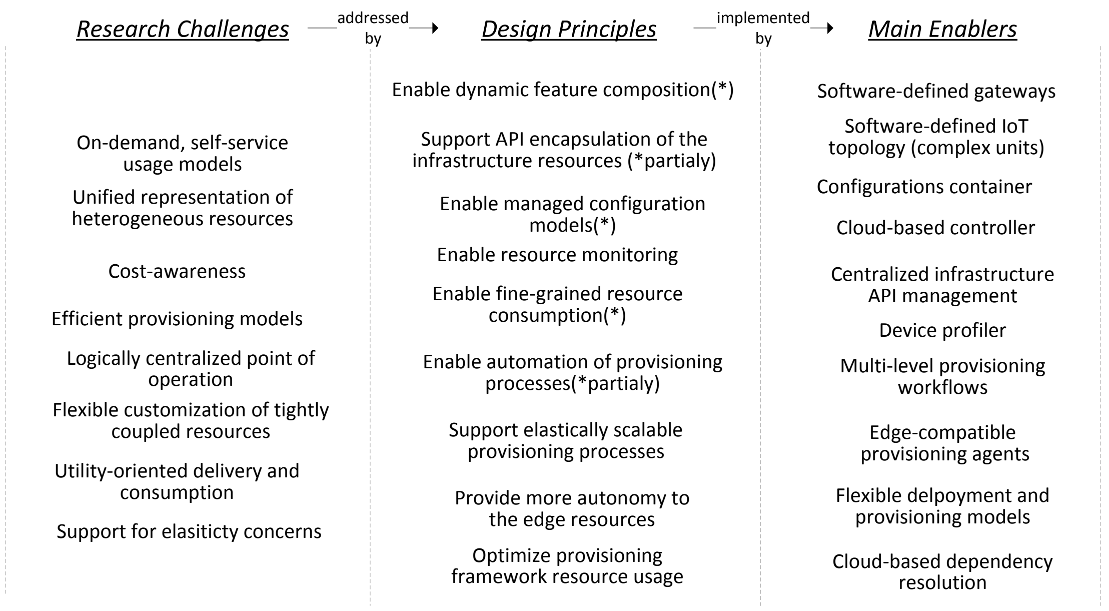
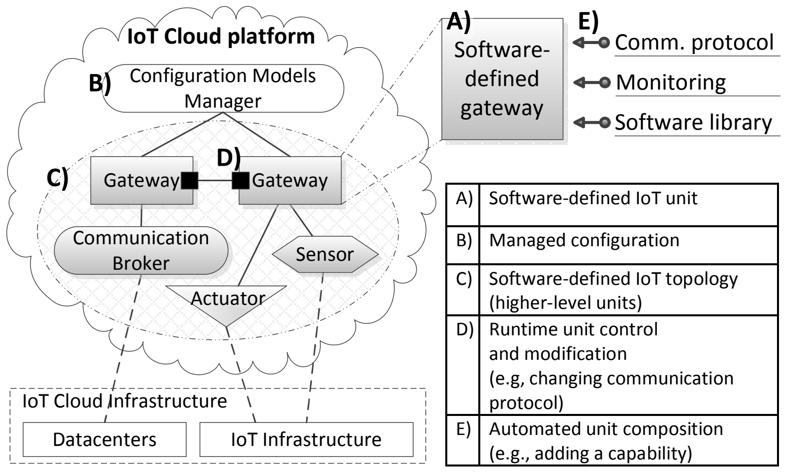

Research Challenges
addressed by
Design Principles
implemented by
Main Enablers
On-demand, self-service usage models
Unified representation of heterogeneous resources
Cost-awareness
Efficient provisioning models
Logically centralized point of operation
Flexible customization of tightly coupled resources
Utility-oriented delivery and consumption
Support for elasiticty concerns
Enable dynamic feature composition(*)
Support API encapsulation of the infrastructure resources (*partialy)
Enable managed configuration models(*)
Enable resource monitoring
Enable fine-grained resource consumption(*)
Enable automation of provisioning processes(*partialy)
Support elastically scalable provisioning processes
Provide more autonomy to the edge resources
Optimize provisioning framework resource usage
Software-defined gateways
Software-defined IoT topology (complex units)
Configurations container
Cloud-based controller
Centralized infrastructure API management
Device profiler
Multi-level provisioning workflows
Edge-compatible provisioning agents
Flexible delpoyment and provisioning models
Cloud-based dependency resolution

Figure 6.1: Summary of main principles and enablers of software-defined IoT Cloud systems.

Software-defined IoT Cloud systems comprise a set of resource components, hosted in IoT Cloud, which can be provisioned and controlled at runtime. The IoT resources (e.g., sensory data streams), their runtime environments (e.g., gateways) and capabilities (e.g., communication protocols, analytics and data point controllers) are described as *software-defined IoT units*. *Software-defined IoT units* are software-defined entities that are hosted in an IoT cloud platform and abstract accessing and operating underlying IoT resources and lower level functionality. Generally, *software-defined IoT units* are used to encapsulate the IoT Cloud resources and lower level functionality and abstract their provisioning and governance, at runtime. To this end, our *software-defined IoT units* expose well-defined API and they can be composed at different levels, creating virtual runtime topologies on which we can deploy and execute IoT cloud systems such as our FM system. The main design principles of software-defined IoT Cloud systems that we discuss in this chapter are marked with "*" in Figure 6.1 and are described in more detail subsequently. Other design principles, shown in the same figure, are discussed later in Chapter 7.

- API Encapsulation – IoT resources and IoT capabilities are encapsulated in well-defined APIs, to provide a unified view on accessing functionality and configurations of IoT cloud systems.
- Fine-grained consumption – The IoT resources and capabilities need to be accessible at different granularity levels to support agile utilization and self-service consumption.



A)	Software-defined IoT unit
B)	Managed configuration
C)	Software-defined IoT topology (higher-level units)
D)	Runtime unit control and modification (e.g, changing communication protocol)
E)	Automated unit composition (e.g., adding a capability)

Figure 6.2: Main enablers of software-defined IoT cloud systems

- Enable dynamic feature composition – The units are specified declaratively and their functionality is defined (composed) programmatically in software, using the well-defined API and available, familiar software libraries.
- Automated provisioning – Main provisioning processes need to be automated in order to enable dynamic, on-demand configuring and operating software-defined IoT systems, on a large-scale (e.g, hundreds gateways).
- Managed configuration models – The configuration models need to be managed automatically, as well as, dynamically propagated and (re)enforced in the edge resources, by a provisioning framework.

Figure 6.1 summarizes how we translate the aforementioned high-level design principles into concrete technical enablers. It serves as a general road map towards achieving our goal of enabling the utility-based provisioning paradigm in IoT Cloud systems. For example, to allow for flexible system customization, we need to enable fine-grained resource consumption, well-defined API encapsulation and provide support for policy-based specification and configuration. Among other things, these principles are enabled by our software-defined IoT units and support for centrally managed configuration models. Figure 6.2 gives high-level graphical overview of the main building blocks and enabling techniques, which are the prime focus of this chapter. Subsequently, we describe them in more detail. In Chapter 7, we will focus on enabling the remaining design principles shown in Figure 6.1.

6.2.2 Conceptual Model of Software-defined IoT Units

Figure 6.3 illustrates the conceptual model of our software-defined IoT units. The units encapsulate functional aspects (e.g., communication capabilities or sensor poll

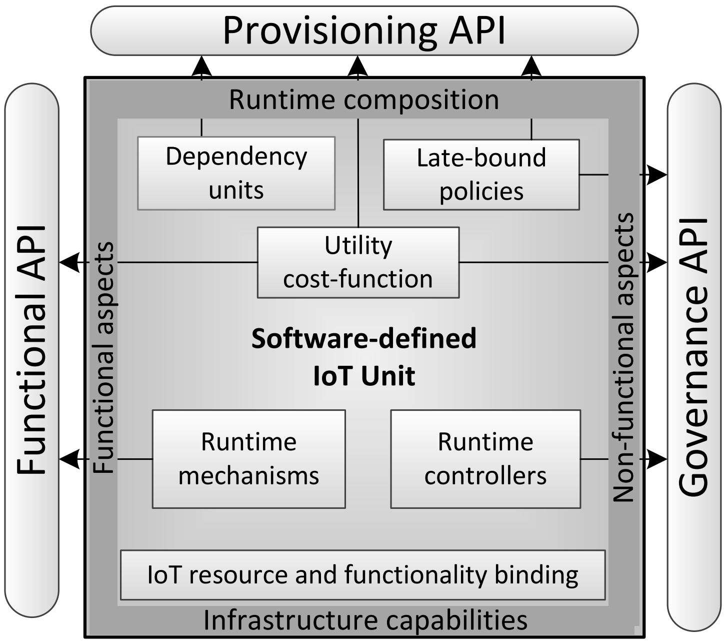


Figure 6.3: Conceptual model of software-defined IoT units.

frequencies) and non-functional aspects (e.g., quality attributes, elasticity capabilities, costs and ownership information) of the IoT resources and expose them in the IoT cloud. The functional, provisioning and governance capabilities of the units are exposed via *well-defined APIs*, which enable provisioning and controlling the units at runtime, e.g., start/stop. Our conceptual model also allows for composing and interconnecting software-defined IoT units, in order to dynamically deliver the IoT resources and capabilities to the applications. The runtime provisioning and configuration is performed by specifying late-bound policies and configuration models. Naturally, the software-defined IoT units support mechanisms to map the virtual resources with the underlying physical infrastructure.

To technically realize our unit model we introduce a concept of *unit prototypes*. They can be seen as resource containers, which are used to bootstrap more complex, higher-level units. Generally, they are hosted in the cloud and enriched with functional, provisioning and governance capabilities, which are exposed via software-defined APIs. The unit prototypes can be based on OS-level virtualization, e.g., VMs, or more finer-grained kernel supported virtualization, e.g., Linux containers. Conceptually, virtualization choices do not pose any limitations, because by utilizing the well-defined API, our unit prototypes can be dynamically configured, provisioned, interconnected, deployed, and controlled at runtime.

Given our conceptual model (Figure 6.3), by utilizing the *provisioning API*, the unit prototypes can be dynamically coupled with late-bound runtime mechanisms. These can be any software components (custom or stock), libraries or clients that can be configured and whose binding with the unit prototypes is differed to the runtime. For example, the mechanisms can be used to dynamically add communication capabilities, new functionality or storage to our software-defined IoT units. Therefore, by specifying policies, which are bound later during runtime, system designers or operations managers can flexibly manage unit configurations and customize their capabilities, at *fine granularity levels*. Our conceptual model also allows for composing the software-defined IoT units at higher levels. By selecting dependency units, e.g., based on their costs, analytics or elasticity capabilities, and linking them together, we can dynamically build more complex units. This enables flexible *policy-based specification and configuration* of complex relationships between the units. Therefore, by carefully choosing the granularity of our units and providing configuration policies we can *automate the units composition process* at different levels and in some cases completely defer it to the runtime. This makes the provisioning process flexible, traceable and repeatable across different cloud instances and IoT infrastructures, thus reducing time, errors and costs.

The runtime *governance API*, exposed by the units, enables us to perform runtime control operations such as starting or stopping the unit or change the topological structure of the dependency units, e.g., dynamically adding or removing dependencies at runtime. Therefore, one of the most important consequences of having software-defined IoT unit is that the functionality of the virtual IoT infrastructure can be (re)defined and customized after it has been deployed. New features can be added to the units and the topological structure of the dependency units can be customized at runtime. This enables automating provisioning and governance processes, e.g., by utilizing the governance API and providing monitoring at unit level, we can enable *elastic horizontal scaling* of our units. Therefore, most important features of software-defined IoT units which enable the general principles of software-defined IoT (see Section 6.2.1) are: i) They provide software-defined API, which can be used to access, configure and control the units, in a unified manner. ii) They support fine-grained internal configurations, e.g, adding functional capabilities like different communication protocols, at runtime. iii) They can be composed at higher-level, via dependency units, creating virtual topologies that can be (re)configured at runtime. iv) They enable decoupled and managed configuration (via late-bound policies) to provision the units dynamically and on-demand. v) They have utility cost-functions that enable pricing the IoT resources as utilities.

6.2.3 Units Classification

Depending on their purpose and capabilities, our software-defined IoT units have different granularity and internal topological structure. Therefore, conceptually we classify them into: (i) *atomic*, (ii) *composed* and (iii) *complex software-defined IoT units*. Depending on their type, the units require specific runtime mechanisms and expose specific provisioning API. Figure 6.4 depicts a simplified model of the software-defined IoT units structure and the most important dependencies among the described unit types.

The *atomic software-defined IoT units* are the finest-grained software-defined IoT units, which are used to abstract the core capabilities of an IoT resource. They provide software-defined API and need to be packaged portably to include components and

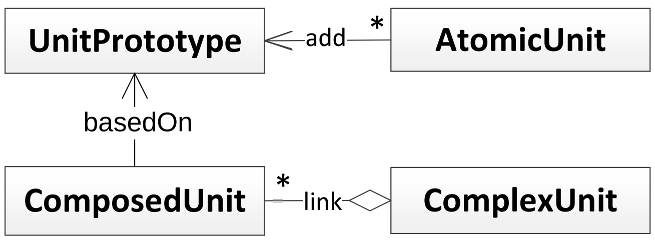


Figure 6.4: Simplified model of software-defined IoT units structure.

libraries, that are needed to provide desired capabilities. Figure 6.5 depicts some examples of the atomic software-defined units. We broadly classify them into functional and non-functional atomic software-defined IoT units, based on the capabilities they provide. Functional units encapsulate capabilities such as communication or IoT compute and storage. Non-functional units encapsulate configuration models and capabilities such as elasticity controllers or data-quality enforcement mechanisms. Therefore, the atomic units are used to identify fine-grained capabilities needed by an application. For example, the application might require the communication to be performed via a specific transport protocol, e.g., MQTT or it might need a specific monitoring component, e.g., Ganglia[1]. Classifications similar to the one presented in Figure 6.5 can be used to guide the atomic units selection process, in order to easily identify the exact capabilities, needed by the application.

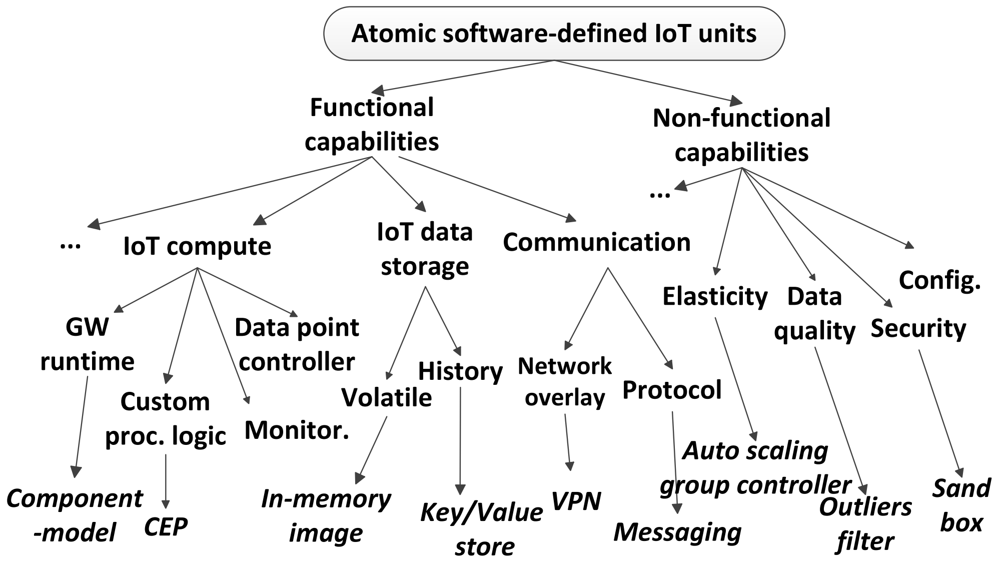


Figure 6.5: Example classification of atomic software-defined IoT units.

The *composed software-defined IoT units* have multiple functional and non-functional

[1]http://ganglia.info/

capabilities, i.e., they are composed of multiple atomic units. Similarly to the atomic units they provide well-defined API, but require additional functionality such as mechanisms to support declaratively composing and binding the atomic units, at runtime (Section 6.3.2). Example of composed unit is a software-defined IoT gateway.

The *complex software-defined IoT units* enable capturing complex relationships among the finer-grained units. Internally, they are represented as a topological network, which can be configured and deployed, e.g., on the cloud. They define an API and can integrate (standalone) runtime controllers to dynamically (re)configure the internal topology, e.g., to enable elastic horizontal scaling of the units. Finally, they rely on runtime mechanism to manage the references, e.g., IP addresses and ports, among the dependency units.

We notice that the software-defined API and our units offer different advantages to the stakeholders involved into designing, provisioning and governing of software-defined IoT systems. For example, IoT infrastructure providers can offer their resources at fine-granularity, on-demand. This enables specifying flexible pricing and cost models and allows for offering the IoT resources as elastic utilities in a pay-as-you-go manner. Because our units support *dynamic and automated composition* on multiple levels, consumers of IoT cloud resources can provision the units to exactly match their functional and non-functional requirements, while still taking advantage of the existing systems and libraries. Further, system designers and operations managers, use late-bound policies to specify and configure the unit's capabilities. Because we treat the functional and configuration units in a similar manner (see Section 6.3.2), configuration models can be stored, reused, modified at runtime and even shared among different stakeholders. This means that we can support *managed configuration models*, which can be centrally maintained via configuration management solutions for IoT cloud, e.g., based on OpsCode Chef[2], Bosh[3] or Puppet[4].

6.3 Main Techniques for Provisioning Software-defined IoT Cloud Systems

6.3.1 Automated composition of software-defined IoT units

Generally, building and deploying software-defined IoT cloud systems includes creating and/or selecting suitable software-defined IoT units, configuring and composing more complex units and building custom business logic components. The deployment phase includes deploying the software-defined IoT units together with their dependency units and required (possibly standalone) runtime mechanisms (e.g., a message broker). In this chapter we mostly focus on provisioning reusable stock components such as gateway runtime environments or available communication protocols.

[2]http://opscode.com/chef
[3]http://docs.cloudfoundry.org/bosh/
[4]http://puppetlabs.org

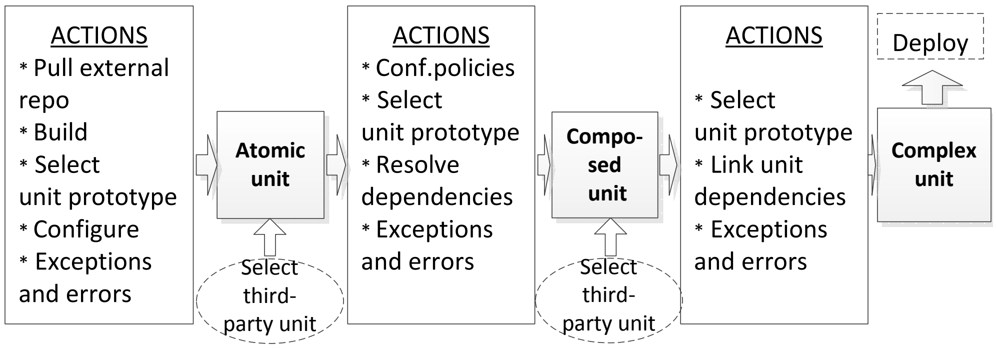


Figure 6.6: Automated composition of software-defined IoT units.

Figure 6.6 illustrates most important steps to compose and deploy our IoT units. There are three levels of configuration that can be performed: (i) Building/selecting atomic units; (ii) Configuring composed units; (iii) Linking into complex units. Each of the phases includes selecting and provisioning suitable unit prototypes. For example, the unit prototypes can be based on different resource containers such as VMs, Linux Containers (e.g., Docker) or OSGi runtime.

The atomic units are usually provided as stock components, e.g., by a third-party, possibly in a market-like fashion. Therefore, this phase usually involves selecting and configuring stock components (e.g., Sedona[5] or Niagara ^{AX}[6] execution environments). Classifications similar to the one presented in Figure 6.5 can be used to guide the atomic units selection process. In case we want to perform custom builds of the existing libraries and frameworks, there are many established build tools which can be used, e.g., for Java-based components, Apache Ant or Maven.

On the second level, we configure the composed units, e.g., a software-defined IoT gateway. This is performed by adding the atomic units (e.g., runtime mechanisms and/or software libraries) to the composed unit. For example, we might want to enable the gateway to communicate over a specific transport protocol, e.g., MQTT and add a monitoring component to it, e.g., a Ganglia agent. To perform this composition seamlessly at runtime, additional mechanisms are required. We describe them in Section 6.3.2.

Third level includes defining the dependencies references between the composed units, which "glue together" the complex units. These links specify the topological structure of the desired complex units. For example, to this end we can set up a virtual private network and provide each unit with a list of IP addresses of the dependency units. In this phase, we can use frameworks (e.g., TOSCA-based, OpenStack Heat, Amazon CloudFormation, etc.) to specify the runtime topological structure of our units and utilize mechanisms (e.g., Ubuntu CloudInit[7]) to bootstrap the composition.

[5]http://www.sedonadev.org/
[6]http://www.niagaraax.com/
[7]http://help.ubuntu.com/community/CloudInit/

6.3.2 Centrally managed configuration models

An important concept behind software-defined IoT cloud systems is the late-bound runtime policies. Our units are configured declaratively, via the policies by utilizing the exposed software-defined API, without worrying about internals of the runtime mechanisms, i.e, the atomic units. To enable seamless binding of the atomic units we provide a special unit prototype, called *bootstrap container*. The bootstrap container provides mechanisms to define (bind) the units based on supplied configurations or to redefine them when configuration policies are changed. Therefore, the units can be simply "droped in" and our bootstrap container (re)binds them together, at runtime without rebooting system. Therefore, in order to support centrally managed configuration models and dynamic feature composition, besides managing the units our provisioning framework is responsible to maintain application-specific configurations. Application configuration models are treated as special components of artifact packages. By decoupling the configuration models from the functional artifacts, we can treat them as any software-defined IoT unit, which adheres to the general principles of software-defined IoT (Section 6.2.1). Our framework provides mechanisms to specify and propagate the configuration models to the edge of IoT cloud infrastructure (e.g., gateways) and our bootstrap container enforces the provided directives.

To support fully-fledged, dynamic feature composition, the configuration container can act as a plug-in system, based on the inversion of control principles. It provides mechanisms to bind the application artifacts (e.g., atomic units) based on supplied configurations or to redefine them when configurations are changed. The container initially binds such functional artifacts based on the configuration models and continuously listens for configuration changes applying them on the affected functional artifacts accordingly. The runtime changes are achieved by invalidating affected parts of the existing dependency tree and dynamically rebuilding them, based on the new configuration directives. This feature is especially useful for managing the communication protocols, which are provided by cloud and device connectivity components (cf. Chapter 7). However, to support dynamic feature composition, our framework requires the artifacts to be wrapped in well-defined APIs, which are known to the provisioning container. Since this imposes some limitations, this feature is optionally provided by the framework. The main advantage of this approach is that it enables updating configuration models without updating the entire artifact package, thus allowing for flexible customizations and dynamic configuration changes without runtime interrupts as well as reducing communication overhead.

6.4 Evaluation & Prototype Implementation

6.4.1 Preliminary implementation of provisioning controller

The main aim of our prototype is to enable developers and operations managers to dynamically, on-demand provision and deploy software-defined IoT systems. This includes providing software-defined IoT unit prototypes, enabling automated unit composition,

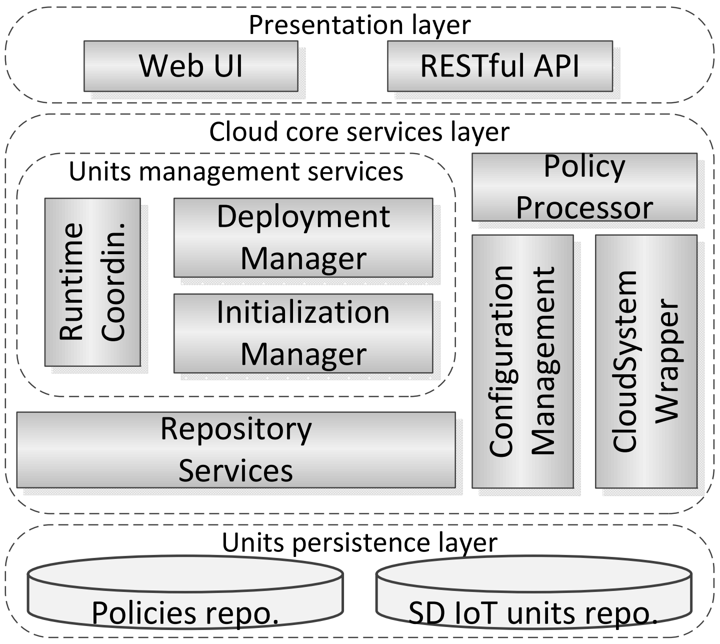


Figure 6.7: Framework architecture overview.

at multiple levels and supporting centralized runtime management of the configuration models.

In Section 6.2 we introduced the conceptual model of our software-defined IoT units. To technically realize our units, we utilize the concept of virtual resource containers. More precisely, we provide different *unit prototypes* that can be customized and/or modified during runtime by adding required runtime mechanisms encapsulated in our atomic units. The unit prototypes provide resources with different granularity, e.g., VM flavors, group quotas, priorities, etc., and boilerplate functionality to enable automated provisioning of custom software-defined IoT units. Figure 6.7 provides a high-level overview of the framework (cloud-based provisioning controller) architecture. Our framework is completely hosted in the cloud and follows a modular design which guarantees flexible and evolvable architecture. The current prototype is implemented atop OpenStack [116], which is an open source Infrastructure-as-a-Service (IaaS) cloud computing platform. *Presentation layer* provides an user interface via Web-based UI and RESTful API. They allow a user to specify various configuration models and policies, which are used by the framework to compose and deploy our units in the cloud. *Cloud core services layer* contains the main functionality of the framework. It includes the *PolicyProcessor* used to read the input configurations and transform it to the internal model defined in our framework. *Units management services* utilize this model for composing and managing the units. The *InitializationManager* is responsible for configuring and composing more complex units. It translates the directives specified in configuration models into concrete initialization actions on the unit level. In our current implementation, the core of the *InitalizationManager* is an OpsCode Chef client, which is passed to the VMs during initialization via Ubuntu cloud-init. *InitalizationManager* also provides mechanisms for configuration management. The *DeploymentManager* is used to deploy the software-defined IoT units in the cloud. Our prototype relies on SALSA[8], a deployment automation framework developed in our department. It utilizes the API exposed by the *CloudSystemWrapper* to enable deployment across various cloud providers, currently implemented for OpenStack cloud. The *DeploymentManager* is responsible to manage and distribute the dependency references for the complex units (Section 6.2.3). The *Units persistence layer* provides functionality to store and manage our software- defined units and policies.

6.4.2 Experiments

Revisiting Motivating Scenario

We now show how our prototype is used to provision a complex software-defined IoT unit, which provides functionality for the real-life FMS location tracking service (Section 6.1.1). The service reports vehicle location in near real-time on the cloud. To enable remote access, the monitored vehicles have an on-board device, acting as a gateway to its data and control points. To improve performance and reliability, the golf course provides on-site gateways, which communicate with the vehicles, provide additional processing and storage capabilities and feed the data into the cloud. Therefore, the physical IoT infrastructure comprises network connected vehicles, on-board devices and local gateways.

Typically, to provision the FMS service system designers and operations manager would need to directly interact with the rigid physical IoT infrastructure. Therefore, they at least need to be aware of its topological structure and devices' capabilities. This means that the FMS service also needs to have understanding of the IoT infrastructure, instead of being able to customize the infrastructure to its needs. Due to inherent inflexibility of IoT infrastructure, its provisioning usually involves long and tedious task such as manually logging into individual gateways, understanding gateway internals or even on site presence. Therefore, provisioning even a simple FMS location tracking service involves performing many complex tasks. Due to a large number of geographically distributed vehicles and involved stakeholders IoT infrastructure provisioning requires a substantial effort prolonging service delivery and increasing costs. Subsequently, we show the advantages our units (Section 6.2.2) and the provisioning techniques (Section 6.3) have to offer to operations managers and application designers in terms of: a) *Simplified provisioning* to reduce time, costs and possible errors; b) *Flexibility* to customize and modify the IoT units and their runtime topologies.

To enable the FMS system we developed a number of atomic software-define IoT units [9] such as: a software-defined sensor that reports vehicle location in realtime, messaging

[8]https://github.com/tuwiendsg/SALSA/
[9]https://github.com/tuwiendsg/SDM

infrastructure based on Apache ActiveMQ [10], software-defined protocol based on MQTT and JSON, the bootstrap container based on the Spring framework[11], and corresponding configuration units. The experiments are simulated on our OpenStack (Folsom) cloud and we used Ubuntu 12.10 cloud image (Memory: 2GB, VCPUS: 1, Storage: 20GB). To display location changes we develop a Web application which displays changes of vehicles' location on Google Maps.

Simplified provisioning

To demonstrate how our approach simplifies provisioning of the virtual IoT infrastructure, we show how a user composes the FMS complex software-defined IoT unit, using our framework. Figure 6.8 shows the custom deployment of the topological structure of the FMS vehicle tracking unit, deployed in the cloud. The unit contains two gateways for the vehicles it tracks, a web server for the Web application and a message broker that connects them.

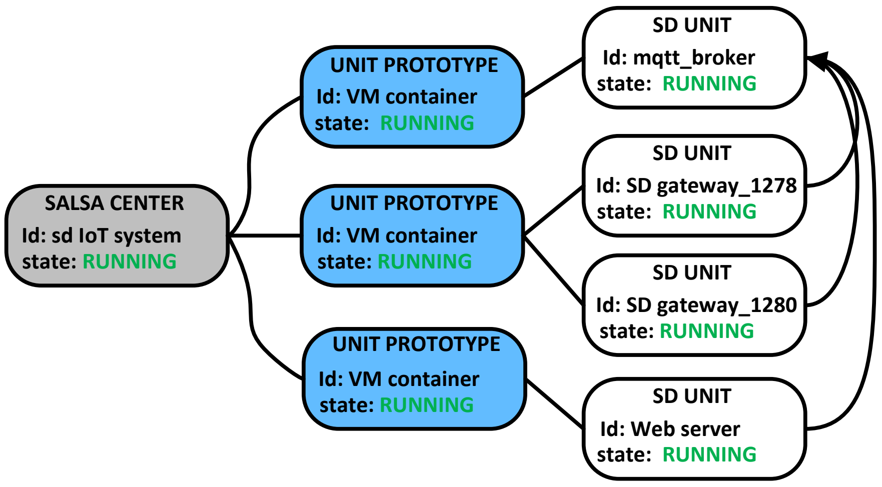


Figure 6.8: Topological structure of FMS vehicle tracking unit (a screen shot).

In order to start provisioning the complex unit, system designer only needs to provide a policy describing the required high-level resources and capabilities required by the FMS service. For example, Listing 6.1 shows a snippet from the configuration policy for FMS location tracking unit, that illustrates specifying a software-defined gateway, for the on-board device.

The policy describes gateway's initial configuration and the cloud instance where it should be deployed. Additionally, it defines a dependency unit, i.e. the MQTT broker and specifies vehicle's Id that can be used to map it on the underlying device. Our framework

[10] http://activemq.apache.org/
[11] http://projects.spring.io/spring-framework/

```
...
<tosca:NodeTemplate id="SD-Gateway"
  name="car_1278" type="vm">
  <tosca:Properties>
   <MappingProperties>
    <MappingProperty type="vm">
     <property name="instanceType">m1.small</property>
     <property name="provider">openstack@dsg</property>
     <property name="baseImage">ami-00000163</property>
    </MappingProperty>
   </MappingProperties>
  </tosca:Properties>
  <tosca:Requirements>
   <tosca:Requirement name="MQTT-broker-IP" type="String"
   id="brokerIp_Requirement"/>
  </tosca:Requirements>
  <tosca:DeploymentArtifacts>
   <tosca:DeploymentArtifact artifactType="chef"
   artifactRef="deployClient"/>
  </tosca:DeploymentArtifacts>
</tosca:NodeTemplate>
...
```

Listing 6.1: Partial TOSCA-like complex unit description.

takes the provided policy, spawns the required unit prototypes and provides them with references to the dependency units. At this stage the virtual infrastructure comprises solely of unit prototypes (VM-based). After performing the high-level unit composition and establish the dependencies between the units, the user continues composing on the finer granularity level. By applying the top-down approach we enable differing design decisions and enable early automation of known functionality, to avoid over-engineering and provisioning redundant resources.

In the next phase, the user provisions individual unit prototypes. To this end, he provides policies specifying desired finer-grained capabilities. Listing 6.2 shows example capabilities, that can be added to the gateway. To enable asynchronous pushing of the location changes it should communicate over the MQTT protocol. Listing 6.3 shows a part of Chef recipe used to add MQTT client to the gateway. Our framework fetches the atomic units, that encapsulate the required capabilities, from the repository and composes them automatically, relying on the software-defined API and our bootstrap container.

```
{"run_list":
  ["recipe[bootstrap_container]",
  "recipe[mqtt-client]",
  "recipe[protocol-config-unit]",
  "recipe[sd-sensor]"]
}
```

Listing 6.2: Run list for software-defined gateway.

```
include_recipe 'bootstrap_container::default'
remote_file "mqtt-client-0.0.1-SNAPSHOT.jar" do
 source "http://128.130.172.215/salsa/upload/files/..."
 group "root"
 mode 00644
 action :create_if_missing
end
```

Listing 6.3: Chef recipe for adding the MQTT protocol.

Therefore, compared to the traditional approaches, which require gateway-specific knowledge, using proprietary API, manually logging in the gateways to set data points, our *automated units composition* (Section 6.3.1), based on declarative unit configuration policies, simplifies the provisioning process and makes it traceable and repeatable. Our units can easily be shared among the stakeholders and composed to provide custom functionality. This enables system designers and operations managers to rely on the existing, established systems, thus reducing provisioning time, potential errors and costs.

Flexible customization

To exemplify the flexibility of our approach let us assume that we need to change configuration of the FMS unit to use CoAP instead of MQTT. This can be due to requirements change (Section 6.1.1), reduced network connectivity or simply to reuse the unit for a golf course with different networking capabilities. To customize the existing unit, an operations manager only needs to change the `recipe[protocol-config-unit]` unit (Listing 6.2) and provide an atomic unit for the CoAP client. This is a nice consequence of our late-bound runtime mechanisms and support for *managed configuration models*, provided by our framework. We treat both functional and configuration units in the same manner and our bootstrap container manages their runtime binding (Section 6.3.2). Compared to traditional approaches that require addressing each gateway individually, firmware updates or even modifications on the hardware level, our framework enables flexible runtime customization of our units and supports operation managers to seamlessly enforce configuration baseline and its modifications on a large-scale.

6.5 Conclusion

In this chapter, we introduced the conceptual model of software-defined IoT units. To our best knowledge this is the first attempt to apply software-defined principles on IoT systems. We showed how they are used to abstract IoT resources and capabilities in the cloud, by encapsulating them in software-defined API. We presented automated unit composition and managed configuration, the main techniques for provisioning software-defined IoT systems. The initial results are promising in the sense that software-defined IoT system enable sharing of the common IoT infrastructure among multiple stakeholders and offer advantages to IoT cloud system designers and operations managers in terms of simplified, on-demand provisioning and flexible customization. Therefore, we believe that software-defined IoT systems can significantly contribute the evolution of the IoT cloud systems.

CHAPTER 7

A Middleware Infrastructure for Utility-based Provisioning of IoT Cloud Systems

The IoT Cloud systems intensively exploit cloud computing models and technologies, predominantly by utilizing large and remote data centers, but also nearby Cloudlets [129] or micro data centers [12] to enhance resource-constrained Edge devices, in terms of computation offloading [36, 29, 98] and data staging [139] or to provide an execution environment for cloud-centric IoT applications [42, 106].

One of the main advantages of cloud computing is reflected in its support for self-service, on-demand resource consumption. To date, we have witnessed numerous benefits of such utility-based provisioning model, in terms of more flexible and cheaper IT operations [23, 90]. Therefore, it would be natural to expect that this flagship property of cloud computing would be inherited by IoT Cloud as well. Unfortunately, this is still not the case, mainly because current approaches, dealing with IoT Cloud provisioning, mostly focus on providing virtualization solutions for the Edge devices [43, 160, 134]. Although device virtualization is a key precondition for utility-based provisioning, such approaches are usually meant to support a specific task, e.g., data integration or data-linking, which contrasts consuming the IoT Cloud resources as general-purpose utilities. Furthermore, most of the contemporary support for IoT Cloud provisioning, in terms of available tools, frameworks and middleware provide only partial solutions, which discriminate against some of the inherent properties of IoT Cloud infrastructures such as heterogeneity, geographical distribution, and the sheer scale of such infrastructures. Therefore, system integrators and operations managers have to rely on provisional solutions, which require combining multitude of provisioning techniques such as manual, script- and service-based provisioning. Additionally, many of these approaches implicitly assume manual logging into Edge devices or even physical on-site presence, making them hardly feasible in practice. The issue is further exacerbated due to strong dependence of IoT Cloud applications on specific properties of the underlying Edge devices (e.g., available sensors) and novel resource features, that also need to be considered during application provisioning. This inherently prevents consuming IoT Cloud infrastructure as traditionally generic compute or storage utilities, thus requires rethinking existing support for representing infrastructure resources, managing their configuration and deployment models as well as composing low-level resource components into usable infrastructures, capable to support novel business logic requirements.

In this chapter, we continue our line of research towards utility-based provisioning of IoT Cloud systems and introduce a novel provisioning middleware for IoT Cloud. Our middleware builds on the previously introduced concepts and frameworks Chapter 6, extending them with a comprehensive support for scalable multi-level provisioning of IoT Cloud systems. This is one of the crucial precondition for realizing utility-based provisioning paradigm in IoT Cloud systems. The main features of our middleware include: i) Support for automated provisioning of infrastructure resources, application components and configuration models in a uniform, logically centralized manner through dynamically managed APIs; ii) Extensible and flexible provisioning models, which support self-service, on-demand consumption of Edge-device resources; iii) A generic, light-weight resource abstraction mechanism, which allows for application-specific customizations of and virtually exclusive access to low-level devices, e.g., sensors and actuators, with well-defined APIs.

The remainder of the chapter is organized as follows: Section 7.1 presents main research challenges and the research context; In Section 7.2 we introduce our middleware and discuss it's architecture in detail; Section 7.3 outlines the major runtime mechanism for multi-level provisioning; Section 7.4 describes experimental results and outlines the current prototype implementation; Finally, Section 7.5 concludes the chapter and gives an outlook of our future research towards realizing fully-fledged utility-based provisioning in IoT Cloud.

7.1 Motivation & Research Challenges

In Chapter 6, we have introduced a conceptual model for software-defined IoT Cloud systems. The core concept of the provisioning model are *software-defined IoT units*. They describe IoT Cloud resources (e.g., virtual sensors), their runtime environments (e.g., gateways) and capabilities (e.g., communication protocols or data point controllers). Such units are used to encapsulate the IoT Cloud resources and abstract their provisioning in software. To this end, they expose well-defined APIs and can be composed at different levels, creating virtual runtime infrastructures for IoT Cloud applications.

The main purpose of such software-defined IoT Cloud infrastructures is to enable *utility-based provisioning of IoT Cloud resources* by providing a uniform and logically centralized view on the entire underlying resource pool, as well as by allowing IoT Cloud applications to customize and consume those resources dynamically and on-demand.

However, due to dynamicity, heterogeneity, geographical distribution and sheer scale of such infrastructures, achieving these features poses a number of challenges. To better motivate our work, in continuation we discuss the properties of IoT Cloud infrastructures and derive a set of key research challenges that currently prevent utility-based provisioning of IoT Cloud resources.

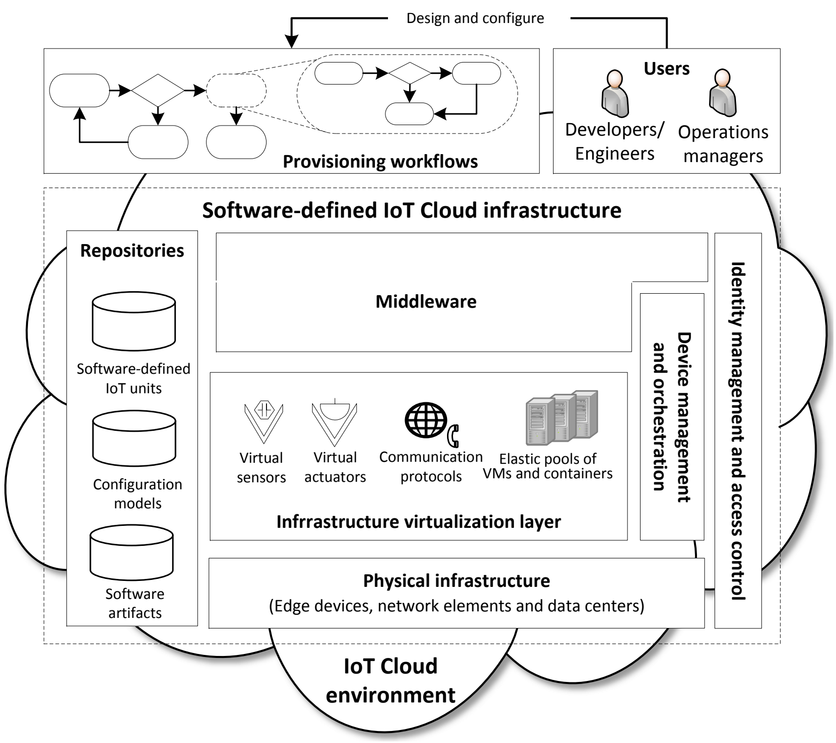

Design and configure
Provisioning workflows
Users
Developers/ Engineers
Operations managers
Software-defined IoT Cloud infrastructure
Repositories
Software-defined IoT units
Configuration models
Software artifacts
Middleware
Virtual sensors
Virtual actuators
Communication protocols
Elastic pools of VMs and containers
Infrrastructure virtualization layer
Device management and orchestration
Identity management and access control
Physical infrastructure
(Edge devices, network elements and data centers)
IoT Cloud environment

Figure 7.1: Overview of Software-defined IoT Cloud Infrastructure.

Figure 7.1 depicts a high-level architecture overview of the software-defined infrastructures, and shows how the main stakeholders interact with such infrastructures. The bottom layer represents the *Physical infrastructure*, which comprises a variety of geographically-dispersed edge devices (e.g., sensors and gateways), network elements (routers and switches) and large data centers. In reality, the physical infrastructure is usually not flat and follows a hierarchical structure, where sensors and actuators are connected to data centers via gateways, which are intermediary nodes that mediate the communication, but also provide constrained computational and storage resources, which are currently largely underutilized. Additionally, it is common to strategically place more powerful processing nodes near the Edge (but within the hierarchy), such as Cloudlets and micro data centers. The communication between the Edge and the data centers is realized over heterogeneous networks which include wired, wireless and cellular communication channels. Moreover, IoT Cloud infrastructure is highly-decentralized and distributed among multiple geographical regions and organizations.

A distinguishing feature of the software-defined IoT Cloud infrastructures is the *Infrastructure virtualization layer*. A number of existing approaches deal with the Edge devices virtualization, exposing them to the upper layers on different levels of abstraction. Most relevant approaches for our discussion are centered around the Unikernels and kernel-supported virtualization, which is discussed in Section 7.2. Other related approaches, such as software-defined networking (SDN) and semantics-based data integration are discussed in Chapter 10.

The *Middleware* is a crucial part of software-defined IoT Cloud infrastructure and, in general, its main responsibility is to provide a uniform representation of the underlying (virtual) infrastructure resources as well as to enable delivery and consumption of such resources. This layer needs to provide mechanisms and tools for infrastructure provisioning, managing configuration models and deployment of applications. The middleware relies on and utilizes a number of different components. In the following, we only briefly discuss those components since they are out of scope of this thesis, although being the main focus of numerous research and industry approaches, e.g., [54], which can be used to complement our approach.

The *Device management and orchestration* component is generally responsible to support discovering and managing physical Edge devices (e.g., to detect newly connected devices), monitoring their status, but also to enable mapping and allocation of virtual resources to the underlying devices. The *Repositories* are used to provide persistent storage facilities for configuration models, infrastructure automation scripts and software-defined units, which are delivered and deployed on the devices by the middleware. The *Identity management and access control* generally deals with assigning and managing dedicated, unique names (IDs) to individual devices, but also provides security techniques to determine which devices are permissible to be provisioned as IoT Cloud resources.

7.1.1 Research Challenges

The utility-based provisioning is a well-established and proven concept in cloud computing [23, 91]. Among other things it requires: on-demand, self-service usage models; enabling ubiquitous access to a shared pool of configurable resource, which can be customized to exactly meet application requirements; as well as autonomous and automated allocation of the consumed resources. However, given the previously described properties of IoT Cloud, realizing these features in the context of IoT Cloud systems is a non-trivial task which creates a number of challenges that need to be addressed.

One of the main challenges is to support *on-demand, self-service usage model*, because it requires support for uniform interactions with the large-scale, heterogeneous IoT Cloud resource pool. This could potentially be achieved by virtualizing and encapsulating the IoT Cloud resources into well-defined APIs and allowing the users to access such resources on multiple levels of abstraction. However, in this case the middleware (cf. Figure 7.1) needs to provide support for a non-trivial task of managing such virtual resources, theirs APIs and mediating all the communication with the heterogeneous devices.

Assuming that IoT Cloud resources are accessible in a uniform manner, another challenge is to enable the users to *automatically provision IoT Cloud resources*. However, strong dependence of IoT Cloud applications on specific properties of the underlying devices and novel resource features intrinsically prevent consuming IoT Cloud infrastructure as traditionally generic compute or storage utilities. This requires providing comprehensive provisioning support on multiple levels such as infrastructure-, platform- and application-level. One way to achieve this is by utilizing provisioning workflows [74] (cf. Figure 7.1 (top)). The main advantage of the workflow approach is that it allows for nested provisioning workflows (shown as dotted nodes in the figure), which are well suited for multi-level provisioning. However, to support their execution on a large resource pool the middleware needs to enable elastically scalable execution of the provisioning tasks.

Enabling ubiquitous access to the large, geographically-distributed resource pool is yet another challenge since it demands a *logically centralized interaction* with underlying devices. However, since the underlying devices are inherently dispersed, the middleware needs to be distributed across the resource-constrained devices, thus optimized for such constrained execution environments. Moreover, to support customizing such resources, the middleware needs to support *management of application components and configuration models*, but also provide suitable mechanisms to dynamically deliver and (re)enforce the configuration models inside the Edge devices.

7.2 IoT Cloud Provisioning Middleware

With respect to the components presented in Figure 7.1, the main focus of this chapter is the Middleware layer. The main purpose of our middleware is to facilitate implementing and executing provisioning workflows in IoT Cloud systems, by addressing the previously-described challenges and enabling the remainder of the design principles, introduced in Chapter 6. The support for the multi-level provisioning is thoroughly discussed in Section 7.3. At the moment it is important to note that IoT Cloud provisioning involves two main tasks: i) *allocating and deploying Software-Defined Gateways (SDG)*, which are a special type of aforementioned software-defined IoT units and ii) *customizing software-defined gateways with application-specific artifacts*.

Figure 7.2 gives a high-level architecture overview of our middleware. Generally, the provisioning middleware is designed based on the microservices architecture [95] and it is distributed across the Cloud and Edge devices. The main components of the provisioning middleware include: i) the *Software-Defined Gateways*, ii) the *Provisioning and Virtual Buffers Deamons* that run in Edge devices and iii) the *Provisioning Controller* which runs in the Cloud. In the remainder of this section, we discuss these components in more detail.

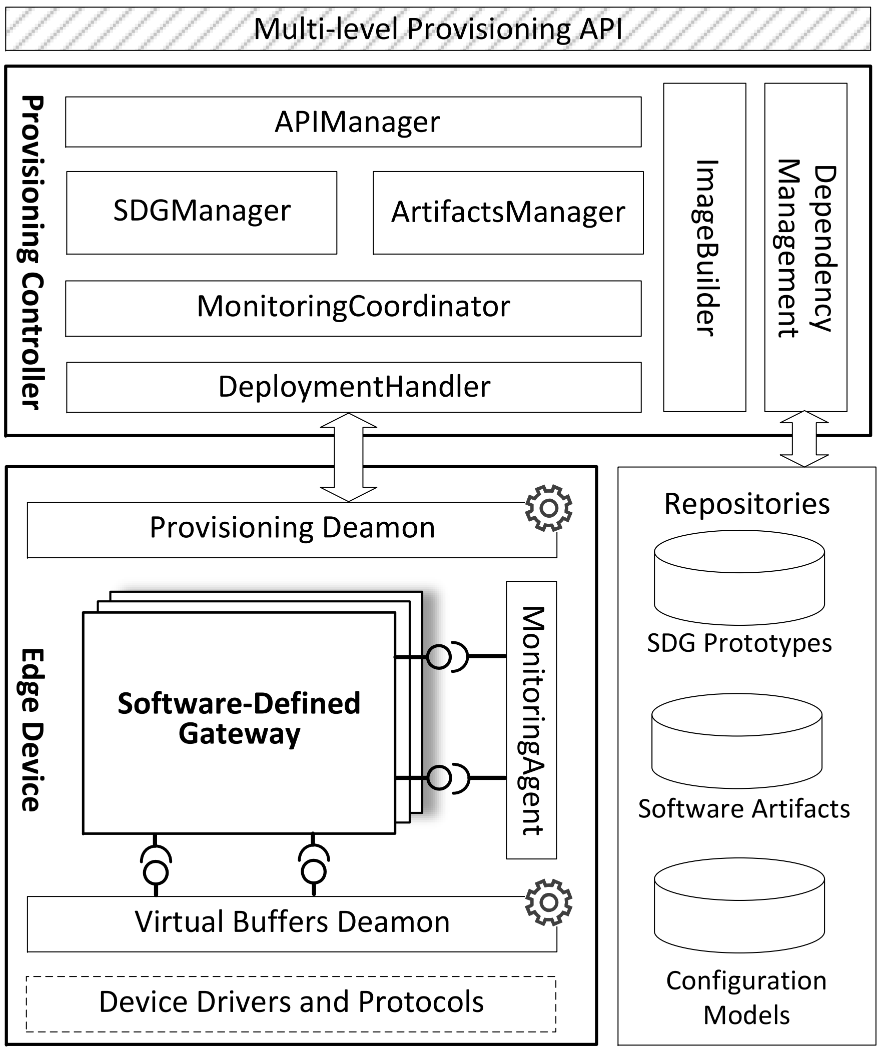


Figure 7.2: Architecture overview of the provisioning middleware.

7.2.1 Software-defined gateways

The software-defined gateways are one particular type of the software-defined IoT units and their main purpose is to support virtualizing the IoT Cloud compute resources, most notably Edge devices, in order to provide isolated and managed application execution environments. Our middleware does not support building custom SDGs from scratch, instead it provides, so called, SDG prototypes and required mechanisms to customize them, based on application-specific requirements. At their core SDG prototypes define an isolated runtime environment for the SDGs and application-specific components. To this end, the main purpose of SDG prototypes is to provide isolated namespaces as well as limit and isolate resource usage such as CPU and memory. Therefore, the SDG prototypes are used to bootstrap higher-level SDG functionality. In Figure 7.3 the double line shows the virtual boundaries of the SDG prototypes. It is important to mention that SDG prototypes do not propose a novel virtualizaton solution, but rely on proven techniques, namely kernel-supported virtualization approaches, which offer a number of light-weight execution environments/drivers such as LXCs, libvirt-sandbox or even chroot, generally referred to as containers that can be used to "wrap" SDGs. Conceptually, virtualization choices do not pose any limitations, because by utilizing the well-defined APIs, our SDGs can be dynamically configured, provisioned, interconnected and deployed, at runtime. The SDG prototypes are hosted in the IoT Cloud and enriched with functional and provisioning capabilities, which are exposed via well-defined APIs. There is a number of middleware components (cf. Figure 7.3), which are pre-installed (except for Artifact Packages) in each SDG prototype in order to support such APIs. Next, we discus these components in more detail.

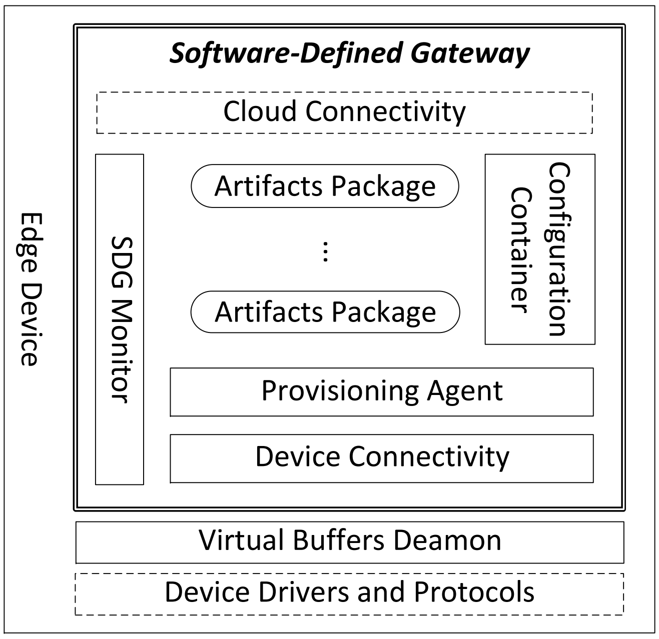


Figure 7.3: Software-defined gateway architecture.

Artifact Packages

Generally, IoT Cloud applications consist of different application components and supporting files (e.g., libraries and binaries), which we refer to as application-specific artifacts. Such artifacts are deployed, configured and executed inside software-defined gateways. Generally, our provisioning middleware does not make any assumptions about application model or concrete artifact implementations. However, in order to enable automated artifacts provisioning, it requires them to be packaged as shown in Figure 7.4. There are two important things to mention here. First, the Artifact Package needs to contain a set of provisioning directives with all the necessary instructions such as installing and uninstalling the package. When a provisioning workflow submits a provisioning request, the middleware maps the request to a concrete implementation of provisioning directive. To support implementing such directives, in Chapter 5 we have introduced a light-weight provisioning DSL. Second, the packages contain Meta-information such as artifacts' hardware requirements and exposed APIs. The specification of the APIs is optional, but they are needed by the middleware if an application wants to completely delegate management of its configuration models to the middleware, as we discussed in Chapter 6.

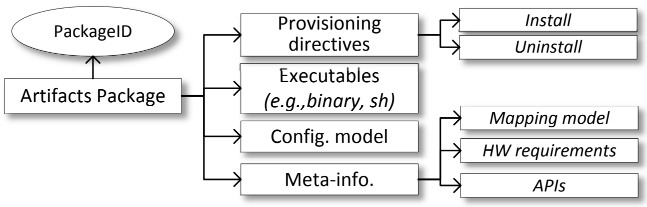


Figure 7.4: Artifacts package structure.

Provisioning Agent

All packages that are not pre-installed on the Edge devices have to be provisioned by the framework during runtime. For this purpose, our middleware provides a light-weight *Provisioning Agent*, which is pre-installed inside SDGs. The agent continuously runs in each SDG and manages local artifact packages. The main responsibility of the provisioning agent is to periodically inspect the Provisioning Controller (cf. Figure 7.2) update queue, download the artifact packages and execute directives referenced in provisioning workflows. Additionally, the agent acts as a local interpreter of provisioning directives specified via our aforementioned provisioning DSL. The agent is also responsible to handle various requests initiated by the Provisioning Controller, by triggering the required actions in SDGs such as creating a snapshot of the current device state via the SDGMonitor and uploading the snapshot to the Controller. The SDGMonitor is discussed together with the Monitoring Agent later in this section.

Device Connectivity

The SDGs are deployed on Edge devices with limited privileges in the sense that they are not permitted to directly access the hardware. An obvious reason for such limitation is security, but also resource contentions and customization requirements, since we can have multiple SDGs executing in same Edge device simultaneously. To enable applications to access the underlying devices, e.g., sensors, SDG offers *Device Connectivity* component. The main part of the device connectivity is a SDG endpoint, which exposes the devices to the SDG and enables service-based interaction with them. The SDG endpoint is a single point of interaction with the underlying *Virtual Buffers Deamon* (cf. Figure 7.3) and at the moment, it is defined up to the transport layer. For this reason the device connectivity component provides a pluggable connectivity layer, which is by default preconfigured with our custom, REST-like application-level protocol. In the current prototype we have also implemented CoAP and MQTT communication protocols, but the device connectivity can be easily extended by plugging in other application-level protocols such as sMAP [38].

7.2.2 Edge Device middleware support

In order to support management of SDGs in Edge devices, our middleware provides lightweight components that are pre-installed and continuously run inside the Edge devices. The most important components are the Virtual Buffers Deamon and Provisioning Deamon, shown in Figure 7.2 on the right-hand side.

Virtual Buffers Deamon

We have discussed how our software-define gateways can be used to virtualizing compute resources of the Edge devices. However, since the SDGs run with reduced privileges, the middleware also needs to virtualize accessing the low-level devices such as sensors and actuators. To this end it provides the Virtual Buffers Deamon (VBD). The main purpose of the VBD is to mediate the communication with the devices connected to a field bus (e.g., via Modbus, CAN, SOX/DASP, I^2C or IP-based) and to provide a virtually exclusive access to such device. In general, the deamon act as multiplexer of the data and control channels, thus enabling the SDGs to have their own view of and define custom configurations for such channels. For example, a software-defined gateway can configure sensor poll rates, activate a low-pass filter for an analog sensory input or configure unit and type of data instances in the stream. Figure 7.5 depicts a simplified UML diagram of the VBD's most important components. The main concept behind VBD are the VirtualBuffers. Generally, the main goal of the virtual buffers is to provide virtual representation of sensors and actuators. They wrap the DeviceDrivers and share a common behavior with them, inherited through the Component Interface. For example, they can be initialized, shutdown and released. Both buffers and drivers lifecycle are managed by the VirtualBuffersManager. The DeviceDrivers Package contains a set of driver implementations. For readability purposes, in the figure we only show the component for I^2C protocol, but each implementation follows similar principle. It contains a set of Ports, which is a VBD internal representation of devices attached to the bus. Such Ports are dynamically instantiated by the VirtualBuffersManager at device bootup during driver initialization phase, based on the provided PortConfig. At the moment, PortConfig is specified as a JSON file that contains the meta-data such as port class (e.g., analog in), name and hardware-related data, e.g., multiplexer address or value correction constants. One of the limitations of the current implementation is that it does not support dynamic device reconfiguration, meaning that if low-level configurations change the VBD must be restarted. Moreover, a virtual buffer references a set of Gatherers and can

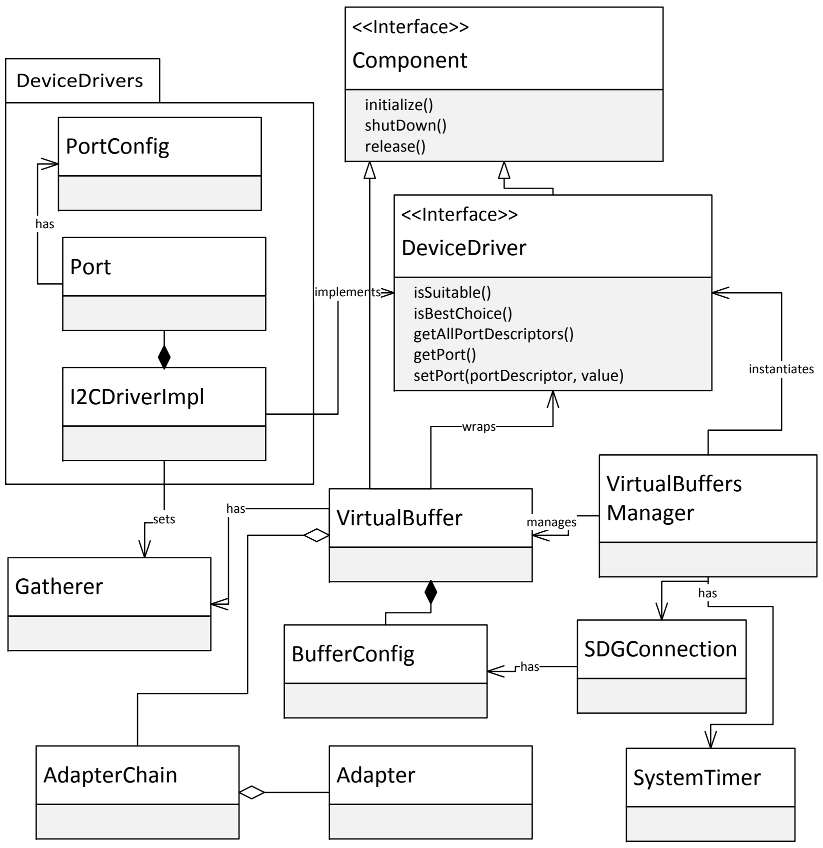


Figure 7.5: Simplified UML diagram of Virtual Buffers Deamon.

contain an optional AdapterChain. Generally, a gatherer is a higher level representation of a port. For example, in case of a sensing device the gatherer represents the most resent value of the hardware interface. To support SDG-specific configurations such as sensor poll rate, filters or scalers, each virtual buffer can have an AdapterChain. Adapter chains reference different Adapters, which are specified and parametrized via BuferConfig. For example, a raw sensing value is passed through such adapter chain before being delivered to a SDG. Finally, the VBD is responsible to instantiate and maintain an open communication channel with software-defined gateways (via SDGConnection) and keep track of the mappings among the SGDs and their VirtualBuffers.

Provisioning Deamon

So far, we have tacitly assumed that SDGs are readily available and deployed in Edge devices. However, this is naturally not the case, thus the SDGs need to be dynamically allocated, instantiated and deployed on Edge devices. These tasks are shared responsibility of the Provisioning Deamon and the Provisioning Controller.

Generally, the provisioning deamon serves two main purposes: i) It continuously runs in each Edge device and provides functionality to remotely manage the SDGs. The remote endpoint is middleware's Provisioning Controller. ii) It acts as a local proxy to the provisioning agents running inside each SDG, mediating all the previously-described provisioning communication with SDGs (Provisioning Agents). At its core the provisioning deamon has a light-weight httpd server to allow for a bidirectional communication between the Provisioning Controller and the Edge devices (i.e. SDGs). It is designed as a pluggable component, which relies on the existing support for managing shared hosting domains (i.e., containers) such as Docker, LXD or virsh. In this context, the main components of the provisioning deamon are an InvocationMapper and a set of plug-in components called Connectors. Among other things, the InvocationMapper is responsible to handle the provisioning requests form the controller and map them to the corresponding Connector as well as to obtain the required SDG prototypes form the Respositories and locally manage their images. The connectors act as wrappers of the underlying mechanisms for managing SDGs, exposing them to the InvocationMapper via uniform APIs. Therefore, to use a different virtualization solution for SDGs, one only needs to develop the needed connector and register it with the InvocationMapper. Second, the provisioning deamon mediates the communication with the SDG provisioning agents. To support this, it manages local network interfaces of SDGs and behaves like a transparent proxy for the inbound communication. Regarding the outbound communication the provisioning deamon treats the monitoring responses in a particular manner. It intercepts the monitoring information delivered by SDGMonitors and enriches it with the current device state information, delivered by the MonitoringAgent (cf. Figure 7.2). The *MonitoringAgent* is used to collect meta information about the SDGs such as ID, but also to continuously monitor the underlying system via available interfaces in order to provide dynamic device information. To this end, it executes a sequence of runtime monitoring actions to complete the dynamic device state-snapshot. For example, such actions include: currently available disk space, available RAM, firewall settings, environment information, list of processes and daemons, as well as a list of currently installed and running SGDs. The created snapshots are transmitted to the Provisioning Controller periodically or on request. The device snapshot is also used by the InvocationMapper to determine if a new SDG can be instantiated and deployed on the Edge device, since current virtualization management solutions only provide a rudimentary support in this regard.

7.2.3 Cloud-based Provisioning Controller

The Provisioning Controller (cf. Figure 7.2) is the cloud counterpart part of our middleware. It provides a mediation layer that enables the users to interact with IoT Cloud in a conceptually centralized fashion, without worrying about geographical distribution and heterogeneity of the underlying Edge devices. Internally, the Provisioning Controller comprises several microservices: *APIManager, MonitoringCoordinator, SDG- and ArtifactsManager, DeploymentHandler, ImageBuilder and DependencyManagement.*

The main responsibility of the *APIManger* is to manage the *Multi-level Provisioning API*, i.e., it encapsulates the middleware provisioning capabilities in well-defined APIs and handles all API calls from user-defined provisioning workflows. Although our middleware provides multi-level provisioning support, this distinction is only relevant to the middleware internal components, since APIManager hides all such details from the users, who effectively observe only simple API calls and corresponding responses. Therefore, the APIManager is responsible to resolve incoming requests, map them to the respective handlers, i.e., *SDGManager* or *ArtifactsManager* (depending on the request type), and deliver results to the calling workflow. Among other things, the actions performed by these managers involve selecting requested SDGs or artifacts by querying the corresponding SDG- and Artifacts-Repository, building the package images and deliver them to the Edge devices. In Section 7.3, we describe this process in more detail.

Since majority of application artifacts and SDG images are not readily available in Edge devices, the *DeploymentHandler* is responsible to deliver them to the Edge devices (i.e., *Provisioning Deamons*) or SDGs (i.e, *Provisioning Agents*) at runtime. The DeploymentHandler relies on the *DependencyManagemet service* to resolve the required artifact dependencies and *ImageBuilder* to prepare (package and compress) them into deployable images. Resolving the dependencies on the cloud is particularly useful, because it saves a lot of processing and networking, from the perspective of whole IoT Cloud infrastructure, since otherwise each Edge device would have to perform the same set of actions, e.g., downloads. Furthermore, as opposed to fully-fledged OS distros, the Edge devices usually provide limited support in terms of packaging or updating tools, since they often run striped down user land such as BusyBox.

To create the aforementioned deployable images, our middleware uses the *Image-Builder*. In order to build an image, the builder performs the following steps: (i) retrieve gateway-specific information from the IoT gateway management, (ii) use the dependency management service to gather a list of suitable plans, (iii) based on the plan, build an image, (iv) if the build was successful, hand over to the deployment handler, (v) if the build failed try next plan in list. Finally, all device state-snapshots are maintained by the MonitoringCoordinator, which manages static device meta-information and periodically sends monitoring request to the *MonitoringAgent* in order to obtain runtime snapshots of current device state. The role of the MonitoringCoordinator and the MonitoringAgents is described in more detail in Section 7.3.

7.3 Runtime Mechanisms for Multi-level Provisioning in IoT Cloud

7.3.1 Runtime execution of provisioning workflows

In general, to provision (a part of) an IoT Cloud application a user might design a workflow resembling our example provisioning workflow shown in Figure 7.6 at the top. Individual actions of such workflow usually reference specific provisioning capabilities, exposed via the middleware APIs, and rely on the middleware to support their execution. Usually, the main execution thread of provisioning workflows (denoted by the solid lines in our example provisioning workflow), represents provisioning directives for the infrastructure-level, such as to deploy a SDG of a specific type on Edge devices (in this case based-on BusyBox) or spin-up a cloud-based Message Queue Broker, e.g., MQTT Broker. The sub-workflows (denoted by dashed lines in the same example), are mainly used to specify application-level provisioning directives. As previously mentioned, this involves customizing the SDGs with the application-specific artifacts and configuration models. For example, this can involve deploying, configuring and starting an application service.

The Figure 7.6 also depicts a simplified sequence of steps performed by the middleware when executing a provisioning workflow. For the sake of clarity, we omit several steps and mainly focus on showing the most common interaction, e.g., we assume no errors or exceptions occur and we do not show interaction with the Repositories.

A provisioning workflow requests an application artifact or a SDG by specifying their respective IDs (currently consisting of a name and a version number) and a specific Edge device ID. Next, the workflow invokes a specific API, e.g., to install or uninstall the artifact. At this point the middleware attempts to execute the specified provisioning directive. The steps 1 to 7 in Figure 7.6 depict the most important actions performed by our middleware in order to support an infrastructure-level provisioning request, e.g., to deploy, instantiate and start a SDG in an Edge device. Therefore, the middleware performs the following actions: i) The *APIManager* initially evaluates the composite predicates (described later in this section) in order to determine a set of devices on witch the SGD will be deployed; ii) The *SDGManager* selects device compatible SDG prototype and registers it with the *DeploymentHandler*; iii) The *MonitoringCoordinator* together with *MonitoringAgent* checks the SDG against current device-state snapshot; iv) The *DeploymentHandler* transfers the SDG prototype image to the *Provisioning Deamon*; v) The *ProvisioningDeamon* configures the SDG's local network interface (based on the supplied mapping model), starts the SGD and registers the new SDG instance with the *Virtual Buffers Deamon*; vi) Finally the *Virtual Buffers Deamon* allocates a set of dedicated virtual buffers and creates a dedicated SDGConnection handler. At this point the SDG instance is running in the Edge device and it is performing internal initialization actions such as starting the Configuration Container, the Provisioning Agent and its local SDG Monitor. After the final initializations the SDG transmits its initial device state to the controller and it is ready to handle application-level provisioning requests.

To support an application-level provisioning request the provisioning middleware performs the following actions (steps 8 to 13 in Figure 7.6): i) Similarly to the step 3 each application artifact is checked against current SDG-state snapshot, delivered by the *SDG Monitor*; ii) The *Dependency Management Service* resolves runtime dependencies of the artifact; iii) The *PackageManager* builds a deployable image and registers it with the *DeploymentHandler*; iv) Similarly to the step 5 the *DeploymentHandler* deliveres the image to the *Provisioning Deamon*; v) Finally, the *Provisioning Deamon* transparently

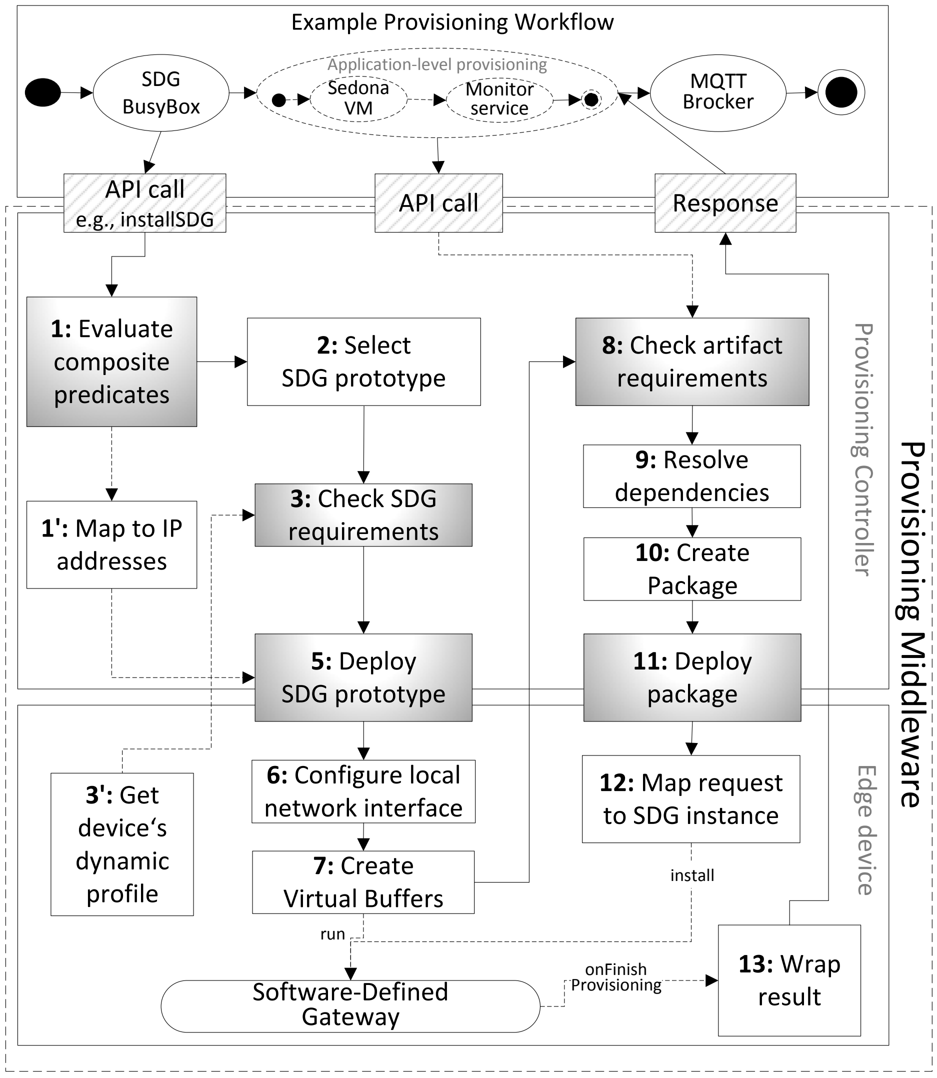


Figure 7.6: Runtime execution of a provisioning workflow.

forwards the image to the SDG's *Provisioning Agent*, which installs the package locally in the SDG. In the remainder of the section we describe the most important runtime mechanisms in more detail.

7.3.2 Evaluating composite predicates

While describing the main steps of the provisioning process, we have mostly focused on the steps performed for a single device and a single SDG. However, usually the provisioning workflows are meant to provision multiple devices, e.g., that share some common properties or belong to the same organization. Therefore, the same provisioning logic should be applicable regardless of specific devices. In this context, it is particularly important to support designing generic provisioning workflows, in the sense that such workflows should be defined independently of the Edge devices, e.g., without referencing device IDs. One of the main preconditions for this is to support the users to dynamically delimit the range of provisioning actions. In our middleware this is achieved by allowing the users to specify the required device properties, as a set of composite predicates. Such predicates reference device or SDG meta information and are used to filter out only the matching devices, which meet the specified criteria. These predicates are specified by the users and delivered to the middleware in a provisioning request as POST parameters.

To bootstrap delimiting the range of a provisioning action, our middleware maintains a set of available devices for a particular user. The current prototype always considers all the connected devices, since at the moment there is only a limited support for managing the device identities and the access control. However, this is not a conceptual drawback and there are many available solutions, which can be used to provide this functionality (as discussed in Section 7.1). The predicates are applied on this set, filtering out all resources that do not match the provided attribute conditions. The middleware uses the resulting set of resources to initiate the provisioning actions with *SDG- and AtrifactsManager*. These managers are also responsible to provide support for gathering results delivered by the *ProvisioningDeamons* and the *ProvisioningAgents*, once the provisioning action is completed (cf. Figure 7.6 step 13). This is needed since after the resources are selected, provisioning actions are performed in parallel and the results are asynchronously delivered to provisioning workflows.

7.3.3 Artifacts and SDGs prototypes runtime validation

Since we are dealing with resource-constrained devices, before deploying a SDG or application artifact the middleware needs to verify that the component can be installed on a specific device, e.g, that there is enough disk space available. This happens during step 3 (Check SDG requirements) and step 8 (*Check artifact requirements*). To this end, the *MonitoringCoordinator* first queries the Repositories. Besides the artifact binaries and SDG prototypes, the repositories store corresponding meta-information, such as required CPU instruction set (e.g., ARMv5 or x86), disk space and memory requirements. After obtaining the meta-information our middleware starts building the current device state-snaphshot. This is done in two stages. First, the device features catalog is queried to obtain relevant static information, such as CPU architecture, kernel version and installed userland (e.g., BusyBox [22]) or OS. Second, the *MonitoringCoordinator* in coordination with the *MonitoringAgent* and *SDGMonitor* executes a sequence of runtime profiling actions to complete the dynamic device state-snapshot. For example, the profiling actions include: currently available disk space, available RAM, firewall settings, environment information, list of processes and daemons, and list of currently installed capabilities. Finally, when the dynamic device snapshot is completed, it is compared with the SDG's/artifact's meta information in order to determine if they are compatible with the device. In this context, the middleware performs in a similar fashion to a fail-safe iterator, in the sense that it works with snapshots of device states. For example, if something changes on the device side, during step 3 or step 8, it cannot be detected by the middleware and in this case its behavior is not defined. Since we assume that all the changes to the underlying devices are performed exclusively by our middleware, this is a reasonable design decision. Other errors, such as failure to install an artifact, in a specific SDG, are caught by the middleware and delivered as notifications to the provisioning workflow, so that they do not interrupt its execution. With this approach the middleware is capable to make autonomous decisions about the provisioned resource. This is one of the main preconditions for supporting automated execution of provisioning workflows, but also for enabling on-demand, self-service provisioning model, since our middleware does not make any implicit assumptions such as user awareness of device properties nor it requires them to manually interact with the underlying devices.

7.3.4 Provisioning models

One of the main goals of our middleware is to support on-demand resource consumption. Previously, we have discussed some of the key preconditions such as the ability to execute multiple SDGs inside an Edge device as well as to dynamically and automatically determine if a SDG or an application package is suitable for a particular device, based on the monitoring device-state snapshot. In the following we discuss the provisioning models currently supported by the middleware prototype and discus some possible optimizations. After the *MonitoringCoordinator* determines an SDG/package is compatible with Edge devices, the middleware needs to create a SDG or Artifact image and deliver it to these devices (steps 5 and 11 in Figure 7.6). This process requires the middleware to make the following decisions: what to deliver to the devices, how to deliver it and where to host the image. Therefore the image delivery process is structured along these three main phases.

Delivery models

In the first phase, the middleware needs to chose whether to deliver a complete image or only a download script. In the first case the *ImageBuilder* creates a SDG or an Artifact image, which is essentially a compressed Artifact Package or SDG prototype. This image is then registered with the *DeploymentHandler* by a corresponding manager, which transfers the whole image to the *ProvisioningDeamon*. The second case the process is done in a similar fashion, but in addition to the image the *ImageBuilder* also generates a download script. The main part of this script is an URL of the location where the actual image resides. Instead of the whole image only this script is sent to the *ProvisioningDeamon*, so it can download and install the image. Since both of these approaches have their advantages [123], the middleware leaves it to the users to make a decision, i.e., to select the most suitable approach and pass it as a configuration parameter in the provisioning request.

Deployment models

In the second phase the *DeploymentHandler* deploys the image (or the download script) to the device. We support two different deployment strategies. The first strategy is poll-based, in the sense that the image is placed in a queue and remains there for a specified period of time (TTL). Both *ProvisioningDeamons* and *ProvisioningAgents* periodically inspect the queue for new provisioning requests. When a request is available, the device can poll the new image when it is ready, e.g., when the load on it is not too high. Although, a provisioning workflow can specify the image priority in the queue, if a device is busy over longer period of time, e.g., there is not enough disk space to install a SDG, this can lead to a request starvation, blocking the execution of the provisioning workflow. For this reason our middleware also supports a push-based deployment. In this case, instead of waiting in the queue, an image is immediately pushed on a device. This gives a greater control to the provisioning framework, but since the previously described image runtime validation performs in a fail-safe manner, the push-based deployment can lead to an undesired behavior. Therefore, when using this strategy a provisioning workflow should also provide compensation actions, to return the device in the previous state. Naturally, these two strategies can be used to create hybrid deployment strategies, such as using the pool-based approach for SDG prototypes and the push-based approach for application artifacts, because pushing artifacts is particularly useful for security updates of hot fixes in SDGs.

Placement models

Finally, the middleware decides where to host the image. This largely depends on a specific deployment strategy, but also on the delivery model. For example, for push-based deployment the *DeploymentHandler* stores the images in-memory, also the download scripts are always kept in-memory, but in case of pool-based strategy, images are usually hosted in middleware local *Repositories*. However, it is not difficult to imagine more complex provisioning models, which can be put in place in order to optimize the provisioning process, e.g., to save bandwidth. For example, to achieve this, our middleware could easily utilize proven technologies such as Content Delivery Networks (CND), Cloudlets or micro data centers. One way of accomplishing this is to deliver a download script to a set of Edge devices and push an image to a Cloudlet, residing in the proximity (single-hop) of these device. The *ProvisioningDeamon* could then use the poll-based approach to obtain the image.

7.4 Evaluation

7.4.1 Middleware performance

In the following experiments we show two main performance aspect of our provisioning middleware: support for: i) *scalable execution of the provisioning workflows* (hundreds of

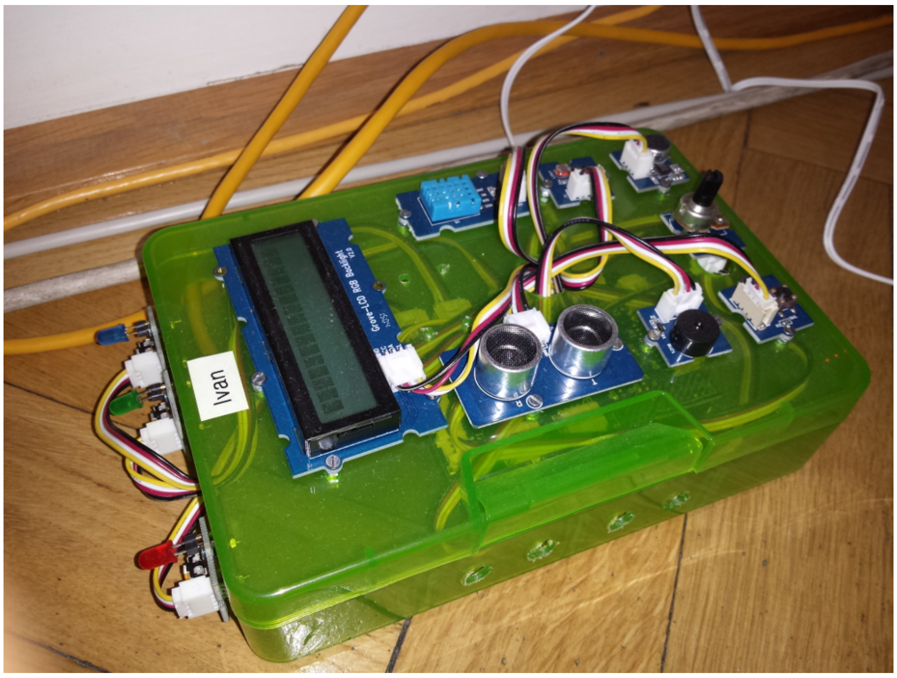


Figure 7.7: An example of our gateways for Building Management Systems.

Edge devices) and at the same time ii) *middleware suitability for constrained devices* in terms of resource consumption, i.e., its memory and CPU usage.

Applications

In the experiments we used two real-life applications from our Building Management System, described in Chapter 2. For our experiments, it is important to note that the first application (SAPP) is written in Sedona [142] and it size is approximately 120Kb, including the SVM and the application (.sab, .sax, .scode and Kits files). The second application (JAPP) is JVM-based (compact profile2) and its size including all binaries, libraries and the JVM is around 14Mb.

Additionally, for the experiments we have developed a SDG prototype, based on BusyBox, which is a very light-weight Linux user land. The SDG prototype is specifically build for Docker's libcontainer virtualization environment and is approximately 1.4Mb in disk size (without applications).

Experiment setup

In order to evaluate middleware performance regarding resource usage, we built 15 physical gateways (cf. Figure 7.7) and installed them throughout our department. The getaways are based on Raspberry Pi 2, with ARMv7 CUP and 1Gb of RAM. They run Raspbian Linux 8 (based on Debian "Jessi") on Linux Kernel 4.1.

In order to evaluate how our middleware behaves in a large-scale setup, we created a virtualized IoT cloud testbed based on CoreOS [35]. In our testbed we use Docker containers to mimic physical gateways in the cloud. These containers are based on a snapshot of a real-world gateway, developed by our industry partners. For the experiments, we deployed a CoreOS cluster on our local OpenStack cloud. The cluster consists of 16 CoreOS 444.4.0 VMs (with 4 VCPUs and 7GB of RAM), each running approximately 250 Docker containers. The Provisioning Controller and the Repositories are also deployed in our Cloud on 3 Ubuntu 14.04 VMs (with 2VCPUs and 3GB of RAM).

Finally, since the physical gateways are attached to our department network, in order to connect them to the cloud network (but avoid potential security risks), we have created a network overlay based on Wave routers [150].

Experiments

Middleware resource consumption at the Edge. Initially, we show the performance of middleware most important components that continuously run in edge devices, namely the *ProvisioningDeamon* and the VirtualBuffersDeamon. The *MonitoringAgent* is not considered in our experiments, since it only periodically executes to create device-state snapshots, thus it does not have statistically significant impact on the performance. We also do not discuss SDGs resource consumption, since it is largely application dependent, but also depends on the underlying virtualization choices. However, it is important to mention that the runtime overhead of middleware components running in the SDGs is almost negligible, since it is less than $1Mb$. The main goal of the following experiments is to demonstrate the validity of our approach w.r.t. resource-constrained devices, since

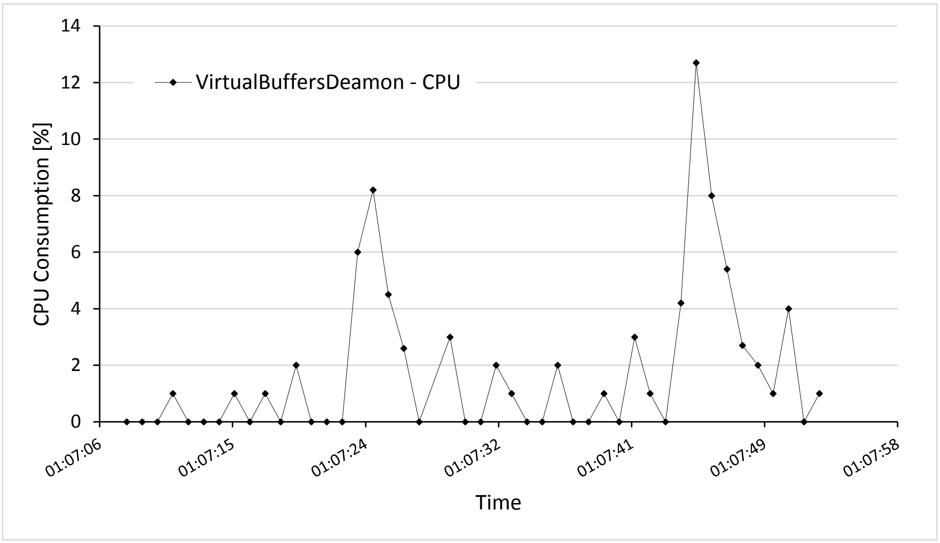


Figure 7.8: CPU consumption of the VirtualBuffersDeamon.

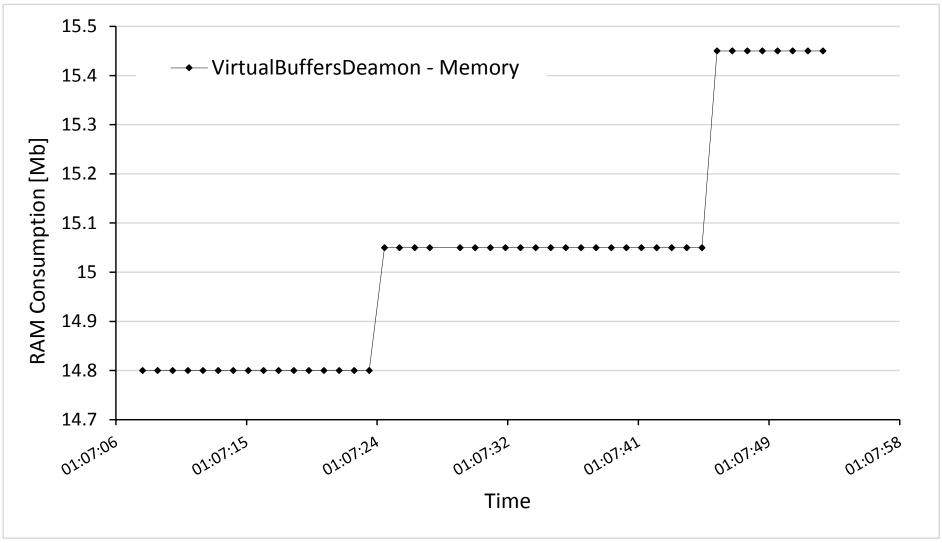


Figure 7.9: Memory consumption of the VirtualBuffersDeamon.

we do not claim that it outperforms related approaches, which provide functionality that partially overlaps with our middleware.

Figure 7.8 and Figure 7.9 respectively show the CPU and memory usage of the VirtualBuffersDeamon, over a period of time. There are several important things to notice here. When there are no SDGs (applications) running in the gateway the deamon is mainly idle, i.e., it only periodically polls the underlying drivers for device status and on average its CPU consumption is less than 2%. This can be observed in Figure 7.8, before the first peak. The two peaks represent SDG deployments for the two applications. The first peak happens when the Sedona-based application is deployed and the second peak signals the deployment of Java-based application. Since SAPP requires smaller number of sensors than JAPP, the deamon needs to allocate and configure less virtual buffers, thus the difference in the two peaks. However, in both cases the maximum CPU usage of the deamon is below 14% and it lasts only a few seconds. For the same scenario we have measured the deamon's memory usage. Figure 7.9, shows the total memory of deamon's JVM process (with heap memory, Perm Size and stack). Initially, we notice that in the idle state the deamon consumes little bit under $15Mb$ of RAM (the initial heap size is configured to a minimum of $1Mb$), what can be considered a low memory footprint. We also observe that memory consumption behaves in a similar manner to CPU consumption. This is represented by the two distinct jumps in memory usage (cf. Figure 7.9). The increase in memory usage is due to newly allocated virtual buffers, adapters (heap) and SDGConnections (stack). The reason for the difference being the same as above. Finally, we notice a monotonic growth of memory usage, the reason for this is that the Deamon does not trigger garbage collection, since both SDGs are ruining and using the buffers, however after an application is stopped the deamon releases its buffers. Therefore, the performance of the VirtualBuffersDeamon can be seen as suitable for resource-constrained devices.

Figure 7.10 and Figure 7.11 show the CPU and memory usage of the ProvisioningDeamon (and the used Connector for the underlying virtualization solution). In this case we only consider infrastructure-level provisioning requests, i.e., configuring and starting SDGs, since only this type of requests are explicitly handled by the ProvisioningDeamon. In Figure 7.10, we notice that in general our provisioning deamon utilizes the CPU resources scarcely, namely its CPU usage is mostly around 1%. This is due to the fact that most of the time the deamon idle, it only periodically checks for new requests from the Provisioning Controller and sends a hart bit. The dramatic spikes in CPU usage happen only during the SDG deployment (we launched 4 SDGs on the gateway during the experiment), since this includes expensive network and computation operations, i.e., downloading SDG prototype, configuring it and starting it. However, the later two operations are performed by the Connectors which execute the commands and quickly terminate. Figure 7.11 shows the memory usage of the provisioning deamon for the same experiment. One can notice that during the experiment the memory usage of the provisioning deamon was always below $30Mb$ and more importantly shortly after an SDG is started the deamon releases the unused (Connector's) memory. Therefore, middleware Edge components in total require under $45Mb$ of memory and consume around 2% of CPU on average. We believe that this is reasonable resource utilization suitable for resource-constrained devices.

Scalable execution of provisioning workflows. The reason we put an emphasis on the scalability of our middleware is that it is one of the key precondition for consistent realization of provisioning workflows across a large resource pool. For example, if the execution of provisioning workflows were to scale exponentially with the size of the

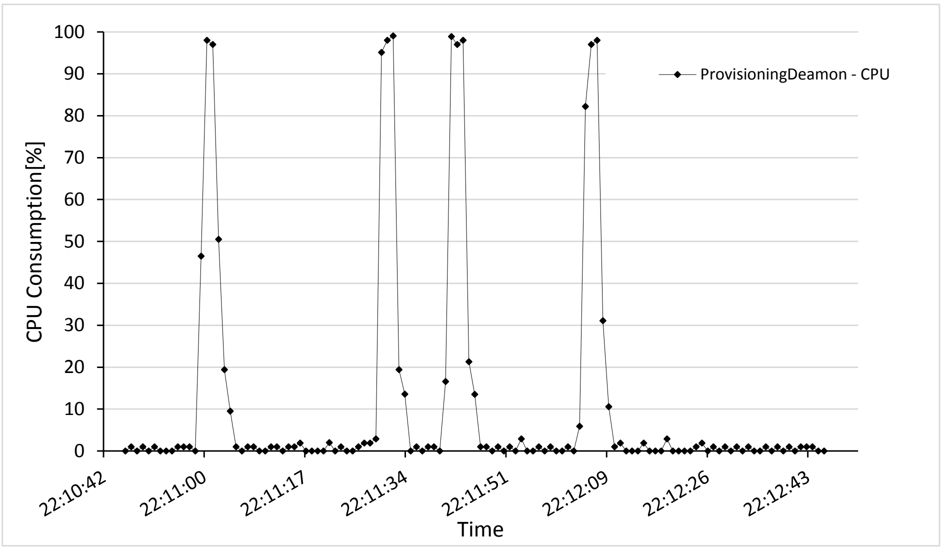


Figure 7.10: CPU consumption of the ProvisioningDeamon.

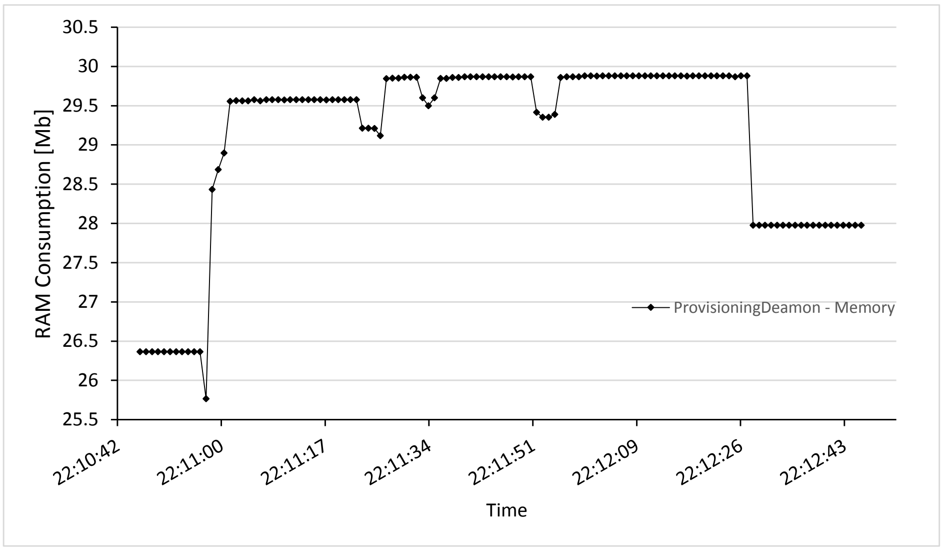
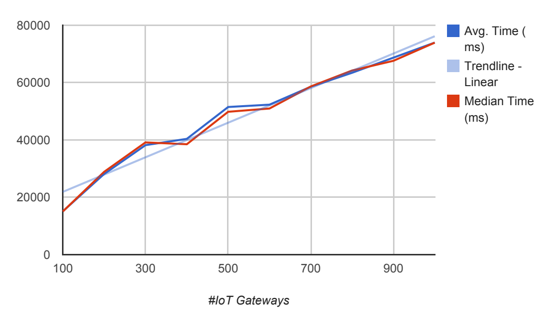


Figure 7.11: Memory consumption of the ProvisioningDeamon.

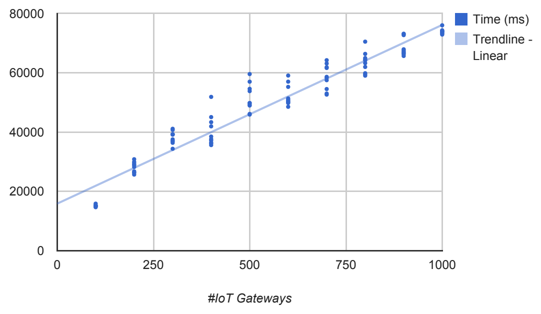
resource pool, theoretically it would take infinitely long to have a consistent infrastructure baseline for the the whole system, given a sufficiently large resource pool.

The experiment presented in Figure 7.12 and Figure 7.13 show execution times (averaged results of 30 repetitions) of the JAPP and SAPP provisioning workflows. In the experiments we have used up to 1000 virtual gateways for the JAPP and up to 4000 gateways for SAPP. This corresponds to a large building management system containing dozens of big buildings (each with more than 10 floors). As a reference, the diagram also shows a plot of a trend line, which turns out to be a $nlog(n) + c$ function, extrapolated from individual experiment runs. Figure 7.12a depicts the provisioning time i.e. execution time of the provisioning workflow for the JAPP application. At 300 gateways we see that the initial overhead of the pushing approach is compensated and therefore the execution time decreases a little bit. From 400 to 500 gateways, the middleware reaches its maximal load. After the deployment size reaches 500, the middleware or more precisely the cloud-based controller scales out and load balancer starts distributing the workload to the newly deployed microservices, i.e. the SDGManager, the AritfactsManager and the DeploymnetHandler. The corresponding scatter plot, depicted in Figure 7.12b, unveils that the deviations of data points are relatively small, thus on average the provisioning execution time scales almost linearly ($nlog(n)$) up to 1000 Edge devices in this experiment.

Figure 7.13 shows the overall execution time of the SAPP provisioning workfolow for different deployments (number of gateways) by using simple push-based approach. In Figure 7.13a we notice that due to the deployment scale the overall execution time got slower compared to the first experiment. This increase in the number of virtualized Edge devices, generates a lot of traffic for the underlying network infrastructure that causes



(a) Provisioning JAPP.

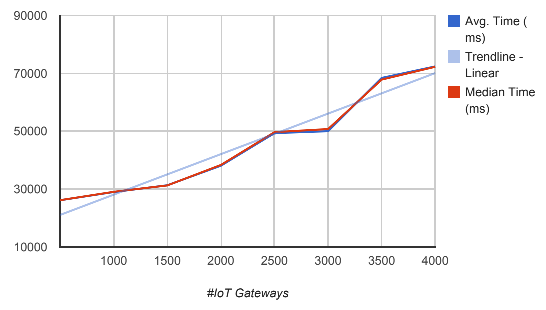


(b) Provisioning JAPP - Scatter Plot.

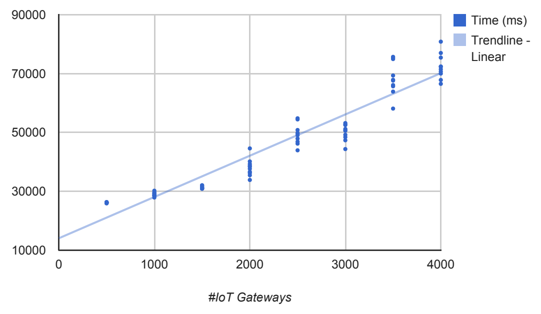
Figure 7.12: Average execution time of provisioning workflow for JAPP application.

slower response times and therefore the execution time of the provisioning workflow takes noticeably more time. For this scenario we changed the load balancing strategy to allow up to 2500 gateways before scaling out. We clearly see that up to 2500 gateways, the execution time increases almost linearly. When reaching 3000 deployments, the execution time rises again, but once more starts to flatten at 4000. When looking at the scatter plot depicted in Figure 7.13b we see that at the beginning of the experiments the deviation among data points is very small and gets bigger with increasing number of IoT gateways. Nevertheless, we clearly see that our framework deals well with this rather large scenario and once again provides almost linear scalability.

Generally, we notice that the middleware mechanisms for workflow execution (Section 7.3.1) scale within $O(nlog(n))$ for relatively large number of Edge devices, which can be considered a satisfactory result. We also notice that computational overheads of the provisioning agents and deamons have no statistically significant impact on the results, since they are distributed among the underlying devices. Finally, the provisioning mechanism behaves in a similar fashion for both application. The reason for this is that all gateways are in the same network, what can be seen as an equivalent to provisioning



(a) Provisioning SAPP.



(b) Provisioning SAPP - Scatter Plot.

Figure 7.13: Average execution time of provisioning workflow for SAPP application.

a complex of collocated buildings.

7.5 Conclusion

In this chapter, we introduced a provisioning middleware that enables developing generic, multi-level provisioning workflows and supports automated and scalable execution of such workflows in IoT Cloud systems. We showed how our middleware supports on-demand, self-service resource consumption by providing flexible provisioning models and support for uniform, logically centralized provisioning of Edge devices, application artifacts and their configuration models. We introduced provisioning support for software-defined gateways to enable application-specific customization of Edge devices through well-defined APIs, while preserving the benefits of proven virtualization mechanism. The initial results of our experiments are promising, since they showed that our middleware enables scalable execution of provisioning workflows across relatively large IoT cloud resource pool and at the same time its overhead in terms of resource consumption is suitable for resource-constrained devices. Additionally, we discussed possible optimizations of the provisioning models as a direct consequence of the middleware architecture. In this regard, the main advantage of middleware's architecture is reflected in its support for flexible and customizable delivery, deployment and placement models for SDGs and application artifacts. For example, it was discussed how our middleware can be configured to optimize the provisioning process by utilizing proven technologies such as CDNs.

Our middleware lays a cornerstone towards realizing our vision of utility-based provisioning of IoT Cloud systems. However, some challenges still remain to realize the utility-based provisioning paradigm. As part of our future work, we plan to address the current limitations of our middleware and the remaining challenges listed in Figure 6.1, by extending our middleware in several directions to: i) Address the mobility aspects of the Edge devices, especially focusing on the dependability issues related to the device mobility and mobility of software components, i.e., runtime migration of SDGs; ii) Support smarter resource allocation, i.e., optimize placement of SDG and applications on Edge devices to include support for dynamic infrastructure properties; iii) Provide more dynamic and finer-grained resource monitoring in order to support pay-as-you-go model; iv) Finally, we plan to extend the current prototype to enable elasticity aspects for IoT Cloud systems, most notably to support elastic, on-demand scaling of the SDGs.

Part III

Governing IoT Cloud Systems

Preface

The ongoing convergence of cloud computing and the IoT gives rise to the proliferation of diverse, large-scale IoT Cloud systems that offer large pools of elastic resources, which need to be operated and governed through their entire lifecycle. Moreover, wide and ever-stronger growing application area of IoT Cloud systems, e.g., in the context of smart cities, has lead to stronger interplay and entanglement among variety of diverse stakeholders, with different objectives, interests and backgrounds. As a consequence, IoT Cloud systems are becoming an integral part of many existing business models and a key enabler for new business opportunities. This calls for a systematic and structured approach to IoT Cloud governance. Unfortunately, vast majority of contemporary approaches dealing with IoT Cloud governance draw a hard line between high-level governance objectives (that mainly concern business stakeholders) and operations processes. The latter concern technical stakeholders such as operations managers that need to implement concrete operations processes, conforming to or enforcing the high-level governance objectives. Therefore, at the moment there is a wide gap between the main stakeholders involved in governing IoT Cloud systems, increasing the risk of lost requirements or causing over-regulated systems, potentially incurring higher operation costs or limiting business opportunities.

In this part of the thesis we aim to respond to the third main research question: "Which models, techniques and tools are required to achieve structured and systematic IoT Cloud governance?". To this end, in Chapter 8, we introduce GovOps – a novel methodology and framework for governing IoT Cloud systems. The main incentive for introducing GovOps is to bring business stakeholders and operations managers closer together and make a step forward in bridging the gap between governance objectives (e.g., standards and regulations) and operations processes. GovOps introduces a novel methodology, governance model and roles, in order to enable seamless integration and alignment of high-level governance objectives and strategies with executable operations processes from early designing stages. Furthermore, Chapter 8 also introduces a runtime framework, which is a reference GovOps implementation, and its main purpose is to support operations managers in implementing and executing GovOps processes in large-scale IoT Cloud systems, without worrying about scale, geographical distribution and dynamicity of such systems. Finally, in Chapter 9, we introduce an uncertainty extension for GovOps. It defines a declarative policy language and a runtime, which enable development of uncertainty- and elasticity-aware GovOps processes, in order to support operations managers to mitigate uncertainties intrinsic to IoT Cloud governance strategies that are mainly caused by the novel interactions of Edge devices, network elements, Cloud resources and humans.

CHAPTER 8

GovOps: A Methodology and a Runtime Framework for Governance in Large-scale IoT Cloud Systems

Wide and ever-stronger growing application area of IoT Cloud systems, e.g., in the context of smart cities, has lead to stronger interplay and entanglement among variety of diverse stakeholders (both business and technical). Various domains, such as smart building and vehicle management, increasingly rely on IoT Cloud resources and capabilities to optimize their key business tasks, improve efficiency of processes and quality of life. As a consequence, IoT Cloud systems are becoming an integral part of many existing business models and a key enabler for new business opportunities and cross-domain applications. Consequently, governance issues such as security, safety, legal boundaries, compliance, and data privacy concerns are being addressed ever-stronger [44, 51, 152], mainly due to their potential impact on the variety of involved stakeholders. However, such approaches are mostly intended for high-level business stakeholders, neglecting support, in terms of tools and frameworks, to realize governance strategies in large-scale, geographically distributed IoT Cloud systems. Approaching IoT Cloud from the operations management perspective, different approaches have been introduced, e.g. [160, 134, 29, 139]. For example, such approaches deal with IoT Cloud infrastructure virtualization and its management, enabling utilization of IoT Cloud computation resources and operating their storage resources. However, most of these approaches do not explicitly consider high-level governance objectives such as legal issues and compliance. This increases the risk of lost requirements or causes over-regulated systems, potentially increasing costs and limiting business opportunities. Therefore, current approaches to IoT governance usually addresses the *Internet* part of the IoT, e.g, in the context of the Future Internet services[1], while operations processes mostly deal with *Things* (e.g, in [32]) as additional resources that need to be operated. The governance objectives (law, compliance, etc.) are not easily mapped to operations processes (e.g., querying sensory data streams or adding/removing devices), so that contemporary models, which assume that business stakeholders define governance objectives, and operations managers implement and enforce them, are hardly feasible in IoT Cloud systems. In practice, bridging this gap between governance and operations management of IoT cloud systems poses a significant challenge for the involved stakeholders. What is more, even with perfectly aligned governance objectives, designing and realizing operational governance processes [133, 70], posses a significant challenge. Traditional operational governance approaches are hardly applicable for IoT Cloud systems, mainly due to the large number of involved stakeholders, novel requirements for shared resources and capabilities, dynamicity, geographical distribution, and the sheer scale of such systems. Supporting tools and mechanisms for runtime operational governance of IoT Cloud systems remain largely undeveloped, thus placing much of the burden on operations managers to perform operational governance processes.

This calls for a systematic approach to govern IoT Cloud resources and applications throughout their entire lifecycle. In this chapter we introduce a GovOps methodology, conceptual model and framework to effectively manage runtime governance in software-defined IoT Cloud systems. The main aim of GovOps is to bridge the gap between high-level governance objectives (e.g., costs, legal issues or compliance) and underlying operations processes that enforce such objectives. Therefore, GovOps mostly focuses to provide conceptual and framework support for designing and executing *operational governance processes*, which represent a subset of the overall IoT Cloud governance and incorporate relevant aspects of both high-level governance strategies and underlying operations management. We present a GovOps reference model that defines required roles, concepts, and techniques, to support seamless mapping between governance and operations, and to facilitate realizing IoT Cloud governance processes. We introduce the rtGovOps, which is a runtime framework for dynamic operational governance of large-scale IoT Cloud systems. Its main objective is to support GovOps managers in implementing and executing operational governance processes in IoT Cloud systems, without worrying about scale, geographical distribution, dynamicity, and other characteristics inherent to such systems that currently hinder operational governance in practice. The rtGovOps framework provides runtime mechanisms and enabling techniques to reduce the complexity of IoT Cloud operational governance, thus enabling the GovOps managers to perform custom operational governance processes more efficiently in large-scale IoT Cloud systems.

The remainder of this chpater is structured as follows: Section 8.1 presents our motivating scenarios. In Section 8.2, we present the GovOps methodological approach to governance and operations management in IoT Cloud systems; Section 8.3 outlines the GovOp reference model and design process for GovOps strategies; Section 8.4 outlines main concepts and the design of the rtGovOps framework; In Section 8.5, we explain

[1]http://ec.europa.eu/digital-agenda/en/internet-things

main runtime mechanisms of rtGovOps; Section 8.6 describes the experimental results and outlines the prototype implementation; Finally, Section 8.7 provides final remarks and concludes the chapter.

8.1 Motivation

Let us consider our real-life FMS scenario, described in Chapter 2, form a perspective of the involved stakeholders and governance requirements. Next, we briefly discuss some of the involved stakeholders, mainly focusing on their requirements and issues related to governing FMS applications and underlying IoT Cloud resources.

As we have mentioned in Chapter 2 FMS is responsible for managing electric vehicles deployed worldwide, e.g., on different golf courses. These vehicles communicate with the Cloud via 3G or Wi-Fi networks to exchange telematic and diagnostic data. On the Cloud, FMS provides different applications and services to manage this data. Examples of such services include realtime vehicle status, remote diagnostics, and remote control. The FMS is currently used by the following three types of stakeholders: vehicle manufacturers, distributors, and golf course managers. These stakeholders have different business models. For example, when a manufacturer only leases vehicles to customers, they are interested in the status and upkeep of the complete fleet, will perform regular maintenance, as well as monitor crashes and battery health. Golf course managers are mostly interested in vehicle security to prevent misuse and ensure safety on the golf course (e.g., using geofencing features). In general, the stakeholders rely on the FMS and its services to optimize their respective business tasks. Figure 8.1 gives a high-level overview of the FMS deployment and infrastructure. We notice that FMS runs atop a nontrivial IoT Cloud infrastructure that includes a variety of IoT Cloud resources. For our discussion, the two most relevant types of IoT Cloud resources are on-board physical gateways (G) and cloud virtual gateways (VG). Most of the vehicles are equipped with on-board gateways that are capable to host lightweight services such as geofencing or local diagnostics services. For legacy cars that are not equipped with such gateways, a device acting as a CAN-IP bridge is used (e.g, Teltonika FM5300[2]). In this case FMS hosts virtual gateways on the cloud that execute the aforementioned services on behalf of the vehicles.

We notice that the FMS is a large-scale system that manages thousands of vehicles and relies on diverse cloud communication protocols. Further, the FMS depends on IoT Cloud resources that are geographically distributed on different golf courses around the globe. Jurisdiction over these resources can change over time, e.g., when a vehicle is handed over from the distributor to a golf course manager. In addition, these resources are usually constrained. This is why the FMS heavily relies on cloud services, e.g., for computationally intensive data processing, fault-tolerance or to reliably store historical readings of vehicle data. While the cloud offers the illusion of unlimited resources, systems of such scale as FMS can incur very high costs in practice (e.g., of computation or networking). Finally, due to the large number of involved stakeholders, the FMS needs

[2]http://www.teltonika.lt/en/pages/view/?id=1024

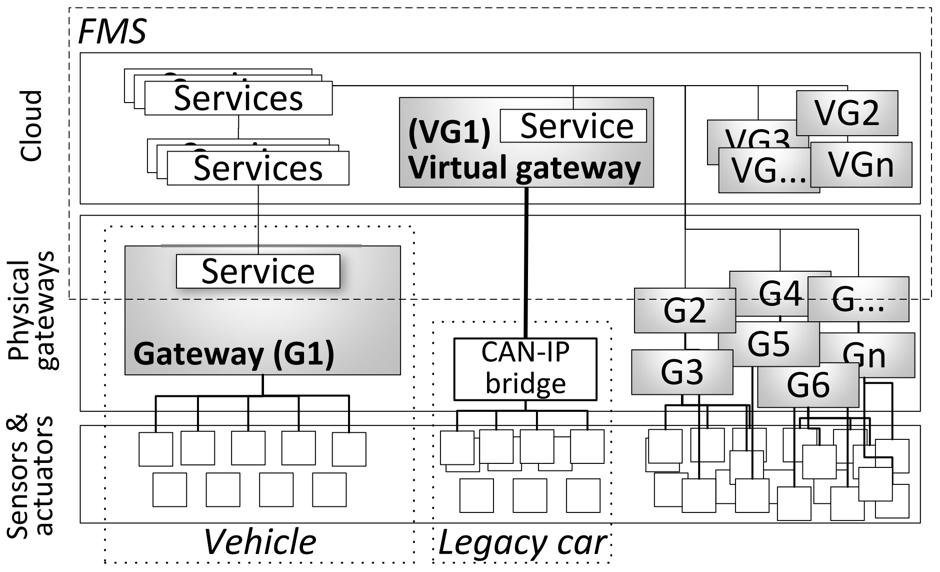


Figure 8.1: Overview of FMS infrastructure.

to enable runtime customizations of infrastructure resources in order to exactly meet stakeholder requirements and allow for operation within specified compliance and legal boundaries. Therefore, the IoT Cloud resources and applications need to be managed and governed throughout their entire lifecycle. In our approach, this is captured and modeled as *operational governance processes*.

Example operational governance processes

Subsequently, we highlight some basic operational governance processes in FMS that are facilitated through our framework:

- Typically, the FMS polls diagnostic data from vehicles (e.g., with CoAP). However, a golf course manager could design an operational governance process that is triggered in specific situations such as in case of emergency. Such process could, for example, increase the update rate of the vehicle sensors and change the communication protocol to MQTT in order to satisfy a high-level governance objective, e.g., company's compliance policy to handle emergency updates in (near) real-time.
- To increase fault-tolerance and guarantee history preservation of vehicle data (e.g., due to governance objectives related to legal requirements), a distributor could decide to spin up additional virtual gateways in a different availability zone.
- After multiple complaints about problems with vehicles of type X, a manufacturer would need to add additional monitoring features to all vehicles of type X to perform more detailed inspections.

This is by no means a comprehensive list of operational governance processes in the FMS. However, due to dynamicity, heterogeneity, geographical distribution, and the large scale of IoT Cloud systems, traditional approaches to realize even basic operational governance processes are hardly feasible in practice. This is mostly because such approaches implicitly make assumptions such as physical on-site presence, manually logging into gateways, understanding device specifics, etc., which are difficult, if not impossible, to meet in IoT Cloud systems. Therefore, due to a lack of systematic approaches for operational governance in IoT Cloud systems, operations managers currently have to rely on ad-hoc solutions to deal with the characteristics and complexity of IoT Cloud systems when performing operational governance processes.

8.2 GovOps – A Novel Methodology for Governance and Operations in IoT Cloud Systems

The main objective of our GovOps approach (Governance and Operations) is twofold. On the one side it aims to enable seamless integration of high-level governance objectives and strategies with concrete operations processes. On the other side, it enables performing operational governance processes for IoT Cloud systems in such manner they are feasible in practice. In general, governance objectives and operations processes define and enforce system invariants that are ideally satisfied at any point in time. The objectives and states are usually associated with rules, conditions and properties, that should hold during system's runtime. In reality, due to the dynamicity and the scale of IoT Cloud systems, this is difficult, if not impossible to achieve, without constantly reinforcing the objectives and desired system states, as well as adapting the processes to the ever changing requirements of the multitude of the involved stakeholders.

Figure 8.2 illustrates how GovOps relates to IoT Cloud governance and operations. It depicts the main idea of GovOps to bring governance and operations closer together and bridge the gap between governance objectives and operations processes, by incorporating the main aspects of both IoT Cloud governance and operations management. To this end, we define *GovOps principles and design process* of GovOps strategies (Section 8.3) that support determining what can and needs to be governed, based on the current functionality and features of an IoT Cloud system, and that allow for aligning such system capabilities with regulations and standards. Additionally, we introduce a novel role, *GovOps manager* (Section 8.2.3) responsible to guide and manage designing GovOps

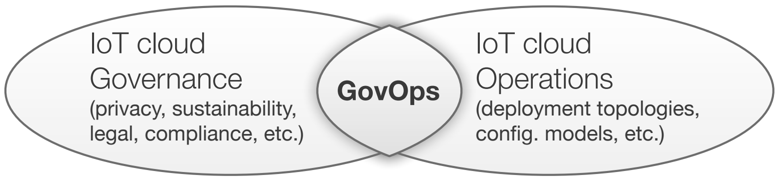

IoT cloud
Governance
(privacy, sustainability, legal, compliance, etc.)
GovOps
IoT cloud
Operations
(deployment topologies, config. models, etc.)

Figure 8.2: GovOps in relation to IoT Cloud governance and operations.

strategies, because in practice it is very difficult, risky, and ultimately very costly to adhere to the traditional organizational silos, separating business stakeholders from operational managers. Therefore, GovOps integrates business rules and compliance constraints with operations capacities and best-practices, from early stages of designing governance strategies in order to counteract system over-regulation and lost governance requirements [51].

It is worth noting that GovOps does not attempt to defined a general methodology for IoT Cloud governance. There are many approaches (Chapter 10), which define governance models and accountability frameworks for managing governance objectives and coordinating decision making processes, and that can usually be applied within GovOps without substantial modifications.

8.2.1 Governance Aspects

From our case studies, we have identified various business stakeholders such as building residents, building managers, governments, vehicle manufacturers and golf course managers. Typically, these stakeholders are interested in energy efficient and greener buildings, sustainability of building assets, legal and privacy issues regarding sensory data, compliance (e.g, regulatory or social), health of the fleet, security and safety issues related to the environments under their jurisdiction.

Depending on the concrete (sub)system and the involved stakeholders, governance objectives are realized via different governance strategies. Generally, we identify the following governance aspects: i) *environment-centric*, ii) *data-centric* and iii) *infrastructure-centric governance*.

Environment-centric governance deals with issues of overlapping jurisdictions in IoT Cloud managed environments. For example, in our BAS, we have residents, building managers and the government that can provide governance objectives, which directly or indirectly affect an environment, e.g., a residential apartment. In this context, we need to articulate multiple governance objectives related to comfort of living, energy efficiency, safety, health and sustainability.

Data-centric governance mostly deals with implementing the governance strategies related to the privacy, quality, and provenance of sensory data. Examples include addressing legal issues, compliance, and user preferences w.r.t. such data.

Infrastructure-centric governance addresses issues about designing, installing, and deploying IoT Cloud infrastructure. This mostly affects the early stages of introducing a IoT Cloud system and involves feasibility studies, cost analysis, and risk management. For example, it supports deciding between introducing new hardware or visualizing the IoT Cloud infrastructure.

8.2.2 Operations Management Aspects

Operations managers implement various processes to manage BAS and FMS at runtime. Generally, we distinguish following operational governance aspects: i) *configuration-centric*, ii) *topology-centric*, and iii) *stream-centric governance*.

Configuration-centric governance includes dynamic changes to the configuration models of the software-defined IoT Cloud systems at runtime. Example processes include: a) enabling/disabling an IoT resource or capability (e.g, start/stop a unit), b) changing an IoT capability at runtime (e.g, communication protocol), and c) configuring an IoT resource (e.g, setting sensors poll rate).

Topology-centric governance addresses structural changes that can be performed on software-defined IoT systems at runtime. For example, a) Pushing processing logic from the application space towards the edge of the infrastructure; b) Introducing a second gateway and an elastic load balancer to optimize resource utilization, e.g., provide more bandwidth; c) Replicating a gateway, e.g., for fault-tolerance or data source history preservation.

Stream-centric governance addresses runtime operation of sensory data streams and continuous queries, e.g., to perform custom filtering, aggregation, and querying of the available data-streams. For example, to perform local filtering the processing logic is executed on physical gateways, while complex queries, spanning multiple data streams are usually executed on VGWs. Therefore, operations managers perform processes like: a) Placing custom filters (e.g., near the data source to reduce network traffic); b) Allocating queries to virtual gateways; and c) Splitting streams, i.e., sending events to multiple virtua gateways.

8.2.3 Integrating Governance Objectives with Operations Processes

The examples presented in Section 8.2.1 and Section 8.2.2 are by no means a comprehensive list of IoT Cloud governance processes. However, due to dynamicity, heterogeneity, geographical distribution and the sheer scale of IoT Cloud, traditional approaches to realize these processes are hardly feasible in practice. This is mostly because such approaches implicitly make assumptions such as physical on-site presence, manually logging into gateways, understanding device specifics, etc., which are difficult, if not impossible, to meet in IoT Cloud systems. Therefore, due to a lack of a systematic approach for operational governance in IoT systems, currently operations managers have to rely on ad hoc solutions to deal with the characteristics and complexity of IoT Cloud systems, when performing governance processes.

Table 8.1 lists examples of governance objectives and according operations management processes to enforce these objectives. The first example comes from the FMS, since many of the golf courses are situated in countries with specific data regulations, e.g., the US or Australia. In order to enable monitoring of the whole fleet (as required by the manufacturer) the operations managers needs to understand the legal boundaries

	Governance objectives	**Operations processes**
1.	Fulfill legal requirements w.r.t. sensory data in country X. Guaranty history preservation.	Spin-up an aggregator gateway. Replicate VGW, e.g., across different availability zones.
2.	Reduce GHG emission. User preferences regarding living comfort. Consider health regulations.	Provide a configuration directives for a IoT Cloud resource (e.g, HVAC).
3.	Data quality compliance regarding location tracking services.	Choose among available services, e.g., GPS vs. GNSS (Global Navigation Satellite System) platform.

Table 8.1: Example governance objectives and operations processes.

regarding data privacy. For example, in Australia, the OAIC[3] has issued a 32 page guidance as to what *"reasonable steps to protect personal information"* might include, that in practice need to be interpreted by operations managers. The second example contains potentially conflicting objectives supplied by stakeholders, e.g., building manager, end user, and the government, leaving it to the operations team to solve the conflicts, at runtime. The third example, hints that GNSS is usually better-suited to simultaneously work in both northern and southern high latitudes. Even for these basic processes, an operations team faces numerous difficulties, since in practice there is no one-size-fits-all solution to map governance objectives to operations processes.

Therefore, GovOps proposes a novel role, *GovOps manager*, as a dedicated stakeholder responsible to bridge the gap between governance strategies and operations processes in IoT Cloud systems. The main rationale behind introducing a GovOps manager is that in practice designing governance strategies needs to involve operations knowledge about the technical features of the system, e.g., physical location of devices, configuration and placement of queries, and component replication strategies. Reciprocally, defining systems configurations and deployment topologies should incorporate standards, compliance, and legal boundaries at early stages of designing operations processes. To achieve this, the GovOps manager is positioned in the middle, in the sense that they continuously interact with both business stakeholders (to identify high-level governance issues) and operations team (to determine operations capacities).

The main task of a GovOps manager is to determine suitable tradeoffs between satisfying the governance objectives and the system's capabilities, as well as to continuously analyze and refine how high-level objectives are articulated through operations processes. In this context, a key success factor is to ensure effective and continuous communication among the involved parties during the decision making process, facilitating i) openness,

[3]Office of the Australian Information Commissioner(OAIC), Australian privacy regulator.

ii) collaboration, iii) establishment of a dedicated GovOps communication channel, along with iv) early adoption of standards and regulations. This ensures that no critical governance requirements are lost and counteracts over-regulation of IoT Cloud systems.

8.2.4 Main principles of GovOps in IoT Cloud systems

Generally, GovOps strategies manipulate the state of IoT Cloud resources at runtime while considering governance objectives and regulations. Therefore, they can be seen as a sequence of runtime state transitions from the current state to some desired, target state (e.g., that satisfies some non-functional properties, enforces compliance or exactly meets custom functional requirements). The core idea of GovOps is to provide abstractions that shield stakeholders from the complexities of underlying system and diversities of various legal and compliance issues, allowing them to focus on integrating governance objectives with practically feasible operations processes. To support performing such processes in IoT Cloud systems, (e.g., listed in Section 8.1), while considering system characteristics (e.g., large-scale, geographical distribution and dynamicity), GovOps relies on concepts that include:

Central point of operation (R1) – Enable conceptually centralized interaction with the software-defined IoT Cloud system to enable a unified view on the system's operations and governance capabilities (available at runtime), without worrying about low-level infrastructure details.

Automation (R2) – Allow for dynamic, on-demand governance of software-defined IoT cloud systems on a large scale and enable governance processes to be easily repeatable, i.e., enforced across the IoT Cloud, without manually logging into individual gateways.

Fine-grained control (R3) – Expose the control functionality of IoT Cloud resources at fine granularity to allow for precise definition of governance processes (to exactly meet requirements) and flexible customization of IoT Cloud system governance capabilities.

Late-bound directives (R4) – Support declarative directives that are bound later during runtime in order to allow for designing generic and flexible operational governance processes.

IoT Cloud resources autonomy (R5) – Provide a higher degree of autonomy to IoT Cloud resources to reduce communication overhead, increase availability (e.g., in case of network partitions), enable local exception and fault handling, support protocol independent interaction, and increase system scalability.

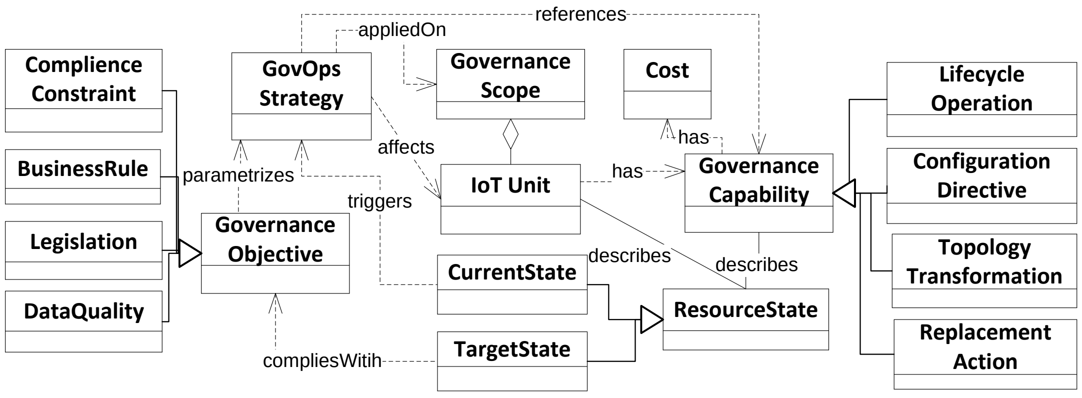


Figure 8.3: Simplified UML diagram of GovOps model for IoT Cloud governance.

8.3 A reference model for GovOps methodology

8.3.1 Overview of GovOps model for IoT Cloud systems

To realize the GovOps approach we need suitable abstractions to describe IoT Cloud resources that allow IoT Cloud infrastructure to be (re)defined after it has been deployed. We show in Chapter 6 how this can be done with *software-defined IoT units*. GovOps model (Figure 8.3) builds on this premise and extends our previous work with fundamental aspects of operational governance processes: i) describing states of deployed IoT resources, ii) providing capabilities to manipulate these states at runtime, and iii) defining governance scopes.

Within our model, the main building blocks of GovOpsStrategies are *Governance-Capabilities*. They represent operations which can be applied on IoT Cloud resources, e.g., query current version of a software, change communication protocol, and spin-up a virtual gateway. These operations manipulate IoT Cloud resources in order to put an IoT Cloud system into a specific (target) state. Governance capabilities are described via well-defined software-defined APIs and they can be dynamically added to the system, e.g, to a VGW. From the technical perspective, they behave like add-ons, in the sense that they extend resources with additional operational functionality. Generally, by adopting the notion of governance capabilities, we allow for processes to be automated to a great extent, but also give a degree of autonomy to IoT Cloud resources.

Since the meaning of a resource state is highly task specific, we do not impose many constraints to define it. Generally, any useful information about an IoT Cloud resource is considered to describe the *ResourceState*, e.g., a configuration model or monitoring data such as CPU load. Technically, there are many frameworks (e.g., Ganglia or Nagios) that can be used to (partly) describe resource states. Also configuration management solutions, such as OpsCode Chef[4], can be used to maintain and inspect configuration

[4]http://opscode.com/chef

states. Finally, design best practices and reference architectures (e.g., AWS Reference Architectures[5]) provide a higher-level description of the desired target states of an IoT Cloud system.

The *GovernanceScope* is an abstract resource, which represents a group of IoT Cloud resources (e.g., gateways) that share some common properties. Therefore, our governance scopes are used to dynamically delimit IoT Cloud resources on which a GovernanceCapability will have an effect. This enables writing the governance strategies in a scalable manner, since the IoT Cloud resources are not individually addressed. It also allows for backwards compatible GovOps strategies, which do not directly depend on the current resource capabilities. This means that we can move a part of the problem, e.g., faults and exceptions handling, inside the governance scope. For example, if a gateway loses a capability the scope simply wont invoke it i.e., the strategy will not fail.

8.3.2 Design process of GovOps strategies

As described in Section 8.2, the GovOps manager is responsible to oversee and guide the GovOps design process and to design concrete GovOps strategies. The design process is structured along three main phases: i) identifying governance objectives and capabilities, ii) formalizing strategy, and iii) executing strategy.

Generally, the initial phase of the design process involves eliciting and formalizing governance objectives and constraints, as well as identifying required *fine-grained* governance capabilities to realize the governance strategy in the underlying IoT Cloud system. GovOps does not make any assumptions or impose constraints on formalizing governance objectives. To support specifying governance objectives the GovOps manager can utilize various governance models and frameworks, such as the 3P [128] or COBIT [63]. However, it requires tight integration of the GovOps manager into the design process and encourages collaboration among the involved stakeholders to clearly determine risks and tradeoffs, in terms of what should and can be governed in the IoT Cloud system, e.g., which capabilities are required to balance building emission regulations and residents temperature preferences. To this end, the GovOps manager gathers available governance capabilities in collaboration with the operations team, identifies missing capabilities, and determines if further action is necessary. Generally, governance capabilities are exposed via well-defined APIs. They can be built-in capabilities exposed by IoT units (e.g., start/stop), obtained from third-parties (e.g., from public repositories or in a market-like fashion), or developed in-house to exactly reflect custom governance objectives. By promoting *collaboration* and early integration of governance objectives with operations capabilities, GovOps reduces the risks of lost requirements and over-regulated systems.

After the required governance capabilities and relevant governance aspects are identified, the GovOps manager relies on the aforementioned concepts and abstractions (Section 8.3.1) to formally define the GovOps strategy and articulate the artifacts defined in the first phase of the design process. Governance capabilities are the main building

[5]http://aws.amazon.com/architecture/

blocks of the GovOps strategies. They are directly referenced in GovOps strategies to specify the concrete steps which need to be enforced on the underlying IoT Cloud resources, e.g., defining a desired communication protocol or disabling a data stream for a specific region. Also in this context, the GovOps reference model does not make assumptions about the implementation of governance strategies, e.g., they can be realized as business processes, policies, applications, or domain specific language. Individual steps, defined in the generic strategy, invoke governance capabilities that put the IoT Cloud resources into desired target state, e.g., which satisfies a set of properties. Subsequently, the generic GovOps strategy needs to be parameterized, based on the concrete constraints and rules defined by the governance objectives. Depending on the strategy implementation these can be realized as process parameters, language constraints (e.g., Object Constraint Language), or application configuration directives. By formalizing the governance strategy, GovOps enables reusability of strategies, promotes consistent implementation of established standards and best practices, and ensures operation within the system's regulatory framework.

The last phase involves identifying the system resources, i.e. the governance scopes that will be affected by the GovOps strategy and executing the strategy in the IoT Cloud system. It is worth mentioning that the scopes are not directly referenced in the GovOps strategies, as the GovOps manager applies the strategies on the resource scopes instead of the actual resources. Introducing scopes at the strategy-level shields the operations team from directly referencing IoT Cloud resources, thus enables designing *declarative, late-bound strategies* in a scalable manner. Furthermore, additional capabilities identified in the previous phase will be acquired and/or provisioned at this point in the underlying IoT Cloud system, whereas unused capabilities will be decommissioned in order to optimize resource consumption.

8.4 rtGovOps – A Runtime Framework for GovOps in Large-scale IoT Cloud Systems

The main aim of our rtGovOps (*runtime GovOps*) framework is to facilitate operational governance processes for software-defined IoT Cloud systems. To this end, rtGovOps provides a set of runtime mechanisms and does most of the "heavy lifting" to support operations managers in implementing and executing operational governance processes in large-scale software-defined IoT Cloud systems, without worrying about scale, geographical distribution, dynamicity, and other characteristics inherent to such systems that currently hinder operational governance in practice. In order to facilitate performing the operational governance processes, while considering the characteristics of the software-defined IoT Cloud systems, the rtGovOps framework follows the set of design principles, introduced in Section 8.2.4. They represent the main requirements, which need to be supported and enforced by our rtGovOps framework.

Figure 8.4 gives a high-level architecture and deployment overview of the rtGovOps framework. Generally, the rtGovOps framework is distributed across the cloud and IoT

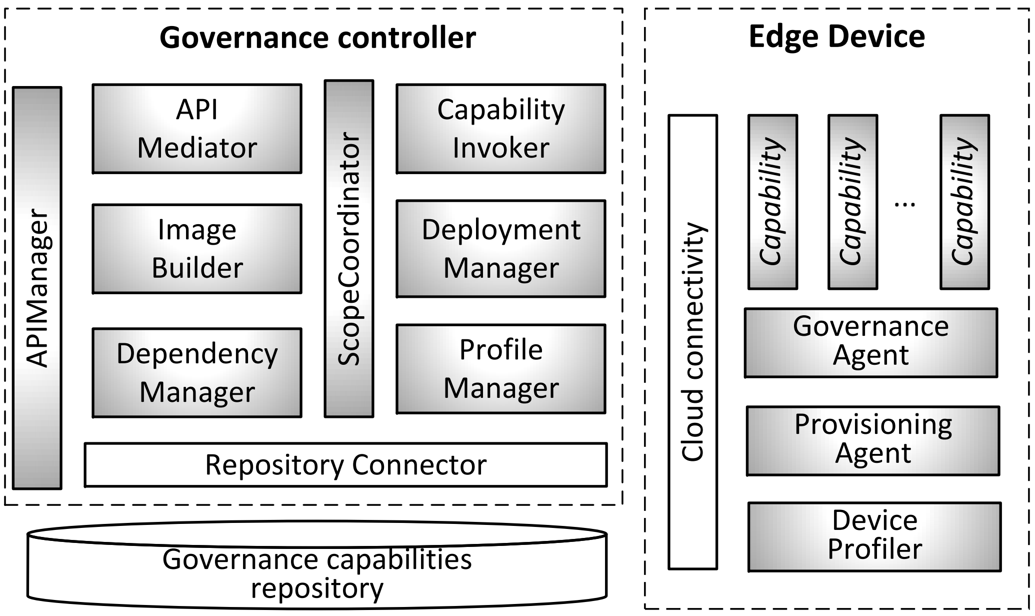


Figure 8.4: Overview of rtGovOps architecture and deployment.

devices. It is designed based on the microservices architecture[6], which among other things enables flexible, evolvable, and fault-tolerant system design, while allowing for flexible management and scaling of individual components. The main components of rtGovOps include: i) the *governance capabilities*, ii) the *governance controller* that runs on the cloud, and iii) the *rtGovOps agents* that run in IoT devices. In the remainder of this section, we will discuss these components in more detail.

8.4.1 Operational governance capabilities

As we described in Section 8.1, operational governance processes govern software-defined IoT units throughout their entire lifecycle. Generally, *Governance capabilities* represent the main building blocks of operational governance processes and they are usually executed in IoT devices. The governance capabilities encapsulate governance operations which can be applied on deployed IoT units, e.g., to query the current version of a service, change a communication protocol, or spin up a virtual gateway. Such capabilities are described via well-defined APIs and are usually provided by domain experts who develop the IoT units. The rtGovOps framework enables such capabilities to be dynamically added to the system (e.g, to gateways), and supports managing their APIs. From a technical perspective, they behave like add-ons, in the sense that they extend the resources with additional operational functionality. Internally, IoT devices host rtGovOps agents that behave like an add-on manager, responsible for installing/enabling, starting/stopping a capability, and managing the APIs they expose. Generally, rtGovOps does not make

[6]http://martinfowler.com/articles/microservices.html

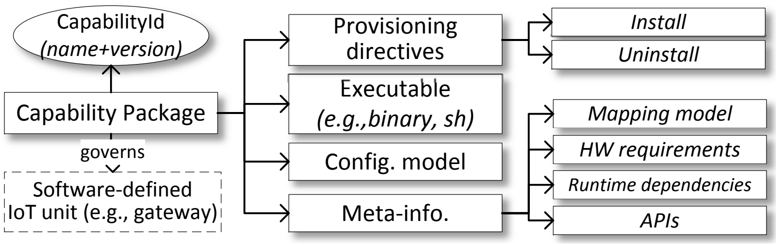


Figure 8.5: Overview of capability package structure.

any assumptions about concrete capability implementations. However, it requires them to be packaged as shown in Figure 8.5. Subsequently, we highlight relevant examples of governance capabilities related to our FMS application.

- *Configuration-specific capabilities* include changes to the configuration models of software-defined IoT Cloud systems at runtime. For example: setting sensor poll rate, changing communication protocol for cloud connectivity, configuring data point unit and type (e.g., temperature in Kelvin as unsigned 10-bit integer), mapping a sensor or CAN bus unit to a device's virtual pin, or activating a low-pass filter for an analog sensory input.
- *Topology-specific capabilities* address structural changes that can be performed on the deployment topologies of software-defined IoT systems. Examples include replicating a virtual gateway to increase fault-tolerance or data source history preservation and push data processing logic from the application space towards the edge of the infrastructure.
- *Stream-specific capabilities* deal with managing the runtime operation of sensory data streams and continuous Complex Event Processing (CEP) queries. Therefore, to enable features like scaling out or stream replaying, operations managers need capabilities such as: filter placement near the data source to reduce network traffic, allocation of queries to gateways, and stream splitting, i.e., sending events to multiple virtual gateways.
- *Monitoring-specific capabilities* deal with adding a general monitoring metric, e.g., CPU load, or providing an implementation of a custom metric to IoT Cloud resources.

For the sake of simplicity, we assume that the capabilities are readily available[7]. In reality, they can be obtained from a central repository, provided by a third-party in a market-like fashion, or custom developed in-house.

As mentioned above, governance capabilities are dynamically added to the IoT Cloud resources. There are several reasons why such behavior is advantageous for operations managers and software-defined IoT Cloud systems. For example, as we usually deal with

[7]We provide example governance capabilities under https://github.com/tuwiendsg/GovOps/

constrained resources, static provisioning of such resources with all available functionality is rarely possible (e.g., factory defaults rarely contain the desired configuration for FMS vehicle gateways). Further, as we have seen in Section 8.1, jurisdiction over resources (in this case FMS vehicles) can change during runtime, e.g., when a vehicle is handed over to a golf course manager. In such cases, because the governing stakeholder changes, it is natural to assume that the requirements regarding operational governance will also change, thus requiring additional or different governance capabilities. As opposed to updating the whole device image at once, we reduce the communication overhead, but also enable changing device functionality without interrupting the system, e.g., to reboot. This provides greater flexibility and enables on-demand governance tasks (e.g, by temporally adding a capability), which are often useful in systems with a high degree of dynamicity. Finally, executing capabilities in the IoT devices improves scalability of the operational governance processes and enables better resource utilization.

8.4.2 Operational governance processes and governance scopes

Operational governance processes represent a subset of the general IoT Cloud governance and deal with operating and governing IoT Cloud resources at runtime. Such processes are usually designed by operations managers in coordination with business stakeholders [104]. The main purpose of such processes is to enable supporting high-level governance objectives such as compliance and legal concerns, which influence system's runtime behavior. To be able to dynamically govern IoT Cloud resources, the operational governance processes rely on the governance capabilities. This means that individual steps of such process usually invoke governance capabilities in order to enforce the behavior of IoT Cloud resources in such manner that it complies with the governance objectives. In this context, our rtGovOps framework provides runtime mechanisms to enable execution of these operational governance processes.

As described in Chapter 6, we use software-defined IoT units to describe IoT Cloud resources. However, these units are not specifically tailored for describing non-functional properties and available meta information about IoT Cloud resources, e.g., location of a vehicle (gateway) or its specific type and model. For this purpose, rtGovOps provides governance scopes. The governance scope is an abstract resource which represents a group of IoT Cloud resources that share some common properties. For example, an operations manager can specify a governance scope to include all the vehicles of type X. The *ScopeCoordinator* (Figure 8.4) provides mechanisms to define and manage the governance scopes. The rtGovOps framework relies on the *ScopeCoordinator* to determine which IoT Cloud resources need to be affected by an operational governance process. Generally, the governance scopes enable implementing the operational governance processes in a scalable and generic manner, since the IoT Cloud resources do not have to be individually referenced within such process.

8.4.3 Governance controller and rtGovOps agents

The *Governance controller* (Figure 8.4) represents a central point of interaction with all available governance capabilities. It provides a mediation layer that enables operations managers to interact with IoT Cloud systems in a conceptually centralized fashion, without worrying about geographical distribution of the underlying system. Internally, the governance controller comprises several microservices, among which the most important include: *DeploymentManager* and *ProfileManager* that are used to support dynamical provisioning of the governance capabilities, as well as *APIManager* and previously mentioned *ScopeCoordinator* that support operational governance processes to communicate with the underlying capabilities. The *APIManger* exposes governance capabilities to operational governance processes via well-defined APIs and handles all API calls from such processes. It is responsible to resolve incoming requests, map them to respective governance capabilities in the IoT devices and deliver results to the calling process. Among other things, this involves discovering capabilities by querying the capabilities repository, and parameterizing capabilities via input arguments or configuration directives.

Since governance capabilities are usually not "pre-installed" in IoT devices, the *DeploymentManager* is responsible to inject capabilities into such devices (e.g., gateways) at runtime. To this end it exposes REST APIs, which are used by the devices to periodically check for updates, as well as by the operational governance processes to push capabilities into the devices. Finally, the *ProfileManager* is responsible to dynamically build and manage device profiles. This involves managing static device meta-information and periodically performing profiling actions in order to obtain runtime snapshots of current device states.

Another essential part of the rtGovOps framework are the *rtGovOps agents*. They include: *ProvisioningAgent*, *GovernanceAgent* and *DeviceProfiler*. These agents are

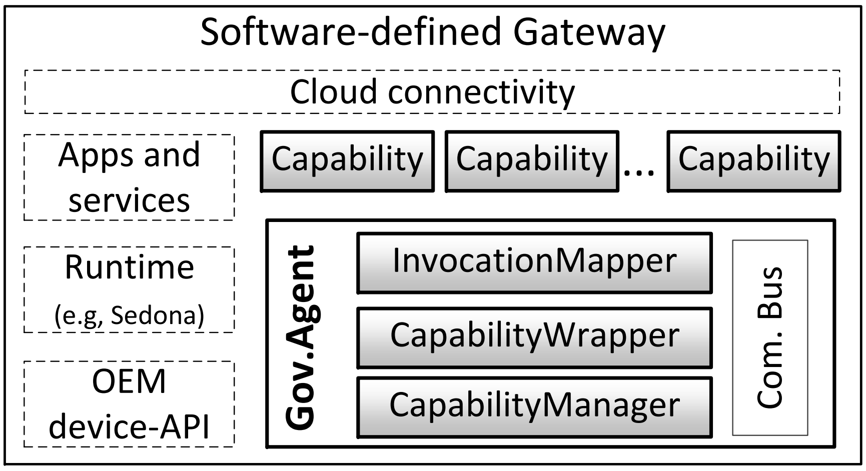


Figure 8.6: Overview of the governance agent architecture.

very light-weight components that run in all IoT Cloud resources that are managed by rtGovOps such as the FMS vehicles. Figure 8.6 shows a high-level overview of the *GovernanceAgent* architecture. It is responsible to manage local governance capabilities, to wrap them in well-defined APIs and to expose them to the *Governance controller*. The rtGovOps agents offer advantages in terms of general scalability of the system and provide a degree of autonomy to the IoT Cloud resources.

8.5 Main Runtime Mechanism of the rtGovOps Framework

Generally, the rtGovOps framework supports operations managers to handle two main tasks. First, the rtGovOps framework enables dynamic, on-demand *provisioning of governance capabilities.* For example, it allows for dynamically injecting capabilities into IoT Cloud resources, and coordinating the dynamic profiles of these resources at runtime. Second, our framework allows for runtime management of governance capabilities throughout their entire lifecycle that, among other things, includes *remote capability invocation and managing dynamic APIs* exposed to users.

As we have mentioned earlier, in order to achieve a high-level governance objective such as enforce (part of) compliance policies for handling emergency situations an operations manager could design an operational governance process similar to the one shown in Figure 8.7 (top). Individual actions of such processes usually reference specific governance capabilities and rely on rtGovOps to support their execution. Figure 8.7 depicts a simplified sequence of steps executed by the rtGovOps framework when a governance capability gets invoked by an operational governance process. For the sake of clarity, we omit several steps performed by the framework and mainly focus on showing the most common interaction, i.e., we assume no errors or exceptions occur. We will discuss the most important steps performed by rtGovOps below. Note that all of these steps are performed transparently to operations managers and operational governance processes. The only thing that such processes observe is a simple API call (similar to REST service invocation) and a response (e.g., a JSON array in this case). Naturally, the process is responsible to provide arguments and/or configuration directives that are used by rtGovOps to parametrize the underlying capabilities.

8.5.1 Automated Provisioning of Governance Capabilities

In order to enable dynamic, on-demand provisioning of governance capabilities whenever a new capability is requested (i.e., referenced in an operational governance process), the rtGovOps framework needs to perform the following steps: i) the *ScopeCoordinator* resolves the governance scope to get a set of devices to which the capability will be added; ii) the *ProfileManager* checks whether the governance capability is available and compatible with the device; iii) the *Dependency Manager* resolves runtime dependencies of the capability; iv) the *ImageBuilder* creates a capability image; v) Finally, the *De-*

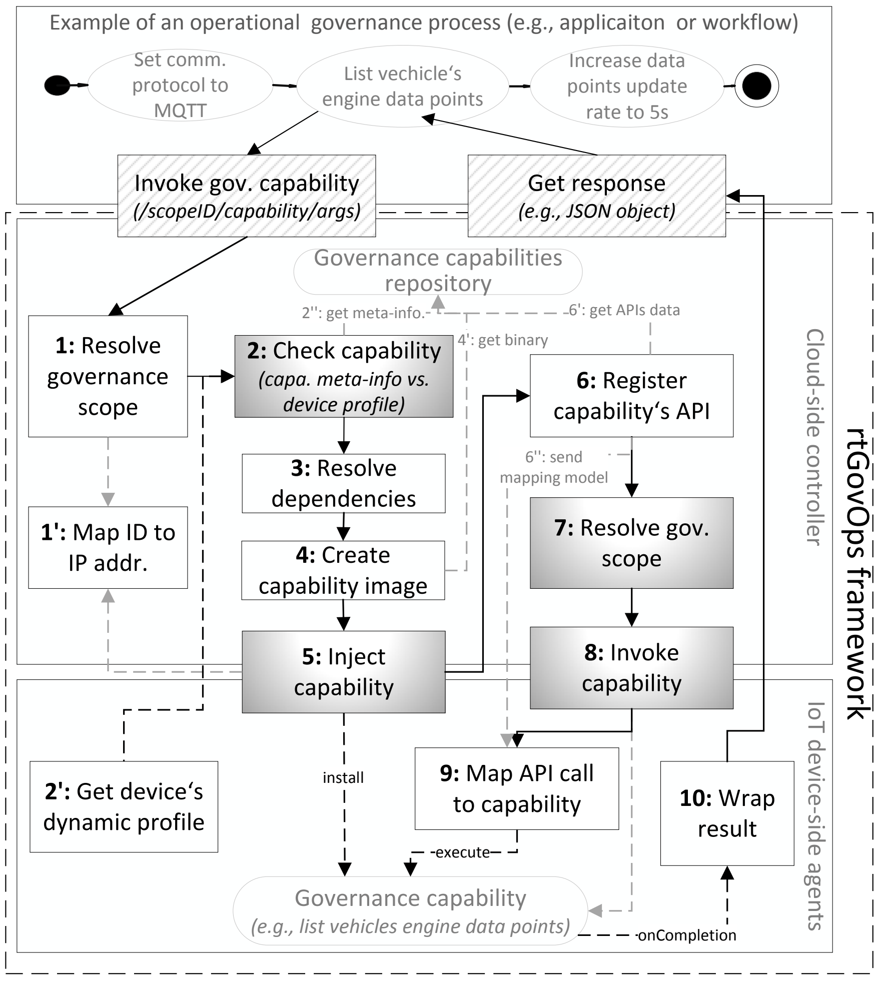

Example of an operational governance process (e.g., applicaiton or workflow)
Set comm. protocol to MQTT
List vechicle's engine data points
Increase data points update rate to 5s
Invoke gov. capability (/scopeID/capability/args)
Get response (e.g., JSON object)
Governance capabilities repository
2'': get meta-info.
6': get APIs data
4': get binary
1: Resolve governance scope
2: Check capability (capa. meta-info vs. device profile)
3: Resolve dependencies
4: Create capability image
5: Inject capability
6: Register capability's API
6'': send mapping model
7: Resolve gov. scope
8: Invoke capability
1': Map ID to IP addr.
2': Get device's dynamic profile
install
9: Map API call to capability
-execute-
10: Wrap result
Governance capability (e.g., list vehicles engine data points)
onCompletion
Cloud-side controller
IoT device-side agents
rtGovOps framework

Figure 8.7: Execution of an operational governance process.

ploymentManager injects the capability into devices. An overview of this process is also shown in steps 1 – 5 in Figure 8.7.

Algorithm 8.1 shows the capability provisioning process in more detail. An operational governance process requests a capability by supplying a capability ID (currently consisting of capability name and version) and an operational governance scope (more detail in Section 8.5.2). After that rtGovOps tries to add the capability (together with its runtime dependencies) to a device. If successful, it continues along the steps shown in Figure 8.7. The algorithm performs in a similar fashion to a fail-safe iterator, in the sense that it works with snapshots of devices states. For example, if something changes on the device side inside *checkComponent* (Algorithm 8.1, lines 2 – 5) it cannot be detected by rtGovOps and in this case the behavior of rtGovOps is not defined. Since we assume that all the changes to the underlying devices are performed exclusively by our framework, this is a reasonable design decision. Other errors, such as failure to install a capability on a specific device, are caught by rtGovOps and delivered as notifications to the operational governance process, so that they do not interrupt its execution.

Algorithm 8.1: Governance capability provisioning.

```
   input : capaID : A capability ID.
           gscope : Operational governance scope.
   result: Capability added to device or error occurred.
1  func checkComponent (component, device)
2      capaMeta ← queryCapaRepo(component)
3      devProfile ← getDeviceProfile(device)
4      status ← isCompatible(capaMeta, devProfile)
5      return status
6  end
   /* Begin main loop.                                   */
7  components ← resolveDependencies(capaID)
8  components ← add(capaID)
9  for device in resolveGovScope(gscope) do
10     for component in components do
11         if not checkComponent(component, device) then
12             error
13         end
14     end
       /* Inject capability.                              */
15     capaImg ← createImg(components)
16     deployCapa(capaImg, device)
17     installCapa(capaImg) // On device-side
18 end
```

Capability checking

From the steps presented in Algorithm 8.1 *checkComponent* (lines 1–6) and *injectCapability* (lines 15 – 17) are the most interesting. The framework invokes *checkComponent* for each governance capability and all of its dependencies for the currently considered device. At this point rtGovOps verifies that the component can be installed on this specific device. To this end, the *ProfileManager* first queries the *central capabilities repository*. Besides the capability binaries, the repository stores capability meta-information, such as required CPU instruction set (e.g., ARMv5 or x86), disk space and memory requirements, as well as installation and decommissioning directives. After obtaining the capability meta-information the framework starts building the current device profile. This is done in two stages. First, the gateway features catalog is queried to obtain relevant static information, such as CPU architecture, kernel version and installed userland (e.g., BusyBox[8]) or OS. Second, the *ProfileManager* in coordination with *DeviceProfiler* executes a sequence of runtime profiling actions to complete the dynamic device profile. For example, the profiling actions include: currently available disk space, available RAM,

[8]http://busybox.net

firewall settings, environment information, list of processes and daemons, and list of currently installed capabilities. Finally, when the dynamic device profile is completed, it is compared with the capability's meta information in order to determine if the capability is compatible with the device.

Capability injection

The rtGovOps *capability injection mechanism*, deals with uploading and installing capabilities on devices, as well as managing custom configuration models. This process is structured along three main phases: Creating a capability image, deploying the capability image on a device and installing the capability locally on the device.

1. After the *ProfileManager* determines a capability is compatible with the gateway, the *ImageBuilder* creates a capability image. The capability image is rtGovOps internal representation of the capability package (see Figure 8.5). In essence it is a compressed capability package containing component binaries and a dynamically created runlist. The runlist is an ordered list of components that need to be installed. It is created by the *DependencyManager* and its individual steps reference component installation or decommissioning directives that are obtained from the *capabilities repository*.

2. In the second phase, *DeploymentManager* deploys the image to the device. We support two different deployment strategies. The first strategy is *poll-based*, in the sense that the image is placed in the update queue and remains there for a specified period of time (TTL). The *ProvisioningAgent* periodically inspects the queue for new updates. When an update is available, the device can poll the new image when it is ready, e.g., when the load on it is not too high. A governance process can have more control over the poll-based deployment by specifying a capability's priority in the update queue. Finally, on successful update the *DeploymentManager* removes the update from the queue. The second deployment strategy allows governance capabilities to be asynchronously *pushed* to gateways. Since the capability is forced onto the gateway, it should be used cautiously and for urgent updates only, such as increasing a sensor poll-rate in emergency situations. Finally, independent of the deployment strategy, the framework performs a sequence of checks to ensure that an update was performed correctly (e.g., compares checksums) and moves to the next phase.

3. In the final phase, the *ProvisioningAgent* performs a local installation of the capability binaries and its runtime dependencies, and performs any custom configurations. Initially, *ProvisioningAgent* unpacks the previously obtained capability image and verifies that the capability can be installed based on the current device profile. In case the conditions are not satisfied, e.g., due to disk space limitation, the process is aborted and an error is sent to the *DeploymentManager*. Otherwise, the *ProvisioningAgent* reads the runlist and performs all required installation or decommissioning steps.

A limitation of the current rtGovOps prototype is that it only provides rudimentary support to specify installation and decommissioning directives. Therefore, capability providers need to specify checks, e.g., if a configuration file already exists, as part of the installation directives. In the future we plan to provide a dedicated provisioning DSL to support common directives and interactions.

8.5.2 rtGovOps APIs and invocation of governance capabilities

When a new governance capability is injected into a gateway, the rtGovOps framework performs the following steps: i) register capability with *APIManager*; ii) *ScopeCoordinator* resolves the governance scope; iii) *APIMediator* provides a mapping model to the *GovernanceAgent*; iv) the *GovernanceAgent* wraps the capability into a well-defined API, dynamically exposing it to the outside world; v) *CapabilityInvoker* invokes the capability and deliver the result to the invoking operational governance process when the capability execution completes. A simplified version of this process is also shown in steps $6-10$ in Figure 8.7.

Before we dive into technical details of this process, it is worth mentioning that currently in the capabilities repository, besides aforementioned capability meta-information and binaries, we also maintain well-defined capability API descriptions, e.g., functional, meta and lifecycle APIs. These APIs are available to operations managers as soon as a capability is added to the repository and independent of whether the capability is installed on any device. Additionally, we provide a general rtGovOps API that is used to allow for more control over the system and its capabilities. It includes *CapabilityManager* API (e.g., list capabilities, check if capability installed/active), capability lifecycle API (e.g., start, stop or remove capability), and *ProvisioningAgent* API (e.g., install new capability). Listing 8.1 shows some examples of such APIs as REST-like services (version numbers are omitted for clarity).

```
/* General case of capability invocation.     */
/govScope/{capabilityId}/{methodName}/{arguments}?
arg1={first-argument}&arg2={second-argument}&...

/* Data points capability invocation example. */
/deviceId/DPcapa/setPollRate/arguments?rate=5s
/deviceId/DPcapa/list

/* Capabilities manager examples.             */
/deviceId/cManager/capabilities/list
/deviceId/cManager/{capabilityId}/stop
```

Listing 8.1: Examples of capabilities and rtGovOps APIs.

Single invocation of governance capabilities

In the following we mainly focus on explaining the steps that are performed by the rtGovOps framework when a capability is invoked on a single device. The more general case involving multiple devices and using operational governance scopes is discussed in the next section.

When a capability gets invoked by an operational governance process for the first time, *APIManager* does not know anything about it. Therefore, it first needs to check, based on the API call (e.g., see Listing 8.1), if the capability exists in the central capabilities repository. After the capability is found and provisioned (Section 8.5.1), the rtGovOps framework tries to invoke the capability. This involves the following steps: registering the capability, mapping the API call, executing the capability, and returning the result.

1. First, the *APIManager* registers the API call with the corresponding capability. This involves querying the capability repository to obtain its meta-information (such as expected arguments), as well as building a dynamic mapping model. Among other things, the mapping model contains the capability ID, a reference to a runtime environment (e.g., Linux shell), a sequence of input parameters, the result type, and further configuration directives. The *APIMediator* forwards the model to the device (i.e. *GovernanceAgent*) and caches this information for subsequent invocations. During future interactions, the rtGovOps framework acts as transparent proxy, since subsequent steps are handled by the underlying devices.

2. In the next step, rtGovOps needs to perform a mapping between the API call and the underlying capability. Currently, there are two different ways to do this. By default, rtGovOps assumes that capabilities follow the traditional Unix interaction model, i.e., that all arguments and configurations (e.g., flags) are provided via the standard input stream (stdin) and output is produced to standard output (stdout) or standard error (stderr) streams. This means, if not specified otherwise in the mapping model, the framework will try to invoke the capability by its ID and will forward the provided arguments to its stdin. For capabilities that require custom invocation, e.g., property files, policies, or specific environment settings, the framework requires a custom mapping model. This model is used in the subsequent steps to correctly perform the API call.

3. Finally, the *CapabilityInvoker* in coordination with the *GovernanceAgent* invokes the governance capability. As soon as the capability completes, the *GovernanceAgent* collects and wraps the result. Currently, the framework provides means to wrap results as JSON objects for standard data types and it relies on the mapping model to determine the appropriate return type. However, this can be easily extended to support more generic behavior, e.g., by using Google Protocol Buffers[9].

Operational governance scopes

When an operational governance process gets invoked on a governance scope, the aforementioned invocation process remains the same, with the only difference that rtGovOps

[9]http://code.google.com/p/protobuf/

performs all steps on a complete governance scope in parallel instead on an individual device. To this end, the *ScopeCoordinator* enables dynamic resolution of the governance scopes.

There are several ways how a governance scope can be defined. For example, an operations manager can manually assign a set of resources to a scope, such as all vehicles belonging to a golf course, or they can be dynamically determined depending on runtime features by querying governance capabilities to obtain dynamic properties such as current configuration model. To bootstrap defining the governance scopes, the *ScopeCoordinator*, defines a global governance scope that is usually associated with all the IoT Cloud resources administered by a stakeholder at the given time. Governance scope specifications are implemented as composite predicates referencing device meta information and profile attributes, The predicates are applied to the global scope, filtering out all resources that do not match the provided attribute conditions. The *ScopeCoordinator* uses the resulting set of resources to initiate capability invocation with the *CapabilityInvoker*. The *ScopeCoordinator* is also responsible to provide support for gathering results delivered by the invoked capabilities. This is needed since the scopes are resolved in parallel and the results are asynchronously delivered by the IoT devices.

8.6 Evaluation & Prototype Implementation

8.6.1 Prototype implementation

In the current prototype, the rtGovOps *Governance controller* microservices are implemented in Java and Scala programming languages. The rtGovOps agents are based on lightweight httpd server and are implemented as Linux shell scripts. The complete source code and supplement materials providing more details about current rtGovOps implementation are publicly available in Git Hub[10].

8.6.2 Experiments setup

In order to evaluate how our rtGovOps framework behaves in a large-scale setup (hundreds of gateways), we created a virtualized IoT Cloud testbed based on CoreOS[11]. In our testbed we use Docker containers to virtualize and mimic physical gateways in the cloud. These containers are based on a snapshot of a real-world gateway, developed by our industry partners. The Docker base image is publicly available in Docker Hub under dsgtuwien/govops-box[12].

For the subsequent experiments we deployed a CoreOS cluster on our local OpenStack cloud. The cluster consists of 4 CoreOS 444.4.0 VMs (with 4 VCPUs and 7GB of RAM), each running approximately 200 Docker containers. Our rtGovOps agents are preinstalled in the containers. The rtGovOps Governance controller and capabilities repository are

[10]http://github.com/tuwiendsg/GovOps
[11]http://coreos.com/
[12]https://registry.hub.docker.com/u/dsgtuwien/govops-box/

deployed on 3 Ubuntu 14.04 VMs (with 2VCPUs and 3GB of RAM). The operational governance processes are executing on a local machine (with Intel Core i7 and 8GB of RAM).

8.6.3 Governing FMS at runtime

We first show how our rtGovOps framework is used to support performing operational governance processes on a real-world FMS application for monitoring vehicles (e.g., location and engine status) on a golf course (Section 8.1). The application consists of several services. On the one side, there is a light-weight service running in the vehicle gateways that interfaces with vehicle sensors via the CAN protocol, and feeds sensory data to the cloud. On the cloud-side of the FMS application, there are several services that, among other things, perform analytics on the sensory data and offer data visualization support. In our example implementation of this FMS application, the gateway service is implemented as a software-defined IoT unit that among other things provides an API and mechanisms to dynamically change the cloud communication protocol without stopping the service.

The FMS application polls diagnostic data from vehicles with CoAP. However, in case of an emergency, a golf course manager needs to increase the update rate and switch to MQTT in order to handle emergency updates in (near) real-time. This can be easily specified with an operational governance process that contains the following steps: change communication protocol to MQTT, list vehicle engine and location data points and set data points update rate, e.g, to 5 seconds. These steps are also depicted in Figure 8.7 (top). The golf course manager relies on rtGovOps governance capabilities to realize individual process steps and rtGovOps mechanisms (Section 8.5) to execute the operational governance process.

Figure 8.8 shows the bandwidth consumption of the FMS application that monitors 50 vehicles over a period of time. We notice two distinct operation modes: normal operation and operation in case of an emergency (emergency operation). Most notable are the two transitions: first, from normal to emergency operation and second, returning from emergency to normal operation. These transitions are described with the aforementioned operational governance process that is executed by the rtGovOps framework. The significant increase in bandwidth consumption happens during the execution of the operational governance process, because it changes the communication protocol from polling the vehicles approximately every minute with CoAP, to pushing the updates every 5 seconds with MQTT.

Typically, when performing processes such as the transition from normal to emergency operation without the rtGovOps framework, golf course managers (or generally operations managers) need to directly interact with vehicle gateways. This usually involves long and tedious tasks such as manually logging into gateways, dealing with device specific configurations or even an on-site presence. Therefore, realizing even basic governance processes, such as the one we presented above, involves performing many manual and error

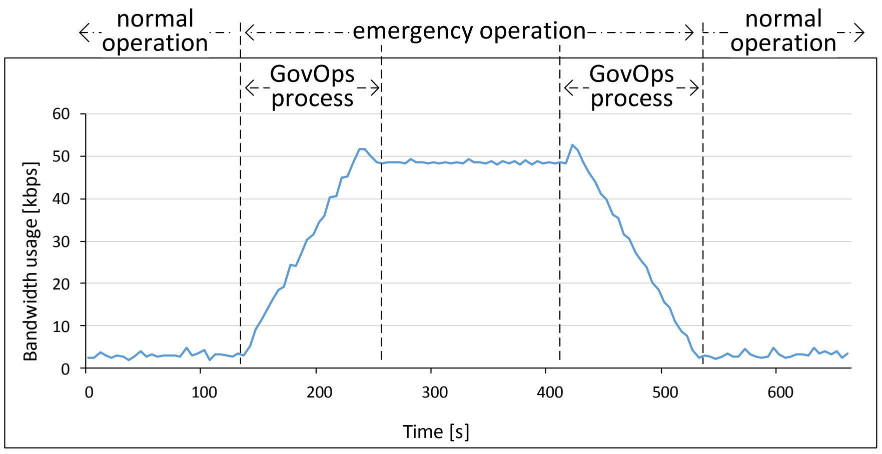


Figure 8.8: Example execution of operational governance process in the FMS.

prone tasks, usually resulting in a significant increases in operations costs. Additionally, in order to be able to have a timely realization of governance processes and consistent implementation of governance strategies across the system, very large operations and support teams are required. This is mainly due to the large scale of the FMS system, but also due to geographical distribution of the governed IoT resources, i.e., vehicles.

Besides the increased efficiency, the main advantage that rtGovOps offers to operations managers is reflected in the flexibility of performing operational governance processes at runtime. For example, in Figure 8.8 the execution of the operational governance process took around 2 minutes. In our framework this is, however, purely a matter of operational governance process configuration (naturally with upper limits as we show in the next section). This means, the operational governance process can be easily customized to execute the protocol transition “eagerly”, in the sense to force the change as soon as possible, even within seconds, or “lazy”, to roll-out the change step-wise, e.g., 10 vehicles at the time. The most important consequence is the opportunity to effectively manage tradeoffs. For example, executing the process eagerly incurs higher costs, due to additional networking and computation consumption, but it is needed in most emergency situations. Conversely, executing the process in a lazy manner can be desirable for non-emergency situations, since operations managers can prevent possible errors to affect the whole system.

Figure 8.8 also shows that rtGovOps introduces a slight communication overhead. This is observed in the two peaks at the end of the first process execution, when the framework performs the final checks that the process completed successfully and also when the second process gets triggered, i.e., when the capabilities get invoked on the vehicles. However, in our experiments this overhead was small enough not to be statistically significant. An additional performance-related concern of using rtGovOps is that network latency can slow down the execution of the operational governance process. However, since rtGovOps follows the microservices architecture style, it is possible to deploy relevant

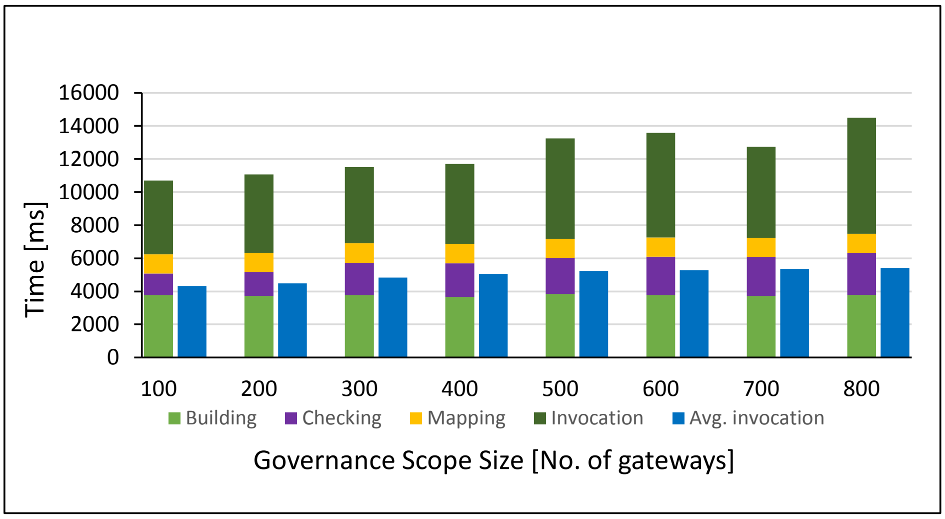


Figure 8.9: Capabilities first invocation.

services (*API*- and *DeploymentManager*) on Cloudlets [129] near the vehicles, e.g., on golf courses, where they can utilize local wireless networks.

8.6.4 Experiments results

To demonstrate the feasibility of using rtGovOps to facilitate operational governance processes in large-scale software-defined IoT Cloud systems, we evaluate its performance to govern approximately 800 vehicle gateways that are simulated in the previously described test-bed. In our experiments, we mainly focus on showing the scalability of the two main mechanisms of the rtGovOps framework: (i) capability invocation and (ii) automated capability provisioning. We also consider the performance of capability checking and governance scope resolution. The reason why we put an emphasis on the scalability of our framework is that it is one of the key factors to enable consistent implementation of governance objectives across a large-scale systems. For example, if the execution of an operational governance process were to scale exponentially with the size of the resources pool, theoretically it would take infinitely long to have a consistent enforcement of the governance objectives in the whole system, with sufficiently large resource pool. The results of the experiments are averaged results of 30 repetitions and we have experimented with 5 different capabilities that have different properties related to their size and computational overhead.

Figure 8.9 shows the execution time of the first invocation of a capability (stacked bar) and an average invocation time of capability execution (plain bar). We notice that the first invocation took approximately between 10 and 15 seconds and average invocation varied between 4 and 6 seconds depending on the scope size (measured in the number of gateways). The main reason for such a noticeable difference is the invocation caching performed by rtGovOps. This means that most of the steps, e.g., building

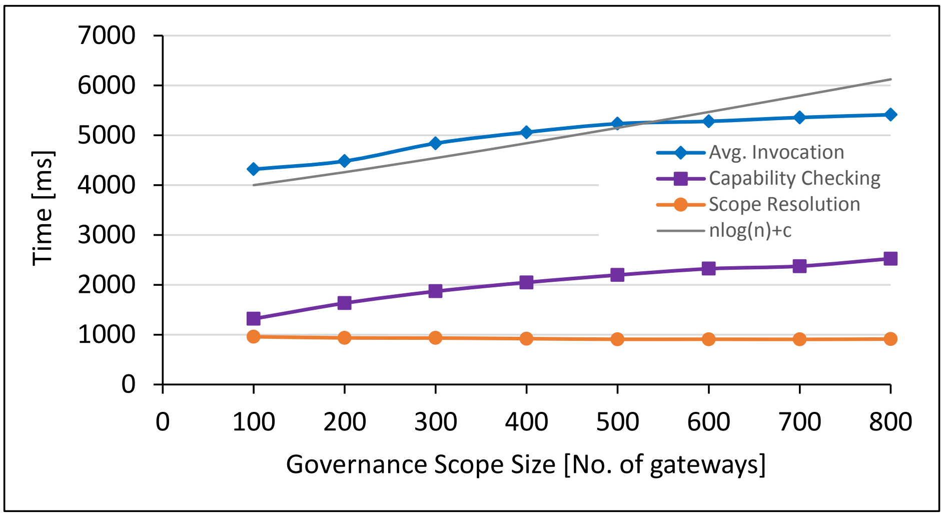


Figure 8.10: Average invocation time of capabilities on a governance scope.

capability image and building the mapping model are only performed when a capability is invoked for the first time, since in the subsequent invocations the capability is already in the gateways and the mapping can be done in cache. In Figure 8.10, we present the average execution time of a capability (as it is observed by an invoking operational governance process on the locale machine), average execution of capability checking mechanism and governance scope resolution. As a reference, the diagram also shows a plot of a $nlog(n) + c$ function. We can see that the mechanisms scale within $O(nlog(n))$ for relatively large governance scopes (up to 800 gateways), which can be considered a satisfactory result. We also notice that computational overheads of the capabilities have no statistically significant impact on the results, since they are distributed among the underlying gateways. Finally, it is interesting to notice that the scope resolution time actually decreases with increasing scope size. The reason for this is that in the current implementation of rtGovOps, scope resolution always starts with the global governance scope and applies filters (lambda expressions) on it. After some time Java JIT "kicks-in" and optimizes filters execution, thus reducing the overall scope resolution time.

In Figure 8.11, we show the general execution times of the rtGovOps capability provisioning mechanism (push-based deployment strategy) for two different capabilities. The first one has a size order of magnitude in MB and second capability size is measured in KB. There are several important things to notice here. First, the capability provisioning also scales similarly to $O(nlog(n))$. Second, after the governance scope size reaches 400 gateways there is a drop in the capability provisioning time. The reason for this is that the rtGovOps load balancer spins-up additional instances of the *DeploymentManger* and *ImageBuilder*, naturally reducing provisioning time for subsequent requests. Finally, the provisioning mechanism behaves in a similar fashion for both capabilities. The reason for this is that all gateways are in the same network, what can be seen as an equivalent to

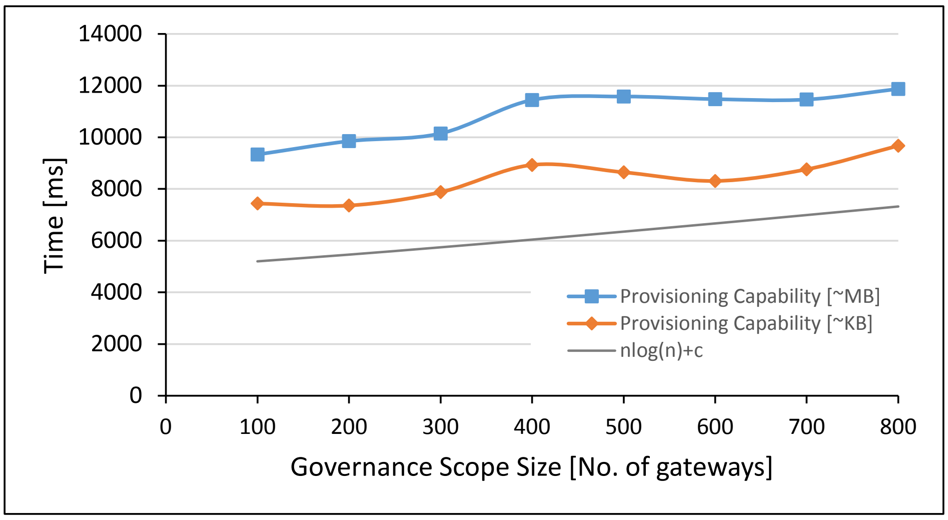


Figure 8.11: Average capability provisioning duration (push-based strategy).

vehicles deployed on one golf course.

8.6.5 Discussion and lessons learned

The observations and results of our experiments show that rtGovOps offers advantages in terms of realizing operational governance processes with greater flexibility, and also makes such processes easily repeatable, traceable and auditable, which is crucial for successful implementation of governance strategies. Generally, by adopting the notion of governance capabilities and by utilizing resource agents, rtGovOps allows for operational governance processes to be specified with *finer granularity (R3)*, but also give a *degree of autonomy (R5)* to the managed IoT Cloud resources. Therefore, by selecting suitable governance capabilities, operations managers can precisely define desired states and runtime behavior of software-defined IoT Cloud systems. Further, since the capabilities are executed locally in IoT Cloud resources (e.g., in the gateways), our framework enables better utilization of the "edge of infrastructure" and allows for local error handling, thus increasing system availability and scalability. Further, the main advantage of approaching provisioning and management of governance capabilities in the described manner is that operation managers do not have to worry about geographically-distributed IoT Cloud infrastructure nor deal with individual devices, e.g., key management or logging in. They only need to *declare (R4)* which capabilities are required in the operational governance process and specify a governance scope. The rtGovOps framework takes care of the rest, effectively giving a *logically centralized view (R1)* on the management of all governance capabilities. Further, by *automating (R2)* the capability provisioning, rtGovOps enables installing, configuring, deploying, and invoking the governance capabilities in a scalable and easily repeatable manner, thus reducing errors, time, and eventually costs of operational governance.

It should be also noted that there is a number of technical limitations of and possible optimizations that can be introduced in the current prototype of the rtGovOps framework. As we have already mentioned, rtGovOps currently offers limited support for specifying provisioning directives. Additionally, while experimenting with different types of capabilities, we noticed that in many cases a better support to deal with streaming capabilities would be useful. Regarding possible optimizations, in the future we plan to introduce support for automatic composition of capabilities on the device level, e.g., similar to Unix piping. This should reduce the communication overhead of rtGovOps and improve resource utilization in general. In spite of the current limitations, the initial results are promising, in the sense that rtGovOps *increases flexibility* and enables *scalable execution of operational governance processes* in software-defined IoT Cloud systems.

8.7 Conclusion

In this chapter, we introduced the GovOps approach to runtime governance of IoT Cloud systems. We presented the GovOps reference model that defines suitable concepts and a flexible process to design IoT Cloud governance strategies. We introduced the GovOps manager, a dedicated stakeholder responsible to determine suitable tradeoffs between satisfying governance objectives and IoT Cloud system capabilities, and ensure early integration of these objectives with operations processes, by continuously refining how the high-level objectives are articulated through operations processes.

Moreover, this chapter introduced the rtGovOps framework that serves as GovOps reference implementation, providing support for designing and executing operational governance processes. We presented rtGovOps' main runtime mechanisms and enabling techniques that support operations managers to handle two main tasks: (i) perform dynamic, on-demand provisioning of governance capabilities and (ii) remotely invoke such capabilities in IoT Cloud remotely, via dynamic APIs. We demonstrated, on a real-world case study, the feasibility of GovOps methodology and framework to facilitate execution of operational governance processes in large-scale IoT Cloud systems.

The initial results are promising in several aspects. We showed that the rtGovOps framework enables operational governance processes to be executed in a scalable manner across relatively large IoT Cloud resource pools. Additionally, we discussed how rtGovOps enables flexible execution of operational governance processes by automating the execution of such processes to a large extent, offering finer-grained control over IoT Cloud resources and providing a logically centralized interaction with IoT Cloud resource pools. Finally, we discussed how GovOps allows for IoT Cloud governance processes to be realized in practice without worrying about the complexity and scale of the underlying IoT Cloud and diversities of various legal and compliance issues.

CHAPTER 9

Governing Elastic IoT Cloud Systems under Uncertainty

Emerging elastic IoT systems extend contemporary cloud systems beyond the data centers to include a variety of edge IoT devices, such as sensors and sensory gateways [143]. On the other hand, these systems utilize cloud resources, to enhance resource-constrained IoT devices, but also to enable elastic delivery and consumption of the vast IoT resources through the cloud computing on-demand pay-per-use model. This has proliferated *unified IoT cloud infrastructures* which comprise large pools of IoT and cloud resources ranging from large data centers and the edge of the network (cf. Chapter 7). In such systems, governance strategies are a useful mechanism to address issues related to risk mitigation, compliance and legal requirements, as we have discussed in Chapter 8. However, due to numerous uncertainties inherently present in the IoT Cloud systems, realizing these strategies poses a plethora of challenges. Such uncertainties are mainly caused by the novel interactions of IoT elements, network elements, cloud resources and humans. For example, uncertainties related to state monitoring, data delivery and performance variability (e.g., due to probe failures, network issues or human error) often lead to imperfect data about the infrastructure. As a result, infrastructural information needed by operational governance processes might be incomplete or inaccurate, thus hindering the operational tasks of both automated management systems and the end users. This serves as the main motivation for extending our GovOps approach (cf. Chapter 8) to include uncertainty considerations in operational governance processes from early design stages.

In this chapter we introduce the U-GovOps framework for governance of elastic IoT cloud systems under uncertainty. The U-GovOps conceptually extends the GovOps approach and technically refines the rtGovOps framework, in which we have tacitly assumed perfect information (e.g., about IoT cloud resource states), and reliable and deterministic behavior of the IoT cloud infrastructure. Unfortunately, due to the infrastructure uncertainties, such assumptions are unrealistic and difficult to meet in practice, thus putting a lot of burden on the developers and operations managers (users) to deal with the uncertainties in ad-hoc fashion. For this purpose, the U-GovOps framework introduces novel techniques to facilitate *developing and executing* the governance strategies under presence of the uncertainty. The main contributions of the framework presented in this chapter include: i) A *declarative policy language* for developing uncertainty- and elasticity-aware governance strategies for IoT cloud systems. ii) *Runtime mechanisms and uncertainty mitigation techniques*, which support execution of such strategies under uncertainty.

The remainder of the chapter is structured as follows: Section 9.1 presents motivation and research challenges; Section 9.2 outlines the design of the U-GovOps framework; In Section 9.3, we present U-GovOps declarative policy language; Section 9.4 introduces U-GovOps runtime mechanisms for uncertainties mitigation; Section 9.5 describes the current prototype implementation and presents the results of our experiments; Finally, Section 9.6 concludes the chapter.

9.1 Motivation & Research Challenges

9.1.1 Scenario

As we discussed in Chapter 2, generally, the BMS comprises various applications and subsystems responsible to monitor and control different building assets such as HVAC, lightning, elevators and plumbing facilities, as well as to handle fault events and alarms (e.g, fire or gas leakage). For our discussion in this chapter, the most relevant application of this system is Predictive Maintenance Application (PMA)[1]. The PMA runs atop a complex IoT cloud infrastructure that includes (i) various edge devices, such as, (software and hardware) sensors, actuators and gateways, (ii) network elements, and (iii) cloud services, e.g., for complex event processing, NoSQL data storage, and streaming data analysis. All of these infrastructural elements need to be governed throughout their entire lifecycle. As shown in Chapter 8 this can be aceived by designing suitable governance strategies. However, numerous uncertainties interfere with the execution of such strategies, making the implementation of even rudimentary governance strategies a challenging task.

To exemplify uncertainties in such governance strategies, let us consider an operational governance process in the PMA. Typically, the PMA polls diagnostic data from equipment, such as chiller plants (e.g., with CoAP), but for optimization purposes (mainly network consumption) not all available sensory data are polled from the cloud. However, in situations such as an emergency or multiple devices failure, the PMA needs to change its operation to be in accordance with company's legal regulatory compliance, e.g., to handle status updates in (near) real time. To satisfy such governance objective a maintenance manager could create a governance strategy which “activates" all available sensors, changes the communication protocol to push-based, e.g., MQTT, and sets the

[1]We provided an implementation of PMA at `https://github.com/tuwiendsg/DaaSM2M/wiki/`

sensors update rate to maximum. Finally, after such situations have been dealt with, the PMA should return to its normal operation mode. In such situations, we need to rely on up-to-date and highly accurate infrastructural state information and stable performance of control actions of various resources to adjust the IoT cloud systems. However, in real world it is hard, if not impossible to achieve them. Therefore, to deal with such situations, on the one hand, we need to capture different types of uncertainties related to state information and performance variability of underlying resources to allow for strategies specified for different uncertainties. On the other hand, we need to develop runtime mechanisms to support these governance strategies under such uncertainties.

9.1.2 Research Challenges

Capturing infrastructure-level uncertainties in IoT cloud systems

Inspired by the traditional fault, error, failure classification [10] and the general belief model [122], in our work we have identified different uncertainties for IoT cloud infrastructure. To systematically document uncertainties, we have developed a taxonomy and use this taxonomy to classify the uncertainties and analyze their effects on the typical governance strategies[2]. Our taxonomy mainly focuses on infrastructure uncertainties that originate at runtime. Other uncertainties such as design- or requirements-level uncertainties [125] are currently not considered. Figure 9.1 gives a high-level overview of the taxonomy and its main concepts (uncertainty classes) which are used to classify the infrastructure-level uncertainties: i) *Temporal manifestation* reflects the duration of the uncertain (or failure) state caused by an uncertainty. ii) *Nonfunctional dimensionality* captures affected nonfunctional properties of the infrastructure. For example, the uncertainties can affect well-known infrastructure's dependability [10] , quality of sensory data, or regulatory compliance [104]. iii) *Cause of* uncertainty can be a natural phenomenon, a human action or a technological phenomenon (anything caused by an infrastructure phenomenon, which is beyond user's control). iv) *Effect propagation* denotes

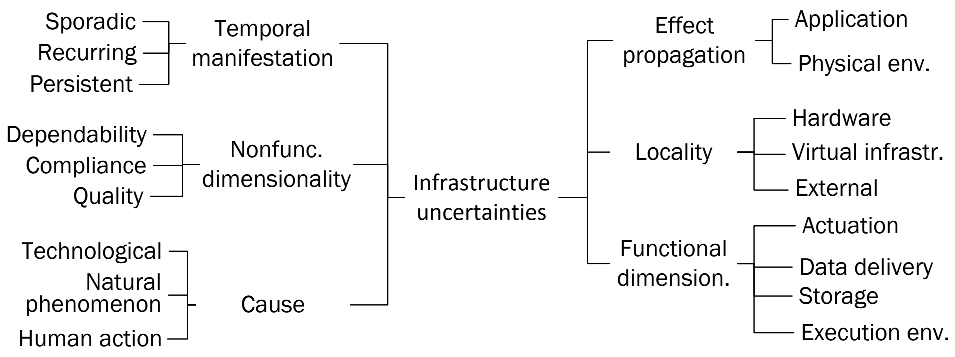


Figure 9.1: Taxonomy for IoT cloud infrastructure uncertainties.

[2]The description of the taxonomy is out of the scope of this chapter. A detailed description of the uncertainty taxonomy is provided as supplement material in Appendix A.

whether an uncertainty affects the application or the physical environment. v) *Locality* describes where an uncertainty occurs. We differentiate between uncertainties present in hardware, platform (virtual infrastructure) or external to infrastructure.vi) *Functional dimensionality* denotes the category of infrastructure's functionality that is affected by an uncertainty, e.g., execution environment, actuation, data delivery and storage facilities.

In order to classify an uncertainty, the users describe its general properties by assigning it to the uncertainty classes shown in Figure 9.1. For example, in our taxonomy, a freezing of sensors could be classified as affecting the dependability of the data delivery facilities, caused by a natural phenomenon, external to infrastructure that sporadically manifests itself during application runtime. When classifying uncertainties, we notice that not all combinations of the uncertainty classes are allowed. For example, it makes no sense to have a natural phenomenon uncertainty which occurs at the virtual infrastructure level (in software). Based on this observation we identified 11 elementary uncertainty families. The introduced taxonomy enables the users (e.g., maintenance manager) to capture their knowledge about potential uncertainties, in a systematic and structured way, based on the set of general, well-defined concepts. Besides the common elements used in the governance strategies, e.g., runtime monitoring information, enabling the users to embed such knowledge in the governance strategies is crucial for development of the strategies that can account for the runtime uncertainties. For our following discussion, we focus on two most relevant uncertainty families, which affect the tasks performed in typical governance strategies namely *DataQualityUncertainties* and *ActuationDependabilityUncertainties.* We elaborate the main governance challenges caused by these uncertainties.

Main challenges of IoT cloud governance under uncertainty

One of the main tasks of governance strategies is to identify a *governance scope.* As discussed in Chapter 8, governance scopes represent a set of IoT cloud resources that should be governed by such strategies. The resources are selected and assigned to the governance scopes based on their properties, i.e, the governance scopes are specified as composite predicates referencing the resource attributes. Such attributes mainly reference resource's meta data (mainly specified by humans) and resource's profile data (mainly based on sensory readings) that are used to compute the governance scopes. However, in practice, the DataQualityUncertainties often lead to *incomplete or missing data* about resources and their states in IoT cloud systems, such as null attribute values (e.g., due to monitoring failures or human error). These quality of data problems make it very challenging to determine the governance scopes. Currently, the users deal with such imperfect information in an ad-hoc fashion, e.g., by writing complex queries or developing sophisticated probabilistic models. This pollutes the governance logic with uncertainty management, making the governance strategies difficult to maintain, less traceable and significantly increasing the development effort.

Another key task performed by governance strategies is (remote) invocation of the *governance actuations* (cf. Chapter 8), e.g., to increase sensors update rate, as well as the *elasticity actuations* [33], e.g., to keep the cloud services' response time within the specified limits, when the sensors update rate is increased. Such actuations are often subject to the ActuationDependabilityUncertainties, which degrade dependability of the actuation facilities (e.g., due to network latency, device failure or race conditions), manifesting itself often transparently to the users, as lost actuations, cascading failures or resource over-consumption. This usually causes an inconsistent realization of governance strategies or even renders them completely useless, thus causing breaches of regulations or compliance.

9.2 The U-GovOps framework

9.2.1 Managing uncertainties in governance strategies

Altdough uncertainty is not limited to the absence of knowledge, is tightly related to the lack thereof [148]. Further, it strongly depends on the task-at-hand and on the system setup and environment [163]. When dealing with uncertainties, generally we are more interested in the effects of such uncertainties than the actual uncertainties. In the previous section we broadly discussed the main challenges, caused by the uncertainty, which affect governance strategies, by causing different problems. Therefore, by categorizing the uncertainties, analyzing their *Effects* and measuring the degree of sensitivity to such uncertainties (in our taxonomy captured with *Nonfunctional dimensionality*), we can formulate more precise statements such as: "An uncertainty X affects the *application dependability* by causing *resource over-consumption* and potentially leads to a *complete functionality failure* of actuation facilities". This allows for streamlining the uncertainty management by enabling the users to employ general, well-defined knowledge concepts, e.g., from software engineering or the domain of interest, to derive requirements, actions and configuration models needed to define suitable mitigation strategies. The main aim of our U-GovOps framework is to facilitate the runtime governance of elastic IoT cloud systems under presence of uncertainty by incorporating such requirements and configurations from the early stages of strategy design. To this end, U-GovOps supports the users to design elasticity- and uncertainty-aware governance strategies.

While governance strategies are mainly used to address issues related to risk mitigation, compliance and legal requirements, it is often useful to incorporate elasticity actuations in the governance strategies to enable the users to also anticipate changes in resource demand, costs and quality of the governed systems (encompassing both IoT and cloud infrastructures). For example, by considering elasticity relationships [143], while designing the elasticity-aware governance strategies, users can anticipate increases in resource demand, e.g., since they know that some other governance actions increase the sensors' update rate. They can utilize this knowledge, for instance to "warm up" VMs in order to mitigate the uncertainties related to the actuation delays (of spinning up the VMs) when scaling out related cloud services. However, due to an intrinsic "bootstrapping problem" this only facilitates uncertainty management to a certain extent, since the mechanisms underpinning the governance and elasticity actuations are also subject to uncertainty (Section 9.1). For this reason the U-GovOps framework allows the users to incorporate uncertainty considerations in governance strategies, effectively raising the awareness level of such strategies. To this end, U-GovOps defines a *governance policy language* for developing uncertainty- and elasticity-aware governance policies (Section 9.3) and provides a language runtime (*Governance and Elasticity Controllers*) that does most of the “heavy lifting" to support executing governance policies, without explicitly worrying about the infrastructure uncertainties.

9.2.2 U-GovOps architecture

The U-GovOps framework is distributed across the Clouds and IoT devices. In U-GovOps, the *GovernancePolicyProcessor* (Figure 9.2) represents a central point of interaction with the *Governance Controller* and the *Elasticity Controller*, i.e., it is responsible to interpret the user-provided governance policies (strategies), described latter in Section 9.3 and map them to the controllers API, exposed by the *API Manager*.

The *Governance Controller* comprises several microservices, the most important being the *GovernanceScopeCoordinator*. It provides mechanisms to define and manage the governance scopes, in order to determine which IoT cloud resources will be affected by a governance strategy. It relies on the *ProfileManager* to dynamically build and manage resource (e.g., device) profiles. This involves managing static device meta-information and performing profiling actions in order to obtain runtime snapshots of current device states. The *Elasticity Controller* is based on rSYBL [33] and it provides general mechanisms to handle elasticity actuations specified in governance policies. Its main microservices include: The *ControlEngine*, which implements the elasticity control algorithms, e.g.,

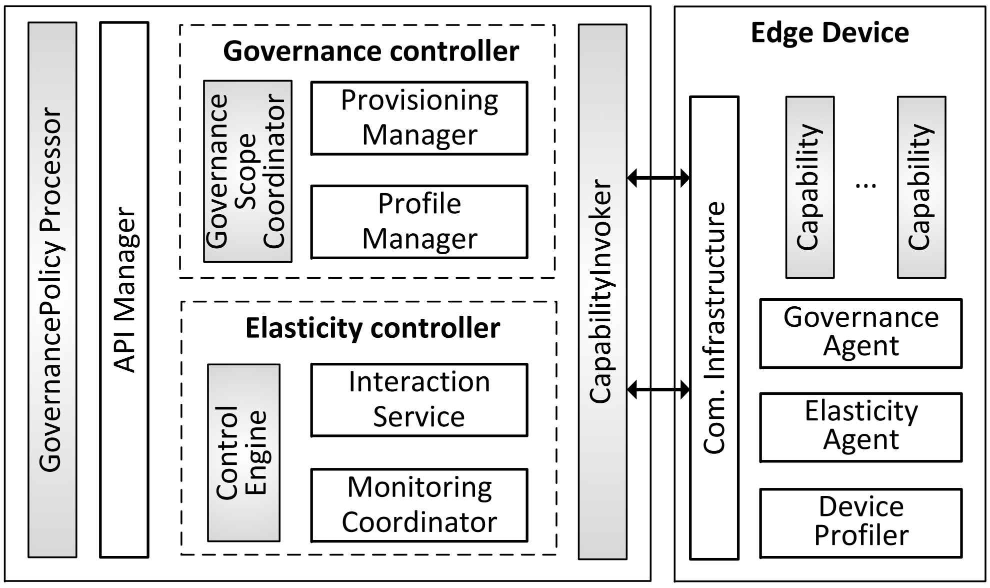


Figure 9.2: Overview of U-GovOps architecture.

greedy planning; The *MonitoringCoordinator* that is used to integrate infrastructure monitoring frameworks such as Nagios or Ganglia and; The *InteractionService*, which encapsulates higher-level control mechanism, e.g., exposed by an IaaS provider.

These controllers rely on the *CapabilityInvoker* to perform the actual invocations of the underlying capabilities. i.e., perform actuations on the IoT cloud resources over the network (denoted as two-way arrows in Figure 9.2). For this purpose the framework uses the *ElasticityAgent*, the *GovernanceAgent* and the *DeviceProfiler*, which are responsible to manage local governance and elasticity capabilities and to expose them to the controllers. They are very light-weight components that run in all IoT cloud resources that are managed by U-GovOps.

9.3 A DSL for Developing Uncertainty- and Elasticity-aware Governance Strategies

9.3.1 U-GovOps declarative policy language

In order to facilitate governing IoT cloud systems under uncertainty, the U-GovOps framework provides a declarative policy language for developing uncertainty- and elasticity-aware governance policies. It is based on our previously developed SYBL [33] and its main aim is to support developers and operations managers (users) to design such policies on a higher-level of abstraction, without explicitly dealing with IoT cloud infrastructure uncertainties. Two main tasks that users perform are identifying the governance scopes and defining the governance and elasticity actuations, to be applied on such scopes (Section 9.1). In our language, `STRATEGY` directive allows the specification of the governance or elasticity actuations to be undertaken (e.g., set sensor update rate) or desired behavior to be enforced (e.g., maximize throughput) when specific conditions are met. Further, to declare the governance scopes, determining which resources should be affected by such actuations, our language offers `GOVERNANCE_SCOPE` directive. Finally, to support the users to articulate their knowledge about the uncertainties, namely to raise the level of awareness in governance policies, the U-GovOps language provides the `CONSIDERING_UNCERTAINTY` construct. It is mainly used to specify configuration directives for determining the behavior of the governance scopes and governance or elasticity actuations.

The full syntax of the U-GovOps language is described in Appendix B. Subsequently, we describe the most important language concepts and supporting runtime mechanisms in more detail, mainly focusing on: 1) Rough governance scopes and 2) Isolated (governance and elasticity) actuations. In the remainder of the chapter we mostly focus on describing our framework's support for managing the uncertainties related to the *actuation dependability* and *incomplete and missing data* about IoT cloud systems, which were identified as most relevant for our work.

Rough governance scopes

In order to support governance policy developers to deal with the uncertainties related to missing and incomplete data (Section 9.1), the U-GovOps framework introduces a new concept called *rough governance scope*. Generally, a rough governance scope represents a formal approximation of a resource set, taking into account resources, which due to uncertainty, cannot be positively (i.e. with absolute certainty) characterized as members of the targeted governance scope. Rough governance scopes are modeled based on the rough set theory, which unlike fuzzy sets or probabilistic models, has an advantage of providing an objective formal approximation of membership relation [120]. Practically, this means that even with no user involvement, U-GovOps can make an objective approximation of resource assignments to governance scopes under data uncertainty.

Formally, a rough governance scope is defined as a tuple $\langle \underline{G}X, \overline{G}X \rangle$, where $\underline{G}X$ and $\overline{G}X$ are traditional (crisp) sets that represent lower and upper approximation in the rough governance scope, given the set of attributes G. The G-lower approximation is the union of all equivalence classes (granules) $G(x)$ that are a subset of the targeted governance scope X: $\underline{G}X = \bigcup\{G(x) \mid G(x) \subseteq X\}$. The G-upper approximation is the union of all in $G(x)$ which have non-empty intersection with the targeted governance scope X: $\overline{G}X = \bigcup\{G(x) \mid G(x) \cap X \neq \emptyset\}$ [120]. Therefore, $\underline{G}X$ represents a positive (or pessimistic) approximation and $\overline{G}X$ represents an optimistic approximation of the targeted governance scope.

```
G:GOVERNANCE_SCOPE
 query:= location=buildingX & type=JACE-545
 CONSIDERING_UNCERTAINTY:
  missing_data = "location<='?',type<='*'" AND
  selection_strategy = optimistic AND
  use_cache = false !\DNumber!
```

Listing 9.1: Example governance scope.

Listing 9.1 shows an example governance scope defined with U-GovOps policy language. In our language, a governance scope is specified as composite predicates referencing device meta information and profile attributes within the `query` parameter. To specify the behavior of governance scopes under data uncertainty users provide additional directives within the `CONSIDERING_UNCERTAINTY` construct. The `selection_strategy` parameter can take values: *optimistic, pessimistic* or *reduct*. It instructs the framework on how to treat the resources belonging to the boundary region ($\overline{G}X - \underline{G}X$), which due to uncertainty (e.g., incomplete attribute set) cannot be positively characterized as members of the governance scope. For example, selecting the optimistic strategy means that U-GovOps will compute the governance scope based on the upper ($\overline{G}X$) approximation. This behavior might be desirable when a governance policy can tolerate false positives, but it must not have any false negatives included in the governance scope. With this knowledge the framework can compute an objective approximation of the governance scope, even if the governed resources are indiscernible with the available attributes in

G. More details about the underlying mechanisms are provided in Section 9.4. This is sufficient to address the uncertainties related to the completeness of the data (e.g., available resource attributes).

However, to be able to handle the missing data, the governance scope membership relation must be refined with a subjective extension. To this end U-GovOps utilizes the concepts of characteristic relations and characteristic sets [61]. Essentially, this enables the users to specify how the missing data should be interpreted. The `missing_data` directive enables the users to generally define interpretation of the missing attribute values as “do not know" [137] or “do not care" [84], depending on the task-at-hand, since there is no universally best interpretation of the missing attribute values [61]. The former concept (denoted with ‘?’) is used to indicate the lost data, e.g., missing sensory readings for an attribute. The latter (denoted with ‘$*$’ or ‘$-$’) indicates the unavailable data, e.g., attributes initially deemed irrelevant by a human, thus potentially not included in all resource descriptions.

Isolated actuations

As mentioned earlier, governance and elasticity actuations are declared via the `STRATEGY` construct. It encapsulates actuations such as “change communication protocol" or "spin up a VM". However, the underlying capabilities (which implement the actuation logic) are mainly running at the edge of the infrastructure, e.g., inside IoT gateways, and are invoked remotely over the network. Therefore, this often leads to failures and functionality degradations (transparent to users) as we discussed in Section 9.1.

In order to support the users in managing such uncertainties, U-GovOps offers two levels of actuation isolation – per governance policy and per capability invocation. To instruct U-GovOps to isolate a governance policy users can specify `run_in_isolation = true` (Listing 9.2). This effectively tells the framework to create a separate resource pool (e.g., a thread pool) for the policy and perform all actuations within that resource pool. More details about the design of this mechanism are given in Section 9.4.

To provide finer-grained control for the isolated policies and actuations, the U-GovOps framework exposes additional configuration parameters. For example, the `keep_alive` parameter enables users to specify the maximal time slot that should be allocated to

```
S:STRATEGY CASE Fulfilled(CND1):
 setUpdateRate(5s) FOR G //see Listing 1
 CONSIDERING_UNCERTAINTY:
   run_in_isolation = true AND
   keep_alive = 5min AND
   degree_parallelism = 200 AND
   tolerate_fault_percentage  = 20% AND
   fallback_count = 2 AND
   time_to_next_fallback = 500ms
```

Listing 9.2: Example of an isolated actuation with uncertainty considerations.

```
1   C:CONSTRAINT responseTime<150ms WHEN nrOfUsers<900
2    CONSIDERING_UNCERTAINTY:decision_confidence >=20%
3   S1:STRATEGY CASE Violated(C):scaleOut()
4   S2:STRATEGY CASE Fulfilled(C):maximize(throughput)
5    CONSIDERING_UNCERTAINTY:
6     considering_strategies = StrategyX
```

Listing 9.3: Example elasticity actuations with uncertainty considerations.

a governance policy to complete. The `tolerate_fault_percentage` is a similar concept, designed to temporarily stop the policy execution in case the percentage of failed actuations exceeds a pre-defined threshold. These two concepts are based on the circuit breaker pattern[3], which are especially useful for handling blocked or zombie policies and reducing the resources tied up in operations which are likely to fail due to uncertainties. Further, the `degree_parallelism` tells U-GovOps how many actuations should be performed in parallel. This is useful in capturing the user's knowledge about the infrastructure's scale and dynamicity in order to optimize resource consumption. For example, if a governance policy is meant to govern all active gateways in a building (e.g., ≈ 300 at the time) it makes little sense to set the degree of parallelism to 1000. Finally, the `fallback_count` and `time_to_next_fallback` parameters are used to handle uncertainty at the level of a single actuation. Its main purpose is to support graceful handling of network latencies and timeouts and to guaranty fail-fast behavior (with quick recoveries) and graceful functionality degradation (with fallback logic).

Listing 9.3 gives an example of using elasticity actuations in governance policies. It first defines a `CONSTRAINT` directive, which describes desired conditions of keeping the response time below 150 ms if the number of current users is below 900. Lines 3 and 4 in Listing 9.3 tell U-GovOps to fire appropriate elasticity actuations based on the status of the constraint. However, the elasticity actuations are also subject to uncertainty, our language also allows for uncertainty configuration directives for elasticity controls. for example, such uncertainties originate due to hardware or platform glitches (e.g., unsuccessful network interface attachment) or infrastructure overload (e.g., collocation issues on physical servers) leading to unexpected behavior such as actuation delays. To account for such issues, U-GovOps allows users to specify their knowledge about the elasticity relationships, such as that increasing sensors update rate will most probably require scaling out the cloud services. The elasticity relationships can be specified via `considering_strategies` parameter, effectively enabling the framework to anticipate the aforementioned situations and for instance preemptively spin up required VMs. Naturally, all the uncertainty directives shown in Listing 9.2 are also valid in this context.

[3] http://martinfowler.com/bliki/CircuitBreaker.html

9.4 U-GovOps Runtime Mechanisms for Mitigating Governance Uncertainties

Resolving rough governance scopes at runtime

When a request to compute a rough governance scope (Listing 9.1) arrives in U-GovOps runtime, the framework performs the following general steps: i) It first evaluates the user-provided `query` and performs the resource selection with the currently available data. If no uncertainty parameter is specified or `use_cache=true` and there is a precomputed governance scope for the query, the U-GovOps framework immediately returns the obtained resource set. Otherwise, it proceeds with the next steps. ii) Parametrize the missing data. iii) Calculate Similarity Classes [137]. iv) Calculate characteristic sets. v) Return a governance scope approximation.

Algorithm 9.1: Computing characteristic sets.

```
input : $res$ : Governed resource, $GS$ : Global scope, $G$ : Attribute list
result: $CS$ : Characteristic set for the $res$.
1  $CS \leftarrow GS$
2  forall the $attr$ in $G$ do
3      switch $attr$ do
4          case $'?' = res.attr$
5              $CS \leftarrow GS$
6          case $'*' = res.attr$
7              foreach $val \in AttrDomain(attr)$ do $CS \leftarrow CS \cup SimilarityClass(attr, val)$
8          case $'-' = res.attr$
9              $V \leftarrow \{r | r \in GS, isDefined(r, attr), r.d = res.d\}$
10             if $V \neq \emptyset$ then
11                 foreach $r \in V$ do
12                     $CS \leftarrow CS \cup SimilarityClass(attr, r.attr)$
13                 end
14             else $CS \leftarrow GS$
15         otherwise /*attr is defined (not missing)*/
16             $CS \leftarrow CS \cap SimilarityClass(attr, res.attr)$
17         end
18     endsw
19 end
```

In order to parametrize the missing data the U-GovOps framework first tries the assignments from *missing_data* directive. The permissible values to assign to the missing attributes include '?', '∗' and '−'. The '?' is used to denote that the attribute value might be lost and '∗' or '−' mean that the user suspects that the attribute values were unavailable in the first place. If no user-provided parameter exists for an attribute, U-GovOps will associate it with the '?' by default. Although straightforward, this process has a significant impact on the framework's decisions how to compute the the governance scope. For example, assigning the '?' to a device's attribute instructs U-GovOps not to include that device in any Similarity Classes for such attribute. Further, the '∗' tells U-GovOps that the original values were irrelevant, thus can be considered as any value consistent with that attribute. Finally, the '$-$' tells the framework that these missing values can be considered as any value consistent with that concept, as discussed in [61, 137, 84].

To calculate the characteristic set for a resource, e.g., a device, the U-GovOps framework performs the calculation as shown in Algorithm 9.1. The intuition behind the algorithm is to enable determining similar resources, under attributes G with missing information, by considering problem-dependent uncertainty parametrization. Please note that the shown algorithm is meant to demonstrate the main calculation steps and it is not necessarily optimized for performance. Finally, based on the specified `selection_strategy` the U-GovOps returns a governance scope. For example, for optimistic selection strategy the governance scope, to be returned, is calculated as upper approximation of the targeted scope X with: $\overline{G}X = \bigcup\{CS_G(r) \mid r \in X\}$, where CS is a characteristic set for a resource r and G is the specified attribute set.

Actuating under uncertainty

Figure 9.3 outlines the most important steps performed by U-GovOps to support the isolated actuations (we omit loops, caching, error handling, etc., for readability purposes.). This mechanism is triggered when a user submits a policy (e.g., as shown in Listing 9.2) to U-GovOps for execution. A user only observes the invocation calls and the returned results (shown hatched in Figure 9.3). The other steps are performed by the framework, transparent to the users.

Initially, the U-GovOps framework resolves the rough governance scope and creates a policy context, which stores the uncertainty parameters (supplied by the user), the computed governance scope and the policy invocations cache. The subsequent steps are mainly determined, by the user-provided uncertainty configuration directives (Listing 9.2). If the `run_in_isolation` is set to true, U-GovOps isolates the policy by allocating a dedicated resource pool for it. In the current prototype this is realized by instantiating a dedicated thread pool (per policy) and performing all policy actuations (on separate threads) within that thread pool. However, other concepts such as Actor Model could be used instead. In case a policy should not be executed in isolation, individual actuations will still remain isolated, but they will share the same global resource pool.

The Policy Monitor (Figure 9.3) implements the circuit breaker and continuously monitors the threads and the thread pools (policies) for the aforementioned conditions such as the permissible fault percentage and keep alive timeouts. Currently this is implemented based on the Netflix OSS Hystrix, since U-GovOps uses HTTP as the underlying protocol for Remote Procedure Calls. Also here exist alternatives such as Twitter Finagle or Google gRPC. If the constraints are violated, the Policy Monitor trips the circuit, denying the further resources to that policy and temporally putting its execution on hold or interrupting its execution if keep alive expired. Generally, this or an actuation failure will trigger the execution of the fallback logic (`if fallback_count>0`). Currently, the U-GovOps framework only provides a rudimentary support for specifying the fallback

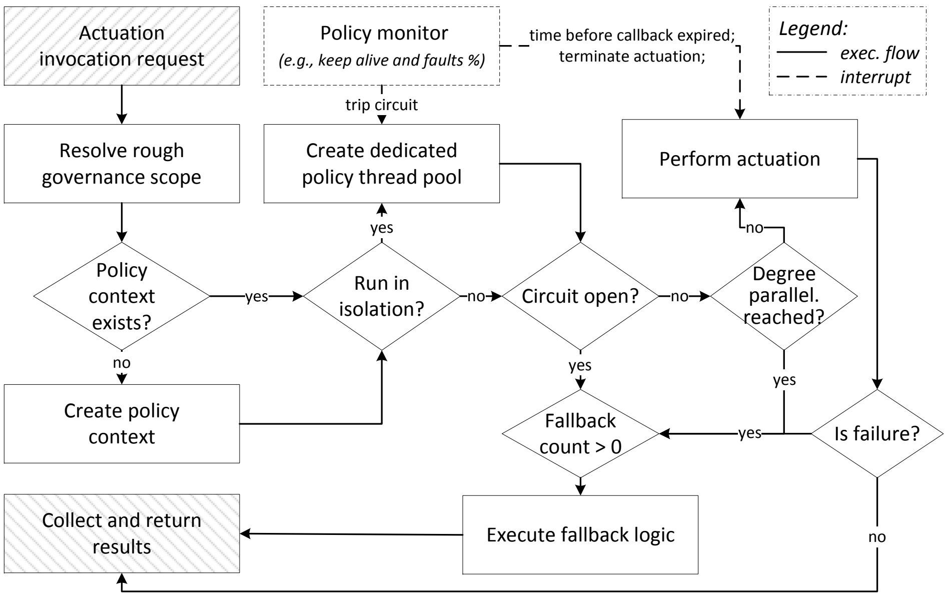

Actuation invocation request
Policy monitor
(e.g., keep alive and faults %)
time before callback expired; terminate actuation;
Legend:
exec. flow
interrupt
trip circuit
Resolve rough governance scope
Create dedicated policy thread pool
Perform actuation
yes
no
Policy context exists?
yes
Run in isolation?
no
Circuit open?
no
Degree parallel. reached?
no
yes
yes
Create policy context
Fallback count > 0
yes
Is failure?
Collect and return results
Execute fallback logic
no

Figure 9.3: Execution flow for isolated actuations.

logic, by retrying the normal flow or returning a generic error if everything else fails. In the future we plan to address this and allow injecting custom fallbacks. Finally, U-GovOps collects the actuation results (if any) and returns them to the calling policy. through the utilization of Futures and Promise pipelining, which enable asynchronous result processing with minimal latency.

9.5 Evaluation

In this section, we present the preliminary results of our experiments. Our experiments comprise two general parts. First, we perform a functional evaluation of U-GovOps's language support for implementing *uncertainty- and elasticity-aware governance policies*, based on our real-life use case (Section 9.1). Second, we evaluate U-GovOps's main runtime mechanisms for mitigating runtime infrastructure uncertainties.

9.5.1 Experiments setup

In order to evaluate how our framework behaves under uncertainty, we created a testbed for virtualized IoT cloud systems using CoreOS. We used Docker containers to virtualize and mimic physical gateways in the cloud. These containers are based on a snapshot of a real-world, proprietary IoT gateway. The Docker base image is publicly available in Docker Hub under dsgtuwien/govops-box.

For the subsequent experiments we deployed the testbed on our local OpenStack cloud, running approximately 1000 Docker containers (simulating the gateways/nodes).

Each of the containers “hosts" different virtual sensors (e.g., location) and are associated with different meta data (e.g., owner). These sensors replay the prerecorded real-life data, obtained in our case study. Since the main aim is to govern the infrastructure services and resources, we only consider the infrastructure state data relevant for the governance policies and not the data used by the business logic cloud services (although these might overlap). The U-GovOps controllers and the demo application (Section 9.1) are deployed separately, in the same cloud on 4 Ubuntu 14.04 VMs (with 2VCPUs and 3GB of RAM) and used to execute our governance policies. Finally, to simulate the uncertainties, i.e., the *missing or incomplete data* (about the infrastructure states) and *actuation uncertainties*, we developed three mechanisms (based on Dell Blockade[4]), which perform random fault injections: (i) killing of the containers, (ii) dropping of data packets and, (iii) slowing down the network.

9.5.2 Example governance policy implementation

We first show how U-GovOps language is used to develop the real-life governance policy for the PMA application, presented in our case study (Section 9.1). Listing 9.4 shows the complete source code of the governance policy. Since it mostly uses the familiar language concepts, presented earlier in the chapter, we refrain from explaining the individual steps and instead focus on the most important features of our language.

We notice that a user utilizes intuitive, high-level abstractions and configuration directives to declare what needs to be done instead of specifying how to do it (e.g., Listing 9.4, lines 4-7). For example, a user does not directly invoke the individual actuations nor has to explicitly handle actuation failures or recovery logic, since the actual

```
1    G1: GOVERNANCE_SCOPE query: location=building3&type=JACE-545&owner=TUW
2          CONSIDERING_UNCERTAINTY: missing_data=location<='?', owner<='*'
3                                  AND selection_strategy=optimistic;

4    M1: MONITORING abnormal_behavior := sensorAlert(G1)==true OR
5      heartBeatAVG(G1)>5min;

6    S1: STRATEGY CASE abnormal_behavior: setProtocol('mqtt'),
7      changeUpdateRate('5s') FOR G1
8          CONSIDERING_UNCERTAINTY: run_in_isolation=true AND
9          keep_alive=1min AND
10         fallback_count=2 AND
11         tolerate_fault_percentage = 20% AND
12         invocation_caching=true;

13   C1: CONSTRAINT cost<200 CONSIDERING_UNCERTAINTY: decision_confidence >=20%;

14   S2: STRATEGY CASE responseTime>250ms: scaleOut()
15              CONSIDERING_UNCERTAINTY:considering_strategies = S1;
```

Listing 9.4: Example PMA governance policy.

[4]https://github.com/dcm-oss/blockade

invocations are pushed down to U-GovOps, who transparently handles lost actuations and prevents cascading failures, based on the user-provided configurations. Further although our framework limits the expressiveness to a certain extent, the users can still express many common behaviors of governance strategies. For example, the user can easily specify the desired elasticity behavior, taking into account possible uncertainties caused by related actions (lines 8-9). Finally, our framework simplifies the user effort in dealing with the data uncertainties (lines 1-2), since the users do not have to write complex queries or explicitly deal with False Positive (FP) and False Negative (FN) results.

9.5.3 Experiments results

Next, we evaluate main U-GovOps runtime mechanisms: *resolving rough governance scopes* and for *isolating the actuations* under presence of two main uncertainties: *missing or incomplete data* and *actuation uncertainties* (simulated as described above). The experiment results are averaged on 50 repetitions and we have experimented with 7 different governance policies, which have different properties regarding query complexity and actuation types (e.g., execution time and computational complexity).

Table 9.1: Averaged F1 scores for the governance scopes.

Percentage of the missing data	10%	20%	30%	40%	50%
F1 scores - optimistic strategy	0.95	0.86	0.86	0.80	0.74
F1 scores - pessimistic strategy	0.90	0.90	0.80	0.79	0.72
F1 scores - no uncertainty consideration	0.91	0.80	0.66	0.50	0.29

To evaluate the coverage of our governance policies, i.e., the “goodness" of approximation of our governance scopes under uncertainty (missing data) we show two relevant metrics: the F1 scores and the error rates, as cumulative metric for the FPs and FNs. The baseline is calculated with “perfect information" (no missing data) and then we repeated the policies execution, while simulating the data losses. Table 9.1, shows the resulting averaged F1 scores. The missing data represents the percentage of missing data instances randomly distributed across the resources and the resource attributes. We run 3 different setups: not considering the data uncertainty (i.e., ignoring the missing data), using optimistic selection strategy and using the pessimistic selection strategy. The corresponding error rates are shown in Figure 9.4.

It is important to notice (Figure 9.4) that not considering uncertainties and pessimistic strategy only contain FNs (i.e., resources that should be included in the governance scope, but were not due to the lack of information), while optimistic strategy only returns FPs (i.e., includes the desired resources with certainty). This shows an important property of

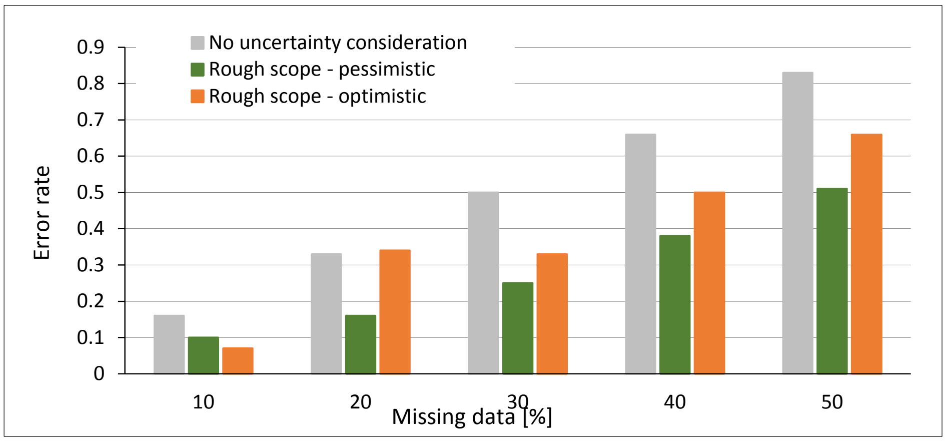


Figure 9.4: Error rates for governance scopes due to missing data.

our approach, i.e., it enables users to make trade-offs depending on the task-at-hand. For example, governance policies that do not care about FPs (formulated as: “ALL resources with specific properties MUST be included") can be easily specified with optimistic selection strategy. Additionally, compared to traditional approaches (no uncertainty consideration) our pessimistic selection strategy generally behaves better, i.e., displays on average about 20% less errors. Finally, it is worth noting that parametrization of missing data (different combinations of ‘?’, ‘$*$’ and ‘$-$’) had a significant impact on the quality of the results. This can be considered a drawback, since it steepens the learning curve of U-GovOps language. In the future we plan to explore this phenomenon in order to derive suitable heuristics for parameterizing the missing data in governance scopes.

Figure 9.5 shows the percentages of lost actuations, with and without U-GovOps mechanism for isolated actuations, for different fault rates. For example, for fault rate of 10% we know that 10% of all actuations will be affected by at least one of the 3 fault injection actions. For all the evaluated policies we use the same base-line, e.g., number of containers ($\approx$ 1000), configurations, etc. The isolated actuations are configured in a greedy fashion, with main objective to mitigate as many uncertainties as possible. Generally, by isolating the actuations we managed to reduce the rate of lost actuation by more then 50% on average, compared with the traditional approaches, which do not consider uncertainties. The majority of unaccounted uncertainties were due to the killed containers, since it is currently not possible to compensate this with U-GovOps.

On the secondary axes (Figure 9.5), we show the average execution time of the governance policies. We notice that average execution time of the policies without uncertainty consideration was only slightly affected by the faults, mainly due to network slowdowns. On the other hand, our approach had an exponential increase in execution time with high fault rates. This is mainly due to the exponential back-off policy implemented by the framework in order to be “fair" to the underlying actuators, i.e., not overload them with requests, e.g., in case of major network problems. This shows an important

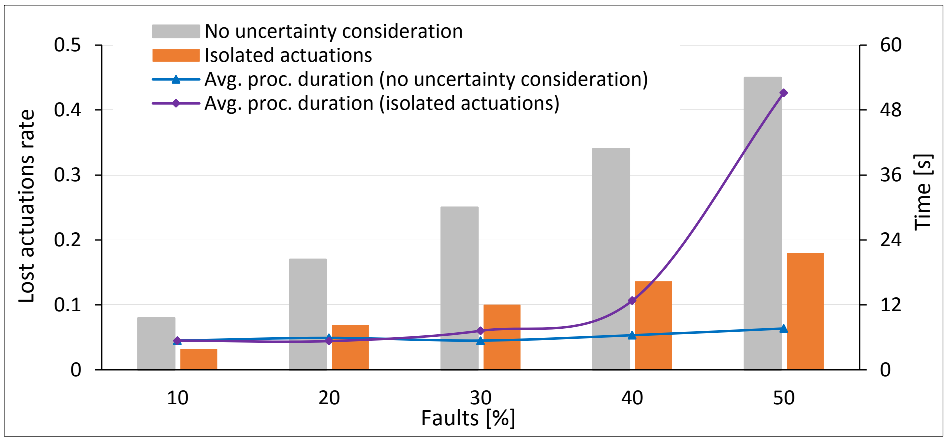


Figure 9.5: Lost actuations rates for isolated actuations.

property of the uncertainty management that it does not come “for free", in the sense that users need to accept some overhead, e.g., of performance or additional costs, in order to account for the uncertainties. Naturally, the users can control these aspects by relaxing the uncertainty constraints.

We are also aware of certain limitations of our framework. For example, maintaining the threads and the thread pools per actuation/policies causes an additional computational overhead, due to thread queueing, scheduling, and context switching. We deliberately decided to make a tradeoff here, since we believe that the overall advantages of having more resilient and fault tolerant governance overweight the additional costs in the long run. Finally, currently U-GovOps mainly focuses on runtime infrastructure uncertainties and does not explicitly consider cumulative effects and uncertainty propagation. This is, however, subject of our future work.

9.6 Conclusion

In this chapter we introduced the U-GovOps framework for governing elastic IoT cloud systems under uncertainty. We presented the U-GovOps *declarative policy language* for developing uncertainty- and elasticity-aware governance policies. The main U-GovOps runtime mechanisms for *managing rough governance scopes* and *enabling isolated actuations* were introduced to facilitate enforcing such polices, by effectively mitigating the infrastructure uncertainties, as demonstrated on a real-life case study. The initial results are promising in the sense that with the U-GovOps framework users can develop custom governance strategies efficiently, by using intuitive, high-level abstractions and configuration parameters without explicitly dealing with the uncertainties of complex interactions in IoT cloud infrastructure.

CHAPTER 10

Related work

10.1 Programming Support for IoT Cloud applications

Developing and managing IoT Cloud systems and applications have been receiving a lot of attention lately. In [160, 42, 64] the authors mostly deal with device virtualization and its management on cloud platforms. A number of different approaches (e.g., [134, 3]) employ semantics aspects to enable discovering, linking and orchestrating heterogeneous IoT devices. In [29, 85] the authors propose utilizing cloud for additional computation resources and approaches presented in [139, 162] focus on utilizing cloud's storage resources for sensory data. Approaches presented in [39, 83] deal with integrating IoT devices and services with enterprise applications based on SOA paradigm. These approaches mostly adopt a cloud-centric view on IoT Cloud applications development. For example, in [42] the authors focus on developing a virtualized infrastructure to enable sensing and actuating as a service on the cloud. They propose a software stack that includes support for management of device identification and device services aggregation. In [160] the authors introduce sensor-cloud infrastructure that virtualizes physical sensors on the cloud and provides management and monitoring mechanisms for the virtual sensors. Although, such approaches facilitates development of IoT Cloud applications to a certain extent, contrary to our approach they usually do not define a structured programming model for developing such applications. Another example of cloud-centric approach is SenaaS [3]. SenaaS mostly focuses on providing a cloud semantic overlay atop physical infrastructure. It defines an IoT ontology to mediate interaction with heterogeneous devices and data formats, exposing them as event streams to the upper layer cloud services. Similarly, the OpenIoT framework [134] focuses on supporting IoT service composition by following cloud/utility based paradigm. It mainly relies on semantic web technologies and CoAP to enable web of things and linked sensory data. Such approaches can be seen as complementary to our own, as abstracting the IoT devices sets the cornerstone for developing IoT Cloud applications. Our programming model relies on the contemporary advances in IoT Cloud and extends them with novel programming abstractions which enable everything-as-code paradigm, facilitating development of IoT Cloud applications and making the entire development process traceable and auditable (e.g., with source control systems), thus improving maintainability and reducing development costs.

Putting more focus on the edge devices, i.e., IoT gateways, network devices, cloudlets and small clouds, different approaches have emerged recently. For example, in [17] the authors present a concept of fog computing and define its main characteristics, such as location awareness, reduced latency and general QoS improvements. They focus on defining a virtualized platform that includes the edge devices and enables running custom application logic atop different resources throughout the network. Further, in [62] the authors focus on abstracting devices as service and enabling two-way communication between enterprise applications and devices via Web Services (WS) and provide mechanisms for service discovery and provisioning. Similar approach is DPWS [113], i.e., SOA4D or WS4D. Also, approaches utilizing RESTful protocols, CoAP[55] and sMAP [38] exist. For example, [83] focuses on defining a CoAP-based runtime to enable composing IoT services. Most of these approaches focus on abstracting underlying hardware and providing service-based access to a device. Although, they provide some key elements, e.g., service discovery and resource management, they implicitly assume developers have a good understanding of the underlying domain, as raw sensory data streams and low-level device services are directly exposed to them and application development is envisioned by composing the atomic services into admissible control sequences or processing schemes. Compared to these approaches our programming model defines high-level abstractions which enable development of cloud-scale IoT Cloud applications.

Another edge-centric approach is usage of component-based frameworks [77], [14] to abstract devices or more precisely to create proxies, which are represented as components and enable remote communication with the devices. These frameworks use OSGi for component management and execution environment. They share similarity with our approach, regarding usage of component-based architecture. However, they abstract devices as components and define a local component model and their applications operate on a residential gateway scale. Contrary to these approaches, our programming frameworks supports development of applications that seamlessly utilize both the Edge and the Cloud.

Some of the related approaches in ubiquitous computing and context-awareness are [126] and [135]. In [135] authors adopt a definition of task as representation of user's everyday activities. They focus on assisting the users during these activities and managing the resources in smart environments. Although, we share some similarities, regarding task as a generic activity, they don't introduce abstractions with a generic view on scopes, needed to enable development of IoT applications in a scalable manner. In general compared to the aforementioned edge-centric approaches, our approach also aims at better utilization of the edge infrastructure, but we also focus on providing a systematic approach, supporting application developers to address most of the application/infrastructure provisioning and governance issues programmatically, in a logically centralized fashion, by offering the software-defined gateways and well-defined provisioning and governance APIs.

Another related field is macroprogramming of sensor networks [93, 30, 101, 25]. For example, in [93] the authors provide an SQL-like interface where the entire network is abstracted as a relational database (table). Contrary to their approach, we utilize more general set theory to define operations on our IntentScopes. This gives more flexibility to developers, since our framework also allows dynamic, custom properties to be included in scope definitions, but comes at the cost of additional performance overhead. Similarly, in [101], the authors deal with enabling dynamic scopes in WSN, mainly addressing the important issues of task placement and data exchange (among the WSN nodes), in order to account for the heterogeneity of the nodes and enable logically localized interactions. Their approach can be seen as conceptually complementing our own, since task allocation and such interaction types are not the main focus of our framework. In [30], the authors propose the notion of logical neighborhood. Their approach is based on logical nodes (templates), which enable instantiating and grouping the nodes, based on their exported attributes. To facilitate communication within the neighborhoods, which is of greater importance in WSN, they also provide an efficient routing mechanism. In [25] the authors introduce an extensible programming framework that unifies the WSN programming abstractions in order to facilitate business processes orchestration with WSN. Despite the relevant efforts to integrate provisioning and business logic (e.g., template-based customizations [30]), the main focus of the aforementioned approaches is application business logic, while we address a more general problem of enabling everything-as-code paradigm, in order to also allow for capturing provisioning and governance logic for IoT Cloud resources programmatically.

10.2 Provisioning Approaches in IoT Cloud

Over the last years, advancing the convergence of Edge (IoT) and Cloud computing has been receiving a lot of attention. This has resulted in a number of approaches which lay a cornerstone for realizing the utility-based provisioning in IoT Cloud. For example, different approaches deal with leveraging more powerful resources such as remote, fully-fledged Clouds or smaller Cloudlets and micro data centers, which are located in the proximity (single hop away) of the Edge, to enhance resource-constrained (mobile) devices. Such approaches, also referred to as cyber-foraging systems [89], mainly focus on specific tasks such as computation offloading [36, 29, 85] or data offloading (data staging) [7, 53, 162, 139]. Although, they offer valuable insights about moving cloud computing closer to the Edge, as well as about smart resource utilization, management and allocation, contrary to our approach they mainly emphasize on algorithms (e.g., solvers), energy efficiency, performance (e.g., of processing or networking) and supporting architectures for the aforementioned tasks.

Other approaches which mainly adopt a cloud-centric view, mostly aim at virtualizing Edge devices, predominantly sensors and actuators, on cloud platforms. In [42] the authors focus on developing a virtualized infrastructure to enable sensing and actuating as a service on the cloud. They propose a software stack that includes support for management of device identification and device services aggregation. In [43], the same authors discus a utility-oriented paradigm for IoT, explicit claiming the resource virtualization and abstraction as their main goal. In [160] the authors introduce sensor-cloud infrastructure that virtualizes physical sensors on the cloud and provides management and monitoring mechanisms for the virtual sensors. In [64] the authors develop an infrastructure virtualization framework for wireless sensor networks. It is based on a content-based pub/sub model for asynchronous event exchange and utilizes a custom event matching algorithm to enable delivery of sensory events to subscribed cloud users. Also the previously-described approaches the SenaaS [3] and the OpenIoT framework [134] provide some support regarding the IoT Cloud provisioning. However, their support is mainly focused on high-level application provisioning aspects such as discovering, linking and orchestrating internet connected objects and IoT services. Finally, there are various commercial solutions such as Xively [156], Carriots [24] and ThingWorx [141], which allow users to connect their sensors to the Cloud and enable remote access to and management of such sensors. The aforementioned approaches mainly focus on providing different virtualization, device interoperability and semantic-based data integration techniques for IoT Cloud. Therefore, such approaches conceptually underpin our middleware, since virtualizing Edge devices is a main precondition towards realizing utility-based provisioning paradigm in IoT Cloud systems, as we illustrate in Figure 1.1. Although, some of the above-described solutions (e.g., [160, 134, 42]) provide support for provisioning and management of virtual sensors and actuators, their support is often based on tightly-coupled provisioning models, e.g., static templates. Moreover, such approaches are usually meant to support specific data-centric tasks, mostly focusing on integrating various data formats, providing data-linking solutions and supporting communication protocols. Contrary, to these approaches our middleware provides support for multi-level provisioning and consuming both IoT and Cloud resources as general-purpose utilities.

Putting more focus on the network virtualization, programming and management, two prominent approaches have recently appeared, namely software-defined and fog computing. Different approaches have exploited and extend software defined concepts to facilitate utilization and management of the pooled sets of shared IoT Cloud resources, e.g., software-defined storage [140] and software-defined data center [37]. Advances in more traditional software-defined networking (SDN) [80, 78, 76] have enabled easier management and programming of the intermediate network resources, e.g., routers, mostly focusing on defining the networking logic, e.g., injecting routing rules into network elements. In [17] the authors present a concept of fog computing and define its main characteristics. Although the general idea of fog computing shares similarities with our approach, there is still a number of challenges to realize its full vision [157]. Further, current advances in fog computing mainly revolve around virtualization, management and programmatic control of the network elements. Although provisioning of network resources is not the focus of our middleware, these approaches can be seen as complementary to our own approach, since the network resources are an integral part of IoT Cloud infrastructures (cf. Figure 1.1).

Finally, since the utility-based provisioning paradigm originated from cloud computing, it is natural that cloud computing has provided numerous tools and frameworks to support the utility-based provisioning. The relevant approaches are centered around infrastructure automation and configuration management solutions such as OpsCode Chef [118], BOSH [18] and Puppet [124] as well as deployment topology orchestration approaches such as OpenStack Heat [115], AWS CloudFormation [11] and OpenTOSCA [117]. The main reasons why these solutions cannot be simply reused in the context of IoT Cloud systems are that they mostly assume unlimited amount of available resources; they do not account for intrinsic dependance of application business logic on underlying devices; they are usually not suited for constrained environments and they often rely on features provided only by fully-fledged OS, e.g., configuration management approaches often hand off dependency resolution to OS package managers.

10.3 IoT Cloud Governance

The IoT governance has been receiving a lot of attention recently. For example, in [152] the author evaluates various aspects of the IoT governance, such as privacy, security and safety, ethics, etc., and defines main principles of IoT governance, e.g., legitimacy and representation, transparency and openness, and accountability. In [151], the authors deal with issues of data quality management and governance. They define a responsibility assignment matrix that comprises roles, decision areas and responsibilities and can be used to define custom governance models and strategies. Traditional IT governance approaches, such as SOA governance [13, 27, 111] and governance frameworks like CMMI [2], the 3P model [128], and COBIT [63], provide a valuable insights and models which can be applied in GovOps processes, usually without substantial modifications. Compared to these approaches, GovOps does not attempt to define a general methodology for IoT Cloud governance. Therefore, such approaches conceptually do not conflict with our approach and they can rather be seen as methodologies and techniques complementing GovOps.

Further, numerous government organizations and standardization bodies deal with IoT Cloud governance. The governance concepts have been already applied on different Internet aspects and there is a range of organizations such as IETF, ICANN, RIRs, ISOC, IEEE, IGF, W3C, which are dealing with specific areas of Internet governance. The EU Commission has also created task forces, research clusters and reports, which deal with the governance issues in IoT [51, 50, 52]. They have identified several challenges in contemporary IoT Cloud governance. For example, the difficulty to find a common definition of IoT governance together with the different positions of many stakeholders. Also, due to the high number and heterogeneity of technologies and devices in the IoT systems, IoT governance requires even more specific solutions compared to the traditional governance solutions. Moreover, current approaches in IoT governance usually addresses the Internet part of the IoT, e.g, in the context of the Future Internet services, while operations processes mostly deal with Things as additional resources that need to be operated. Although, there are approaches that facilitate operating the Edge devices (e.g., [160, 43] as we discussed in the previous section), mapping the governance objectives (law, compliance, etc.) to operations processes largely remain elusive to the contemporary governance approaches. The GovOps model builds on these approaches and addresses the issue of bridging the gap between governance objectives and operations processes, by introducing the GovOps manager as a dedicated stakeholder, as well as defining the suitable GovOps reference model to support early integration of governance objectives and operations processes. For high-level business stakeholders, GovOps enables continuous analysis, verification, and improvement of governance objectives and implemented strategies using a systematic approach. Furthermore, implementing the GovOps approach enables technological advantages such as greater flexibility, reduction of time-to-delivery, improved ease of operation, and shielding operations from regulatory issues.

Although, to our best knowledge there are no related approaches in the literature that deal with uncertainty issues in IoT Cloud governance, a number of approaches exist which address system uncertainties and faults. In the field of self-adaptive systems (SAS) there are many approaches dealing with uncertainties and faults. For example, in [125], the authors present a taxonomy of uncertainty for dynamically SAS. Whittle et al. [154] developed RELAX, a textual requirements language that provides fuzzy logic-based operators to facilitate the specification of uncertainties in SAS at requirements level. Weyns et al. [153] introduced FORMS, a formal reference model for self-adaptation that builds upon feedback loops to enable addressing uncertainties at design level. The runtime uncertainties are addressed in [56], mainly using the concept of reactive feedback loops. Such approaches conceptually complement our own, by providing valuable insights and techniques to understand and analyze uncertainties. However, the distinct feature of our operational governance approach is that it considers both elasticity and uncertainty at the level of governance policies. Approaches form Wireless Sensor Networks (WSN) also deal with uncertainties, e.g., [132, 94]. However, they mostly deal with sensor network deployments and detecting redundant sensors in the WSN. Contrary to such approaches, to our best knowledge, our GovOps approach is the first attempt to enable developing uncertainty- and elasticity-aware governance strategies encompassing both IoT and cloud infrastructures.

CHAPTER 11

Conclusion & Research Outlook

In this chapter, we reflect on the main results of the research conducted during the course of this thesis. In Section 11.1 we summarize the main contributions presented in the thesis and provide final remarks. Section 11.2, revisits the main research questions formulated in Section 1.1 and discusses how and to what extent the presented contributions address these research questions. Finally, Section 11.3 concludes the thesis and gives an outlook of open topics for the future research that can be build on this thesis' contributions.

11.1 Summary of Contributions

This thesis tackled a series of timely and relevant issues hindering the development and operation of IoT Cloud systems. The presented thesis contributions advance the state of the art in programming, provisioning and governing IoT Cloud systems, by introducing a rich ecosystem comprising novel models, frameworks and middleware for novel IoT Cloud systems. The thesis introduced three main contributions in the emerging field of IoT Cloud, which are presented coherently, resembling the three main parts of the thesis:

First part of the thesis presented the first main contribution that deals with programming IoT Cloud systems. The main problems addressed by this contribution include: (i) Enabling development of generic IoT Cloud applications, which seamlessly utilize both the Edge and the Cloud resources; (ii) Supporting the variety of involved developer roles; and (iii) Accounting for the complexity of software stack, ranging from resource-constrained Edge devices development to high-level Cloud services. To systematically address these problems the contribution was divided into three parts, which were presented in chapters 3, 4 and 5, respectively. Chapter 3 presented a high-level programming model and a runtime for developing cloud-centric IoT Cloud applications. The programming model introduced programming constructs (*Intents and IntentScopes*) and operators, which raise the level of programming abstraction, enabling developers to implement IoT Cloud applications without worrying about the diversity and complexity of the underlying Edge devices. We also presented a supporting runtime framework, which provides a cloud-based application execution environment and a set of mechanisms, which enable loosely-coupled communication with the Edge devices. Chapter 4 presented a programming model and a runtime for resource-constrained Edge devices, e.g., gateways. We discussed the main programming abstractions introduced by the programming model, namely *the Data and Control Points*, which are intended to support domain expert developers in developing common monitor and control tasks for Edge devices. An application runtime was presented that provides mechanisms which act as multiplexers of the data and control channels, providing a virtually exclusive access to the underlying devices, thus enabling the edge-device applications to have their own view of and define custom configurations for such channels. In Chapter 5 we introduced SDG-Pro – a unifying programming framework for IoT Cloud systems, based on everything-as-code paradigm. The main problem addressed by the SDG-Pro framework is a lack of programming support to account for complex and strong dependence of application business logic on specific capabilities and features of the IoT devices. To address this problem, the presented framework combines the Intents with the Data and Control Points to provide a uniform support for application business logic development. Further, it provides additional support for programmatic provisioning and governance of IoT Cloud systems, unifying it with the support for application business logic development. The SDG-Pro framework mainly focuses on defining programming support, but it relies on the provisioning and governance models and techniques, which are developed in the other two contributions of the thesis. Finally, we showed that the introduced approach is designed in such manner to provide multiple logical views on the application development process, while retaining a uniform view (in code) on the produced application artifacts.

In the second part of this thesis the focus was shifted from application-level support to runtime middleware and tooling support for operating IoT Cloud systems. The work presented in this part of the thesis was mainly motivated by a stringent need: To enable refactoring the IoT Cloud infrastructure into finer-grained resource components whose behavior can be defined in software; To provide conceptually unified representation of both Edge and Cloud resources; As well as to enable automated and scalable management of IoT Cloud resources, application components and their configuration models in a logically centralized fashion. To address these problems, the second contribution of this thesis comprises two main parts that were presented in Chapter 6 and Chapter 7, respectively. Chapter 6 introduced a unified provisioning model and a framework support for logically centralized provisioning of IoT Cloud systems. We showed how our provisioning model enables the IoT Cloud resources (e.g., virtual sensors and data point controllers), their runtime environments (e.g., IoT gateways) and configuration models (e.g., for communication protocols) to be descrribed as *software-defined IoT units.* Such units were introduced as the core concept of the provisioning model. We discussed how these units can be used to encapsulate the IoT Cloud resources and abstract their provisioning in software through managed APIs. A concept of unit prototypes was introduced to technically underpin the provisioning model. The unit prototypes are hosted in the IoT Cloud and enriched with provisioning capabilities (delivered by framework's provisioning agents), that allow them to be dynamically configured, composed and deployed. At this point, it is important to remind that the unit prototypes do not introduce novel virtualization solutions, but rely on proven technologies, namely kernel-supported virtualization to abstract the IoT Cloud resources. In Chapter 7, we introduced a middleware infrastructure for utility-based provisioning IoT Cloud systems, which conceptually extends and technically refines the first part of this contribution. We presented the middleware's main components: a cloud-based provisioning controller and edge-based provisioning agents and deamons. Further, we discussed the main runtime mechanisms of the provisioning middleware: i) A light-weight mechanism for resource abstraction (based on the unit prototypes), which allow for application-specific customizations of IoT Cloud resources; ii) Support for automated provisioning and management of infrastructure resources, application components and configuration models in a uniform, logically centralized manner through middleware-managed APIs and; iii) Extensible and flexible provisioning models, which support on-demand consumption of the Edge-device resources. We discussed how our provisioning middleware provides a comprehensive support for multi-level provisioning of IoT Cloud systems, in order to support execution of provisioning processes that are based on the previously-introduced provisioning model. It was shown how the controller architecture and the provisioning mechanisms are specifically tailored to account for the large-scale of IoT Cloud, but also for the resource-constrained nature of Edge devices. Finally, we discussed how our provisioning approach: enables logically centralized point of operation in IoT Cloud system; facilitates fine-grained on-demand resource consumption; allows for automating the provisioning processes, making them easily repeatable; and supports elasticity scalable execution of such processes, which are some of the main preconditions for provisioning large-scale, geographically-distributed systems.

The last part of this thesis dealt with governance in IoT Cloud systems. We identified critical problems in contemporary IoT Cloud governance: (i) A wide gap between the main stakeholders involved in governing IoT Cloud systems; and (ii) enforcing governance strategies in a large-scale, geographically distributed systems in a time-consistent manner. To address these problems, the third contribution of this thesis comprises three main parts, presented in Chapter 8 and Chapter 9. Firstly, Chapter 8 introduced *GovOps* – a methodology and a reference model for operational governance processes. The main objective of GovOps is to bring business stakeholders and operations managers closer together, making a step forward in bridging the gap between governance objectives (e.g., standards and regulations) and supporting operations processes. GovOps introduced a governance model and a design methodology for operational governance processes, in order to enable seamless integration and alignment of the high-level governance objectives with executable operations processes from early designing stages. We showd how the GovOps model builds on the software-defined IoT unit model, extending it among other things, with a concept of governance capabilities that encapsulate governance operations which can be dynamically applied on such units during runtime. We also introduced GovOps manager role, responsible to guide and oversee the design of the operational governance processes. Secondly, this chapter also introduced the *rtGovOps framework* that serves as GovOps reference implementation, providing support for designing and executing operational governance processes. We presented rtGovOps' main runtime mechanisms and enabling techniques that support GovOps managers to handle two main tasks: (i) perform dynamic, on-demand provisioning of governance capabilities and (ii) remotely invoke such capabilities in IoT Cloud, via dynamic APIs. We demonstrated, on a real-world case study, the feasibility of GovOps methodology and framework to facilitate execution of operational governance processes in large-scale IoT Cloud systems. Finally, Chapter 9 introduces an uncertainty extension for GovOps. The main motivation for the last part of this contribution was to enable mitigating uncertainties inherent to operational governance processes, mainly caused by the novel interactions of Edge devices, network elements, Cloud resources and humans. This refined U-GovOps framework introduced a declarative policy language and its runtime in order to enable development of uncertainty- and elasticity-aware governance processes. The main U-GovOps runtime mechanisms for *managing rough governance scopes* and *enabling isolated actuations* were introduced to facilitate enforcing such polices, by effectively mitigating the infrastructure uncertainties, as demonstrated on a real-life case study.

Generally, the thesis has striven to achieve a fair balance between formal, systematic problem abstractions and concrete technology mappings, with runnable prototypes. Most of the developed prototypes have been provided to the community as open source frameworks or middleware. Each contribution has been rigorously evaluated on the developed proof-of-concept prototypes and representative real-life scenarios. In generally, the quantitative experiments were mainly designed to evaluate the implemented prototypes with respect to two distinct performance requirements. On the one side, they aimed to prove scalability of the introduced mechanisms and algorithms. On the other side, they needed to validate the the prototypes' suitability for resource-constrained devices, in terms of their resource consumption requirements. Additionally, the presented programming models were evaluated qualitatively against the widely-accepted design requirements for programming models and languages. For the evaluation purposes, an IoT Cloud testbed was developed, which combines physical IoT devices, which were built and installed in our department, with cloud-based simulated IoT devices, deployed on our private cloud infrastructure.

11.2 Revisiting Research Questions

In this section, we discuss the research questions formulated in Section 1.1 and reflect on how and to what extent this thesis has addressed them:

- *Q1: What is a suitable programming model and methodology for developing IoT Cloud applications in an efficient, uniform and generic manner?*

  To respond to this general research question, in Part I of this thesis we introduced programming model and framework, specifically tailored for IoT Cloud systems. The presented programming model is designed to offer multiple logical views on the IoT Cloud application development process, while retaining a uniform view (in code) on the produced application artifacts. For this purpose we introduced suitable abstractions for cloud-centric applications, namely Intents and IntentScopes, which raise the level of programming abstraction, enabling developers to implement IoT Cloud applications more efficiently and intuitively, without worrying about the diversity and complexity of the underlying Edge devices. To support the domain expert developers in programming edge-centric IoT Cloud applications and services, the introduced Data and Control Points allow for multiplexing of low-level the data and control channels, providing a virtually exclusive access to the underlying devices. This enables edge-device applications to have their own view of and define custom configurations for such channels, thus supporting development of generic IoT Cloud applications. Besides supporting business logic development, the presented programming framework introduces additional support and provides a unified programmatic view on the entire development process (everything as code), by encapsulating most important aspects of IoT Cloud provisioning and governance.

We also recognize a number of limitations and shortcomings of the presented approach. Our current approach is mainly intended for one particular type of IoT Cloud applications, i.e., reactive applications, which are characterized by receiving (monitoring) information and as a response performing a sequence of (control) actions. However, to enable wider utilization of IoT Cloud applications, the proposed programming model needs to be extended to provide better support for both online and offline Big Data analytics. Further, our programming model provides only a rudimentary support for synchronous delivery of Intents to the Edge devices. In this context, a mechanisms which would enable more reliable, RPC-like communication with the Edge are needed, especially for time-critical tasks such as handling emergency situations. The approach also needs to be extended to provide support for automated mapping of the tasks to edge-devices, which can be mobile and utilized in an opportunistic fashion, but also to enable runtime task migrations among the Edge devices. Finally, although our approach provides support for programmatically controlling the physical entities, a number of issues still need to be addressed. For example, at the moment our programming framework only provides a rudimentary support for concurrent execution of control actions and conflicts resolution, based on static priority levels assigned to applications. In addition, considerations of actuator's physical limitations and safety-related issues of invoking an actuator need to be introduced at the middleware level, to relieve application developers from coping with such issues in ad hoc manner.

- *Q2: Which provisioning models, techniques and tools can be applied to enable on-demand, self-service provisioning of IoT Cloud resources at fine granularity?*

As a response to this research question, Part II of the thesis introduced a provisioning model for IoT Cloud system, based on the main software-defined principles and a middleware infrastructure for provisioning of IoT Cloud systems. The main objective of the provisioning model and middleware was to make a step forward towards enabling utility-based provisioning paradigm in IoT Cloud systems. To this end, initially (in Chapter 6) we laid out a road map towards utility-based provisioning of IoT Cloud systems, which identified the key challenges and introduced a set of design principles and technical enablers to address these challenges. The introduced provisioning model enables uniform representation of both IoT and Cloud resources, based on the concept of software-defined IoT units. We discussed how these units can be used to encapsulate the IoT Cloud resources at fine granularity levels and abstract their provisioning in software through managed APIs, in a unified manner. The introduced provisioning middleware provides a comprehensive support for multi-level provisioning of IoT Cloud systems, which facilitates provisioning of infrastructure resources, application artifacts and configuration models. Finally, we demonstrated the middleware capabilities to: enable logically centralized point of operation in IoT Cloud system; facilitate on-demand, self-service resource consumption; allow for automating the provisioning processes; and to support elasticity scalable execution of such processes, which are of the main challenges to enable utility-based provisioning of large-scale, geographically-distributed IoT Cloud systems.

Our approach lays a cornerstone towards realizing the vision of utility-based provisioning in IoT Cloud, but a number of challenges still remain. In the road map, we have identified resource monitoring and cost awareness as some of the main preconditions for enabling pay-per-use model. Although, our approach provides monitoring support, it mainly provides a coarse-grained information about resources usage and needs to be refined and extended to enable true utility-oriented, pay as you go resource consumption. Further, one of the main traits of utility-based consumption is autonomous and automated allocation of the consumed resources. Although our middleware provides good support for automating the provisioning tasks it still requires manual interactions and human-supported decision making to efficiently allocate the software-defined gateways on the underlying Edge devices. Therefore, support for smarter resource allocation is required, which would optimize placement of software-defined gateways and application artifacts on Edge devices, based on autonomous decisions with respect to dynamic properties of IoT Cloud infrastructure. Finally, the approach needs to be extended to address the mobility aspects of the Edge devices, especially focusing on the dependability issues related to the device mobility and mobility of software components, i.e., runtime migration of the software-defined gateways.

- *Q3: Which models, techniques and tools are required to achieve structured and systematic IoT Cloud governance?*

  Part III of this thesis aimed to respond to this research question, by introducing GovOps – a novel governance model and runtime framework for operational governance in IoT Cloud systems. We discussed how GovOps can bring business stakeholders and operations managers closer together, making a step forward in bridging the gap between governance objectives (e.g., standards and regulations) and the supporting operations processes. It was shown how GovOps enables structured integration and alignment of the high-level governance objectives with executable operations processes from early designing stages. Moreover, we showed that the supporting GovOps runtime framework facilitates realizing governance strategies in geographically-distributed IoT

Cloud systems, by supporting governance managers to perform dynamic, on-demand provisioning of governance capabilities and to invoke such capabilities in IoT Cloud remotely, in a centralized manner via dynamic APIs. We demonstrated how our approach allows for systematic, time-consistent enforcement of operational governance processes in large-scale IoT Cloud systems, making them more traceable and auditable, thus effectively reducing costs and potential business limitations due to inefficient governance.

Also here we identify several shortcomings of our approach. The GovOps approach should be extended with additional support to enable structured management of high-level governance objectives, beyond operational governance processes. This is out of scope of this thesis, but it is envisioned as one of the crucial future research directions in the context of IoT Cloud systems. Furthermore, although the runtime GovOps framework supports dynamic changes in deployment topologies of IoT Cloud systems to a certain degree (mainly those related to elasticity requirements), the current approach should be extended to support similar functionality related to other governance requirements such as “redirecting” sensory data streams to comply with legal regulations.

11.3 Future Work

This thesis proposed several models, frameworks and middleware to address crucial issues in programming, provisioning and governing IoT Cloud systems. In spite this, a number of challenges remain that were out of scope of the thesis. In the following we conclude the thesis with a summary of possible future research directions.

- The ever-stronger need to process and analyze the Big Data generated by different IoT Cloud systems (e.g., in the context of smart cities) calls for structured programming support for developing data-centric IoT Cloud applications. To this end, the future research is expected to build on the programming models presented in this thesis to extend its support for online and offline data analytics specifically tailored for large-scale data-centric IoT Cloud applications.

- Elasticity is increasingly becoming key enabling feature for many contemporary applications. We believe that novel IoT Cloud systems can significantly benefit from elastic computing, not only in the cloud, but also across the entire IoT Cloud resource pool. However, in this context besides technical also numerous governance challenges arise. In the future, we plan to extend the current provisioning and governance approaches to enable elasticity aspects for IoT Cloud systems, most notably to support elastic scaling of the software-defined gateways across entire IoT Cloud.

- Although, testing is one of the crucial tasks in application development lifecycle, it is out of the scope of this thesis. In the future we plan to continue our current research in testing the IoT Cloud systems and applications. We plan to utilize the benefits of our current approach, which besides developing application business logic, also supports programmatic provisioning and governance of IoT Cloud applications. Conceptually, this allows for systematic approaches to test the provisioning and governance processes (e.g., scripts, workflows and policies), however novel techniques are required to support defining and identifying the states and behavior of system under test (SUT).

- In the future, it is expected that the GovOps model will be refined and extended in several directions: First, we envision integration with existing high-level governance and accountability frameworks for managing governance objectives and coordinating decision making processes, but also development of novel governance approaches specifically designed for IoT Cloud. Second, due to novel interactions of Edge devices, Cloud resources and humans, we envision a number of extensions of our current governance approach to account for the novel interactions among various stakeholders, as well as for the ever-stronger entanglement of humans and technology.
- Finally, we plan to continue our line of research towards fully-fledged utility-based delivery/consumption of IoT Cloud resources, possibly in a market-like fashion. To achieve full automation and provide autonomy to the IoT Cloud resources, one of the key remaining challenges is enabling the IoT Cloud applications to autonomously “compensate" the infrastructure owners for using their resources, such as sensory data or computing power. To this end, we will explore novel billing and payment solutions based-on cryptocurrencies, extending them with comprehensive support for monetary micro-transactions (time- and size-wise), automated cash handling and scalable processing of business transactions.

Bibliography

[1] Gregory D Abowd, Anind K Dey, Peter J Brown, Nigel Davies, Mark Smith, and Pete Steggles. Towards a better understanding of context and context-awareness. In *Handheld and ubiquitous computing*, pages 304–307. Springer, 1999.

[2] Dennis M Ahern, Aaron Clouse, and Richard Turner. *CMMI distilled: a practical introduction to integrated process improvement*. Addison-Wesley Professional, 2004.

[3] Sarfraz Alam, Mohammad Chowdhury, and Josef Noll. Senaas: An event-driven sensor virtualization approach for internet of things cloud. In *NESEA*, 2010.

[4] Apache Software Foundation. Apache cassandra. `http://cassandra.apache.org/`. [Online; accessed Jun-'13].

[5] Stefan Appel, Sebastian Frischbier, Tobias Freudenreich, and Alejandro Buchmann. Eventlets: Components for the integration of event streams with soa. In *SOCA*, 2012.

[6] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[7] Trevor Armstrong, Olivier Trescases, Cristiana Amza, and Eyal de Lara. Efficient and transparent dynamic content updates for mobile clients. In *Proceedings of the 4th international conference on Mobile systems, applications and services*, pages 56–68. ACM, 2006.

[8] Taimur Aslam, Ivan Krsul, and Eugene H Spafford. Use of a taxonomy of security faults. 1996.

[9] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[10] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, et al. *Fundamental concepts of dependability*. University of Newcastle, Computing Science, 2001.

[11] AWS. CloudFormation. URL: `https://aws.amazon.com/cloudformation/`. [Online; accessed Feb.-2015].

[12] Victor Bahl. Cloud 2020: Emergence of micro data centers (cloudlets) for latency sensitive computing (keynote). In *Middleware 2015*, 2015.

[13] Muneera Bano, Didar Zowghi, and Naveed Ikram. Alignment between business requirements and services: the state of the practice. In *ICSSEA*, 2013.

[14] Jonathan Bardin, Philippe Lalanda, and Clement Escoffier. Towards an automatic integration of heterogeneous services and devices. In *APSCC*, pages 171–178, 2010.

[15] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[16] Ketan Bhardwaj, Sreenidhy Sreepathy, Ada Gavrilovska, and Karsten Schwan. ECC: Edge Cloud Composites. In *MobileCloud 2014*, pages 38–47. IEEE, 2014.

[17] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the Internet of Things. In *MCC workshop on Mobile cloud computing*, pages 13–16, 2012.

[18] BOSH. BOSH. URL: `http://docs.cloudfoundry.org/bosh/`. [Online; accessed Feb.-2015].

[19] Thomas Buchholz and Michael Schiffers. Quality of context: What it is and why we need it. In *In Proceedings of the 10th Workshop of the OpenView University Association: OVUA'03*, 2003.

[20] Nicola Bui, Angelo P Castellani, Paolo Casari, and Michele Zorzi. The internet of energy: a web-enabled smart grid system. *Network, IEEE*, 26(4):39–45, 2012.

[21] Nicola Bui and Michele Zorzi. Health care applications: a solution based on the internet of things. In *Proceedings of the 4th International Symposium on Applied Sciences in Biomedical and Communication Technologies*, page 131. ACM, 2011.

[22] BusyBox. BusyBox: The Swiss Army Knife of Embedded Linux. URL: `https://busybox.net/about.html`. [Online; accessed Jan.-2015].

[23] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation computer systems*, 25(6):599–616, 2009.

[24] carriots.com. Carriots–IoT Application Platform. URL: `https://www.carriots.com`. [Online; accessed Jan.-2015].

[25] Fabio Casati, Florian Daniel, Guenadi Dantchev, Joakim Eriksson, Niclas Finne, Stamatis Karnouskos, Patricio Moreno Montera, Luca Mottola, Felix Jonathan Oppermann, and Gian Pietro Picco. Towards business processes orchestrating the

physical enterprise with wireless sensor networks. In *ICSE'12*, pages 1357–1360, 2012.

[26] Marie Chan, Daniel Estève, Christophe Escriba, and Eric Campo. A review of smart homes—present state and future challenges. *Computer methods and programs in biomedicine*, 91(1):55–81, 2008.

[27] Anis Charfi and Mira Mezini. Hybrid web service composition: business processes meet business rules. In *ICSOC*, pages 30–38. ACM, 2004.

[28] Harry Chen, Tim Finin, and Amupam Joshi. Semantic web in the context broker architecture. Technical report, DTIC Document, 2005.

[29] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: elastic execution between mobile device and cloud. In *Conference on Computer systems*. ACM, 2011.

[30] Pietro Ciciriello, Luca Mottola, and Gian Pietro Picco. Building virtual sensors and actuators over logical neighborhoods. In *International workshop on Middleware for sensor networks*, pages 19–24. ACM, 2006.

[31] Cognizant Reports. Reaping the Benefits of the Internet of Things. URL: `http://www.cognizant.com/InsightsWhitepapers/Reaping-the-Benefits-of-the-Internet-of-Things.pdf`, 2015. [Online; accessed Mar-'15].

[32] Adrian Copie, T Fortis, Victor Ion Munteanu, and Viorel Negru. From cloud governance to iot governance. In *Advanced Information Networking and Applications Workshops*, pages 1229–1234. IEEE, 2013.

[33] Georgiana Copil, Daniel Moldovan, Hong-Linh Truong, and Schahram Dustdar. Sybl: an extensible language for controlling elasticity in cloud applications. In *CCGRID'13*.

[34] Georgiana Copil, Daniel Moldovan, Hong-Linh Truong, and Schahram Dustdar. Multi-level elasticity control of cloud services. In *Service-Oriented Computing*, pages 429–436. Springer Berlin Heidelberg, 2013.

[35] CoreOs. CoreOS - a Linux for Massive Server Deployments. URL: `http://coreos.com/`. [Online; accessed Mar.-2016].

[36] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 49–62. ACM, 2010.

[37] Davidson, Emily A (Softchoice Advisor). The Software-Defined-Data-Center (SDDC): Concept Or Reality? URL: http://tinyurl.com/omhmbfv. [Online; accessed Jan-'15].

[38] Stephen Dawson-Haggerty, Xiaofan Jiang, Gilman Tolle, Jorge Ortiz, and David Culler. smap: a simple measurement and actuation profile for physical information. In *SenSys*, pages 197–210, 2010.

[39] Luciana Moreira Sá De Souza, Patrik Spiess, Dominique Guinard, Moritz Köhler, Stamatis Karnouskos, and Domnic Savio. Socrades: A web service based shop floor integration infrastructure. In *The internet of things*, pages 50–67. 2008.

[40] DevOps.com. Surprise! Broad Agreement on the Definition of DevOps. URL: http://devops.com/2015/05/13/surprise-broad-agreement-on-the-definition-of-devops/, 2016. [Online; accessed Jan-'16].

[41] Anind K Dey, Gregory D Abowd, and Daniel Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-computer interaction*, 16(2):97–166, 2001.

[42] Salvatore Distefano, Giovanni Merlino, and Antonio Puliafito. Sensing and actuation as a service: a new development for clouds. In *NCA*, pages 272–275, 2012.

[43] Salvatore Distefano, Giovanni Merlino, and Antonio Puliafito. A utility paradigm for IoT: The sensing Cloud. *Pervasive and mobile computing*, 20:127–144, 2015.

[44] Don DeLoach. Internet of Things Part 4: Critical issues around governance for the Internet of Things. URL: http://tinyurl.com/mxnq3ma. [Online; accessed July-2014].

[45] Avri Doria, J Hadi Salim, Robert Haas, Horzmud Khosravi, Weiming Wang, Ligang Dong, Ram Gopal, and Joel Halpern. Forwarding and control element separation (forces) protocol specification. *Internet Requests for Comments, RFC Editor, RFC*, 5810, 2010.

[46] Charalampos Doukas and Ilias Maglogiannis. Bringing iot and cloud computing towards pervasive healthcare. In *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012 Sixth International Conference on*, pages 922–926. IEEE, 2012.

[47] Schahram Dustdar, Yike Guo, Benjamin Satzger, and Hong-Linh Truong. Principles of elastic processes. *Internet Computing, IEEE*, 15(5):66–71, 2011.

[48] Dustin Whittle. An Introduction to DevOps. http://devops.com/2014/04/02/introductiontodevops/, 2014.

[49] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, 2003.

[50] European Commission. Report on the Consultation on IoT Governance. URL: `http://tinyurl.com/mx24d9o`. [Online; accessed August-2014].

[51] European Commission. Report on the public consultation on IoT governance. URL: `http://tinyurl.com/mx24d9o`. [Online; accessed August-2014].

[52] European Research Cluster on the Internet of Things. IoT Governance, Privacy and Security Issues. URL: `http://www.internet-of-things-research.eu/pdf/IERC_Position_Paper_IoT_Governance_Privacy_Security_Final.pdf`, 2016. [Online; accessed Jan-'16].

[53] Jason Flinn, Shafeeq Sinnamohideen, Niraj Tolia, and Mahadev Satyanarayanan. Data staging on untrusted surrogates. In *FAST*, volume 3, pages 15–28. Citeseer, 2003.

[54] forgerock.com. Forge Rock. URL: `https://www.forgerock.com/`. [Online; accessed June-2014].

[55] B Frank, Z Shelby, K Hartke, and C Bormann. Constrained application protocol (coap). *IETF draft, Jul*, 2011.

[56] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[57] Gartner. Top Seven Considerations for Configuration Management for Virtual and Cloud Infrastructures. `http://img2.insight.com/graphics/no/info2/insight_art6.pdf`, 2015. Last accessed: Jun 2015.

[58] Google. Google protocol buffers. `http://code.google.com/p/protobuf/`. [Online; accessed Jun-'13].

[59] Jim Gray. Why do computers stop and what can be done about it? In *Symposium on reliability in distributed software and database systems*, pages 3–12. Los Angeles, CA, USA, 1986.

[60] David Gregorczyk, T Bubhaus, and Stefan Fischer. A proof of concept for medical device integration using web services. In *SSD*, 2012.

[61] Jerzy W Grzymala-Busse. Three approaches to missing attribute values: A rough set perspective. In *Data Mining: Foundations and Practice*, pages 139–152. Springer, 2008.

[62] Dominique Guinard, Vlad Trifa, Stamatis Karnouskos, Patrik Spiess, and Domnic Savio. Interacting with the soa-based internet of things: Discovery, query, selection, and on-demand provisioning of web services. *Services Computing, IEEE Transactions on*, 3(3):223–235, 2010.

[63] Gary Hardy. Using IT governance and COBIT to deliver value with IT and respond to legal, regulatory and compliance challenges. *Information Security technical report*, 11(1):55–61, 2006.

[64] Mohammad Mehedi Hassan, Biao Song, and Eui-Nam Huh. A framework of sensor-cloud integration opportunities and challenges. In *ICUIMC*, 2009.

[65] Wu He, Gongjun Yan, and Li Da Xu. Developing vehicular data cloud services in the iot environment. *Industrial Informatics, IEEE Transactions on*, 10(2):1587–1595, 2014.

[66] Karen Henricksen, Jadwiga Indulska, Ted McFadden, and Sasitharan Balasubramaniam. Middleware for distributed context-aware systems. In *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE*, pages 846–863. Springer, 2005.

[67] Robert G Hollands. Will the real smart city please stand up? intelligent, progressive or entrepreneurial? *City*, 12(3):303–320, 2008.

[68] Jan Holler, Vlasios Tsiatsis, Catherine Mulligan, Stefan Avesand, Stamatis Karnouskos, and David Boyle. *From Machine-to-machine to the Internet of Things: Introduction to a New Age of Intelligence*. Academic Press, 2014.

[69] Jason I Hong and James A Landay. An architecture for privacy-sensitive ubiquitous computing. In *Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pages 177–189. ACM, 2004.

[70] IBM . SOA pages - Definition of SOA governance. URL: `http://ibm.com/software/solutions/soa/gov/`. [Online; accessed June-2014].

[71] Christian Inzinger, Stefan Nastic, Sanjin Sehic, Michael Vögler, Fei Li, and Schahram Dustdar. MADCAT - A methodology for architecture and deployment of cloud application topologies. In *Service-Oriented System Engineering*, 2014.

[72] IoT Now Magazine. Employee safety and security regulations raise the stakes for fleet operators. URL: `http://tinyurl.com/huzgrmr`, 2015. [Online; accessed Jan-'15].

[73] ITU-T Study Group 13. Recommendation ITU-T Y.2060. URL: `http://handle.itu.int/11.1002/1000/11559`, 2016. [Online; accessed Mar-'16].

[74] Alexander Keller and Remi Badonnel. Automating the provisioning of application services with the bpel4ws workflow language. In *Utility Computing*, pages 15–27. Springer, 2004.

[75] Sean Dieter Tebje Kelly, Nagender Kumar Suryadevara, and Subhas Chandra Mukhopadhyay. Towards the implementation of iot for environmental condition monitoring in homes. *Sensors Journal, IEEE*, 13(10):3846–3853, 2013.

[76] Hyojoon Kim and Nick Feamster. Improving network management with software defined networking. *Communications Magazine, IEEE*, 51(2):114–119, 2013.

[77] Jeffrey King, Raja Bose, Hen-I Yang, Steven Pickles, and Abdelsalam Helal. Atlas: A service-oriented sensor platform: Hardware and middleware to enable programmable pervasive spaces. In *LCN*, pages 630–638, 2006.

[78] Keith Kirkpatrick. Software-defined networking. *Communications of the ACM*, 56(9):16–19, 2013.

[79] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230, 2007.

[80] Boris Koldehofe, Frank Dürr, Muhammad Adnan Tariq, and Kurt Rothermel. The power of software-defined networking: line-rate content-based routing using openflow. In *MW4NG'12*, 2012.

[81] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, et al. Onix: A distributed control platform for large-scale production networks. In *OSDI*, volume 10, pages 1–6, 2010.

[82] Gerd Kortuem, Fahim Kawsar, Daniel Fitton, and Vasughi Sundramoorthy. Smart objects as building blocks for the internet of things. *Internet Computing, IEEE*, 14(1):44–51, 2010.

[83] Matthias Kovatsch, Martin Lanter, and Simon Duquennoy. Actinium: A restful runtime container for scriptable internet of things applications. In *Internet of Things*, pages 135–142, 2012.

[84] Marzena Kryszkiewicz. Rules in incomplete information systems. *Information sciences*, 113(3):271–292, 1999.

[85] Karthik Kumar and Yung-Hsiang Lu. Cloud computing for mobile users: Can offloading computation save energy? *Computer*, 43(4):51–56, 2010.

[86] Dimosthenis Kyriazis, Theodora Varvarigou, Anna Rossi, Douglas White, and Joshua Cooper. Sustainable smart city iot applications: Heat and electricity management & eco-conscious cruise control for public transportation. In *World of*

Wireless, Mobile and Multimedia Networks (WoWMoM), 2013 IEEE 14th International Symposium and Workshops on a, pages 1–5. IEEE, 2013.

[87] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *SIGCOMM Workshop on Hot Topics in Networks*. ACM, 2010.

[88] Marek Leszak, Dewayne E Perry, and Dieter Stoll. A case study in root cause defect analysis. In *Proceedings of the 22nd international conference on Software engineering*, pages 428–437. ACM, 2000.

[89] Grace Lewis, Sebastián Echeverría, Soumya Simanta, Ben Bradshaw, and James Root. Tactical cloudlets: Moving cloud computing to the edge. In *Military Communications Conference (MILCOM), 2014 IEEE*, pages 1440–1446. IEEE, 2014.

[90] Frank Leymann. Cloud Computing: The Next Revolution in IT. In *52th Photogrammetric Week '09*, pages 3–12, 2009.

[91] Frank Leymann and Dieter Roller. Production workflow: concepts and techniques. 2000.

[92] Ma łgorzata Steinder and Adarshpal S Sethi. A survey of fault localization techniques in computer networks. *Science of computer programming*, 53(2):165–194, 2004.

[93] Samuel R Madden, Michael J Franklin, Joseph M Hellerstein, and Wei Hong. TinyDB: an acquisitional query processing system for sensor networks. *ACM Transactions on database systems (TODS)*, 30(1):122–173, 2005.

[94] Sanchita Mal-Sarkar, Iftikhar U Sikder, Chansu Yu, and Vijay K Konangi. Uncertainty-aware wireless sensor networks. *International Journal of Mobile Communications*, 7(3):330–345, 2009.

[95] Martin Fowler. Microservices - a definition of this new architectural term. URL: `http://martinfowler.com/articles/microservices.html`. [Online; accessed Jan.-2016].

[96] Friedemann Mattern and Christian Floerkemeier. From the internet of computers to the internet of things. In *From active data management to event-based systems and more*, pages 242–259. Springer, 2010.

[97] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[98] Alan Messer, Ira Greenberg, Philippe Bernadat, Dejan Milojicic, Deqing Chen, Thomas J Giuli, and Xiaohui Gu. Towards a distributed platform for resource-constrained devices. In *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, pages 43–51. IEEE, 2002.

[99] Mike Loukides. What is DevOps? O'Reilly Media, 2012.

[100] Parastoo Mohagheghi and Øystein Haugen. Evaluating domain-specific modelling solutions. In *Advances in Conceptual Modeling - Applications and Challenges*, pages 212–221, 2010.

[101] Luca Mottola, Animesh Pathak, Amol Bakshi, Viktor K Prasanna, and Gian Pietro Picco. Enabling scope-based interactions in sensor network macroprogramming. In *MASS 2007*, pages 1–9, 2007.

[102] Taewoo Nam and Theresa A Pardo. Conceptualizing smart city with dimensions of technology, people, and institutions. In *Proceedings of the 12th Annual International Digital Government Research Conference: Digital Government Innovation in Challenging Times*, pages 282–291. ACM, 2011.

[103] Stefan Nastic, Georgiana Copil, Hong-Linh Truong, and Schahram Dustdar. Governing Elastic IoT Cloud Systems under Uncertainty. In *The 7th International Conference on Cloud Computing Technology and Science (CloudCom 2015)*, 2015.

[104] Stefan Nastic, Christian Inziger, Hong-Linh Truong, and Schahram Dustdar. GovOps: The Missing Link for Governance in Software-defined IoT Cloud Systems. In *WESOA14*, 2014.

[105] Stefan Nastic, Sanjin Sehic, Duc-Hung Le, Hong-Linh Truong, and Schahram Dustdar. Provisioning Software-defined IoT Cloud Systems. In *FiCloud'14*.

[106] Stefan Nastic, Sanjin Sehic, Michael Voegler, Hong-Linh Truong, and Schahram Dustdar. PatRICIA - A novel programing model for IoT applications on cloud platforms. In *SOCA*, 2013.

[107] Stefan Nastic, Hong-Linh Truong, and Schahram Dustdar. Sdg-pro: a programming framework for software-defined iot cloud gateways. *Journal of Internet Services and Applications*, 6(1):1–17, 2015.

[108] Stefan Nastic, Hong-Linh Truong, and Schahram Dustdar. A Middleware Infrastructure for Utility-based Provisioning of IoT Cloud Systems. In *The First IEEE/ACM Symposium on Edge Computing*, 2016. (In review.).

[109] Stefan Nastic, Hong-Linh Truong, and Schahram Dustdar. Data and Control Points: A Programming Model for Resource-constrained IoT Cloud Edge Devices. In *IEEE International Conference on Systems, Man, and Cybernetics (SMC 2016)*, 2016. In review.

[110] Stefan Nastic, Michael Voegler, Christian Inziger, Hong-Linh Truong, and Schahram Dustdar. rtGovOps: A Runtime Framework for Governance in Large-scale Software-defined IoT Cloud Systems. In *Mobile Cloud 2015*, 2015.

[111] Michael Niemann, André Miede, Wolfgang Johannsen, Nicolas Repp, and Ralf Steinmetz. Structuring SOA governance. *International Journal of IT/Business Alignment and Governance*, 1(1):58–75, 2010.

[112] Bruno AA Nunes, Manoel Mendonca, Xuan-Nam Nguyen, Katia Obraczka, and Thierry Turletti. A survey of software-defined networking: Past, present, and future of programmable networks. *Communications Surveys & Tutorials, IEEE*, 16(3):1617–1634, 2014.

[113] OASIS. Device Profile for Web Services (DPWS) Specification. `http://docs.oasis-open.org/ws-dd/ns/dpws/2009/01`. [Online; accessed Jul-'13].

[114] OASIS. MQ Telemetry Transport Specification. `http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html`. [Online; accessed Mar-'16].

[115] Open Stack Orchestration. Heat Project. URL: `https://wiki.openstack.org/wiki/Heat`. [Online; accessed Feb.-2015].

[116] OpenStack.org. OpenStack – Open source software for creating private and public clouds. `http://www.openstack.org/`. [Online; accessed Mar-'14].

[117] OpenTOSCA. OpenTOSCA. URL: `http://www.iaas.uni-stuttgart.de/OpenTOSCA/`. [Online; accessed Feb.-2015].

[118] OpsCode. Chef. URL: `http://opscode.com/chef`. [Online; accessed Feb.-2015].

[119] Pacific Controls. Galaxy platfrom of platforms. `http://pacificcontrols.net/products/`. [Online; accessed Jun-'13].

[120] Zdzisław Pawlak. Rough sets. *International Journal of Computer & Information Sciences*, 11(5):341–356, 1982.

[121] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing (draft). NIST special publication, 800:145, 2011.

[122] Kristin Potter, Paul Rosen, and Chris R Johnson. From quantification to visualization: A taxonomy of uncertainty visualization approaches. In *Uncertainty Quantification in Scientific Computing*, pages 226–249. Springer, 2012.

[123] Giuseppe Procaccianti, Patricia Lago, and Grace A Lewis. A catalogue of green architectural tactics for the cloud. In *Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA), 2014 IEEE 8th International Symposium on the*, pages 29–36. IEEE, 2014.

[124] Puppet Labs. Puppet. URL: `http://puppetlabs.org`. [Online; accessed Feb.-2015].

[125] Andres J Ramirez, Adam C Jensen, and Betty HC Cheng. A taxonomy of uncertainty for dynamically adaptive systems. In *SEAMS'12*, 2012.

[126] Anand Ranganathan, Shiva Chetan, Jalal Al-Muhtadi, Roy H Campbell, and M Dennis Mickunas. Olympus: A high-level programming model for pervasive computing environments. In *PerCom*, 2005.

[127] Rouan Wilsenach. DevOps Culture. `http://martinfowler.com/bliki/DevOpsCulture.html`, 2015.

[128] Bop Sandrino-Arndt. People, portfolios and processes: The 3p model of it governance. *Information Systems Control Journal*, 2:1–5, 2008.

[129] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. *Pervasive Computing*, 8(4):14–23, 2009.

[130] Andreas Schaefer, Marc Reichenbach, and Dietmar Fey. Continuous integration and automation for devops. In *IAENG Transactions on Engineering Technologies*, pages 345–358. Springer, 2013.

[131] Sanjin Sehic, Fei Li, Stefan Nastic, and Schahram Dustdar. A programming model for context-aware applications in large-scale pervasive systems.

[132] Mustapha Reda Senouci, Abdelhamid Mellouk, Latifa Oukhellou, and Amar Aissani. Uncertainty-aware sensor network deployment. In *GLOBECOM'11*. IEEE, 2011.

[133] SOA Software. Integrated SOA governance. URL: `http://www.soa.com/solutions/integrated_soa_governance`. [Online; accessed June-2014].

[134] John Soldatos, Martin Serrano, and Manfred Hauswirth. Convergence of utility computing with the internet-of-things. In *IMIS*, pages 874–879, 2012.

[135] João Pedro Sousa and David Garlan. Aura: an architectural framework for user mobility in ubiquitous computing environments. 2002.

[136] Stanford Plato. Stanford Plato Phenomenolgy. URL: `http://plato.stanford.edu/entries/phenomenology/`, 2015. [Online; accessed Jan-'15].

[137] Jerzy Stefanowski and Alexis Tsoukias. Incomplete information tables and rough classification. *Computational Intelligence*, 17(3), 2001.

[138] Diane M Strong, Yang W Lee, and Richard Y Wang. Data quality in context. *Communications of the ACM*, 40(5):103–110, 1997.

[139] Patrick Stuedi, Iqbal Mohomed, and Doug Terry. Wherestore: Location-based data storage for mobile devices interacting with the cloud. In *MCS*, 2010.

[140] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. IoTFlow: A software-defined storage architecture. In *SOSP*, pages 182–196. ACM, 2013.

[141] thingworx.com. ThingWorx. URL: `http://thingworx.com`. [Online; accessed Jan.-2015].

[142] Tridium. Sedona Virtual Machine. URL: `http://www.sedonadev.org/`. [Online; accessed Jan.-2016].

[143] Hong-Linh Truong and Schahram Dustdar. Principles for engineering IoT Cloud systems. *Cloud Computing, IEEE*, 2:68–76, 2015.

[144] Ovidiu Vermesan and Peter Friess. *Internet of things: converging technologies for smart environments and integrated ecosystems*. River Publishers, 2013.

[145] Ovidiu Vermesan, Peter Friess, Patrick Guillemin, Sergio Gusmeroli, Harald Sundmaeker, Alessandro Bassi, Ignacio Soler Jubert, Margaretha Mazura, Mark Harrison, M Eisenhauer, et al. Internet of things strategic research roadmap. *Internet of Things-Global Technological and Societal Trends*, pages 9–52, 2011.

[146] Michael Voegler, Johannes M. Schleicher, Christian Inziger, Stefan Nastic, Sanjin Sehic, and Schahram Dustdar. Leonore – large-scale provisioning of resource constrained iot deployments. In *SOSE*, 2015.

[147] Carl A Waldspurger. Memory resource management in vmware esx server. *ACM SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.

[148] Warren E Walker, Poul Harremoës, Jan Rotmans, Jeroen P van der Sluijs, Marjolein BA van Asselt, Peter Janssen, and Martin P Krayer von Krauss. Defining uncertainty: a conceptual basis for uncertainty management in model-based decision support. *Integrated assessment*, 4(1):5–17, 2003.

[149] Nanbor Wang, Douglas C Schmidt, Aniruddha Gokhale, Christopher D Gill, Balachandran Natarajan, Craig Rodrigues, Joseph P Loyall, and Richard E Schantz. Total quality of service provisioning in middleware and applications. *Microprocessors and Microsystems*, 27(2):45–54, 2003.

[150] Waveworks. Wave router. URL: `https://github.com/weaveworks/weave`. [Online; accessed Mar.-2015].

[151] Kristin Weber, Boris Otto, and Hubert Österle. One size does not fit all—a contingency approach to data governance. *Journal of Data and Information Quality (JDIQ)*, 1(1):4, 2009.

[152] Rolf H Weber. Internet of things–governance quo vadis? *Computer Law & Security Review*, 29(4):341–347, 2013.

[153] Danny Weyns, Sam Malek, and Jesper Andersson. Forms: a formal reference model for self-adaptation. In *Proceedings of the 7th international conference on Autonomic computing*, pages 205–214. ACM, 2010.

[154] Jon Whittle, Pete Sawyer, Nelly Bencomo, Betty HC Cheng, and Jean-Michel Bruel. Relax: Incorporating uncertainty into the specification of self-adaptive systems. In *RE'09*, 2009.

[155] WSO2. Stratos. `http://wso2.com/cloud/stratos/`. [Online; accessed Jun-'13].

[156] Xively. Xively. URL: `http://xively.com`. [Online; accessed Jan-'15].

[157] Shanhe Yi, Cheng Li, and Qun Li. A survey of fog computing: concepts, applications and issues. In *Proceedings of the 2015 Workshop on Mobile Big Data*, pages 37–42. ACM, 2015.

[158] Lamia Youseff, Maria Butrico, and Dilma Da Silva. Toward a unified ontology of cloud computing. In *Grid Computing Environments Workshop, 2008. GCE'08*, pages 1–10. IEEE, 2008.

[159] Miao Yun and Bu Yuxin. Research on the architecture and key technology of internet of things (iot) applied on smart grid. In *Advances in Energy Engineering (ICAEE), 2010 International Conference on*, pages 69–72. IEEE, 2010.

[160] Madoka Yuriyama and Takayuki Kushida. Sensor-cloud infrastructure-physical sensor management with virtualized sensors on cloud computing. In *Network-Based Information Systems (NBiS), 2010 13th International Conference on*, pages 1–8. IEEE, 2010.

[161] Andrea Zanella, Nicola Bui, Angelo Castellani, Lorenzo Vangelista, and Michele Zorzi. Internet of things for smart cities. *Internet of Things Journal, IEEE*, 1(1):22–32, 2014.

[162] Arkady Zaslavsky, Charith Perera, and Dimitrios Georgakopoulos. Sensing as a service and big data. *arXiv preprint arXiv:1301.0159*, 2013.

[163] Man Zhang, Shaukat Ali, Tao Yue, Dipesh Pradhan, Bran Selic, Oscar Okariz, and Roland Norgren. An Uncertainty Taxonomy to Support Model-Based Uncertainty Testing of Cyber-Physical Systems. Technical report, Simula, 2015.

APPENDIX A

A Taxonomy of Infrastructure-level Uncertainties in CPS

Overview of CPS Infrastructure

The infrastructure of Cyber-Physical Systems (CPS) is complex and usually comprises variety of sensors, actuators, gateways, multiplicity of network elements and (multi) cloud platforms (e.g., VMs and cloud services). In general, there are three main logical parts of CPS infrastructure: Physical layer (containing physical devices, data centers, etc.), virtual layer (encompassing virtualized CPS infrastructure, most importantly cloud platforms and software-defined gateways) and Virtual verticals (in this context application/system specific configurations and policies that directly affect CPS infrastructure).

Generally, errors, faults and uncertain behaviors at the infrastructure level affect the execution of CPS applications independent of their business logic. Therefore, classifying the infrastructure level uncertainties can be generic to a large extent, i.e., based on the functionality CPS applications usually expect from such infrastructures to deliver. Some of the responsibilities (functionality) of the CPS infrastructure include:

- Providing communication facilities (i.e., network) among the sensors/actuators and CPS applications/services;
- Providing an execution environment for such applications (e.g., on gateways or in the cloud);
- Providing (temporary and/or permanent) storage for the large amounts of sensory data;

- Providing facilities for generating, preprocessing and delivering sensory data;
- Enabling routing/buffering of actuation requests (from applications to physical actuators).

We mainly focus on uncertainties that affect the aforementioned functionality of the infrastructure. More specifically, such uncertainties affect the expected state of the infrastructure, i.e., the outcome when an application utilizes (e.g., invokes) some of the infrastructure functionality. Generally, such uncertainties can cause the CPS infrastructure to display faulty behavior (i.e., come into an error state) or some uncertain state (not necessarily an error state).

In the traditional fault, error, failure classifications, faults lead to some form of errors which are manifested as failures at application or service level [10]. The main difference between the traditional (latent) error state and the uncertain state are the causes that lead the CPS infrastructure to transition to such state and in how such state manifests at application level or in the surrounding environment. In our context, uncertainties can coincide with faults, but are much broader category. For example, an empty data channel can be considered as an uncertain infrastructure state, since it can be caused by a sensor failure (error state) or because there is no change in the physical environment, thus nothing is detected by a sensor (normal state).

Infrastructure level uncertainties taxonomy for CPS systems

Our taxonomy classifies the (at design time) known sources of the error and the uncertain states, e.g., the behaviors of system units which are potentially, positively or cumulatively responsible for the error/uncertain states of CPS infrastructure. Figure A.1 gives an overview of the infrastructure level uncertainties taxonomy for CPS systems. The taxonomy shown in Figure A.1 comprises a set of concepts (i.e., uncertainty properties classes), which are a extensions of the concepts defined in the meta model introduced in [163]. We have identified 7 main uncertainty properties classes at the infrastructure level. Next we describe these property classes in more detail.

Ingress/egress uncertainties (What the uncertainties affect) Uncertainties at the CPS infrastructure can manifest themselves as failures or as functionality degradation at application level (e.g., [49]) or in the physical environment [72]. For example, empty data channel will obviously be noticed by an application, while malfunctioning chiller wing will be noticed in the physical environment.

Uncertainty locality (Where uncertainties occur): Depending on the locality of the uncertainties occurrence, we differentiate between the uncertainties that are present in the infrastructure itself, i.e., in hardware (e.g., sensors, actuators, gateways, etc.), CPS platform/virtual part of the infrastructure (e.g., cloud services or elasticity controllers) and the uncertainties that occur outside the infrastructure and affect the infrastructure,

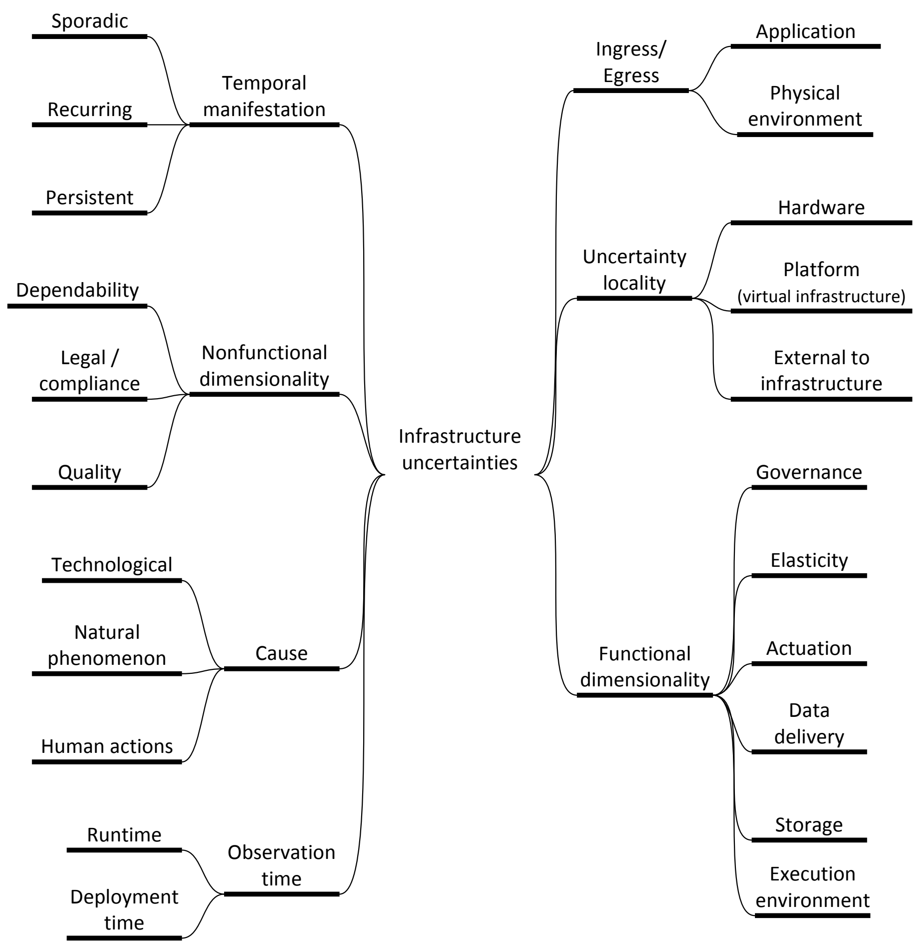


Figure A.1: Overview of uncertainties taxonomy for CPS.

e.g., smoke interfering with normal operation of surveillance cameras. These uncertainty properties are based on the previous work on fault localization [8, 92] and root cause analysis [88].

Nonfunctional dimensionality (Which nonfunctional property they affect): The uncertainties can affect the dependability [10] (e.g., safety, availability, reliability, security, etc.), data quality or legal/compliance of the CPS infrastructure [138, 19, 152]. It is worth noticing here that the nonfunctional dimensionality can be used to measure the degree of sensitivity to an uncertainty, where a complete functionality failure is the highest degree and no functionality degradation (e.g., no availability degradation) is the lowest degree.

Causes of uncertainty (What causes them): The uncertainties can be caused by some natural phenomenon in the surrounding environment, they can be a consequence of human actions or they can be technology caused uncertainties. Under uncertainties with technological cause, we classify all the uncertainties that are caused by some infrastructure phenomenon, which is beyond application developer's control. For example, these can be infrastructure hardware failures or bugs in the virtual infrastructure. Generally, these uncertainty properties represent the phenomenological cause of an uncertainty [136].

Temporal manifestation (How they manifest in time): The uncertainties can manifest in time as persistent, sporadic or as recurring. Generally, temporal manifestation denotes the duration of the infrastructure uncertainty state caused by that uncertainty. For example persistent uncertainties will cause permanent uncertainty state, i.e., until an outside action (e.g., human intervention) causes the infrastructure to return from the uncertain state to a normal state. These properties are inspired by the traditional software bugs classifications [59].

Functional dimensionality (Which functional properties they affect): As already discussed at the beginning of this section, CPS infrastructure is responsible to provide a specific functionality to the applications. Depending on what functionality class they affect, we differentiate among elasticity, governance, actuation, data delivery, storage or execution environment uncertainties. The functional dimensionality is derived from state-of-the-art in CPS infrastructure research [9, 160, 139, 129].

Observation time (When do they manifest/become active): Depending on when in the application lifecycle an uncertainty becomes active, i.e., potentially manifests itself as a failure, we have deployment time or runtime uncertainties.

Elementary uncertainty families

When classifying the uncertainties, we notice that not all the combination of the uncertainty properties are allowed. For example, it makes no sense to have a natural phenomenon uncertainty which occurs at the platform level (in software). Subsequently we identify the uncertainty families that are most common in practice. The uncertainty families are the permissible combinations of uncertainty properties (without claim of completeness). The families are mainly categorized depending on the functional dimensionality of the uncertainties.

Data delivery uncertainties family

The data delivery uncertainties family includes such uncertainties that affect the infrastructure's facilities for generating, preprocessing and delivering (sensory) data. It includes three main elementary categories, i.e., uncertainties affecting the dependability of the data delivery facilities, uncertainties affecting the quality of data and uncertainties related to legal/compliance. In the following, we describe them to more detail. Examples of these uncertainties are shown in Figure A.2.

Data delivery dependability uncertainties – These uncertainties affect the general dependability of the data delivery facilities. They can originate due to a human action or have a technological cause. They are located in hardware or platform. They can have any temporal manifestation and can be observed at any phase of application lifecycle.

Data quality uncertainties – These uncertainties affect the quality of the data generated and/or delivered by the CPS infrastructure. They can have a technological cause or originate due to a human action or some natural phenomenon. They can have any defined locality. They can have any of the defined temporal manifestations and can be observed at any phase of application lifecycle.

Data delivery legal/compliance uncertainties – These uncertainties affect the legal or compliance aspects of the data delivery process. They originate due to human actions, external to infrastructure. They are persistent and observed during application's runtime.

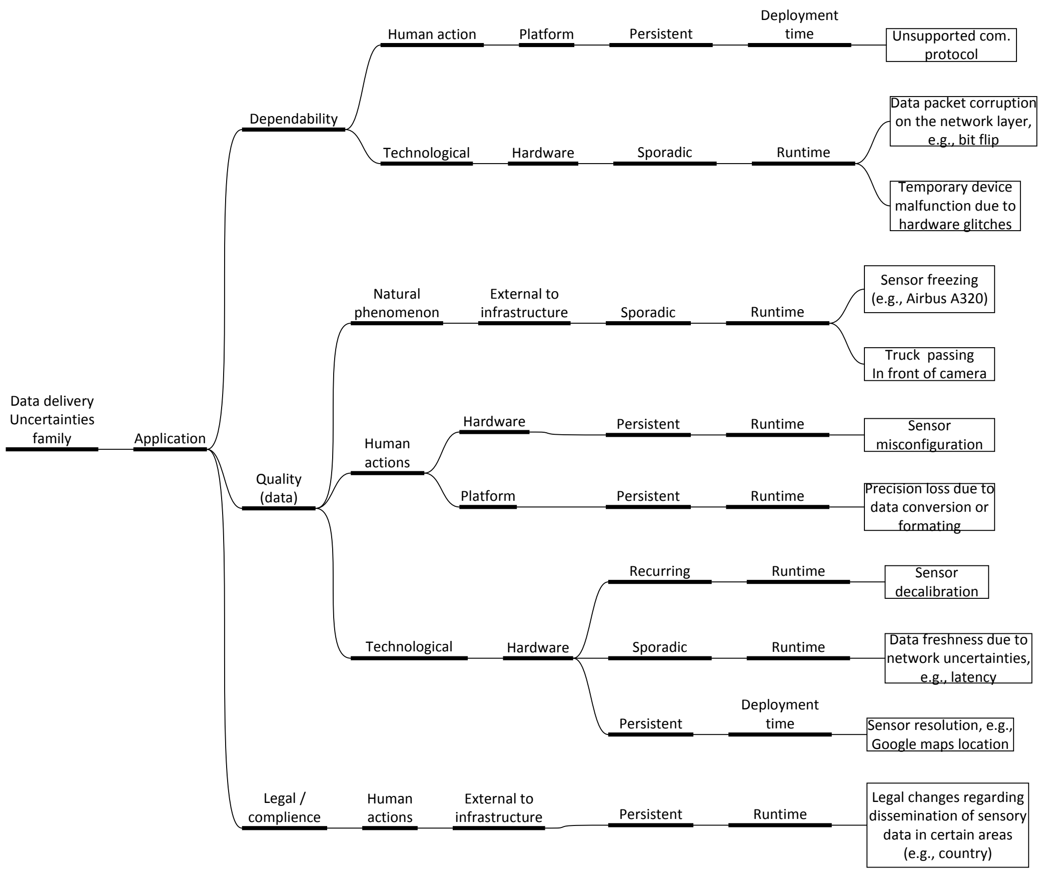

Data delivery Uncertainties family
Application
Dependability
Human action
Platform
Persistent
Deployment time
Unsupported com. protocol
Technological
Hardware
Sporadic
Runtime
Data packet corruption on the network layer, e.g., bit flip
Temporary device malfunction due to hardware glitches
Quality (data)
Natural phenomenon
External to infrastructure
Sporadic
Runtime
Sensor freezing (e.g., Airbus A320)
Truck passing In front of camera
Human actions
Hardware
Persistent
Runtime
Sensor misconfiguration
Platform
Persistent
Runtime
Precision loss due to data conversion or formating
Technological
Hardware
Recurring
Runtime
Sensor decalibration
Sporadic
Runtime
Data freshness due to network uncertainties, e.g., latency
Persistent
Deployment time
Sensor resolution, e.g., Google maps location
Legal / complience
Human actions
External to infrastructure
Persistent
Runtime
Legal changes regarding dissemination of sensory data in certain areas (e.g., country)

Figure A.2: Overview of data delivery uncertainties family.

Actuation uncertainties family

The actuation uncertainties family includes such uncertainties that affect the infrastructure's mechanisms related to routing, buffering, delivering and ordering (e.g., by priorities) of actuation requests that originate on the application level and are propagated to the physical or virtual actuators. All uncertainties from this family are observed during application runtime. The actuation uncertainties family comprises three main elementary categories: Actuation legal/compliance uncertainties, actuation uncertainties affecting the dependability of applications and actuation uncertainties affecting the dependability of environment. In the following, we describe them to more detail. Examples of these uncertainties are shown in Figure A.3.

Actuation legal/compliance uncertainties – These uncertainties affect the legal or compliance aspects of the actuation process. They are caused by human actions, in the platform and are mainly persistent uncertainties.

Actuation dependability uncertainties in applications – These uncertainties affect the general dependability of the applications, i.e., actuation facilities. They can be caused by a human action or technology. They are located in hardware or platform. They can have any temporal manifestation defined in the taxonomy.

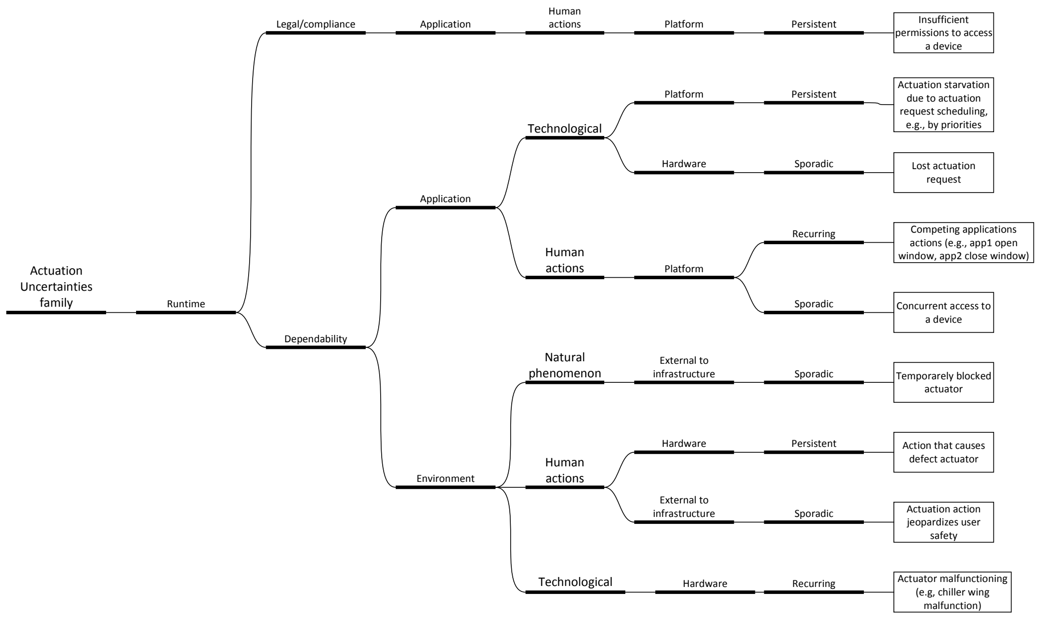
Actuation dependability uncertainties in environment – These uncertainties affect the general dependability of the physical environment. They can have any origin specified in the taxonomy. They usually located in the hardware or external to the infrastructure and have any of the specified temporal manifestations.

Execution environment uncertainties family

The execution environment uncertainties family comprises uncertainties about the assumptions made by application developers about the underlying infrastructure functionality. They interfere with the infrastructure's ability to support application execution, thus are classified as application uncertainties. The execution environment uncertainties family comprises two main elementary categories: Execution environment uncertainties observed at application deployment and execution environment uncertainties observed at application runtime. In the following, we describe them to more detail. Examples of these uncertainties are shown in Figure A.4.

Deployment time execution environment uncertainties – These uncertainties are observed during application's deployment phase. The nonfunctional dimensionality of such uncertainties is either dependability or legal/compliance and their locality manifestation is mostly at hardware or platform level. They have a technological origin or can be caused by human actions. They can have any of the defined temporal manifestations.

Runtime time execution environment uncertainties – These uncertainties interfere with application's execution, thus are observed during its runtime, mainly by affecting



Actuation Uncertainties family
Runtime
Legal/compliance
Application
Human actions
Platform
Persistent
Insufficient permissions to access a device
Dependability
Application
Technological
Platform
Persistent
Actuation starvation due to actuation request scheduling, e.g., by priorities
Hardware
Sporadic
Lost actuation request
Human actions
Platform
Recurring
Competing applications actions (e.g., app1 open window, app2 close window)
Sporadic
Concurrent access to a device
Environment
Natural phenomenon
External to infrastructure
Sporadic
Temporarely blocked actuator
Human actions
Hardware
Persistent
Action that causes defect actuator
External to infrastructure
Sporadic
Actuation action jeopardizes user safety
Technological
Hardware
Recurring
Actuator malfunctioning (e.g, chiller wing malfunction)

Figure A.3: Overview of actuation uncertainties family.

infrastructure's dependability at hardware of platform level. They can have any of the defined temporal manifestations or origin.

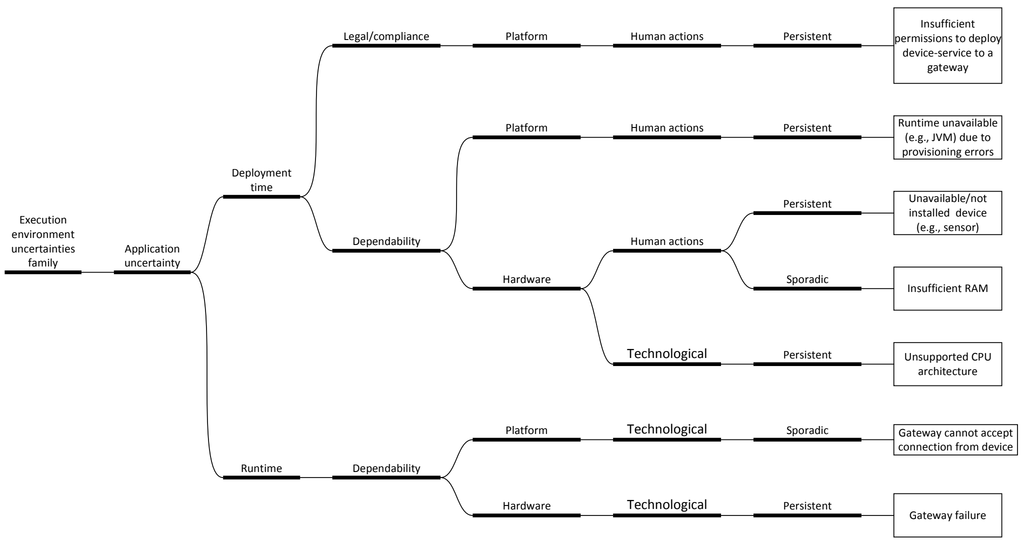


Figure A.4: Overview of execution environment uncertainties family.

Storage uncertainties family

The storage uncertainties family includes uncertainties that affect the infrastructure's facilities for persistent storage of monitoring (sensory) data. This family mainly manifests as failure at application level when such applications perform batch data analytics (As opposed to the data delivery facilities, where the focus is on real-time data processing). All uncertainties from this family are observed during application runtime. The storage uncertainties family comprises three main elementary categories: Uncertainties affecting the dependability of the storage facilities, uncertainties affecting the quality of the historical data and uncertainties related to legal/compliance regulating sensory data storage. In the following, we describe them to more detail. Examples of these uncertainties are shown in Figure A.5.

Storage quality uncertainties – These uncertainties affect the quality of the data (most notably historical sensory data) stored in the CPS infrastructure. They can have a technological origin or are caused by a human action at hardware or platform level. They can have any of the temporal manifestations specified in the taxonomy.

Storage dependability uncertainties – These uncertainties affect the general dependability of the data storage facilities. They can have a technological origin or are caused by human action. They are located in hardware or platform and can have any temporal manifestation.

Storage legal/compliance uncertainties – These uncertainties affect the legal or compliance aspects related to the data storage. They originate due to human actions, external to infrastructure and are persistent uncertainties.

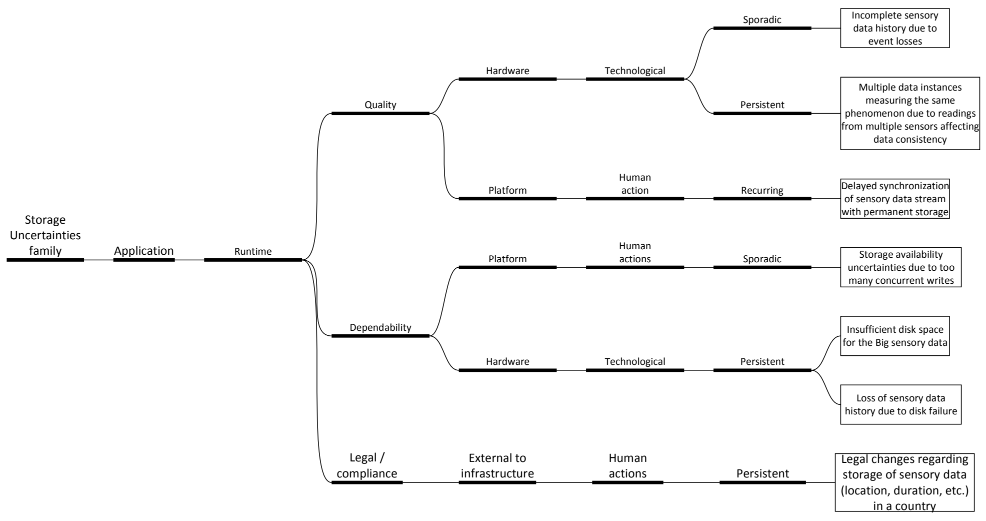

Storage Uncertainties family
Application
Runtime
Quality
Hardware
Technological
Sporadic
Incomplete sensory data history due to event losses
Persistent
Multiple data instances measuring the same phenomenon due to readings from multiple sensors affecting data consistency
Platform
Human action
Recurring
Delayed synchronization of sensory data stream with permanent storage
Dependability
Platform
Human actions
Sporadic
Storage availability uncertainties due to too many concurrent writes
Hardware
Technological
Persistent
Insufficient disk space for the Big sensory data
Loss of sensory data history due to disk failure
Legal / compliance
External to infrastructure
Human actions
Persistent
Legal changes regarding storage of sensory data (location, duration, etc.) in a country

Figure A.5: Overview of storage uncertainties family.

Composite uncertainties families

Composite uncertainties appear mostly in infrastructure's higher-level functionality – most notably, but not limited to governance and elasticity facilities, thus they mostly manifest at the higher levels in the infrastructure stack, e.g., the platform. Composite uncertainties mostly come into effect through the uncertainties propagation and/or uncertainties aggregation from the elementary uncertainties families. It is also worth noticing that composite uncertainties can be used as an extension point of the infrastructure uncertainties classification.

Although unknown uncertainties (unknown unknowns) are out-of-scope of this taksonomy, we notice that a very large number of such uncertainties can manifest themselves at the infrastructure level. This is mainly due to complex dependencies among the infrastructure components and effects of uncertainty propagation and/or uncertainty aggregation between such components. Generally, the root cause, locality, temporal manifestation, etc., of unknown uncertainties are inherently difficult if not impossible to determine. Therefore, classification of such uncertainties is usually application specific and can be classified under different or even multiple elementary classes depending on the task-at-hand.

Governance uncertainties family

The governance uncertainties family includes uncertainties that affect the infrastructure's facilities responsible to realize CPS governance processes or the uncertainties which make such processes invalid. Figure A.6 shows UML diagram of the composed uncertainties families related to governance uncertainties:

Governance process execution uncertainties – Governance process execution uncertainties affect the dependability of the governance process during runtime. They are observed at applications runtime and are mainly located in the platform. They usually have a synthetic origin and any permissible temporal manifestation.

For example a golf course management application polls diagnostic data from vehicles (e.g., with CoAP). However, a golf course manager could design a governance process that is triggered in specific situations such as in case of emergency. Such process could, for example, increase the update rate of the vehicle sensors and change the communication protocol to MQTT in order to satisfy a high-level governance objective, e.g., company's compliance policy to handle emergency updates in (near) real-time. In this context it is uncertain whether the governance process will be executed consistently across the infrastructure, because some vehicle sensors might not support functionality to dynamically change their update rate.

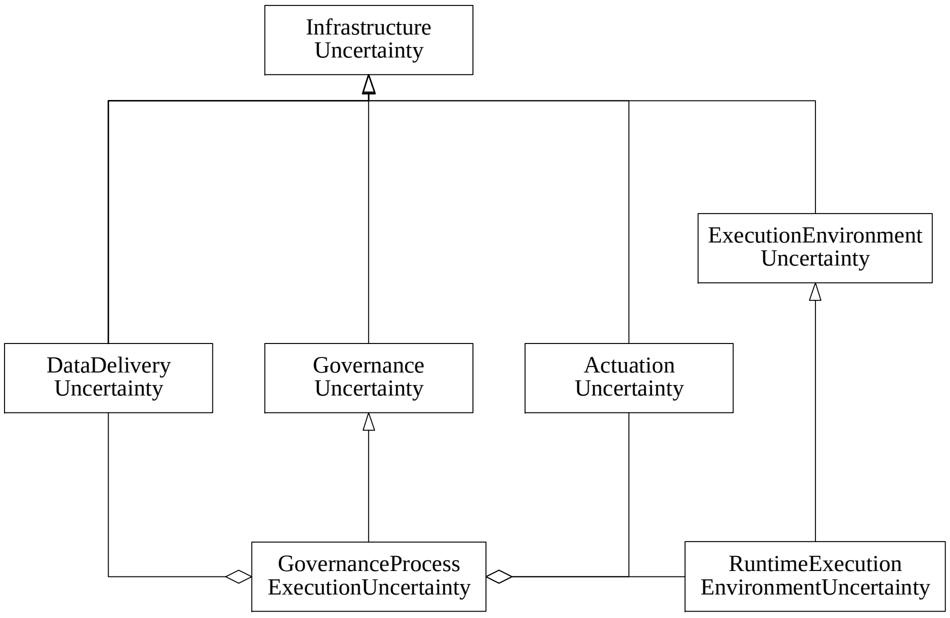

Infrastructure Uncertainty
ExecutionEnvironment Uncertainty
DataDelivery Uncertainty
Governance Uncertainty
Actuation Uncertainty
GovernanceProcess ExecutionUncertainty
RuntimeExecution EnvironmentUncertainty

Figure A.6: Overview of governance composite uncertainties family.

Elasticity uncertainties family

Elasticity is dependent on multiple factors. First, elasticity decisions are taken based on monitoring information, so uncertainty related to monitoring has great importance. Based on monitoring information, elasticity decisions are enforced through a combination of software and hardware actuation mechanisms, each of them also potentially introducing their own uncertainties. Figure A.7 shows UML diagram of the composed uncertainties families related to elasticity uncertainties:

Monitoring data uncertainties – These uncertainties affect elasticity of the system, and can refer to uncertainty of monitoring data quality, e.g., availability or freshness. They usually have a synthetic origin, i.e., required information is not collected and monitored due to a software error. Another cause can be software failure of monitoring system, or of monitoring information data source. Another cause is data collection mechanism and intervals, especially considering poll-based data collection systems, which collect and report monitoring information only at certain time intervals. They are located in platform. They can have any temporal manifestation and can are observed at application runtime.

Cloud Service behavioral uncertainty after actuation – These uncertainties affect the elasticity of the application by reducing the effectiveness, or affecting the impact of enforced elasticity actions. They originate in the (cloud provider's) platform not offering consistent performance across different instances of the same used cloud service, either to colocation or congestion or virtual resources, or complete/partial failure due to underlying cloud software and hardware infrastructure.

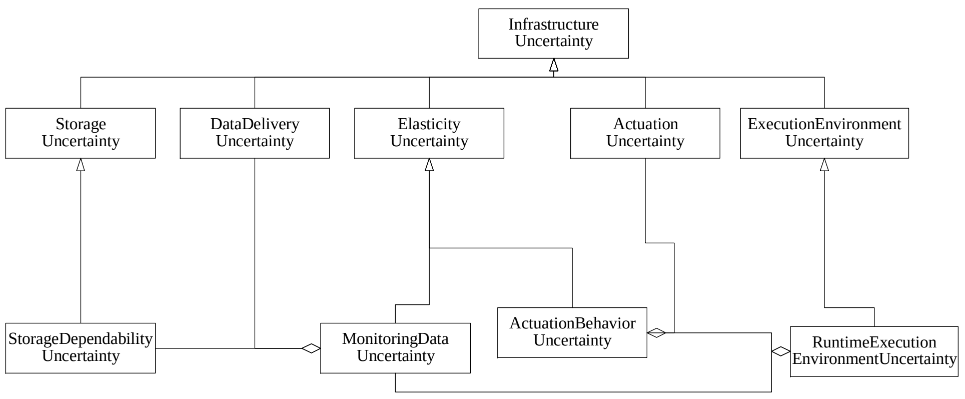

Infrastructure Uncertainty
Storage Uncertainty
DataDelivery Uncertainty
Elasticity Uncertainty
Actuation Uncertainty
ExecutionEnvironment Uncertainty
StorageDependability Uncertainty
MonitoringData Uncertainty
ActuationBehavior Uncertainty
RuntimeExecution EnvironmentUncertainty

Figure A.7: Overview of elasticity composite uncertainties family.

APPENDIX B

GovOps Policy Language BNF

```
Constraint := constraintName : CONSTRAINT ComplexCondition |
    CONSTRAINT ComplexCondition UncertaintyDetails

Monitoring := monitoringName : MONITORING varName=MetricFormula

UncertaintyParameter:= String| String BitwiseOperator UncertaintyParameter
UncertaintyDetails:= CONSIDERING_UNCERTAINTY UncertaintyParameter

Strategy := strategyName :
    STRATEGY CASE ComplexCondition : action(parameterList) |
    STRATEGY CASE ComplexCondition : action(parameterList)
        FOR GovName UncertaintyDetails |
    strategyName : STRATEGY WAIT ComplexCondition |
    strategyName : STRATEGY STOP ComplexCondition |
    strategyName : STRATEGY RESUME ComplexCondition


Query := query:= QueryParameter
QueryParameter= paramType=paramValue |
    paramType=paramValue AND QueryParameter
GovernanceScope:= govName: GOVERNANCE_SCOPE Query UncertaintyDetails

MetricFormula := metric | number |
    metricFormula MathOperator metric |
    metricFormula MathOperator number

Condition := metric RelationOperator number|
    number RelationOperator metric |
    Violated(name)|Fulfilled(name)
ComplexCondition := Condition | ComplexCondition BitwiseOperator Condition |
    (ComplexCondition BitwiseOperator Condition)


MathOperator := + | - | * | /
BitwiseOperator := OR | AND | XOR | NOT
RelationOperator := <|>|>=|<=|==|!=
```