IoT@run-time: a model-based approach to support deployment and self-adaptations in IoT systems

A Dissertation
by
Iván Alfonso

Submitted to the Faculty of Engineering of the
Universidad de los Andes
in partial fulfillment for the requirements for the Degree of
Doctor in Engineering
and
Submitted to the Faculty of Computer Science, Multimedia and Telecommunications of the
Universitat Oberta de Catalunya
in partial fulfillment for the requirements for the Degree of
Doctor in Network and Information Technologies

Advisors:	Dr. Kelly Garcés
	Dr. Harold Castro
	Dr. Jordi Cabot
Jury:	Dr. Rafael Capilla
	Dr. Guilherme Travassos
	Dr. Carlos Lozano

December 2022

Abstract

In recent years, the Internet of Things (IoT) has expanded its fields and areas of application, becoming a key component in industrial processes and even in the activities we perform daily. The growth of IoT has generated increasingly restrictive requirements, mainly in systems that analyze information in real time. Similarly, IoT system architectures have evolved to implement new strategies and patterns (such as edge and fog computing) to meet system requirements. Traditionally, an IoT system was composed of two layers: the device layer (sensors and actuators) and the cloud layer for information processing and storage. Today, most IoT systems leverage edge and fog computing to bring computation and storage closer to the device layer, decreasing bandwidth consumption and latency. Although the use of these multi-layer architectures can improve performance, it is challenging to design them because the dynamic and changing IoT environment can impact Quality of Service (QoS) and system operation. IoT systems are often exposed to changing environments that induce unexpected runtime events such as signal strength instability, latency growth and software failures. To cope with these events, system adaptations should be automatically executed at runtime, i.e., IoT systems should have self-adaptation capabilities.

In this sense, better support in the design, deployment, and self-adaptation stages of multilayer IoT systems is needed. However, the tools and solutions found in the literature do not address the complexity of multi-layered IoT systems, and the languages for specifying the adaptation rules that govern the system at runtime are limited.

Therefore, we propose a modeling-based approach that addresses the limitations of existing studies to support the design, deployment, and management of self-adaptive IoT systems. Our solution is divided into two stages:

Modeling (design time): to support the design tasks, we propose a Domain Specific Language (DSL) that enables to specify the multi-layered architecture of the IoT system, the deployment of container-based applications, and rules for the self-adaptation at runtime. Additionally, we design a code generator that produces YAML manifests for the deployment and management of the IoT system at runtime.

Self-adaptation (runtime): we have designed a framework based on the MAPE-K loop to monitor and adapt the IoT system following the rules specified in the model (design time). The deployment and configuration of the tools and technologies used by this framework is performed using the YAML manifests produced by the code generator.

Additionally, we have designed two extensions to our DSL. The first one is an extension focused on modeling IoT systems deployed in the underground coal mining industry. This DSL addresses new mining domain concepts such as mine structure specification and control points. The second DSL extension is focused on modeling IoT systems deployed in Wastewater Treatment Plants (WWTPs). This DSL extension addresses the modeling of the WWTP process block diagram using a graphical notation. Even with these two DSL extensions, there is no need to modify our framework that manages system adaptation at runtime.

Finally, we have conducted experimental studies classified into three groups: (1) we validated the expressiveness and usability of the DSL through experiments with 13 participants who performed modeling exercises (using the DSL) and answered surveys reporting their experience; (2) we functionally validated the architectural adaptations using our framework and comparing the performance and availability between a non-self-adaptive system versus a self-adaptive system implementing our approach; (3) finally, we evaluate the ability and performance of our framework to address the growth of concurrent adaptations on an IoT system. The results of these experiments demonstrated that (1) the DSL has the expressiveness to model multi-layered IoT systems (including its self-adaptive behaviour) and the learning curve is favorable; (2) functional tests demonstrated how the performance and availability of the system improves when using our approach; and (3) we have identified scalability limitations of the framework and proposed insights to address them.

Keywords: Internet of Things, Model-Driven Engineering, Self-Adaptive System, Domain Specific Language, Edge and Fog Computing.

Resumen

Durante los últimos años, el Internet de las Cosas (IoT) ha ampliado sus campos y áreas de aplicación convirtiéndose en un componente clave en los procesos industriales e incluso en las actividades que realizamos a diario. El crecimiento en el uso de IoT ha generado requerimientos cada vez más restrictivos, principalmente en sistemas que requieren análisis de información en tiempo real. De igual forma, las arquitecturas de los sistemas IoT han evolucionado para implementar nuevas estrategias y patrones (como la computación de borde y niebla) que permitan cumplir con los requerimientos del sistema. Tradicionalmente, un sistema IoT se componía de dos capas: la capa de dispositivo (sensores y actuadores) y la capa de nube para el procesamiento y almacenamiento de la información. Hoy en día, la mayoría de los sistemas IoT aprovechan la computación de borde y niebla para acercar físicamente la computación y el almacenamiento hacia la capa de dispositivo, disminuyendo así el consumo de ancho de banda y latencia. Aunque el uso de estas arquitecturas multicapa favorecen el desempeño, es un reto diseñarlas debido a que el ambiente IoT dinámico y cambiante puede impactar la Calidad del Servicio (QoS) y operación del sistema. Los sistemas IoT suelen estar expuestos a entornos cambiantes que inducen eventos inesperados en tiempo de ejecución como la inestabilidad de la intensidad de la señal, el crecimiento de la latencia y los fallos de software. Para hacer frente a estos eventos, adaptaciones del sistema deben ser automáticamente ejecutadas en tiempo de ejecución, es decir, los sistemas IoT deberían tener capacidades de autoadaptación.

En este sentido, es necesario brindar soporte en las etapas de diseño, despliegue y autoadaptación de sistemas IoT multicapa. Sin embargo, las herramientas y soluciones encontradas en la literatura no abordan la complejidad de los sistemas IoT multicapa, y los lenguajes para especificar las reglas de adaptación que gobiernan el sistema en tiempo de ejecución son limitados.

En esta tesis, proponemos una solución basada en modelado que supera las limitaciones de los estudios existentes para soportar el diseño, despliegue, y gestión de sistemas IoT autoadaptables. Nuestra solución se divide en dos etapas:

Modelamiento (tiempo de diseño): para soportar las tareas de diseño, proponemos un Lenguaje de Dominio Especifico (DSL) que permite representar la arquitectura multicapa del sistema IoT, el despliegue de aplicaciones basadas en contenedores, y las reglas para la autoadaptación en tiempo de ejecución. Adicionalmente, diseñamos un generador de código que produce manifiestos YAML para el despliegue y gestión del sistema IoT en tiempo de ejecución.

Autoadaptación (tiempo de ejecución): hemos diseñado un framework basado en el ciclo MAPE-K para monitorear y adaptar el sistema IoT, siguiendo las reglas especificadas en el modelo (tiempo de diseño). El despliegue y configuración de las herramientas y tecnologías usadas por este framework se realiza mediante los manifiestos YAML producidos por el generador de código.

Adicionalmente, hemos diseñado dos extensiones de nuestro DSL demostrando su capacidad de extensibilidad. La primera, es una extensión enfocada al modelado de sistemas IoT desplegados en la industria de minería subterránea de carbón. Este DSL aborda nuevos conceptos del dominio minero como la especificación de la estructura de las minas subterráneas y los puntos de control. La segunda extensión del DSL está enfocada en el modelado de sistemas IoT desplegados en Plantas de Tratamiento de Aguas Residuales (PTAR). Esta extensión del DSL aborda el modelado del diagrama de bloques de procesos de la PTAR usando una notación gráfica. Incluso con estas dos extensiones del DSL, no es necesario modificar nuestro framework que gestiona la adaptación del sistema en tiempo de ejecución.

Finalmente, hemos realizado tres estudios experimentales: (1) validamos la expresividad y facilidad de uso del DSL mediante experimentos con 13 participantes que realizaron ejercicios de modelado (usando el DSL) y respondieron encuestas reportando su experiencia; (2) validamos funcionalmente las adaptaciones arquitecturales que nuestro framework realiza, comparando el desempeño y disponibilidad entre un sistema no autoadaptable y un sistema autoadaptable que implementa nuestra solución; (3) por último, evaluamos la capacidad y el desempeño de nuestro framework para abordar el aumento de adaptaciones concurrentes en un sistema IoT. Los resultados de estos experimentos demuestran que (1) el DSL tiene la expresividad para modelar sistemas IoT multicapa (incluyendo su comportamiento autoadaptable) y la curva de aprendizaje es favorable; (2) las pruebas funcionales demuestran como el desempeño y disponibilidad del sistema mejora al usar nuestra solución; y (3) hemos identificado limitaciones de escalabilidad del framework y hemos propuesto sugerencias para abordarlas.

Palabras clave: Internet de las cosas, Ingeniería Basada en Modelos, Sistema Autoadaptable, Lenguaje de Dominio Específico, Computación de Borde y Niebla

Acknowledgements

The development of this thesis has been possible thanks to the support of my family, friends, and colleagues. To each and every one, thank you.

First and foremost, I would like to thank my three mentors Kelly, Harold, and Jordi. Thank you for giving me the opportunity to do a PhD thesis under your supervision, for trusting me, and for dedicating your time to me. I am very grateful to Kelly and Harold for their advice from the beginning of the thesis. They have shaped me as a researcher and have taught me to love this beautiful work. To Jordi, I want to thank him for his advice, guidance, patience, severity, and friendship. Thank you for the opportunity to meet and join your great working group (*SOM Research Lab*) in Barcelona.

I would also like to thank the jury for their time and effort in evaluating this dissertation. Your comments, questions, and suggestions will be used to further improve this thesis.

I would also like to thank my friends at *Software Evolution Lab* and *SOM Research Lab* for all their collaboration in this research, and support as friends and colleagues. I especially want to thank Olga Vega, Jaime Chavarriaga, Edouard Batot, Marc Oriol, Marcos Gómez, Abel Gómez, Joan Giner and the others for their friendship and support.

This dissertation have been partially funded by a PhD Scholarship of MINCIENCIAS (*Becas del Bicentenario* program, 2018). It have been also funded by a teaching assistance of Universidad de los Andes and a joint-project (European TRANSACT Project) with SOM Research Lab, UOC. I am very grateful with all these institutions for the opportunity they gave me.

I would also like to thank all my family: Jesus, Marlen, Andrea, Andres, Marien, and Maleja. Thank you for helping and supporting me during these years of work. Without your trust and love this would not have been possible.

Last but not least, I would like my wife Karen for her love, patience, and support in all the difficult and happy moments. Gracias por estar siempre a mi lado.

Contents

Chapter 1

Introduction

The emergence of the Internet of Things (IoT) has dramatically changed how physical objects are conceived in our society and industry. In this new IoT era, every object becomes a complex cyber-physical system (CPS) [87], where both the physical characteristics and the software that manages them are highly intertwined. More specifically, IoT is defined by the International Telecommunication Union (ITU) 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" [155].

Although the term IoT was first used by Kevin Ashton in 1999 to describe a system where the physical world is connected to the Internet through sensors and RFID (Radio Frequency Identification) technologies, years ago other projects were already addressing the connection of devices to the Internet.

In **1982**, the first thing was connected to the Internet. At Carnegie Mellon University, a Coca-Cola machine was connected to the Internet to check the availability and temperature of drinks. In **1990**, a toaster machine was connected to the Internet via TCP/IP protocol to monitor its usage time. In **1993** students at Cambridge University connected the first camera to monitor the availability of cage in the department's machines. In **1999**, the same year that the term IoT was coined, device to device communications was introduced by Bill Joy in his taxonomy of the Internet.

In **2000**, the popularity of wireless connections begins to grow together with the number of objects connected to the internet. In this year, LG launches the first internet-connected refrigerator. **2008** was the first year in which the number of devices connected to the Internet exceeded the number of people connected. Although the term IoT was already being used in closed communities, Kevin Ashton first introduced it in his paper tittled *That "Internet of Things" Thing* [12]. Since then, IoT has had quite a big growth, and a lot of startups and companies working in this area emerged. Today manufacturing, transportation, agriculture, healthcare and other domains have benefited from the use of IoT technologies. Industry is one of the sectors highly motivated to invest in IoT to improve business processes, automate tasks, and offer better customer experiences.

This evolution of IoT systems throughout history has also involved the design of new technologies and architectures to meet increasingly restrictive requirements and improve Quality of Service (QoS). For example, the use of multi-layered architectures leveraging edge and fog computing, or the system's ability to self-adapt and cope with unexpected changes are requirements that can be critical to ensure its performance. However, designing and managing these IoT systems are challenging and complex tasks that can be extremely time and effort consuming. In this thesis, we investigate how to support the design and operation of multilayered IoT systems and its self-adaptive behaviour.

1.1 Motivation

The ideas behind the IoT have been especially embraced by industry in the so-called Industrial IoT (IIoT) or Industry 4.0. Currently billions of devices are connected with potential capabilities to sense, communicate, and share information about their environment. Traditional IoT systems rely on cloud-based architectures, which allocate all processing and storage capabilities to cloud servers. Although cloud-based IoT architectures have advantages such as reduced maintenance costs and application development efforts, they also have limitations in bandwidth and communication delays [89]. Given these limitations, edge and fog computing have emerged with the goal of distributing processing and storage close to data sources (i.e. things). Today, developers tend to leverage the advantages of edge, fog, and cloud computing to design multi-layered architectures for IoT systems.

Nevertheless, creating such complex designs is a challenging task. Even more challenging is managing, and adapting IoT systems at runtime to ensure the optimal performance of the system while facing changes in the environment conditions. IoT systems are commonly exposed to changing environments that induce unexpected events at runtime (such as unstable signal strength, latency growth, and software and hardware aging) that can impact its QoS. To deal with such events, a number of runtime adaptations should be automatically applied, e.g. *architectural adaptations* such as auto-scaling and offloading tasks.

In this sense, a better support to define and execute complex IoT systems and their (self)adaptation rules to semi-automate the deployment and evolution process is necessary [133]. One of the most widely used approaches to deal with the complexity of these large systems is Model-Driven Engineering (MDE) [75]. The use of models allows to raise the level of abstraction and capture the aspects of interest of a real system. This helps to avoid the complexity and uncertainty of real and complex scenarios.

Models are built using domain-specific languages (DSLs). In short, a DSL offers a set of abstractions and vocabulary closer to the one already employed by domain experts, facilitating the modeling of new systems for that domain. Nevertheless, current DSLs for IoT do not typically cover multi-layered architectures [130, 67, 72, 43] and even less include a sublanguage to ease the definition of the dynamic rules governing the adaptability of the IoT system.

This thesis is focused on the design of a model-based approach to support the deployment and self-adaptation of multilayer IoT systems. The main research activities include a systematic literature review (SLR) to analyze the state-of-the-art, the design and development of our approach for modeling and managing self-adaptive IoT systems, and experimental studies to validate our approach. This approach is composed of a DSL to model the multi-layered IoT architecture and its self-adaptation, a code generator, and a framework to monitor and support system adaptations at runtime.

1.2 Problem Statement

To tackle the complexity of designing and implementing self-adaptive IoT systems based on multi-layer architectures, this thesis addresses two research questions that remain as a challenge despite the related work. These research questions have been carefully studied throughout the course of this dissertation

RQ1 How to model multi-layer IoT systems including their adaptation scheme to ensure their run-time operation in changing environments?

RQ2 How to manage real-time adaptations for multilayered IoT systems operating in changing environments?

1.3 Contributions

The main contribution of this thesis is a model-based approach for the deployment and management of multilayer IoT systems with self-adaptive capabilities. Specifically, it consists of:

C1 **DSL**. We have designed a new DSL for IoT systems focusing on three main contributions: (1) modeling of multi-layer architectures of IoT systems, including concepts such as IoT devices (sensors or actuators), edge, fog and cloud nodes; (2) modeling the deployment and grouping of container-based applications on those nodes; and (3) a specific sublanguage to express adaptation rules to guarantee QoS at runtime, availability, and performance. We have implemented this DSL using a projection-based editor that allows mixing various notations to define the concrete syntax of the language. In this way, the IoT system can be specified by a model containing text, tables, and graphics.

C2 **Code generator**. The model (built using the DSL) describing the self-adaptive IoT system is the input to a code generator we have designed. This generator produces several YAML[1] manifests with two purposes: (1) to configure and deploy the IoT system container-based applications and (2) to configure and deploy the tools and technologies used in the framework that supports the execution and adaptation of the system at runtime.

C3 **Runtime framework**. We have developed a framework to monitor and adapt the IoT system following the adaptation plan specified in the model. This framework is based on the MAPE-K [92] loop (a reference model to implement adaptation mechanisms in auto-adaptive systems) composed of several stages including system status monitoring, data analysis, action planning, and execution of adaptations.

C4 **DSL extensions**. We propose two extensions of our DSL focused on the modeling IoT systems for two different domains: (1) a DSL extension focused in the IoT systems deployed for underground mining industry, and (2) a DSL extension for specifying IoT systems for Wastewater Treatment Plants (WWTPs). These extensions involve the definition of new domain-specific concepts (e.g. mine and tunnel concepts for the mining industry), and the design of graphical editors to model specific scenarios.

C5 **Empirical evaluations**. We have designed and conducted experimental studies classified in three categories: (1) two experimental studies to validate the expressiveness and usability of our DSL extended to the mining domain (13 participants attended the experiment), (2) three experimental studies to test the self-adaptive capability of our approach, and (3) two set of experimental studies to identify approach scalability to perform concurrent adaptations of our MAPE-K based framework to adapt the IoT system at runtime. We present here a report of our experience including the results of the experiments.

[1]YAML is a data serialization language typically used in the design of configuration files

Table 1.1: Organization of the Document

Introduction	Ch 1	Introduction
State of the Art	Ch 2	State of the Art
Our Approach	Ch 3	Overview
	Ch 4	Modeling Self-adaptive IoT Architectures
	Ch 5	Adapting IoT systems at runtime
	Ch 6	Extending DSL for specific cases
	Ch 7	Experimental Validation
Related Work	Ch 8	Related Work
Conclusions	Ch 9	Conclusions

1.4 Organization of the Document

This thesis document is organized in five parts. (1) there is an Introduction, (2) the State of The Art, (3) the description of our approach, (4) related work, and (5) a final part with the Conclusions. Table 1.1 shows the structure and chapters.

Chapter 2 (State of the Art) presents the backgroud information that has served as the basis for this dissertation, and an SLR to obtain a comprehensive view of the overall topic and identify open challenges.

Our Approach is presented in five chapters. Chapter 3 introduces an overview of our approach. Chapter 4 discusses our DSL for modeling multi-layered IoT systems and its self-adaptive behavior. In Chapter 5, we describe the framework that supports and manages the self-adaptive IoT system at runtime. Chapter 6 presents two extensions of our DSL for modeling IoT systems in specific domains and Chapter 7 describes the validation of our approach through empirical experiments.

Chapter 8 discusses related work on methods for designing, deploying, and managing multilayer IoT systems with self-adaptive capabilities. Finally, Conclusions are presented in Chapter 9, including future directions.

Chapter 2

State of the Art

Over the past few years, the relevance of the Internet of Things (IoT) has grown significantly and is now a key component of many industrial processes and even a transparent participant in various activities performed in our daily life. IoT systems are subjected to changes in the dynamic environments they operate in. These changes (e.g. variations in the bandwidth consumption or new devices joining/leaving) may impact the Quality of Service (QoS) of the IoT system. A number of self-adaptation strategies for IoT architectures to better deal with these changes have been proposed in the literature. Nevertheless, they focus on isolated types of changes. We lack a comprehensive view of the trade-offs of each proposal and how they could be combined to cope with dynamic situations involving simultaneous types of events.

In this chapter, we identify, analyze, and interpret relevant studies related to IoT systems adaptation and develop a comprehensive view of the interplay of different dynamic events, their consequences on the architecture QoS, and the alternatives for the adaptation. To do so, we have conducted a Systematic Literature Review (SLR) of existing scientific proposals and defined a research agenda based on the findings and weaknesses identified in the literature.

This SLR was the first research activity conducted in this doctoral thesis and the results obtained were the inspiration to define our research objectives.

Rest of this chapter is organized as follows: Section 2.1 present an overview of the concepts related to IoT, self-adaptive systems, and Model-Driven Engineering (concepts used in this and other chapters of this document). Section 2.2 presents the SLR method and findings, and Section 2.3 summarizes the chapter.

2.1 Preliminary Concepts

This section covers the basic concepts involved in this research. Section 2.1.1 introduces the concept of the Internet of Things (IoT) and presents some generalities about its architecture. Self-adaptive systems including the MAPE-K loop are described in Section 2.1.2. Finally, in Section 2.1.3 we introduce the concepts Model-Driven Engineering (MDE) and domain specific language (DSL) that serve as a basis to design our solution.

2.1.1 Internet of Things

The term IoT has acquired several definitions since it was coined in 1999 by Kevin Ashton [12]. All these definitions agree that IoT aims to exchange information between things-people, people-people, or things-things. According to Madakam et al., the best definition of IoT is *"An open and comprehensive network of intelligent objects that have the capacity to auto-organize, share information, data, and resources, reacting and acting in face of situations and changes in the environment"* [104]. IoT has transformed and improved the activities we carry out on a daily basis in various aspects such as transport, agriculture, healthcare, industrial automation, and emergency response.

The increase in IoT applications and connected devices in recent years requires improved solutions to meet the requirements of the system and data management [141]. Usually, the cloud is used in the IoT systems for data storage and processing. However, cloud servers are often located far from data sources or IoT devices causing system delays and performance problems mainly for real-time IoT applications. Distributed or multi-layered architectures have emerged to reduce dependence on the cloud and meet non-functional IoT system requirements. In particular, edge computing and fog computing are solutions that propose to bring processing and storage closer to data sources (IoT devices).

There are similarities between these two solutions, but the main difference is the location into the IoT network where processing and storage of data can be performed [53]. Edge computing focuses more on the things side (sensors and actuators), while fog computing focuses more on the infrastructure side [145]. Figure 2.1 shows the architecture of a distributed IoT system where fog computing is performed on the system's fog nodes while edge computing is performed on edge nodes, e.g. gateways. Edge and fog computing moves computation, storage, communication, control, and decision making closer to the network edge where data is being generated; i.e., in the device layer. The combination of edge computing, fog and cloud results in distributed (multi-layered) architectures to ensure QoS compliance in IoT applications. Fog and edge computing offer advantages mainly in terms

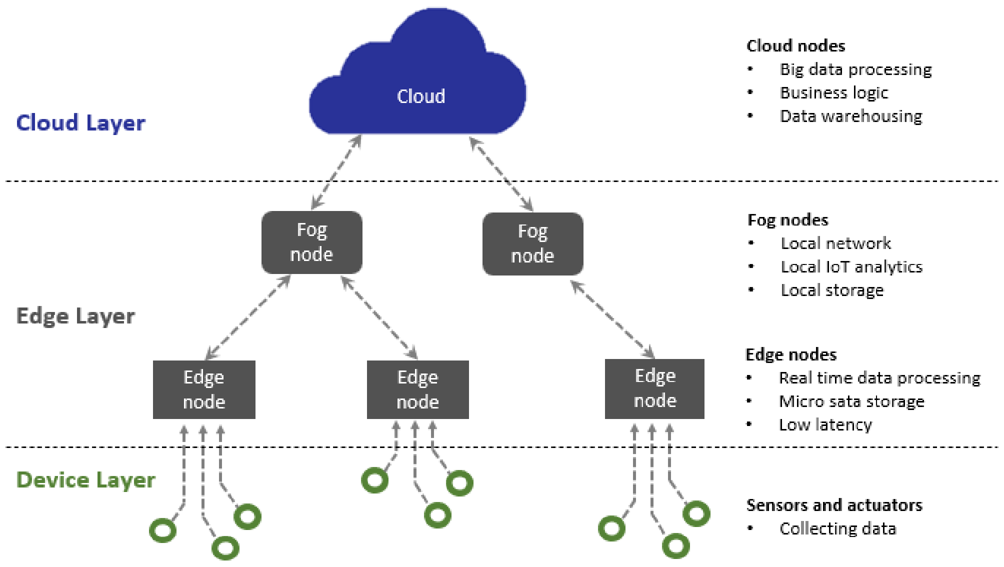


Figure 2.1: Distributed architecture for IoT systems

of latency and bandwidth usage, since they allow data processing at the edge rather than in the cloud.

Communication

Communication technologies and standards used in IoT systems can be classified into two groups according to the application level [2]: device-level communication, and application-level communication.

Device level. The selection of communication protocols and technologies at the device and edge layer, i.e. between IoT devices and gateways, depends on several aspects such as communication range, data transmission rate, power consumption, network topology, interoperability, and security. The communication protocols in this group can be categorized according to the coverage area [3]: (1) Low Power Wide Area Network (LPWAN) and (2) short range network. LPWAN encompasses wireless communication technologies that allow data to be transmitted between a device and a base station/Gateway separated by hundreds of meters or kilometers with very low power consumption. LPWAN encompasses communication protocols such as SigFox, LoRaWAN, and Cellular technologies. On the other hand, short range networks are implemented in local networks such as residential areas, commercial buildings, or greenhouses. Communication protocols implemented in such networks include WiFi, Zigbee, Bluetooth, or NFC.

Application level. In terms of communication standards and technologies for the application layer, there are two communication models that span different protocols [50]: request-response and publish-subscribe. Request-response represents a message exchange pattern commonly used in client-server architectures (synchronous communication), where the client makes a request to the server, which processes some information and returns a response to the client. The most commonly used request-response protocols are REST HTTP and CoAP. On the other hand, publish-subscribe model offers a loosely coupled asynchronous communication between data generators and destinations. Three components are involved in a publish-subscribe communication (Figure 2.2): the **publisher** (e.g. a sensor) sends information to the topics of a **broker** (central point of this architecture), which routes and retransmits the information to the **subscriber**s (e.g. a server or an actuator) that are subscribed to the broker topics. This model is today one of the most popular for message-driven asynchronous architectures [73]. Publish-subscribe model encompasses communication protocols such as MQTT, CoAP, AMQP, DDS, and XMPP.

One of the most popular application layer protocols in IoT systems is MQTT (Message Queuing Telemetry Transport) [173, 50]. This protocol is suitable for devices with restricted memory capabilities and limited processing power. MQTT has very small message header compared to other publish-subscribe protocols, and due to its simplicity of use, it has become one of the most prominent protocols in restrictive IoT systems.

2.1.2 Self-adaptive Systems

Systems commonly address functional and non-functional requirements at design and development time. However, knowledge at design time is sometimes limited to deal with unpredictable or uncertain circumstances. Therefore, designers often prefer to deal with this uncertainty at run-time, when more knowledge is available [105]. This unpredictable behavior of the system can be addressed in run-time by self-adaptations of the system. A self-adaptive system modifies its behavior at run-time in response to changes in the system or its environment [38] (such as system failures, environment changes, and new requirements). System adaptations are given to meet both functional requirements (specific system functions) and non-functional requirements (such as performance, usability, security, availability, and others). Assurance of these requirements is commonly achieved through feedback loops that follow a set of steps to apply self-adaptations to the system. For example, monitoring is one of the key stages in self-adaptive systems.

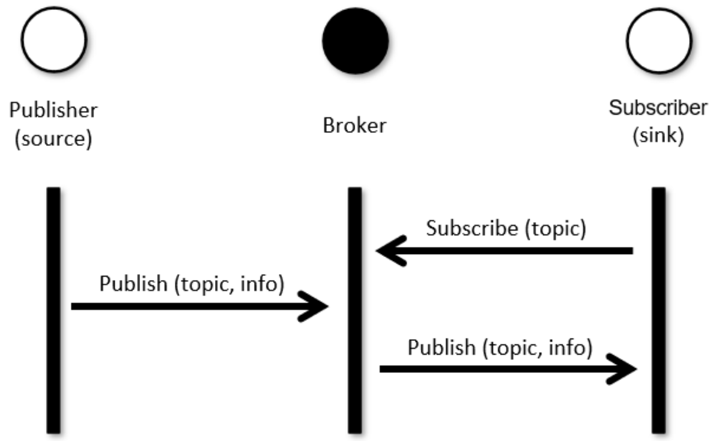


Figure 2.2: Publish-subscribe model

The MAPE-K loop, proposed by IBM for autonomous computing [92], has been implemented in several studies for the design of self-adaptive systems. MAPE-K is a reference model to implement adaptation mechanisms in auto-adaptive systems. MAPE-K includes four activities (monitor, analyze, plan, and execute) in an iterative feedback cycle that operate on a knowledge base (see Figure 2.3). These four activities produce and exchange knowledge and information to apply adaptations due to changes in the managed element.

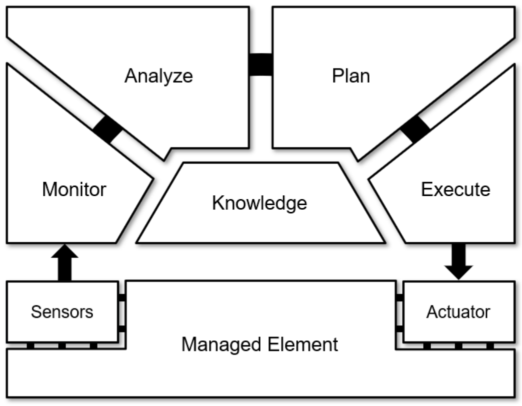


Figure 2.3: The MAPE-K loop [135]

MAPE-K loop activities are described below.

- Monitor: information about the current state of the system is collected, aggregated, filtered, and reported. Data such as functional and non-functional system properties are collected.

- Analyze: in this stage the information collected in the monitoring phase is analyzed, and changes in the system that require adaptations are identified. The analyzer uses policy information to determine when a change is required. If a change is required, the analyzer sends a change request to the next activity component (plan).

- Plan: according to the analysis made in the previous stage, an adaptation plan is generated with the appropriate actions to adapt the system at run-time. The adaptation plan contains tasks that could be either a complex workflow or simple commands. This adaptation plan is sent to the execution component of the next stage.

- Execute: the adaptations are applied to the system following the actions defined in the adaptation plan.

2.1.3 Model-Driven Engineering

To better understand and analyze the characteristics of complex domains such as software systems and their application domains, the definition of abstractions is often used. Models are abstractions or generalized representations of a real world system. Model-Driven Engineering (MDE) is focused on the use of models at different levels of abstraction for software engineering activities [26]. For example, MDE combines languages and model transformations for the automated generation of software artifacts [156], improving productivity and quality in software development processes.

Models and transformations are the two core aspects of MDE. Models are specified by a model notation or language known as *Domain-Specific Language* (DSL). DSLs are based on a metamodel that captures the essential concepts of the domain. A model transformation is a mapping that takes a source model and generates an objective model following transformation rules. Commonly a model transformation chain results in the generation of system artifacts (e.g. source code and configuration files). There are three types of model transformations: model-to-model transformations (m2t), text-to-model transformations (t2m), and model-to-text transformations (m2t). In this thesis, we will implement m2t transformations to automatically generate software artifacts from a model according to a metamodel.

Models and Metamodels

In the case of software engineering, models represent aspects of a software system such as requirements, structure, architecture, behavior, or deployment. The purposes in defining these models are to support the construction of the system, to provide documentation that transcends the project, to facilitate communication between the system that transcends the project, facilitate communication between developers, and automatically generate code for the system implementation [65].

Models are defined following a metamodel structure. A metamodel specifies the concepts of a language, the relationships between these concepts, and the structural rules that constrain and validate the correctness of a model. Ecore [150] is currently the most widespread metamodeling language (based on Meta-Object Facility MOF[1]), which provides the basic concepts for creating metamodels: concepts are represented as classes, properties of a class refer to attributes of the concept, and relationships between classes represent associations.

Domain-Specific Languages (DSLs)

DSLs are designed to describe or model things in a specific domain, context, or company [26]. DSLs address the needs of a specific application domain which cannot be covered by a *General-Purpose Language* (GPL). Some advantages of DSLs are (1) the abstractions supplied to represent particular concepts of domain application, and (2) the natural notation for a given domain and the avoidance of the syntactic disorder that usually occurs when using a GPL [45]. However, a lot of time and effort can be consumed to develop a DSL. This involves tasks that require expertise in both domain and language development.

According to Van Deursen et al. [157] the development of a DSL involves 7 tasks distributed in three main stages. The first stage is *analysis*, which includes the tasks of (1) identifying the problem, (2) gathering relevant domain information, (3) representing domain knowledge in semantics and operations, and (4) designing the DSL. The second stage is *implementation*, which involves the tasks of (5) build a library that implements semantic notions, and (6) design and implement a compiler that translates DSL programs into a sequence of library calls. Finally, stage three (*use*) has a single task, write and compile DSL programs for the desired applications.

A DSL is composed of three elements: the abstract syntax, the concrete syntax, and the semantics. The **abstract syntax** is usually represented by a metamodel that defines the concepts of the domain, relationships between concepts, and wellformedness rules that constrain and validate the model. The **concrete syntax** defines the notation (graphical, textual, or hybrid) for the abstract syntax. The

[1]https://www.omg.org/mof/

notation used by users to define models can greatly impact the usability of the language. To represent concepts with a graphic DSL, graphic objects such as connectors, blocks, axes, arrows, and others are used. In contrast, a textual DSL is based on grammar, e.g., SQL is a textual DSL used to perform database hides.

2.2 Analysis of IoT System Adaptation

In most IoT systems, it is critical to guarantee the QoS to the users, according to the requirements of the application domain. For example, in continuous monitoring systems, a decrease in the quality could generate wrong or late alerts stemming from the monitored system (imagine the effects of late alerts in monitoring systems in hospitals). However, satisfying these commitments is challenging due to the dynamic nature of the environment surrounding the IoT system. All types of unexpected events (such as unstable signal strength, growth in the number of connected devices, and software and hardware aging [30, 123]) can happen at any time, posing a risk to the QoS. This unpredictable behavior of the system can be addressed at runtime by self-adaptive systems.

Most studies found in the literature individually address particular dynamic events in IoT systems and propose specific strategies to ensure QoS. However, each proposal provides only a partial view (and solution) to the self-adaptation problem. It is necessary to have a comprehensive view of all the different kinds of events (for example, environmental) addressed in the literature. Indeed, we need: (1) a classification of the dynamic events that impact the QoS; (2) a classification of the self-adaptation strategies of the IoT system architecture; and (3) gaps and challenges in the proposed strategies and their relationships.

In this sense, we decided that an SLR was the best way to systematically reach to a comprehensive and fair assessment of these topics. The results of this SLR provide an overview of the gaps and challenges that inspired the definition of the research objectives of this thesis.

2.2.1 Method

A systematic literature review (SLR) is a methodology used for the identification, analysis, and interpretation of relevant studies to address specific research questions [91]. Our SLR consists of six main steps and is based on the methodology proposed by Kitcheham et al. [94]. The steps followed for this SLR are illustrated in Figure 2.4 and documented below.

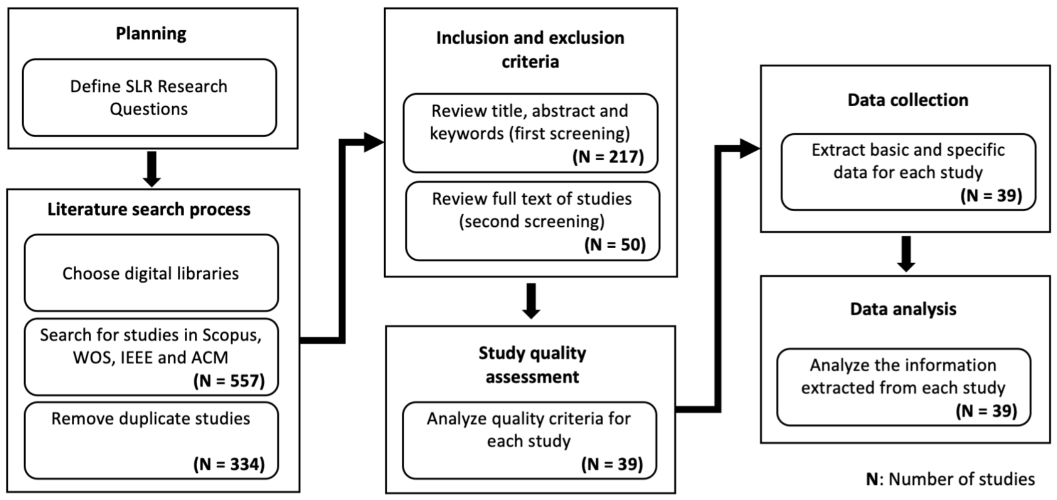

Planning
Define SLR Research Questions
Literature search process
Choose digital libraries
Search for studies in Scopus, WOS, IEEE and ACM (N = 557)
Remove duplicate studies (N = 334)
Inclusion and exclusion criteria
Review title, abstract and keywords (first screening) (N = 217)
Review full text of studies (second screening) (N = 50)
Study quality assessment
Analyze quality criteria for each study (N = 39)
Data collection
Extract basic and specific data for each study (N = 39)
Data analysis
Analyze the information extracted from each study (N = 39)
N: Number of studies

Figure 2.4: Application of the SLR process

Research questions

Our goal is to identify the dynamic environmental events in the device and edge/fog layers of an IoT system that could impact its QoS and therefore require the trigger of self-adaptations of the system. In addition, we classify the strategies to achieve this self-adaptation. For this purpose, our SLR addresses the following two research questions:

- SLR-RQ1. Which dynamic events present in the edge/fog and device layers are the main causes for triggering adaptations in an IoT system?
- SLR-RQ2. How do existing solutions adapt their internal behavior and architecture in response to dynamic environmental events in the edge/fog and device layers to ensure compliance with its non-functional requirements?

Literature search process

The search process step had three phases [94]: first, we selected the digital libraries; next, we defined the search queries; and finally, we carried out the search and discarded the repeated studies.

- Digital libraries: we chose four digital libraries for our search: Scopus, Web

Table 2.1: Search queries

	Search query
SQ1	("fog" OR "edge" OR "osmotic") AND ("IoT" OR "internet of things" OR "cyber-physical") AND ("architecture") AND ("adapt*" OR "self-adapt*")
SQ2	"fog" AND "adapt*" AND "architecture" AND "orchestration"
SQ3	("orchestration" OR "choreography") AND "fog" AND "architecture" AND "dynamic"

Table 2.2: Studies per digital library

Digital library	Studies found
Scopus	229
Web of Science (WOS)	120
IEEE	176
ACM	32
Total	557
Total without duplicates	334

of Science (WOS), IEEE Explore, and ACM. These libraries are frequently updated and contain a large number of studies in the area of this research.

- Search queries: as shown in Table 2.1, we defined four search queries. We used keywords including IoT, architecture, dynamic, adapt (or variations of this word; e.g., adaptation), fog and edge (to retrieve studies that use distributed architectures with fog and edge computing), orchestration or choreography (two resource management techniques in the fog layer of an architecture). We looked for matches in the title, abstract, and keywords of the articles.

- Search results: Table 2.2 shows the search results; we obtained 557 studies, out of which 223 were duplicates, for a total of 334 studies.

Inclusion and exclusion criteria

To screen and obtain the primary studies that address the research questions, we defined inclusion and exclusion criteria. We applied two screening phases: in the first screening of the titles, abstracts and keywords, we used three exclusion criteria, to exclude 117 out of the 334 studies. Then, in the second filter we analyzed the full texts, and we discarded 170 additional studies. Finally, using Snowballing to check the list of study citations we included three additional studies, for a total of 50 studies (see Figure 2.4). The inclusion and exclusion criteria for each screening phase are presented below.

First screening:

- (Exclusion) It is not a primary study. Literature reviews are discarded.
- (Exclusion) It is not a journal, conference or workshop paper.
- (Exclusion) The paper is written in a language other than English

Second screening:

- (Inclusion) The study addresses a dynamic event in IoT systems that impacts QoS.
- (Inclusion) The study proposes, takes advantage or analyzes a strategy of self-adaptation of architecture for IoT systems.

Quality assessment

The quality assessment step consists of reading the studies in detail, and answering the assessment questions to get a quality score for each study. We have defined five quality assessment questions as follows:

- QA1. Are the aims clearly stated? (Yes) the purpose and objectives of the research are clear; (Partly) the aims of the research are stated, but they are not clear; (No) The aims of the research are not stated, and these are not clearer to identify.
- QA2. Is the research compared to related work? (Yes) the related work is presented and compared to the proposed research; (Partly) the related work is presented, but the contribution of the current research is not differentiated; (No) the related work is not presented.
- QA3. Is there a clear statement of findings and do they have theoretical support? (Yes) the findings are explained clearly and concisely, and are supported by a theoretical foundation; (Partly) the findings are clearly explained, but they lack theoretical support; (No) findings are not clear and have no foundation or theoretical support.

- QA4. Do the researchers explain future implications? (Yes) the author presents future work; (No) future work is not presented.
- QA5. Has the proposed solution been tested in real scenarios? (Yes) The solution is tested in a real scenario; (Partly) the solution is tested in a particular test bed; (No) the solution is not tested in any scenario.

The score given to each answer was: Yes = 1, Partly = 0.5, and No = 0. We calculated the quality score for each study and excluded those that scored less than 3, in order to select the primary studies that would be used for data extraction and analysis. We analyzed 50 studies and excluded eleven because they obtained a quality score of less than three. In total we have obtained 39 primary studies for the remaining steps of this SLR, and the quality scores for each is presented in Table 2.4. In the remainder of this chapter, we reference these studies in the text by their assigned ID in the table.

Data collection

The extracted information was stored in an Excel spreadsheet. Table 2.3 shows the Data Collected (DC) for each study and the research question addressed. First, we extracted standard information such as title, authors, and year of publication (DC1 to DC4). Second, we extracted relevant information to address the research questions defined in section 2.2.1. DC5 records the environmental event addressed by the study, and this information is used to address research question SLR-RQ1. DC6 to DC10 are data collected about proposed solutions and strategies to achieve self-adaptations in the IoT system, and this information is used to address research question SLR-RQ2.

Data analysis

Table 2.4 presents the list of the 39 studies relevant to this SLR, with the following information: the assigned identification number (ID), the author, the type of publication, the year of publication, the answers to the quality questions, and the quality score obtained. In the following sections, we will refer to primary studies by the assigned ID code.

From the standard information extracted from the papers, we can note that the relevant publications for this SLR are relatively recent. The largest number of studies were published in recent years: 12 studies from 2019, 16 studies from 2018, 7 studies from 2017, 3 studies from 2016, and one study from 2015. As to the type of publication, 25 are conference publications, 10 are journal publications, and 4 are workshop publications.

Table 2.3: Data collection

#	Field	RQ
DC1	Author	N/A
DC2	Title	N/A
DC3	Year	N/A
DC4	Publication venue	N/A
DC5	Environmental event addressed by the solution	SLR-RQ1
DC6	Favored quality attributes	SLR-RQ2
DC7	Adaptation strategies and techniques	SLR-RQ2
DC8	Architecture description	SLR-RQ2
DC9	Architectural styles and patterns	SLR-RQ2
DC10	Key responsibilities of architectural components	SLR-RQ2

2.2.2 List of dynamic events

Table 2.5 provides an overview of the answer to the research question SLR-RQ1. The table presents a classification of the dynamic environmental events present in the edge/fog and device layers and the studies that addressed that event. We propose this list of events that we have obtained from the detailed analysis of the studies. We then classify each study according to the event it addresses. Strategies for adapting architecture in response to these events are presented in Section 2.2.4.

E1. Client mobility

Mobile devices such as cell phones or automobiles produce events in the device layer of the IoT system causing challenges to ensure QoS. When the system devices change their location, it is necessary to make network re-configurations, storage synchronizations, and rescheduling processes among the edge/fog nodes by taking into account available resources. Client mobility is an event or requirement of IoT systems that poses challenges due to the constant movement of devices, the heterogeneity of communication technologies, and resources which can be requested on demand simultaneously by multiple devices in different locations [139].

E2. Dynamic data transfer rate

The data transmission rate of the devices is another dynamic event that significantly influences the system's QoS. In IoT systems, the data transmission rate from the device layer to the edge/fog layer may vary depending on the circumstances, objects,

Table 2.4: Studies

ID	Author	Type	Year	QA1	QA2	QA3	QA4	QA5	QA Score
S1	Young, R. et al. [176]	Conference	2018	Y	Y	Y	Y	P	4,5
S2	Wang, J. et al. [163]	Workshop	2017	Y	Y	Y	N	P	3,5
S3	Muñoz, R. et al. [118]	Article	2018	Y	Y	Y	N	P	3,5
S4	Cheng, B. et al. [37]	Conference	2015	Y	Y	Y	Y	N	4
S5	Kimovski, D. et al. [93]	Conference	2018	Y	Y	Y	Y	P	4,5
S6	Young, R. et al. [175]	Conference	2018	Y	Y	Y	Y	P	4,5
S7	Tseng, C. et al. [154]	Conference	2018	Y	N	Y	Y	P	3,5
S8	Peros, S. et al. [126]	Conference	2018	Y	Y	Y	Y	P	4,5
S9	Rausch, T. et al. [131]	Conference	2018	Y	Y	Y	N	Y	4
S10	Pahl, C. et al. [121]	Conference	2018	Y	Y	Y	N	P	3,5
S11	Lorenzo, B. et al. [103]	Article	2018	Y	Y	Y	Y	N	4
S12	Prabavathy, S. et al. [129]	Article	2018	Y	Y	Y	Y	P	4,5
S13	Yigitoglu, E. et al. [174]	Conference	2017	Y	Y	Y	Y	P	4,5
S14	Morabito, R. et al. [114]	Workshop	2017	P	P	Y	Y	P	3,5
S15	Desikan, K. S. et al. [48]	Workshop	2017	Y	Y	Y	N	P	3,5
S16	de Brito, M. S. et al. [46]	Conference	2017	Y	Y	Y	Y	N	4
S17	Velasquez, K. et al. [158]	Conference	2017	Y	P	Y	Y	P	4
S18	Flores, H. et al. [63]	Conference	2017	Y	N	Y	Y	P	3,5
S19	Pizzolli, D. et al. [128]	Conference	2016	Y	N	P	Y	P	3
S20	Montero, D. et al. [113]	Conference	2016	Y	Y	Y	Y	P	4,5
S21	Chen, L. et al. [34]	Article	2018	Y	Y	Y	Y	Y	5
S22	Mass, J. et al. [108]	Conference	2018	Y	Y	Y	Y	Y	5
S23	Li, X. et al. [100]	Article	2018	Y	Y	Y	N	Y	4
S24	Suganuma, T. et al. [151]	Article	2018	Y	Y	Y	Y	Y	5
S25	Deng, G. et al. [47]	Conference	2018	Y	Y	Y	N	Y	4
S26	Sami, H. et al. [137]	Conference	2018	Y	Y	Y	Y	Y	5
S27	Wu, D. et al. [170]	Conference	2019	Y	Y	Y	N	Y	4
S28	Skarlat, O. et al. [148]	Conference	2019	Y	Y	Y	Y	Y	5
S29	Mechalikh, C. et al. [110]	Conference	2019	Y	Y	Y	Y	Y	5
S30	Castillo, E. et al. [32]	Conference	2019	Y	P	Y	N	Y	3,5
S31	Breitbach, M. et al. [27]	Conference	2019	Y	Y	Y	Y	Y	5
S32	Torres Neto, J. et al. [153]	Article	2019	Y	Y	Y	Y	Y	5
S33	Theodorou, V. et al. [152]	Workshop	2019	Y	N	Y	Y	P	3,5
S34	Guntha, R. [78]	Conference	2019	Y	Y	Y	N	P	3,5
S35	Jutila, M. [90]	Article	2016	Y	Y	Y	Y	Y	5
S36	Cui, K. et al. [44]	Conference	2019	Y	Y	Y	Y	Y	5
S37	Bedhief, I. et al. [20]	Conference	2019	Y	Y	Y	P	N	3,5
S38	Asif-Ur-Rahman, Md et al. [13]	Article	2019	Y	Y	Y	Y	Y	5
S39	Yousefpour, A. et al. [177]	Article	2019	Y	Y	Y	Y	Y	5

Table 2.5: Dynamic environmental events

ID	Dynamic event	Studies
E1	Mobility client	S5, S9, S10, S17, S19, S20, S22, S29
E2	Dynamic data transfer rate	S3, S6, S7, S11, S15, S18, S19, S21, S26, S32, S39
E3	Important event detected by sensors	S1, S2, S8, S24, S27, S31, S36, S38
E4	Failures and software aging	S4, S13, S14, S16, S28
E5	Network connectivity	S1, S23, S25, S30, S33, S34, S35, S37
E6	Attack from the traffic sensor	S12

or conditions in which the devices are surrounded. The system devices may increase or decrease the frequency of data transmission due to different stimuli. The consequences generated by this dynamic event in IoT systems commonly lead to increased latency and the unavailability of system services, because increased data volume could congest the network and generate bottlenecks. In addition, this dynamic event implies growth in the data to be analyzed or processed by the edge devices, which likely have limited computer resources. Therefore, the edge nodes could be overloaded with processing work until they generate delays, down times, or unavailability.

E3. Important event detected by sensors

When an alert or alarm is generated by sensor data in an IoT monitoring system, a set of tasks is triggered to inform the end user and/or control the emergency. These tasks may increase network, processing, and storage consumption at some layer of the system architecture (device, edge/fog, and cloud). For example, in a smart city when a vehicular accident is detected by video surveillance cameras, new processing tasks begin to run in edge/fog nodes or cloud servers: 1) there are increases in the processing and storage of video taken by surveillance cameras; 2) visual alerts are generated to other drivers on the road; 3) tasks are executed to synchronize street lights to address the emergency and reduce vehicle traffic. System tasks generated by alarms or alerts commonly require additional network, processing, or storage resources.

E4. Failures and software aging

The software embedded in the devices, nodes, and servers of an IoT system needs to be updated and redeployed by developers to fix service errors, improve application performance, improve system security, etc. Some upgrades or deployments of system services and application software may involve adaptations to the layers of the system architecture. First, when new services are deployed at edge/fog nodes, it may be necessary to adapt the bindings (e.g., service registry, network topology) established between the services deployed in the nodes and the components that consume said services in order to ensure the communication. Second, software upgrades are sometimes unsuccessful due to storage, hardware, or connectivity failures. In these cases, the system should detect the problem and fix it. Third, the device layer and edge/fog devices have limited processing capabilities that may bring risks to successful software upgrades. This implies increased latency and, in some cases, unresponsive services.

E5. Network connectivity

According to Muñoz, R. et al. [118], the main network requirements for IoT services are low latency, high-speed traffic, large capacity traffic, and massive connections. Although these requirements depend on the domain of the IoT application, most systems require the fulfillment of at least one of these. IoT systems constantly present variations in network connectivity characteristics that make it difficult to meet network requirements. These variations, mainly present in wireless communications, can generate negative effects on the transmission and reception of data between the system's devices, nodes, and cloud servers: (1) out-of-date information due to communication delays; (2) incomplete information due to intermittent or interrupted communication; (3) unavailability of services or system applications due to lost or broken communication.

E6. Cyber-attacks in IoT applications

Although the security topic was not intentionally addressed in this study, we found the work of Prabavathy et al. [129], which proposes a strategy based on the use of fog computing to detect attacks. The threats that come from the data of the device layer devices towards the edge/fog layers and cloud are events induced by attackers that violate the confidentiality, integrity, and availability of the system. In an IoT system, sensors and actuator devices frequently capture and share personal data from our daily life, detect critical physical variables in industrial processes, and control the vehicular flow in a city. The impact of an attack on the devices in any of the layers of the architecture can cause loss of critical information, disasters in the processes that control the system, and unavailability of the system, among others. This is why it is essential to ensure the security of the IoT system by designing self-adaptation techniques to defend against attacks.

2.2.3 Detecting dynamic events

Monitoring is an important task to detect dynamic events in the IoT systems. These events are detected by analyzing metrics about node resource consumption (such as CPU, memory, and energy consumption), network behavior (such as bandwidth consumption and communication latency), availability, and data collected by sensors. Table 2.6 presents the monitored metrics to detect the dynamic events for each study. The resource consumption in the edge/fog nodes is the most monitored feature to detect events. In particular, CPU and memory consumption are used to detect three of the dynamic events: *Client mobility (E1)*, *Dynamic data transfer (E2)*, and *Failures and software aging (E4)*. Sensor data (column 9) is not a QoS metric, but its analysis is used to detect the dynamic events *Important event detected by sensors*, *Network connectivity*, and *Cyber-attacks in IoT applications*.

Availability and Latency are seldom monitored metrics to detect dynamic events. However, ensuring low latency is one of the important requirements for real-time applications. Similarly, ensuring the availability of services and applications in IoT systems is also a common requirement.

S10, S20, S22, and S29 are not included in Table 2.6 because they do not monitor any QoS metrics. These four studies address the dynamic event client mobility, which they detect by identifying new clients joining or leaving the system. Studies S31 and S36 (also not included in the table) do not focus on the detection of the dynamic event, instead they cover the architectural adaptations to cope with the event.

2.2.4 List of adaptation strategies

Table 2.7, which presents a classification of the strategies used by each study to support specific dynamic events, provides a preliminary answer to the research question SLR-RQ2. Similar to the classification of dynamic events (2.2.2), we propose this list of adaptations after analyzing the studies in detail.

The adaptive strategies are described below.

Data flow reconfiguration

The routing of data traveling from the device layer to the Edge/Fog or cloud layer is modified mainly to improve latency. The direction of the data flow and the de-

Table 2.6: Monitored metrics

Event	Study	CPU	Memory	Storage	Bandwidth	Availability	Latency	Sensor data
E1	S5	X	X					
E1	S9						X	
E1	S17	X	X	X				
E1/E2	S19			X		X		
E2	S3			X				
E2	S6	X						
E2	S7	X						
E2	S11				X			
E2	S15						X	
E2	S18	X	X	X				
E2	S21					X		
E2	S32	X						
E3	S1				X			X
E3	S2							X
E3	S8							X
E3	S24							X
E3	S27							X
E3	S38							X
E4	S4	X	X	X	X	X	X	
E4	S13	X	X	X	X			
E4	S14	X	X	X				
E4	S16	X	X	X				
E4	S28	X	X	X				
E5	S1				X			X
E5	S23						X	
E5	S25				X		X	
E5	S30						X	
E5	S33						X	
E5	S34				X			
E5	S35						X	
E5	S37				X			
E5	S39						X	
E6	S12							X

Table 2.7: Adaptations

ID	Adaptation	Studies
A1	Data flow reconfiguration	S3, S5, S8, S9, S10, S11, S14, S15, S17, S19, S20, S23, S25, S35, S37, S38
A2	Auto Scaling of services and applications	S2, S7, S17, S18, S19, S22, S31, S39
A3	Software deployment and upgrade	S4, S13, S16, S28
A4	Offloading tasks	S1, S6, S21, S26, S27, S29, S30, S32, S33, S34, S36

vices involved in the communication, such as gateways and messaging servers, are strategically selected to carry the data to the nodes that perform the processing.

Some authors propose to reconfigure the data flow for balancing the load between the edge/fog nodes, or to redirect the data flow to the node with the best conditions (resource availability and lower response latency). For example, S8 proposes a framework that enables the developer to specify dynamic QoS rules. A rule is made up of a source device (e.g. a video camera), a target device (e.g., a web server), a rule activation event (e.g. when a system sensor detects motion), and a QoS requirement that must be guaranteed (e.g. 200ms communication latency between source and target). When the event configured in the rule is triggered, the path of the data flow between the source and the destination is reconfigured to establish the optimal path through a set of switches. This architecture assumes that there are several switches that enable communication between the device layer devices and the cloud layer. However, the edge/fog layer is not included to do edge processing, which could improve system QoS by lowering latency and bandwidth. The system architecture proposed in S8 assumes that the edge/fog layer is composed of devices that only serve the function of relaying the data, but the data processing capacity in the edge devices is ignored. Additionally, it is necessary to consider using the MQTT protocol and broker for communication which offers lower power consumption and low latency due to its very small message header and packet message size (approximately 2 bytes) [171].

Auto-scaling of services and applications

This strategy consists of automatically deploying or terminating replicated services and applications on the system's edge/fog nodes or cloud servers. Auto-scaling is used to ensure stable application performance, and it is one of the most widely used techniques in web applications deployed in the cloud. Auto-scaling is also used in IoT systems but with additional considerations to take into account. For example, when scaling a service on an edge node or fog, it is necessary to strategically select the node that has availability of the necessary computing resources and that offers the greatest communication latency benefits.

In S2, an auto-scaling method is proposed for a distributed intelligent urban surveillance system. The proposed architecture has three layers: video cameras in the device layer, desktops in the edge layer to analyze the video information, and cloud servers that host the web application for the end user. When the video cameras detect an emergency, the frame rates of video capturing increase and image analysis for some objects turn to high-priority tasks. The system then scales the data analysis application by deploying virtual machines to the edge nodes closest to the emergency site. However, deploying the application at the node closest to the device layer device does not always guarantee the best performance. Other factors such as network latency and node specifications should be considered for application allocation decisions. Additionally, the use of virtual machines has limitations given the resource scarcity that characterizes edge nodes. Other virtualization technologies such as containers have advantages for deploying applications to edge/fog layer nodes. In particular, the reduced size of the images and the low startup time are advantages that make containers suitable for IoT systems.

Software deployment and upgrade

The process of deploying and updating software in a semi-automatic way is one of the strategies used to solve problems, correct software issues, improve application performance, and improve system security. However, software updates in distributed IoT systems are also prone to failure during the process.

Containerization is one of the most used technologies that facilitates the semi-automatic deployment of software, given the reduced size of images and the low start time compared to virtual machines. These four studies (S4, S13, S28, and S39) use docker technology to package and run the software versions in containers on Fog nodes. For example, S13 proposes Foggy, a framework for continuous automated deployment in fog nodes. Foggy enables the definition of software containers allocation rules in Fog nodes. Foggy's architecture is based on an orchestration server responsible for monitoring the resources in the nodes and dynamically adapting the software allocation according to the rules defined by the user. However, Foggy's software allocation rules can only be configured according to fixed hardware characteristics of the nodes, i.e., node selection does not depend on dynamic system metrics such as latency, bandwidth consumption, and power consumption.

These QoS factors should also be considered for software allocation decisions in fog nodes. Additionally, Foggy does not monitor the state of the running docker containers to detect and fix failures through actions such as rollback to the previous stable version or redeployment of the software container.

Offloading tasks

The processing tasks executed at the edge/fog nodes can be classified according to their importance and their required response time. While there are system tasks that do not require immediate processing, other tasks such as real-time data analysis are critical to the system and require low response latency. It is necessary to guarantee low latency for these critical tasks, but it is not trivial to achieve this when dynamic events occur in the system such as increased data flow from the device layer. The adaptation strategy *Offloading tasks* addresses this problem in the following way: to guarantee low response latency for critical processing tasks performed by the edge/fog nodes, non-critical tasks are offloaded to the cloud servers to free up capacity in the edge/fog nodes. However, it is necessary to establish when it is really necessary to offload tasks to the cloud servers.

S6 proposes an architecture that coordinates data processing tasks between an edge node and the cloud servers. The edge node performs data processing tasks of the data collected by IoT devices. A monitoring component frequently checks the CPU usage of the edge node, and every time the value exceeds a usage limit (75%) one of the non-critical tasks executed by the node is offloaded to a cloud server. This frees up resources on the fog node for processing tasks that require low latency. However, before moving tasks to cloud servers, the offload tasks between neighboring edge/fog nodes that have the necessary resources available should be considered to take advantage of edge and fog computing. In particular, response latency is lower for tasks that can be executed in the edge/fog layer rather than in the cloud layer. Additionally, decisions to move tasks from one node to another node or to a cloud server could be determined by other factors such as latency, RAM usage, power consumption, and battery level (if the node is battery powered). These factors must be monitored and analyzed to make intelligent offloading decisions according to the QoS requirements of the system.

2.2.5 Open Challenges

The design of IoT systems involves coping with several challenges to ensure a good QoS even when considering the dynamic nature of the IoT environment. Some specific challenges were pointed out by the studies analyzed in this paper. Indeed, the conclusions above suggest already some areas that are not yet fully developed even if some works start to appear that address them.

Nevertheless, we want to highlight additional significant open challenges we believe need to be addressed to improve current adaptation strategies.

We have classified the problems and challenges into four topics that we summarize below. In particular, topic 4 (*Global self-adaptive architecture*) is studied in depth in this thesis.

1. Monitoring and logging the dynamic events themselves

Monitoring the system infrastructure is a key process in the design of a self-adaptive architecture. However, designing a continuous, scalable, resilient, and non-intrusive monitoring system for IoT systems is a challenge. In the literature, efforts are focused on designing strategies to adapt the IoT system at run time. But self-adaptations for system monitoring components also require attention. The monitoring system must self-adapt to the characteristics of the heterogeneity of devices (e.g. gateways, servers, switches, and user devices), heterogeneity according to virtualization (e.g. virtual machines, containers, and pods), and scalability (join and leave of devices). Additionally, it is necessary to effectively monitor and store the data for historical queries and analysis to identify system improvements. Logging of monitoring data implies the design of a domain model that abstracts the main concepts of self-adaptive and multilayer IoT architectures.

2. Software deployment on heterogeneous devices

Some adaptation strategies such as service auto-scaling, software deployment, and upgrades involve the deployment of new software versions in the different layers of the system architecture. Making intelligent allocation decisions is one of the challenging tasks for software deployment at edge/fog nodes. When deploying or moving an application in the system, it is necessary to select the edge/fog nodes that have enough resources to run the application, and to offer the appropriate QoS. Orchestrators like Kubernetes provide functionality through the scheduler component to make allocation decisions based on CPU and RAM required by the container, but other factors such as energy consumption, network latency, reliability, and bandwidth usage should be considered when making allocation decisions.

3. Machine learning for self-adaptable systems

Machine learning systems can automatically identify normal and abnormal patterns and alert a client or third parties when things deviate from observed standards, without requiring prior configuration by human operators. For IoT systems, learning algorithms can also help to prevent disruptive events affecting system availability and QoS. While there are traditional challenges for the design of a learning algorithm such as the selection of the efficient model, the amount of data, and data cleaning, there are also other problems related to the technologies and processes to obtain the data or features. For example, the monitoring of non-functional properties such as accuracy, frequency, sensitivity, and drift is one of the challenges due to the heterogeneity of IoT devices in the device layer.

4. Global self-adaptive architecture

The studies included in this SLR propose techniques and strategies to address at most two of the dynamic events. However, in some scenarios or domains, it is necessary to propose solutions to support various/simultaneous dynamic events. For example, a smart city system synchronizes the basic functions of a city based on seven key components, including natural resources and energy, transport and mobility, buildings, life, government, economy, and people [42]. Due to the large number of IoT devices considered, a smart city system can experience all the dynamic events that we have identified in table 2.5.

Therefore, it is necessary to design a general architecture for IoT systems with components to monitor, detect events, and self-adapt the system: an architecture with the ability to adapt to various dynamic events. For example, a system that can detect failures in software updates and perform operations such as software rollback, while supporting new devices being added to the system. This same system could also support other types of events such as dynamic data transfer rate and network connectivity failures.

For designing this general self-adaptive architecture for IoT systems, some base technologies are especially promising. For example, the MQTT communication protocol is ideal for IoT applications since it presents advantages concerning scalability, asynchronism, decoupling between clients, low bandwidth, and power consumption. Regarding virtualization technology, containerization offers several advantages for software deployment in IoT systems. In particular, it is possible to deploy containers on various types of hardware and operating systems, something very useful considering the heterogeneity of nodes in the edge/fog layer. For example, it is possible to deploy a container with an application on both a RaspberryPI[2] and a Linux server.

[2]https://www.raspberrypi.org

2.2.6 Our direction

This thesis focuses on addressing the challenges and concerns classified in Topic 4 of Section 2.2.5: **design a general architecture/framework to support self-adaptations in IoT systems**. This general framework must support the activities performed at design-time (to specify the system) and at runtime (to self-adapt the system).

The complexity of the multi-layered architectures that are nowadays implemented by IoT systems challenges their design and more so for systems that require self-adaptation schemes. To facilitate the definition of these complex systems and provide a better understanding of these, models are commonly used. Models raise the level of abstraction to focus on the relevant concepts of a domain, in this case, self-adaptive multi-layered IoT systems domain. However, to build models, a domain-specific language (DSL) is required to specify system concepts such as nodes, sensors, actuators, applications, and adaptation rules.

Designing a DSL involves an in-depth study to understand the domain to be modeled. In this thesis, we design a DSL to describe multi-layered IoT systems and their adaptation scheme to cope with several types of dynamic events by performing adaptations. This DSL supports the design time phase of our framework.

To manage the IoT system at runtime, it is necessary to build a solution capable of self-adapting the IoT system according to the adaptation scheme specified at design time. To achieve this, it is required to design an architectural approach with capabilities to monitor, detect dynamic events, and apply changes on the target IoT system. This implies at least (1) the deployment of monitors that continuously collect infrastructure and QoS metrics such as those in Table 2.6; (2) an analyzer component that checks and detects abnormal values in the metrics (dynamic events), and (3) an adaptation engine to perform changes or adaptations to the system. The design of the framework to manage the system at runtime is another research activity that we address in this thesis.

In this thesis, we address actions and adaptations patterns grouped in two categories: (1) **architectural adaptations** (such as those identified in Table 2.7) to guarantee system availability and performance despite dynamic events; and (2) **system actuators control** to meet system functional requirements involving system actuator management (e.g., activating/deactivating alarms, turning on/off lamps, and increasing the power of a fan). In Chapter 4, we discussed the specification of rules involving these two types of actions or adaptations.

2.3 Conclusion

As the first research activity of this thesis, we have conducted an SLR to study the dynamic events that impact the QoS of IoT systems, to analyze the strategies implemented by the literature in order to address them, and to identify the weaknesses of the approaches found in the state-of-the-art.

We identified six types of dynamic events or unexpected changes and four adaptation strategies in response to the events. Monitoring the resource consumption of the edge/fog nodes is one of the most used strategies to detect some dynamic events of the system. In particular, the consumption of CPU and RAM memory are metrics frequently monitored to identify when a node fails or is close to failure.

We have identified open challenges that we believe need to be addressed to improve current adaptation strategies. These challenges are classified into four topics: (1) *monitoring and logging the dynamic events themselves*, (2) *software deployment on heterogeneous devices*, (3) *machine learning for self-adaptable systems*, and (4) *global self-adaptive architecture*. In this thesis, we focus on the challenges of topic 4 to support the design and management of self-adaptive multi-layer IoT architectures. The design of a DSL for the specification of these systems at desig-time, and the design of a framework to support the system at runtime are some of the tasks we conducted to address these challenges.

Chapter 3

Overview

To meet increasingly restrictive requirements and improve QoS, Internet of Things (IoT) systems have embraced multi-layered architectures leveraging edge and fog computing. However, the dynamic and evolution of IoT environment can impact QoS due to unexpected events. Therefore, proactive evolution and adaptation of the IoT system becomes a necessity and concern that we address according to our research objectives. We propose IoT@runtime, an approach for specifying and managing self-adaptive IoT systems. This approach supports both design time (for specification) and runtime (for adaptation and reconfiguration) activities.

This chapter introduces our proposal. Section 3.1 presents an overview of our architecture by distinguishing design time and runtime activities. Section 3.2 details a running example used in several chapters of this document to better illustrate our approach. Our research methodology is presented in Section 3.3. Finally, Section 3.4 concludes this chapter.

3.1 Framework Overview

IoT@runtime is a comprehensive approach for modeling and managing self-adaptive, multi-layer IoT systems. This approach involves multiple technologies, techniques, components and software tools in two stages: design time for the specification of the self-adaptive IoT system, and runtime to support the operation and adaptation of the system. Figure 3.1 summarizes an operational view of our architecture by distinguishing design time (left-hand side) and runtime (right-hand side).

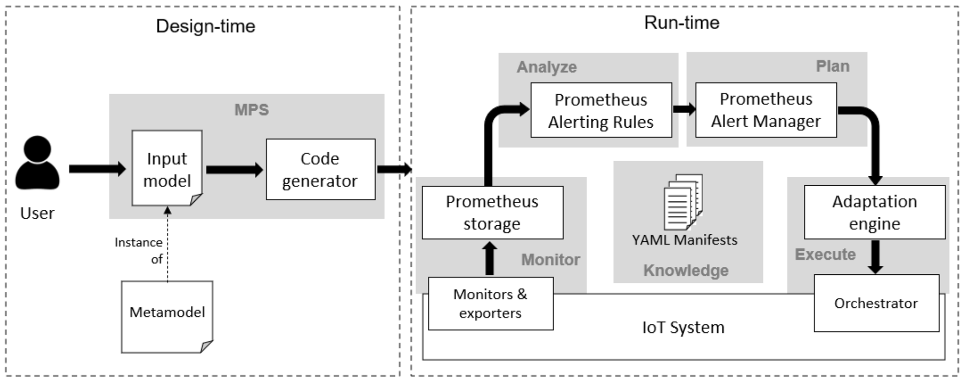


Figure 3.1: Overview

3.1.1 Design time stage

The first step in designing and managing self-adaptive IoT systems is the specification of the system, its environment, its adaptation plan and other domain properties. To address this task, we have designed a DSL for modeling multi-layer IoT architectures (including devices and nodes of the physical, edge/fog and cloud layers), container-based applications deployed on the nodes, and the adaptation rules to ensure system operation.

As shown in the Figure 3.1, the user builds a model (using our DSL) that describes the multi-layered IoT system and its self-adaptive behaviour, based on a defined metamodel. The code generator then transforms the model into text, producing the code for the deployment of the IoT applications and the code required to support the execution of the system at runtime (including the code for infrastructure monitoring and system management tools). Both the DSL and the code generator are implemented using MPS [1], a language workbench developed by JetBrains to design DSLs. We have used MPS because of the ability to configure multiple notations (such as textual, graphical, or tabular) on a single model, and because of the variety of integrated components and libraries that enable the design of the abstract syntax, projectional editors, constraints, and the code generator (transformation chain) in an integrated way. Other toolkits (such as GMF[2], Sirius[3], or Xtext[4]) focus on a single notation, and would require additional effort for integration. The DSL and the code generator are detailed in Chapter 4.

[1]https://www.jetbrains.com/mps/
[2]https://www.eclipse.org/modeling/gmp/
[3]https://www.eclipse.org/sirius/overview.html
[4]https://www.eclipse.org/Xtext/

3.1.2 Runtime stage

In the runtime stage, the operation and self-adaptation of the IoT system is performed. To achieve this, we have designed the architecture based on the MAPE-K loop, which has been widely employed for the design of self-adaptive systems. The four stages of the MAPE-K loop enable the **Monitoring** or collection of information on the current state of the system, the **Analysis** of the collected information, the **Planning** of the list of actions or adaptations to be performed on the system, and the **Execution** of the adaptation plan.

Our approach leverages different technologies and tools for each stage of the MAPE-K loop. For example, we use Prometheus and several of its modules (such as Alerting Rules and Alert Manager) to store the information, analyze it, and detect when an adaptation should be applied to the system. We have also built the Adaptation Engine to executes the actions and adaptations on the IoT system through the orchestrator. This runtime approach is detailed in Chapter 5, and the code for the deployment and configuration of the tools and technologies is produced by the code generator (see Section 4.4 for details).

3.2 Running Example: Smart Building Scenario

Smart buildings seek to optimize different features such as ventilation, heating, lighting, energy efficiency, etc [98]. Optimizing these features requires making quick, real-time decisions about security, temperature settings, emergency response, and other types of critical system tasks. This involves the collection and generation of terabytes of data per day, increasing bandwidth consumption and becoming unmanageable by the cloud alone. In a single emergency scenario several automatic tasks could be executed threatening the availability of the system due to the increased consumption of bandwidth and computing resources. For example, when a fire is detected inside the building, sensors increase their monitoring frequency, fire alarms are turned on, video surveillance systems analyze the spread of the fire, the shortest available evacuation routes are calculated for each zone and people are oriented by signage, power is cut on circuits that can worsen the emergency, the fire department is notified immediately, and a number of other automatic tasks can be executed to protect and assist occupants.

3.2.1 Multilayered Architecture

Aiming to meet functional and non-functional requirements, smart buildings systems implement multi-layered architectures that leverage edge/fog computing. While critical tasks that require real-time data analysis are executed on edge/fog nodes (e.g. real time detection of emergencies), other tasks that do not demand immediate response are executed in the cloud (e.g. generation of historical data reports). We will use a simple Smart Building scenario as a running example to better illustrate our approach. Other case studies modeling real-world Underground Mining and Wastewater Treatment Plants (WWTPs) are presented in Chapter 6. We prefer to introduce our approach modeling a Smart Building scenario because it has been well-studied in the literature and may result more suitable to ease understanding.

Adopting the concept of smart building, a hotel company (*Hotel Beach*) wants to reduce fire risks by automating disaster management in its hotels. A fire alarm and monitoring system are implemented in each of the company's hotels. We will assume that all buildings (hotels) have three floors with two rooms each. Fig. 3.2 presents an overview of the 1st floor of this building. According to this, the infrastructure (device, edge/fog, and cloud layers) of the company hotel IoT system are as follows.

- **Device layer**. Each room has a temperature sensor, a carbon monoxide (CO) gas sensor, and a fire water valve. Furthermore, an alarm is deployed on the lobby. Each sensor has a threshold measurement to activate the corresponding alarm, e.g. a person should not be continuously exposed to CO gas level of 50 parts per million (ppm) for more than 8 hours, and 400 ppm for more than 4 minutes.

- **Edge layer**. In each room, an edge node receives the information collected by the sensors of the device layer and run a software container (C1 and C2) for analyzing sensor data in real time to check for the presence of smoke and generate an alarm state that activates the actuators. A fog node (linked to the edge nodes) is located in the 1st floor of the building. This node runs the C3 container (running App2, a machine learning model to predict fires on any of the building's floors), and C4 (running App3, in charge of receiving and distributing data, typically a MQTT broken as we will see later on).

- **Cloud layer**. The cloud layer has a server or cloud node that runs the C5 container, a web application (App4) to display historical information of sensor data and of fire incidents in any of the hotels property of the company.

Table 3.1 summarizes the applications features and the containers deployed in the IoT system infrastructure. Ports, memory, and CPU values are approximate to real applications. In Chapter 4 we introduce the modeling of the architecture of this system, including the applications, containers, and adaptation.

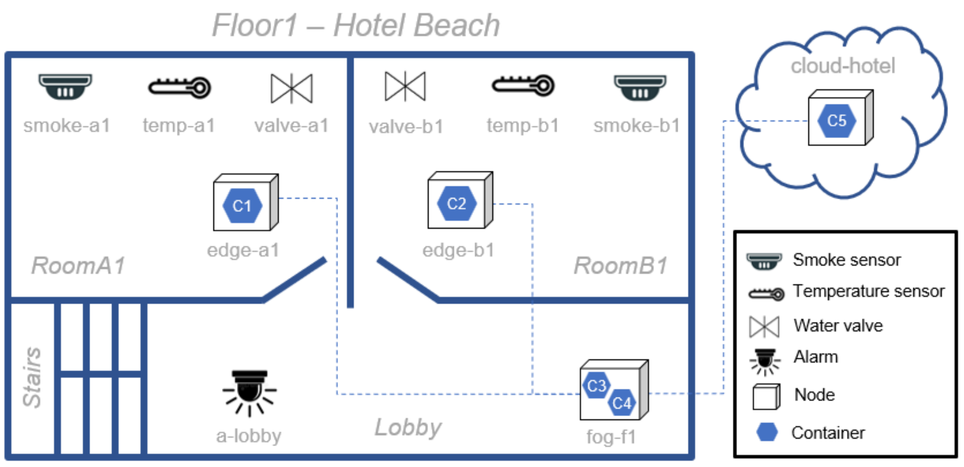


Figure 3.2: Overview of the smart building IoT system, first floor

3.2.2 System Adaptation

Although multi-layer architectures improve latency, bandwidth consumption and reliability, it is still necessary to guarantee the availability of applications that perform critical system tasks. In any IoT system, there are critical applications that should be available all the time. For example, the availability of the containers running App1 (real-time smoke monitoring to detect and alarm the presence of smoke) should be guaranteed. However, some environmental factors can generate unexpected events impacting system operation and services availability. For example, when an emergency state is detected, the increase in data collected at the device layer may increase bandwidth consumption leading to system failures.

In these cases, the IoT system must self-adapt to guarantee its operation. For instance, if the *edge-a1* node fails, it will be necessary to migrate the *C1* container to another suitable node. These types of system architectural adaptations are addressed in our research, but also actions at the device layer to fulfill functional requirements. For this running example, if a gas sensor detects CO gas greater than 400 ppm, an alarm should be triggered. Chapter 4 describes the modeling of system rules to specify these scenarios.

3.3 Research Methodology

To develop and evaluate the artifacts of our proposal, we have followed the guidelines of the Design Science Research (DSR) methodology, which has been adopted in

Table 3.1: Description of containerized applications, first floor

Application	Requirements	Containers
App1 (real-time smoke monitoring)	Memory: 500 MB CPU: 500 mCores Port: 8000 Repo: hotel/app1:latest	C1, C2
App2 (fire predictive model)	Mem: 900 MB CPU: 900 mCores Port: 5000 Repo: hotel/app2:latest	C3
App3 (MQTT broker)	Memory: 700 MB CPU: 700 mCores Port: 1883 Repo: mosquitto:2.0	C4
App4 (web application)	Memory: 2000 MB CPU: 2000 mCores Port: 8080 Repo: hotel/app4:latest	C5

information systems and computer science research areas for the creation and evaluation of IT artifacts [125]. DSR involves the construction of artifacts as decision support systems, modeling tools, governance strategies, and methods for system evaluation. Two high-level stages are performed in the DSR methodology: build (construct an artifact for a specific purpose) and evaluate (determine how well the artifact behaves) [106]. Peffers et al. [125] synthesizes this methodology into 6 activities as shown in figure 3.3.

We addressed the first research activities (problem identification and definition of research objectives) by conducting an SLR. Through this SLR (published in the Journal of Internet Services and Applications [6]) we have identified the open challenges covered in this thesis. Then, we built the artifacts that make up our proposed solution, defined a suitable context for its illustration, performed one or several evaluations, and finally sought a communication method such as conferences, journals, or workshops.

Developing our approach solution involves the design and implementation of several software artifacts such as metamodels, a code-generator, monitors, services, and the remaining artifacts of the architecture presented in Figure 3.1. The development and evaluation of the artifacts have been classified in two groups: artifacts

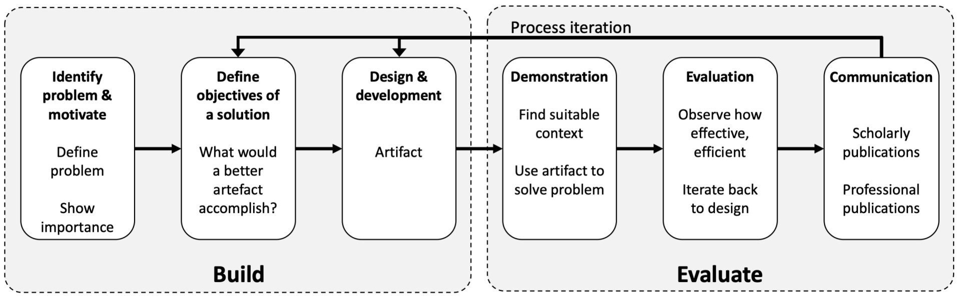


Figure 3.3: Design Science Research Methodology (DSR) process model [125]

for design time and artifacts for runtime.

Desing time: we first address the design of the DSL to model self-adaptive multilayer IoT systems, and the code generator to obtain YAML manifest for application deployment. In a second iteration, we developed two extensions of the DSL for modeling IoT systems in two different domains; underground mining, and Wastewater Treatment Plants (WWTPs). The usability of one of these DSL extensions was validated through empirical experiments with a group of participants.

Runtime: To support the system at runtime, we have designed or configured the artifacts that make up each of the stages of the MAPE-K loop. We also modified the code generator to obtain the manifests that deploys and configures these artifacts. To evaluate the effectiveness of the tool, we have simulated scenarios to test the different self-adaptations of the system.

3.4 Conclusion

In this chapter we presented an overview of our proposal, a comprehensive approach for modeling and managing self-adaptive, multi-layer IoT systems. This approach involves multiple technologies, techniques, components and software tools in two stages: design time for the specification of the multi-layered IoT architecture and its adaptive behaviour, and runtime to support the operation and adaptation of the system.

A running example of a smart building system is also presented in this chapter. This example is to better illustrate our approach in the following chapters. Finally, we presented the main steps of the Design Science research methodology adopted to develop and evaluate the software artifacts of this thesis.

Chapter 4

Modeling Self-adaptive IoT Architectures

Modeling IoT architectures is a complex process that must cover as well the specification of self-adaptation rules to ensure the optimized execution of the IoT system. To facilitate this task, we propose a new IoT Domain-Specific Language (DSL) covering both the static and dynamic aspects of an IoT deployment. Our DSL is focused on three main contributions: (1) modeling primitives covering multi-layered IoT architectures, including IoT devices (sensors and actuators), edge, fog, and cloud nodes; (2) modeling the deployment and grouping of container-based applications on those nodes; and (3) a specific sublanguage to express rules.

In this chapter, we address the design of the components involved in the design time phase of our approach illustrated in Figure 3.1 (i.e., the DSL and the code generator). Our DSL for modeling the static and dynamic aspects of the IoT system is introduced as follows. First, Section 4.1 describes the abstract syntax and the concrete syntax of the DSL elements for the specification of static aspects including architecture and deployment of containerized applications. Then, Section 4.2 covers the dynamic ones, i.e. the specification of rules. Section 4.3 describes the DSL implementation and Section 4.4 presents the code generator developed to produce YAML manifests and configuration files. Finally, Section 4.5 presents an installation and configuration guide for using the DSL, and Section 4.6 concludes this chapter. To illustrate the concepts of the metamodel, we will use the running example presented in Section 3.2.

4.1 Modeling of the IoT Architecture

Multi-layered architectures have emerged to increase the flexibility of pure cloud deployments and help meet non-functional IoT system requirements. In particular, edge computing and fog computing are solutions that propose to bring computation, storage, communication, control, and decision making closer to IoT devices.

Edge and fog computing often leverage containerization as a virtualization technology. Containers, compared to virtual machines, are lightweight, simple to deploy, support multiple architectures, have a short start-up time, and are suitable for dealing with the heterogeneity of edge and fog nodes. To support high scalability, message-driven asynchronous architectures are commonly implemented in cyber-physical and IoT systems [73]. The publisher/subscriber pattern and the Message Queing Telemetry Transport (MQTT) protocol are becoming the standard for M2M communications [111], where messages are sent (by publishers) to a message broker server and routed to destination clients (subscribers).

This DSL enables the specification of all these concepts as part of a multi-layered IoT architecture.

4.1.1 Abstract syntax

The abstract syntax of the DSL is commonly defined through a metamodel that represents the domain concepts and their relationships. Fig. 4.1 shows the metamodel that abstracts the concepts to define multi-layered IoT architectures. The sensors and actuators of the device layer are modeled using the *Sensor* and *Actuator* concepts that inherit from the *IoTDevice* concept. This generalization is restricted to be complete and disjoint. All *IoTDevice*s have a connectivity type (such as ethernet, wifi, ZigBee, or another) represented through the *Connectivity* attribute.

We also cover the concepts for specifying publish/subscribe communication between IoT devices and nodes. *IoTDevices* are publishers or subscribers to a topic specified by the relationship to the *Topic*) concept. The gateway of an *IoTDevice* can be modeled through the *gateway* relationship with the *EdgeNode* concept. Via this gateway, the sensor can communicate with other nodes, e.g. the MQTT broker node. Note the multiplicity of *gateway* relationship (0..1), this means that the sensor/actuator may not connect to a gateway. Sometimes IoT devices may have an integrated communication module that enable sending data directly to the cloud without a gateway.

Monitoring systems commonly generate alarms when one of the sensors detects a value that exceeds a limit. For example, gas monitoring systems set emergency thresholds for each of the gases checked. In this metamodel, the threshold value and unit of the monitored variable by a sensor can be represented through the attributes *threshold* and *unit*.

The location of IoTDevices can be specified through geographic coordinates (*latitude* and *longitude* attributes). Both *Sensors* and *Actuators* have a type represented by the concepts *SensorType* and *ActuatorType*. For instance, following the running example (Fig. 3.2), there are temperature and smoke type sensors, and there are valve and alarm type actuators.

Physical (or even virtual) spaces such as rooms, stairs, buildings, or tunnels can be represented by the concept *Region*. A *Region* can contain subregions (relationship *subregion*s in the metamodel). For example, region *Floor1* (Fig. 3.2) contains subregions *Room1*, *Room2*, *Lobby*, and *Stairs*. *IoTDevice*s, *EdgeNode*s, and *FogNode*s are deployed and are located in a region or subregion (represented by *region* relationships in the metamodel). Back to the running example, the *edge-a1* node is located in the *RoomA1* region of *Floor1* of the *Hotel Beach*, while the *fog-f1* node is located in the *Lobby* region of *Floor1*.

Edge, fog and cloud nodes are all instances of *Node*, one of the key concepts of the metamodel. A node has the ability to host the software containers. Communication between nodes can be specified via the *linkedNodes* relationship as we may want to indicate what nodes on a certain layer could act as reference nodes in another layer (e.g. what cloud node should be the first option for a fog node). Nodes can also be grouped in clusters that work together. A *Cluster* has at least one master node (represented by the *master* relationship) and one or several worker nodes (represented by the *workers* relationship). The details of each node are expressed via attributes such as ip address (ipAddress), operating system (OS), number of cores in the processor (cpuCores), RAM memory (memory), storage capacity (storage), and processor type (processor enum).

A *Node* can host several software containers according to its capabilities and resources (primarily *cpuCores*, *memory*, and *storage*). The cpu and memory usage of a container can be restricted through *cpuLimit* and *memoryLimit* attributes. Each software container runs an application (represented by the concept *Application*) that has a minimum of required resources specified by the attributes *cpuRequired* and *memmoryRequired*. The repository of the application image is specified through the *imageRepo* attribute, and the used ports through *port* and *k3sPort* attributes. The container volumes and their paths (a mechanism for persisting data used and generated by containers) are represented by *Volume* concept. Finally, the MQTT broker that receives and distributes the messages can also be specified and deployed in a software container, and its broker topics are represented by the *topics* relationship.

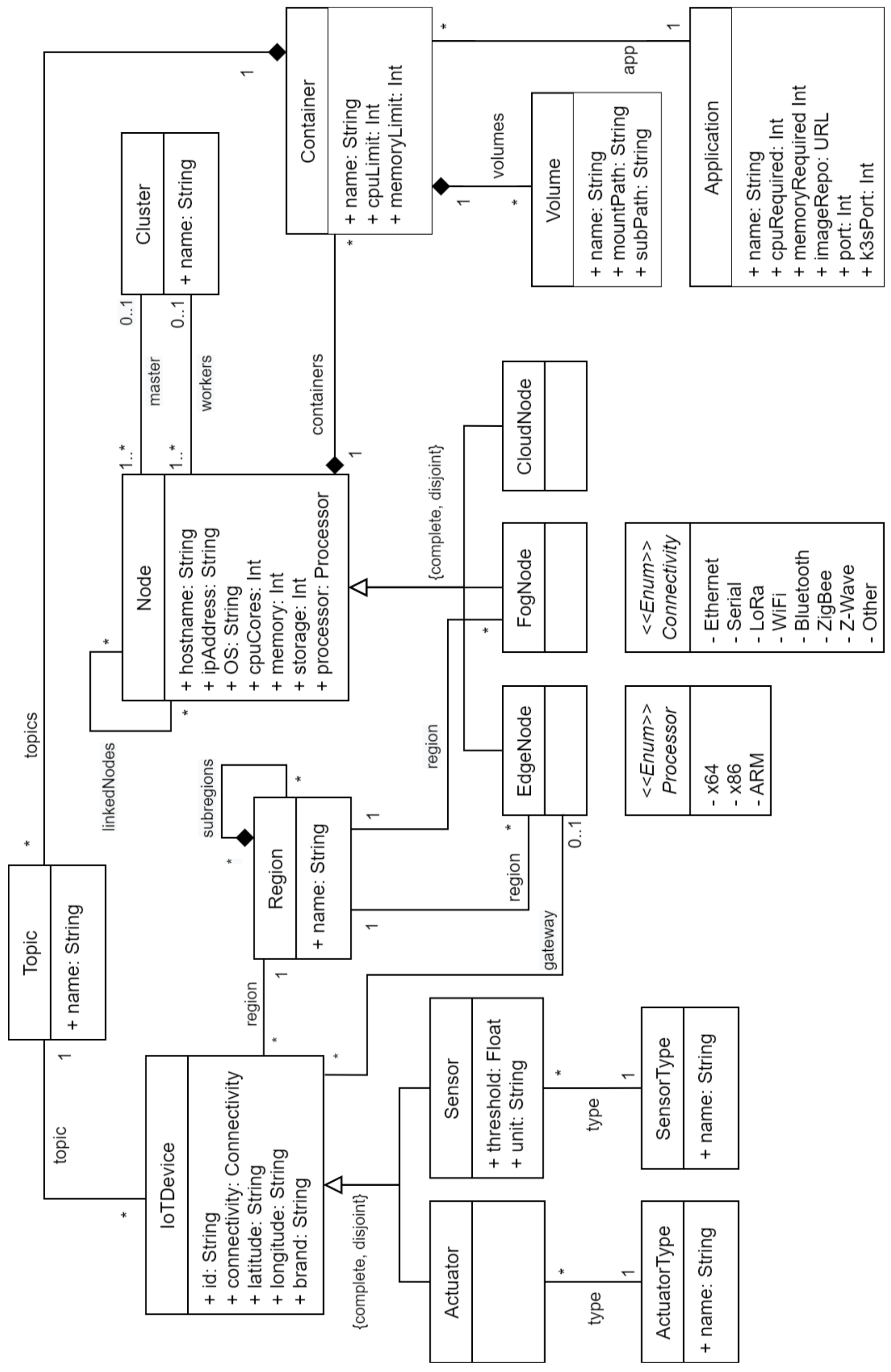


Figure 4.1: Metamodel depicting the multi-layer architecture

4.1.2 Concrete syntax

The concrete syntax refers to the type of notation (such as textual, or graphical) to represent the concepts of the metamodel. Graphical DSLs involve the development of models using graphic items such as blocks, arrows, axes, and so on. Textual DSLs involve modeling using configuration files and text notes. Though most DSLs employ a single type of notation they could benefit from offering several alternative notations, each one targeting a different type of DSL user profile. This is the approach we follow here, leveraging the benefits of using a projectional editor.

Projectional editors enable the user's editing actions to directly change the Abstract Syntax Tree (AST) without using a parser [23]. That is, while the editing experience simulates that of classical parsing-based editors, there is a single representation of the model stored as an AST and rendered in a variety of perspectives thanks to the corresponding projectional editors that can deal with mixed-language code and support various notations such as tables, mathematical formulas, or diagrams.

Indeed, we take advantage of MPS projectional editors to define a set of complementary notations for the metamodel concepts. We blend textual, tabular, and tree view, depending on the element to be modeled. We next employ these notations to model our running example. More technical details about MPS and the implementation of our concrete notations are presented in Section 4.3.

4.1.3 Well-Formedness Rules

Some metamodel constraints cannot be defined using only elements of the graphical metamodel syntax [26]. An alternative to address this is to use the Object Constraint Language (OCL), a declarative language for describing metamodel rules that are validated at the model level (known as well-formedness rules). The definition and implementation of these rules improves the accuracy of the DSL and avoids errors that could occur at runtime.

Figure 4.2 shows the well-formedness rules that we have defined using the OCL language. The *Node* concept has four rules: one (WFR1) to guarantee that the *hostname* is unique and the other three (WFR2, WFR3 and WFR4) to guarantee that the attributes *cpuCores*, *memory*, and *storage* take positive values. We have also defined a rule (WFR5) to ensure that the *IoTDevices id* is unique. Similarly, the *Container*, *Region*, and *Cluster* concepts have a rule (WFR6) for the *name* attribute to be unique. Additionally, we have defined a rule (WFR7) for the *Container* concept that guarantees that the owner *Node* has enough available resources (memory and CPU) to host it. These rules have been expressed as invariants (*inv*), i.e., conditions that must always be true for all instances of the class defined in the *context*.

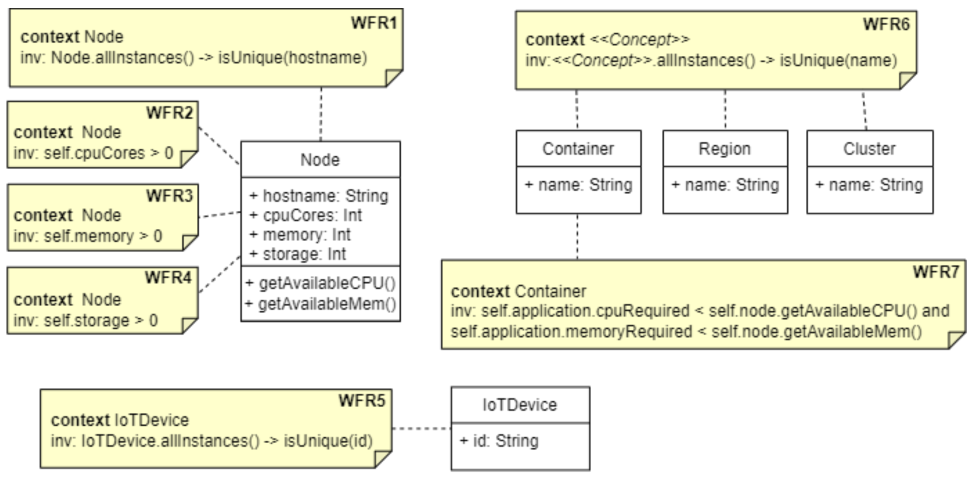


Figure 4.2: Well-formedness rules for multi-layer architecture metamodel

4.1.4 Example scenario

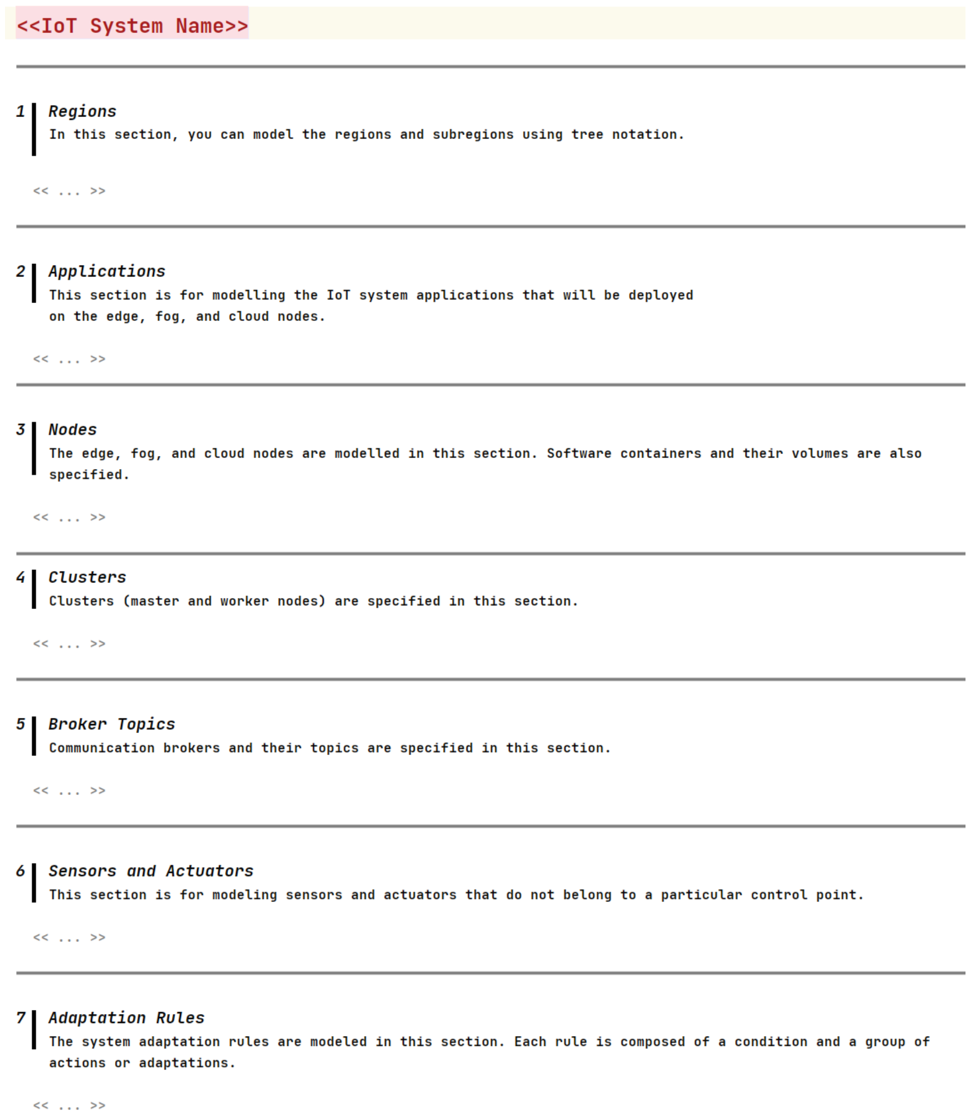
We present next how to model the IoT architecture of the running example (from Section 3.2) using our DSL. When the user creates a new model, a template with the concepts for specifying the IoT system is provided (see Figure 4.3). The definition of concepts in the model template follow a logical order to describe and specify the system architecture. For example, the regions defined in section 1 of the template are required to specify the location of the nodes and IoT devices in sections 3 and 6.

Regions

Figure 4.4 shows the specification of the *Hotel Beach* regions, in particular those on *Floor 1*, i.e. four subregions: two *Rooms*, the *Lobby* and the *Stairs*. The regions defined in this tree diagram are then referenced in the specification of the nodes, sensors, actuators and rules of the system.

Applications

Fig. 4.5 depicts the modeling of the IoT system applications, including its technical requirements and repository address. The memory and cpu requirements are primarily used to determine if the nodes that will host the application container have the necessary resources. The port specifications are to configure the container ports, and the repository to download the image of the containerized application.

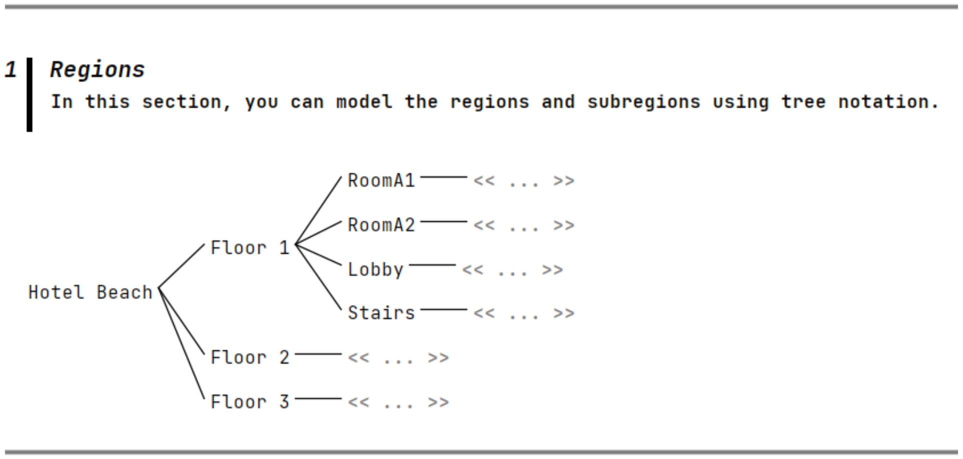


```
<<IoT System Name>>

1 Regions
  In this section, you can model the regions and subregions using tree notation.

  << ... >>

2 Applications
  This section is for modelling the IoT system applications that will be deployed
  on the edge, fog, and cloud nodes.

  << ... >>

3 Nodes
  The edge, fog, and cloud nodes are modelled in this section. Software containers and their volumes are also
  specified.

  << ... >>

4 Clusters
  Clusters (master and worker nodes) are specified in this section.

  << ... >>

5 Broker Topics
  Communication brokers and their topics are specified in this section.

  << ... >>

6 Sensors and Actuators
  This section is for modeling sensors and actuators that do not belong to a particular control point.

  << ... >>

7 Adaptation Rules
  The system adaptation rules are modeled in this section. Each rule is composed of a condition and a group of
  actions or adaptations.

  << ... >>
```

Figure 4.3: Model template

Nodes

For describing the system nodes, we propose a tabular notation. Figure 4.6 shows the specification of the nodes deployed in Floor 1 of the Hotel. The node description includes the layer it belongs to (edge, fog, or cloud), the hardware properties (such as



1 **Regions**

In this section, you can model the regions and subregions using tree notation.

Figure 4.4: *Regions* modeling example

memory and storage resources), the regions where it is located, and the application containers it hosts. Note that C4 is the only container that uses a volume for the MQTT broker configuration parameters (mosquitto[1] in this example).

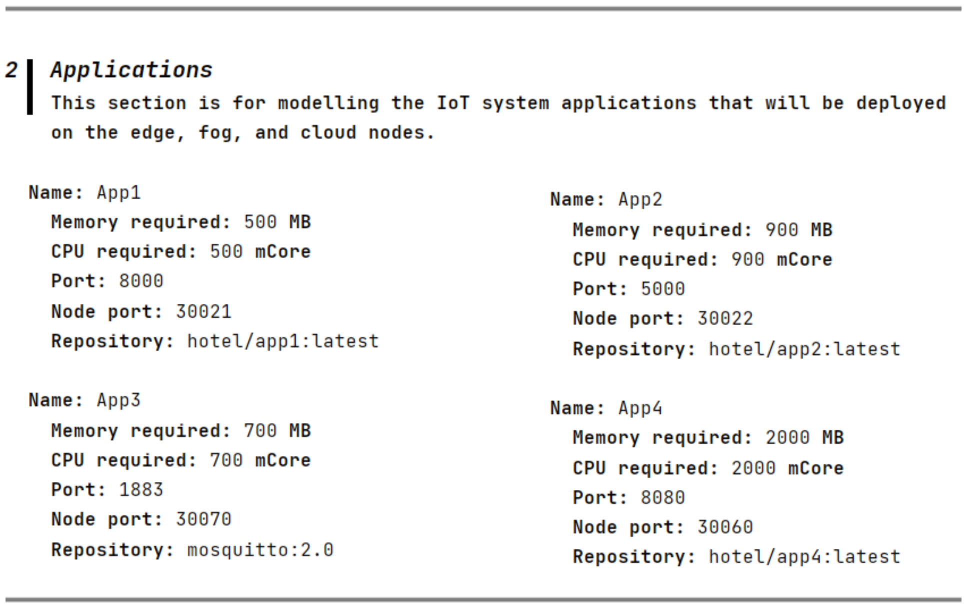
Clusters

To specify a cluster of nodes, at least one master node and one worker node are required. An example of cluster modeling is shown in Figure 4.7, in which the cluster composed of the Hotel nodes is modeled. Although there is one or more master nodes managing the cluster (usually cloud nodes), constant Internet connection is not a mandatory for multi-layered architectures. The edge/fog nodes can operate as a standalone network node with limited internet connectivity.

Broker topics

To specify the MQTT topics, the container running the broker must be selected. Figure 4.8 shows the topics defined for the sensors and actuators deployed on *Floor 1* of the Hotel, whose broker is running on the *C4* container. The topics in this example follow the nomenclature floor/room/sensor_type, however, these could be specified following a more complex nomenclature according to the case.

[1]https://mosquitto.org/



```
2 | Applications
  | This section is for modelling the IoT system applications that will be deployed
    on the edge, fog, and cloud nodes.

  Name: App1
    Memory required: 500 MB
    CPU required: 500 mCore
    Port: 8000
    Node port: 30021
    Repository: hotel/app1:latest

  Name: App2
    Memory required: 900 MB
    CPU required: 900 mCore
    Port: 5000
    Node port: 30022
    Repository: hotel/app2:latest

  Name: App3
    Memory required: 700 MB
    CPU required: 700 mCore
    Port: 1883
    Node port: 30070
    Repository: mosquitto:2.0

  Name: App4
    Memory required: 2000 MB
    CPU required: 2000 mCore
    Port: 8080
    Node port: 30060
    Repository: hotel/app4:latest
```

Figure 4.5: *Application* modeling example

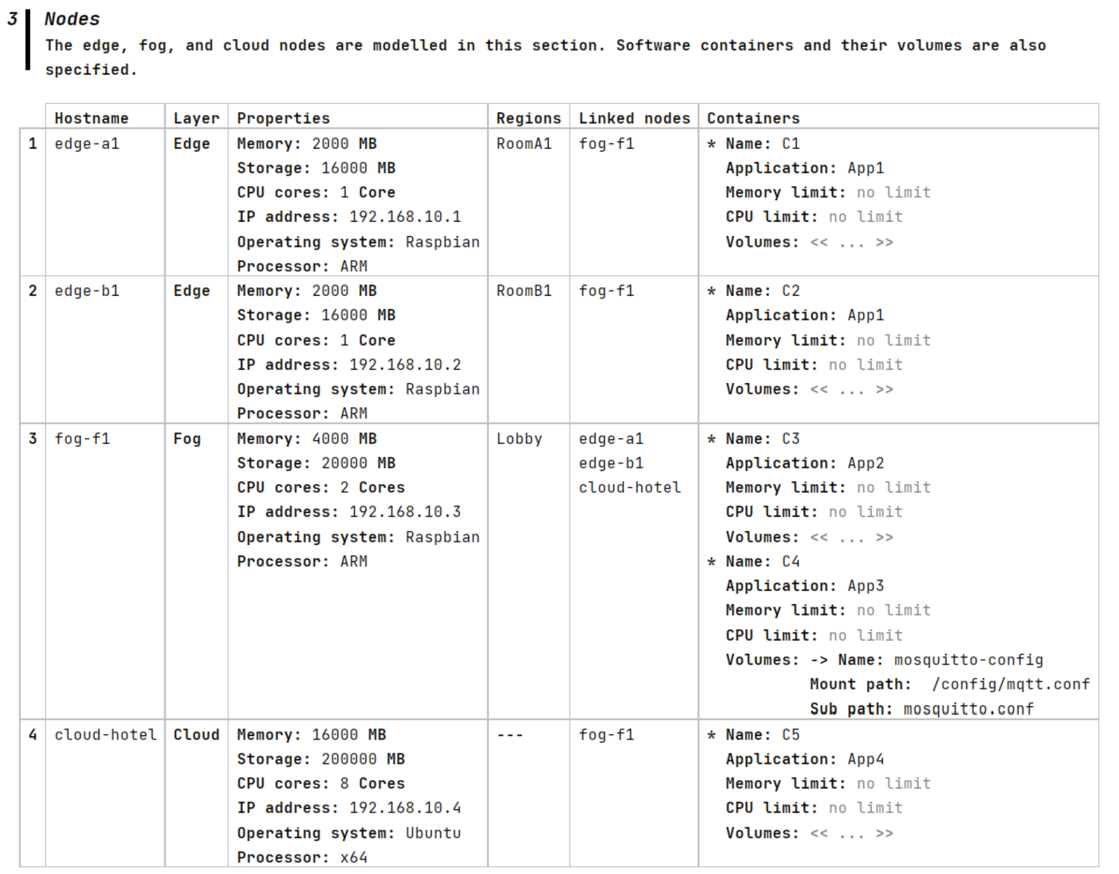
Sensors and Actuators

IoT devices can be modeled using a tabular notation similar to Nodes. Fig. 4.9 shows the list of sensors and actuators located in the *Floor 1* region, including their descriptions such as units and threshold (for sensors only), region where it is deployed, brand, communication protocol, gateway (if any), topic (either publisher or subscriber), and location coordinates.

4.2 Modeling of Rules

The dynamic environment of an IoT system requires dealing with expected an unexpected events. The former may trigger actions to comply with the standard behaviour of the system (e.g. to turn on an alarm upon detection of fire), unexpected ones may require a self-adaptation of the system itself to continue its normal operation. This section presents a rule-based language that can cover both types of events (and even mix them in a single rule). This facilitates an homogeneous of all the dynamic aspects of an IoT system.

To decide what unexpected environmental situation should we include and what the standard patterns of response are common in the self-adaptation of IoT systems,



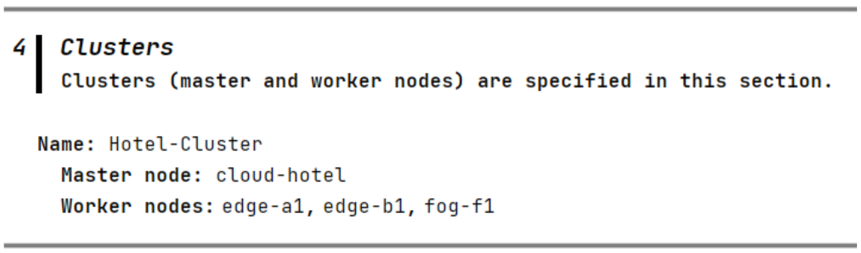
3 **Nodes**

The edge, fog, and cloud nodes are modelled in this section. Software containers and their volumes are also specified.

	Hostname	Layer	Properties	Regions	Linked nodes	Containers
1	edge-a1	Edge	Memory: 2000 MB Storage: 16000 MB CPU cores: 1 Core IP address: 192.168.10.1 Operating system: Raspbian Processor: ARM	RoomA1	fog-f1	* Name: C1 Application: App1 Memory limit: no limit CPU limit: no limit Volumes: << ... >>
2	edge-b1	Edge	Memory: 2000 MB Storage: 16000 MB CPU cores: 1 Core IP address: 192.168.10.2 Operating system: Raspbian Processor: ARM	RoomB1	fog-f1	* Name: C2 Application: App1 Memory limit: no limit CPU limit: no limit Volumes: << ... >>
3	fog-f1	Fog	Memory: 4000 MB Storage: 20000 MB CPU cores: 2 Cores IP address: 192.168.10.3 Operating system: Raspbian Processor: ARM	Lobby	edge-a1 edge-b1 cloud-hotel	* Name: C3 Application: App2 Memory limit: no limit CPU limit: no limit Volumes: << ... >> * Name: C4 Application: App3 Memory limit: no limit CPU limit: no limit Volumes: -> Name: mosquitto-config Mount path: /config/mqtt.conf Sub path: mosquitto.conf
4	cloud-hotel	Cloud	Memory: 16000 MB Storage: 200000 MB CPU cores: 8 Cores IP address: 192.168.10.4 Operating system: Ubuntu Processor: x64	---	fog-f1	* Name: C5 Application: App4 Memory limit: no limit CPU limit: no limit Volumes: << ... >>

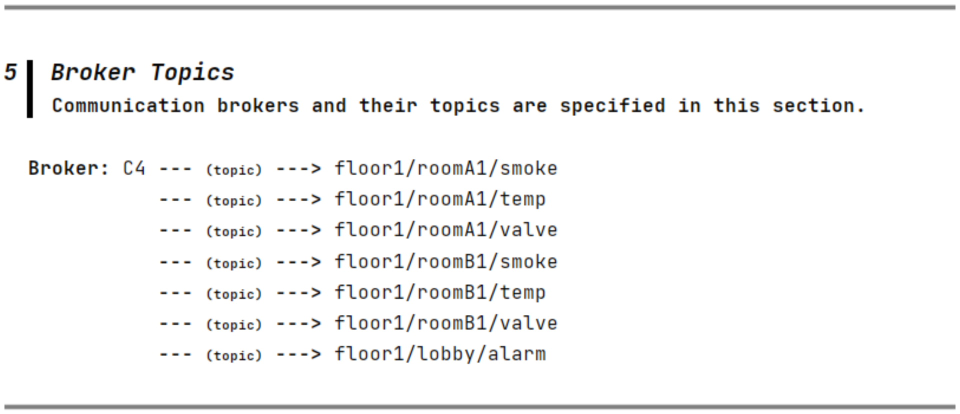
Figure 4.6: Nodes modeling example

we rely on our previous systematic literature review presented in Chapter 2.2. For instance, the three architectural adaptations (offloading, scaling, and redeployment) addressed in this study were identified in the SLR. Our language covers all of them and even enables complex rules where policies involving several strategies can be attempted in a given order.



```
4 | Clusters
  | Clusters (master and worker nodes) are specified in this section.

Name: Hotel-Cluster
  Master node: cloud-hotel
  Worker nodes: edge-a1, edge-b1, fog-f1
```

Figure 4.7: Cluster modeling example

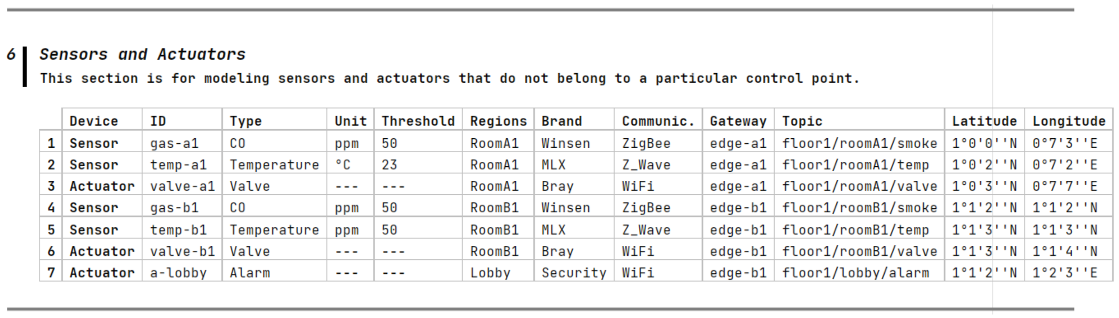


```
5 | Broker Topics
  | Communication brokers and their topics are specified in this section.

Broker: C4 --- (topic) ---> floor1/roomA1/smoke
           --- (topic) ---> floor1/roomA1/temp
           --- (topic) ---> floor1/roomA1/valve
           --- (topic) ---> floor1/roomB1/smoke
           --- (topic) ---> floor1/roomB1/temp
           --- (topic) ---> floor1/roomB1/valve
           --- (topic) ---> floor1/lobby/alarm
```

Figure 4.8: Broker topics modeling example

4.2.1 Abstract syntax

The metamodel representing the abstract syntax for defining the rules is presented in Fig. 4.10.



6 ***Sensors and Actuators***

This section is for modeling sensors and actuators that do not belong to a particular control point.

	Device	ID	Type	Unit	Threshold	Regions	Brand	Communic.	Gateway	Topic	Latitude	Longitude
1	**Sensor**	gas-a1	CO	ppm	50	RoomA1	Winsen	ZigBee	edge-a1	floor1/roomA1/smoke	1°0'0''N	0°7'3''E
2	**Sensor**	temp-a1	Temperature	°C	23	RoomA1	MLX	Z_Wave	edge-a1	floor1/roomA1/temp	1°0'2''N	0°7'2''E
3	**Actuator**	valve-a1	Valve	---	---	RoomA1	Bray	WiFi	edge-a1	floor1/roomA1/valve	1°0'3''N	0°7'7''E
4	**Sensor**	gas-b1	CO	ppm	50	RoomB1	Winsen	ZigBee	edge-b1	floor1/roomB1/smoke	1°1'2''N	1°1'2''N
5	**Sensor**	temp-b1	Temperature	ppm	50	RoomB1	MLX	Z_Wave	edge-b1	floor1/roomB1/temp	1°1'3''N	1°1'3''N
6	**Actuator**	valve-b1	Valve	---	---	RoomB1	Bray	WiFi	edge-b1	floor1/roomB1/valve	1°1'3''N	1°1'4''N
7	**Actuator**	a-lobby	Alarm	---	---	Lobby	Security	WiFi	edge-b1	floor1/lobby/alarm	1°1'2''N	1°2'3''E

Figure 4.9: IoT devices modeling example

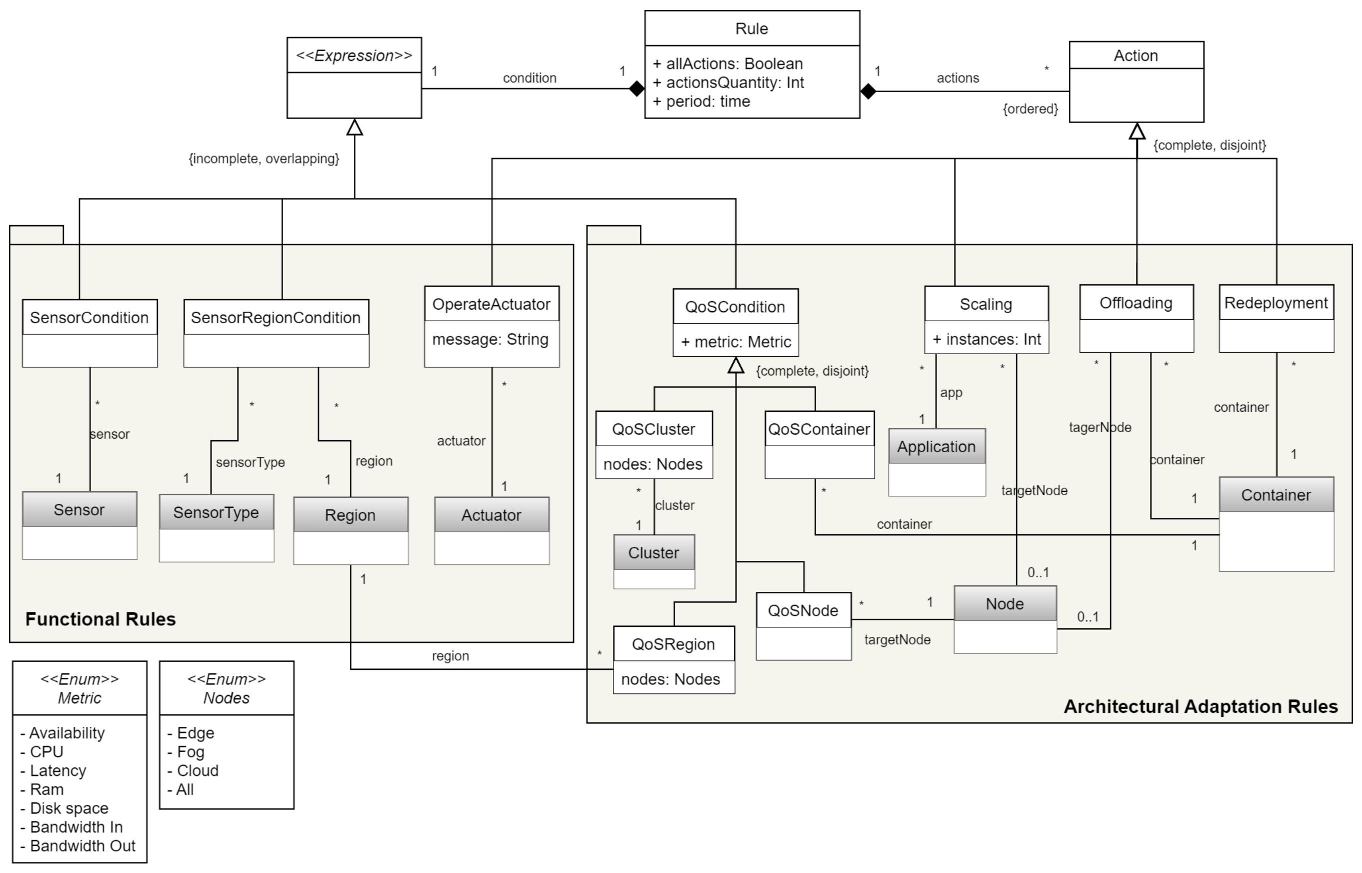


Figure 4.10: Metamodel depicting rules. The concepts shaded with gray color (such as Cluster, Application, and Node) have been previously defined in the metamodel in Figure 4.1

Every rule is an instance of *Rule* that has a triggering condition which is an expression. We reuse an existing *Expression* sublanguage to avoid redefining in our language all the primitive data types and all the basic arithmetic and logic operations to manipulate them. Such Expression language could be for instance the Object Constraint Language (OCL) but to facilitate the implementation of the DSL later on, we directly reused the MPS *BaseLanguage*[2].

The *BaseLanguage* is a Java counterpart in MPS, since it shares with Java almost the same set of constructs. The *BaseLanguage* is the most extended language in MPS that includes concepts such as *Expression* (the concept we extend in our metamodel). *Expression* is an abstract concept that represents expressions of the form "$a + b$", "$ab < cd$", "$a||b$", and other types that include mathematical and logical operators. Figure 4.11 shows an excerpt of the *Expression* metamodel with the concepts we most use to represent rules. An *Expression* can be a boolean constant (*BooleanConstant*), a numeric value with unit of measure (*Num_Value*), or a binary operation (*BinaryOperation*) composed of a left expression and a right expression (*leftExpression* and *rightExpression* relationships). *BinaryOperation*s can be mathematical (such as *Plus*, *Mul*, and *Minus*), logical (such as *Or*, *And*, and *Equals*), or binary comparison operations (*BinaryCompare*) such as >, >=, <, or <=. This sublanguage allows the design of complex rules composed of several expressions involving mathematical and logical operators.

Our metamodel (Figure 4.10) extends the generic *Expression* concept by adding sensor (*SensorCondition* and *SensorRegionCondition*) and QoS (*QoSCondition*) conditions that can be combined also with all other types of expressions (e.g. BinaryOperation) in a complex conditional expression.

A *SensorCondition* represents the occurrence of an event resulting from the analysis of sensor data (e.g., the detection of dioxide carbon gas by the *gas-a1* sensor). On the other hand, a *SensorRegionCondition* can be linked to sensor types in a region to express conditions involving a group of sensors in that region. For example, the detection of temperature rise by any of sensors in the *Floor1* region.

Similarly, the *QoSCondition* condition is a relational expression that represents a threshold of resource consumption or QoS metrics. This condition allows to check a *Metric* (such as Latency, CPU consumption, and others) on a specific node (*QoSNode* concept), on a specific container (*QoSContainer* concept), or a group of nodes belonging to a *Region* (*QoSRegion* concept) or *Cluster* (*QoSCluster* concept). For example, the condition $cpu(HotelBeach- > edge\_nodes) > 50\%$ is *QoSRegion* type and is triggered when the CPU consumption on the edge nodes of the *HotelBeach* exceeds 50%.

Moreover, we can define that the condition should be true over a certain period

[2]https://www.jetbrains.com/help/mps/base-language.html

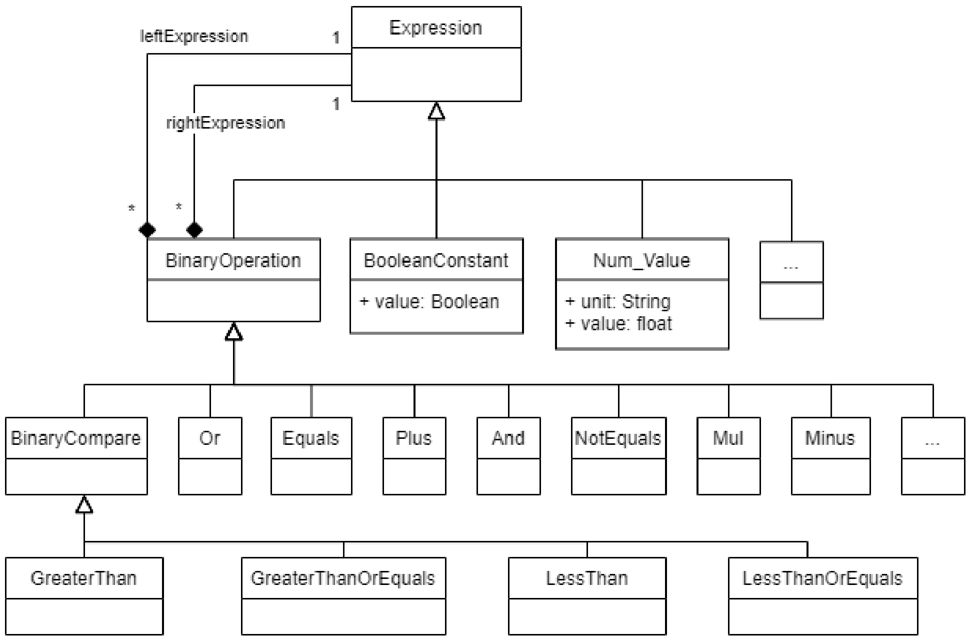


Figure 4.11: Excerpt of Expression Language metamodel

(to avoid firing the rule in reaction to minor disturbances) before executing the rule. Once fired, all or some of the actions are executed in order, depending on the *allActions* attribute. If set to false, only the number of *Action*s specified by the attribute *actionsQuantity* must be executed, starting with the first one **in order** and continuing until the required number of actions have been successfully applied.

For the sake of clarity, we have grouped the rule concepts into two categories: *Architectural Adaptation Rules* and *Functional Rules* but note that they could be all combined, e.g., a sensor event could trigger a functional response such as triggering an alarm and, at the same time, an automatic self-adaptation action, such as scaling of apps related to the event to make sure the IoT system has the capacity to collect more relevant data).

Among the *Action*s:

- the *Offloading* action consists in migrating a container from a source node to a destination node. This migration can be between nodes of different layers. The *container* relationship represents the container that will be offloaded. The target node is specified by the *targetNode* relationship. However, if the target node does not have the resources to host the container, a cluster or a group of nodes in a *Region* can be specified (*targetRegion* and *targetCluster* relationships) to offload the container.

- The *Scaling* action involves deploying replicas of an application. This application is represented by the *app* relationship, and the number of replicas to be deployed is defined by the *instances* attribute. The replicas of the application are deployed in one or several nodes of the system represented by the *targetNodes*, *targetCluster*, and *targetRegion* relationships.

- The *Redeployment* action consists in stopping and redeploying a container running on a node. The container to redeploy is indicated by the *container* relationship.

- Finally, the *OperateActuator* action is to control the actuators of the system (e.g. to activate or deactivate an alarm). The message attribute represents the message that will be published in the broker and interpreted by the actuator. This action can control actuators that require only one control value such as On/Off.

4.2.2 Concrete syntax

These rules are specified thanks to a textual notation using as keywords the names of the metaclasses of the abstract syntax. The conditions follow the grammar of a relational expression with the use of mathematical symbols (such as $<$, $>$, and $=$) and logical operators (such as & and ||). The rule editor (see Section 4.3) offers a powerful autocomplete feature to guide the designer through the rule creation process.

4.2.3 Well-Formedness Rules

We have defined three well-formedness rules using OCL (see Figure 4.12) to validate the correctness of the model describing the IoT system rules. The WFR8 rule is to initialize the *allActions* parameter equal to true of each rule. The WFR9 rule guarantees that when creating an event of type *SensorRegionCondition* there is at least one *Sensor* of type *SensorType* related in the specified *Region*. For example, the condition $Stairs[temperature] > 50^oC$ detects if the temperature in the *Stairs* region exceeds 50^oC, but this condition can only be specified if there is at least one temperature sensor deployed on the *Stairs*. Finally, the WFR10 rule ensures that the instances parameter of the *Scaling* concept is greater than zero.

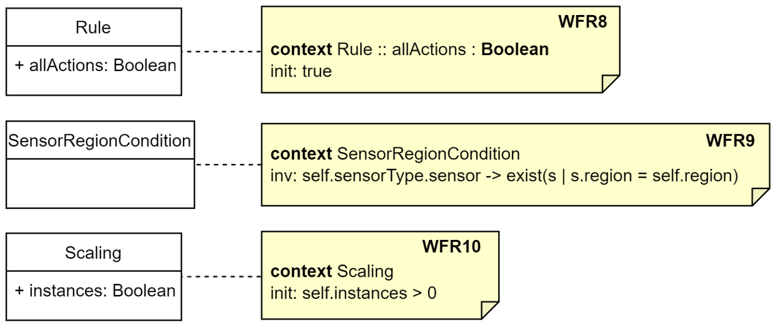


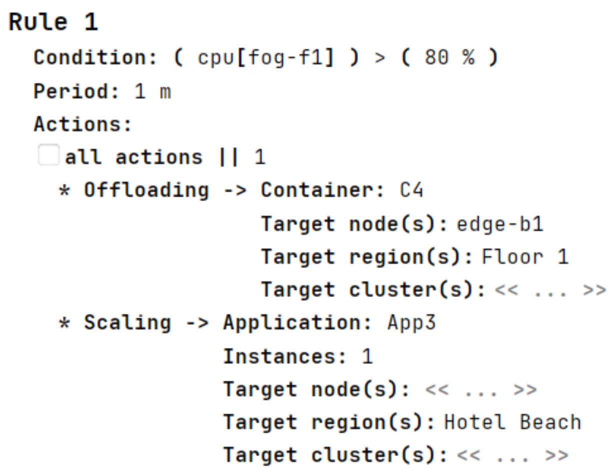
Figure 4.12: Well-formedness rules for rules metamodel

4.2.4 Example Scenario

We show how to use the rule's concrete syntax to model two rules from the smart building example.

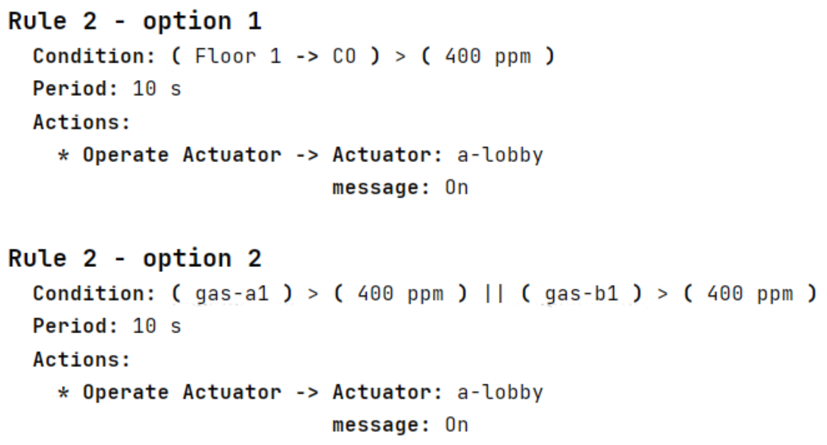
First, to guarantee the execution of the *C4* container deployed on the *fog-f1* node, we modeled the rule as shown in (Fig. 4.13). This rule offloads the container *C4* hosted on node *fog-f1* to a nearby node (e.g., node *edge-b1*) when the CPU consumption exceeds 80% for one minute. If the *edge-b1* node does not have the necessary resources to host that new container (when the rule is activated), a Region (e.g. *Floor1*) can be specified so that a suitable node will be searched there. However, if this offloading action cannot be executed, for example, because in *Floor1* there is no node capable of hosting the container, then we must define a backup action. Therefore, we have modeled a second action (*Scaling*) to deploy a new container instance of the *App3* application on any of the nodes of the *Hotel Beach*. For this rule, only one action (the first or the second one) will be executed. Therefore, the checkbox *all actions* must be unchecked and the number of actions to be performed must be set to one.

Secondly, we model another rule (see Fig. 4.14) to activate the alarm (a-lobby) when any gas sensors in the *Floor1* region (gas-a1 or gas-b1) detects a gas concentration greater than $400ppm$ for 10 seconds. The "On" message is published in the broker topic consumed by the actuator (*a-lobby* alarm). Note that there are two ways to model this rule. While Option 1 involves all CO type sensors on *Floor 1*, Option 2 directly involves both gas sensors.



```
Rule 1
  Condition: ( cpu[fog-f1] ) > ( 80 % )
  Period: 1 m
  Actions:
  all actions || 1
    * Offloading -> Container: C4
                    Target node(s): edge-b1
                    Target region(s): Floor 1
                    Target cluster(s): << ... >>
    * Scaling -> Application: App3
                 Instances: 1
                 Target node(s): << ... >>
                 Target region(s): Hotel Beach
                 Target cluster(s): << ... >>
```

Figure 4.13: *Rule 1* modeling example



```
Rule 2 - option 1
  Condition: ( Floor 1 -> CO ) > ( 400 ppm )
  Period: 10 s
  Actions:
    * Operate Actuator -> Actuator: a-lobby
                          message: On

Rule 2 - option 2
  Condition: ( gas-a1 ) > ( 400 ppm ) || ( gas-b1 ) > ( 400 ppm )
  Period: 10 s
  Actions:
    * Operate Actuator -> Actuator: a-lobby
                          message: On
```

Figure 4.14: *Rule 2* modeling example

4.3 Building a Modeling Environment for the DSL

Our DSL is implemented using MPS (Meta Programming System), an open-source language workbench developed by JetBrains. This workbench enables language oriented programming with a projectional editor in persistent abstract representation [161]. The projective edition of MPS supports language extension and composition possibilities, as well as a flexible combination of a wide range of textual, tabular, mathematical and graphical notations [29].

Projectional editors such as MPS enable editing of the model by means of projections of the abstract syntax, but the model is stored in a format (e.g. XML) independent of its concrete syntax. In other words, the user interacts with these projections, which are then translated by the editor to modify the persisted model. Some benefits of projectional editing are discussed below [162].

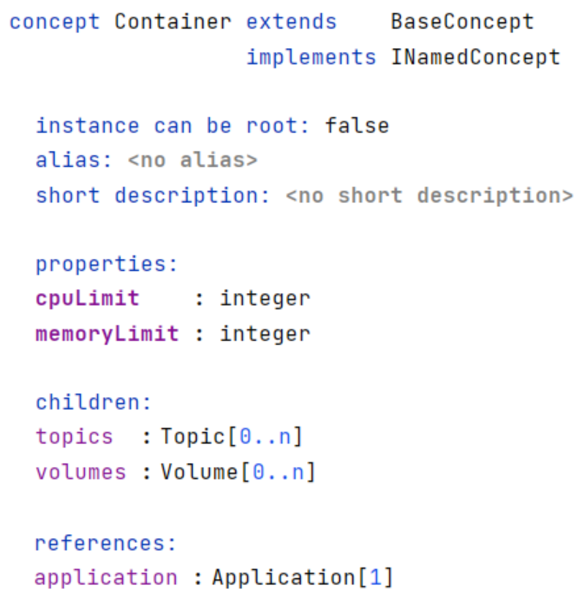
- No grammar or parser is required.
- Graphical, semi-graphical, and textual notation can be combined. For example, in Chapter 6.2 we present a graphical notation for specifying a block diagram, which uses text to define the name of each block.
- The IDE used for model editing can provide code completion, error checking and syntax highlighting.
- It is possible to provide several representations (projections) for the same model, since the model is stored independently of the concrete syntax. For example, we provide two different notations (textual and tabular) to specify the sensors and actuators of the IoT system.

Defining a language in MPS involves the design of several aspects. The definition of our DSL includes six aspects: *Structure* to define the language concepts (abstract syntax), *Editor* to define the editors for those concepts (concrete syntax), *Constraints* and *Type-System* to define a set of time-system rules and constraints (well-formedness rules), *Behaviour* to define reusable methods and functions, and *Generator* to define a code generator. These aspects are described below.

4.3.1 Structure Aspect

The definition of language begins with abstract syntax. In MPS, the structure defines the Abstract Syntax Tree (AST) of the DSL by defining all metamodel concepts. Each concept has properties (attributes), children (composition relationships), and references. Concepts can extend from other concepts to represent inheritance relationships. For example, in our metamodel (Figure 4.1), the *EdgeNode* concept extends from the *Node* concept.

Figure 4.15 shows the definition of the *Container* concept. *Container* extends *BaseConcept* (root concept in MPS) and implements the *INamedConcept* interface to inherit a name field. This concept cannot be root and has no alias or description. *Container* concept has two integer properties or attributes (*cpuLimit* and *memoryLimit*), two children or composition relations (*topics* and *volumes*), and a reference to *Application* concept.



```
concept Container extends    BaseConcept
                  implements INamedConcept

  instance can be root: false
  alias: <no alias>
  short description: <no short description>

  properties:
  cpuLimit    : integer
  memoryLimit : integer

  children:
  topics  : Topic[0..n]
  volumes : Volume[0..n]

  references:
  application : Application[1]
```

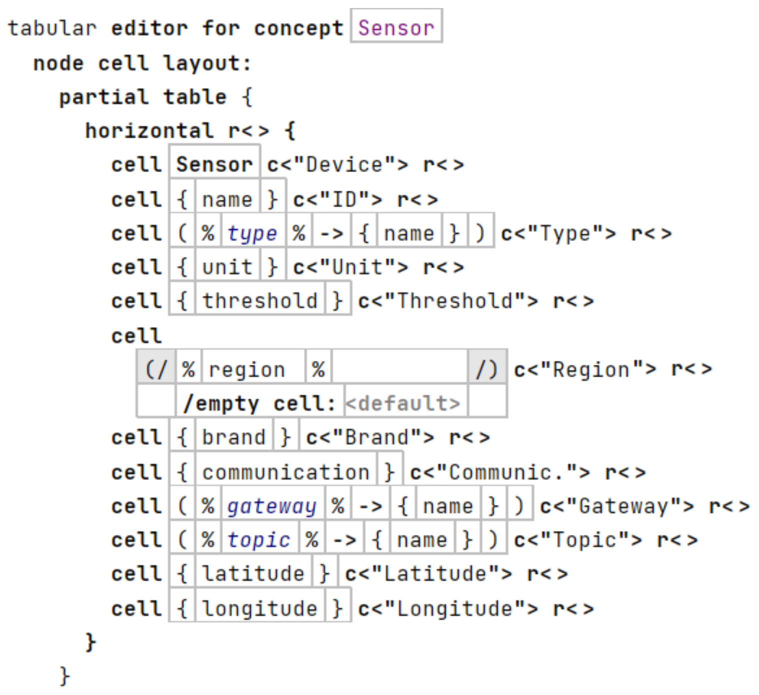
Figure 4.15: *Container* concept definition in MPS

4.3.2 Editor Aspect

Projection editors define the AST code editing rules, i.e. the concrete syntax of the language. The design of the editors greatly influences the DSL usability, since these define the notation that the user will actually use to edit the model.

Editors in MPS are based on different types of cells (the smallest unit relevant for projection). Defining an editor consists of arranging cells and defining their content [162]. We have defined textual, tabular, and tree view editors by implementing the mbeddr[3] extension of MPS. This extension simplifies the definition of cells to build different types of notations. For example, Fig. 4.16 shows the tabular editor for modeling the *Sensor* concept. We have used the *partial table* command to define the table structure (cells, content, and column headers). Cell contents can be keywords (e.g., *Sensor* in the first defined cell), concept properties (e.g., *name* in the second defined cell), properties of child nodes (e.g., *name* of concept *Type* in the third defined cell), or refer to a group of nodes (e.g., *regions* in the sixth defined cell). By defining this editor, the user is enabled to model *Sensors* using a tabular notation as shown in Fig. 4.9.

[3]http://mbeddr.com/



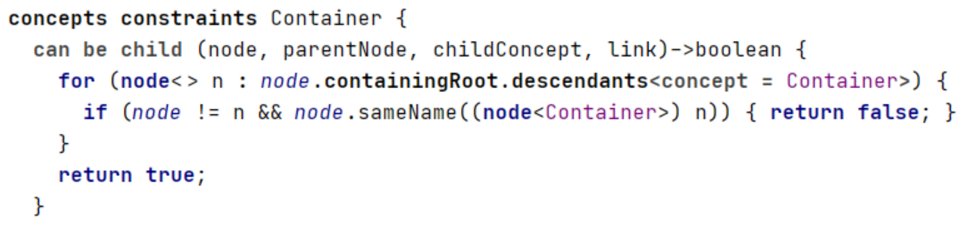
```
tabular editor for concept Sensor
  node cell layout:
    partial table {
      horizontal r<> {
        cell Sensor c<"Device"> r<>
        cell { name } c<"ID"> r<>
        cell ( % type % -> { name } ) c<"Type"> r<>
        cell { unit } c<"Unit"> r<>
        cell { threshold } c<"Threshold"> r<>
        cell
          (/ % region %             /) c<"Region"> r<>
             /empty cell: <default>
        cell { brand } c<"Brand"> r<>
        cell { communication } c<"Communic."> r<>
        cell ( % gateway % -> { name } ) c<"Gateway"> r<>
        cell ( % topic % -> { name } ) c<"Topic"> r<>
        cell { latitude } c<"Latitude"> r<>
        cell { longitude } c<"Longitude"> r<>
      }
    }
```

Figure 4.16: Definition of the tabular editor for the *Sensor* concept

4.3.3 Constraints and Type-System Aspects

Constraints aspects define well-formedness rules in the editors. Constraints restrict the relationships between nodes as well as the allowed values for properties. We have used this constraint mechanism to embed in the editor several well-formedness rules required in our DSL specification. For instance, we have added constraints to avoid repeated names, constraints to limit the potential values of some numerical attributes, constraints to restrict the potential relationships between nodes, and other constraints that prevent ill-formed models from being built. As an example, the well-formedness rule to avoid repeated names for containers (WFR6 in Figure 4.2) is shown in Fig. 4.17. We use the *can be child* method to check whether instances of the *Container* concept can be hooked as child nodes of other nodes. MPS invokes this method whenever a container node is modified in the AST, and returns false (i.e., container not allowed) if its name is repeated with that of another container.

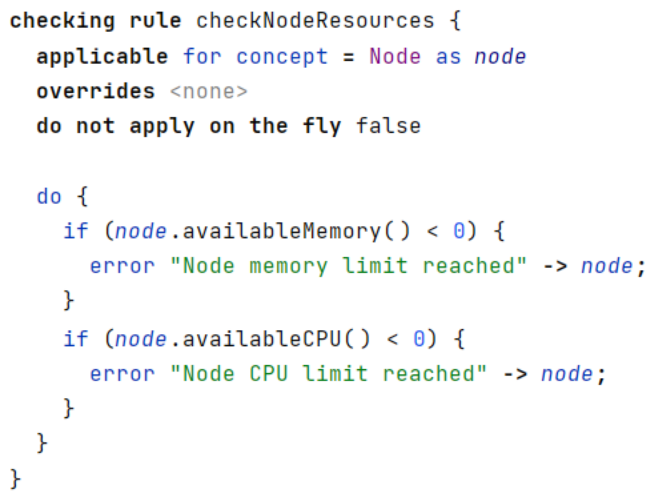
Similar to constraints, the type-system aspect allows to provide rules to check the model code.The MPS type-system engine will evaluate the rules on-the-fly, calculate types for nodes, and report errors. For example, Figure 4.18 shows the rule



```
concepts constraints Container {
  can be child (node, parentNode, childConcept, link)->boolean {
    for (node<> n : node.containingRoot.descendants<concept = Container>) {
      if (node != n && node.sameName((node<Container>) n)) { return false; }
    }
    return true;
  }
```

Figure 4.17: Container constraint to avoid repeated names

to generate an error message in the model when the containers allocated to a node exceed the node's memory and CPU capacities (WFR7 in Figure 4.2). That is, when the available memory and CPU of the node is less than zero. The methods *availableMemory* and *availableCPU* are defined in the Behaviour aspect of MPS (Section 4.3.4).



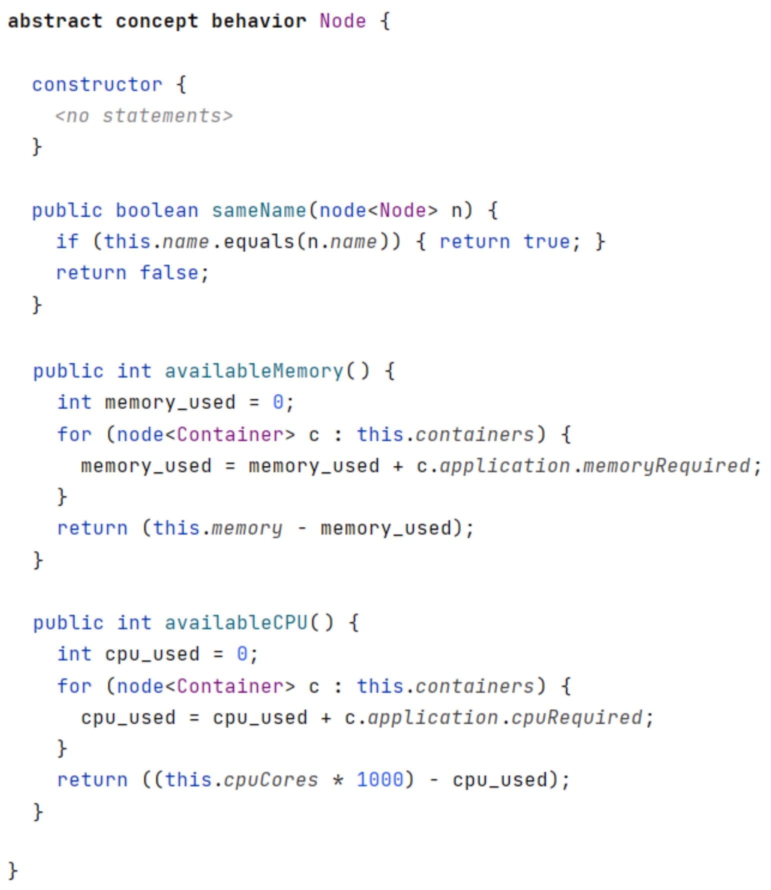
```
checking rule checkNodeResources {
  applicable for concept = Node as node
  overrides <none>
  do not apply on the fly false

  do {
    if (node.availableMemory() < 0) {
      error "Node memory limit reached" -> node;
    }
    if (node.availableCPU() < 0) {
      error "Node CPU limit reached" -> node;
    }
  }
}
```

Figure 4.18: Type-system rule to check node resources

4.3.4 Behaviour Aspect

The behavior aspect is used to encapsulate functionality that is related to the concept in the way of object-oriented programming. The methods defined in the behaviour can be instantiated in the other aspects of the model. For example, Figure 4.19 shows the methods defined for the Node concept (Note that the *availableMemory* and *availableCPU* methods are used in the type-system rule showed in Figure

4.18). To calculate the available memory of a node, first the total memory required (memory_used variable) of the containerized applications that are hosted on the node is calculated by iterating the node's containers (for loop). Then the method calculates and returns the available memory by subtracting the used memory from the node memory.



```
abstract concept behavior Node {

  constructor {
    <no statements>
  }

  public boolean sameName(node<Node> n) {
    if (this.name.equals(n.name)) { return true; }
    return false;
  }

  public int availableMemory() {
    int memory_used = 0;
    for (node<Container> c : this.containers) {
      memory_used = memory_used + c.application.memoryRequired;
    }
    return (this.memory - memory_used);
  }

  public int availableCPU() {
    int cpu_used = 0;
    for (node<Container> c : this.containers) {
      cpu_used = cpu_used + c.application.cpuRequired;
    }
    return ((this.cpuCores * 1000) - cpu_used);
  }

}
```

Figure 4.19: Behaviour methods of *Node* concept

4.4 Code Generator

We have implemented a model-to-text (M2T) transformation that generates several YAML[4] files to deploy IoT system container-based applications and runtime sup-

[4]YAML is a data serialization language typically used in the design of configuration files

port tools including the implementation of rules using PromQL[5] language. More specifically, the generated code includes configuration and deployment files to the following components (most of these components are detailed in Chapter 5):

- The container-based IoT applications specified in the input model. Following the running example of Section 3.2, the YAML manifests for deployment of containers C1, C2, C3, C4, and C5 are generated
- YAML Manifests to deploy the monitoring tools and exporters such as kube-state-metrics[6], node-exporter, and mqtt-exporter[7]
- YAML code to deploy and configure the Prometheus Storage, Prometheus Alerting Rules, and Prometheus Alert Manager components. The PromQL code to define the rules (e.g., the code shown in Listing 5.4) is also generated as a Prometheus configuration file
- YAML manifiest to deploy the Adaptation Engine
- and the Grafana application to display the monitored data stored in the Prometheus database.

To design the M2T transformation in MPS, we have used template-based generators, which consist of two main-blocks: Mapping configurations and templates [162].

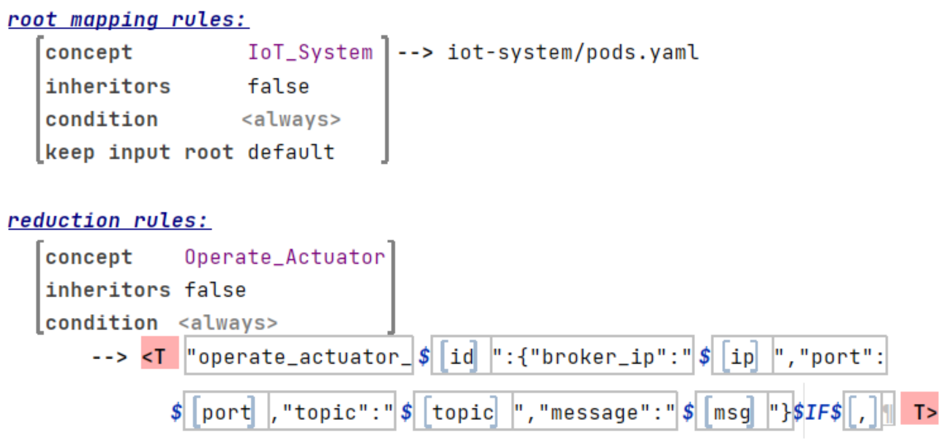
4.4.1 Mapping Configuration

Mapping configurations link the model elements with the generation of templates by means of generation rules. Our Maping configuration define 29 root mapping rules to create new artefacts (templates that generate the YAML files) from the existing model, and 15 reduction rules for in-place transformations.

Figure 4.20 shows a root mapping rule and a reduction rule. The mapping rule generates the deployment and configuration template for pods[8] and application containers named *iot-system/pods.yaml*. This rule creates a *pods.yaml* file for each *IoT_System* (root concept of the AST) defined. The reduction rule transforms adaptations of type *Operate_Actuator* into code that will be later executed by the Adaptation Engine at runtime (details of the execution engine are presented in Chapter 5). Several *property macros* (dollar sign) are used to replace the properties of the model.

[5]Prometheus Query Language to select and aggregate time series data in real time
[6]https://github.com/kubernetes/kube-state-metrics
[7]https://github.com/fhemberger/mqtt_exporter
[8]A pod is the smallest unit of Kubernetes applications

For example, the property macro *$[ip]* was configured to return the ip address of the node hosting the MQTT broker. For each adaptation and some expressions we have defined a reduction rule.



```
root mapping rules:
  [concept          IoT_System ] --> iot-system/pods.yaml
  [inheritors       false      ]
  [condition        <always>   ]
  [keep input root default     ]

reduction rules:
  [concept    Operate_Actuator]
  [inheritors false           ]
  [condition  <always>        ]
      --> <T "operate_actuator_$[id]":{"broker_ip":"$[ip]","port":
            $[port],"topic":"$[topic]","message":"$[msg]"}$IF$[,]¶ T>
```

Figure 4.20: Mapping configuration rules

4.4.2 Templates

In the template is where the transformation and code generation is performed. We have implemented the PlainText Generator [9] plugin to define the templates of our generator. The templates contain different types of macros used to calculate the value of a property (e.g., to get the name of a container), to get the target of a reference, or to control template filling at generation time.

Figure 4.21 shows an excerpt of the template that generates the YAML code for the deployment of container-based applications via pods (Note that this template es referenced in the root mapping rule in Figure 4.20). First we attach two *LOOP* macros to the template that contain the *node.nodes* and *node.containers* expressions respectively (these expressions are entered through the inspector window and are not shown in the Figure). This enables the loop through and generation of the deployment code of a pod for each container. We also use the *property macro* (dollar sign) to replace the properties of the container in the generated template such as the name, the image repository, the limit and required resources, the ports, and the volumes if any.

[9]https://jetbrains.github.io/MPS-extensions/extensions/plaintext-gen/

The code generated to deploy the application container *C1* in the *edge-a1* node of the running example is shown in Figure 4.22. Note that the parameters such as *image*, *request resources* (memory and cpu), and *containerPort* match the App1 specification (Figure 4.5). Finally, the node that will host the pod is restricted by the *nodeSelector* tag.

4.5 Installation and Configuration

You can find the DSL code in our repository[10]. To use the DSL, you have to install MPS, install some plugins, and open the project using the Toolkit. The software requirements are listed below.

- Jetbrains MPS 2020.3.1 or 2021.2.2[11]
- mbeddr platform (MPS plugin)[12]
- plaintext-gen (MPS plugin)
- MPS Table Editor Component (MPS plugin)

In the Appendix B you can find the detailed guide to install and use the DSL, generate the code, and run the framework.

4.6 Conclusion

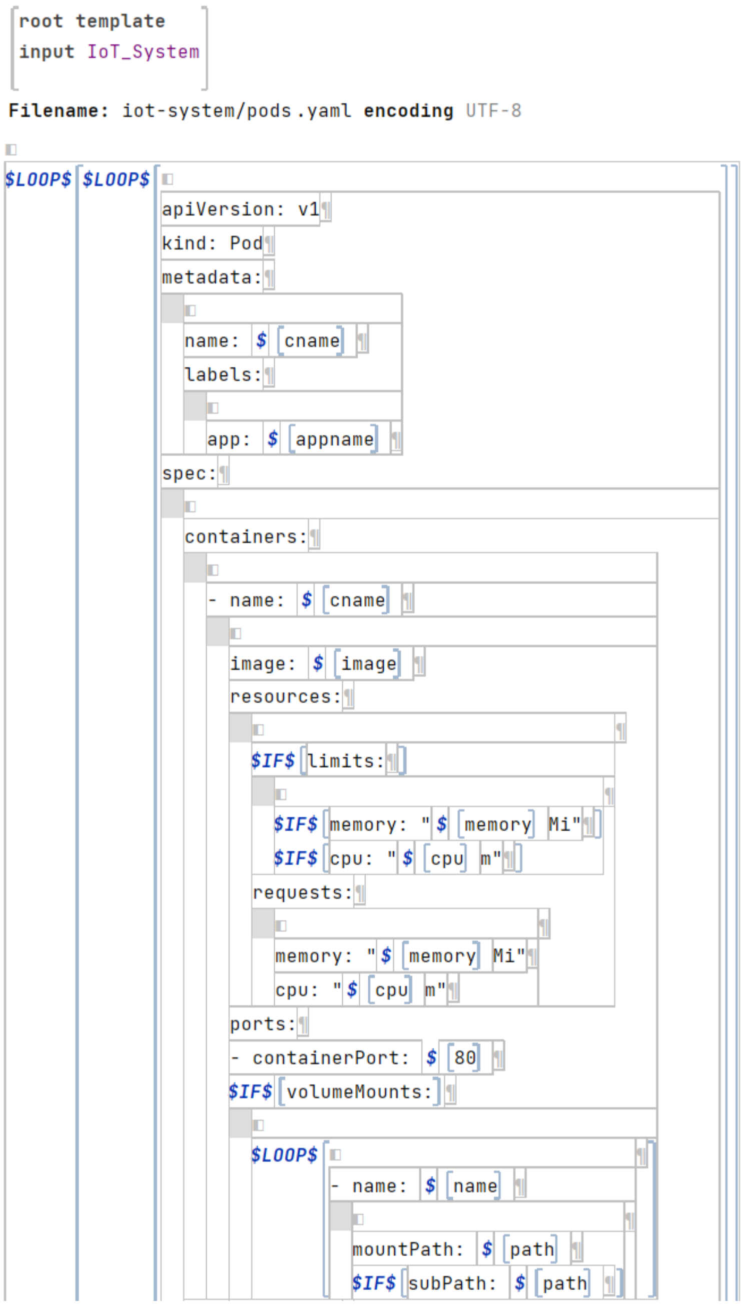
In this chapter we have presented a DSL for modeling multi-layered architectures of IoT systems and their rules (architectural adaptations and functional rules). We have introduced the abstract and concrete syntax of the DSL by illustrating the concepts through a running example of smart building. The abstract syntax is presented through meta-models that abstract the concepts that allow specifying the multi-layer architecture of the IoT system (including devices and nodes of the device, edge, fog and cloud layers), the deployment of container-based applications, and the dynamic rules to guarantee the operation of the system.

The DSL is implemented as a projectional editor created with the Jetbrains MPS tool. This gives us the flexibility to offer, and mix, a variety of concrete notations for the different concepts of the DSL. The DSL design includes the definition of several

[10]https://github.com/SOM-Research/selfadaptive-IoT-DSL.git
[11]https://blog.jetbrains.com/mps/2021/05/mps-2021-1-has-been-released/
[12]http://mbeddr.com/platform.html

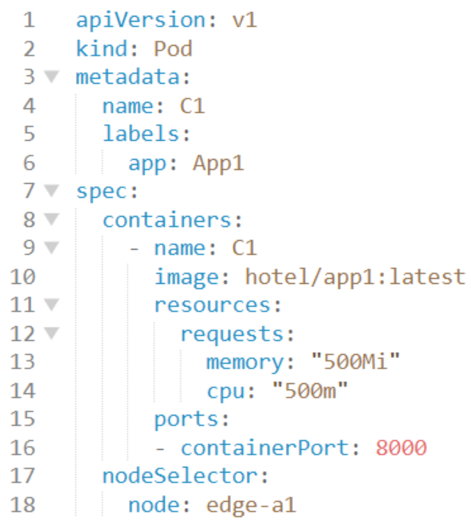
aspects such as Structure for the abstract syntax, Editor for the concrete syntax, and Constraints to define well-formedness rules.

We have also presented a code generator designed in MPS to generate software artifacts (YAML files) for the deployment and management of the IoT system at runtime. To generate the code, M2T transformations are performed by *configuring mappings* and *templates* in MPS. Transformation rules (such as root mapping and reduction rules) are defined to generate templates and obtain the generated code. The software artifacts generated include YAML manifests for deploying and configuring containerized IoT applications, kubernetes monitoring tools, Pormetheus tools, the Adaptation Engine, and Grafana.



```
root template
input IoT_System

Filename: iot-system/pods.yaml encoding UTF-8

$LOOP$ $LOOP$
apiVersion: v1
kind: Pod
metadata:
  name: $ cname
  labels:
    app: $ appname
spec:
  containers:
    - name: $ cname
      image: $ image
      resources:
        $IF$ limits:
          $IF$ memory: "$ memory Mi"
          $IF$ cpu: "$ cpu m"
        requests:
          memory: "$ memory Mi"
          cpu: "$ cpu m"
      ports:
      - containerPort: $ 80
      $IF$ volumeMounts:
        $LOOP$ - name: $ name
                 mountPath: $ path
                 $IF$ subPath: $ path
```

Figure 4.21: Excerpt of the M2T transformation for the YAML code generation of the IoT applications



```yaml
apiVersion: v1
kind: Pod
metadata:
  name: C1
  labels:
    app: App1
spec:
  containers:
    - name: C1
      image: hotel/app1:latest
      resources:
        requests:
          memory: "500Mi"
          cpu: "500m"
      ports:
      - containerPort: 8000
  nodeSelector:
    node: edge-a1
```

Figure 4.22: Generated code for *C1* container deployment

Chapter 5

Adapting IoT systems at runtime

Engineering IoT systems is a challenging task in part due to the dynamicity and uncertainty of the environment [9]. IoT systems should be designed to meet their goals by adapting to dynamic changes. In Chapter 4, we have presented our DSL for the specification of multilayered IoT architectures, architectural adaptation and functional rules, and the code generator. In this chapter we detail the design of our approach to support the self-adaptation of the IoT system at runtime. This approach is based on the MAPE-K loop, a reference model to design of self-adaptive systems [92].

The rest of this chapter is organized as follows: Section 5.1 describes our runtime approach to support IoT system adaptations at runtime. Section 5.2 illustrates our approach through the running example of the smart building. Finally, Section 5.4 concludes the chapter.

5.1 Runtime Framework

The MAPE-K loop, proposed by IBM for autonomous computing, has been often employed for the design of self-adaptive systems. Indeed, MAPE-K is a reference model to implement adaptation mechanisms in auto-adaptive systems. This model includes four activities (monitor, analyze, plan, and execute) in an iterative feedback cycle that operate on a knowledge base (see Figure 3.1). These four activities produce and exchange knowledge and information to apply adaptations due to changes in the IoT system.

Based on the MAPE-K loop, our architecture is composed of a set of components and technologies to monitor, analyze, plan, and execute adaptations as illustrated in Figure 5.1).

We next describe how our architecture particularizes the generic MAPE-K concepts for self-adaptive IoT systems.

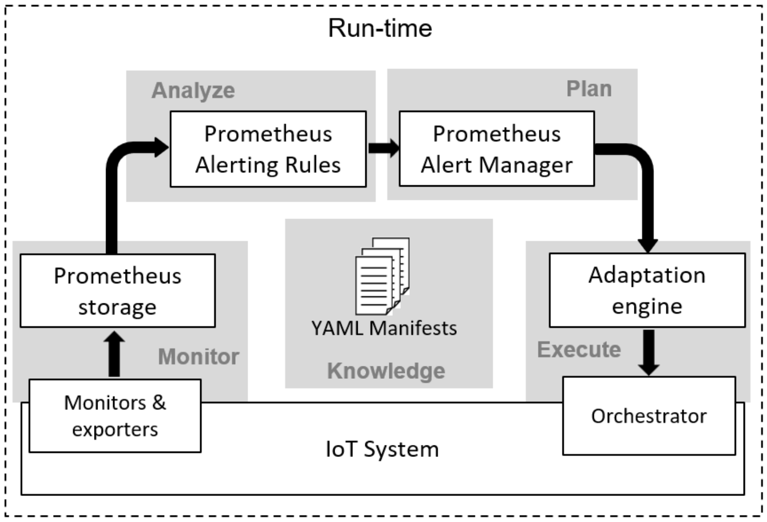


Figure 5.1: Runtime Approach based on MAPE-K loop

5.1.1 Monitor

In the Monitor stage, information about the current state of the IoT system is collected and stored. Figure 5.2 shows the Monitor stage of our framework and the YAML manifests used for deployment and configuration. We have implemented Prometheus Storage[1] (a time-series database TSDB) to store the information collected by three monitors and exporters (kube-state metrics, Node exporter, and MQTT exporter). We have adopted a time-series database because, compared to other types of databases (e.g., documentary or relational databases), Prometheus is optimized to store information in a time-efficient format, enhancing the queries performed in time windows. These queries are necessary to verify the activation of adaptation rules at run-time. Additionally, Prometheus contains modules and components that facilitate the tasks performed in the later stages of our framework such as analysis and planning (discussed in previous sections).

Four YAML manifests are required to deploy and configure Prometheus TSDB (other similar manifests are needed for monitors and exporters): deployment.yaml to deploy the Prometheus TSDB inside a Kubernetes pod, service.yaml to make it

[1]https://prometheus.io/docs/prometheus/latest/storage/

accessible from outside the cluster, config-map.yaml for configuration parameters, and clusterRole.yaml to assign the necessary privileges inside the cluster.

Exporters are deployed to convert existing metrics from third-party apps to Prometheus metrics format. Prometheus stores data as streams of timestamped values belonging to the same metric and the same set of labeled dimensions. Four types of metrics are handled by Prometheus:

1. **Counter** is an accumulative metric whose value can only increase but not decrease.
2. **Gauge** is a metric that represents a numerical value that can go up and down at any given time (e.g., processor temperature).
3. **Histograms** and
4. **Summaries** are complex metrics that record a number of observations and the sum of the observed values.

The storage of infrastructure metrics (such as CPU usage) and QoS is mostly through Gauge metrics, but for some database queries we use Histograms and Summaries (e.g. to get the CPU usage of a node in the last 5 minutes).

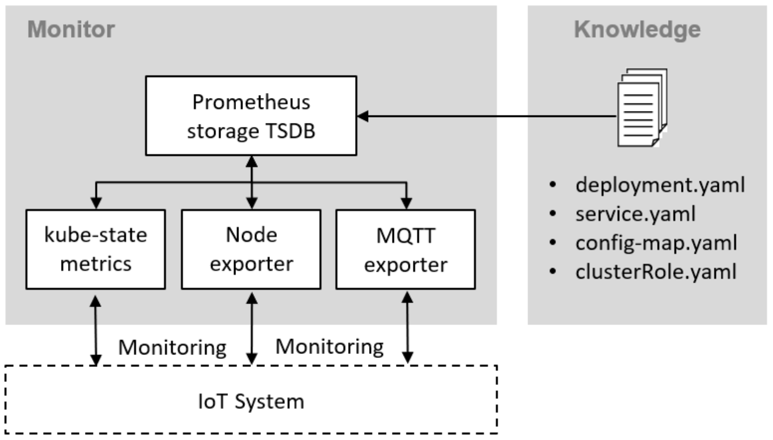


Figure 5.2: Monitor stage

The collected information is classified into two groups: (1) infrastructure and QoS metrics, and (2) data sensor metrics (published in the system's MQTT broker). These two kinds of information are aligned with the addressed types of rules, i.e., architectural adaptation and functional rules.

Infrastructure and QoS Metrics

To collect infrastructure and QoS metrics we used two monitors in addition to the basic Kubernetes monitor: the *kube-state-metrics*[2] service to collect metrics about the states of objects in Kubernetes such as nodes, pods, and deployments; and the *Node Exporter*[3] to collect Linux system-level metrics from all Kubernetes nodes. It collects hardware and operating system level metrics that are exposed by the kernel such as CPU usage and network traffic received.

These two technologies enable the monitoring of metrics such as those presented in Table 2.6 to detect different types of dynamic events. Table 5.1 lists the infrastructure and QoS metrics that can be configured in a rule using our DSL, the time series name stored in Prometheus which is used to calculate the DSL metric, the type of metric, the exporter/service capturing the metric, and a description of the Prometheus metric. Although the DSL allows to relate latency in the rules, this is the only metric that is not collected. There are several types of latency that could be monitored in the system, however, the technologies we have implemented do not support this monitoring.

[2]https://github.com/kubernetes/kube-state-metrics
[3]https://github.com/prometheus/node_exporter

Table 5.1: QoS and Infrastructure metrics

DSL metric	Prometheus time series name	Metric type	Exporter/Service	Description
Availability	up	Gauge	Kube-state-metrics	Equal to 1 if the component being monitored is available, 0 otherwise
CPU	node_cpu_seconds_total	counter	Node Exporter	Counts the number of seconds the CPU has been running in a particular mode
RAM	node_memory_MemAvailable_bytes node_memory_MemTotal_bytes	Gauge	Node Exporter	Gets the available and total Ram memory of the node
Disk usage	node_filesystem_avail_bytes node_filesystem_size_bytes	Gauge	Node Exporter	Gets the available and total disk space of the node
Bandwidth in	node_network_receive_bytes_total	Counter	Node Exporter	Counts the number of bytes of incoming network traffic to the node
Bandwidth out	node_network_transmit_bytes_total	Counter	Node Exporter	Counts the number of bytes of outgoing network traffic from the node

Data Sensor Metrics

To analyse information collected by sensors, first the data published in the MQTT broker topics must be stored in the Prometheus database. To achieve this, the *MQTT exporter*[4] is deployed, which subscribes to the broker topics to receive the sensor data and converts it to the Prometheus metric format.

Similar to other exporters, *MQTT Exporter* is deployed in a Kubernetes pod and configured through a *ConfigMap* object, which allows storing data such as key-value pairs, environment variables or command-line arguments. The configuration includes the host and port of the MQTT broker, the topics to be subscribed, the type of metric and other setup parameters. Listing 5.1 presents a portion of the *ConfigMap* produced by the code generator for our running example. Note that the host (line 9) and port (line 10) correspond to the ip address of the node hosting the broker (fog-f1 in Figure 4.6) and the port specified for the application (App3 in Figure 4.5). Lines 13-23 configure the exporter to subscribe to the smoke sensor topic deployed in *roomA1*: line 13 defines the name of the Prometheus time series that will store the topical data; line 14 provides a description of the time series; line 15 defines the type of Prometheus metric (Gauge in this example); line 15 define the topic to subscribe; and lines 17-23 configure labels on the stored data.

```
1  apiVersion: v1
2  kind: ConfigMap
3  metadata:
4    name: mqtt-exporter-config
5    namespace: monitoring
6  data:
7      conf.yaml: |
8      mqtt:
9        host:      '192.168.10.3'
10       port:       30070
11
12     metrics:
13       - name:     'floor1_roomA1_smoke'
14         help:     'Topic floor1/roomA1/smoke'
15         type:     'gauge'
16         topic:    'floor1/roomA1/smoke'
17         label_configs:
18           - source_labels:  ['__msg_topic__']
19             separator:      '/'
20             regex:          '(.*)'
21             target_label:   '__topic__'
22             replacement:    '\1'
23             action:         'replace'
```

[4]https://github.com/fhemberger/mqtt_exporter

Listing 5.1: MQTT exporter configuration

5.1.2 Analyze

The information collected in the Monitor stage must be analyzed, and dynamic events that require adaptations must be identified. To deal with this, we have used Prometheus Alerting Rules[5] (see Figure 5.3) to define alert conditions based on the rules consigned in the manifest rules.yaml. Prometheus Alerting Rules queries Prometheus TSDB using the PromQL query language. An alert is sent to the next MAPE-K loop stage (Plan) whenever one of the rule conditions is firing. Alerting Rules is a Prometheus TSDB feature so it does not require dedicated manifests to deploy.

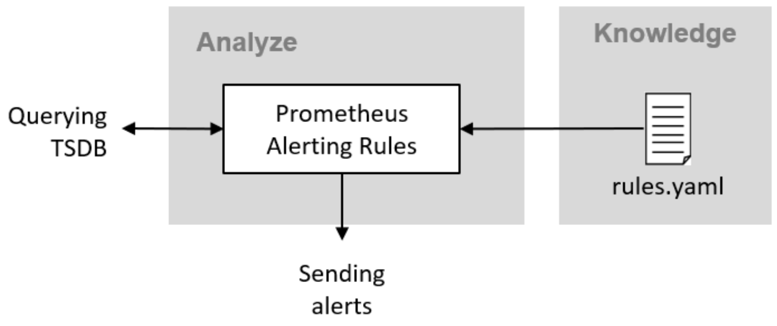


Figure 5.3: Analyze stage

Each rule consists of a name, an expression, a time period, and labels and annotations to store alert information. The code presented in Listing 5.2 is an example of an alert rule configuration for Prometheus Alerting Rules. The expression of this rule get the percentage of ram consumption (using the total ram and the available ram) of the*fog-f1* node and checks if it exceeds 80%. The *for* tag defines how long the expression must be true to generate the alert (one minute). Finally, we use the labels and annotations to include information about the actions linked to the alert. Each IoT system rule (either architectural adaptation or functional rule) specified through the DSL is transformed into an alert rule of Prometheus.

```
 - alert: HighMemoryConsumption
```

[5]https://prometheus.io/docs/prometheus/latest/configuration/alerting_rules/

```
    expr: (node_memory_MemTotal_bytes{node_hostname="fog-f1"} -
    node_memory_MemAvailable_bytes{node_hostname="fog-f1"}) /
    node_memory_MemTotal_bytes{node_hostname="fog-f1"} * 100 > 80
    for: 1m
    labels:
      severity: critical
    annotations:
      adaptations: "{...}"
```

Listing 5.2: Example rule configuration

5.1.3 Plan

According to the analysis performed in the previous stage, an adaptation plan is generated with the appropriate actions to adapt the system at runtime. The adaptation plan contains the list of actions (scaling, offloading, redeployment, or operate actuator) that the user has defined for each rule via the DSL. In this stage (see Figure 5.4), Prometheus Alert Manager is used to handle the alerts from the previous stage (Analyze) and routing the adaptation plan to the next stage (Execute). *Notification receivers* can be configured to send the alert message to third-party systems such as email, slack, or telegram. We configured a *webhook receiver* to notify the alerts and adaptation plan to the Adaptation Engine. Therefore, the adaptation plan is sent as an HTTP POST request in JSON[6] format to the configured endpoint (i.e., to the Adaptation Engine).

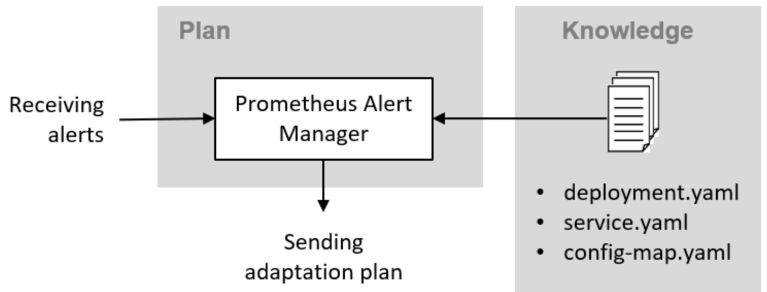


Figure 5.4: Plan stage

Three YAML manifests are required for the deployment and configuration of Prometheus Alert Manager: deployment.yaml to deploy it as a container inside a pod, service.yaml to make it accessible from outside the cluster, and config-map.yaml

[6](JavaScript Object Notation) is a lightweight data exchange format

for its configuration which includes the notification receiver (the webhook configured).

5.1.4 Execute

In the Execute stage (see Figure 5.5), adaptations are applied to the IoT system following the actions defined in the adaptation plan. To achieve this, we have built the Adaptation Engine, an application developed using Python[7], flask[8], and the python API[9] to manage the Kubernetes or K3S orchestrator. The Adaptation Engine is freely available in our repository[10] and also the image of the container[11] ready to be executed. Similar to the Prometheus Alert Manager, three YAML manifests are needed at this stage: deployment.yaml to deploy the Adaptation Engine as a container, service.yaml to configure its accessibility, and clusterRole.yaml to assign management privileges over the IoT system infrastructure (e.g. privileges to delete or create pods).

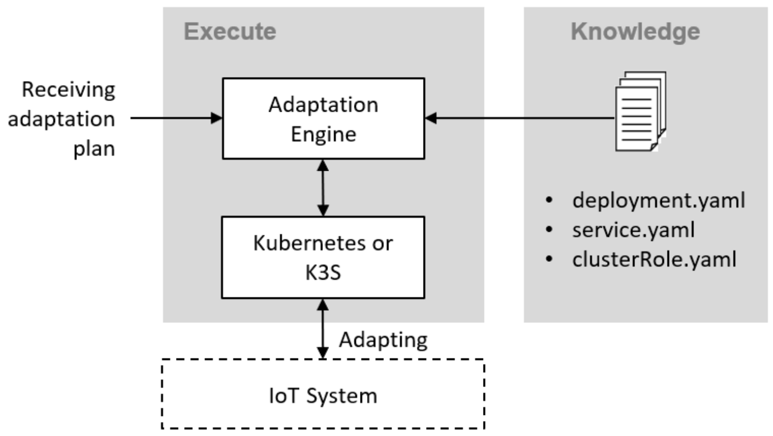


Figure 5.5: Execute stage

The Adaptation Engine can apply two sets of actions: (1) **architectural adaptations** through the orchestrator (e.g., autoscaling an application or offloading a pod); and (2) **system actuators control** to meet system functional requirements involving system actuator management (e.g., activating/deactivating alarms, turning on/off lamps, and increasing the power of a fan)

[7]https://www.python.org/
[8]https://flask.palletsprojects.com/en/2.1.x/
[9]https://github.com/kubernetes-client/python
[10]https://github.com/ivan-alfonso/adapter-engine.git
[11]https://hub.docker.com/r/ivanalfonso/adaptation-engine

There are three architectural adaptations that the Adaptation Engine is able to perform: (1) scaling an application by deploying a new pod on one of the nodes, (2) offloading a container/pod to a different node, and (3) redeploying a pod/container. These three architectural adaptations mainly benefit IoT application availability and system performance. On the other hand, the system actuators control is performed by sending control messages (defined by the user) to the actuator MQTT topic.

Using the Python API for Kubernetes, the Adaptation Engine manages the objects in the Kubernetes cluster to perform the architectural adaptations. We have defined methods for creating, deleting, and scaling pods. For example, Listing 5.3 shows an excerpt the method used by the Adaptation Engine to create a pod without node selection preferences to host it. Line 1 imports the library. Line 3 defines the method with all the input parameters needed to create the pod (these parameters are included in the adaptation plan sent from the Plan stage). Lines 4-5 create the pod object and assign its metadata (such as name). Line 6 defines the memory and cpu requirements of the container created in line 7. Line 8 sets the repository of the container image. Line 10 verifies that the pod has no node selection preferences. The software containers are assigned to the pod on lines 11-12, and the pod is created on line 13. Finally, we verify if the pod was created correctly (lines 14-16). We have created the *verify_pod_creation* method to obtain the pod status and verify its creation. To create pods that have node selection preferences, we use the Kubernetes *affinity* and *anti-affinity* specifications, which restrict the nodes that will host the pod.

```
1  from kubernetes import client, config
2
3  def create_pod(v1, pod_name, c_name, image, namespace, requirements,
       selector_nodes):
4      pod=client.V1Pod()
5      pod.metadata=client.V1ObjectMeta(name=pod_name)
6      requirements=client.V1ResourceRequirements(requests=requirements)
7      container=client.V1Container(name=c_name,resources=requirements)
8      container.image=image
9
10     if selector_nodes=="":
11         spec=client.V1PodSpec(containers=[container])
12         pod.spec = spec
13         v1.create_namespaced_pod(namespace=namespace,body=pod)
14         if verify_pod_creation(v1, pod_name):
15             return True
16         return "pod creation failed..."
```

Listing 5.3: Code excerpt to create a pod

5.2 Example scenario

As a scenario, we will exemplify the stages of the framework through the rule specified in Figure 4.13. Code for the rule management is automatically generated (Section 4.4), including YAML manifests for deployment, configuration and execution of the monitors, exporters, Prometheus, the Adaptation Engine and other software components implemented in the MAPE-K loop.

- In the Monitoring stage, the exporters gather information about CPU consumption of the *fog-f1* node. This information is stored in the Prometheus database.
- Then, in the Analysis stage, the condition of the rule is verified by executing query expressions in PromQL language. For example, the expression (executed by Prometheus Alerting Rules) that checks if the CPU consumption of the fog-f1 node exceeds 80% for 1 minute is presented in listing 5.4. Note that we are calculating the average amount of CPU time used excluding the idle time of the node. If the condition is true, the alert signal is sent to the Alert Manager component of the next stage of the cycle (Plan).
- When the alert is received, the adaptation plan is built containing the two actions (offloading and scaling) and their corresponding information in JSON format. For example, Listing 5.5 shows the JSON built by the code generator for the offloading action. The information attached to the JSON object includes the name of the pod/container to be offloaded (pod_name), the image of the application running the container (image), the Kubernetes/K3S namespace, the memory and cpu requirements of the application (requirements), and the taget nodes and target regions where the *C4* container would be offloaded, for this example, the *edge-b1* node and the *Floor 1* region.
- In the Execute stage, the Adaptation Engine component first performs the Offloading action, and only if it fails, then the second action (Scaling) is performed.

```
- alert: ram-consumption
  expr: 100 - (avg by(node_hostname) (rate(node_cpu_seconds_total{
        mode= "idle",node_hostname="fog-f1"}[15s])) * 100) > 80
  for: 1m
```

Listing 5.4: Query expression to check CPU consumption of fog1-f1 node

```
{"offloading":{
  "pod_name":"C4",
  "image":"mosquitto:2.0",
  "namespace":"default",
  "requirements":{
    "memory":"700M",
    "cpu":"700m"
  },
  "hosts":{
    "node":{
      "operator":"In",
      "values":["edge-b1"]
    },
    "region":{
      "operator":"In",
      "values":["Floor 1"]},
      "cluster":{
        "operator":"In",
        "values":[]
      }
    }
  }
}
```

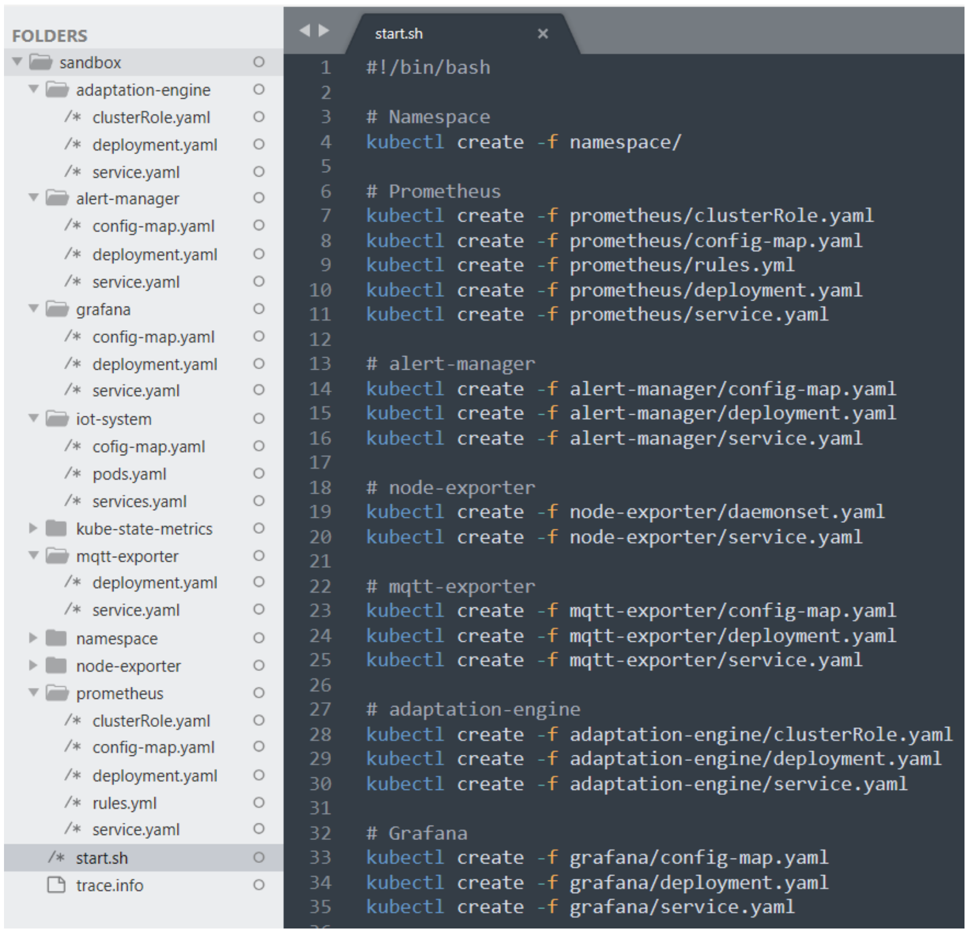
Listing 5.5: JSON adaptation plan for offloading action

5.3 Installation and Configuration

The requirements to deploy and run our framework are listed below.

- Kubernetes (v1.23.8 or later) or K3S (v1.23.8+k3s1 or later) orchestrator to manage the node cluster. We suggest K3S, a lightweight Kubernetes distribution built for IoT and edge computing.
- kubectl (v1.23.8 or later).

To run the framework tools and applications on the IoT system infrastructure, the YAML manifests (built by the code generator) must be executed on the master node of the cluster. Figure 5.6 presents the directory of generated folders and files (left side) and a snippet of the *start.sh* script (right side) for the deployment of IoT tools and applications. Executing the generated code creates several Kubernetes objects in the cluster such as *ConfigMap*s, *Deployment*s, *Service*s, and *Pod*s. To run the framework and all these Kubernetes objects just run the *start.sh* script, which uses kubectl (the command line tool of kubernetes/K3S).



```
#!/bin/bash

# Namespace
kubectl create -f namespace/

# Prometheus
kubectl create -f prometheus/clusterRole.yaml
kubectl create -f prometheus/config-map.yaml
kubectl create -f prometheus/rules.yml
kubectl create -f prometheus/deployment.yaml
kubectl create -f prometheus/service.yaml

# alert-manager
kubectl create -f alert-manager/config-map.yaml
kubectl create -f alert-manager/deployment.yaml
kubectl create -f alert-manager/service.yaml

# node-exporter
kubectl create -f node-exporter/daemonset.yaml
kubectl create -f node-exporter/service.yaml

# mqtt-exporter
kubectl create -f mqtt-exporter/config-map.yaml
kubectl create -f mqtt-exporter/deployment.yaml
kubectl create -f mqtt-exporter/service.yaml

# adaptation-engine
kubectl create -f adaptation-engine/clusterRole.yaml
kubectl create -f adaptation-engine/deployment.yaml
kubectl create -f adaptation-engine/service.yaml

# Grafana
kubectl create -f grafana/config-map.yaml
kubectl create -f grafana/deployment.yaml
kubectl create -f grafana/service.yaml
```

Figure 5.6: Directory of generated files

In Appendix B you can find the detailed guide to install and use the DSL, generate the code, and run the framework.

5.4 Conclusion

In this chapter we have presented the runtime approach to support the operation and self-adaptation of the IoT system specified by the DSL. This runtime framework is based on the MAPE-K loop and involves the implementation of several technologies to perform monitoring and adaptation of the system. Both the monitoring data from the infrastructure metrics and the data collected by the system sensors are stored in the Prometheus time series database. The collection of infrastructure and QoS metrics is performed with technologies such as kube-state-metrics and node exporter, while an MQTT exporter subscribed to the broker's topology is used to obtain the data collected by the sensors.

The analysis of the information and the triggering of alerts is performed by PromQL queries to the database and rules configured in the Prometheus Alerting Rules component. Finally, an adaptation engine developed in Python adapts the system according to an adaptation plan generated by the detection of a dynamic event.

Chapter 6

Extending DSL for specific cases

Extensibility is the ability of software to enhance or accept significant extensions without major code modifications to its basic architecture [84]. Extensible software systems take into account future growth by anticipating the need to add new functionality. Extensible DSLs allow new concepts to be added to the metamodel to enrich the language. Regarding IoT system modeling, new concepts could be specified depending on the application domain to obtain a better description of the system and its environment. For example, to model an IoT system deployed in a underground mine, it would be useful to specify domain-specific concepts such as tunnels, rooms, and working faces.

Our DSL can be used as is to model any type of multi-layered IoT system. However, it has also been designed to be easily extensible (i.e., including new concepts and updating the editors to enrich the language) so that we can further tailor it to specific types of IoT systems. This Chapter presents two extensions of our DSL: (1) in Section 6.1, we describe a DSL extension to model IoT systems for underground mining as this is a key economic sector in the local region of the PhD student and there is a need for a better way to model these systems, e.g. for analysis of regulatory compliance; (2) Section 6.2 introduces a DSL extension to model IoT systems for Wastewater Treatment Plants (WWTPs), including its its process block diagram. The implementation of IoT systems in this domain is one of the case studies addressed in the European project TRANSACT[1], in which we participate. Finally, Section 6.3 concludes the chapter.

[1]https://transact-ecsel.eu/

6.1 Modeling IoT systems for the Underground Mining Industry

Since 2016, policies have been proposed in some countries to reduce the production of carbon to reduce the pollution that this activity causes. For example, in China, the main coal-producing and consuming country, laws were proposed to achieve a 15% carbon reduction by 2040 compared to 2016 consumption [59]. These policies will lead to a decrease in global carbon production and also increase competition in the market [101]. Therefore, the coal mining industry has been significantly improving aspects such as ecological restoration, worker safety, production efficiency, and environmental pollution. This has been possible by implementing systems and technologies that allow monitoring, control, and automation of processes.

One of the aspects most enhanced by the implementation of IoT systems in underground mining is worker safety. Workers are exposed to many risk factors such as explosive and toxic gases, risk of geotechnical failure, fire, high temperatures, and humidity [96]. IoT can improve worker safety through the use of monitoring and alarm systems, voice communication systems, and geolocation of workers.

In Colombia, the safety regulation for underground mining works (decree 1886 of 2015) determines limits for the concentration of explosive and toxic gases. If any of these limits is exceeded, a series of actions/adaptations must be performed such as turning on alarms, activating the ventilation system, closing or opening ventilation ducts, and other tasks that can be executed by manipulating the system actuators on specific regions of the mine. Automating these tasks, ensuring real-time data analysis for hundreds or thousands of sensors, and addressing cloud connectivity constraints due to the remote areas in which mines are often located requires non-cloud-dependent IoT systems with multi-layered architectures that leverage edge/-fog computing.

6.1.1 Extending the metamodel

To better cope with these underground mining scenarios, our extended DSL offers new modeling primitives (see Figure 6.1). All concepts that inherit from *Region* represent physical spaces within an underground coal mine such as tunnels, working faces, Entries, and rooms. A *Tunnel* can be *Internal* or *Access*. Each mine access tunnel (*Drift*, *Slope*, or *Shaft*) must have one or more entrances (represented by the *entries* relationship). Finally, checkpoints (specific locations of the mine where gases, temperature, oxygen, and airflow are monitored) are specified through the *CheckPoint* concept. Each *CheckPoint* could contain multiple *IoTDevices* (sensors or actuators) represented by the *devices* relationship in the metamodel.

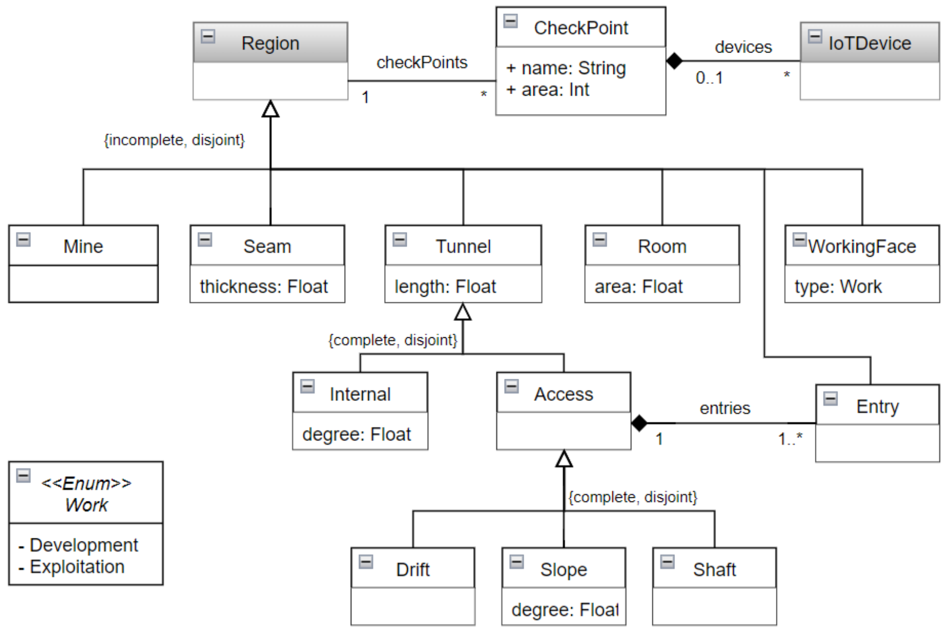


Figure 6.1: Excerpt of the DSL extension metamodel for underground mines specification. The *Region* and *IoTDevice* concepts have been previously defined in the metamodel in Figure 4.1

We enable a tree-based notation for modeling the relevant regions[2] that make up the mine structure. Figure 6.2 presents an example of the modeling of an underground mine containing two entries (*Entry A* and *Entry B*) in each of its inclined access tunnels (*Slope A* and *Slope B*), an internal tunnel (*Internal*), and a (*Room*) with two exploitation work fronts (*W-front 1* and *W-front 2*). The control points are modeled using textual+tabular notation while the rest of the concepts to represent the IoT system and the rules (architectural adaptation and functional rules) are modeled following the concrete syntax presented in Chapter 4. Figure 6.3 presents an example of modeling a control point that is located in the mine room. This control point contains an actuator (alarm) and three sensors, one for methane (CH4), one for carbon dioxide (C02), and one for temperature.

[2]Note that our DSL is focused on the structure and rules governing the “behaviour” of the IoT system of the mine, it does not intend to replace other types of 3D mine mining models

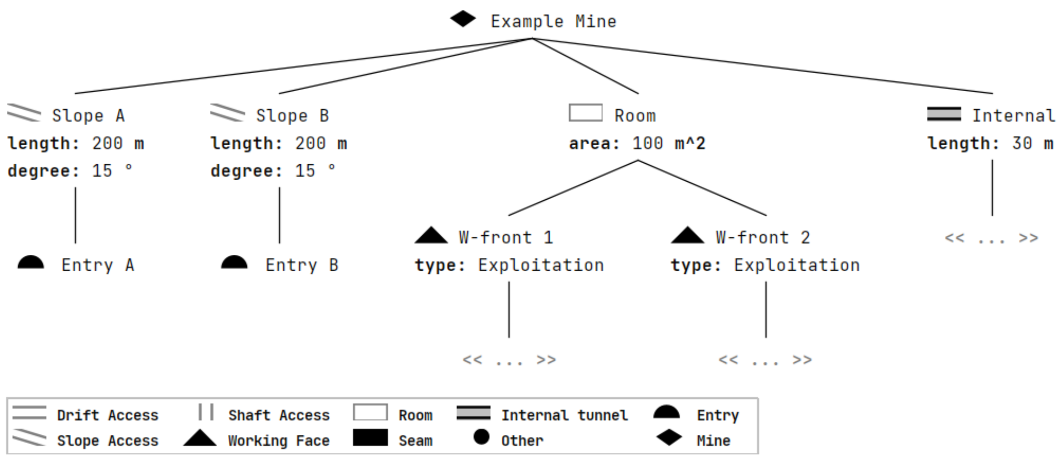
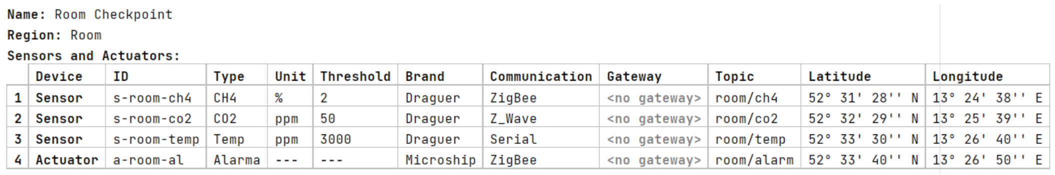


Figure 6.2: Mine structure modeling



Name: Room Checkpoint
Region: Room
Sensors and Actuators:

	Device	ID	Type	Unit	Threshold	Brand	Communication	Gateway	Topic	Latitude	Longitude
1	**Sensor**	s-room-ch4	CH4	%	2	Draguer	ZigBee	<no gateway>	room/ch4	52° 31' 28'' N	13° 24' 38'' E
2	**Sensor**	s-room-co2	CO2	ppm	50	Draguer	Z_Wave	<no gateway>	room/co2	52° 32' 29'' N	13° 25' 39'' E
3	**Sensor**	s-room-temp	Temp	ppm	3000	Draguer	Serial	<no gateway>	room/temp	52° 33' 30'' N	13° 26' 40'' E
4	**Actuator**	a-room-al	Alarma	---	---	Microship	ZigBee	<no gateway>	room/alarm	52° 33' 40'' N	13° 26' 50'' E

Figure 6.3: Mine checkpoint modeling

System actuator control

Our DSL enables the specification of functional rules in order to address functional requirements of the system by controlling the state of the actuators. For example, activating emergency alarms, turning on the ventilation system, or disabling machinery. The functional rules are specified as explained in Section 4.2. However, in this DSL extension we added support for involving control points directly in the rules.

At each mine control point, the airflow should be controlled by the fans. While very fast air currents can produce and propagate fires, very low air currents may not be efficient in dissipating gas concentrations. This extension of the DSL enables the modeling of conditions such as $(ControlPointA \rightarrow airFlow) > (2m/s)$. This condition checks if any of the airflow sensors belonging to the *ControlPointA* exceeds $2m/s$.

Architectural adaptations

Architectural adaptations of the system are also necessary in the underground mining scenario. There are several factors that can impact system operation. For example, the sampling frequencies of the sensors deployed in the mine may vary depending on conditions such as the time of day (higher monitoring frequency during working hours), the number of workers, or gas concentrations detected. Very high collection frequencies increase bandwidth and resource consumption causing failures if the system is not designed to cope.

Another factor that impacts system performance is sudden node unavailability. When an edge/fog node fails, the tasks or applications it hosted should be offloaded to nearby nodes. Node unavailability can occur due to depletion of the battery that powers the node (when it is battery powered), node overload, or damage due to hostile environment or emergencies (e.g. a landslide).

These scenarios that impact system performance can be addressed by specifying architectural adaptation rules. The IoT system architecture and rules can be specified using the textual and tabular editors presented in Section 4.2. For example, Figure 6.4 shows a rule that scales 5 instances of the gas-detection application when an increase in sensor sampling frequency (i.e., increase in input bandwidth consumption in the *gateway* node) is detected. The new 5 instances will be deployed on any of the nodes located in the *Room* region.

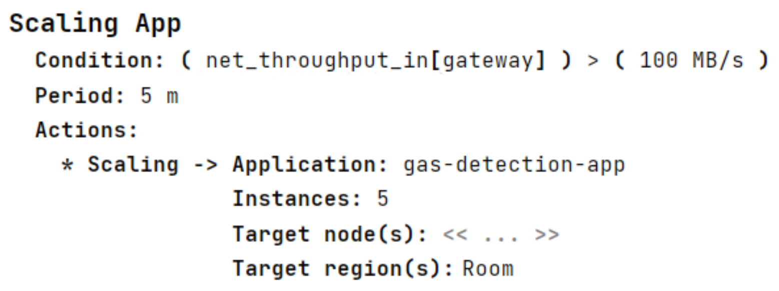

```
Scaling App
  Condition: ( net_throughput_in[gateway] ) > ( 100 MB/s )
  Period: 5 m
  Actions:
    * Scaling -> Application: gas-detection-app
                 Instances: 5
                 Target node(s): << ... >>
                 Target region(s): Room
```

Figure 6.4: Rule example for IoT system in Mining

The DSL editor extended to the underground mining domain is freely available in our repository[3]

Concrete Syntax

This DSL extension includes the definition of new editors in MPS for modeling the mine structure (tree-view editors) and the control points (tabular editors). Figure 6.5

[3]https://github.com/SOM-Research/IoT-Mining-DSL

shows the tree-view editor for the *Drift* concept composed of a swing component, the name, the length of the Drift access, and the set of *subregions* represented as the branches of the tree.

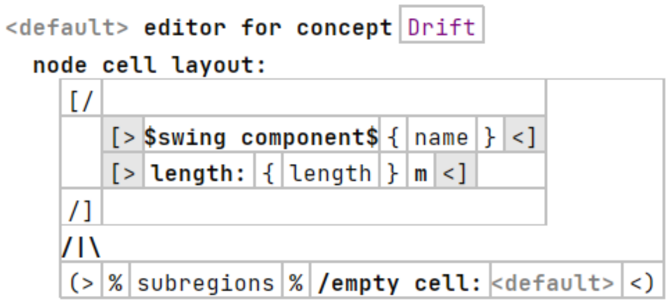
Java Swing components are inserted into the editor to create graphical shapes. The code for the swing component to create the *Drift Access* graphical shape is shown in Listing 6.1. The graphical shape is drawn on a JPanel object (container to place a set of components that can generate a graphical representation) defined in line 4. The dimensions of this panel depend on the size of the font used in the model. The *paintComponent* method defined on line 10 contains the instructions to define the shape, lines, and colors of the drawing. We have defined a graphical shape (swing component) for each concept that makes up the mine structure such as *Working Face*, *Drift Access*, *Room*, and *Mine*, as shown in Figure 6.2.

```
1  component provider: (node, editorContext)->JComponent {
2    final int fontSize = EditorSettings.getInstance().getFontSize();
3    final int desiredWidth = fontSize * 2;
4    JPanel panel = new JPanel() {
5      @Override
6      public Dimension getPreferredSize() {
7        return new Dimension(desiredWidth, fontSize);
8      }
9      @Override
10     protected void paintComponent(Graphics g) {
11       super.paintComponent(g);
12       int height = getHeight();
13       g.setColor(Color.GRAY);
14       ((Graphics2D) g).setStroke(new BasicStroke(2));
15       ((Graphics2D) g).setRenderingHint(RenderingHints.
     KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
16       g.drawLine(0, height / 4, desiredWidth, height / 4);
17       g.drawLine(0, height - 2, desiredWidth, height - 2);
18     }
19   };
20   panel.setBackground(new Color(1, 0, 0, 0));
21   panel;
22 }
```

Listing 6.1: Swing component code to draw the *Drift Access* shape

Code Generator and Framework

This extension of our DSL does not involve modifications to the code generator nor to the framework that supports runtime adaptations of the IoT system. Therefore,



```
<default> editor for concept Drift
  node cell layout:
    [/
      [> $swing component$ { name } <]
      [> length: { length } m <]
    /]
    /|\
    (> % subregions % /empty cell: <default> <)
```

Figure 6.5: Tree-view editor for the *Drift* concept

the code generator and framework implemented with this DSL are the same as those presented in Chapters 4.4 and 5 respectively.

6.2 Modeling IoT Systems for Wastewater Treatment Plants (WWTPs)

WWTPs collecting urban and industrial wastewater combine several serial processes for achieving the required quality to be discharged into the environment or reused. Sometimes, treated water is discharged into sensitive or protected environmental areas, so ensuring proper water treatment has a strong impact on the environment, the welfare of the population and agriculture in the surrounding areas. Therefore, physic-chemical and biological procedures and treatments must be rigorously monitored and controlled.

According to the European TRANSACT Project, WWTPs are automated with Supervisory Control and Data Acquisition (SCADA) systems that centralize the control and monitoring of the WWTP executing multiple local control loops in a monolithic and constrained system. One of the objectives of the project is to design a distributed multilayer architecture for the transition and evolution of the monolithic systems currently implemented by WWTPs. By integrating cloud and edge/fog capabilities into the architecture, the project seeks to optimize data analysis processes, implement spill prediction models, adopt predictive maintenance strategies, and scale critical processes by leveraging the edge layer.

However, to address the TRANSCAT objectives in this WWTPs use case, a language is required to enable system architecture modeling, application deployment, and an adaptation plan including system device control adaptations and architecture adaptations.

6.2.1 Extending the metamodel

The process flow of a WWTP is typically represented by a process block diagram, in which each block represents a physicochemical or biological treatment to remove solids and contaminants, break down organic matter and restore the oxygen content of the treated water [79]. Each treatment involves sensors and actuators that help monitor and automate the decontamination processes. For example, in desanding and degreasing treatments, probes are installed to monitor water pH, conductivity, suspended solids (SS), chemical oxygen demand (COD) and other important variables depending on the type of process.

To provide a way to specify the block process diagram of WWTPs and IoT systems immersed in these treatment processes, our extended DSL enables the modeling of new primitives that make up the process block diagram of the plant (see Figure 6.6). Concepts that inherit from *Treatment* represent a sector of the plant that performs either water or sludge treatments such as *Decanter*, *GristCahmber*, or *Filtrator*. The *Flow* concept represents the inflows and outflows (either sludge or water) to each *Treatment*. A *Treatment* can have several inflows and outflows represented by the relationships *from* and *to*. The relationship *region* specifies the region in which the *Treatment* is physically located, and the relationship *devices* allows specifying the IoT devices, either sensors or actuators, used in each *Treatment*.

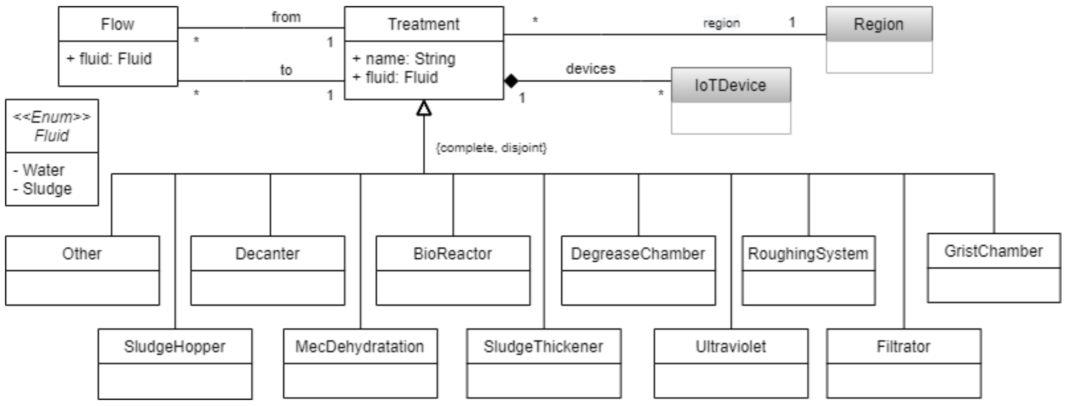


Figure 6.6: Excerpt of the DSL extension metamodel for WWTPs specification. The *Region* and *IoTDevice* concepts have been previously defined in the metamodel in Figure 4.1

To model the process block diagram we have developed a graphical editor using the MPS Diagrams plugin[4]. This plugin allows to define shadows using java code to graphically represent the metamodel concepts. For example, shapes for *Treatments* and arrows for water and sludge *Flows*. Figure 6.7 shows the process block diagram of the Algemesí WWTP, in Spain (one of the plants studied in the TRANSACT project). Each shaded shape represents one of the plant *Treatments*, while the arrows represent the flows of water or sludge between the *Treatments*. The color of the shaded shape or arrow denotes the type of fluid: blue when the *Treatment* or *Fluid* is water, and yellow when it is sludge.

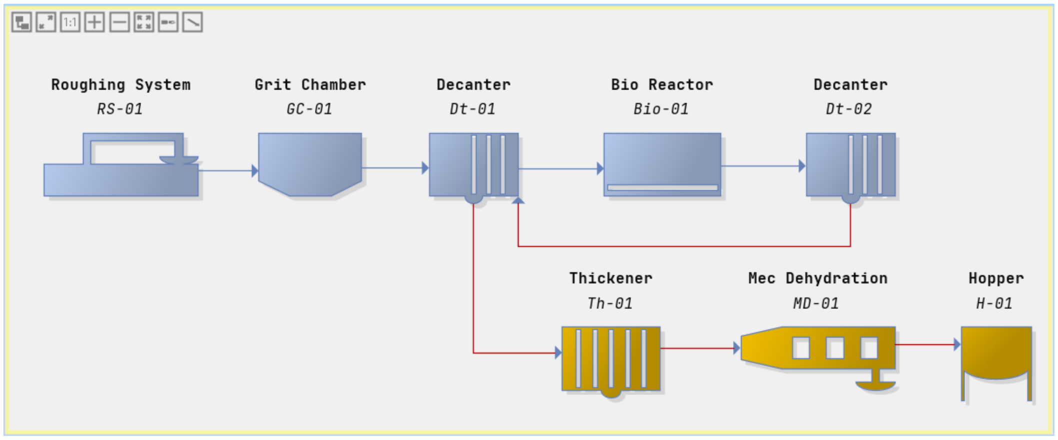
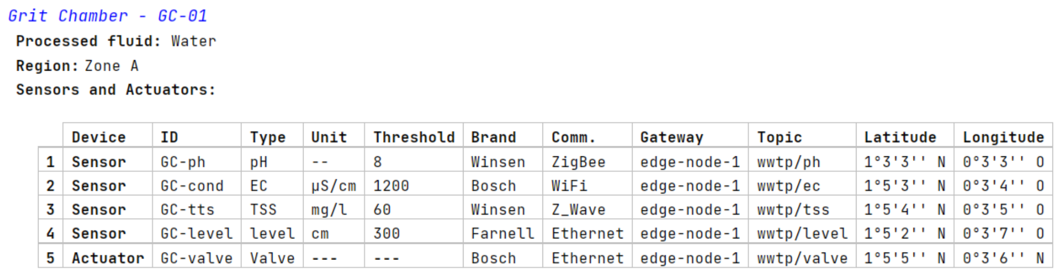


Figure 6.7: Algemesí WWTP block process diagram specification using the DSL

In the Algemesí WWTP, there are five water treatments, three sludge treatments, and several water and sludge flows between them (Figure 6.7). In each of these treatments, several variables are monitored to supervise and control the purification processes. For the specification of the sensors and actuators for each treatment, we provide a textual and tabular notation. For example, Figure 6.8 shows the specification of the Grit Chamber, which contains three sensors to monitor the liquid characteristics (pH, electrical conductivity, and total suspended solids), a sensor to measure the tank level, and an actuator (valve) to regulate the fluid level in the tank.

The dynamic environment also generates unexpected changes in WWTPs that must be dealt at runtime. For example, in rainy seasons, some WWTPs exceed treatment capacity causing a negative impact on the quality of the environment due to overflow and discharge of wastewater into the environment [79]. Although

[4]https://jetbrains.github.io/MPS-extensions/extensions/diagrams



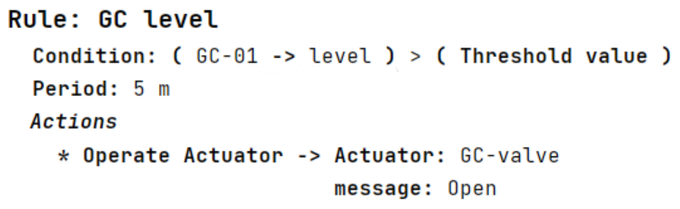
```
Grit Chamber - GC-01
 Processed fluid: Water
 Region: Zone A
 Sensors and Actuators:
```

	Device	ID	Type	Unit	Threshold	Brand	Comm.	Gateway	Topic	Latitude	Longitude
1	Sensor	GC-ph	pH	--	8	Winsen	ZigBee	edge-node-1	wwtp/ph	1°3'3'' N	0°3'3'' O
2	Sensor	GC-cond	EC	µS/cm	1200	Bosch	WiFi	edge-node-1	wwtp/ec	1°5'3'' N	0°3'4'' O
3	Sensor	GC-tts	TSS	mg/l	60	Winsen	Z_Wave	edge-node-1	wwtp/tss	1°5'4'' N	0°3'5'' O
4	Sensor	GC-level	level	cm	300	Farnell	Ethernet	edge-node-1	wwtp/level	1°5'2'' N	0°3'7'' O
5	Actuator	GC-valve	Valve	---	---	Bosch	Ethernet	edge-node-1	wwtp/valve	1°5'5'' N	0°3'6'' N

Figure 6.8: Grit Chamber Treatment modeling

increases in plant inflow are difficult to deal with, some actions can be taken to reduce the environmental impact of unwanted runoff. For example, the automatic generation of alarms, the control of valves, or the manipulation of other plant actuators when an unexpected event is detected.

These types of scenarios can be modeled using this extension of the DSL, including rules that directly involve the variables monitored in the plant *Treatment*s. For example, assuming that in the *Grit Chamber* of the Algemesí WWTP, the valve should be opened when the water level exceeds the limit (300 cm according to the sensor specification in Figure6.8) for 5 minutes, then the rule could be specified as shown in Figure 6.9.

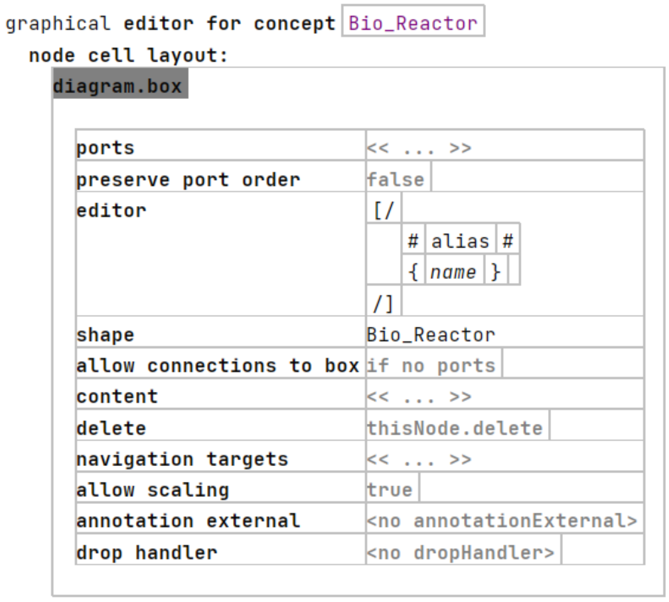


```
Rule: GC level
  Condition: ( GC-01 -> level ) > ( Threshold value )
  Period: 5 m
  Actions
    * Operate Actuator -> Actuator: GC-valve
                          message: Open
```

Figure 6.9: WWTP rule example

Concrete Syntax

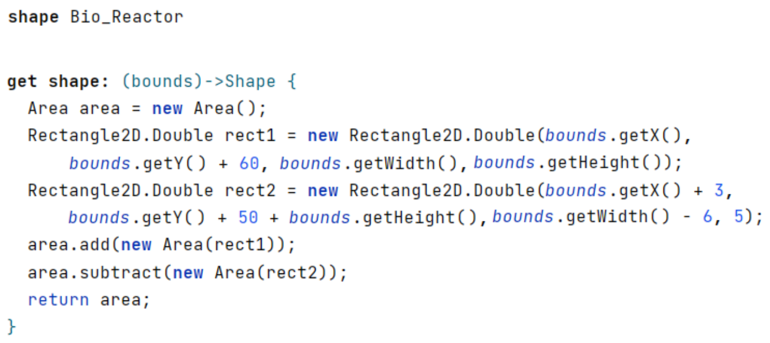
To enable graphical notation of the WWTP process block diagram, we have defined graphical editors in MPS to use shapes for draw the *Treatment*s and arrows for water or sludge *Flows*. Figure 6.10 presents the graphical editor designed for the *BioReactor* concept. The graphical form of the concept is defined using the diagram.box instruction. In the editor section we define the displayed text and in the shape section the graphical shape (i.e. *Bio_Reactor*). To define these shapes we use a DSL provided by MPS, which allows to define shapes using Java objects such as arcs, rectangles, lines, and areas. For example, Figure 6.11 shows the Bio_Reactor shape specification formed by an area that subtracts the rectangle *rect2* from *rect1*. The resulting shape for *BioReactor* concept can be seen in Figure 6.7.



```
graphical editor for concept Bio_Reactor
  node cell layout:
  diagram.box
```

ports	<< ... >>
preserve port order	false
editor	[/ # alias # { name } /]
shape	Bio_Reactor
allow connections to box	if no ports
content	<< ... >>
delete	thisNode.delete
navigation targets	<< ... >>
allow scaling	true
annotation external	<no annotationExternal>
drop handler	<no dropHandler>

Figure 6.10: Graphical editor to represent the *BioReactor* concept in the process block diagram



```
shape Bio_Reactor

get shape: (bounds)->Shape {
  Area area = new Area();
  Rectangle2D.Double rect1 = new Rectangle2D.Double(bounds.getX(),
      bounds.getY() + 60, bounds.getWidth(), bounds.getHeight());
  Rectangle2D.Double rect2 = new Rectangle2D.Double(bounds.getX() + 3,
      bounds.getY() + 50 + bounds.getHeight(), bounds.getWidth() - 6, 5);
  area.add(new Area(rect1));
  area.subtract(new Area(rect2));
  return area;
}
```

Figure 6.11: Definition of the *Bio_Reactor* shape

Code Generator and Framework

The DSL for modeling IoT systems for WWTPs requires modifications to the code generator. Specifically, it is necessary to add the *Treatment* concept in the transformation rules and generation of the rules in PromQL language. Although the DSL allows the specification of rules involving a specific *Treatment* (e.g. the rule in Figure 6.9), currently the code generator does not support these rules to be processed by our runtime framework. However, all other rules, including architectural adaptation and functional rules that do not directly involve a *Treatment* in their condition, are supported by the code generator and managed at runtime by the framework.

6.3 Conclusion

In this chapter we presented two DSL extensions for modeling self-adaptive IoT systems. The first one focused on systems implemented in the underground coal mining domain, and the second one focused on WWTP systems.

The first DSL extension covers the specification of concepts of the underground coal mining domain, including the modeling of mine areas (e.g., tunnels, working faces, and rooms) using a tree notation. Modeling of control points within the mine and their IoT devices (sensors and actuators) involved is also addressed in this DSL extension. These new concepts can be used for the specification of the architectural and functional rules.

The second DSL extension address the modeling of the WWTP process block diagrams through a graphic representation of the sequence of various treatments. The treatments are modeled as shaded shapes and the water or sludge flows are represented as directed arrows. The IoT devices involved in each water/sludge treatment are specified using tabular notation. Similar to the first DSL extension (mining domain), the new concepts can be linked to build rules.

Chapter 7

Experimental Evaluation

Although DSLs are intended to reduce the complexity of software system development, a poorly designed DSL can complicate its adoption by domain users. This is why usability studies are so important in software engineering [15]. Usability is a measure of effectiveness, efficiency and satisfaction with which users can perform tasks with a tool.

This chapter presents the empirical evaluations we have conducted to assess the usability of the DSL, and the self-adaptive capability of our approach. In Section 7.1, we present two empirical evaluations of the DSL to assess its expressiveness and ease of use. In Section 7.2 We evaluated the self-adapting feature of our approach in three scenarios to test the three adaptations (scaling, offloading and redistribution) that we addressed. Section 7.3 present the evaluation of our MAPE-K based framework to identify scalability limitations and boundaries to perform concurrent adaptations. Finally, Section 7.4 concludes this chapter.

7.1 DSL Empirical Evaluation

Based on the basic methodology for conducting usability studies [134], we have designed and conducted two experimental studies to validate the expressiveness and ease of use of our DSL. Materials and exercises provided to the subjects, the questionnaires used and the anonymized data collected can be found in our Github repository[1]. The experiments and their results are outlined below.

[1]https://github.com/SOM-Research/IoT-Mining-DSL

7.1.1 Experimental Study 1: DSL Validation - Architectural Concepts

The first experimental study aimed to validate the expressiveness and easy of use of concepts more focused on multi-level cloud architecture modeling: edge, fog, and cloud nodes, container-based applications, and architectural adaptation rules. Five computer science researchers accepted our invitation to participate in this experiment: two doctoral students, two postdoc, and one master student. A pre-questionnaire and an exercise divided into two sessions (Sessions 1 and 2) was conducted.

Design and setup

The experimental study consisted of an asynchronous screening test (pre-questionnaire) to assess subjects' prior knowledge and suitability for participation, and a synchronous exercise (virtual meeting) with two parts (Sessions 1 and 2). The materials and exercises provided to the participants, the questionnaires and the anonymized answers can be found in the repository of our DSL extended to the mining industry domain[2].

- Pre-questionnaire (10 min): the questionnaire Q0 (screening test) was completed by the participants to ensure that they had a basic level of knowledge of IoT systems, containerization technologies, and modeling tools.
- Session 1 (50 min): we have introduced a 20 minutes presentation with the basic concepts of IoT system architectures (layers, nodes, container-based applications) and modeling examples to describe the architecture using the DSL. Then the participants performed a modeling exercise of an IoT system architecture with five edge nodes, two fog nodes, one cloud node, and several software containers. Finally, the questionnaire Q1 was filled out about the expressiveness and ease of use of the DSL for modeling the system architecture.
- Session 2 (40 min): We have presented the basic concepts and examples for specifying system architectural adaptation rules. Then, participants performed a modeling exercise of five architecture adaptation rules involving infrastructure metrics (such as CPU consumption, RAM, and availability) and architecture adaptations such as application scaling or container offloading. Finally, the questionnaire Q2 was completed to gather information about the expressiveness and ease of use perceived by the participants.

[2]https://github.com/SOM-Research/IoT-Mining-DSL

The experimental study was conducted in Spanish on three different dates in 2022. The PhD student of this thesis conducted the virtual meetings and ensured that all were equally well executed.

Results

Figure 7.1 shows the level of knowledge reported by the participants about IoT systems (architecture, deployment, and operation) and containerization as a virtualization technology. Although most participants reported a low level knowledge, they are familiar with monitoring QoS metrics (such as latency, availability, bandwidth, CPU consumption) and architecture adaptations such as auto-scaling and offloading.

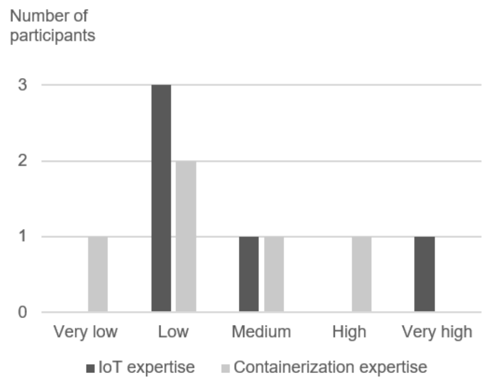


Figure 7.1: IoT and containerization expertise of participants

According to the information collected from the questionnaires, most participants have reported that it was very easy or easy to model the system architecture (nodes, applications and containers) and its adaptation rules (see Figure 7.2). Furthermore, the results presented in Figure 7.3 (errors in the models built by the participants) demonstrate the ease of use, even for non-expert users in the IoT domain. The only mistake made by a participant in specifying the condition of an adaptation rule was a wrong selection of the *targetNode* to be checked.

The suggestions and opinions of the participants about including new features or improvements to the DSL are as follows:

- Some suggestions about typo errors and minor interface improvements (editors) were reported and have already been addressed.

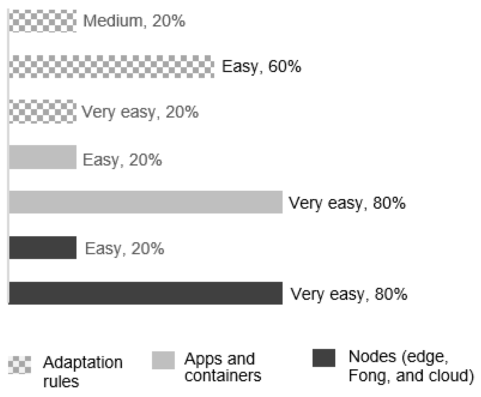


Figure 7.2: DSL ease of use (experiment 2)

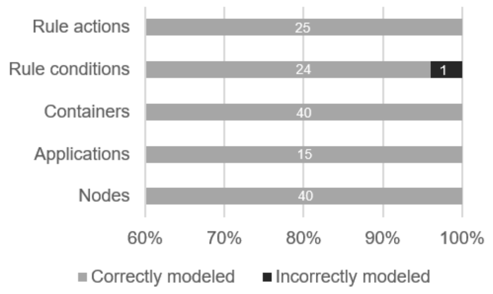


Figure 7.3: Percentage and number of right and wrong answers (experiment 2)

- Although the DSL provides textual and tabular notation for modeling the architecture nodes, including graphical notation (such as a deployment diagram) could be useful to easily follow the hierarchy of the architecture nodes.

- There may be applications that require more than one port to be exposed. However, the DSL does not allow more than one port to be associated with each application. The suggestion is to enable the specification of multiple ports for a single application.

7.1.2 Experimental Study 2: DSL Validation - Mining Concepts

We have designed the second experimental study to validate the expressiveness and usability of the DSL regarding the modeling of the mine structure, the control points, sensors and actuators, and functional rules involving control of actuators (e.g. turn on the mine ventilation system when the methane gas sensor exceeds the threshold value).

Design and setup

Eight subjects participated in the experimental study: Three electronic engineers, one industrial engineer, one systems engineer, one mining engineer, and two automation engineers. All participants have previously conducted projects or work in the mining domain, so they were familiar with this domain, but had not been exposed to our DSL before. The goal was to check whether they were able to use it and get their feedback on the experience.

The protocol for this experimental study was similar to the previous one. A pre-questionnaire and an exercise divided into two sessions (Sessions 1 and 2) were conducted. The materials and exercises provided to the participants, the questionnaires and the anonymized answers can be found in the repository of our DSL extended to the mining industry domain[3].

- Pre-questionnaire (10 min): this screening questionnaire (Q0) was conducted prior to the start of Sessions 1 and 2 to ensure that participants had a basic level of mining knowledge including the structure of underground coal mines, gas monitoring systems, and modeling tools in this domain.

- Session 1 (50 min): In the first 20 minutes of Session 1, we introduce the basic knowledge of IoT systems and the use of the DSL implemented in MPS to model the structure of an underground mines, the control points and the IoT devices deployed (sensors and actuators). Next, the participants performed the first modeling exercise about an underground coal mine (with the structure shown in Figure 6.2), two control points (one at each working face) with three gas sensors and an alarm, a fan, and a control door in the internal tunnel. Each participant was provided with a virtual machine configured with the necessary software to perform the modeling exercise. Finally, the participants filled out a questionnaire (Q1) about the usability and expressiveness of the DSL to model the concepts of the first exercise.

- Session 2 (40 min): In Session 2, we first introduced the basic concepts of self-adaptive systems and the design of functional rules using our DSL. Next, participants performed the second exercise: modeling three functional rules involving sensor data and actuator operation. For example, if any of the methane gas sensors throughout the mine exceed the threshold value for 5

[3]https://github.com/SOM-Research/IoT-Mining-DSL

seconds, then turn on the fan and activate the alarms. Finally, participants completed the questionnaire Q2 to report their experience modeling the rules. Q2 also contained open-ended questions to obtain feedback on the use of the entire tool and suggestions for improvement.

Results

Four of the participants were involved in education (either students, teachers, or researchers), while the remaining four were involved in industry. Figure 7.4 presents the level of general mining knowledge (very low, low, medium, high, and very high) of the eight participants. All of them are aware of the terminology used in the design and structure of underground coal mines. Only two participants were not familiar with cyber-physical or IoT systems for mining. The modeling tools in mining context that they have used are AutoCAD[4] and Minesight[5] for the graphical design of the mine structure, and VentSim[6] for ventilation system simulations. However, these mining modeling tools do not allow modeling of self-adaptive IoT systems. None of the participants were familiar with MPS.

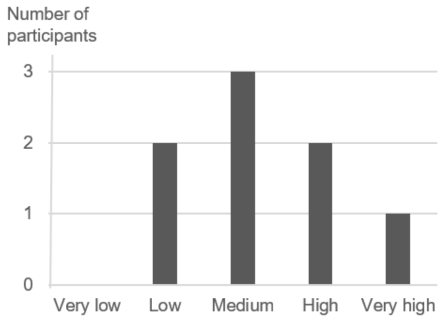


Figure 7.4: Participants' mining expertise

Figure 7.5 presents the responses from questionnaires Q1 and Q2 related to the ease of use of the DSL. Most of them reported that the modeling of the mine structure, the control points, the devices (sensors and actuators), and the adaptation rules were easy. The results are positive and can also be evidenced by the number of right and wrong concepts modeled by the participants (Figure 7.6). The number of errors were low (12 of 188 modeled concepts): three incorrect *Rule-conditions* by wrong selection of the unit of measure, four incorrect *Actuators* by wrong assignment of

[4]https://www.autodesk.com/products/autocad
[5]https://www.ici.edu.pe/brochure/cursos-personalizados/ICI-MINESIGHT-Personalizado.pdf
[6]https://ventsim.com

actuator type and location within the mine, three missing *Sensors* not modeled, and two incorrect *Regions* (working faces) whose type was not selected.

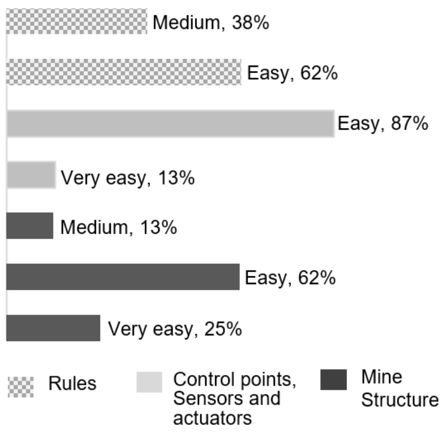


Figure 7.5: DSL ease of use (experiment 1)

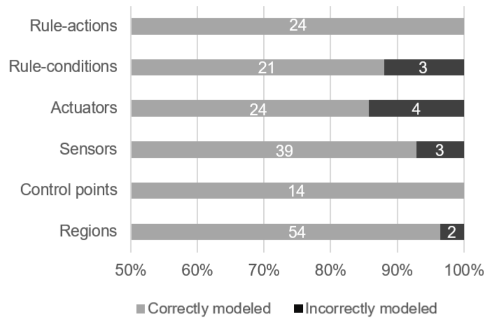


Figure 7.6: Percentage and number of right and wrong answers (experiment 1)

Through the open-ended questions in questionnaires Q1 and Q2, participants suggested the following improvements to the DSL.

- Include the specification of the coordinates for each region and control points of the mine. Additionally, it would be useful to specify the connection between internal tunnels.

- The condition of a rule has a single time period. However, it would be useful to associate two time periods for conditions composed of two expressions. For example, the condition $tempSensorA > 30C(10seconds)$ && $tempSensorB > 35C(20seconds)$.

- The mine ventilation system can be activated periodically at the same time each day. It would be useful if the DSL could model rules whose condition is associated with the time of day.

7.1.3 Threats to Validity

Although validation problems in empirical studies are always possible, we have looked for methods to ensure the quality of the results analyzing two types of threats: internal and external.

Internal validation concerns factors that could affect the results of the evaluation. To avoid defects in the planning of the experiments and the questionnaires (protocol), the PhD student and advisors discussed the protocol including the modeling exercises, the dependent variables, and the questions of the questionnaires. In addition, a senior researcher in empirical experiments validated the questionnaires. Another common thread is related to the low number of samples to successfully reveal a pattern in the results. Thirteen users total participated in this empirical validation. Five participants were involved in Experiment 1, and eight different ones in Experiment 2. Although the size of the participant group for this type of validation continues to be a matter of discussion, studies suggest 3 to 10 participants (depending on the level of complexity) are sufficient. For example, a popular guideline in this area is given by Nielsen and Landauer [120], who suggest that five participants are likely to discover 80% of the problems

External validity addresses threats related to the ability to generalize results to other environments. For example, to validate the population and avoid sampling bias, we conducted a pre-questionnaire to the participants to ensure that they had the necessary basic knowledge and that there was no substantial difference between participants. In addition, at the beginning of each session, we introduced the definition of the concepts required for the experiment. It is important to emphasize that the participants of Experiment 1 were computer science researchers while those of Experiment 2 were related to the topics of the mining domain.

7.2 Evaluation of System Self-Adaptations

To evaluate the self-adaptation capability of our approach, we conducted three empirical evaluations, one for each type of architectural adaptation (scaling, offloading, and redeployment). For each adaptation assessment we have designed a simple scenario in which an IoT system faces an event that forces adaptations. **The goal of**

this evaluations is to compare the availability and performance of a non-adaptive IoT system with that of a self-adaptive IoT system that is modeled and managed using our approach. To do so, we have collected and analyzed metrics such as CPU consumption, node availability, and data processing time.

7.2.1 Experiment Design and Setup

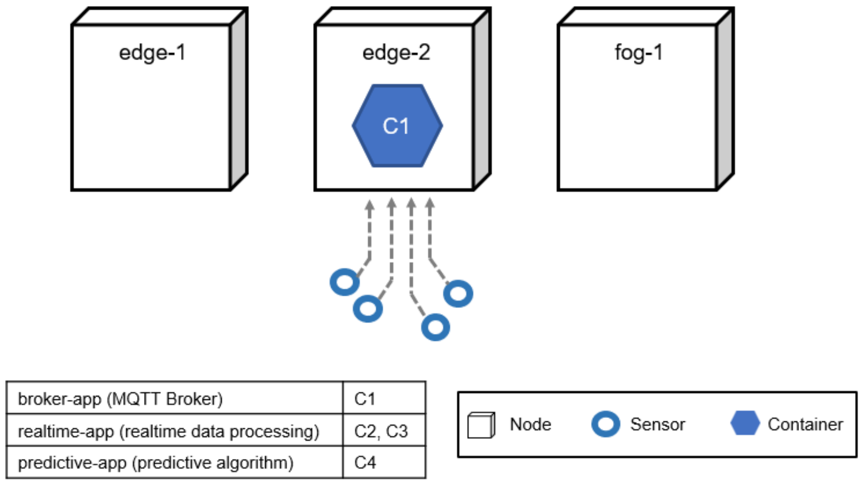
To test the three architectural adaptations, we have designed a test scenario with the IoT system shown in Figure 7.7. The system architecture consists of four temperature sensors, two edge nodes (t2.micro AWS instances[7]), one fog node (t2.medium AWS instance), and three applications (*broker-app*, *realtime-app*, and *predictive-app*) executed by four software containers (C1, C2, C3, and C4).

- broker-app: MQTT broker that manages messages published by sensor devices. We used Eclipse Mosquitto[8] as message broker because it is lightweight, easy to configure, and is suitable for running at the edge layer. This broker is deployed on the *edge-2* node and executed by the C1 container.
- realtime-app: application subscribed to the MQTT broker to to consume the data, coming from the sensors, and perform real-time data analysis. For each sensor data received on the broker topics, this application creates a thread or subprocess that performs a series of operations to intentionally generate workload on the node. After processing each sensor data, the result is published in another broker topic.
- predictive-app: this application simulates the execution of a predictive algorithm to forcast possible temperature emergencies. The algorithm (subscribed to the broker's topics) receives data from sensors and performs mathematical routines using the NumPy[9] package.

Additionally, we have developed a Python script which publishes random values on topics of the MQTT broker to simulate the four temperature sensors. During the test execution, the sampling rate of the sensors is increased incrementally to perturb the system and induce self-adaptation.

Figure 7.7 does not show how containers C2, C3, and C4 are deployed on the nodes, because the use of these containers will depend on the type of adaptation to be tested. Section 7.2.2 shows the protocol of the experiments including the deployment and adaptation of these containers.

[7]https://aws.amazon.com/es/ec2/instance-types/t2/
[8]https://mosquitto.org/
[9]https://numpy.org/



broker-app (MQTT Broker)	C1
realtime-app (realtime data processing)	C2, C3
predictive-app (predictive algorithm)	C4

Figure 7.7: General test scenario for adaptations

7.2.2 Protocol

For each type of architectural adaptation we performed a trial that generally follows the same protocol consisting of the following steps:

1. Model the IoT system (using our DSL) including the rule to be tested. The model built for these experiments can be consulted in Appendix C.

2. Run the code generator using the model built in the first step.

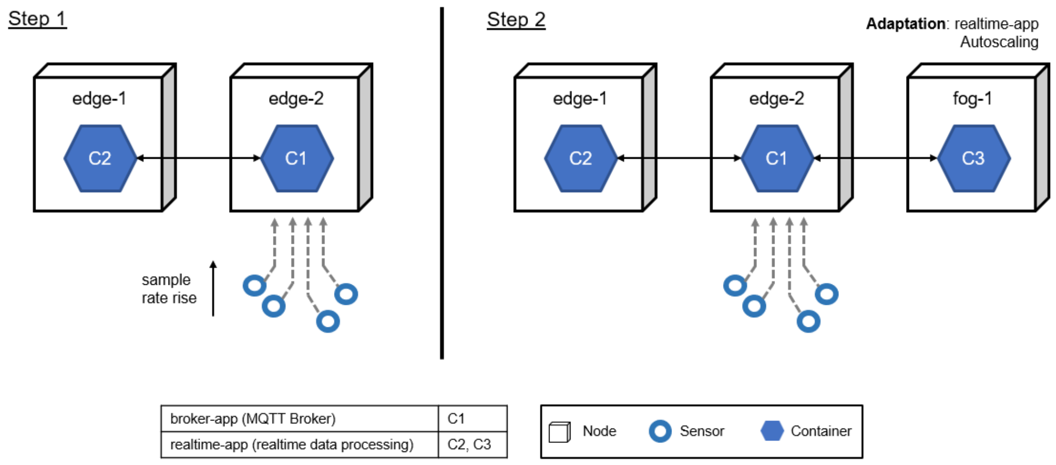
3. Deploy the IoT applications and execute our runtime framework (which is described in Chapter 5) using the YAML manifests built by the code generator.

4. Generate disturbances or dynamic events and monitor availability and performance of the system when: (a) the system has no self-adaptation capabilities, and (b) the system self-adapts using our framework.

The variable we manipulate in these experiments (**independent variable**) is the sampling frequency of the sensors in order to generate node overhead, while the variables we monitor (**dependent variables**) are the availability and performance of the nodes. Although the protocol of the three experiments are similar, some aspects change depending on the type of adaptation (scaling, offloading, redeploying) to be tested.

Scaling an application

The experiment scenario to test *Scaling* adaptation consists of two steps as shown in Figure 7.8.



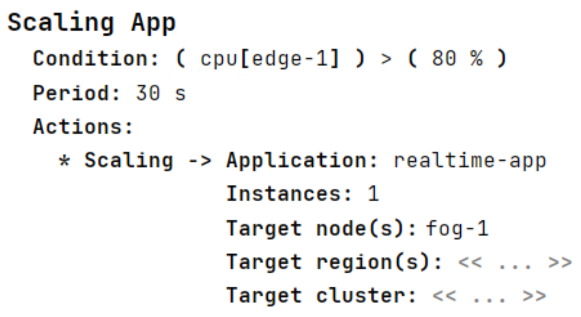
broker-app (MQTT Broker)	C1
realtime-app (realtime data processing)	C2, C3

Figure 7.8: *Scaling* adaptation test scenario

In step 1, we gradually increase the sampling or monitoring frequency of the sensors to overload the *edge-1* node by increasing the amount of data to be processed by the *C2* container. Initially, we simulate sending 5 data per second to the MQTT broker for two minutes. Then we increase to 12 data per second for the next two minutes. Finally we increase to 30 data per second. We have set these values to intentionally generate a progressive increase in the node's workload.

In step 2, after overloading the *edge-1* node, the realtime-app application is scaled. The *Adaptation Engine* of our framework deploys the *C3* container on the *fog-1* node. Then, the data coming from the sensors are distributed for analysis between the *C2* and *C3* containers. This distribution of messages among subscribers is achieved by a balancing strategy known as MQTT shared subscription [111]. In a shared subscription, all clients sharing the same subscription receive messages alternately, i.e., each message will only be sent to one of the subscribed clients.

Figure 7.9 shows the rule that we have specified using the DSL to test the *Scalability* adaptation. This rule states that if the CPU usage of the *edge-1* node is greater than 80% for 30 seconds, then scale an instance of the *realtime-app* application on the *fog-1* node. For this rule, we have chosen to check the CPU usage of the node as it is a metric that reflects the work overhead of the node.

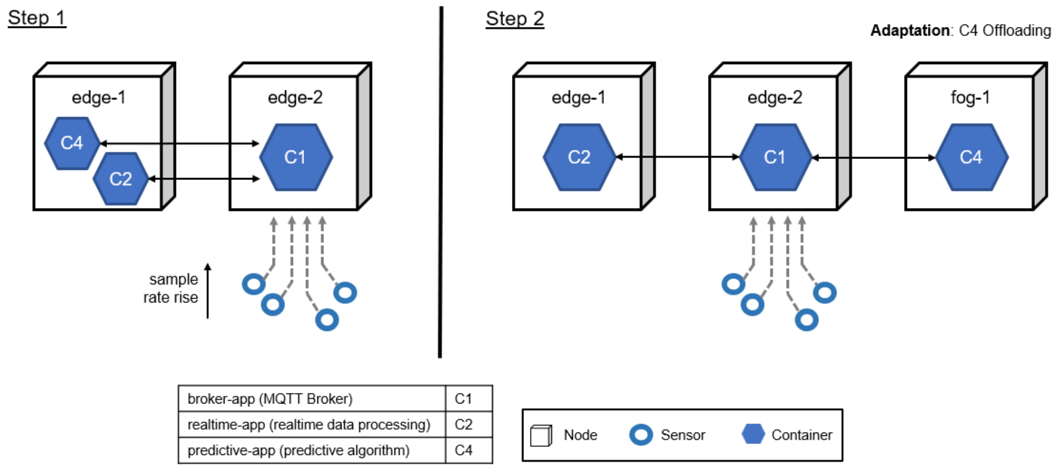


```
Scaling App
  Condition: ( cpu[edge-1] ) > ( 80 % )
  Period: 30 s
  Actions:
    * Scaling -> Application: realtime-app
                 Instances: 1
                 Target node(s): fog-1
                 Target region(s): << ... >>
                 Target cluster: << ... >>
```

Figure 7.9: *Scaling* adaptation rule

Offloading a container

The experiment scenario to test the *Offloading* adaptation consists of two steps as shown in Figure 7.10.

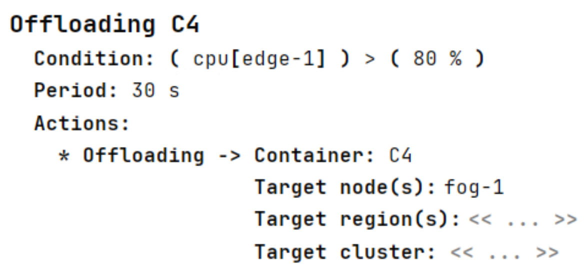


broker-app (MQTT Broker)	C1
realtime-app (realtime data processing)	C2
predictive-app (predictive algorithm)	C4

Figure 7.10: *Offloading* adaptation test scenario

Similar to our previous experiment, in step 1 the sampling frequency of the sensors is gradually increased (5, 12 and 20 data per second) until an overload is generated in the node *edge-1* due to the increase of sensor data processed by the *C2* container. The frequency of data generation is different from our previous trial (*Scaling an application*), because the scenario is different (for this trial, two containers on the node are executed). This node (*edge-1*) hosts the *C2* and *C4* containers that run the *realtime-app* and *predictive-app* applications. Then, in step 2, the system offloads the *C4* container to the *fog-1* node freeing resources on the *edge-1* node.

The rule modeled is shown in Figure 7.11: if the CPU usage of node *edge-1* exceeds 80% for 30 seconds, then the *C4* container is offloaded to node *fog-1*.



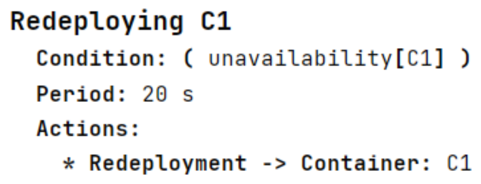
```
Offloading C4
  Condition: ( cpu[edge-1] ) > ( 80 % )
  Period: 30 s
  Actions:
    * Offloading -> Container: C4
                    Target node(s): fog-1
                    Target region(s): << ... >>
                    Target cluster: << ... >>
```

Figure 7.11: *Offloading* adaptation rule

Redeploying a container

The scenario for testing the *Redeployment* adaptation is the same as shown in Figure 7.7. First we force a failure in the C1 container by logging into the pod using the command line tool and stopping some system processes. Then, our runtime framework, which constantly monitors the state of containers, detects container unavailability and redeploys it using the *Adaptation Engine*.

The rule modeled for testing the *Redeployment* of a container is shown in Figure 7.12. If the container *C1* is detected to be unavailable for more than 20 seconds, then it is redeployed.



```
Redeploying C1
  Condition: ( unavailability[C1] )
  Period: 20 s
  Actions:
    * Redeployment -> Container: C1
```

Figure 7.12: *Redeployment* adaptation rule

7.2.3 Results

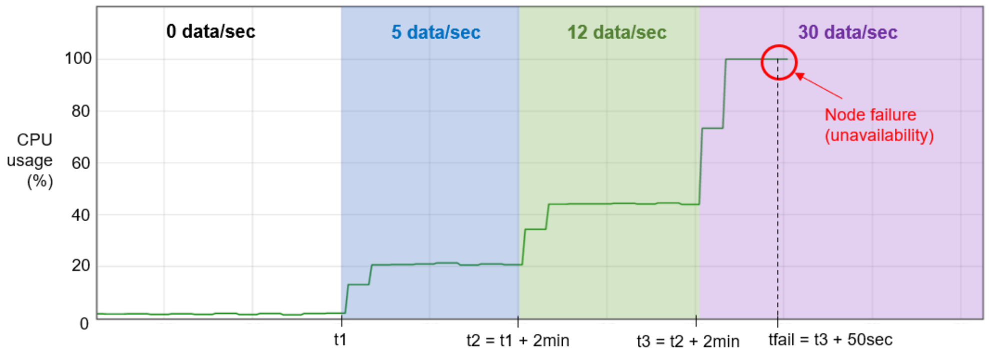
In the test scenario for each type of adaptation, the system was monitored in two cases: (1) without implementing adaptations, and (2) self-adapting the system according to the configured adaptation rules. The results for each type of adaptation are compared below.

Scaling an application

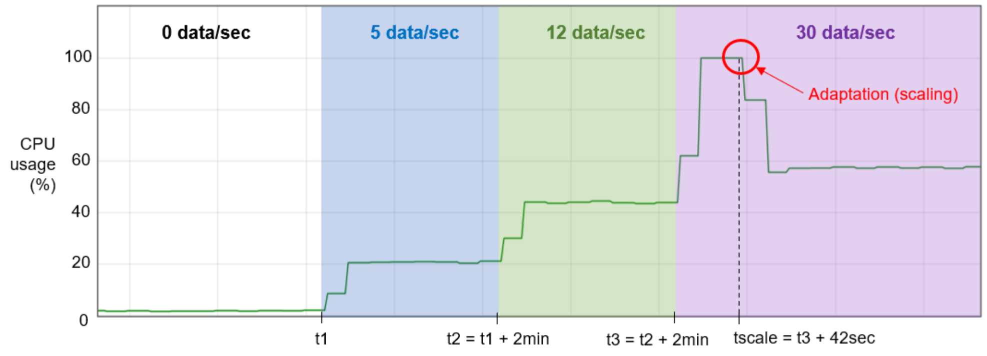
Figure 7.13 presents the CPU usage of the *edge-1* node by increasing the sampling frequency of the sensors. The colored shaded areas represent different sampling frequencies of the sensors. The blue shading indicates that the sensors publish data to the broker at a frequency of 5 data/sec, the green shading 12 data/sec, and the purple shading 30 data/sec. 30 data/sec induces 100% CPU usage on the node. This is dependent on the type of applications implemented and the characteristics of the node. In these tests, we deployed applications that generate a high workload on the most limited AWS instances.

For both cases (non-adaptive and self-adaptive), it was evidenced how the CPU consumption of the node increased when the amount of data to be processed increased too. However, when the CPU usage grew to 100%, the *edge-1* node failed at *tfail* time (Figure 7.13(a)) for the case of not using adaptations, while implementing the adaptation rule the system auto-scaled the *realtime-app* application (at *tscale* time), the workload was reduced for the *edge-1* node (7.13(b)), preventing it from failing.

Similarly, Figure 7.14 shows the time spent by the C2 container to process the data published in the broker by the sensors. For a non-adaptive system (Figure 7.14(a)) the processing time for some data was reached to grow up to 13.5 sec until the node failed due to work overload. On the other hand, the self-adaptive system (Figure 7.14(b)) reached processing times of 8.8 sec, then the system auto-scaled the *realtime-app* application at *tscale* time and the data processing time dropped back below 1 sec.

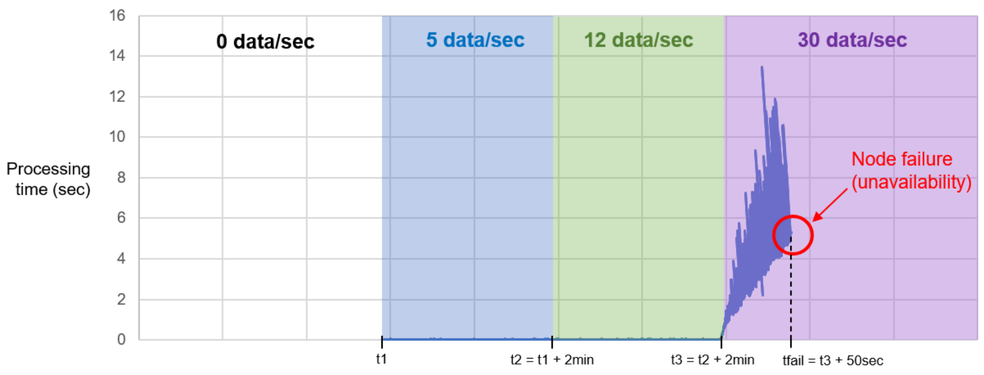


(a) *edge-1* node CPU usage (non-adaptive system)

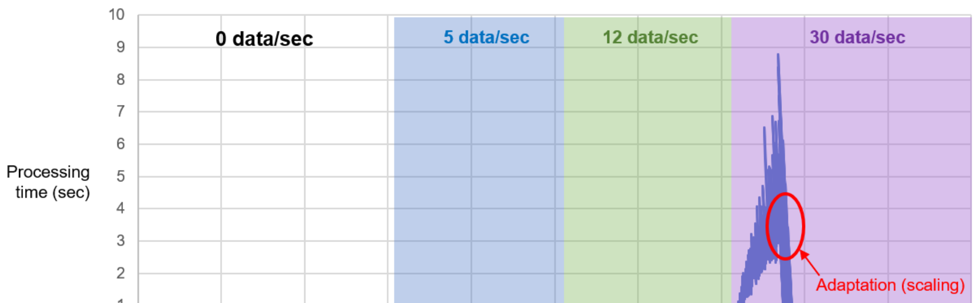


(b) *edge-1* node CPU usage (self-adaptive system; scaling the *realtime-app*) application

Figure 7.13: *edge-1* node CPU usage; scaling adaptation



(a) Data processing time (non-adaptive system)



Offloading a container

The CPU consumption of the *edge-1* node is shown in Figure 7.15. As in the results of the *Scaling* adaptation fitting, the colored shades in the figure represent different data sending frequencies from the sensors. Note that there is a constant CPU usage (43% approx.) before increasing the sampling frequency of the sensors. This CPU usage is caused by *C4* container that simulates the predictive algorithm. As the sampling frequency increased, the CPU consumption of the node also increased. For the non-adaptive system (Figure 7.15(a)) the node overloaded (CPU usage reached 100%) and failed 26 seconds (*tfail*) after increasing the sampling rate from 12 to 20 data per second. In the case of the self-adaptive system, the CPU consumption of the node did not reach 100% due to the adaptation. The *C4* container was offloaded to the *fog-1* node at *toffload* time reducing the workload on the *edge-1* node, preventing it from failing.

Figure 7.16 shows the processing time of the *C2* container data. This processing time increased considerably when the node used 100% of CPU as is the case of the non-adaptive system (Figure 7.16(b)), in which the processing time of some data reached 17 seconds before the node failed. On the other hand, the self-adaptive system experience processing times of less than 0.3 seconds. This behavior because the *edge-1* node never reached high CPU consumption.

Redeploying a container

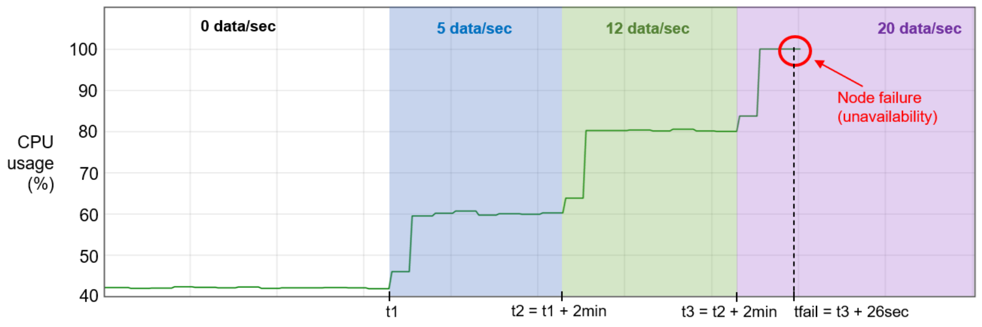
For this Redeployment test, the dependent variable is the availability of the container to be adapted (i.e. to be redeployed). Therefore, we present and analyze the results about the availability of the container.

Figure 7.17 shows the state (available or unavailable) of the *C4* container for both cases: non-adaptive and self-adaptive systems. In the case of the non-adaptive system, the container is unavailable once its failure has been induced at time *t1*. On the contrary case, the self-adaptive system detects that the container is unavailable for 20 seconds and starts the redeployment process. It took approximately 35 seconds to remove the C4 container and redeploy it. After this procedure, the container changed its status to available again.

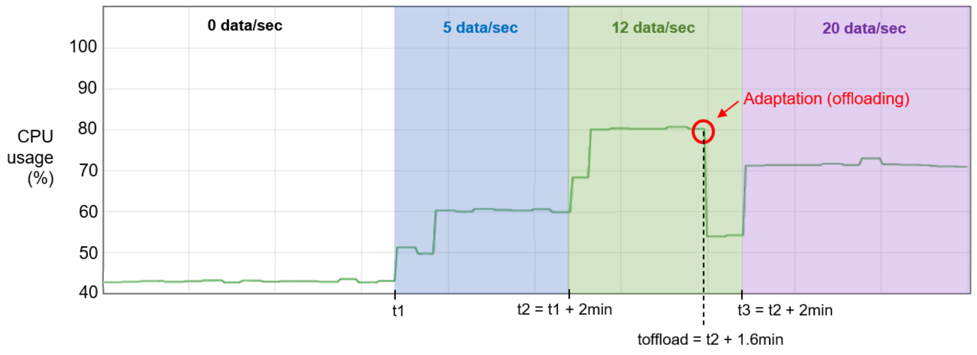
7.2.4 Threats to Validity

The threats identified in this experiment and how we addressed them are presented below.

Random irrelevancies in the setting: some factors outside the experiment (such as network instability or unexpected node failures) could disturb the outcome. In this experiment, we tried to guarantee the stability of the infrastructure using



(a) *edge-1* node CPU usage (non-adaptive system)



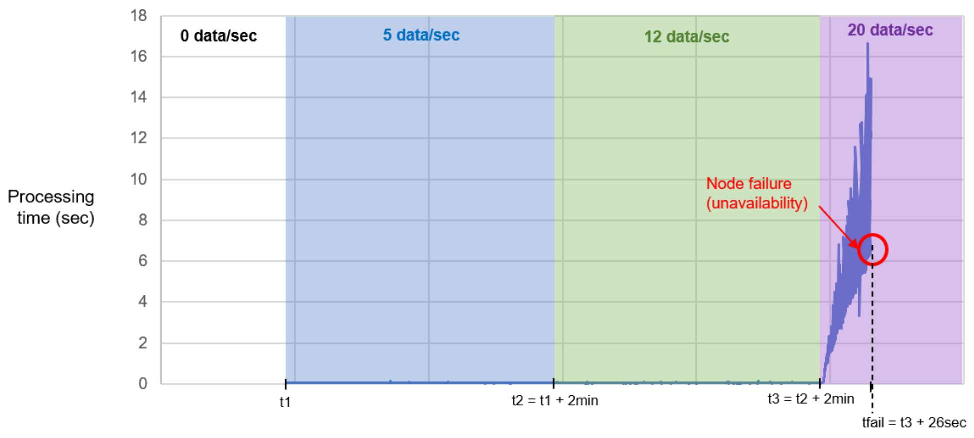
(b) *edge-1* node CPU usage (self-adaptive system; offloading *C4* container)

Figure 7.15: *edge-1* node CPU usage; offloading adaptation

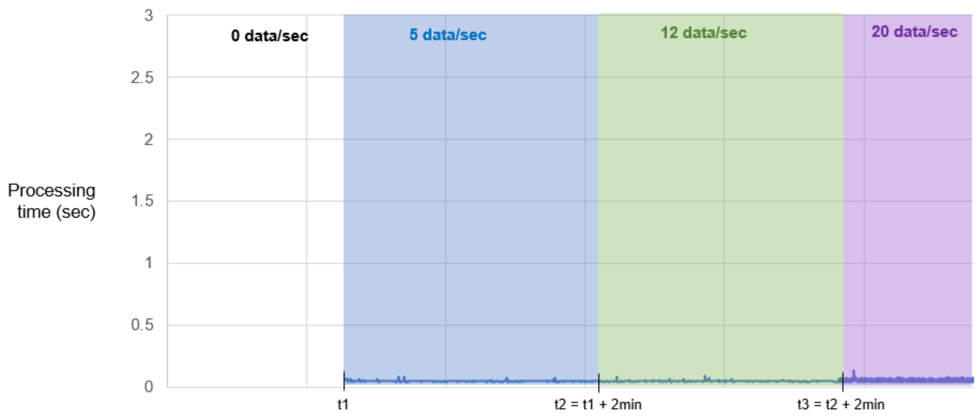
AWS cloud services, which guarantee high availability. Additionally, to avoid disturbances in the delivery of the data sent by sensors, we ran the scripts (simulating the sensors) on an EC2 instance of AWS deployed in the same virtual private cloud as the nodes. This way we guarantee that the data generated in the device layer will be published in the broker on a regular basis and with low latency.

Measurements of dependent variables should be reliable: to ensure the reliability of the measurement of dependent variables (e.g., latency and availability), we have run each experiment at least three times and obtained very similar results. To collect these variables data, we used the monitors and exporters deployed by the framework and consulted the information in the Prometheus time series database.

Mono-operation bias: the study should include the analysis of more than one dependent variable. In our experiments, we analyzed various QoS metrics and in-



(a) Data processing time (non-adaptive system)



(b) Data processing time (self-adaptive system; offloading *C4* container)

Figure 7.16: Processing time data of *C2* container; offloading adaptation

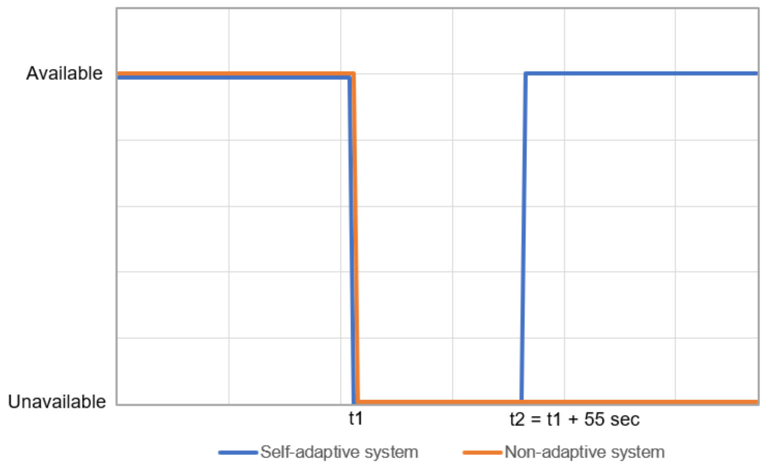


Figure 7.17: Availability of the *C4* container; deployment adaptation

frastructure of nodes and containers. For example, we collected and displayed CPU utilization, availability, and data processing latency. Additionally, for the *Redeploying* a container adaptation experiment, in addition to analyzing the availability metric, we also capture the unavailability time of the container while it is redeployed.

Size of the test scenario: one of the threats is related to the size of the IoT system to be modeled and tested. Since the objective of this validation was to test the architectural adaptations individually, we proposed a small scenario composed of an IoT system with two edge nodes and one fog node. This scenario was sufficient to model an adaptation rule that allowed testing each adaptation. Nevertheless, we have planned a large scenario (as presented in Section 7.3) to validate the scalability of our approach and the execution of concurrent adaptations.

7.3 Evaluation of Framework Scalability

To evaluate the ability and performance of our framework to address the growth of concurrent adaptations (i.e. increase in the number of triggered rules and actions to be performed) on an IoT system, we have conducted two experiments: the first one that triggers simultaneous rules composed of a single action, and the second one that triggers simultaneous rules composed of a group of actions. **The goal of this experiment study** is to identify approach scalability to perform concurrent adaptations of our MAPE-K based framework to adapt the IoT system at runtime. The design, protocol, and results of the experiment are presented below.

7.3.1 Design and Setup

To test the scalability of our framework, we have designed a test scenario emulating an IoT environmental monitoring system in three underground coal mines (Figure 7.18). For simplicity, we assume that the three mines have the same structure (tunnels, work fronts, etc.) and the same monitoring system (nodes and applications). Each mine has ten work fronts [10] constantly monitored to ensure the safety of the workers. The IoT system architecture is composed as follows.

- The **device layer** is composed of several types of sensors deployed at different work fronts. Each work front contains sensors to monitor methane (CH4), carbon dioxide (CO2), carbon monoxide (CO), Hydrogen sulfide (H2S), Sulfur dioxide (SO2), nitric oxide (NO), nitrogen dioxide (NO2), temperature, and air velocity. The information collected by the sensors is sent to a messaging broker (deployed on *edge-1* for the *Mine 1*).

- The **edge and fog layer** nodes run different applications to detect emergencies, control actuators, store information locally and aggregate information to be sent to the cloud nodes.

- The **cloud layer** web application to visualize incident reports and query aggregated historical data on the environmental status of mines. A database is also deployed on one of the cloud nodes to store aggregated data.

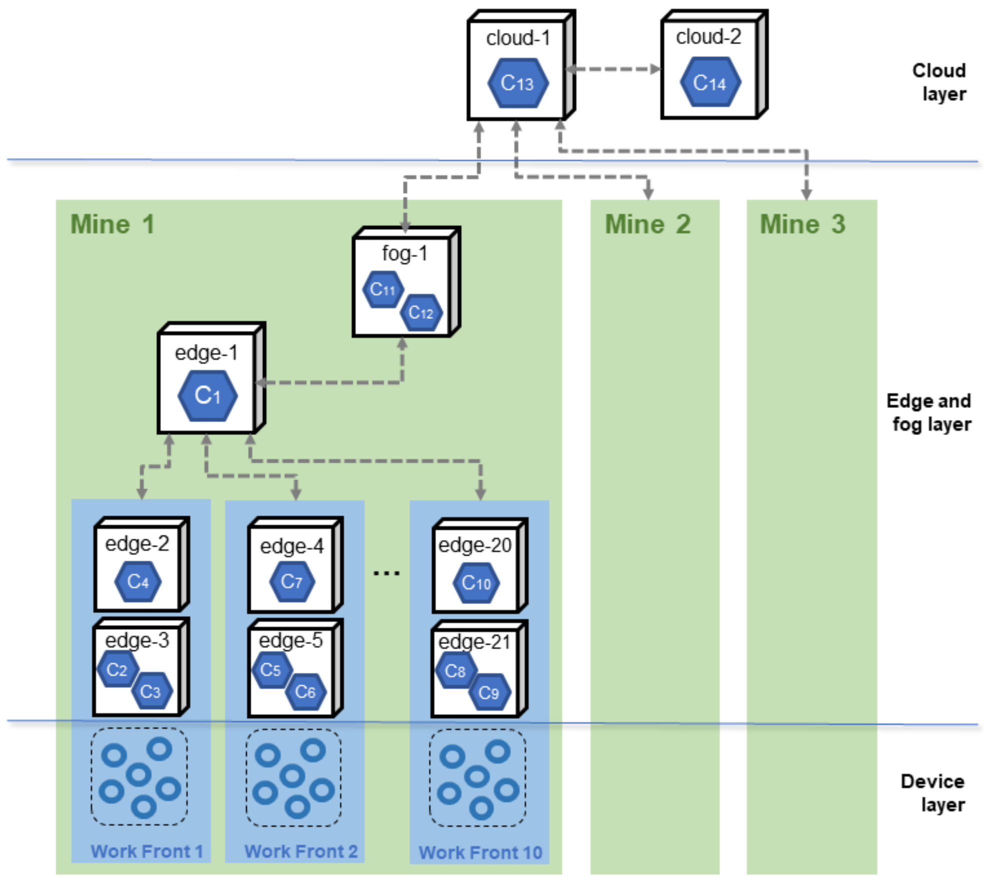
To set up the test environment, we provisioned EC2 instances from AWS. Instances t2.micro (1 vCPU and 1 GB of memory) for edge nodes, instances t2.small (1 vCPU and 2 GB of memory) for fog nodes, and instances t2.medium (2 vCPU and 4 GB of memory) for cloud nodes. The collection and sending of data from the sensors was simulated using a script written in Python language.

As indicated in the legend of Figure 7.18, the containerized applications deployed on the nodes include: an MQTT broker that receives and distributes all sensor data; *stel-app* and *twa-app* check that the values of the monitored gases do not exceed their allowable STEL and TWA values; *temp-app* checks that the temperature at the different work fronts does not exceed the allowable limit value (depending on wind speed); *local-app* is a local application for real-time querying of sensor data; *local-database* stores the data locally before being aggregated and sent to the cloud; finally, *web-app* and *cloud-database* enable to store and query the aggregated data in the cloud. In this experiment, the development of all functional requirements of these applications is out of scope. Instead, we have developed applications with

[10] A large underground coal mine in Colombia could have around ten active work fronts, either mining, advancement, or development.

the basic functionalities including the container images. For example, the *twa-app* application we developed subscribes to the broker to receive the sensor data and performs a data analysis, but does not calculate the actual TWA value.

This IoT scenario for underground coal mining is intended to be close to a real implementation following the rules established in the Colombian mining regulations [132]. For example, the STEL and TWA values are suggested by this regulation.



Application	Container
broker (MQTT Broker)	C1
*stel-app (realtime gas data analysis)	C2, C5, C8
**twa-app (gas limit value 8h)	C3, C6, C9
temp-app (temperature analysis)	C4, C7, C10
local-app	C11
local-database	C12
web-app	C13
cloud-database	C14

⬢	Container
□	Node or server
○	Sensor/Actuator
←--→	Communication

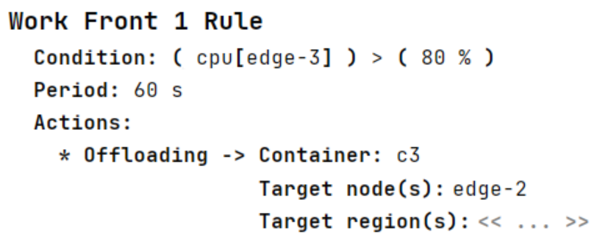
* STEL value is the permissible limit value for a short exposure time (max. 15 min.)
** TWA value is the permissible limit value for an average time of 8 hours

Figure 7.18: Large mining IoT system to be modeled

Experiment 1

In Experiment 1, we have tested the activation and execution of simultaneous rules composed of a single action. We have defined and triggered a rule for 1, 5, 10, 20, and 30 work fronts (five independent tests).

Figure 7.19 shows the rule for *Work Front 1* (a similar rule was specified for each work front): if the CPU consumption of node *edge-3* exceeds 80% for 60 seconds, then offloading the container *c3* to node *edge-2*. In this experiment we have chosen the offloading action, which implies more effort for the *Adaptation Engine*, since it involves the removal and creation of a container on a different node.



```
Work Front 1 Rule
  Condition: ( cpu[edge-3] ) > ( 80 % )
  Period: 60 s
  Actions:
    * Offloading -> Container: c3
                    Target node(s): edge-2
                    Target region(s): << ... >>
```

Figure 7.19: Rule for *Work front 1* - single action

Experiment 2

In Experiment 2, we have defined rules involving several actions. Similar to Experiment 1, we have defined rules for 1, 5, 10 and 20 work fronts (4 independent tests). Figure 7.20 shows the rule for *Work Front 1* (for the other work fronts, the rules are similar), whose list of actions includes one offloading, three scaling, and two redeployment. Note that each scaling action is intended to deploy three instances of the application into any node in the mine.

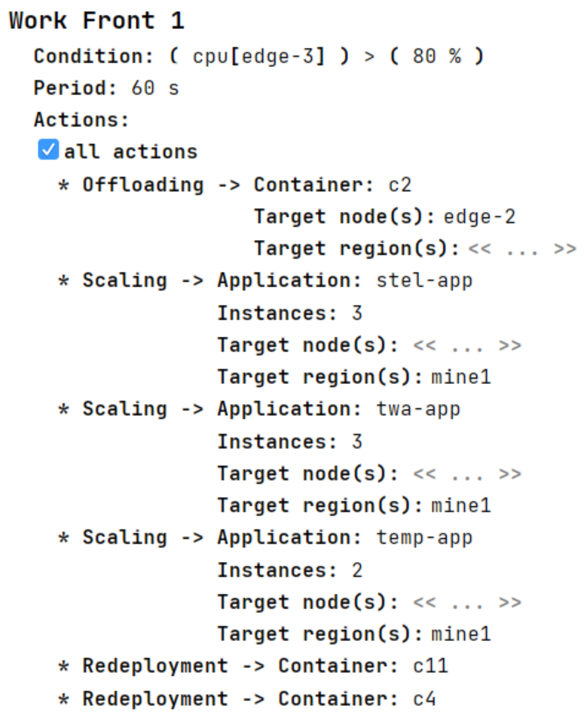

```
Work Front 1
  Condition: ( cpu[edge-3] ) > ( 80 % )
  Period: 60 s
  Actions:
  ✓all actions
    * Offloading -> Container: c2
                    Target node(s): edge-2
                    Target region(s): << ... >>
    * Scaling -> Application: stel-app
                 Instances: 3
                 Target node(s): << ... >>
                 Target region(s): mine1
    * Scaling -> Application: twa-app
                 Instances: 3
                 Target node(s): << ... >>
                 Target region(s): mine1
    * Scaling -> Application: temp-app
                 Instances: 2
                 Target node(s): << ... >>
                 Target region(s): mine1
    * Redeployment -> Container: c11
    * Redeployment -> Container: c4
```

Figure 7.20: Rule for *Work front 1* - multiple actions

7.3.2 Experiment Protocol

Both Experiment 1 and Experiment 2 follow the same protocol, consisting of the following steps.

1. Model the IoT system (using our DSL) including the rules to be tested. The model built for these experiments can be consulted in Appendix A.
2. Run the code generator using the model built in the first step.
3. Deploy the IoT applications and execute our runtime framework using the YAML manifests built by the code generator.
4. Execute the Python script that simulates the generation of sensor data and publishes the messages to the broker. In this way, the necessary workload is generated on the nodes for the rules to be fired.

The independent variable in both experiments is the frequency of data generation. We set the necessary data frequency (90 samples per minute for each sensor) to trigger the rules. As for the dependent variables, in Experiment 1 we focused on monitoring the time spent by the system to detect the event, to generate the adaptation plan, and to adapt the system. In Experiment 2, in addition to monitoring the time spent on adaptation, we also focused on analyzing if there is any error that is raised when performing adaptations and the reasons for them.

7.3.3 Results and Analysis

Experiment 1

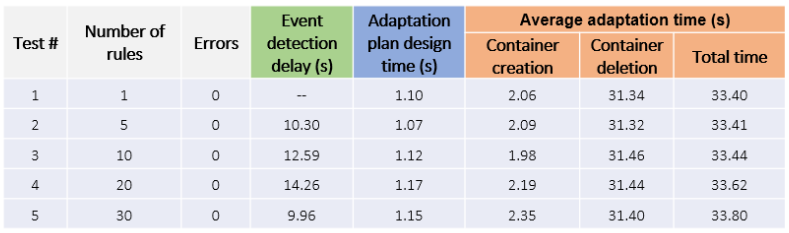
Figure 7.21 shows the results of the tests performed in this experiment. The information in the table includes:

- the number of rules configured in the test;
- the number of errors or failed adaptations;
- the event detection delay refers to the time the system (Prometheus Alerting Rules) takes from the detection of the first rule, to the last one (e.g., for test # 2, the system takes 10,30 seconds from the detection of the event of rule 1 to the detection of the event of rule 5);
- the time the system (Prometheus Alert Manager) takes to generate the adaptation plan and send it to the Adaptation Engine;
- and finally, the time the system (Adaptation Engine) takes to perform the Offloading adaptation, which involves the creation of a new container and the deletion of the old one.

The time spent in the monitoring stage to collect QoS and infrastructure metrics has not been monitored because this is a configurable fixed value for Prometheus. For these experiments, we have set the Prometheus monitoring frequency equal to 4 times every minute (i.e., monitoring every 15 seconds), and the rule evaluation frequency equal to 6 times per minute (i.e., evaluation every 10 seconds). These values were adequate to not generate significant workload on the edge nodes (EC2 t2.micro instances with limited resources).

Findings from the results of this experiment are presented below.

- Ideally the event detection delay (column 4 in Table of Figure 7.21) should be equal to zero, i.e., all events should be detected at the same time since the increase in CPU consumption was caused at the same time in all nodes. However, there are delays between 10 and 15 seconds approximately, due to



Test #	Number of rules	Errors	Event detection delay (s)	Adaptation plan design time (s)	Average adaptation time (s)		
					Container creation	Container deletion	Total time
1	1	0	--	1.10	2.06	31.34	33.40
2	5	0	10.30	1.07	2.09	31.32	33.41
3	10	0	12.59	1.12	1.98	31.46	33.44
4	20	0	14.26	1.17	2.19	31.44	33.62
5	30	0	9.96	1.15	2.35	31.40	33.80

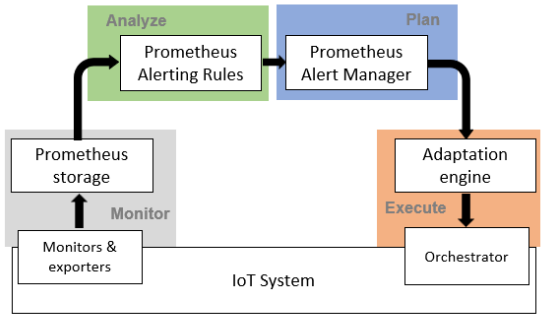


Figure 7.21: Experiment 1 results

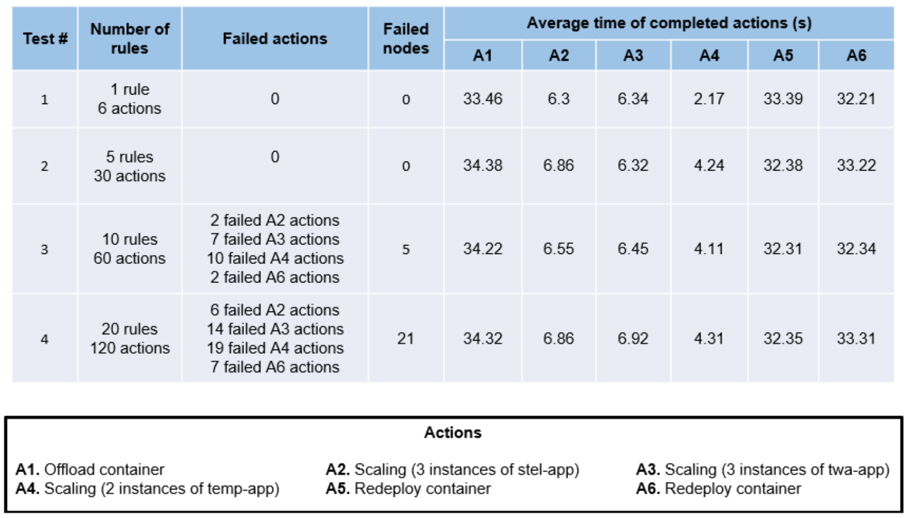
the parameters configured in Prometheus ($monitoring\_interval = 15s$ and $evaluation\_interval = 10s$). Because Prometheus does not synchronize the monitoring and evaluation times, this delay could take a random value between zero (if monitoring and evaluation happen at the same time as the event) and 25 seconds ($monitored\_interval + evaluation\_interval$). This is the reason why test # 4 (with 30 triggered rules) has a lower delay (9.96 seconds) compared to the other tests involving less number of triggered rules. In sum, if the monitoring and evaluation frequencies are increased, the event detection delays could be reduced. However, increasing these frequencies would produce significant workload on resource-poor nodes.

- In all cases (even configuring 30 rules), all actions (offloading) were performed successfully. Approximately one second is required for Prometheus Alert Manager to process an alert, generate the adaptation plan, and send it to the Adaptation Engine. The adaptation time depends on the type of action: the Adaptation Engine, via the K3S orchestrator, takes approximately 2 seconds to create a pod (which hosts a container) and 31 seconds to delete a pod. In this experiment, MAPE-K components did not fail. In Experiment 2 we subjected the framework to more exhaustive tests increasing the number of adaptation executes (details can be found in Section 7.3.3).

- The average time taken by the adaptation engine to perform a container offload is about 33 seconds, with the removal of the container being the most time consuming task (about 31 seconds). This time is due to the grace period (default 30 seconds) that K3S uses to perform the deletion of a pod. When K3S receives the command or API call to terminate a pod, it immediately changes its status to "Terminating" and stops sending traffic to the pod. When the grace period expires, all processes within the pod are killed and the pod is removed. Although our DSL does not currently support grace period configuration, we plan to include the specification of this parameter to ensure safe termination of containers for adaptations that require it (such as offloading or redeployment actions).

Experiment 2

In Experiment 2, we set up rules composed of several actions (summarized in the legend of Figure 7.22) and ran multiple tests by increasing the number of configured rules. The results obtained are presented in Figure 7.22, including the number of tested rules and actions, the number of failed or unsuccessful actions, the number of nodes that failed (some tests produced high memory and cpu pressure, inducing failed nodes), and the average time taken by the Adaptation Engine to perform the successful actions.



Test #	Number of rules	Failed actions	Failed nodes	Average time of completed actions (s)					
				A1	A2	A3	A4	A5	A6
1	1 rule 6 actions	0	0	33.46	6.3	6.34	2.17	33.39	32.21
2	5 rules 30 actions	0	0	34.38	6.86	6.32	4.24	32.38	33.22
3	10 rules 60 actions	2 failed A2 actions 7 failed A3 actions 10 failed A4 actions 2 failed A6 actions	5	34.22	6.55	6.45	4.11	32.31	32.34
4	20 rules 120 actions	6 failed A2 actions 14 failed A3 actions 19 failed A4 actions 7 failed A6 actions	21	34.32	6.86	6.92	4.31	32.35	33.31

Actions

A1. Offload container	**A2.** Scaling (3 instances of stel-app)	**A3.** Scaling (3 instances of twa-app)
A4. Scaling (2 instances of temp-app)	**A5.** Redeploy container	**A6.** Redeploy container

Figure 7.22: Experiment 2 results

Findings from the results of this experiment are presented below.

- For tests 1 and 2 all actions were performed successfully. However, tests 3 and 4 presented failed actions (i.e. adaptations that could not be completed by the *Adaptation Engine* component, mainly Scaling type actions (A2, A3, and A4). These scaling actions were not completed, because there were no more resources available on any of the mine nodes to deploy the new container instances. Even some edge nodes (5 for test 3 and 21 for test 4) failed due to work overload causing that some of the A6 actions were also not completed successfully. These results demonstrate that the successful implementation of the adaptations strongly relies on the availability of resources of the target nodes of the actions. In this sense, one of the improvements for our DSL could be to generate warnings to the user when insufficient resources are detected to perform the modeled rules. In this way we could prevent the implementation of infeasible rules that could fail due to lack of resources.

- The nodes that failed during tests 3 and 4 showed high CPU consumption due to the number of containers assigned to them. Whenever a new pod is deployed on a cluster of nodes (e.g., on one of the edge nodes in mine1), the Kubernetes Scheduler[11] becomes responsible for finding the best node for that pod to run on. Although the Scheduler checks the node resources, it does not analyze the real-time consumption of CPU and Ram memory. Therefore, for Scaling actions in tests 3 and 4, the Scheduler assigned containers to nodes (without checking their current state) causing them to fail. The design and implementation of a Scheduler that analyzes real-time metrics (such as CPU consumption) could avoid this kind of errors.

- The types of actions that require more time to be performed are Offloading and Redeployment. This is because these actions involve the removal of a pod, a task that takes about 31 seconds due to the default grace period set by the K3S orchestrator. On the other hand, the average time it takes to perform the Scaling action depends on the number of instances to be deployed. The Adaptation Engine takes about six seconds to scale three pods or containers, while it takes about four seconds to scale two pods or containers.

7.3.4 Threats to Validity

The threat *random irrelevances in the setting* was addressed in the same way as we discussed in Section 7.2.4. Other threats identified in this experiment and how we addressed them are presented below.

[11]https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/

Measurements of dependent variables should be reliable: to ensure the reliability of the measurement of the dependent variables (e.g., time consumed adapting the system), we performed each experiment at least three times and obtained very similar results. To obtain these metrics, we generated and reviewed log files that record the time and result of each of the tasks performed by the framework components. For example, a log file records the time at which the Adaptation Engine receives the adaptation plan, the time at which each action finishes (including the K3S API command responses), and the result.

Mono-operation bias: To avoid this threat, we have planned two experiments involving several tests. Each experiment contains different adaptation rules to analyze constraints on the performance of concurrent adaptations. We increased the number of configured rules and collected more than one dependent variable to analyze the system behavior.

Size of the test scenario: The number of nodes (edge, fog, and cloud), sensors, and the size of the mine structure was one of the threats we had to validate. For this, we proposed a scenario with underground coal mines containing 10 work fronts, something usual in medium and large scale mining in the department of Boyacá, Colombia (region that supports part of the doctoral thesis). The simulated sensors covered the generation of data for the variables required by Colombian mining regulations (seven types of gases, temperature, and wind speed). The applications performed the operations that are also required by the regulation. Finally, we provisioned 68 AWS EC2 instances (63 edge nodes, 3 fog nodes, and 2 cloud nodes).

7.4 Conclusion

In this chapter, we present the empirical evaluations we performed to validate our approach: (1) experiments to validate the usability and expressibility of the DSL, (2) experiments to validate the functionality of the three types of architectural adaptations, and (3) experiments to identify scalability limitations and boundaries to perform concurrent adaptations of our MAPE-K based framework.

The validation of usability and expressiveness of the DSL is divided into two experiments. The first experiment was conducted with computer science participants (doctoral students and postdoctoral researchers) to evaluate the modeling of architectural aspects of the IoT system, container deployment, and architectural adaptation rules. The second experiment was conducted with participants from the mining area to evaluate the modeling of concepts (in the extended DSL for mining) such as mine structure, control points, sensors, actuators and functional rules (e.g. triggering of alarms due to toxic gas detection). The participants found the DSL useful, sufficiently expressive, and easy to use. Although they were not familiar with the MPS prior to the experiment, most participants reported that the learning curve is low. The lower error rate demonstrates the ease of use of the DSL, even for users who are not experts in Mining, IoT, MPS, or other modeling tools.

We designed three experiments to functionally validate the architectural adaptations and compare the availability and performance of a non-adaptive IoT system with that of a self-adaptive IoT system that is modeled and managed using our approach. The protocol of these experiments includes the modeling of the self-adaptive IoT system, code generation, deployment and self-adaptation of the system. The results of these experiments show that the framework is functionally enabled to execute the modeled architectural adaptations for an IoT system using our DSL. Additionally, the results show that these runtime adaptations can favor and maintain desirable values for IoT system performance and availability.

Finally, experiments to evaluate the scalability of the framework revealed some important limitations and considerations. First, the frequency of monitoring (infrastructure and QoS) and rule evaluation are factors that can delay the detection of events. Setting these frequencies to high values (e.g., 1 check per second) would allow Prometheus to identify events in very short times. However, high monitoring frequencies can generate considerable overheads inducing failures in nodes with low resources (e.g., some edge nodes). Second, although the Scheduler performs a filtering process to select the appropriate node when a new container is deployed, this component of the orchestrator does not analyze metrics of the current CPU and Ram memory consumption of the nodes. This can lead to node failures when the Scheduler assigns pods to nodes that are being overloaded. One strategy to address this concern is to design a Scheduler component that analyzes additional metrics (such as current CPU, memory, bandwidth, and power consumption) to select the appropriate node for deployment tasks.

Chapter 8

Related Work

Self-adaptive systems have been studied for several decades. Wong et. al. [167] classify the evolution of self-adaptive systems into stages. In the first stage (1990-2002), a theoretical model of self-adaptive systems was proposed and the first studies on evolution, self-supervision, control theory, and run-time design emerged [36, 19]. The second stage (2003-2005) was dominated by studies proposing novel perspectives but without concrete implementations. In the third stage (2006-2010), research was focused on autonomous and self-adaptive web services. Runtime solutions predominated over design time solutions. For example, modesl@runtime approach [25] was introduced (the use of software models for adaptive mechanisms to manage complexity in runtime environments). The last stage (2011-2022) shows a transition between the domains of research interest. The adaptability of IoT systems and Infrastructure as a Code (IaaS) becomes the focus. However, the exponential increase and variability of IoT devices, and the unpredictable behavior of the environment introduces self-adaptation challenges to maintain quality levels.

In this chapter, we analyze and compare the studies published to date that are related to our research topic. In particular, languages for specifying IoT systems are analyzed in Section 8.1 and frameworks for supporting system adaptability at runtime are studied in Section 8.2.

8.1 Languages and metamodels for modeling IoT systems

Although there is currently no globally accepted metamodel for specifying the architecture and adaptability of IoT systems, languages for this purpose must be sufficiently expressive to represent the domain. Some approaches [112, 99, 58, 174] use generic languages such as Unified Modeling Language (UML), Finite-state machine

(FSM), Queuing Network (QNs), and YAML to model aspects of the IoT system such as its architecture, software deployment, or self-adaptive capabilities. However, to model the complexity of self-adaptive multilayer architectures, it is necessary to define DSLs that allow representing the entire domain. We have classified the literature studies into two groups: DSLs for modeling the IoT system architecture, and DSLs that address the specification of self-adaptive capabilities. Some studies belong to both groups.

8.1.1 DSLs for IoT Architectures

The **modeling of cloud architectures** is one domain that has been extensively studied. DSLs are proposed in [138, 149, 24, 11] to provision infrastructure, develop and deploy applications for different cloud providers. These studies aim to automate and improve continuous deployment processes through the design of high-level models using textual or graphical notations. Nevertheless, these proposals do not cover multi-layered architectures involving fog nodes, edge nodes, or IoT devices. This is also the case for Infrastructure-as-code (IaC) tools such as Terraform, Chef, Ansible, or Puppet.

Given the complexity of the IoT domain, MDE and DSLs have also been exploited to support the specification of several aspects of an IoT system.

Some **IoT DSLs have been focused on reducing application development and deployment** of IoT applications at nodes and end-devices. For example, Gomez et al. [72] presents EL4IoT, a DSL to model the software system and the operating system configuration in order to generate code for low-end devices running the Contiki-OS[1]. EL4IoT generates C code and XML configuration files for the end device. Similarly, Pramudianto et al. [130] proposes IoTLink, a model driven tool to support the connection tasks between IoT devices and virtual objects (such as databases or REST services). IoTLink generates Java executable code that, when executed, can act as proxies for the physical devices involved in the system. Another DSL focused on application development is presented in [67]. This graphical DSL called MOCSL aims to support the development of native applications for interconnected smart objects. MOCSL offers a graphical editor to specify the sensors and actuators used on a smartphone or an Arduino board. A native application is generated (source code for smartphone or c code for Arduino) to collect sensor data or manipulate actuators. However, EL4IoT, IoTLink, and MOCLS are solutions focused primarily on supporting device layer tasks, without addressing other layers and system adaptability.

[1]https://www.contiki-ng.org/

Other studies propose **DSLs focused exclusively on modeling IoT architectures** such as a [58, 43, 136, 55] discussed below.

A DSL is presented in [58] for modeling IoT systems through a stereotype-based UML profile diagrams. An IoT system is made up of Things, either virtual (e.g., software) or real (e.g., a node). A Thing can contain a collection of items such as inputs (sensors), outputs (actuators), and software components. The DSL enables the specification of methods inside a UML Class block that represents actions at the device layer (e.g., opening a window). However, this DSL requires more specific stereotypes that allow specifying the concept of a multilayer architecture such as edge and fog nodes. Similarly, Costa et al. [43] propose SySML4IoT, a SysML profile for modeling IoT applications. In SySML4IoT a system is composed of *Devices* (sensor, actuator, or tag) and *Services.* The concept *Area* is used to specify the physical location that is affected by the service (e.g. room, building, or floor). However, SySML4IoT is focused on modeling the physical layer without studying the specification of architectural concepts of other layers such as servers and their specifications, or communication protocols.

Salihbegovic et al. [136] propose DSL-4-IoT, a graphical DSL to represent the structure of IoT systems using hierarchical blocks grouped as systems, subsystems, devices, and virtual or physical channels. Configuration files are generated to run the system using the OpenHAB[2] platform, a framework for managing gateways for smart home applications. Although DSL-4-IoT enables the modeling of the system structure including the definition of edge nodes as the gateway, other aspects such as asynchronous communications and run-time system adaptability are not addressed.

Erazo et al. [55] propose Monitor-IoT, a domain-specific language to facilitate and streamline the design of multi-layer monitoring architectures for IoT systems. Monitor-IoT provides a graphical notation for the specification of multi-layer IoT systems including support for modeling data collection, transportation, processing, and storage processes. Both asynchronous and synchronous communications can be modeled with this DSL, but system self-adaptive capabilities are not addressed.

8.1.2 DSLs for IoT Self-Adaptation

The studies analyzed in this section, in addition to covering the modeling of the IoT system architecture, also address the adaptability of the system in some aspects. For example, Barriga et al. [16] presents SimulateIoT, an approach to design and run IoT simulation environments. SimulateIoT addresses the design of a DSL to model the IoT system and its environment. This DSL includes the specification of IoT devices,

[2]https://www.openhab.org/

nodes in different layers (edge, fog and cloud), asynchronous communication (publish/subscribe), databases, and event processing engines. SimulateIoT also includes the modeling of rules for the generation of notifications by analyzing topic data from sensors/actuators. for instance, a notification can be configured when the temperature collected by a sensor exceeds a threshold. However, infrastructure metrics, QoS monitoring, and architectural adaptations are not supported.

CAPS [116] is a Cyber-Physical Systems (CPS) modeling tool that addresses the specification of software architecture, hardware configuration, and physical space. The software architecture specification allows modeling software components and their behavior based on events and actions. Events are responses to some internal change in the software component (e.g., a timer fired or a message received), and actions are atomic tasks that the component can perform (e.g., starting or stopping a timer, or sending a message to another component). However, the modeling of the system behavior addresses adaptations at the software component level without supporting architectural or device-level adaptations. Furthermore, CAPS does not cover the specification of the concepts of multi-layer architectures.

SMADA-Fog [127] is a model-based approach not exclusively focused on the IoT domain, but it could be used to model self-adaptive IoT systems. SMADA-Fog address the deployment and adaptation of container-based applications in Fog computing scenarios. SAMADA-Fog proposes a metamodel that enables modeling the deployment and adaptation of containers on nodes (edge, fog, and cloud). SMADA-Fog enables the specification of consumer devices (such as laptops, smartphones, and IoT devices) and network devices, but does not address the modeling of sensors and actuators because it is a DSL focused on fog computing applications. SMADA-Fog addresses the specification of rules whose conditionality involves QoS metrics and architectural adaptations such as scaling, optimizing a metric, blocking a service, and creating/shutting down a service. The deployment and adaptation model of the system is specified by means of environments designed in Node-RED[3]. However, rules for operating or controlling the system's actuators are not supported. In addition, the physical regions and location of devices and nodes are not addressed concepts by the metamodel.

8.1.3 Discussion

Table 8.1 presents a comparative analysis between the DSLs studied and our DSL. The comparative information includes: (1) the notation of the language (e.g., textual, graphical, or tabular); (2) whether the language addresses modeling of IoT devices, including sensors and actuators; (3) whether the language addresses specification

[3]Node-RED is a programming tool for wiring together hardware devices, APIs and online services

of physical regions or locations such as rooms, buildings, or tunnels; (4) whether the language addresses modeling of edge, fog, and cloud nodes; (5) whether the language enables the specification of hardware properties of nodes such as memory, CPU, storage, operating system, and other characteristics; (6) the software resources that can be modeled with the language (e.g., databases, software containers, and message brokers); and (7) the types of conditions and actions that make up the rules. The main limitations we identified in the literature for modeling self-adaptive IoT systems are listed below.

- The literature has focused primarily on application development and deployment issues, without addressing the specification of rules or policies that govern the adaptability of the IoT system at runtime. The few DSLs that do address adaptations focus on a single type (e.g., architectural, physical layer, or network adaptations) and do not provide a sufficiently expressive language for creating complex rules. Our DSL addresses the specification of rules for modeling self-adaptation schemes and functional rules for controlling IoT system actuators. Definition of complex composite rules relating infrastructure metrics (such as CPU and RAM usage), QoS (such as availability and latency) and sensor data (such as temperature and oxygen) are covered in our proposal.

- Multi-layer architectures that take advantage of edge and fog computing are becoming increasingly popular. Modeling languages for IoT architectures must address the specification of the concepts that enable the implementation of these technologies. There are only three studies [55, 16, 141] that enable the modeling of sensor/actuator devices, edge, fog and cloud nodes; and one of these does not allow modeling the specifications (CPU, RAM, storage, etc.) of the nodes (an important aspect considering that edge and fog nodes have limited resources). These are aspects that we address in our DSL.

- Specifying the location (e.g. by coordinates or regions) of the devices and nodes that make up the system infrastructure is important for defining rules that involve conditions or adaptations linked to a physical region of the environment. For example, in monitoring systems it is necessary to guarantee the availability of services deployed in critical surveillance zones. This is another aspect that we cover by enabling the modeling of physical regions and location coordinates. On the other hand, a few works such as [130, 43, 16, 116] address the modeling of physical regions or spaces such as buildings, rooms, or corridors. However, none of these enable the definition of rules involving the region.

- None of the analyzed DSLs evaluate their extensibility capability to expand the abstract syntax by including new concepts to the metamodel. This is one of the features supported by our DSL. In Chapter 6 we present the design of two extensions of the DSL to model self-adaptive IoT systems in two different domains.

To summarize, although there is some literature focused on the design of DSLs to define IoT systems at different levels of abstraction, the self-adaptation capabilities for multi-layer IoT systems (including device, edge, fog, and cloud) has not been properly explored. Our DSL is the first proposal that enables the modeling of multi-layer IoT architectures and the definition of complex rules covering all layers (and combinations of) and involving multiple conditions and actions that can, potentially, involve groups of nodes in the same region or cluster of the IoT system.

Reference	Notation	IoT device	Location	Node			Hardware prop.	Software	Rules	
		(sensor/actuator)		Edge	Fog	Cloud	(cpu, ram, memory, ip, OS, etc.)		Conditions	Actions
EL4IoT [72]	Textual	Yes	No	Yes	No	No	No	application and Contiki-OS config.	No	No
IoTLink [130]	Textual	Yes	Yes	No	No	No	No	API, service, database	No	No
Eterovic et al. [58]	Graphical	Yes	No	No	No	No	No	Software components	No	System actuator control
SySML4IoT [43]	Graphical	Yes	Yes	Yes	No	No	No	Services	No	No
Monitor-IoT [55]	Graphical	Yes	No	Yes	Yes	Yes	Yes	App, broker, API, and middleware, database	No	No
MOCSL [67]	Graphical	Yes	No	No	No	No	No	Application	No	No
DSL-4-IoT [136]	Graphical	Yes	No	No	No	Yes	No	Service	No	No
SimulatorIoT [16]	Graphical	Yes	Yes	Yes	Yes	Yes	No	Event processing engine, broker, databases	Sensor data	Notifications
CAPS [116]	Graphical	Yes	Yes	Yes	No	No	Yes	Soft. component	Internal change in the soft. component	Soft. component tasks
SMADA-Fog [127]	Graphical	No	No	Yes	Yes	Yes	Yes	Container	QoS conditions	Architectural
Our DSL	Graphical, textual, tabular	Yes	Yes	Yes	Yes	Yes	Yes	App, broker, container	Sensor data or QoS conditions	Architectural and system actuator control

Table 8.1: Comparative analysis of DSLs and metamodels for IoT system modeling

8.2 Frameworks for IoT system self-adaptations

In this section, we analyze frameworks for handling self-adaptive IoT systems at runtime. We have categorized the analyzed studies into two groups: (1) general frameworks or approaches which use general-purpose languages and (2) model-based frameworks which use DSLs such as those discussed in Section 8.1.1.

8.2.1 General Approaches or Frameworks

The Rainbow framework [68] represents one of the earliest attempts to support self-adaptation of software systems. This framework, based on MAPE-K loop, enables the specification of the system architectural model using the Acme language[4] and the specification of adaptation rules by means of scripts. Although Rainbow defines the stages of the MAPE-K cycle to perform the continuous feedback of the system state and to generate adaptation plans, the monitors (in charge of monitoring the system) and the effectors or actuators (in charge of applying the system actions) must be developed and provided by the user. This is why Rainbow does not offer a fixed list of adaptations, since they depend on the monitors and effectors that the system provide.

Moghaddam et al. [112] propose IAS, an IoT architectural self-adaptation framework. IAS is based on Queuing Networks (QNs) for modeling architectural patterns that the system adopts to improve non-functional attributes. Three architectural patterns can be modeled using IAS and QNs: *centralized* comprises processing on a central local or remote controller, *distributed* includes processing on independent or collaborative controllers, and *hierarchical* contains independent or hybrid controllers (i.e., with distributed collaboration). The controllers are based on MAPE-K loop which ensure compliance with the functional requirements of the system. Although the adaptations addressed by IAS include modification of the architectural pattern of the system drivers, support for deployment and adaptation of container-based applications is out of scope. In addition, the use of QNs as a modeling language hampers the specification of IoT domain-specific aspects.

Lee et al. [99] present a self-adaptive framework to dynamically satisfy functional requirements for IoT systems. This framework, based on MAPE-K loop, enables modeling of the IoT environment through a finite-state machine, which includes four types of states: *satisfied* to represent satisfied requirements, unsatisfied to represent unsatisfied requirements, adaptive represents states in which an adaptive activity can be performed, and *normal* represents states that do not affect software adaptation. This framework is focused on managing adaptations of the device

[4]Acme is a simple, generic software architecture description language that can be used as a foundation for developing new architectural design and analysis tools

layer of the system by manipulating the actuators (e.g., opening a window when the temperature exceeds a limit). However, architectural rules for the other layers of the system are not supported.

Weyns et al. [165] propose MARTA, an architecture-based adaptation approach to automate the management of IoT systems employing runtime models and leveraging the MAPE-K loop. Each rule is specified by a quality model that stores a condition and one or more adaptations (e.g., packet loss < 10%, minimize energy consumption). Although MARTA addresses the monitoring of network metrics such as latency and packet loss, other infrastructure metrics (such as CPU and node Ram usage) are not collected. In addition, the monitors and effectors that adapt the system are designed for a particular case, making reusability challenging.

A few works such as [174, 143, 85, 140] focus on **system adaptations to optimize the deployment of IoT applications on edge and fog nodes**. These studies propose the use of orchestrators such as Kubernetes and Docker Swarm. For example, Yigitoglu et al. [174] present Foggy, a framework for continuous automated deployment in fog nodes. Foggy enables the definition of four software deployment rules in Fog nodes. Foggy's architecture is based on an orchestration server responsible for monitoring the resources in the nodes and dynamically adapting the software allocation according to the rules defined by the user. However, Foggy is focused on adapting the system exclusively to support the continuous deployment of applications. IoT system adaptations caused by dynamic events other than software deployment failures are not supported.

8.2.2 Modeling-based solutions

Self-adaptive modeling-based systems have also been extensively studied for **cloud-based applications**. Works such as [39, 61, 35, 81, 56] propose partial solutions as they either restrict the parts of the system that could be adapted (e.g. only the global monitoring component to adapt to manual changes in the deployed components) or offer some type of adaptation rules but with limited expressiveness and rules that must be manually triggered instead of being self-adaptive.

A MAPE-K based framework to evaluate and select the architectural pattern that favors quality attributes in different scenarios is porposed by Muccini et al. [117]. In particular, the power consumption of the devices is prioritized and evaluated by implementing architectural patterns such as master/slave and centralized for the components of the MAPE-K cycle. This framework enables system modeling using CAPS [116], a tool for architecting situational-Aware Cyber-Physical systems. However, the CAPS DSL, being focused on modeling cyber-physical systems, has limitations for modeling the multi-layer architectures of IoT systems as mentioned in Section 8.1. Additionally, the specification of rules for operating or controlling the system actuators is not supported.

The following works are more related to our proposal. Hussein et al. [83] present a framework to support the design and operation of self-adaptive IoT systems. The specification of the system requires two models: a model that captures the functionality of the system and a model that describes the adaptation. The functionality is specified using SySML4IoT [43] (a SySML profile for modeling IoT applications), while the adaptations are modeled using a state machine. The adaptations addressed by this framework consist of changing the state (active/inactive) of the system services when an availability failure is detected in sensors or services. For example, when a temperature sensor in a room fails, the temperature control inside the room is deactivated. However, SySML does not address the modeling concepts of multi-tier architectures such as edge and cloud nodes making it difficult to specify and execute architectural adaptations. In addition, infrastructure metrics and sensor data monitoring are not collected.

Petrovic et al. [127] propose SMADA-Fog, a semantic model-driven approach to deployment and adaptation of container-based applications in Fog computing scenarios. The deployment and adaptation model of the system is specified by means of environments designed in Node-RED[5]. SMADA-Fog addresses architectural adaptations to manage containers deployed using Docker-Swarm, an orchestrator to manage Docker containers. To adapt the system, a code generator produces Docker commands and SDN rules that spawn the desired services on target servers and shape the traffic between them. However, SMADA-Fog does not allow the specification of complex rules composed of various conditions and actions. Moreover, grouping nodes and IoT devices according to their location is not possible, forbidding the possibility to apply adaptations on group of nodes belonging to a cluster or a given region.

Discussion

Table 8.2 compares the frameworks analyzed in this section with our proposal. The comparative information includes: (1) the application domain of the framework; (2) the approach used for specification or modeling of the self-adaptive system; (3) the aspects of the system that are modeled such as system architecture, software deployment, or adaptation policies; (4) the actions and adaptation strategies addressed (e.g., architectural adaptations or system actuator control); and (5) the metrics collected to monitor the state of the system and detect events that trigger adaptations. The relevant findings and differences are listed below.

[5]Node-RED is a programming tool for wiring together hardware devices, APIs and online services

- The specification of rules for an IoT system requires a language expressive enough to model and support complex rules composed of various types of expressions and adaptations. Most frameworks use general-purpose languages to model both the system and its adaptivity. For example, Acme in Rainbow [68], QNs in IAS [112], or finite state machine models in Lee et al. [99]. In contrast, our framework uses a DSL specialized in modeling rules (architectural adaptations and functional rules) for the IoT system.

- System state monitoring is one of the key tasks in frameworks that support self-adaptation. There are several types of metrics that can be collected to monitor system state: QoS metrics such as availability, latency, and power consumption; infrastructure metrics such as CPU consumption, Ram memory consumption, and free disk space; and sensor data metrics collected with IoT system sensors such as temperature, humidity, motion, and gas concentration. Most of the frameworks analyzed focus on monitoring only one type of metrics. For example, IAS [112] focuses on performance, MARTA [165] and SMADA-Fog focus on QoS, and Lee et al. [99] focus on sensor data. Our framework uses monitoring tools that collect QoS, infrastructure, and sensor data metrics.

To summarize, our model-based framework is the first to support the specification of architectural adaptations and functional rules, collecting infrastructure, QoS, and sensor data metrics. This framework integrates a DSL for self-adaptive IoT system specification that encompasses architecture modeling, software, and rules (both architectural adaptation and functional rules). In addition to self-adapting the system at runtime, this framework also supports the deployment of container-based applications using orchestrators such as Kubernetes and K3S, technologies that become popular for running applications on edge, fog, and cloud nodes.

Reference	Domain	Modelling approach	Modeled aspects	Actions and adapt. strategies	Metrics collected
Rainbow [68]	General purpose	Acme	Architecture and adaptation rules	Depends on user-supplied effectors	depends on user-supplied monitors
IAS [112]	IoT	Queuing Networks (QNs)	Architectural pattern adaptation	Architectural pattern adaptation	Performance
Lee et al. [99]	IoT	Finite-state machine	Functional rules	System actuator control	Sensor data
MARTA [165]	IoT	Quality models	Functional rules	System actuator control (setting the power and sampling frequency of the devices)	Packet loss, latency, and energy consumption
Muccini et al.	IoT	CAPS	Architertural patterns	Architectural pattern adaptation	Energy consumption
Hussein et al. [83]	IoT	SySML4IoT and finite-state machine	Functionality and adaptability	on/off services	Availability of sensors and services
SMADA-Fog [127]	Fog Comp.	Node-Red	Architecture, application deployment, and adaptation rules	Architectural adaptations	QoS metrics
Foggy [174]	Fog Comp.	YAML	Software Deployment	Allocation strategies	Infrastructure and latency
Our proposal	IoT	DSL	Architecture, application deployment, architectural adapt. and functional rules	Architectural adaptations and system actuator control	Infrastructure, QoS, and sensor data

Table 8.2: Comparative analysis of frameworks to support IoT system self-adaptations

Chapter 9

Conclusions and Further Research

In this chapter, we first synthesize and conclude all the contributions of this thesis (Section 9.1). In Section 9.2.1, we list the publications and software artifacts developed and available. Finally, Section 9.3 presents several perspectives for future research.

9.1 Summary of Contributions

The exponential growth of the Internet of Things (IoT) over the last few decades has revolutionized many areas such as education, healthcare, industry, and even our social relationships. Today, we interact with IoT systems in many daily activities to optimize or improve our quality of life. This growth of IoT has generated new and increasingly restrictive requirements for the system in terms of performance. Because of this, IoT system architectures have evolved by implementing paradigms such as edge and fog computing. Multi-layer architectures that leverage edge and fog computing can improve quality of service (QoS), latency, and bandwidth consumption. However, the design of these complex architectures is challenging, especially when the system must self-adapt at runtime due to the dynamicity of the environment.

IoT systems are exposed to dynamic environments that generate unexpected changes impacting QoS. Commonly systems are not designed to cope with these dynamic events, but architectures with self-adapting capabilities could be defined to address this problem. In this sense, the main contribution of this thesis is a model-based approach to support the design, deployment, and management of self-adaptive IoT systems. This approach is divided into two stages: design time for the specification of the self-adaptive IoT system, and runtime to support the operation and adaptation of the system.

To enable the modeling of the IoT system and generate code for the deployment and self-adaptation of the system, the design time stage is composed of:

- a DSL for IoT systems focusing on three main contributions: (1) modeling primitives covering multi-layer architectures of IoT systems, including concepts such as IoT devices (sensors or actuators), edge, fog and cloud nodes; (2) modeling the deployment and grouping of container-based applications on those nodes; and (3) a specific sublanguage to express architectural adaptation rules (to guarantee QoS at runtime, availability, and performance), and functional rules (to addres fuctional requirements that involve system actuator control). We have implemented this DSL using a projectional-based editor that allows mixing various notations to define the concrete syntax of the language. In this way, the IoT system can be specified by a model containing text, tables, and graphics.

- A code generator. The model (built using the DSL) describing the self-adaptive IoT system is the input to a code generator we have designed. This generator produces several YAML[1] manifests with two purposes: (1) to configure and deploy the IoT system container-based applications and (2) to configure and deploy the tools and technologies used in the framework that supports the execution and adaptation of the system at runtime.

To support the system adaptation at runtime stage of our approach, we have developed a framework to monitor and adapt the IoT system following the adaptation plan specified in the model. This framework is based on the MAPE-K loop composed of several stages including system status monitoring, data analysis, action planning, and execution of adaptations. Our framework deploys exporters to collect infrastructure metrics (such as CPU and Ram usage), QoS (such as availability), and system sensor data. These metrics are stored using Prometheus (a time series database) and queried using PromQL language to verify rules. We have developed an *Adaptation Engine* to perform two types of system actions when necessary: architecture adaptations (such as offloading and scaling apps) and system actuator control (to meet system functional requirements).

We have introduced two extensions to our DSL highlighting the extensibility capability to add new concepts in the abstract syntax. The first extension focuses on modeling IoT systems in the underground mining industry while the second extension focuses on IoT systems implemented in wastewater treatment plants (WWTPs).

[1]YAML is a data serialization language typically used in the design of configuration files

In addition to the metamodel, the projectional editors were also extended to offer new modeling notations for underground mine specification, and for modeling of WWTP process block diagrams.

Finally, to validate our DSL and framework, we have designed and conducted empirical experiments: (1) to validate the expressiveness and usability of our DSL extended to the mining domain (13 participants attended the experiment), (2) to test the self-adaptive capability of our approach (one test scenario for each of the architectural adaptations), and (3) to evaluate the ability and performance of our framework to address the growth of concurrent adaptations. The reported results demonstrate that the DSL is expressive enough to model self-adaptive IoT systems and has a favorable learning curve. Moreover, experiments with the framework validate its functionality and ability to self-adapt the system at runtime.

9.2 Publications and Software Artifacts

This section shows how the work that supports this dissertation has been published in journals and conferences. We also list the software artifacts developed and their repositories.

9.2.1 Publications

Conferences

- D. Prens, I. Alfonso, K. Garcés and J. Guerra-Gomez. Continuous Delivery of Software on IoT Devices. *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, 2019, pp. 734-735. This paper was one of our first steps in defining a metamodel to support the deployment of applications in IoT systems.

- I. Alfonso, "A Software Deployment and Self-adaptation of IoT Systems" *Proceedings of the XXIII Iberoamerican Conference on Software Engineering, CIbSE 2020*, November 9-13, 2020, pp. 630-637. This paper contains the thesis proposal presented at the CIBSE 2020 Doctoral Symposium.

- I. Alfonso, K. Garcés, H. Castro and J. Cabot. Modeling self-adaptative IoT architectures. *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, 2021, pp. 761-766. This paper contains the first version of our DSL and supports part of the content of Chapter 4.

- I. Alfonso, K. Garcés, H. Castro and J. Cabot. Modelado de Sistemas IoT para la Industria en Minería Subterránea de Carbón. XXVI Jornadas de Ingeniería del Software y Bases de Datos (JISBD), 2022. In this paper we present the extension of our DSL for modeling IoT systems in the mining industry domain. This paper supports part of Chapter 6.1.

Journals

- I. Alfonso, K. Garcés, H. Castro and J. Cabot. Self-adaptive architectures in IoT systems: a systematic literature review. *Journal of Internet Services and Applications*, 2021, 12(1), 1-28. This paper support the content presented in Chapter 2.2.
- **Under revision:** I. Alfonso, K. Garcés, H. Castro and J. Cabot. A model-based infrastructure for the specification and runtime execution of self-adaptive IoT architectures. *Computing journal*. Submitted in June 2022. This paper presents the design of the final version of our DSL, the framework to support the runtime self-adaptations, and the empirical experiments to validate the usability of the language. This paper supports part of the chapters 4, 5, and 7.

Award

- Premio al trabajo liderado por estudiante de doctorado del track de Ingeniería del Software Dirigido por Modelos (ISDM) de las XXVI Jornadas de Ingeniería del Software y Bases de Datos (JISBD), por el artículo "Modelado de Sistemas IoT para la Industria en Minería Subterránea de Carbón".

9.2.2 Software Artifacts

The artifacts developed in this thesis include a DSL, two DSL extensions, a code generator, and an adaptation engine.

- Code generator and language workbench for the specification of self-adaptive IoT systems.

  Repository: https://github.com/SOM-Research/selfadaptive-IoT-DSL

- Code generator and language workbench extension for the specification of self-adaptive IoT systems in the Underground Mining Industry.

  Repository: https://github.com/SOM-Research/IoT-Mining-DSL

- Code generator and language workbench extension for the specification of self-adaptive IoT systems in WWTPs.

  Repository: https://github.com/SOM-Research/WWTP-DSL

- Adaptation engine to adapt the system at runtime.

  Repository: https://github.com/ivan-alfonso/adapter-engine.git

  Docker-hub image: https://hub.docker.com/r/ivanalfonso/adaptation-engine

9.3 Further Research

The research conducted in this thesis could be extended into three lines of future research: (1) specific enhancements to our approach, (2) software deployment strategies in IoT systems, and (3) integration of machine learning strategies to suggest rules.

9.3.1 Approach improvements

Although our approach supports the specification, deployment and management of self-adaptive IoT systems, we have identified some limitations and specific improvements that we plan to address in the future as follows.

Mobility of devices

One of the dynamic events addressed that can affect the IoT system is device mobility. When a device changes location, a set of steps are performed: (1) new communication must be established between the device and the suitable edge/fog node; (2) the availability of resources must be guaranteed to deploy the service in the edge/-fog nodes in order to manage that device; and (3) in case the device changes location again, it is evaluated if it must be connected to other edge/fog nodes that are closer to obtain better latency. The mobility of many devices could lead to increased latency, higher resource consumption, and unavailability of system services, as the increased volume of data generated by devices can congest the network and create bottlenecks.

Although our approach enables the configuration of rules to deal with the effects of device mobility, the DSL does not currently address the modeling of mobile devices. We are interested in including the necessary concepts to the metamodel for modeling device mobility, as well as including new metrics to quickly identify this event. For example, monitor the number of clients connected to gateways or edge nodes and create adaptation rules to identify and deal with the increase of connected devices. To achieve this, it would also be necessary to generate code to deploy new monitors with the ability to collect these new metrics and run exporters to translate the information into the Prometheus database format.

Communication protocols support

In order to achieve a high degree of scalability, message-driven asynchronous architectures are typically preferred [73]. In particular, the MQTT protocol is implemented to set up asynchronous (publish/subscribe) communications between IoT devices and higher layer nodes. For this reason, we address the specification of these concepts in our DSL. However, other protocols such as AMQP (Advanced Message Queuing Protocol), CoAP (Constrained Application Protocol), and DDS (Data Distribution Service) are gaining popularity for use in some IoT systems. The choice of protocol depends on the requirements and characteristics of the system. For example, unlike MQTT, CoAP has the capability of automatic discovery of devices, but since it is built on top of UDP, SSL (Secure Sockets Layer) and TLS (Transport Layer Security) are not available for this protocol.

Addressing other communication protocols is another projected future task. This would involve incorporating the concepts for each protocol in our metamodel, defining new editors to enable the configuration of the protocols, and generating the code to deploy new monitors and exporters that collect the information according to the protocol used.

DSL tool support

Regarding the implementation of DSL using MPS, there are two directions we would like to address.

The first activity consists of bringing the IoT system modeling to the web, i.e. providing a web version of our DSL. There are a few tools that could be explored for this purpose. For example, Modelix[2] is an open source platform that aims to allow the editing of models from the browser, for languages created in MPS. Another alternative could be MPSServer[3], a tool to remotely access the project and edit models in MPS framework.

The second activity consists of designing a graphical editor to model the IoT system architecture. For some of the participants of our DSL usability validation it would be more comfortable to model the architecture using graphical notation. This graphical editor should at least include the use of different types of shapes to represent the IoT devices, the nodes (edge, fog, and cloud), the regions, the software containers, and arrows to represent the data flow. MPS currently provides plugins to support graphical modeling. For example, our DSL extension for WWTPs uses the MPS Diagrams plugin to enable process block diagram specification using graphical

[2]https://modelix.github.io/

[3]https://github.com/strumenta/mpsserver

notation. These plugins could be reused to provide graphical notation for system architecture modeling.

AsyncAPI Integration

To achieve high degree of scalability, improved performance and reliability, IoT systems often implement event-driven architectures [74]. One of the most commonly used patterns in this type of architecture is publish/subscribe. One biggest challenge of these architectures is to maintain message consistency. That is, the topics and format of the messages published by the system's sensors must be consistent throughout the life cycle of the system. A slight change in the format of the messages could cause a system failure. The AsyncAPI[4] specification was proposed to address this challenge. This specification allows to represent concepts such as message brokers, topics of interest, and the different message formats associated with each topic. One of the future tasks is to integrate the AsyncAPI specification with our DSL to support different formats in the messages published by the sensors and to address the consistency issues that may currently arise.

9.3.2 Software Deployment Strategies

Managing software deployments and updates in multilayered IoT systems poses challenges due to resource limitations and the heterogeneity of communication, devices, and protocols [31]. Edge and fog nodes have limited resources that induce functional failures if software deployments are not properly configured and planned. Setup of software deployments across edge, fog, and cloud nodes can be a time-consuming and error-inducing task. While our approach supports the deployment of container-based applications using an orchestrator such as Kubernetes, another of our future directions is focused on implementing deployment strategies to enable continuous deployment and reduce the likelihood of failure. The strategies to be investigated are categorized into two groups: deployment patterns y allocation strategies.

Deployment Patterns

Deployment patterns provide control over the deployment of new software versions to reduce the risk of a process failure and increase reliability. Implementing these patterns reduces application downtime in an upgrade process and enables incidents to be managed and resolved with minimal impact to end users [18]. There are three popular patterns for managing deployment: canary, blue-green, and rolling.

[4]https://www.asyncapi.com/

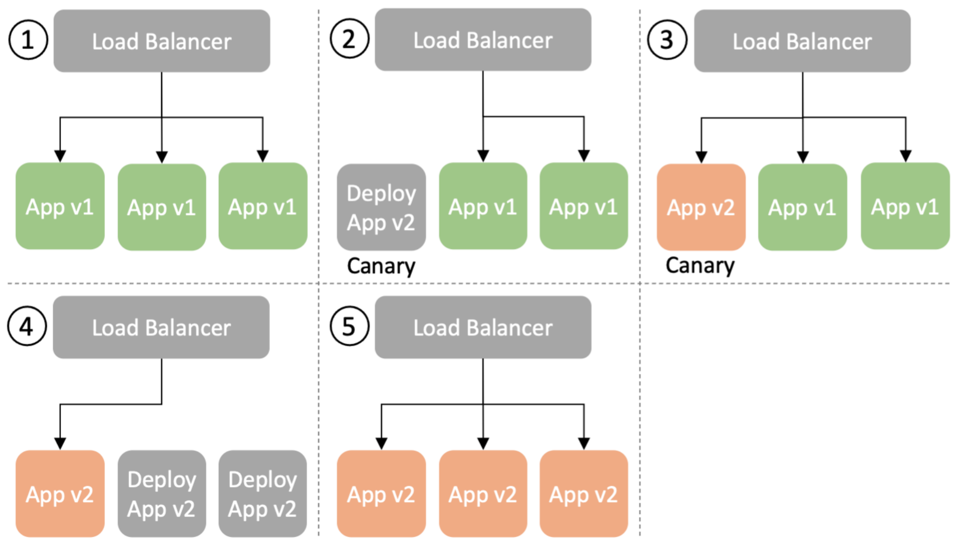


Figure 9.1: Deployment process of the canary pattern

The **canary pattern** suggests deploying the new software version in a subgroup of nodes (known as canary) to evaluate this version before deploying it in all the nodes of the system. Figure 9.1 shows the stages of canary deployment in a cluster of three nodes: (1) the canary node(s) that will host the new software version is chosen (commonly 30% of all nodes); (2) the canary node(s) are deactivated to deploy the new release (App v2); (3) traffic is redirected to the canary node to evaluate functional and non-functional aspects and to determine if the application is stable; (4) if the assessment of the canary was successful and the application runs properly, the new release (App v2) is deployed in the rest of the nodes; (5) finally, the traffic is redirected to all the nodes. On the other hand, if the assessment of the canary detects errors or inconsistencies in the application, the rollback is performed in the canary nodes before deploying the application in the remaining nodes.

The blue-green pattern, also called big flip or red/black deployment [18] consists of two different environments. The blue environment runs in production with the current software version, and the green environment is idle waiting for a new deployment. After performing a new deployment in green and verifying that it works correctly, the traffic is switched to the green environment, and the nodes in the blue environment are put into idle mode. On the other hand, the rolling pattern allows the software to be progressively updated (node by node) in a group of nodes or servers. Each node is taken offline while the new software version is deployed and evaluated. If the evaluation is successful, the node is enabled to receive traffic, and the next node is taken offline to be upgraded.

As part of future work, we plan to study the implementation of software deployment patterns for the IoT system. Our DSL could be extended to enable the specification of application deployment and update using these patterns. This involves an in-depth study to abstract the domain concepts, merge them to the current metamodel, and define the concrete syntax. System monitoring to collect QoS and infrastructure metrics is currently supported by our framework. Some additional metrics should be collected to measure the impact of deployed software updates. Additionally, adaptations such as canary propagation or rollback should be integrated into the adaptation engine.

Allocation Strategies

Unlike the cloud layer, the edge and fog layers are composed of nodes with processing and storage limitations that restrict application deployment. One challenges posed by this fact is related to making intelligent allocation decisions to guarantee QoS. To deploy or offload an application in the system, it is important to select edge/fog nodes that have sufficient resources to host and run the application properly. Orchestrators commonly provide a component in charge of making allocation decisions. For example, Kubernetes uses its scheduler component to determine and select the appropriate node to host the pod and container. However, this scheduler only analyze the resources requested (CPU and RAM) by the container [140]. Other factors such as node CPU consumption, energy consumption, network latency, reliability, and bandwidth usage should be considered to make allocation decisions. For example, when deploying a container that houses a real-time application, in which low latency is one of the essential requirements, it is important to select the nodes that can offer the lowest latency.

Currently, our framework uses the Kubernetes scheduler for allocation decisions. Especially when scaling or offloading adaptations are executed without define a target node to deploy the new container. For example, the Scaling adaptation defined in the adaptation rule in Figure 4.13, defines a target region (*Beach Hotel*) but not a target node. Then, when the scaling is performed, the Kubernetes scheduler selects a node in the *Hotel Beach* region with the necessary resources to host the node. However, the scheduler has some limitations as described above. For this reason, the design (or implementation if it exists) of a scheduler that considers additional factors such as node CPU consumption, power consumption, latency, and bandwidth consumption is one of the future directions of this thesis.

9.3.3 Security Strategies

With the growth of IoT systems and interconnected devices, the number of vulnerabilities that put sensitive or relevant information at risk is also increasing. The impact of an attack on the devices in any of the layers of the architecture can cause loss of critical information, disasters in the processes that control the system, unavailability of the system, among others. Therefore, it is essential to ensure the security of the IoT system to defend against attacks.

Although this thesis does not address the security of IoT systems, it is a relevant topic to cover in the future. The implementation of security strategies to ensure the confidentiality, integrity, and availability of information at different levels is a necessity for IoT systems. Below we propose two contributions to promote security.

- Role-based access control (RBAC) is an access control strategy that restricts users based on roles and privileges. RBAC enables the assignment of permissions by grouping users into a set of roles that are ordered by hierarchy [107]. For instance, security strategies based on roles and privileges commonly implemented in MQTT asynchronous communications are username and password authentication and Access Control List (to control subscribing and publishing on broker topics). Additionally, some adaptations are so critical that they require authorization from a user (at runtime) before being executed automatically by the system. For example, in a WWTP the opening of a valve that pours a toxic chemical into a tank can be semi-automated by defining a rule, but each time the valve is to be opened, the authorization of the plant manager is required. This type of processes and privileges can be handled by RBAC. Therefore, We plan to extend our DSL to enable the specification of RBAC policies at different levels. For example to control publication or subscription to the broker, or to authorize critical system adaptations.

- There are a large number of attacks that can be conducted in IoT environments. These are grouped into four categories [129]: Probe (consists of exploiting network vulnerabilities), User to Root Attacks (U2R consists of illegally gaining root access to a computer resource), Remote to Local Attacks (R2L consists of exploiting vulnerabilities by sending packets to gain illegal local access to resources on that network), and Denial of Service Attacks (DoS consists of making a service inaccessible). Several studies such as [88, 129, 66, 60] have proposed strategies to detect these attacks by analyzing the data stream in real time. One of our future lines is focused on the design or reuse of detection strategies for these attacks to enable the definition of rules involving security concerns. For example, rules to define an action in case of detecting a DoS attack.

9.3.4 Machine Learning Algorithms to Support Adaptation

Machine learning is a branch of artificial intelligence that consists of a program's ability to learn automatically to identify patterns and make predictions with minimal human intervention. Machine learning algorithms can be designed to support activities at different stages of the MAPE-K loop of self-adaptive systems [70]. We have identified some tasks in our MAPE-K based framework that could be supported by machine learning algorithms.

- **Prediction of dynamic events**. The prediction of dynamic events and abnormal use of resources (such as CPU, memory, and energy) by the nodes and containers of the IoT system is one of the topics that we seek to address through learning algorithms. For example, in [144], a learning algorithm is designed to predict the energy consumption of smart buildings, and in [52], neural network algorithms are proposed to predict the CPU usage of cloud nodes. Predicting these dynamic events could enable the system to adapt before the event occurs, ensuring better availability and QoS levels.

- Another line of future research consists in the design of learning algorithms capable of **suggesting optimal adaptations to meet non-functional system requirements**. That is, algorithms that help with the design of the adaptation plan in the Plan stage of the MAPE-K loop. For example, a system must guarantee a response latency of less than 100ms, then the learning algorithm should suggest an adaptation plan (which could comprise several actions) depending on the current state of the system to meet that requirement.

- Finally, another possible contribution of the use of machine learning algorithms to our thesis is the support in the **updating of previously defined rules** taking into account the changing conditions of the system architecture. For example, at the design stage, the user defines a rule that involves offloading a container to the target node *edge-1*. If the node *edge-1* has an irreparable failure and the rule has not yet been fired, it should be updated to establish a new target node. This update decision can be supported by a machine learning algorithm.

Bibliography

[1] F. Ahmadighohandizi and K. Systä. Application development and deployment for iot devices. In *European Conference on Service-Oriented and Cloud Computing*, pages 74–85. Springer, 2016.

[2] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash. Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE communications surveys & tutorials*, 17(4):2347–2376, 2015.

[3] S. Al-Sarawi, M. Anbar, K. Alieyan, and M. Alzubaidi. Internet of things (iot) communication protocols. In *2017 8th International conference on information technology (ICIT)*, pages 685–690. IEEE, 2017.

[4] M. Alaa, A. A. Zaidan, B. B. Zaidan, M. Talal, and M. L. M. Kiah. A review of smart home applications based on internet of things. *Journal of Network and Computer Applications*, 97:48–65, 2017.

[5] A. H. Alavi, P. Jiao, W. G. Buttlar, and N. Lajnef. Internet of things-enabled smart cities: State-of-the-art and future trends. *Measurement*, 129:589–606, 2018.

[6] I. Alfonso, K. Garcés, H. Castro, and J. Cabot. Self-adaptive architectures in IoT systems: a systematic literature review. *Journal of Internet Services and Applications*, 12(1):1–28, 2021.

[7] I. Alfonso, K. Garcés, H. Castro, and J. Cabot. Modeling self-adaptative IoT architectures. In *2021 ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 761–766, 2021.

[8] I. Alfonso, C. Goméz, K. Garcés, and J. Chavarriaga. Lifetime optimization of wireless sensor networks for gas monitoring in underground coal mining. In *2018 7th International Conference on Computers Communications and Control (ICCCC)*, pages 224–230. IEEE, 2018.

[9] F. Alkhabbas, I. Murturi, R. Spalazzese, P. Davidsson, and S. Dustdar. A goal-driven approach for deploying self-adaptive iot systems. In *2020 IEEE International Conference on Software Architecture (ICSA)*, pages 146–156. IEEE, 2020.

[10] M. Alrowaily and Z. Lu. Secure edge computing in iot systems: Review and case studies. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 440–444. IEEE, 2018.

[11] M. Artac, T. Borovšak, E. Di Nitto, M. Guerriero, D. Perez-Palacin, and D. A. Tamburri. Infrastructure-as-code for data-intensive architectures: A model-driven development approach. In *2018 IEEE International Conference on Software Architecture (ICSA)*, pages 156–15609. IEEE, 2018.

[12] K. Ashton et al. That 'internet of things' thing. *RFID journal*, 22(7):97–114, 2009.

[13] M. Asif-Ur-Rahman, F. Afsana, M. Mahmud, M. S. Kaiser, M. R. Ahmed, O. Kaiwartya, and A. James-Taylor. Toward a heterogeneous mist, fog, and cloud-based framework for the internet of healthcare things. *IEEE Internet of Things Journal*, 6(3):4049–4062, 2018.

[14] U. Aßmann, S. Götz, J.-M. Jézéquel, B. Morin, and M. Trapp. A reference architecture and roadmap for models@ run. time systems. In *Models@ run. time*, pages 1–18. Springer, 2014.

[15] A. Barišić, V. Amaral, and M. Goulão. Usability driven dsl development with use-me. *Computer Languages, Systems & Structures*, 51:118–157, 2018.

[16] J. A. Barriga, P. J. Clemente, E. Sosa-Sánchez, and Á. E. Prieto. Simulateiot: Domain specific language to design, code generation and execute iot simulation environments. *IEEE Access*, 9:92531–92552, 2021.

[17] L. Bass, P. Clements, and R. Kazman. *Software architecture in practice*. Addison-Wesley Professional, 2003.

[18] L. Bass, I. Weber, and L. Zhu. *DevOps: A software architect's perspective*. Addison-Wesley Professional, 2015.

[19] J. Beauquier, B. Bérard, and L. Fribourg. A new rewrite method for proving convergence of self-stabilizing systems. In *International Symposium on Distributed Computing*, pages 240–255, Bratislava, 1999. Springer.

[20] I. Bedhief, L. Foschini, P. Bellavista, M. Kassar, and T. Aguili. Toward self-adaptive software defined fog networking architecture for iiot and industry 4.0. In *2019 IEEE 24th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*, pages 1–5. IEEE, 2019.

[21] N. Bencomo, R. B. France, B. H. Cheng, and U. Aßmann. *Models@ run. time: foundations, applications, and roadmaps*, volume 8378. Springer, 2014.

[22] N. Bencomo and L. H. G. Paucar. Ram: Causally-connected and requirements-aware runtime models using bayesian learning. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 216–226. IEEE, 2019.

[23] T. Berger, M. Völter, H. P. Jensen, T. Dangprasert, and J. Siegmund. Efficiency of projectional editing: A controlled experiment. In *Proc. of the 24th ACM SIGSOFT Int. Symposium on Foundations of Software Engineering*, pages 763–774, 2016.

[24] A. Bergmayr, U. Breitenbücher, O. Kopp, M. Wimmer, G. Kappel, and F. Leymann. From architecture modeling to application provisioning for the cloud by combining uml and tosca. In *CLOSER (2)*, pages 97–108, 2016.

[25] G. Blair, N. Bencomo, and R. B. France. Models@run.time. *Computer*, 42(10):22–27, 2009.

[26] M. Brambilla, J. Cabot, and M. Wimmer. *Model-driven software engineering in practice, 2nd edn. Synthesis Lectures on Software Engineering*. Morgan & Claypool Publishers, USA, 2017.

[27] M. Breitbach, D. Schäfer, J. Edinger, and C. Becker. Context-aware data and task placement in edge computing environments. In *2019 IEEE International Conference on Pervasive Computing and Communications (PerCom*, pages 1–10. IEEE, 2019.

[28] D. Bri, M. Fernández-Diego, M. Garcia, F. Ramos, and J. Lloret. How the weather impacts on the performance of an outdoor wlan. *IEEE Communications Letters*, 16(8):1184–1187, 2012.

[29] A. Bucchiarone, A. Cicchetti, F. Ciccozzi, and A. Pierantonio. *Domain-specific Languages in Practice: With JetBrains MPS.* Springer, 2021.

[30] R. Buyya and S. N. Srirama. *Fog and edge computing: principles and paradigms.* John Wiley & Sons, 2019.

[31] A. Cañete, M. Amor, and L. Fuentes. Supporting iot applications deployment on edge-based infrastructures using multi-layer feature models. *Journal of Systems and Software*, 183:111086, 2022.

[32] E. A. Castillo and A. Ahmadinia. Iot-based multi-view machine vision systems. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 5206–5212. IEEE, 2019.

[33] A. Chehri, T. El Ouahmani, and N. Hakem. Mining and iot-based vehicle adhoc network: industry opportunities and innovation. *Internet of Things*, page 100117, 2019.

[34] L. Chen, P. Zhou, L. Gao, and J. Xu. Adaptive fog configuration for the industrial internet of things. *IEEE Transactions on Industrial Informatics*, 14(10):4656–4664, 2018.

[35] W. Chen, C. Liang, Y. Wan, C. Gao, G. Wu, J. Wei, and T. Huang. More: A model-driven operation service for cloud-based it systems. In *2016 IEEE International Conference on Services Computing (SCC)*, pages 633–640. IEEE, 2016.

[36] A. M. K. Cheng. Self-stabilizing real-time rule-based systems. In *Proceedings 11th Symposium on Reliable Distributed Systems*, pages 172–173, Houston, 1992. IEEE Computer Society.

[37] B. Cheng, A. Papageorgiou, F. Cirillo, and E. Kovacs. Geelytics: Geodistributed edge analytics for large scale iot systems based on dynamic topology. In *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, pages 565–570. IEEE, 2015.

[38] B. H. Cheng, K. I. Eder, M. Gogolla, L. Grunske, M. Litoiu, H. A. Müller, P. Pelliccione, A. Perini, N. A. Qureshi, B. Rumpe, et al. Using models at runtime to address assurance for self-adaptive systems. In *Models@ run. time*, pages 101–136. Springer, 2014.

[39] L. Cianciaruso, F. di Forenza, E. Di Nitto, M. Miglierina, N. Ferry, and A. Solberg. Using models at runtime to support adaptable monitoring of multiclouds applications. In *2014 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 401–408. IEEE, 2014.

[40] F. Ciccozzi and R. Spalazzese. Mde4iot: supporting the internet of things with model-driven engineering. In *International Symposium on Intelligent and Distributed Computing*, pages 67–76. Springer, 2016.

[41] S. Cirani, L. Davoli, G. Ferrari, R. Léone, P. Medagliani, M. Picone, and L. Veltri. A scalable and self-configuring architecture for service discovery in the internet of things. *IEEE Internet of Things Journal*, 1(5):508–521, 2014.

[42] J. Colistra. The evolving architecture of smart cities. *2018 IEEE International Smart Cities Conference, ISC2 2018*, 2019. cited By 0.

[43] B. Costa, P. F. Pires, F. C. Delicato, W. Li, and A. Y. Zomaya. Design and analysis of iot applications: A model-driven approach. In *2016 IEEE 14th Intl Conf on Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech)*, pages 392–399. IEEE, 2016.

[44] K. Cui, W. Sun, and W. Sun. Joint computation offloading and resource management for usvs cluster of fog-cloud computing architecture. In *2019 IEEE International Conference on Smart Internet of Things (SmartIoT)*, pages 92–99. IEEE, 2019.

[45] K. Czarnecki. Overview of generative software development. In *International Workshop on Unconventional Programming Paradigms*, pages 326–341. Springer, 2004.

[46] M. S. de Brito, S. Hoque, T. Magedanz, R. Steinke, A. Willner, D. Nehls, O. Keils, and F. Schreiner. A service orchestration architecture for fog-enabled infrastructures. In *2017 Second International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 127–132. IEEE, 2017.

[47] G.-C. Deng and K. Wang. An application-aware qos routing algorithm for sdn-based iot networking. In *2018 IEEE Symposium on Computers and Communications (ISCC)*, pages 00186–00191. IEEE, 2018.

[48] K. Desikan, M. Srinivasan, and C. Murthy. A novel distributed latency-aware data processing in fog computing-enabled iot networks. In *Proceedings of the ACM Workshop on Distributed Information Processing in Wireless Networks*, page 4. ACM, 2017.

[49] X. E. DevOps. Best practices for devops: Advanced deployment patterns. urlhttps://www.wsta.org/wp-content/uploads/2018/09/Best-Practices-for-DevOps-Advanced-Deployment-Patterns.pdf, 2018.

[50] J. Dizdarević, F. Carpio, A. Jukan, and X. Masip-Bruin. A survey of communication protocols for internet of things and related challenges of fog and cloud computing integration. *ACM Computing Surveys (CSUR)*, 51(6):1–29, 2019.

[51] Á. Domingo, J. Echeverría, Ó. Pastor, and C. Cetina. Comparing uml-based and dsl-based modeling from subjective and objective perspectives. In *International Conference on Advanced Information Systems Engineering*, pages 483–498. Springer, 2021.

[52] M. Duggan, K. Mason, J. Duggan, E. Howley, and E. Barrett. Predicting host cpu utilization in cloud computing using recurrent neural networks. In *2017 12th international conference for internet technology and secured transactions (ICITST)*, pages 67–72. IEEE, 2017.

[53] S. Dustdar, C. Avasalcai, and I. Murturi. Edge and fog computing: Vision and research challenges. In *2019 IEEE Int. Conf. on Service-Oriented System Engineering (SOSE)*, pages 96–9609. IEEE, 2019.

[54] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano. Devops. *Ieee Software*, 33(3):94–100, 2016.

[55] L. Erazo-Garzón, P. Cedillo, G. Rossi, and J. Moyano. A domain-specific language for modeling iot system architectures that support monitoring. *IEEE Access*, 10:61639–61665, 2022.

[56] J. Erbel, F. Korte, and J. Grabowski. Comparison and runtime adaptation of cloud application topologies based on occi. In *CLOSER*, pages 517–525, 2018.

[57] D. Ernst, A. Becker, and S. Tai. Rapid canary assessment through proxying and two-stage load balancing. In *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 116–122. IEEE, 2019.

[58] T. Eterovic, E. Kaljic, D. Donko, A. Salihbegovic, and S. Ribic. An internet of things visual domain specific modeling language based on uml. In *2015 XXV International Conference on Information, Communication and Automation Technologies (ICAT)*, pages 1–5. IEEE, 2015.

[59] I. G. EV. International energy agency: Paris, 2018.

[60] N. Farnaaz and M. Jabbar. Random forest modeling for network intrusion detection system. *Procedia Computer Science*, 89:213–217, 2016.

[61] N. Ferry, F. Chauvel, H. Song, A. Rossini, M. Lushpenko, and A. Solberg. Cloudmf: Model-driven management of multi-cloud applications. *ACM Transactions on Internet Technology (TOIT)*, 18(2):1–24, 2018.

[62] N. Ferry, H. Song, A. Rossini, F. Chauvel, and A. Solberg. Cloudmf: applying mde to tame the complexity of managing multi-cloud applications. In *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, pages 269–277. IEEE, 2014.

[63] H. Flores, X. Su, V. Kostakos, A. Y. Ding, P. Nurmi, S. Tarkoma, P. Hui, and Y. Li. Large-scale offloading in the internet of things. In *2017 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 479–484. IEEE, 2017.

[64] F. Fouquet, E. Daubert, N. Plouzeau, O. Barais, J. Bourcier, and J.-M. Jézéquel. Dissemination of reconfiguration policies on mesh networks. In *IFIP International Conference on Distributed Applications and Interoperable Systems*, pages 16–30. Springer, 2012.

[65] M. Fowler. *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional, 2004.

[66] S. Ganapathy, K. Kulothungan, S. Muthurajkumar, M. Vijayalakshmi, P. Yogesh, and A. Kannan. Intelligent feature selection and classification techniques for intrusion detection in networks: a survey. *EURASIP Journal on Wireless Communications and Networking*, 2013(1):1–16, 2013.

[67] C. G. García, D. Meana-Llorián, V. García-Díaz, A. C. Jiménez, and J. P. Anzola. Midgar: Creation of a graphic domain-specific language to generate smart objects for internet of things scenarios using model-driven engineering. *IEEE Access*, 8:141872–141894, 2020.

[68] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[69] D. Garlan, B. Schmerl, and S.-W. Cheng. Software architecture-based self-adaptation. In *Autonomic computing and networking*, pages 31–55. Springer, 2009.

[70] O. Gheibi, D. Weyns, and F. Quin. Applying machine learning in self-adaptive systems: A systematic literature review. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 15(3):1–37, 2021.

[71] N. K. Giang, R. Lea, M. Blackstock, and V. C. Leung. Fog at the edge: Experiences building an edge computing platform. In *2018 IEEE International Conference on Edge Computing (EDGE)*, pages 9–16. IEEE, 2018.

[72] T. Gomes, P. Lopes, J. Alves, P. Mestre, J. Cabral, J. L. Monteiro, and A. Tavares. A modeling domain-specific language for iot-enabled operating systems. In *IECON 2017-43rd Annual Conference of the IEEE Industrial Electronics Society*, pages 3945–3950. IEEE, 2017.

[73] A. Gómez, M. Iglesias-Urkia, L. Belategi, X. Mendialdua, and J. Cabot. Model-driven development of asynchronous message-driven architectures with asyncapi. *Software and Systems Modeling*, pages 1–29, 2021.

[74] A. Gómez, M. Iglesias-Urkia, A. Urbieta, and J. Cabot. A model-based approach for developing event-driven architectures with asyncapi. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 121–131, 2020.

[75] J. Greenfield and K. Short. Software factories: assembling applications with patterns, models, frameworks and tools. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 16–27, 2003.

[76] S. Gregor and A. R. Hevner. Positioning and presenting design science research for maximum impact. *MIS quarterly*, pages 337–355, 2013.

[77] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future generation computer systems*, 29(7):1645–1660, 2013.

[78] R. Guntha. Iot architectures for noninvasive blood glucose and blood pressure monitoring. In *2019 9th International Symposium on Embedded Computing and System Design (ISED)*, pages 1–5. IEEE, 2019.

[79] Z. Guo, Y. Sun, S.-Y. Pan, and P.-C. Chiang. Integration of green energy and advanced energy-efficient technologies for municipal wastewater treatment plants. *International journal of environmental research and public health*, 16(7):1282, 2019.

[80] A. Hagedorn, D. Starobinski, and A. Trachtenberg. Rateless deluge: Over-the-air programming of wireless sensor networks using random linear codes. In *2008 International Conference on Information Processing in Sensor Networks (ipsn 2008)*, pages 457–466. IEEE, 2008.

[81] T. Holmes. Facilitating migration of cloud infrastructure services: A model-based approach. In *CloudMDE@ MoDELS*, pages 7–12, 2015.

[82] G. Huang, G.-B. Huang, S. Song, and K. You. Trends in extreme learning machines: A review. *Neural Networks*, 61:32–48, 2015.

[83] M. Hussein, S. Li, and A. Radermacher. Model-driven development of adaptive iot systems. In *MODELS (Satellite Events)*, pages 17–23, 2017.

[84] J. Ingeno. *Software Architect's Handbook: Become a successful software architect by implementing effective architecture concepts.* Packt Publishing Ltd, 2018.

[85] J. Islam, E. Harjula, T. Kumar, P. Karhula, and M. Ylianttila. Docker enabled virtualized nanoservices for local IoT edge networks. In *IEEE Conf. on Standards for Communications and Networking (CSCN)*, pages 1–7, 2019.

[86] S. Y. Jang, Y. Lee, B. Shin, and D. Lee. Towards application-aware virtualization for edge iot clouds. In *Proceedings of the 13th International Conference on Future Internet Technologies*, page 4. ACM, 2018.

[87] N. Jazdi. Cyber physical systems in the context of industry 4.0. In *IEEE Int. Conference on Automation, Quality and Testing, Robotics*, pages 1–4. IEEE, 2014.

[88] P. G. Jeya, M. Ravichandran, and C. Ravichandran. Efficient classifier for r2l and u2r attacks. *International Journal of Computer Applications*, 45(21):28–32, 2012.

[89] Y. Jiang, Z. Huang, and D. H. Tsang. Challenges and solutions in fog computing orchestration. *IEEE Network*, 32(3):122–129, 2017.

[90] M. Jutila. An adaptive edge router enabling internet of things. *IEEE Internet of Things Journal*, 3(6):1061–1069, 2016.

[91] S. Keele et al. Guidelines for performing systematic literature reviews in software engineering. Technical report, Technical report, Ver. 2.3 EBSE Technical Report. EBSE, 2007.

[92] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[93] D. Kimovski, H. Ijaz, N. Saurabh, and R. Prodan. Adaptive nature-inspired fog architecture. In *2018 IEEE 2nd International Conference on Fog and Edge Computing (ICFEC)*, pages 1–8. IEEE, 2018.

[94] B. Kitchenham, O. P. Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman. Systematic literature reviews in software engineering–a systematic literature review. *Information and software technology*, 51(1):7–15, 2009.

[95] B. Kitchenham and P. Brereton. A systematic review of systematic review process research in software engineering. *Information and software technology*, 55(12):2049–2075, 2013.

[96] P. Knights and B. Scanlan. A study of mining fatalities and coal price variation. *International Journal of Mining Science and Technology*, 29(4):599–602, 2019.

[97] C. Krupitzer, F. M. Roth, S. VanSyckel, G. Schiele, and C. Becker. A survey on engineering approaches for self-adaptive systems. *Pervasive and Mobile Computing*, 17:184–206, 2015.

[98] A. Latifah, S. H. Supangkat, and A. Ramelan. Smart building: A literature review. In *Int. Conf. on ICT for Smart Society (ICISS)*, pages 1–6, 2020.

[99] E. Lee, Y.-D. Seo, and Y.-G. Kim. Self-adaptive framework based on mape loop for internet of things. *sensors*, 19(13):2996, 2019.

[100] X. Li, D. Li, J. Wan, C. Liu, and M. Imran. Adaptive transmission optimization in sdn-based industrial internet of things with edge computing. *IEEE Internet of Things Journal*, 5(3):1351–1360, 2018.

[101] Y. Li, Y.-h. Chiu, and T.-Y. Lin. Coal production efficiency and land destruction in china's coal mining industry. *Resources Policy*, 63:101449, 2019.

[102] Y. Liao, E. d. F. R. Loures, and F. Deschamps. Industrial internet of things: A systematic literature review and insights. *IEEE Internet of Things Journal*, 5(6):4515–4525, 2018.

[103] B. Lorenzo, J. Garcia-Rois, X. Li, J. Gonzalez-Castano, and Y. Fang. A robust dynamic edge network architecture for the internet of things. *IEEE Network*, 32(1):8–15, 2018.

[104] S. Madakam, V. Lake, V. Lake, V. Lake, et al. Internet of things (iot): A literature review. *Journal of Computer and Communications*, 3(05):164, 2015.

[105] S. Mahdavi-Hezavehi, P. Avgeriou, and D. Weyns. A classification framework of uncertainty in architecture-based self-adaptive systems with multiple quality requirements. In *Managing Trade-Offs in Adaptable Software Architectures*, pages 45–77. Elsevier, 2017.

[106] S. T. March and G. F. Smith. Design and natural science research on information technology. *Decision support systems*, 15(4):251–266, 1995.

[107] S. Martínez, A. Fouche, S. Gérard, and J. Cabot. Automatic generation of security compliant (virtual) model views. In *International Conference on Conceptual Modeling*, pages 109–117. Springer, 2018.

[108] J. Mass, C. Chang, and S. N. Srirama. Context-aware edge process management for mobile thing-to-fog environment. In *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*, pages 1–7, 2018.

[109] A. Mavromatis, A. P. Da Silva, K. Kondepu, D. Gkounis, R. Nejabati, and D. Simeonidou. A software defined device provisioning framework facilitating scalability in internet of things. In *2018 IEEE 5G World Forum (5GWF)*, pages 446–451. IEEE, 2018.

[110] C. Mechalikh, H. Taktak, and F. Moussa. A scalable and adaptive tasks orchestration platform for iot. In *2019 15th International Wireless Communications & Mobile Computing Conference (IWCMC)*, pages 1557–1563. IEEE, 2019.

[111] B. Mishra and A. Kertesz. The use of mqtt in m2m and iot systems: A survey. *IEEE Access*, 8:201071–201086, 2020.

[112] M. T. Moghaddam, E. Rutten, P. Lalanda, and G. Giraud. Ias: an iot architectural self-adaptation framework. In *European Conference on Software Architecture*, pages 333–351. Springer, 2020.

[113] D. Montero and R. Serral-Gracià. Offloading personal security applications to the network edge: A mobile user case scenario. In *2016 International Wireless Communications and Mobile Computing Conference (IWCMC)*, pages 96–101. IEEE, 2016.

[114] R. Morabito and N. Beijar. A framework based on sdn and containers for dynamic service chains on iot gateways. In *Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems*, pages 42–47. ACM, 2017.

[115] K. Morris. *Infrastructure as code: managing servers in the cloud.* " O'Reilly Media, Inc.", 2016.

[116] H. Muccini and M. Sharaf. Caps: Architecture description of situational aware cyber physical systems. In *2017 IEEE International Conference on Software Architecture (ICSA)*, pages 211–220. IEEE, 2017.

[117] H. Muccini, R. Spalazzese, M. T. Moghaddam, and M. Sharaf. Self-adaptive iot architectures: An emergency handling case study. In *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*, pages 1–6, 2018.

[118] R. Muñoz, R. Vilalta, N. Yoshikane, R. Casellas, R. Martínez, T. Tsuritani, and I. Morita. Integration of iot, transport sdn, and edge/cloud computing for dynamic distribution of iot analytics and efficient use of network resources. *Journal of Lightwave Technology*, 36(7):1420–1428, 2018.

[119] D. Nandan Jha, K. Alwasel, A. Alshoshan, X. Huang, R. K. Naha, S. K. Battula, S. Garg, D. Puthal, P. James, A. Y. Zomaya, et al. Iotsim-edge: A simulation framework for modeling the behaviour of iot and edge computing environments. *arXiv e-prints*, pages arXiv–1910, 2019.

[120] J. Nielsen and T. K. Landauer. A mathematical model of the finding of usability problems. In *Proc. of the INTERACT'93 and CHI'93 conf. on Human factors in computing systems*, pages 206–213, 1993.

[121] C. Pahl, N. El Ioini, S. Helmer, and B. Lee. An architecture pattern for trusted orchestration in iot edge clouds. In *2018 Third International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 63–70. IEEE, 2018.

[122] C. Pahl and B. Lee. Containers and clusters for edge cloud architectures–a technology review. In *2015 3rd international conference on future internet of things and cloud*, pages 379–386. IEEE, 2015.

[123] P. Patel, M. I. Ali, and A. Sheth. On using the intelligent edge for iot analytics. *IEEE Intelligent Systems*, 32(5):64–69, 2017.

[124] P. Patel and D. Cassou. Enabling high-level application development for the internet of things. *Journal of Systems and Software*, 103:62–84, 2015.

[125] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee. A design science research methodology for information systems research. *Journal of management information systems*, 24(3):45–77, 2007.

[126] S. Peros, H. Janjua, S. Akkermans, W. Joosen, and D. Hughes. Dynamic qos support for iot backhaul networks through sdn. In *2018 Third International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 187–192. IEEE, 2018.

[127] N. Petrovic and M. Tosic. Smada-fog: Semantic model driven approach to deployment and adaptivity in fog computing. *Simulation Modelling Practice and Theory*, 101:102033, 2020.

[128] D. Pizzolli, G. Cossu, D. Santoro, L. Capra, C. Dupont, D. Charalampos, F. De Pellegrini, F. Antonelli, and S. Cretti. Cloud4iot: A heterogeneous, distributed and autonomic cloud platform for the iot. In *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 476–479. IEEE, 2016.

[129] S. Prabavathy, K. Sundarakantham, and S. M. Shalinie. Design of cognitive fog computing for intrusion detection in internet of things. *Journal of Communications and Networks*, 20(3):291–298, 2018.

[130] F. Pramudianto, M. Eisenhauer, C. A. Kamienski, D. Sadok, and E. J. Souto. Connecting the internet of things rapidly through a model driven approach. In *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, pages 135–140. IEEE, 2016.

[131] T. Rausch, S. Nastic, and S. Dustdar. Emma: distributed qos-aware mqtt middleware for edge computing applications. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 191–197. IEEE, 2018.

[132] M. d. M. y. E. Republica de Colombia. Reglamento de seguridad en labores minera subterráneas, 2015.

[133] A. Rhayem, M. B. A. Mhiri, and F. Gargouri. Semantic web technologies for the internet of things: Systematic literature review. *Internet of Things*, page 100206, 2020.

[134] J. Rubin and D. Chisnell. *Handbook of usability testing: how to plan, design and conduct effective tests.* John Wiley & Sons, New Jersey, 2008.

[135] E. Rutten, N. Marchand, and D. Simon. Feedback control as mape-k loop in autonomic computing. In *Software Engineering for Self-Adaptive Systems III. Assurances*, pages 349–373. Springer, 2017.

[136] A. Salihbegovic, T. Eterovic, E. Kaljic, and S. Ribic. Design of a domain specific language and ide for internet of things applications. In *2015 38th international convention on information and communication technology, electronics and microelectronics (MIPRO)*, pages 996–1001. IEEE, 2015.

[137] H. Sami and A. Mourad. Towards dynamic on-demand fog computing formation based on containerization technology. In *2018 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 960–965. IEEE, 2018.

[138] J. Sandobalin, E. Insfran, and S. Abrahão. Argon: A model-driven infrastructure provisioning tool. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 738–742. IEEE, 2019.

[139] J. Santos, T. Wauters, B. Volckaert, and F. De Turck. Fog computing: Enabling the management and orchestration of smart city applications in 5g networks. *Entropy*, 20(1):4, 2018.

[140] J. Santos, T. Wauters, B. Volckaert, and F. De Turck. Resource provisioning in fog computing: From theory to practice. *Sensors*, 19(10):2238, 2019.

[141] B. Sarma, G. Kumar, R. Kumar, and T. Tuithung. Fog computing: An enhanced performance analysis emulation framework for iot with load balancing smart gateway architecture. In *2019 International Conference on Communication and Electronics Systems (ICCES)*, pages 1–5. IEEE, 2019.

[142] M. Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.

[143] R. Scolati, I. Fronza, N. El Ioini, A. Samir, and C. Pahl. A containerized big data streaming architecture for edge cloud computing on clustered single-board devices. In *9th Int. Conf. on Cloud Computing and Services Science*, pages 68–80, 2019.

[144] M. K. M. Shapi, N. A. Ramli, and L. J. Awalin. Energy consumption prediction by using machine learning for smart building: Case study in malaysia. *Developments in the Built Environment*, 5:100037, 2021.

[145] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE internet of things journal*, 3(5):637–646, 2016.

[146] M. Singh and G. Baranwal. Quality of service (qos) in internet of things. In *2018 3rd International Conference On Internet of Things: Smart Innovation and Usages (IoT-SIU)*, pages 1–6, Feb 2018.

[147] S. Singh and N. Singh. Containers & docker: Emerging roles & future of cloud technology. In *2016 2nd International Conference on Applied and Theoretical*

Computing and Communication Technology (iCATccT), pages 804–807. IEEE, 2016.

[148] O. Skarlat, V. Karagiannis, T. Rausch, K. Bachmann, and S. Schulte. A framework for optimization, service placement, and runtime operation in the fog. In *2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC)*, pages 164–173. IEEE, 2018.

[149] K. Sledziewski, B. Bordbar, and R. Anane. A dsl-based approach to software development and deployment on cloud. In *2010 24th IEEE International Conference on Advanced Information Networking and Applications*, pages 414–421. IEEE, 2010.

[150] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[151] T. Suganuma, T. Oide, S. Kitagami, K. Sugawara, and N. Shiratori. Multiagent-based flexible edge computing architecture for iot. *IEEE Network*, 32(1):16–23, 2018.

[152] V. Theodorou and N. Diamantopoulos. Glt: Edge gateway elt for data-driven intelligence placement. In *2019 IEEE/ACM Joint 4th International Workshop on Rapid Continuous Software Engineering and 1st International Workshop on Data-Driven Decisions, Experimentation and Evolution (RCoSE/DDrEE)*, pages 24–27. IEEE, 2019.

[153] J. R. Torres Neto, G. P. Rocha Filho, L. Y. Mano, L. A. Villas, and J. Ueyama. Exploiting offloading in iot-based microfog: experiments with face recognition and fall detection. *Wireless Communications and Mobile Computing*, 2019, 2019.

[154] C.-L. Tseng and F. J. Lin. Extending scalability of iot/m2m platforms with fog computing. In *2018 IEEE 4th World Forum on Internet of Things (WF-IoT)*, pages 825–830. IEEE, 2018.

[155] I. T. Union. Internet of things global standards initiative, 2012.

[156] M. F. van Amstel, M. G. van den Brand, and P. H. Nguyen. Metrics for model transformations. In *Proceedings of the Ninth Belgian-Netherlands Software Evolution Workshop (BENEVOL 2010), Lille, France (December 2010)*, 2010.

[157] A. Van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.

[158] K. Velasquez, D. P. Abreu, D. Gonçalves, L. Bittencourt, M. Curado, E. Monteiro, and E. Madeira. Service orchestration in fog environments. In *2017 IEEE 5th International Conference on Future Internet of Things and Cloud (FiCloud)*, pages 329–336. IEEE, 2017.

[159] R. Vilalta, C. V. Apte, J. L. Hellerstein, S. Ma, and S. M. Weiss. Predictive algorithms in the management of computer systems. *IBM Systems Journal*, 41(3):461–474, 2002.

[160] M. Villari, M. Fazio, S. Dustdar, O. Rana, and R. Ranjan. Osmotic computing: A new paradigm for edge/cloud integration. *IEEE Cloud Computing*, 3(6):76–83, 2016.

[161] M. Voelter. Embedded software development with projectional language workbenches. In *International Conference on Model Driven Engineering Languages and Systems*, pages 32–46. Springer, 2010.

[162] M. Völter. Language and ide modularization, extension and composition with mps. *Pre-proceedings of Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE)*, pages 395–431, 2011.

[163] J. Wang, J. Pan, and F. Esposito. Elastic urban video surveillance system using edge computing. In *Proceedings of the Workshop on Smart Internet of Things*, page 7. ACM, 2017.

[164] Z. Wen, R. Yang, P. Garraghan, T. Lin, J. Xu, and M. Rovatsos. Fog orchestration for internet of things services. *IEEE Internet Computing*, 21(2):16–24, 2017.

[165] D. Weyns, M. U. Iftikhar, D. Hughes, and N. Matthys. Applying architecture-based adaptation to automate the management of internet-of-things. In *European Conf. on Software Architecture*, pages 49–67, 2018.

[166] WINSYSTEMS. Cloud, fog and edge computing – what's the difference? urlhttps://www.winsystems.com/cloud-fog-and-edge-computing-whats-the-difference/, 2017.

[167] T. Wong, M. Wagner, and C. Treude. Self-adaptive systems: A systematic literature review across categories and domains. *arXiv preprint arXiv:2101.00125*, 2021.

[168] B. Wood and A. Azim. Triton: a domain specific language for cyber-physical systems. In *2021 22nd IEEE International Conference on Industrial Technology (ICIT)*, volume 1, pages 810–816. IEEE, 2021.

[169] World Wide Web Consortium (W3C). Semantic sensor network ontology. URL: https://www.w3.org/TR/2017/REC-vocab-ssn-20171019/, 10 2017.

[170] D. Wu, M. M. Omwenga, Y. Liang, L. Yang, D. Huston, and T. Xia. A fog computing framework for cognitive portable ground penetrating radars. In *ICC 2019-2019 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2019.

[171] B. Wukkadada, K. Wankhede, R. Nambiar, and A. Nair. Comparison with http and mqtt in internet of things (iot). In *2018 International Conference on Inventive Research in Computing Applications (ICIRCA)*, pages 249–253. IEEE, 2018.

[172] B. Yang, A. Sailer, S. Jain, A. E. Tomala-Reyes, M. Singh, and A. Ramnath. Service discovery based blue-green deployment technique in cloud native environments. In *2018 IEEE International Conference on Services Computing (SCC)*, pages 185–192. IEEE, 2018.

[173] M. B. Yassein, M. Q. Shatnawi, S. Aljwarneh, and R. Al-Hatmi. Internet of things: Survey and open issues of mqtt protocol. In *2017 international conference on engineering & MIS (ICEMIS)*, pages 1–6. Ieee, 2017.

[174] E. Yigitoglu, M. Mohamed, L. Liu, and H. Ludwig. Foggy: a framework for continuous automated iot application deployment in fog computing. In *2017 IEEE International Conference on AI & Mobile Services (AIMS)*, pages 38–45. IEEE, 2017.

[175] R. Young, S. Fallon, and P. Jacob. Dynamic collaboration of centralized & edge processing for coordinated data management in an iot paradigm. In *2018 IEEE 32nd International Conference on Advanced Information Networking and Applications (AINA)*, pages 694–701. IEEE, 2018.

[176] R. Young, S. Fallon, and P. Jacob. A governance architecture for self-adaption & control in iot applications. In *2018 5th International Conference on Control, Decision and Information Technologies (CoDIT)*, pages 241–246. IEEE, 2018.

[177] A. Yousefpour, A. Patil, G. Ishigaki, I. Kim, X. Wang, H. C. Cankaya, Q. Zhang, W. Xie, and J. P. Jue. Fogplan: a lightweight qos-aware dynamic fog service provisioning framework. *IEEE Internet of Things Journal*, 6(3):5080–5096, 2019.

Appendix A

Large Modeling Example

In this appendix, using our DSL we model the IoT system for environment control in underground coal mines tested in the experiments of Section 7.3. Figure 7.18 shows the scenario to be modeled: an IoT system deployed in three underground coal mines. The description of this system as the device, edge/fog, cloud, and application layers is presented in Section 7.3).

A.1 Modeling Mine Structure and Control Points

Figure A.1 shows the modeling of a portion of the *Mine 1* and *Mine 2* structures. To monitor the environment inside the mine, there are control points at each work face. Each control point has ten sensors to monitor several physical variables: methane (CH_4), carbon dioxide sensor (CO_2), carbon monoxide sensor (CO), hydrogen sulfide (H_2S), sulfurous anhydride (SO_2), nitric oxide (NO), nitrogen dioxide (NO_2), temperature, and air velocity.

These physical variables and their thresholds were extracted from the Colombian mining regulations [132]. According to this mining regulation, there are two types of thresholds for gas measurements: the permissible limit for an 8-hour averaging time known as TWA, and the permissible limit value for a short exposure time (max. 15 minutes) known as STEL. Figure A.2 shows the modeling of the control point located at the *m1-WFace-1* working face in the *Mine 1*. The thresholds for the gas sensors correspond to the STEL limit values.

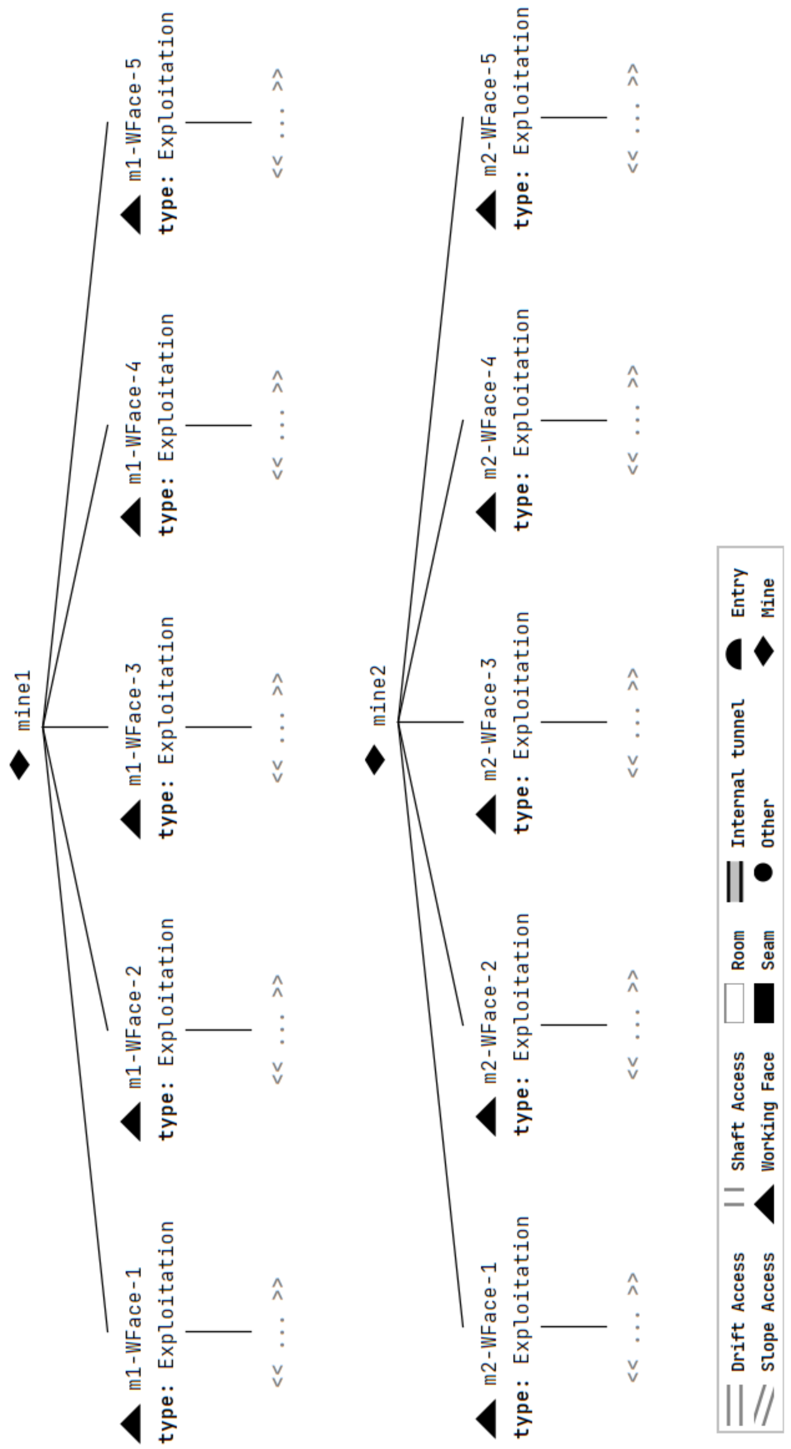
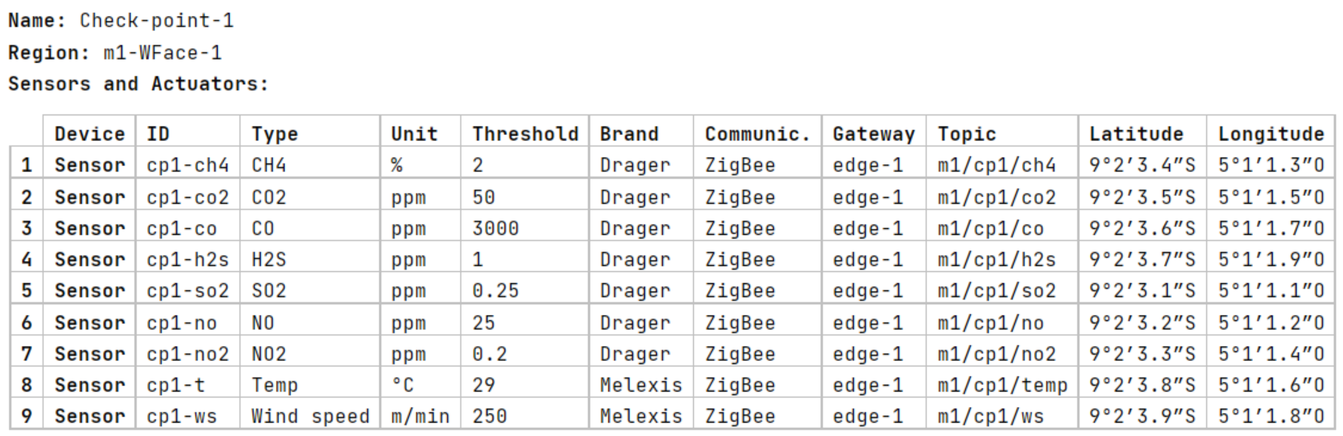


Figure A.1: Model of the *Mine 1* and *Mine 2* structures



Name: Check-point-1
Region: m1-WFace-1
Sensors and Actuators:

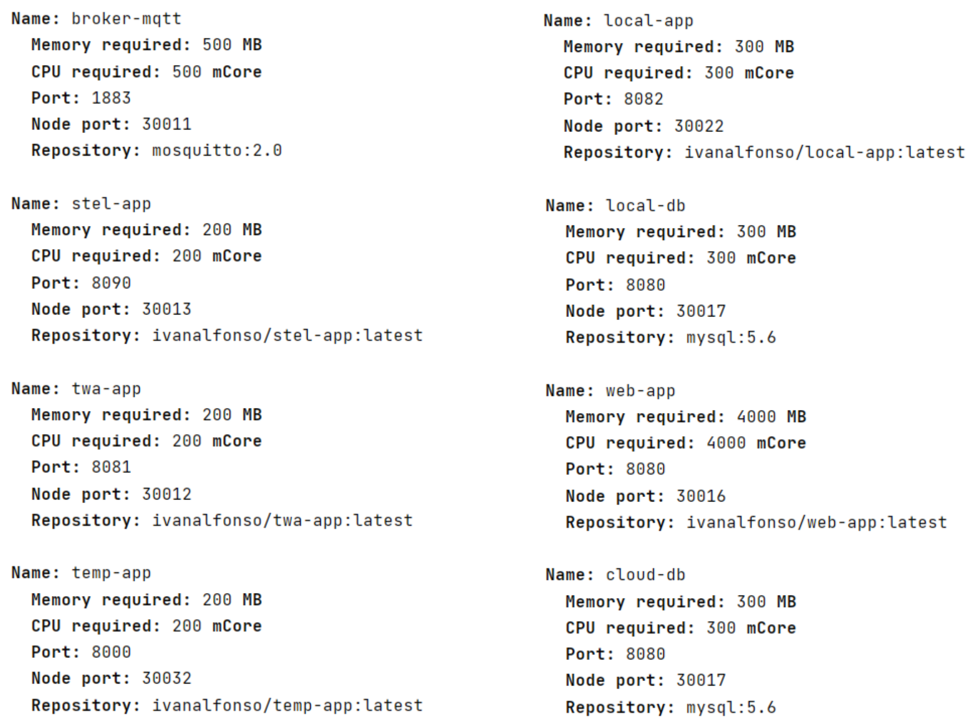
	Device	ID	Type	Unit	Threshold	Brand	Communic.	Gateway	Topic	Latitude	Longitude
1	Sensor	cp1-ch4	CH4	%	2	Drager	ZigBee	edge-1	m1/cp1/ch4	9°2′3.4″S	5°1′1.3″O
2	Sensor	cp1-co2	CO2	ppm	50	Drager	ZigBee	edge-1	m1/cp1/co2	9°2′3.5″S	5°1′1.5″O
3	Sensor	cp1-co	CO	ppm	3000	Drager	ZigBee	edge-1	m1/cp1/co	9°2′3.6″S	5°1′1.7″O
4	Sensor	cp1-h2s	H2S	ppm	1	Drager	ZigBee	edge-1	m1/cp1/h2s	9°2′3.7″S	5°1′1.9″O
5	Sensor	cp1-so2	SO2	ppm	0.25	Drager	ZigBee	edge-1	m1/cp1/so2	9°2′3.1″S	5°1′1.1″O
6	Sensor	cp1-no	NO	ppm	25	Drager	ZigBee	edge-1	m1/cp1/no	9°2′3.2″S	5°1′1.2″O
7	Sensor	cp1-no2	NO2	ppm	0.2	Drager	ZigBee	edge-1	m1/cp1/no2	9°2′3.3″S	5°1′1.4″O
8	Sensor	cp1-t	Temp	°C	29	Melexis	ZigBee	edge-1	m1/cp1/temp	9°2′3.8″S	5°1′1.6″O
9	Sensor	cp1-ws	Wind speed	m/min	250	Melexis	ZigBee	edge-1	m1/cp1/ws	9°2′3.9″S	5°1′1.8″O

Figure A.2: Model of the Check Point in *m1-WFace-1* work front

A.2 Modeling Applications and Nodes

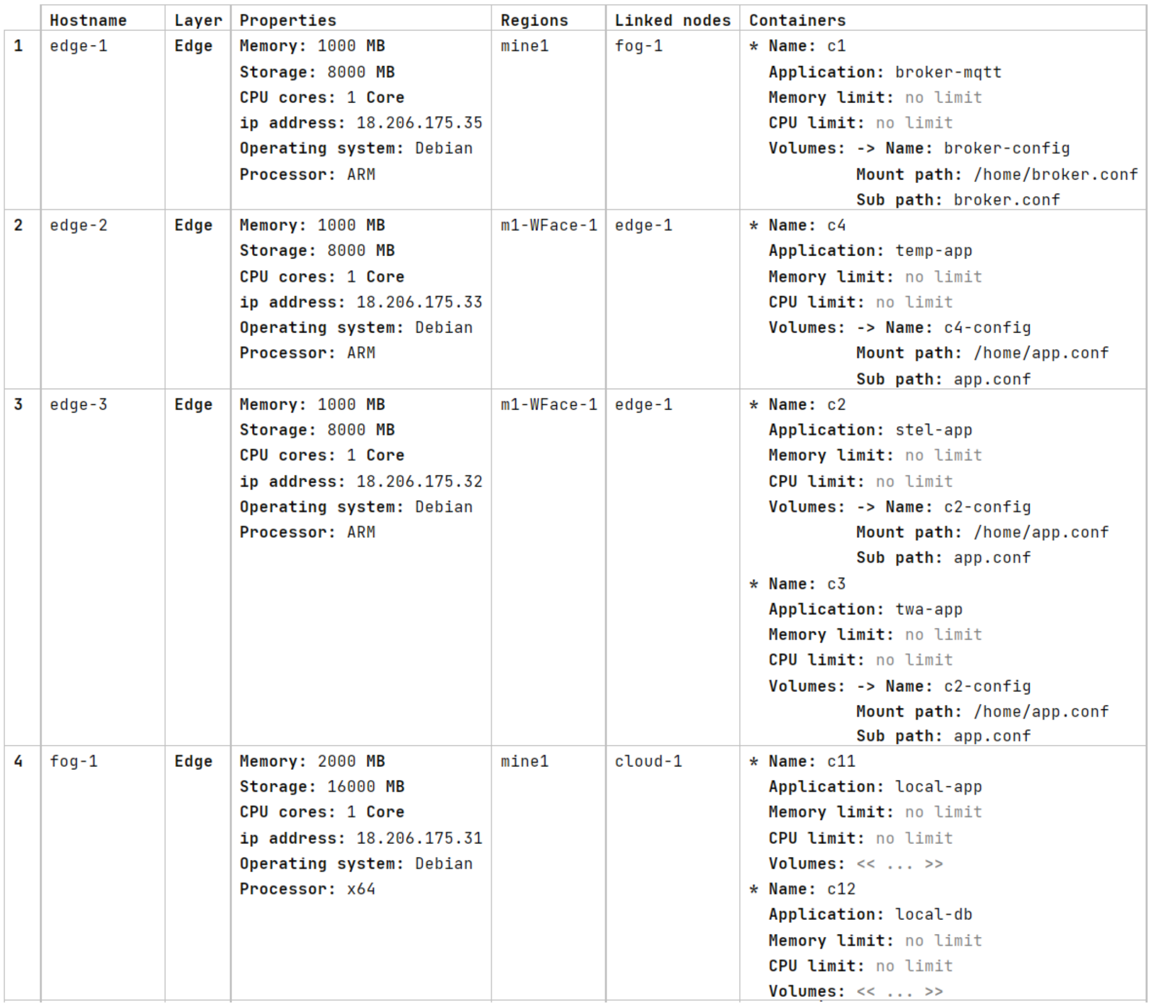
Figure A.3 shows the model of the list of applications that are deployed in the system nodes. The *twa-app* application checks that the TWA thresholds are not exceeded by the gas sensors, the *stel-app* application checks the STEL thresholds, and the *temp-app* application checks the temperature status taking into account the wind speed. These three applications are deployed on edge nodes. *local-app* (local application to access real time data sensors) and *local-db* (local database to store sensor data) are deployed in fog nodes. Finally, the applications *web-app* (web application to query historical data) and *cloud-db* (database to store aggregated information) are hosted on cloud nodes.

Figure A.4 shows the modeling of some nodes of *Mine 1*: three edge nodes hosting the broker and, the *stel-app*, the *twa-app*, and the *temp-app* applications, and a fog node hosting the *local-db* and *local-app* applications. Some of the deployed containers require ConfigMap objects for configuration. For example, the MQTT broker settings.



```
Name: broker-mqtt
  Memory required: 500 MB
  CPU required: 500 mCore
  Port: 1883
  Node port: 30011
  Repository: mosquitto:2.0

Name: stel-app
  Memory required: 200 MB
  CPU required: 200 mCore
  Port: 8090
  Node port: 30013
  Repository: ivanalfonso/stel-app:latest

Name: twa-app
  Memory required: 200 MB
  CPU required: 200 mCore
  Port: 8081
  Node port: 30012
  Repository: ivanalfonso/twa-app:latest

Name: temp-app
  Memory required: 200 MB
  CPU required: 200 mCore
  Port: 8000
  Node port: 30032
  Repository: ivanalfonso/temp-app:latest

Name: local-app
  Memory required: 300 MB
  CPU required: 300 mCore
  Port: 8082
  Node port: 30022
  Repository: ivanalfonso/local-app:latest

Name: local-db
  Memory required: 300 MB
  CPU required: 300 mCore
  Port: 8080
  Node port: 30017
  Repository: mysql:5.6

Name: web-app
  Memory required: 4000 MB
  CPU required: 4000 mCore
  Port: 8080
  Node port: 30016
  Repository: ivanalfonso/web-app:latest

Name: cloud-db
  Memory required: 300 MB
  CPU required: 300 mCore
  Port: 8080
  Node port: 30017
  Repository: mysql:5.6
```

Figure A.3: Application modeling



	Hostname	Layer	Properties	Regions	Linked nodes	Containers
1	edge-1	Edge	Memory: 1000 MB	mine1	fog-1	* Name: c1
			Storage: 8000 MB			Application: broker-mqtt
			CPU cores: 1 Core			Memory limit: no limit
			ip address: 18.206.175.35			CPU limit: no limit
			Operating system: Debian			Volumes: -> Name: broker-config
			Processor: ARM			Mount path: /home/broker.conf
						Sub path: broker.conf
2	edge-2	Edge	Memory: 1000 MB	m1-WFace-1	edge-1	* Name: c4
			Storage: 8000 MB			Application: temp-app
			CPU cores: 1 Core			Memory limit: no limit
			ip address: 18.206.175.33			CPU limit: no limit
			Operating system: Debian			Volumes: -> Name: c4-config
			Processor: ARM			Mount path: /home/app.conf
						Sub path: app.conf
3	edge-3	Edge	Memory: 1000 MB	m1-WFace-1	edge-1	* Name: c2
			Storage: 8000 MB			Application: stel-app
			CPU cores: 1 Core			Memory limit: no limit
			ip address: 18.206.175.32			CPU limit: no limit
			Operating system: Debian			Volumes: -> Name: c2-config
			Processor: ARM			Mount path: /home/app.conf
						Sub path: app.conf
						* Name: c3
						Application: twa-app
						Memory limit: no limit
						CPU limit: no limit
						Volumes: -> Name: c2-config
						Mount path: /home/app.conf
						Sub path: app.conf
4	fog-1	Edge	Memory: 2000 MB	mine1	cloud-1	* Name: c11
			Storage: 16000 MB			Application: local-app
			CPU cores: 1 Core			Memory limit: no limit
			ip address: 18.206.175.31			CPU limit: no limit
			Operating system: Debian			Volumes: << ... >>
			Processor: x64			* Name: c12
						Application: local-db
						Memory limit: no limit
						CPU limit: no limit
						Volumes: << ... >>

Figure A.4: Nodes modeling

Appendix B

Installation and Configuration Guide

This appendix is a guide to the installation and configuration necessary to use our approach. Section B.1 presents the instructions for installing and configuring MPS to use our DSL. Section B.2 presents guidelines for using the DSL, and Section B.3 contains instructions for implementing our framework.

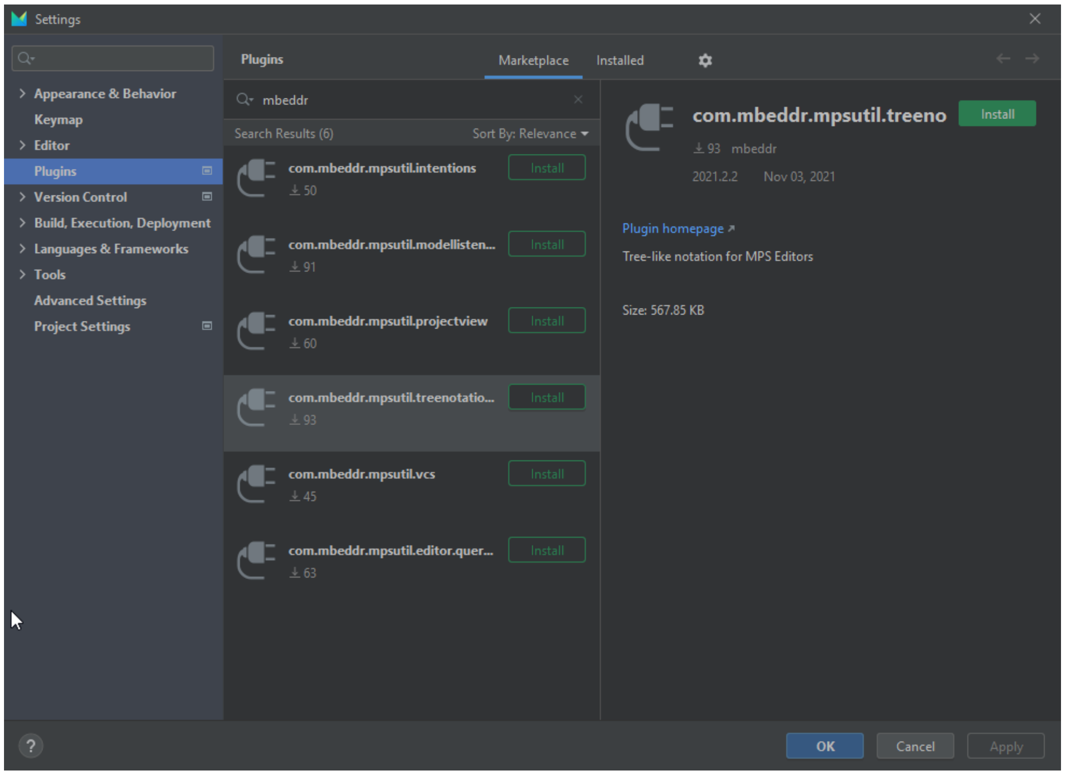
B.1 Installation and Configuration of MPS for the DSL

1. Download and install MPS version 2021.2.2[1]
2. Download or clone the project from GitHub repository[2]
3. Open MPS, and then open the DSL project by choosing the folder
4. Some plugins must be installed. Select File -> Settings -> Plugins and install the following plugins:

   Note: Some of these plugins require additional plugins that MPS will suggest you install (if this happens, select install). For example, the *com.dslfoundry.plaintext* plugin will require the *Mouse Selection Support* plugin.

[1]https://www.jetbrains.com/mps/download
[2]https://github.com/SOM-Research/selfadaptive-IoT-DSL

a) *com.mbeddr.mpsutil.treenotation*



mbeddr

Search Results (6) Sort By: Relevance

com.mbeddr.mpsutil.intentions ⤓ 50 Install

com.mbeddr.mpsutil.modellisten... ⤓ 91 Install

com.mbeddr.mpsutil.projectview ⤓ 60 Install

com.mbeddr.mpsutil.treenotatio... ⤓ 93 Install

com.mbeddr.mpsutil.vcs ⤓ 45 Install

com.mbeddr.mpsutil.editor.quer... ⤓ 63 Install

com.mbeddr.mpsutil.treeno Install

⤓ 93 mbeddr

2021.2.2 Nov 03, 2021

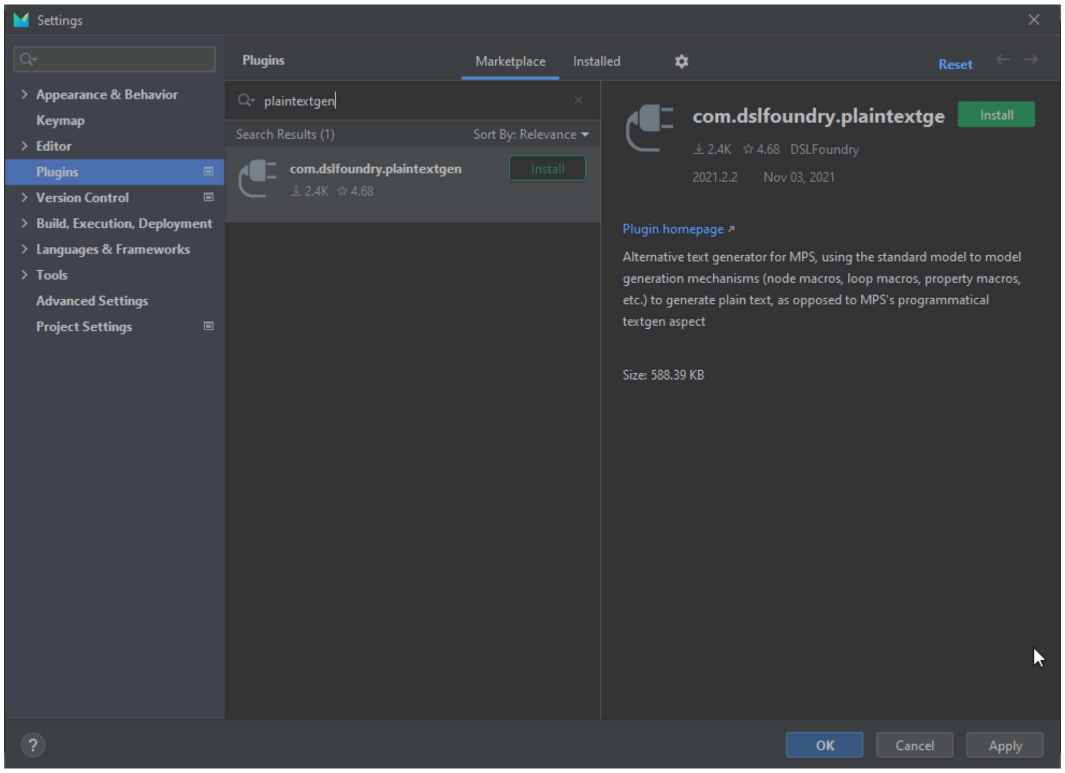
Plugin homepage ↗

Tree-like notation for MPS Editors

Size: 567.85 KB

Figure B.1: *treenotation* plugin installation in MPS

b) *com.dslfoundry.plaintextgen*



Alternative text generator for MPS, using the standard model to model generation mechanisms (node macros, loop macros, property macros, etc.) to generate plain text, as opposed to MPS's programmatical textgen aspect

Size: 588.39 KB

Figure B.2: *plaintextgen* plugin installation in MPS

c) *MPS Table Editor Component*

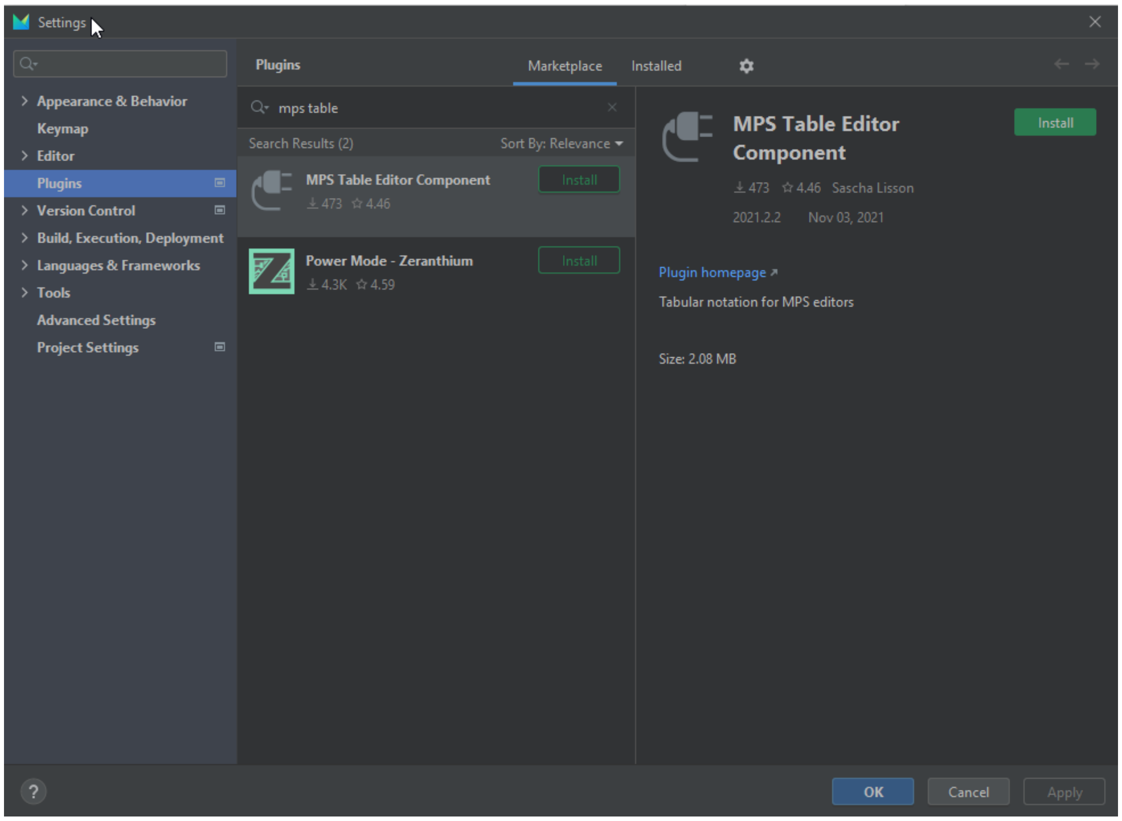


Figure B.3: *Table Editor* plugin installation in MPS

5. Restart MPS and you will now be able to use the DSL to model IoT systems. In the left pane (Logical View) you find an example of a modeled IoT system (Hotel Beach first floor). You can open this example model by double clicking and explore the concepts modeled for an IoT system.

B.2 DSL use and code generation

B.2.1 Changing the Notation

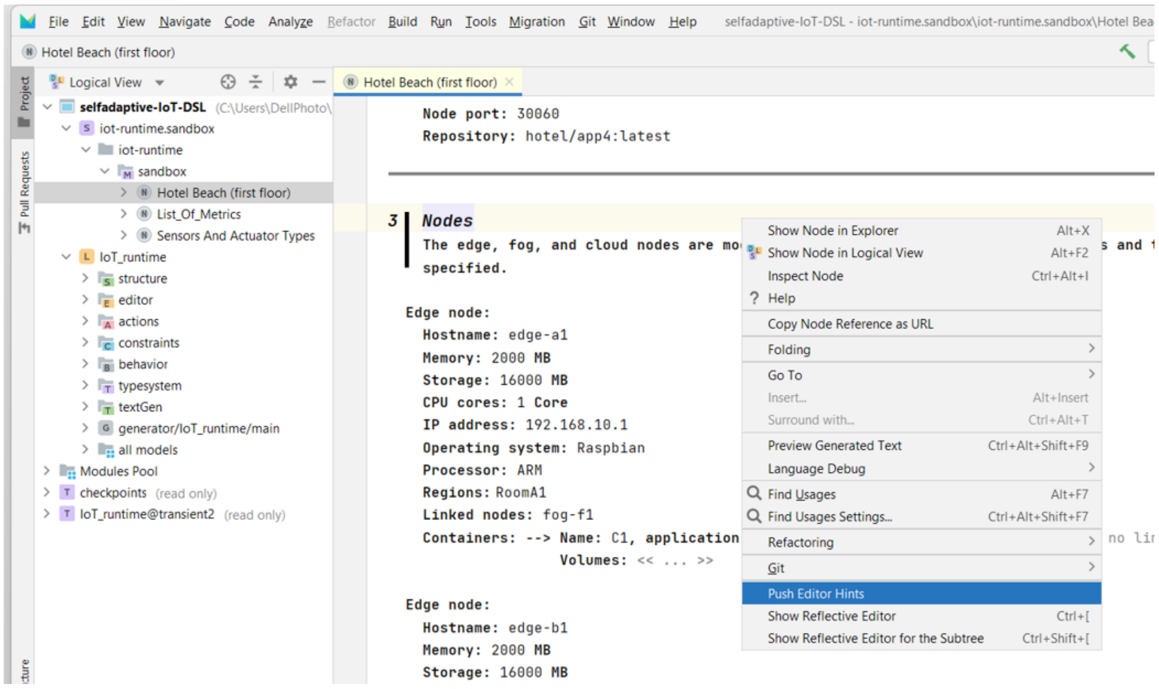
The DSL has three notations (textual, tabular, and tree) to model the IoT system concepts. The notations for each concept are shown in Table B.1.

Table B.1: DSL notation

Textual	Tabular	Tree
Applications	Nodes	Regions
Nodes	Containers	
Containers	IoT Devices	
IoT Devices		
Clusters		
Adaptation rules		

Three concepts (Nodes, Containers, and IoT Devices) can be modeled using two different notations (tabular and textual). The user is free to choose the notation. To change notation, follow the instructions below.

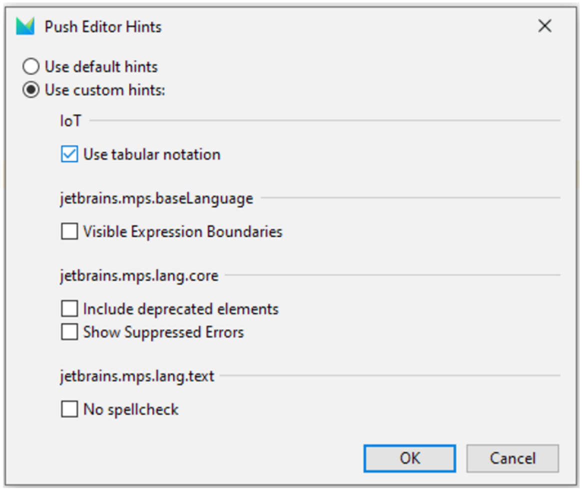
1. Right-click anywhere in the model workspace and select *Push Editor Hints.*


File Edit View Navigate Code Analyze Refactor Build Run Tools Migration Git Window Help
selfadaptive-IoT-DSL - iot-runtime.sandbox\iot-runtime.sandbox\Hotel Bea
Hotel Beach (first floor)
Project
Pull Requests
Logical View
Hotel Beach (first floor)
selfadaptive-IoT-DSL (C:\Users\DellPhoto\
iot-runtime.sandbox
iot-runtime
sandbox
Hotel Beach (first floor)
List_Of_Metrics
Sensors And Actuator Types
IoT_runtime
structure
editor
actions
constraints
behavior
typesystem
textGen
generator/IoT_runtime/main
all models
Modules Pool
checkpoints (read only)
IoT_runtime@transient2 (read only)
Show Node in Explorer Alt+X
Show Node in Logical View Alt+F2
Inspect Node Ctrl+Alt+I
Help
Copy Node Reference as URL
Folding
Go To
Insert... Alt+Insert
Surround with... Ctrl+Alt+T
Preview Generated Text Ctrl+Alt+Shift+F9
Language Debug
Find Usages Alt+F7
Find Usages Settings... Ctrl+Alt+Shift+F7
Refactoring
Git
Push Editor Hints
Show Reflective Editor Ctrl+[
Show Reflective Editor for the Subtree Ctrl+Shift+[

```
  Node port: 30060
  Repository: hotel/app4:latest

3 Nodes
  The edge, fog, and cloud nodes are mo                          s and t
  specified.

Edge node:
  Hostname: edge-a1
  Memory: 2000 MB
  Storage: 16000 MB
  CPU cores: 1 Core
  IP address: 192.168.10.1
  Operating system: Raspbian
  Processor: ARM
  Regions: RoomA1
  Linked nodes: fog-f1
  Containers: --> Name: C1, application                          no lin
                  Volumes: << ... >>

Edge node:
  Hostname: edge-b1
  Memory: 2000 MB
  Storage: 16000 MB
```

Figure B.4: Notation change

2. Select *Use custom hints* and then check *Use tabular notation*.



Push Editor Hints

( ) Use default hints
(●) Use custom hints:

IoT

[✓] Use tabular notation

jetbrains.mps.baseLanguage

[ ] Visible Expression Boundaries

jetbrains.mps.lang.core

[ ] Include deprecated elements
[ ] Show Suppressed Errors

jetbrains.mps.lang.text

[ ] No spellcheck

Figure B.5: Push Editor Hints

Now, you can see the model in tabular notation for *Nodes*, *Containers*, and *IoT devices*.

B.2.2 Creating New Model

To start modeling a new IoT system, you must create a new solution and model within the project. To do this, follow the instructions below.

1. Create new solution by right clicking on *selfadaptive-IoT-DSL -> New -> Solution*

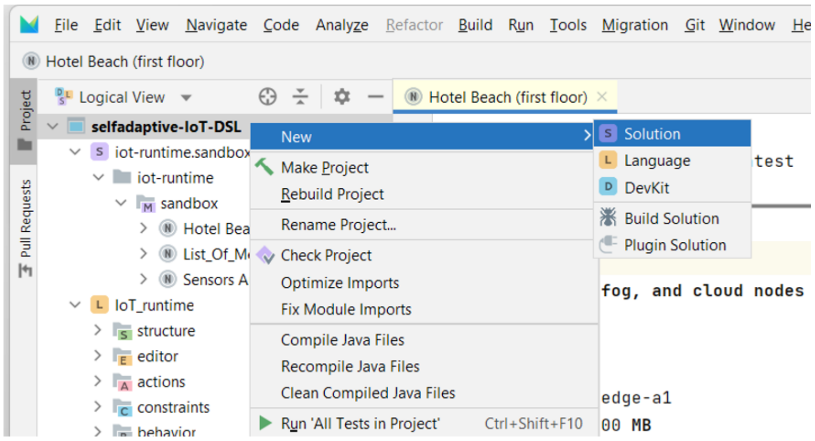


Figure B.6: Create new *Solution*

2. Then, create a new model by right clicking on *NewSolution -> New -> Model*

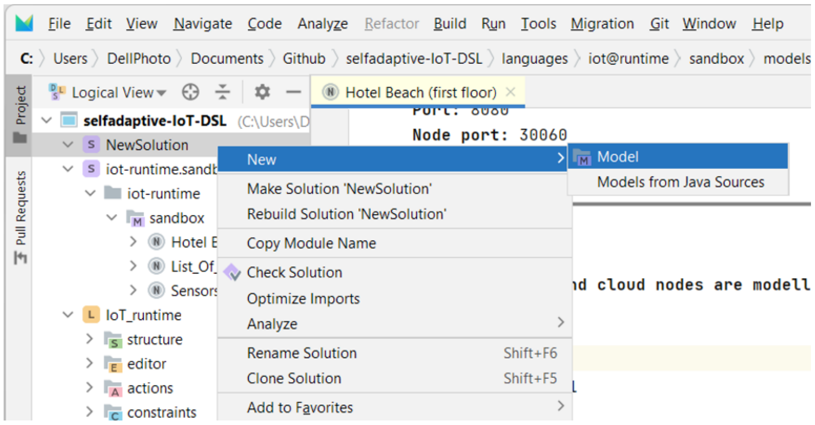
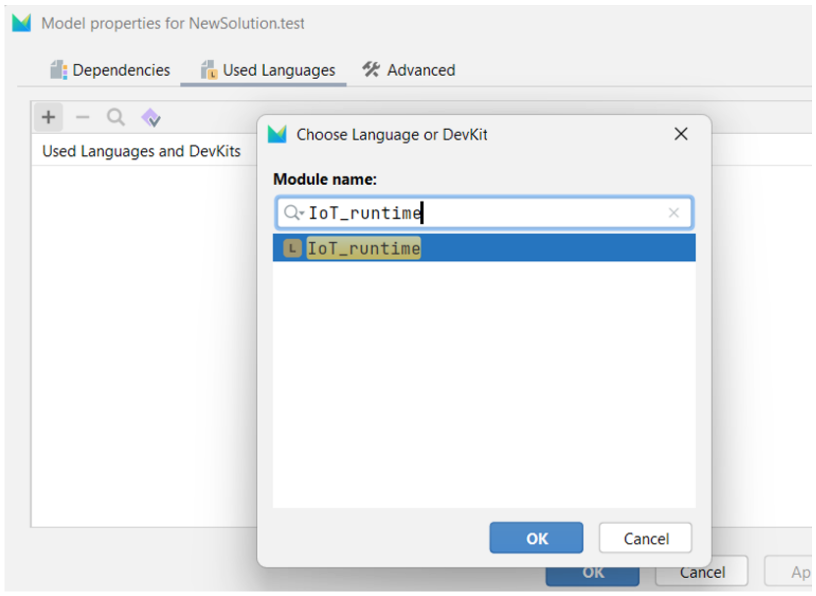

File Edit View Navigate Code Analyze Refactor Build Run Tools Migration Git Window Help
C: Users DellPhoto Documents Github selfadaptive-IoT-DSL languages iot@runtime sandbox models
Logical View
Hotel Beach (first floor)
Project
Pull Requests
selfadaptive-IoT-DSL (C:\Users\D
NewSolution
iot-runtime.sandb
iot-runtime
sandbox
Hotel E
List_Of_
Sensors
IoT_runtime
structure
editor
actions
constraints
Port: 8080
Node port: 30060
New
Make Solution 'NewSolution'
Rebuild Solution 'NewSolution'
Copy Module Name
Check Solution
Optimize Imports
Analyze
Rename Solution Shift+F6
Clone Solution Shift+F5
Add to Favorites
Model
Models from Java Sources
nd cloud nodes are modell

Figure B.7: Create new *Model*

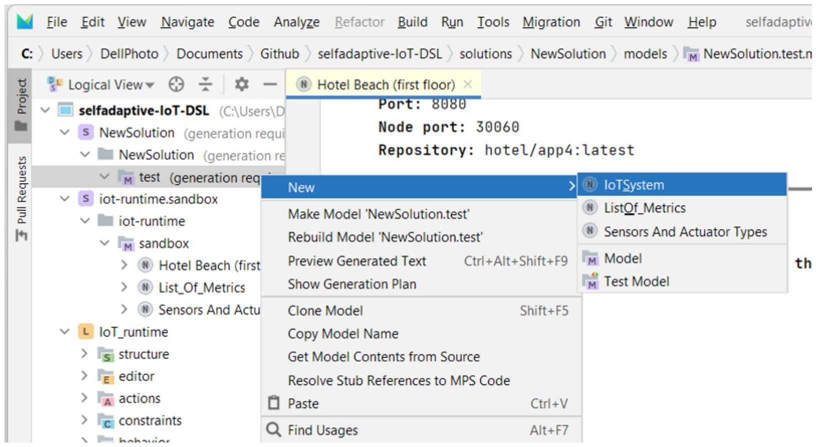
3. When you are creating a model, you have add IoT_runtime to *Used Languages*.


Model properties for NewSolution.test
Dependencies
Used Languages
Advanced
Used Languages and DevKits
Choose Language or DevKit
IoT_runtime
OK
Cancel
Ap

Module name: IoT_runtime

Figure B.8: Used languages by the new model

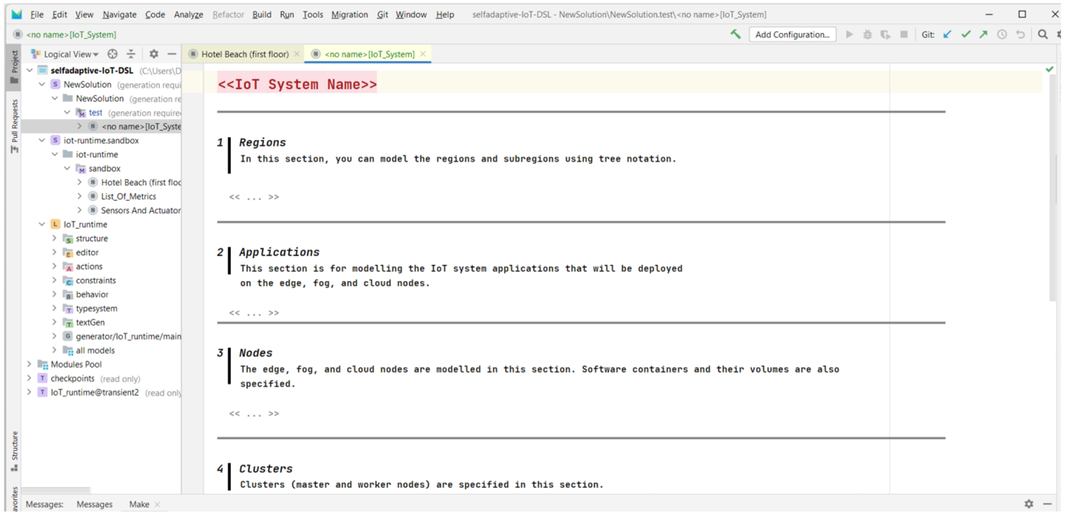
4. Now, you can create a new *IoT System model*.


File Edit View Navigate Code Analyze Refactor Build Run Tools Migration Git Window Help selfadaptiv
C: Users DellPhoto Documents Github selfadaptive-IoT-DSL solutions NewSolution models NewSolution.test.m
Logical View
Hotel Beach (first floor)
Project
Pull Requests
selfadaptive-IoT-DSL (C:\Users\D
NewSolution (generation requi
NewSolution (generation re
test (generation req
iot-runtime.sandbox
iot-runtime
sandbox
Hotel Beach (first
List_Of_Metrics
Sensors And Actu
IoT_runtime
structure
editor
actions
constraints
behavior
New
Make Model 'NewSolution.test'
Rebuild Model 'NewSolution.test'
Preview Generated Text Ctrl+Alt+Shift+F9
Show Generation Plan
Clone Model Shift+F5
Copy Model Name
Get Model Contents from Source
Resolve Stub References to MPS Code
Paste Ctrl+V
Find Usages Alt+F7
IoTSystem
ListOf_Metrics
Sensors And Actuator Types
Model
Test Model
th

```
Port: 8080
Node port: 30060
Repository: hotel/app4:latest
```

Figure B.9: Create *IoT System model*

5. Finally, you get a template for modeling the IoT system.

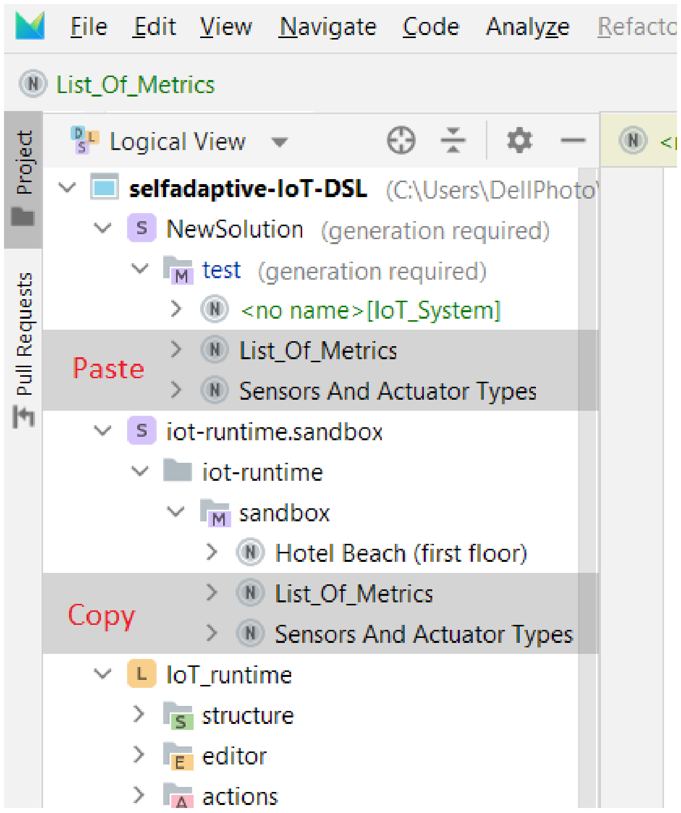


```
<<IoT System Name>>

1 | Regions
  | In this section, you can model the regions and subregions using tree notation.

  << ... >>

2 | Applications
  | This section is for modelling the IoT system applications that will be deployed
  | on the edge, fog, and cloud nodes.

  << ... >>

3 | Nodes
  | The edge, fog, and cloud nodes are modelled in this section. Software containers and their volumes are also
  | specified.

  << ... >>

4 | Clusters
  | Clusters (master and worker nodes) are specified in this section.
```

Figure B.10: Model template

B.2.3 Modeling an IoT system

Some concepts of the IoT system model must be created in different models. Specifically, the types of sensors and actuators, and the metrics that make up the adaptation rules are concepts that must be instantiated from other models. You could define these models from scratch, but a quick alternative is to reuse our sandbox models, which already contain predefined sensors, actuators, and metrics. To reuse these two models, copy them (right click and copy) from the sandbox and paste them (right click and paste) into your Solution (see Figure B.11).

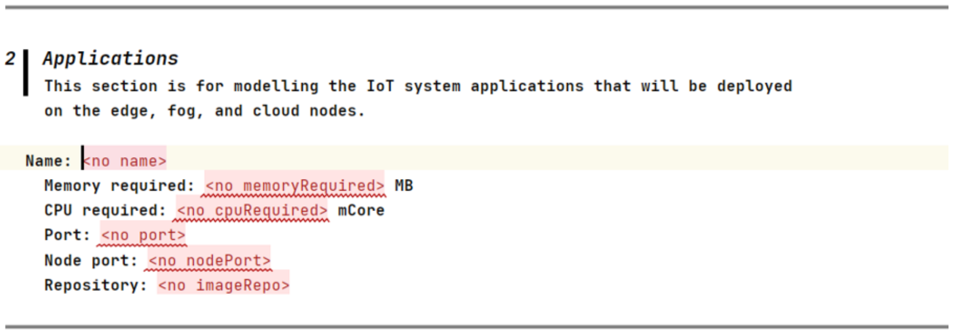


List_Of_Metrics

```
selfadaptive-IoT-DSL (C:\Users\DellPhoto\
  NewSolution (generation required)
    test (generation required)
      <no name>[IoT_System]
      List_Of_Metrics
      Sensors And Actuator Types
  iot-runtime.sandbox
    iot-runtime
      sandbox
        Hotel Beach (first floor)
        List_Of_Metrics
        Sensors And Actuator Types
  IoT_runtime
    structure
    editor
    actions
```

Figure B.11: Application template

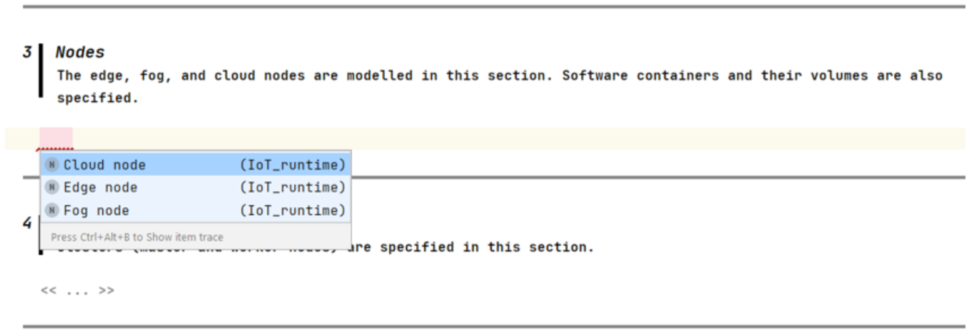
To model any aspect of the IoT system, just press the *Enter* key in the corresponding section and you will get a template with the attributes to be specified. For example, to model an application, press enter in the *Applications* section and you will get the model portion as shown in Figure B.12.



```
2  Applications
   This section is for modelling the IoT system applications that will be deployed
   on the edge, fog, and cloud nodes.

 Name: <no name>
   Memory required: <no memoryRequired> MB
   CPU required: <no cpuRequired> mCore
   Port: <no port>
   Node port: <no nodePort>
   Repository: <no imageRepo>
```

Figure B.12: Application template

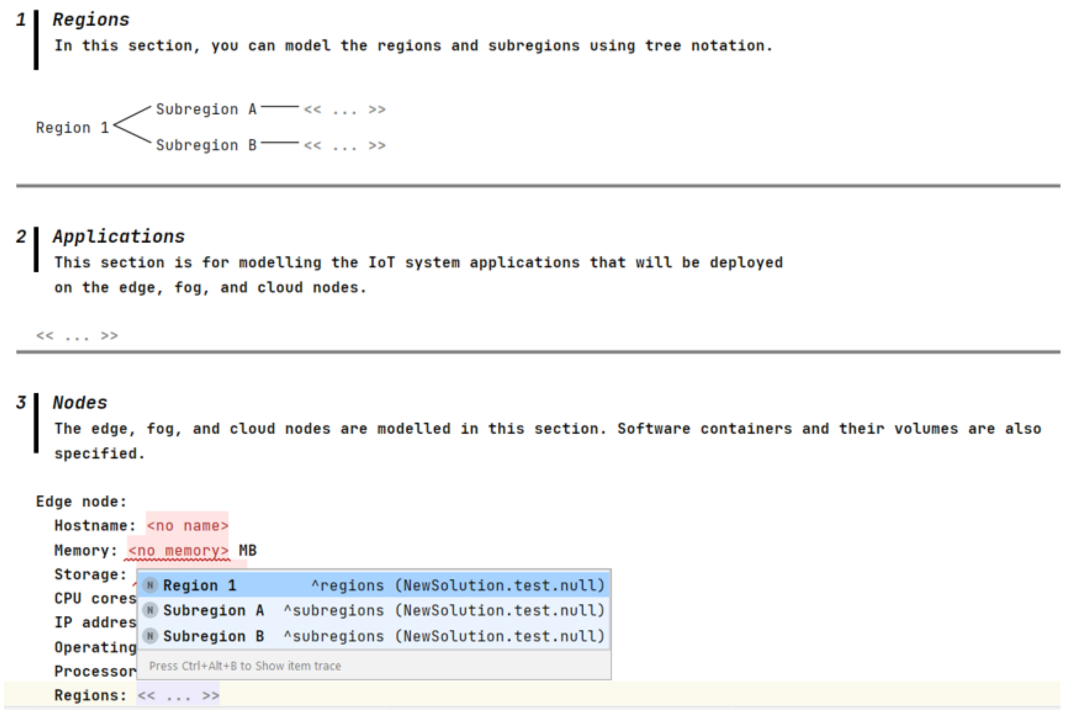
Some fields can be supported with the MPS autocomplete function. For example, when creating a new node, it is necessary to select the node type. To do this, press the *Enter* key in the *Nodes* section, and then the auto-complete function (by pressing *Ctrl+space* on windows or *Cmd+space* on MacOS). This will allow you to select one of the three types of nodes a shown in Figure B.13.



```
3  Nodes
   The edge, fog, and cloud nodes are modelled in this section. Software containers and their volumes are also
   specified.

   Cloud node            (IoT_runtime)
   Edge node             (IoT_runtime)
   Fog node              (IoT_runtime)
   Press Ctrl+Alt+B to Show item trace
4                                   are specified in this section.

 << ... >>
```

Figure B.13: MPS auto-complete function (Node type)

You can use the auto-complete function on any of the fields or attributes of a concept. In the example of Figure B.14, we have defined two subregions. Then, when modeling the region of an *Edge* node, the autocomplete function can be used to quickly select one of the subregions defined earlier.



```
1 Regions
  In this section, you can model the regions and subregions using tree notation.

Region 1 ─ Subregion A ── << ... >>
         ─ Subregion B ── << ... >>

2 Applications
  This section is for modelling the IoT system applications that will be deployed
  on the edge, fog, and cloud nodes.

<< ... >>

3 Nodes
  The edge, fog, and cloud nodes are modelled in this section. Software containers and their volumes are also
  specified.

Edge node:
  Hostname: <no name>
  Memory: <no memory> MB
  Storage:
  CPU cores
  IP addres
  Operating
  Processor
  Regions: << ... >>

  Region 1        ^regions (NewSolution.test.null)
  Subregion A  ^subregions (NewSolution.test.null)
  Subregion B  ^subregions (NewSolution.test.null)
  Press Ctrl+Alt+B to Show item trace
```

Figure B.14: MPS auto-complete function (Region)

B.2.4 Generating the Code

When the self-adapting IoT system model is finalized, you can verify the validity of the model and use the code generator to obtain the YAML manifests for deployment and execution of the runtime framework. Right click on the model and select Rebuild Model (see Figure B.15).

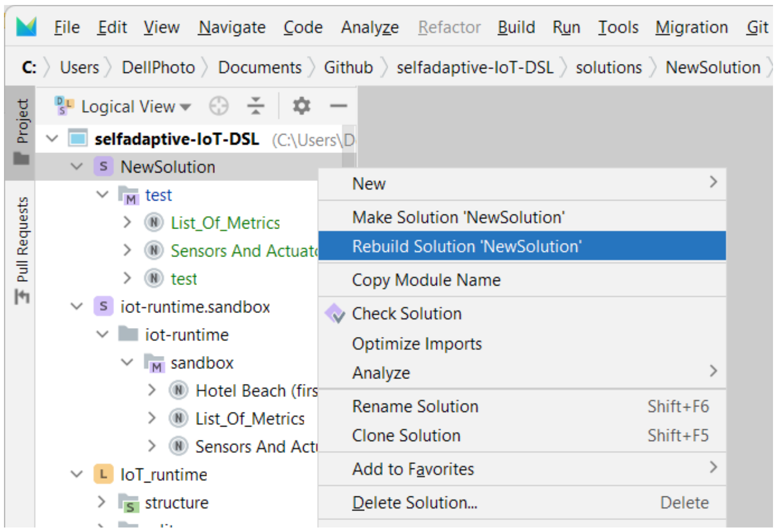
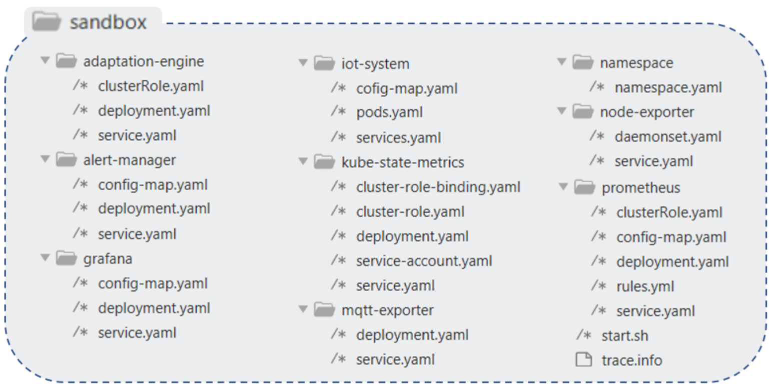


Figure B.15: Sandbox model compilation

If the model has no errors and the compilation is successful, then the generated code can be found in the directory «Project_directory»/solutions/«name_solution».

For example, the list of files generated when compiling the sandbox model (Beach Hotel) is shown in Figure B.16.



```
sandbox
▼ adaptation-engine          ▼ iot-system                      ▼ namespace
    clusterRole.yaml             cofig-map.yaml                    namespace.yaml
    deployment.yaml              pods.yaml                     ▼ node-exporter
    service.yaml                 services.yaml                     daemonset.yaml
▼ alert-manager              ▼ kube-state-metrics                  service.yaml
    config-map.yaml              cluster-role-binding.yaml     ▼ prometheus
    deployment.yaml              cluster-role.yaml                 clusterRole.yaml
    service.yaml                 deployment.yaml                   config-map.yaml
▼ grafana                        service-account.yaml              deployment.yaml
    config-map.yaml              service.yaml                      rules.yml
    deployment.yaml          ▼ mqtt-exporter                       service.yaml
    service.yaml                 deployment.yaml                 start.sh
                                 service.yaml                    trace.info
```

Figure B.16: Generated files

B.3 Framework deployment

The generated code and YAML manifests are used to deploy and configure the framework. The IoT applications, tools, database, adaptation engine, and other components are deployed in pods using an orchestrator such as Kubernetes or K3S.

The requirements to deploy and run our framework are listed below.

- Kubernetes (v1.23.8 or later) or K3S (v1.23.8+k3s1 or later) orchestrator to manage the node cluster. We suggest K3S, a lightweight Kubernetes distribution built for IoT and edge computing.
- kubectl (v1.23.8 or later).

Once you have configured the cluster, you must run the *start.sh* script found inside the files built by the code generator. This script will automatically deploy all the tools in pods using *kubectl*. To run the script execute the following commands.

1. For Linux:
   a) Set the script executable permission by running chmod command.
   ```
           sudo chmod 777 start.sh
   ```
   b) Execute the shell script.
   ```
           ./start.sh
   ```
2. For Windows 10/11:
   a) Install WSL or Windows Subsystem for Linux.
   b) Execute the shell script.
   ```
           bash start.sh
   ```

The time it takes to deploy the framework depends on the number of applications modeled (it could be minutes). To verify the deployment, you can execute the following commands from the master node.

- To check the status of monitoring and adaptation tools such as kube-state-metrics, prometheus, alert-manager, adaptation engine, etc.

```
    kubectl get pods -n monitoring
```

- To check the status of IoT applications modeled in the input model.

```
    kubectl get pods
```

Note: if the IoT system model does not have adaptation rules involving sensors, then the *mqtt-exporter* will not be deployed. This will not interfere with the execution and operation of the framework.

Finally, you will be able to access the Prometheus and Grafana user interface to configure dashboards and view system status and adaptation rules in real time. From the browser, enter the following url.

- Prometheus: http://«ip-master-node»:30000
- Alert Manager: http://«ip-master-node»:31000
- Grafana: http://«ip-master-node»:32000

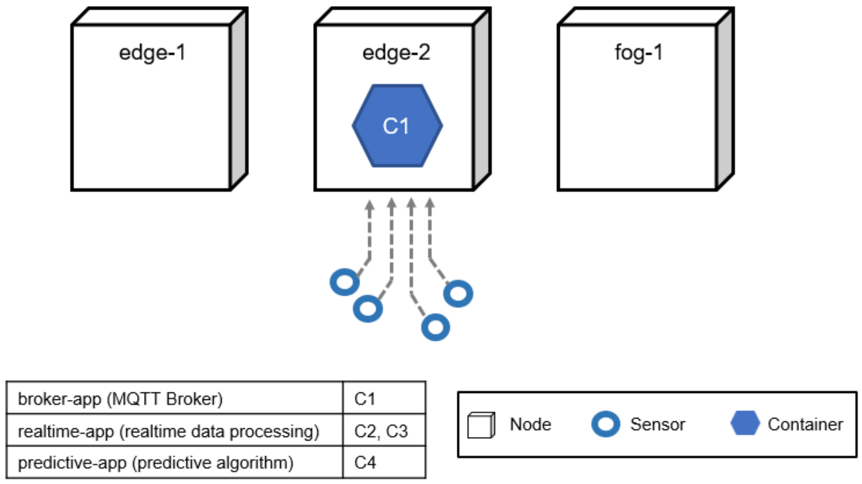
Appendix C

Modeling an IoT System for the Self-adaptation Evaluation

In this appendix we present the modeling of the experiment scenario (IoT system) to perform the self-adaptation validations of our approach discussed in Chapter 7.2.

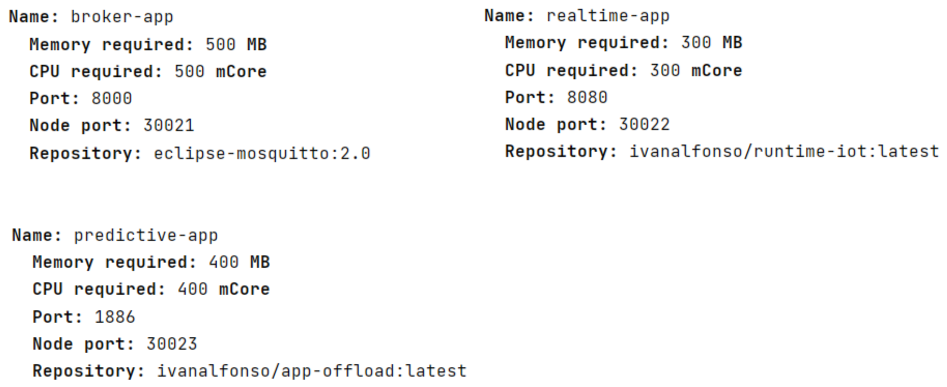
C.1 General Test Scenario

To test the three architectural adaptations, we have designed the test scenario shown in Figure C.1. The modeling of the applications, the edge-2 node, sensors, and the MQTT broker (using our DSL) are presented in Figures C.2, C.3, C.4, and C.5 respectively. The modeling of the edge-1, fog-1 nodes, and the adaptation rules change according to the type of adaptation tested.



broker-app (MQTT Broker)	C1
realtime-app (realtime data processing)	C2, C3
predictive-app (predictive algorithm)	C4

Figure C.1: General test scenario for adaptations

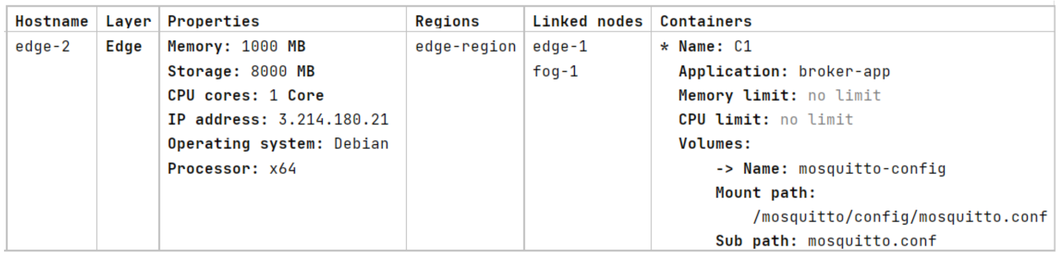


```
Name: broker-app
  Memory required: 500 MB
  CPU required: 500 mCore
  Port: 8000
  Node port: 30021
  Repository: eclipse-mosquitto:2.0
```

```
Name: realtime-app
  Memory required: 300 MB
  CPU required: 300 mCore
  Port: 8080
  Node port: 30022
  Repository: ivanalfonso/runtime-iot:latest
```

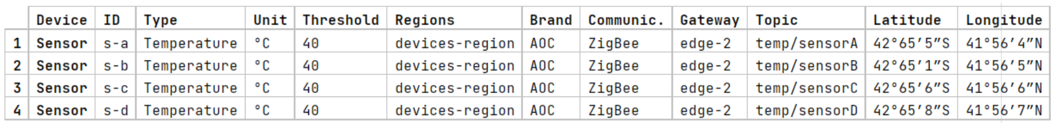
```
Name: predictive-app
  Memory required: 400 MB
  CPU required: 400 mCore
  Port: 1886
  Node port: 30023
  Repository: ivanalfonso/app-offload:latest
```

Figure C.2: Applications modeling (validation scenario)



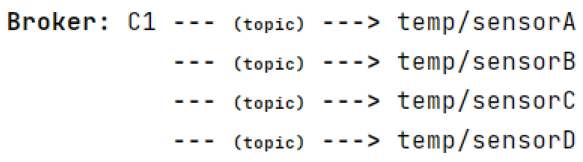
Hostname	Layer	Properties	Regions	Linked nodes	Containers
edge-2	Edge	**Memory:** 1000 MB **Storage:** 8000 MB **CPU cores:** 1 Core **IP address:** 3.214.180.21 **Operating system:** Debian **Processor:** x64	edge-region	edge-1 fog-1	* **Name:** C1 **Application:** broker-app **Memory limit:** no limit **CPU limit:** no limit **Volumes:** -> **Name:** mosquitto-config **Mount path:** /mosquitto/config/mosquitto.conf **Sub path:** mosquitto.conf

Figure C.3: *edge-2* node modeling (validation scenario)



	Device	ID	Type	Unit	Threshold	Regions	Brand	Communic.	Gateway	Topic	Latitude	Longitude
1	Sensor	s-a	Temperature	°C	40	devices-region	AOC	ZigBee	edge-2	temp/sensorA	42°65′5″S	41°56′4″N
2	Sensor	s-b	Temperature	°C	40	devices-region	AOC	ZigBee	edge-2	temp/sensorB	42°65′1″S	41°56′5″N
3	Sensor	s-c	Temperature	°C	40	devices-region	AOC	ZigBee	edge-2	temp/sensorC	42°65′6″S	41°56′6″N
4	Sensor	s-d	Temperature	°C	40	devices-region	AOC	ZigBee	edge-2	temp/sensorD	42°65′8″S	41°56′7″N

Figure C.4: Sensors modeling (validation scenario)

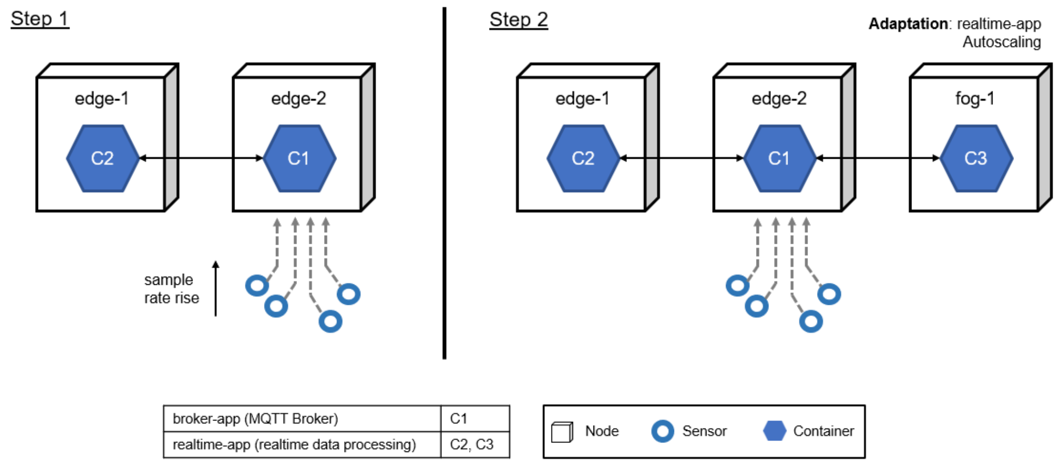


```
Broker: C1 --- (topic) ---> temp/sensorA
           --- (topic) ---> temp/sensorB
           --- (topic) ---> temp/sensorC
           --- (topic) ---> temp/sensorD
```

Figure C.5: MQTT broker modeling (validation scenario)

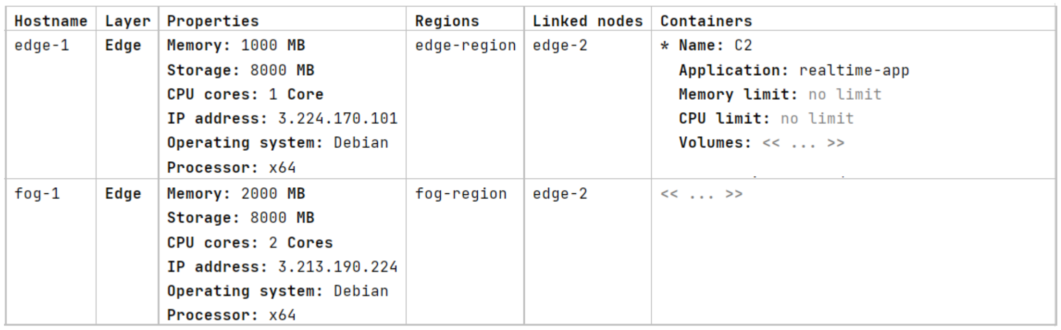
C.2 Scale Adaptation

Figure C.6 shows the test scenario for testing the Scaling adaptation. In addition to the general test scenario, it was necessary to model the nodes edge-1, fog-1, and the adaptation rule to test the self-adaptation of the system. For this, Figures C.7 and C.8 show the modeling of the nodes (including the C2 container), and the specified adaptation rule.



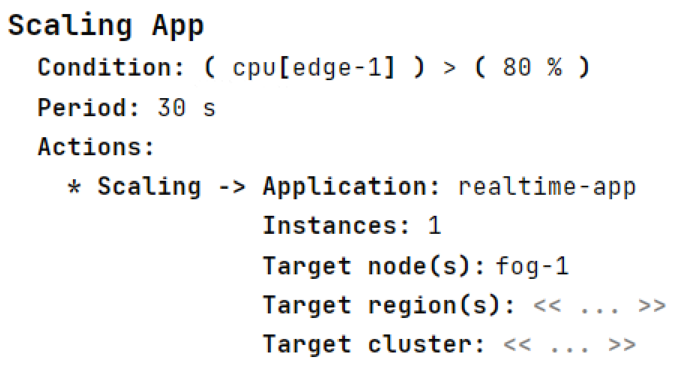
broker-app (MQTT Broker)	C1
realtime-app (realtime data processing)	C2, C3

Figure C.6: Scaling scenario



Hostname	Layer	Properties	Regions	Linked nodes	Containers
edge-1	Edge	Memory: 1000 MB Storage: 8000 MB CPU cores: 1 Core IP address: 3.224.170.101 Operating system: Debian Processor: x64	edge-region	edge-2	* Name: C2 Application: realtime-app Memory limit: no limit CPU limit: no limit Volumes: << ... >>
fog-1	Edge	Memory: 2000 MB Storage: 8000 MB CPU cores: 2 Cores IP address: 3.213.190.224 Operating system: Debian Processor: x64	fog-region	edge-2	<< ... >>

Figure C.7: Scaling scenario nodes modeling

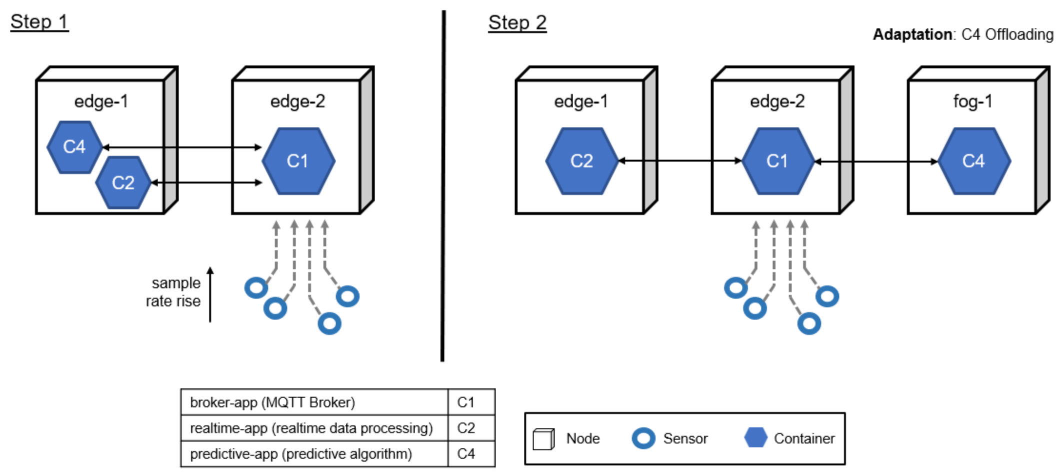


```
Scaling App
  Condition: ( cpu[edge-1] ) > ( 80 % )
  Period: 30 s
  Actions:
    * Scaling -> Application: realtime-app
                 Instances: 1
                 Target node(s): fog-1
                 Target region(s): << ... >>
                 Target cluster: << ... >>
```

Figure C.8: Scaling rule modeling

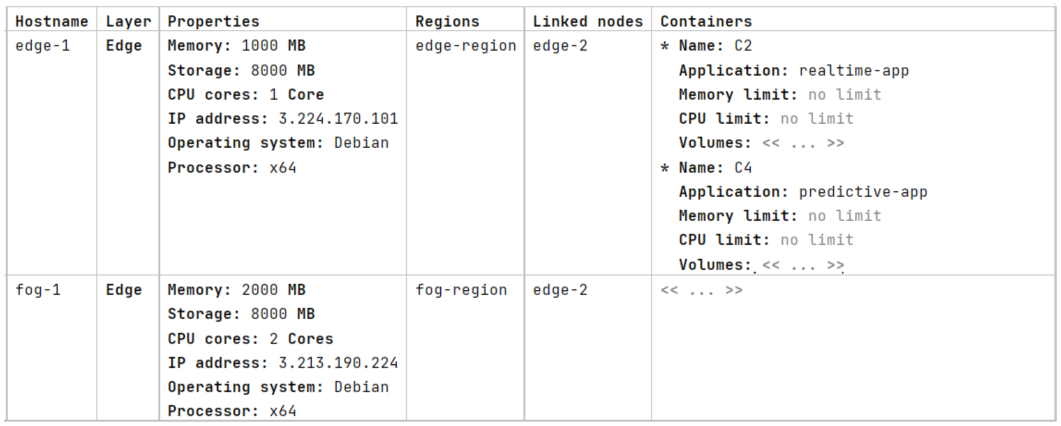
C.3 Offload Adaptation

Figure C.9 shows the test scenario for testing the Offloading adaptation. In addition to the general test scenario, it was necessary to model the nodes edge-1, fog-1, and the adaptation rule to test the self-adaptation of the system. For this, Figures C.10 and C.11 show the modeling of the nodes (including the *C2* and *C4* containers), and the specified adaptation rule.



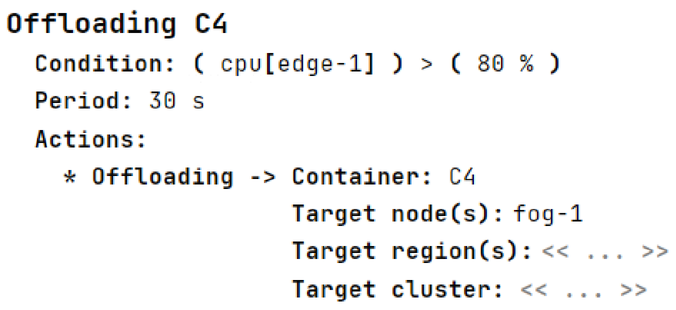
broker-app (MQTT Broker)	C1
realtime-app (realtime data processing)	C2
predictive-app (predictive algorithm)	C4

Figure C.9: Offloading scenario



Hostname	Layer	Properties	Regions	Linked nodes	Containers
edge-1	Edge	Memory: 1000 MB Storage: 8000 MB CPU cores: 1 Core IP address: 3.224.170.101 Operating system: Debian Processor: x64	edge-region	edge-2	* Name: C2 Application: realtime-app Memory limit: no limit CPU limit: no limit Volumes: << ... >> * Name: C4 Application: predictive-app Memory limit: no limit CPU limit: no limit Volumes: << ... >>
fog-1	Edge	Memory: 2000 MB Storage: 8000 MB CPU cores: 2 Cores IP address: 3.213.190.224 Operating system: Debian Processor: x64	fog-region	edge-2	<< ... >>

Figure C.10: Offloading scenario nodes modeling

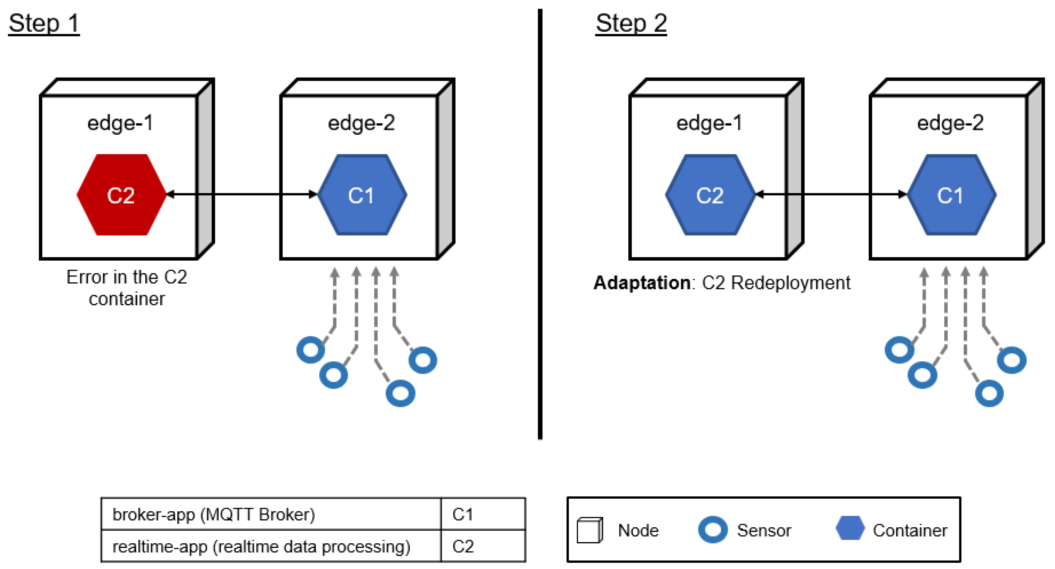


```
Offloading C4
  Condition: ( cpu[edge-1] ) > ( 80 % )
  Period: 30 s
  Actions:
    * Offloading -> Container: C4
                    Target node(s): fog-1
                    Target region(s): << ... >>
                    Target cluster: << ... >>
```

Figure C.11: Offloading rule modeling

C.4 Redeployment Adaptation

Figure C.12 shows the test scenario for testing the Redeployment adaptation. The scenario for testing this adaptation is the same as for testing Scaling. The difference is the adaptation rule specified, and the stimulus to generate the failure. The stimulus for this scenario is to intentionally generate a failure and the adaptation rule is shown in Figure C.13.



broker-app (MQTT Broker)	C1
realtime-app (realtime data processing)	C2

Figure C.12: Offloading scenario

```
Redeploying C1
  Condition: ( unavailability[C1] )
  Period: 20 s
  Actions:
    * Redeployment -> Container: C1
```

Figure C.13: Redeployment rule modeling