MSE | MASTER OF SCIENCE IN ENGINEERING

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts Western Switzerland

Master of Science HES-SO in Engineering
Av. de Provence 6
CH-1007 Lausanne

Master of Science HES-SO in Engineering

Orientation : Computer science (CS)

Cloud-Native IoT Reference Architecture with Arm SystemReady

Made by

Xavier Clivaz

Under the direction of
Prof. Fabien Vannel
Hepia/inIT/CoRES

Sion, HES-SO//Master, 2024

Information about this report

Contact Information

Author:	Xavier Clivaz Master Student HES-SO//Master Switzerland
Email:	xavierclivaz@hotmail.com

Declaration of honor

I, undersigned, Xavier Clivaz, hereby declare that the work submitted is the result of a personal work. I certify that I have not resorted to plagiarism or other forms of fraud. All sources of information used and the author quotes were clearly mentioned.

Place, date: Sion, February 9, 2024

Signature: 

Acknowledgements

This master's thesis would not have been written over five months without the support of a number of people. It's time to thank all these contributors, whether they are closely involved in the project or are far removed from the digital world.

- First of all, I would like to thank my family and especially my parents, without whom I wouldn't be able to fulfil all my dreams. So I'm always able to fulfil my potential in my daily life. It's the strength of spirit I've inherited that will always enable me to move forward despite the obstacles I encounter.
- I would also like to express my deep gratitude to my supervisor, Mr Fabien Vannel, who was always available at any time. His sound advice was invaluable throughout this work, making a significant contribution to achieving the desired end result.
- Finally, I would like to express my sincere thanks to everyone at 56K.Cloud. Their exceptional team brought joy and good humour to the organisation. Their invaluable contribution in sharing their knowledge has been invaluable when obstacles have been encountered. In addition, my professional experience has been enriched by my involvement with the company, and I've had the opportunity to take part in a number of enriching events. I'm also grateful for the opportunity to take the AWS Certified Cloud Practitioner course.

Abstract

Cloud-Native IoT Reference Architecture with Arm SystemReady is an open-source project designed to meet the integration challenges between embedded systems and the cloud. This architecture facilitates the automatic provisioning of a cloud infrastructure and the integration of a fleet of embedded systems during their initial start-up. Designed for use with AWS, it focuses on core components while leveraging Arm processors and SystemReady certifications. The project aims to streamline collaboration between embedded systems engineers and cloud professionals, allowing them to focus on their end products. By following Cloud-Native best practice, the continuous integration and delivery tools guarantee a robust, functional architecture on an Arm-based embedded system that is SystemReady certified. The ultimate aim is to encourage the creation of a community around this architecture, enabling engineers to adopt it and apply it easily to their projects. In addition, a demonstration is included involving the deployment of a cloud infrastructure on AWS and the automatic configuration of cloud services for seamless interaction with an embedded system, plus data visualisation accessible via a minimal web interface.

Key words: Arm, Arm SystemReady, AWS, cloud, Cloud-Native, IoT, reference architecture

Contents

List of Figures

1 Introduction

For a long time, embedded systems evolved in silos, isolated from each other. However, technology is evolving, and it's time to unite these systems with the world of the cloud. Hardware devices have had limitations in terms of resources, although designs have improved these aspects depending on the use case. However, physical limitations remain. The emergence of cloud computing, without being for this problem, has offered partial solutions to these challenges.

Cloud computing offers a number of advantages, not least the freeing up of hardware resources to optimise the operation of embedded systems. These devices, often limited in their complex processing capabilities, can now benefit from the computing power of the cloud. Storage, another constraint, also finds infinite solutions in cloud computing, offering virtually unlimited storage capacity. However, the benefits of cloud computing are not limited to these aspects. They also include the constant scalability of infrastructures, with independent services that can be easily connected and the possibility of vertical and horizontal scaling.

This development has led to growing interest from engineers looking to interconnect their embedded systems with the cloud, a practice now at the heart of the **I**nternet **o**f **T**hings (IoT). Faced with increasingly ambitious projects, coupled with the growing integration of artificial intelligence, engineers are looking to cloud computing environments to eliminate constraints and facilitate interconnection. To simplify the use of a cloud infrastructure, companies are offering cloud platforms providing a multitude of services tailored to different needs (AWS, Microsoft Azure, etc.). This approach allows engineers to focus more on their business projects rather than spending time maintaining and securing the underlying infrastructure. In this context, the integration of embedded systems with the cloud is becoming crucial to meeting the growing demands of the IoT.

1.1 56K.Cloud

56K.Cloud is a company established in Sion (Valais) in 2018, specialising in the provision of cloud-related services. It offers digital transformation to businesses looking to optimise their processes while reducing their costs. As a consultancy, it is committed to demystifying the cloud for customers who may find it difficult to grasp the concepts of the ever-changing digital world. Its collaborative partnerships enhance its skills, enabling it to deliver complete solutions to customers. In order to share its vision of the cloud more widely, 56K.Cloud also has additional premises in Winterthur (Zurich).

Figure 1.1 56K.Cloud company logo [1]

Digital transformation is a global process aimed at integrating digital technologies into all aspects of a company, from its business model and organisational culture to its processes and operations.

As a consultant, 56K.Cloud introduces customers to the Cloud-Native approach and explores concepts such as DevOps and containerisation.f DevOps and containerisation.

1.2 Problem

As part of its core business focused on cloud solutions, 56K.Cloud is also active in the IoT sector. Recently, the company identified a significant problem relating to collaboration between engineers specialising in embedded systems and those from the cloud domain. There is a growing demand for engineers to establish fast and efficient links between embedded devices and cloud infrastructures. However, there is currently a lack of reference architectures for deploying an infrastructure while directly provisioning a fleet of embedded systems. A reference architecture is a preconceived model for a particular domain. It provides a solid foundation on which other architectures can be built, simplifying the work of software developers in that specific domain.

This problem stems from the time required and the complexity for engineers to establish the connection between hardware and a cloud infrastructure. Because of specialist skills in two distinct areas, a cloud professional will need time to understand and implement an operating system capable of connecting to an infrastructure, while an embedded systems expert will need to acquire knowledge of cloud infrastructure by following Cloud-Native best practice.

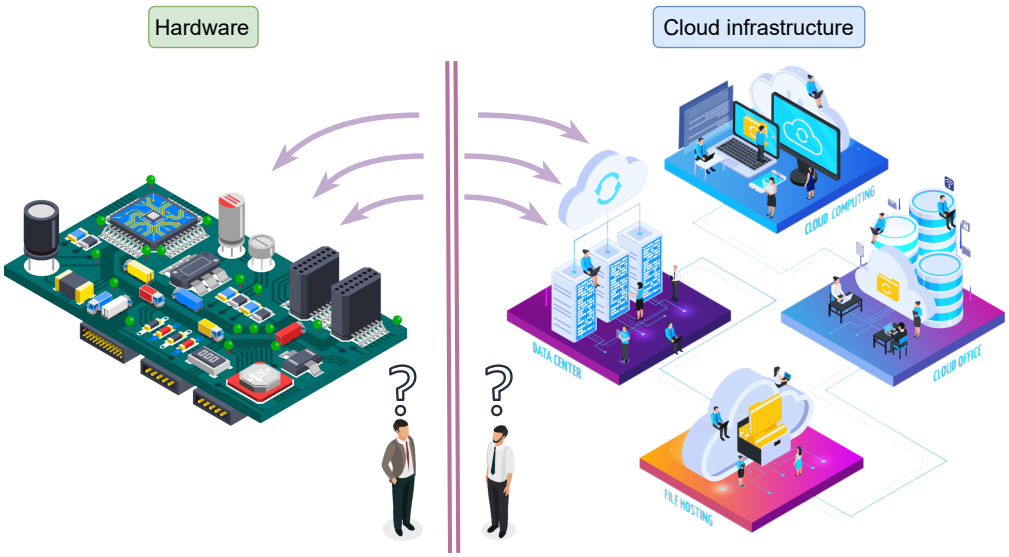


Figure 1.2 The problem of linking the hardware to the cloud infrastructure [2]

The need to find a solution arose when engineers began to approach electronic board manufacturers to design equipment offering services for linking to a cloud infrastructure. However, these manufacturers, who specialise in hardware, are generally not inclined to devote the time to developing such solutions. The main players affected are hardware engineers, who struggle to meet new customer requirements, and cloud engineers, who have to invest time in integrating embedded concepts. This learning phase can quickly become time-consuming.

Although a few solutions are beginning to emerge, most of them remain proprietary. To date, there is no open source reference architecture capable of providing a comprehensive response to the needs of a wide range of products.

1.3 Objectives

The main objective of this project is to design a reference architecture facilitating the deployment of a cloud infrastructure while ensuring the automated provisioning of a fleet of embedded systems. The emphasis is on the ability to provision devices automatically when they are first started up. The infrastructure must contain the essential minimum of services to make the architecture as universal as possible. Since 56K.Cloud is a partner of processor manufacturer Arm, the use of embedded systems equipped with Arm processors and certified Arm SystemReady is favoured, in line with the quality standards set by Arm. The Arm SystemReady programme guarantees that an operating system and the following software layers can function correctly in their processors.

The infrastructure will be deployed on the AWS platform, 56K.Cloud's partner cloud provider. A fundamental aspect of this project is to promote open source, encouraging the formation of a community around this reference architecture. The aim is to enable engineers specialising in embedded systems and the cloud to adopt this architecture easily, while focusing on the development of their end products. Cloud-Native best practices are integrated into the development process, while the use of **C**ontinuous **I**ntergration (CI) and **C**ontinuous **D**elivery (CD) tools is crucial to ensure a robust architecture. Compatibility must be taken into account to ensure the functionality of the architecture on different embedded systems with Arm processors and Arm SystemReady certification.

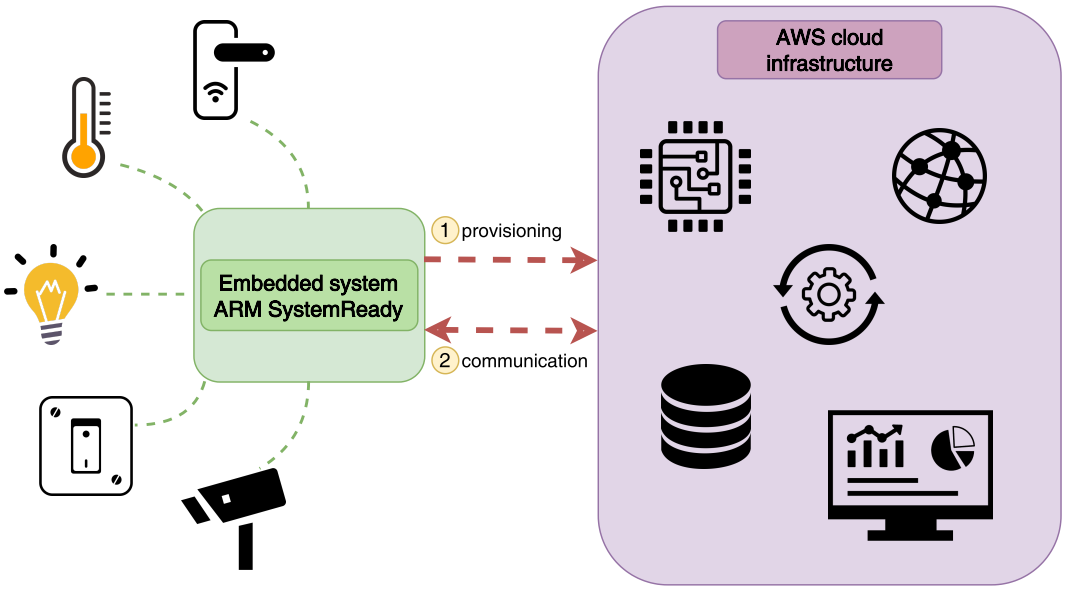


Figure 1.3 Overview of objectives

In parallel with the reference architecture, a small demonstration project is to be based on it. This phase involves deploying a cloud infrastructure on AWS, with the development of a few applications to be deployed on the embedded systems. The data must pass between these two environments, while offering the possibility of viewing it from a web interface.

1.4 Structure of this report

The 2 chapter (Analysis) provides important definitions related to the project and examines the state of the art by looking at similar existing solutions.

The 3 chapter (Design) provides a comprehensive overview of the reference architecture, encompassing cloud infrastructure, embedded systems integration, the CI/CD process and applications.

The 4 chapter (Implementation) details the implementation of each component of the reference architecture, highlighting the tools used and their functionalities.

The 5 chapter (Validation) analyses the results of the tests carried out on the reference architecture, validating its performance.

The 6 chapter (Project methodology) sets out the project plan, the methodologies adopted and describes the tools used as well as the associated training.

The 7 chapter (Conclusions) closes the thesis by presenting the current state of the research, the obstacles encountered and the future directions of the project.

2 Analysis

The analysis begins by detailing the fundamental concepts addressed throughout the project. It then reviews the state of the art in terms of the various tools, methods and technologies used in cloud computing and IoT.

Contents

2.1 Definitions

Before delving into the subject of this work, it is important to clarify the definitions of frequently used terms.

2.1.1 Reference architecture

A reference architecture is a solution model in a specific domain. It must be built so that an architecture can be established on its foundations, to make the task of software developers easier.

The aim is to generalise a solution that shows how it works from an overview. It must be possible to observe the relationships and their interactions between the multiple components of the application based on the reference architecture. There are different layers of abstraction depending on the field of application. A high level of abstraction means that it will be possible to understand the solution through more abstract elements such as the general components that will consolidate the application, for example. A low level of abstraction means that the solution will be more precise and certainly more specific to a use case. It will be possible to find detailed relationships between the services contained in the general components.

2.1.2 Cloud computing

The National Institute of Standards and Technology (NIST) [3] has standardised the definition of cloud computing as follows:

> *Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. [3]*

Their document [3] adds that this model of cloud consists of three service models and four deployment models. The service models are as follows:

- **I**nfrastructure **a**s **a** **S**ervice (IaaS) : The capability offered to the consumer consists of providing processing, storage, network and other fundamental computing resources where the consumer can deploy and run arbitrary software, which may include operating systems and applications. The consumer manages the operating systems, storage and applications.
- **P**latform **a**s **a** **S**ervice (PaaS) : The capacity provided to the consumer consists of deploying on the cloud infrastructure applications created or acquired by the consumer using programming languages, libraries, services and tools supported by the provider. The consumer manages only the applications and storage.
- **S**oftware **a**s **a** **S**ervice (SaaS) : The consumer has the option of using the supplier's applications running on a cloud infrastructure. The consumer manages nothing of the infrastructure apart from any configuration of the applications.

Figure 2.1 *Comparison of different service models [4]*

The deployment models are as follows [3] :

- **Private cloud** : The cloud infrastructure is made available for exclusive use by a single organisation comprising multiple consumers. It may be owned, managed and operated by the organisation, by a third party or by a combination of both, and it may exist on the organisation's premises or off-site.
- **Community cloud** : The cloud infrastructure is reserved for the exclusive use of a specific community of consumers from organisations with common concerns. It may be owned, managed and operated by one or more organisations within the community, by a third party or by a combination of such organisations, and it may exist on or off premises.
- **Public cloud** : The cloud infrastructure is made available to the general public for open use. It may be owned, managed and operated by a business, educational institution or government organisation, or a combination of these. It is located on the provider's premises.
- **Hybrid cloud** : The cloud infrastructure is a composition of two or more distinct infrastructures (private, community or public) which remain unique entities, but which are linked by a standardised or proprietary technology that allows the portability of data and applications.

2.1.3 Cloud-Native

The Cloud-Native approach is a software development methodology focused on the design, implementation and management of applications, integrated in the cloud. It requires cloud. Cloud computing environments to run workloads efficiently.

This approach enables infrastructure to be deployed in private, public or hybrid cloud environments, promoting the development of modern applications with easy scalability. Flexibility is further enhanced by techniques for decoupling the different services within an application. Cloud computing also offers reliability and durability, with redundancy ensuring security in the event of an incident, with migration of an application's execution. It also facilitates real-time and recurring updates.

The main benefits include high efficiency, reduced costs and high availability. Applications can exploit optimal resources according to their needs, and the pay-as-you-go model means that you only pay for the actual use of resources, independently of other aspects such as maintenance and hardware security. Availability is ensured by the diversity of resources provided by the different cloud service providers.

2.1.4 Internet of Things

Internet **o**f **T**hings (IoT) refers to the interconnection between physical objects and the internet. These objects can encompass a variety of items, from light bulbs to medical devices, and many more, mainly in the home automation and medical fields.

The IoT aims to establish connections between various objects and applications, helping to create an automated world known as the IoT network. This growing expansion requires significant resources, made available thanks to cloud computing. Fast and secure global accessibility is essential. Several communication technologies are used to ensure reliable connectivity.

2.2 Problem formulation

The general question for analysis is: "Which technologies are best suited to the development of this reference architecture".

To answer this question, a number of key aspects need to be considered :

- Technology
- Environment
- Tools
- Certification

With this in mind, the analysis begins by tracing the evolution of the Cloud-Native approach. Next, the integration of IoT in the context of cloud computing is studied through various research studies. An exploration of the IoT services offered by the various cloud service providers is undertaken to discern their specific features. Given that the reference architecture is based on **I**nfrastructure **a**s **C**ode (IaC), an investigation is carried out into the IaC tools available. Finally, particular attention is paid to Arm SystemReady certification.

2.3 History of the Cloud-Native approach

Cloud-Native is a term that has been around for several years. Figure 2.2 shows the evolution of this term since 2006. The boom took place around 2016. It is undoubtedly due to the birth of Docker (2013) [5] and Kubernetes (2014) [6]. In 2015, the Cloud Native Computing Foundation [7] was created with the aim of making the Cloud-Native approach ubiquitous [8].

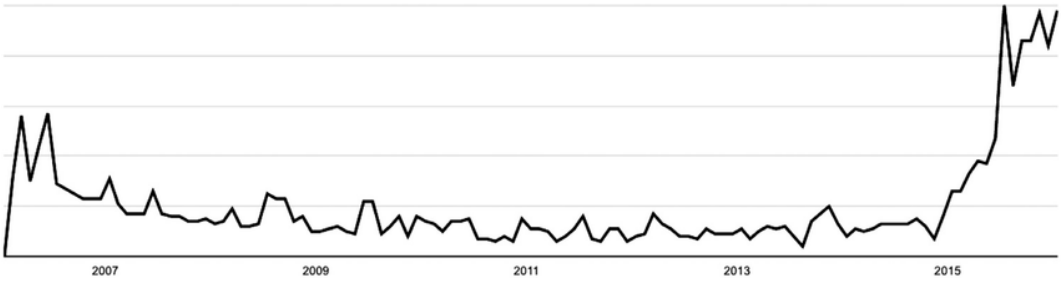


Figure 2.2 Google trends (01.01.2006 until 22.05.2016) of term Cloud-Native [9]

Before all this, there were only on-site data centres. Generally speaking, each company had its own servers running in its own data centre. This meant that servers were always set up for a specific application. [10]

2.3.1 The virtualisation

The first change came in the 2000s with the virtualisation of servers, although this has been around since the 1960s. When a new application had to be developed, new physical servers had to be bought. With virtualisation, it is no longer necessary to buy new ones. It is possible to run several applications on the same server using virtualisation, which limits the hardware resources for each application (figure 2.3). [10]

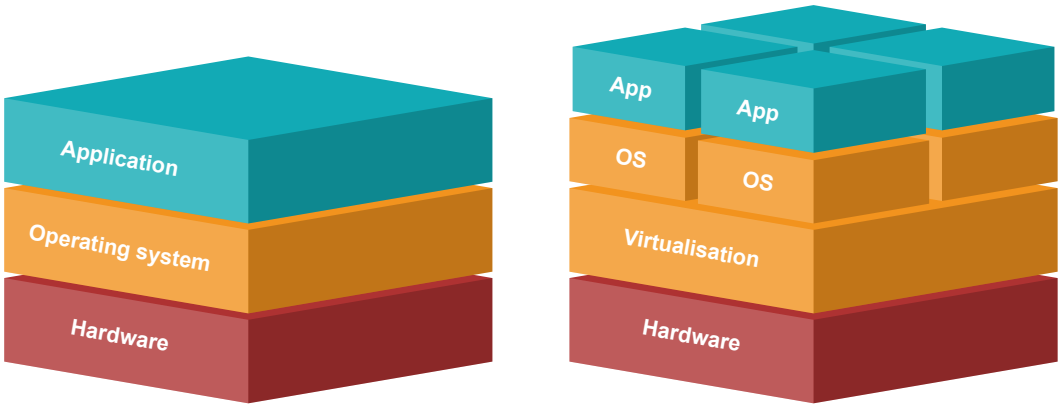


Figure 2.3 Traditional server (left) and virtualisation (right)

2.3.2 The hybrid

Despite this, a minimum of one server had to be purchased for an application to work. There were also resource limits for a certain number of applications. Unfortunately, not everyone could afford to buy new servers. In 2006, AWS launched three web services, Elastic Compute Cloud (EC2), Simple Storage Service (S3) and Simple Queue Service (SQS), to allow organisations and individuals to use Amazon's IT infrastructure on an

as-needed basis at basic prices [11]. This initiative stems from a restructuring of the Amazon platform for better scalability and this has made it possible to sell virtual servers as a service [12]. By extension, the myth of making money from servers not in use for the majority of the year has been debunked by Benjamin Black, co-founder of the EC2 [12]. This is where the world of cloud really began. Kratzke and Quint confirm this in their scientific article [9]. The purpose of the EC2 service, which is still used today, is to offer a virtual machine in the cloud. This is a cloud computing environment for running workloads. It is now possible to migrate your infrastructure from the database to the cloud. The corresponding expression is "Lift and Shift" [13]. These were undoubtedly the first so-called Cloud-Native approaches [10]. However, this term only appeared in papers for the first time in 2012 [9]. These were two conference papers proposing solutions for Cloud-Native applications [14, 15]. Organisations have learned a lot about only paying for what you use. From there, there's a move to hybrid where companies are still using their on-premises servers alongside servers in the cloud. [10]

2.3.3 All to the cloud

A new barrier has now been crossed. It's no longer a question of doing hybrid, but of transferring the entire infrastructure to the cloud. This was made possible when cloud service providers such as AWS, Microsoft Azure, Google Cloud and many others developed several services for all types of use. The reason there are different services is quite simply to decouple application functionality as much as possible. This is known as a microservices architecture. The risk of an entire application being interrupted in the event of a problem is much less likely. However, some organisations still have applications that are several decades old and have a monolithic architecture. It is not necessarily possible to decouple functionalities. That's why they work on a hybrid basis. [10]

2.3.4 The Cloud-Native approach

If we ask several engineers today about the definition of the Cloud-Native approach, we will find varying interpretations. There are currently three main schools of thought on this subject [10]. The first group considers that an approach is Cloud-Native when all the workloads run in the cloud. A minority within this group argue that you can be partially Cloud-Native, i.e. have one complete application deployed in the cloud while maintaining another application locally. However, others feel that this is still a hybrid approach. A second group says that to be truly Cloud-Native, you need to fully exploit the capabilities offered by the cloud. This means not only using the basic services of a cloud provider, but also making full use of the advanced features on offer, such as serverless functions. Some even consider that a company has to be born in the cloud to be truly Cloud-Native, a phenomenon that is increasingly being observed, with companies launching their first applications directly in the cloud without ever having used on-premises servers. [10]

In 2018, a brief history of cloud application architectures described the term Cloud-Native as follows :

> *Cloud infrastructures (IaaS) and platforms (PaaS) are built to be elastic. Elasticity is understood as the degree to which a system adapts to workload changes by provisioning and de-provisioning resources automatically. Without this, cloud computing is very often not reasonable from an economic*

point of view. Over time, system engineers learned to understand this elasticity options of modern cloud environments better. Eventually, systems were designed for such elastic cloud infrastructures, which increased the utilization rates of underlying computing infrastructures via new deployment and design approaches like containers, microservices or serverless architectures. This design intention is often expressed using the term Cloud-Native. [16]

2.4 Integrating cloud computing with IoT embedded systems

With technology advancing at a rapid pace, today's world appreciates help in making everyday life better [17]. Embedded systems that have become increasingly connected now offer this service. However, since they need to attend any location for any domain and task, they need to be small for better integration. As a result, resources are limited for computing operations, data storage, data processing and so on. The built-in security and low energy consumption that these devices must also ensure are not enough to allow them to move forward. This is why cloud computing needs to be integrated to fill the gaps left by IoT [17]. Cloud of Things is one of the paradigms given to the combination of these two technologies in 2014 [18].

2.4.1 Overview

A model for integrating cloud computing into an embedded system is shown in figure 2.4.

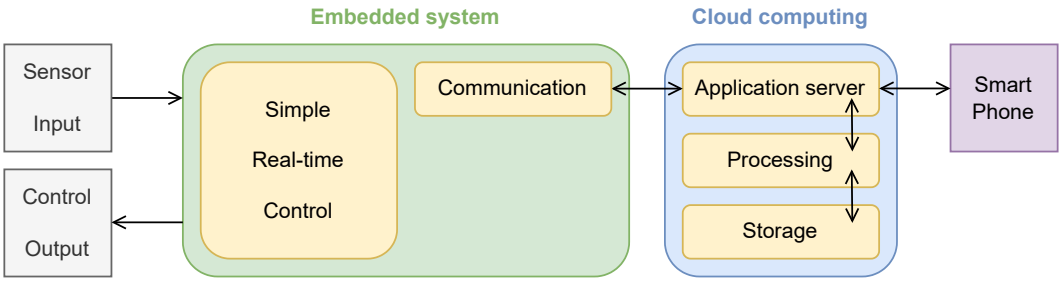


Figure 2.4 Model for integrating cloud computing into an embedded system [19]

The main idea is to use only what is necessary on the IoT device. In other words, all computing and storage functions should be moved to the cloud. The device should only be linked to the input and output peripherals using a simple controller and establish communication with the cloud. For the integration to work, the embedded system must have an internet connection. Cloud computing would contain the core of the application as well as various services to perform analysis, processing and calculation operations, and store data in real time. From there, it would also be possible to view the data from a smartphone. This approach is a solution that was thought up by Furuichi and Yamada in 2014 [20]. It offers a number of advantages, such as reduced energy consumption by eliminating large workloads and reduced size by freeing up electronic components. The cloud also makes it easy to scale up if necessary. The application can then be highly scalable. [19]

Furuichi and Yamada wanted to use a project to prove that their suggested approach worked. They developed a traffic jam detection system that recognises car licence plate numbers along with their location and time. For a plate number to be recognised from a raw image, image processing is carried out using machine learning. They found that this application, split between an IoT device and a cloud computing environment, scored much better than the same application managed entirely on a laptop or embedded system. The evaluation looked at cost, battery life, performance, scalability and reliability. [20]

2.4.2 Reference architecture

A reference architecture has been developed to support IoT objects in cloud computing (figure 2.5). The Standards Development Organization has decided to make this a standard by launching the Cloud Standards Customer Council programme to drive forward the adoption of cloud computing. In this architecture, various aspects are taken into account: scalability, security, reliability and protection of privacy. [21]

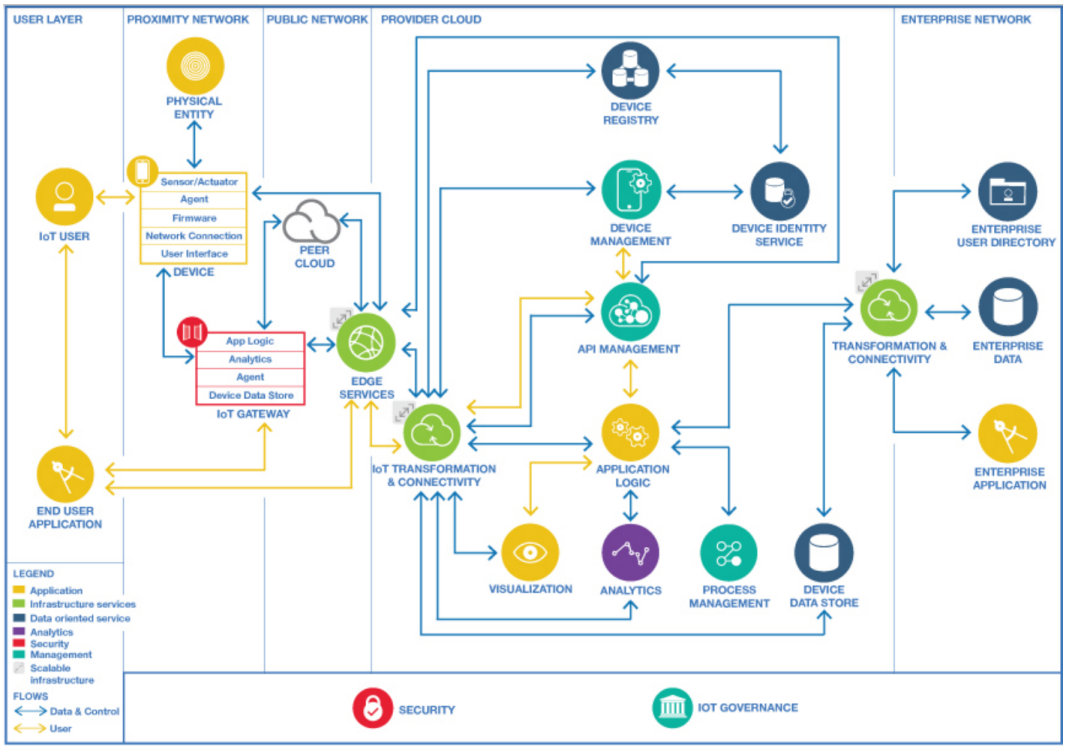


Figure 2.5 Cloud Customer Reference Architecture for IoT [21]

The architecture proposed in figure 2.5 is made up of components and their relationships. They are separated under a three-tier architecture model, called 3-Tier (presentation, logic, data). There is edge-tier, the platform tier and the enterprise tier.

The edge-tier contains the "Proximity Network" and "Public Network" parts. It represents the place where data is collected. It is collected from an IoT device in a proximity network. The data will either be sent to an IoT gateway or directly to the cloud provider.

The cloud provider tier manages the entire logical part of the application. It includes the collection, processing and analysis of data flows. Each device is also managed in this tier, as is its identity, for example.

The enterprise tier contains the "Enterprise Network" part. It includes the enterprise data and the application. Data from cloud computing can be stored in enterprise data.

2.4.3 Existing problems and solutions linked to integration

Integrating IoT and cloud computing poses a number of challenges. They are to be found both on the hardware side and in the cloud. [18]

There are many communication protocols. There are protocols for every type of use. On the IoT side, for example, there are Zigbee [22], Bluetooth and **B**luetooth **L**ow **E**nergy (BLE) [23], LoRaWAN (long range) [24], WiFi, Thread [25] and the latest Matter protocol [26]. In an IoT network, there can be a multitude of sensors linked to a gateway. Interoperability is a problem in this case. Since 2022, version 1.0 of the Matter protocol has been published to address this problem. However, few devices are yet compatible. In research projects, Zigbee is very popular [17, 18, 27, 28]. Two main technologies are used for communication between the IoT gateway and the cloud: **M**essage **Q**ueuing **T**elemetry **T**ransport (MQTT) [29] and **Co**nstrained **A**pplication **P**rotocol (CoAP) [30]. Several projects have used CoAP for resource reasons [31, 32, 33]. It is a protocol dedicated to constrained devices with small amounts of memory [30], using UDP. It is also suitable for devices that need to bind to web services as it uses the **RE**presentational **S**tate **T**ransfer (REST) model. REST is an architectural style for web services [31]. It would have been possible to use the **S**ervice-**O**riented **A**rchitecture (SOA) style, but it is less suited to this use case [31]. Service-based architecture styles allow a high degree of decoupling from the application. This is an advantage in the event of a problem with a service, so as not to completely freeze the application. Finally, MQTT is specially designed for IoT [29]. It uses the TCP transport protocol. Cloud provider AWS adopts the MQTT protocol in its IoT sector [34]. A communication between an Arm microcontroller and AWS proved that this protocol works [33]. In fact, the data passing through is often in **J**ava**S**cript **O**bject **N**otation (JSON) format [19, 28, 35]. This is a lightweight data representation syntax for storing and exchanging textual information [28]. There is also the **H**yper**T**ext **T**ransfer **P**rotocol (HTTP) communication protocol. It would be suitable for updating IoT devices. It allows more bytes to be sent per packet, which increases the update speed. Furthermore, in industry, HTTP is generally less constrained by firewalls than MQTT or CoAP [36].

A lot of energy is consumed in IoT devices. The reason for this is the increasing amount of data being transferred to the cloud. Energy needs to be managed efficiently, for example by setting a sleep mode. If this is not possible, natural energy could be used to provide power, such as solar, wind or vibration. [18]

Resource allocation is another debate. Each IoT object may have a different function. They do not necessarily require the same resources in the cloud. What's more, unexpected events could occur requiring a greater workload. The requirements could not necessarily be predictable. Even if they were, the cost of increasing unused resources for most of the time would be a loss. Cloud service providers offer a number of solutions to this problem. For example, AWS offers a service that balances the load in the cloud

computing environment [37]. The combinatorial auction approach is popular for resource allocation in the cloud [38]. It must satisfy **Q**uality **o**f **S**ervice (QoS) constraints and maximise the cloud provider's profit. Provider and user profit, resource utilisation and quality of service are the dominant performance factors. Research has succeeded in optimising this approach [38]. Nevertheless, as soon as a new IoT node is added, a sample of data could be sent to the cloud to set up the minimum resources required. [18]

Identity management in an internet network is very important to ensure that each device is unique. Given the number of IoT devices today, IPv6 addressing would be the optimal solution. It would be sufficient to support this type of network. However, deployment remains problematic. The coexistence of IPv6 with IPv4 is not so easy [18]. There are three possible techniques for cooperation between the two [39]. The IoT device may support both types of addressing. Alternatively, IPv6 packets must be encapsulated in IPv4 packets, or a **N**etwork **A**ddress **T**ranslation (NAT) must be available that translates both types of packet.

The discovery of new IoT devices to be integrated into the cloud needs to be managed. At any time, a device can be part of a service and leave it. It is also necessary to monitor its status once it is connected and keep it up to date. AWS has developed a solution to this which must be installed on IoT devices. From there, the devices can be easily integrated. In addition, it is possible to perform **O**ver-**T**he-**A**ir (OTA) updates [40] and monitor the state of devices [41]. Azure also provides these different services [42, 43]. An update method was designed in this area in 2022 [36]. It is based on the standard update architecture for IoT [44], proposed by the Internet Engineering Task Force (IETF).

Cloud computing environments must respond correctly to data flows, whether large or small. The urgency with which certain data is transmitted is also an integral part of this QoS. QoS is assessed in terms of bandwidth, delay, packet loss rate and delay in the transmission of data packets (jitter). [18]

Another point to consider is where the data is stored. Depending on the volume of data, it is best to store it in the physical location closest to the user. Sensitive data must also be managed in accordance with each country's data protection laws. Fortunately, cloud providers offer the option of choosing the region where data is stored. [18]

Security seems to be an issue in integration. In 2016, there was a lack of trust in cloud service providers and a lack of awareness of Service-Level Agreements (SLAs) [45]. A Service-Level Agreement is a document defining the provision of cloud services and the responsibilities of the provider and the customer. However, when data is transferred to the cloud, the location of databases is not always transparent. Distributing data over several locations, while ensuring high availability, also increases the chances of sensitive data being leaked. A distributed system of this kind is exposed to various potential attacks, such as SQL injections, cross-site scripting attacks and many others. Significant vulnerabilities can also be exposed, including session hijacking and virtual machine evasion. In addition, the computing power constraints associated with connected objects limit the application of public key cryptography to all layers of the system. Two years later, in 2018, in an attempt to partially address this problem, two security models have

been proposed using two encryption algorithms (AES and RSA) [27]. They can be used to integrate IoT and cloud computing. However, they remain sub-optimal and could be the subject of future research.

Unnecessary data communication is a current trend. IoT devices generate all kinds of data. It would be interesting to have an IoT gateway that manages traffic by letting data circulate only when it is necessary and only that which is necessary [18]. This would save energy and make better use of network and cloud resources. An architecture for this approach has been presented [46].

2.5 IoT cloud platforms

A number of projects have created their own cloud platform [47]. The University of Glasgow decided to create its own cloud computing environment based on a Raspberry Pi [48], called PiCloud [49]. It emulates every layer of a cloud stack, from resource virtualisation to network behaviour. A fleet of 56 Raspberry Pi devices was able to consolidate this cloud platform. This project is ideal for education, thanks to its low cost. Another study, based on the previous one, carried out the same style of cloud platform, but with 300 Raspberry Pi devices. Called the Bolzano Raspberry Pi [50], this cloud computing environment incorporates several **N**etwork **A**ttached **S**torage (NAS) as storage units. However, these are traditional cloud infrastructures.

Cloud infrastructures have been adapted for the use of IoT. What differentiates them from traditional infrastructures is the processing of data generated by events in real time [51]. There are now a large number of public cloud service providers for IoT. There are also a number of open source IoT solution providers. The main ones today are AWS, Microsoft Azure and Google Cloud [52].

AWS [53] provides a cloud platform with over 200 services. Resources are distributed across 32 regions worldwide. It claims to be the most widely adopted platform in the world. Products include: compute, storage, databases, analytics, networking, mobile, development tools, management tools, IoT, security and enterprise applications. It offers its services on demand and payment is on a pay-as-you-go basis. Through a survey published in 2019 on the various platforms [54], a solution dedicated to IoT, called AWS IoT, was summarised. This service aims to collect, store and analyse data from devices. Amazon FreeRTOS and AWS IoT Greengrass are made available by AWS IoT, facilitating the development of embedded applications on these devices. For embedded systems, both Arm and x86 architectures with Linux are supported [55]. For device management, there is AWS IoT Core, which facilitates device connectivity with cloud services. This can be complemented by AWS IoT Things Graph to visualise the data emitted by devices. In terms of data processing, the platform provides AWS IoT Analytics, offering developers a convenient way to analyse the data generated. The results of this process can be routed to other devices or systems via the AWS IoT Events service [54] service. The aim of a thesis project was to define an IoT architecture for connecting Arm microcontrollers to AWS [33]. An example of a smart home application was made on this architecture. A machine learning service was used in the cloud computing to make predictions from data sent from the IoT devices. The company 56K.Cloud is also carrying out a few projects linking Arm devices with a cloud infrastructure AWS. It typically uses the AWS IoT Greengrass Core solution to run applications on embedded systems.

Microsoft Azure [56] is another public cloud provider. Launched in 2008, it offers more than 200 cloud products and services. Its data centres are located in more than 60 regions around the world. The company operates in the healthcare, finance, public sector, manufacturing and retail sectors. It says it invests a billion dollars a year in security to protect customer data. As far as IoT is concerned, it offers a solution called Azure IoT. This solution includes several services. Azure IoT Hub is a service that establishes connections, administers and develops the ability to manage billions of IoT devices, from the peripheral to the cloud. Azure IoT Central provides a user interface and APIs for connecting and managing large-scale embedded systems. Data is visualised using Azure Time Series. Azure Sphere offers security to protect IoT devices. Physical device spaces can be replicated using the Azure Digital Twins service. System logic can be managed by Azure IoT Edge. For the development of integrated applications, there is Azure RTOS. [56]

Google Cloud [57] is one of the world's leading cloud providers. It currently distributes its resources in 39 regions across the globe. Launched in 2008, the provider is trying to keep up with the competition by offering around a hundred services for all types of business. However, Google has announced that it will be pulling out of the IoT business in 2023. The Google Cloud spokesperson said that the needs of their customers could be better served by specialist IoT partners [58].

There are other cloud providers offering IoT services. Some of these are mentioned in figure 2.6, where a comparison is made.

IoT platform	Open source	Protocols	Data store	Push notification	Trigger	Visualisation
AWS IoT	No	MQTT, HTTPS	Yes	Yes	Yes	Yes
Azure IoT	No	MQTT, AMQP, HTTPS	Yes	Yes	Yes	Yes
IBM Watson IoT Platform	No	MQTT, JMS	Yes	Yes	Yes	Yes
Kaa Enterprise IoT Platform	Yes	MQTT, HTTPS	Yes	Yes	Yes	Yes
ThingsBoard	Yes	MQTT, CoAP, HTTP, LwM2M, SNMP	Yes	Yes	Yes	Yes

Figure 2.6 Comparison of IoT cloud platform providers (sources from providers)

Looking at the back end of platforms, they would have an advantage in using containerisation over traditional virtual machines [59]. It is seen as a lightweight virtualisation solution, with greater flexibility. In addition, containers are proving particularly suitable for solving the platform issues commonly associated with PaaS services in the cloud. This includes aspects such as application packaging and coordination [59]. Docker is the most popular container solution [5]. As for container orchestration, there's Kubernetes, an open source system [6].

2.6 Cloud infrastructure tools

When you want to host applications in the cloud, you need to deploy a cloud infrastructure. It is perfectly possible to build the infrastructure manually. All you have to do is visit a cloud provider's web page and select the resources you need from the services on offer. However, this infrastructure could be deployed several times. Fortunately, there are tools that can automatically deploy cloud infrastructures, called **I**nfrastructure **a**s **C**ode (IaC). The automatic deployment of cloud infrastructures corresponds to provisioning. These tools use templates containing a description of the infrastructure based on executable code or a configuration file. They save time on deployment and ensure that the implementation policy behind the infrastructure is always the same (security, rules, etc.). In addition to provisioning, IaC tools are capable of updating and deleting an infrastructure. There are universal provisioning tools for different cloud providers and there are integrated tools for each provider. [60]

2.6.1 Integrated IaC tools

Having mentioned the AWS and Microsoft Azure providers in section 2.5, each of them has its own IaC tool reserved for use with its platform. Google Cloud is not mentioned in this section due to the cessation of IoT activities.

At AWS, there are two IaC tools. AWS CloudFormation [61] is the first. AWS **C**loud **D**evelopment **K**it (CDK) [62] is the second for defining and building infrastructure in the AWS cloud environment. AWS CDK gives users the flexibility to define infrastructure in distinct programming languages in the form of an application, with imperative syntax. It uses AWS CloudFormation as the deployment engine and allows users to define the infrastructure using programming idioms to model the system design. The CDK application deployment process includes the construction of the defined elements, preparation, validation and synthesis of the deployment artefacts. The CDK application is then transferred to the AWS CloudFormation service for real deployment. An overview of the deployment is shown in figure 2.7. The basic elements of CDK applications are called "constructs", and they represent cloud components containing services. Constructs can be nested to create a hierarchy of dependencies called a "construct tree". Ultimately, this hierarchy defines how constructs are synthesized into resources in the final AWS CloudFormation model. In a CDK application, it is possible to define several stacks. These are unique deployment units, each containing a hierarchy of constructs. [60]

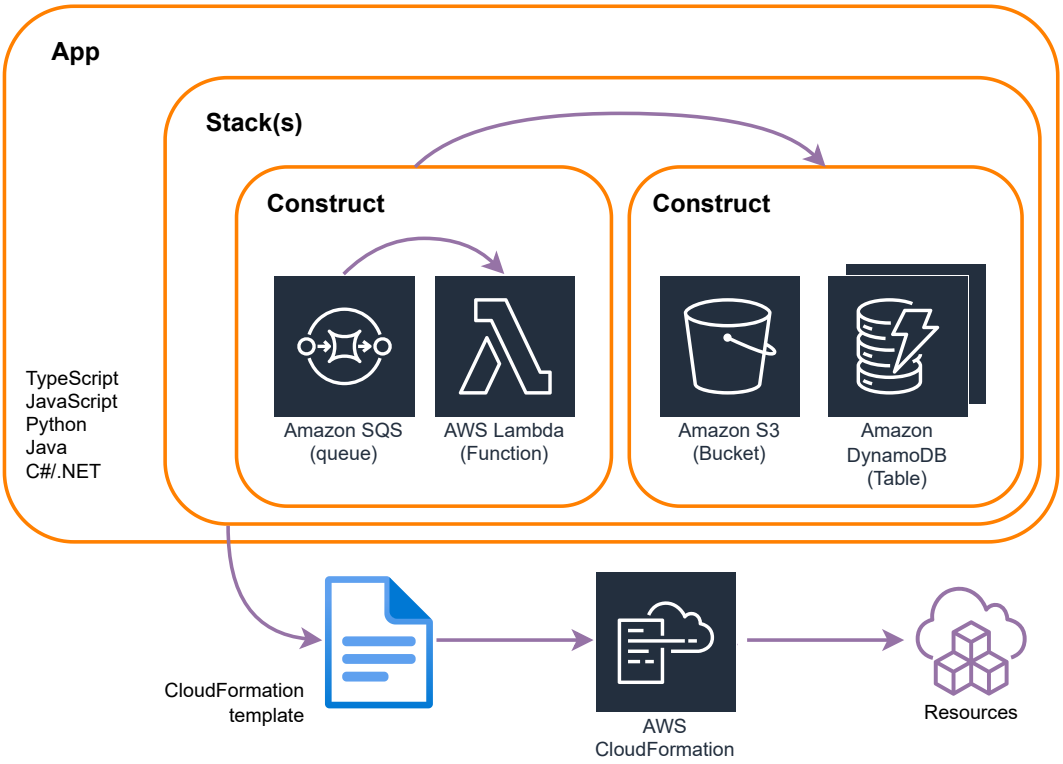


Figure 2.7 Overview of AWS CDK [62] deployment [62]

Microsoft Azure offers the **A**zure **R**esource **M**anager (ARM) deployment and management service [63]. All actions performed on resources by the user go through the ARM manager. It authenticates and authorises them. At the deployment level, there are ARM models, which are JSON files defining the infrastructure and configuration of a cloud environment. JSON uses declarative syntax. Before deployment, the ARM API checks the models for possible errors, guaranteeing successful deployment. ARM models ensure an identical architecture for deployments in different environments, such as development, testing and production, for example. The fewer dependencies there are, the faster ARM can deploy infrastructure resources in parallel. ARM models can be segmented into modular files for easy reuse. A group of resources deployed in the Azure platform can be extracted in the form of an ARM model. [60]

2.6.2 Universal IaC tools

There are several IaC tools compatible with different cloud platforms such as AWS, Microsoft Azure, and many others. The 56K.Cloud company mainly uses Terraform [64] and Pulumi [65].

Terraform is a tool created in 2014 by HashiCorp, written in the Go programming language [66]. It mainly provisions **I**nfrastructure **a**s **a** **S**ervice, but can also deploy **P**latform **a**s **a** **S**ervice and **S**oftware **a**s **a** **S**ervice. Terraform works using two inputs: configuration and state. The desired resources in the cloud infrastructure are described in a configuration file. This file works with the Terraform language using declarative syntax. The state corresponds to the current state of the infrastructure, or in other words the stack. The state is managed by the Terraform API and is stored locally by default. The deployment engine compares the desired infrastructure with the current

state and determines which resources need to be created, updated or deleted. Terraform uses provider plugins to interact with cloud providers. It offers five basic commands for different stages: "init", "validate", "plan", "apply" and "destroy", enabling the infrastructure to be managed efficiently and easily [64]. The "init" command prepares the project directory. The "validate" command checks that the configuration is valid. The "plan" command displays the changes required by the current configuration. The "apply" command creates or updates the cloud infrastructure. Finally, the "destroy" command destroys the cloud infrastructure. An overview of the deployment can be seen in figure 2.8. [60]

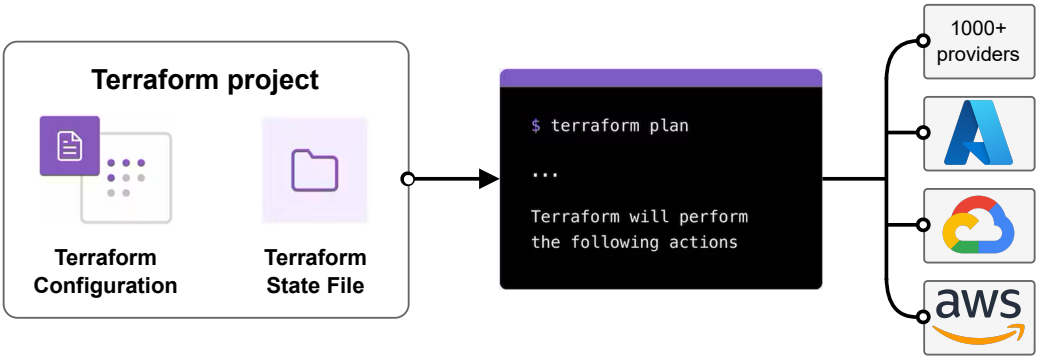


Figure 2.8 Overview of Terraform deployment [64]

Pulumi is an IaC tool similar to Terraform, but open source. It is written in the Go language [67]. Version 1.0 of Pulumi was released in 2019. This tool can be used with several programming languages, such as TypeScript, JavaScript, Python, Go and .NET. It therefore offers developers the possibility of creating cloud infrastructures using standard languages, unlike Terraform. Deployment follows the Terraform model, where a Pulumi application is run to schedule the desired resources in the cloud infrastructure. The deployment engine compares these desired resources with the current state of the stack and determines the necessary actions. The state of the last infrastructure deployment is saved in the Pulumi Cloud platform by default. It can also be saved in other storage spaces (Amazon S3 e.g.). With the Pulumi tool, we're talking about applications, because it uses imperative syntax. Applications describe how the cloud infrastructure should be composed by allocating resource objects whose properties correspond to the desired state. Resources are deployed in parallel if possible, but some resources have dependencies between them. Deployment is managed by the Pulumi CLI. An overview of deployment is shown in figure 2.9. [60]

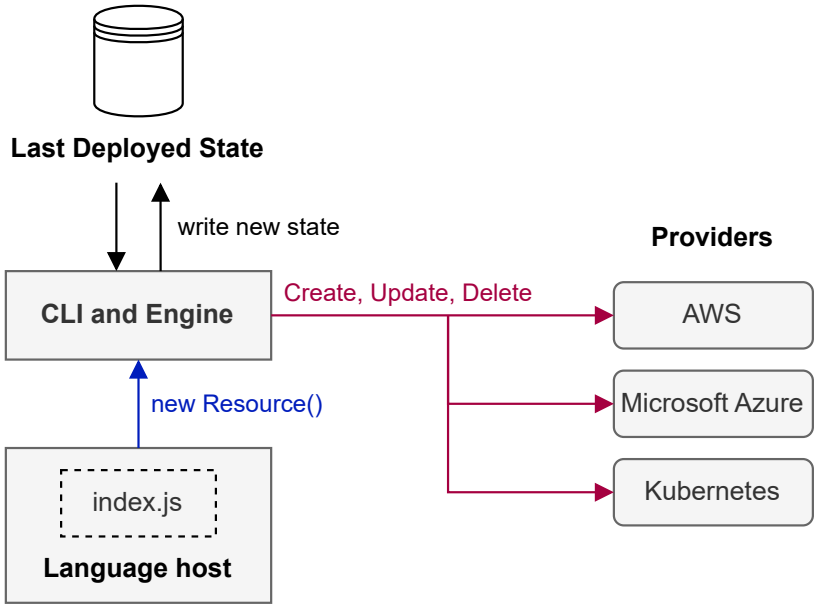


Figure 2.9 Overview of Pulumi deployment [65]

2.6.3 Comparison

Without even using the IaC tools, it is possible to identify a few differences. The tools integrated with the cloud providers cannot be used outside their platform. The other two tools, Terraform and Pulumi, can be provisioned by different providers. One notable point concerns the syntax. Terraform and Microsoft Azure use descriptive syntaxes. These limit the variety of programming languages. Terraform has its own language which you need to familiarise yourself with the first time. AWS and Pulumi use imperative syntax, which leaves the choice of programming languages open.

Focusing on the practical side, other disparities were noted. These stem from a project that was carried out with the aim of comparing the different IaC tools mentioned in this section 2.6 [60]. Terraform has several advantages, including clear documentation, an understandable configuration language, real-time execution plans and the ability to structure the infrastructure by dividing scripts into separate files. Because cloud providers have different services, it is impossible to reuse the same code with each of them. It is possible to maintain the overall architecture of the infrastructure if the entities are clearly separated in the scripts. On the Pulumi side, resources can be separated into classes or functions. Common code between providers is therefore limited to abstractions of classes and functions. Pulumi has developed a tool capable of converting Terraform scripts into Pulumi scripts. It differs from Terraform by using Pulumi Cloud for infrastructure state management by default instead of a self-managed API. This is an advantage of not needing to store state files locally and manage them. Terraform and Pulumi generally use APIs from cloud providers to create resources. One case stands out with Pulumi, when it works with AWS, it uses the AWS SDK service directly, which will create the resources. Concerning the **A**zure **R**esource **M**anager tool, it is less efficient than AWS CDK because of the complexity of the configuration files. AWS CDK offers better readability by being able to divide up the code as required. Despite all this, it should be noted that these four IaC tools were able to provision the project in question. [60]

2.7 Arm SystemReady

Manufacturer Arm has decided to introduce a certification programme called SystemReady [68]. In 2020, as part of Arm's Cassini project, SystemReady was introduced to address the compatibility concerns of users planning to move from x86 architectures to systems based on Arm's processors [69]. Arm defines its programme as follows:

> *Arm SystemReady is a compliance certification program based on a set of hardware and firmware standards: **B**ase **S**ystem **A**rchitecture (BSA) and **B**ase **B**oot **R**quirements (BBR) specifications, plus a selection of supplements. This ensures that subsequent layers of software also 'just work'. The compliance certification program tests and certifies that systems meet the SystemReady standards, giving confidence that operating systems OS and subsequent layers of software just work. [68]*

The Arm BSA specification establishes a hardware basis for system software, including OSs, hypervisors and firmware, based on the 64-bit Arm architecture. This specification ensures that users can install, boot and run generic OSs and hypervisors [70]. The BBR specification complements BSA by defining the basic firmware requirements necessary to support any OS or hypervisor compatible with the BSA specification [71].

A number of advantages were mentioned in relation to the program [68]:

- Compliance allows OSs and workloads to operate seamlessly on different Arm platforms.
- Standardisation of a range of different devices and systems provides a stable basis and a choice of systems for all sectors.
- Compliance with standards inspires confidence in software compatibility, so developers can concentrate on innovation and adding value to their products.
- Compliance simplifies and accelerates time-to-market for partners.
- Compliance makes it easy to identify Arm-based devices thanks to the "SystemReady certified" stamp.
- Compliance makes it easy to deploy and maintain standard firmware interfaces, reducing maintenance costs.
- Compliance reduces the costs associated with adopting a new platform by eliminating custom firmware engineering.

2.7.1 SystemReady certifications

To be able to adapt to different types of devices and markets, Arm has introduced four certifications. The first is SystemReady SR. It ensures the smooth operation of servers or workstations on a Arm chip. It is specially designed for Windows, Linux, VMware and **B**erkeley **S**oftware **D**istribution (BSD) environments. It targets generic out-of-the-box OSs and also supports legacy OSs on new devices. [68]

SystemReady ES is designed to meet the needs of Windows, Linux, VMware and BSD ecosystems based on Arm embedded systems. It also targets generic out-of-the-box OSs and supports legacy OSs on new devices. [68]

SystemReady IR ensures that Linux and BSD work perfectly on Arm embedded systems. It is ideally suited to the IoT sector. It mainly targets the Linux environment, but also custom images (Yocto, OpenWRT, buildroot) and pre-built images (Debian, Fedora, SUSE). [68]

SystemReady LS specifies the correct operation of Linux OSs on Arm chips designed for servers. This certificate is mainly for hyperscalers. Hyperscalers are large cloud service providers. [68]

At present, a multitude of hardware design companies are partnering with Arm to follow the SystemReady standards. Several embedded systems are already certified and on the market. This is the case with the Raspberry Pi 4, for example.

3 Design

The design phase provides an overview of the various components of the reference architecture. It sets out the conceptualisation of the structure of the cloud infrastructure with the integration of multiple embedded systems. An overview of the automation process is presented, as well as an explanation of the general operation of each application included in this architecture.

Contents

3.1 Reference architecture

The reference architecture is divided into two parts: the first concerns embedded systems based on Arm architectures and certified Arm SystemReady with a Linux **O**perating **S**ystem, while the second relates to the cloud and is hosted by the AWS provider. An overview is clearly shown in figure 3.1.

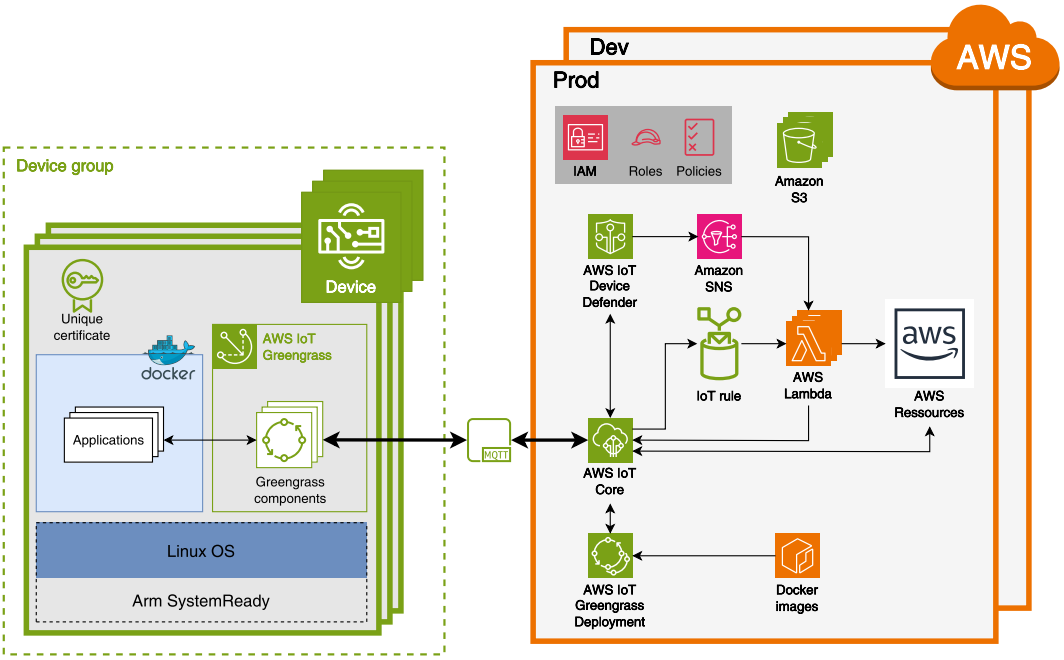


Figure 3.1 Reference architecture overview

3.1.1 Cloud infrastructure

In the cloud, two distinct infrastructures are presented. The first infrastructure is dedicated to development, enabling all kinds of operations to be tested. The second infrastructure is the production infrastructure, encompassing the final product intended for customers. Each infrastructure uses different AWS services.

The central service managing the IoT is AWS IoT Core, which exchanges data with all connected devices. Its preferred communication protocol is MQTT. Provisioned devices are registered in a specific group within this service. Only authorised devices are provisioned. AWS IoT Greengrass Deployment handles the deployment of new Greengrass components or updates to a group of devices. A component, in the context of AWS IoT Greengrass, is a unit of deployment and execution that encapsulates code, dependencies and resources, making it easy to deploy, manage and update IoT applications. In this architecture, Docker components are used, encapsulating containers to offer greater flexibility in software deployment. Docker images are stored in the Amazon **E**lastic **C**ontainer **R**egistry (ECR). Another service, AWS IoT Device Defender, checks daily whether a certificate is about to expire, as each device has a unique certificate enabling a secure connection. If a certificate expires within 30 days, it is replaced by a new one. Other Lambda functions, triggered by rules, can enrich the architecture, as can other AWS resources, depending on the needs of each developer. Storage spaces such as Amazon S3 are used to save the state of the infrastructure, configuration files

for device provisioning and OS images. The configuration files contain a list of the serial numbers of all the IoT devices authorised to be provisioned. If a device is not listed and attempts to be provisioned, access will be denied. If a device is already provisioned and its serial number is removed from the list, its access will be withdrawn and it will be deleted from the cloud. The IAM service is there to assign roles and policies, limiting authorised actions on resources to the strict minimum.

3.1.2 Embedded systems integration

Embedded systems must be based on Arm architecture and be Arm SystemReady certified. The devices are flashed with a custom Linux OS image. On first boot, two software programs are installed : Docker Engine for running Docker containers, and AWS IoT Greengrass Core for provisioning, communicating with AWS IoT Core and managing Greengrass components. This software provisions the device by exchanging the claim certificate with a unique certificate, securing the connection of the device, which is then registered in the device group. Deployment of Greengrass components from the cloud to devices is automated when a new device is associated or a new version of a component becomes available. Applications are contained in Docker containers to ensure optimum portability.

3.2 CI/CD pipeline

The pipeline is made up of several workflows, each dedicated to a specific function. An overview of the various stages in the pipeline is shown in figure 3.2. The pipeline uses CI/CD methods to automate the deployment of the cloud infrastructure, as well as preparing for the integration of embedded systems.

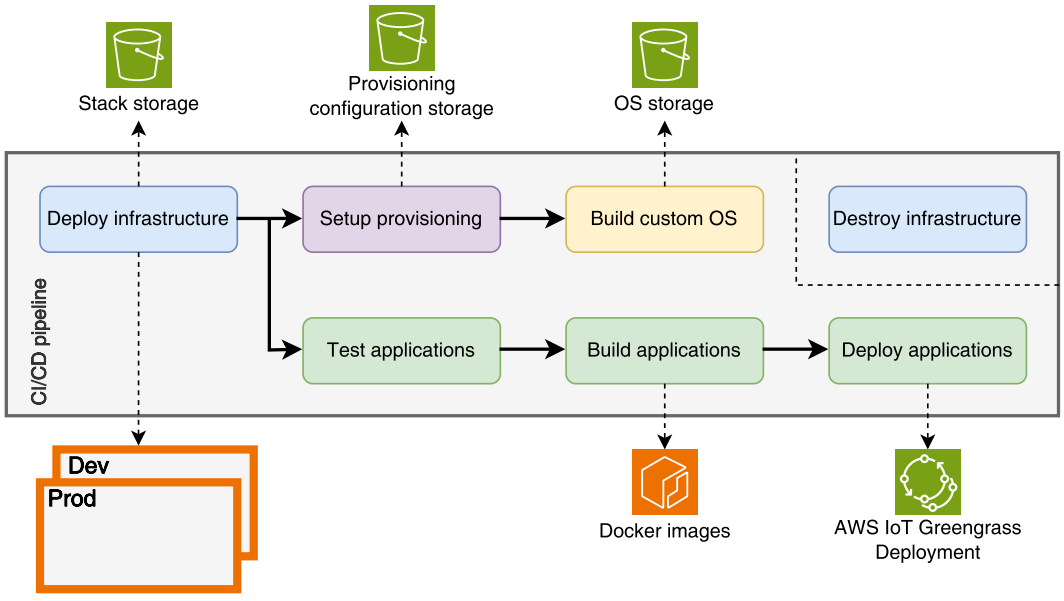


Figure 3.2 CI/CD workflows overview

The first job is to deploy the AWS cloud infrastructure (*Deploy infrastructure*). It can enable the deployment of development or production infrastructure. Since it uses a tool called IaC, the state of the infrastructure must be stored. This flow also enables resources to be updated.

If the entire infrastructure is deployed without error, the provisioning configuration workflow (*Setup provisioning*) takes care of preparing configuration files for provisioning in a storage space. It also manages device control. It checks the list of authorised devices to identify whether a previously provisioned device has been removed from the list. If so, it will be removed from the cloud. It may also be the case that authorised devices have already been provisioned and need to be connected to the new infrastructure. This operation is also checked and carried out.

If the previous step has been successful, the next step (*Build custom OS*) is to prepare and create the OS image with all the files and scripts needed to automatically provision the devices when they start up for the first time. The image must be saved in a storage space that allows it to be downloaded for flashing the devices' SD cards.

In parallel with the previous two workflows, the application workflows are launched. The first (*Test applications*) performs various tests on the applications. It includes security and unit tests.

If the tests have not failed, the next job (*Build applications*) is to build the Greengrass components from the applications. A Docker image is first created for each of the applications, coded in a freely chosen programming language. These images are then linked to their respective components. Finally, the components must be saved in the AWS IoT Greengrass service to be ready for deployment.

The last flow (*Deploy applications*) takes care of deploying applications to the device group. It uses the AWS IoT Greengrass Deployment service.

After all these tasks, there is still the infrastructure dismantling workflow (*Destroy infrastructure*). It is independent and can be launched manually. It is available in the event that a problem occurs and the infrastructure needs to be redeployed.

3.3 Applications

Several applications have been developed. Some are essential to the reference architecture, while others are demonstration applications. An overview of these is shown in figure 3.3.

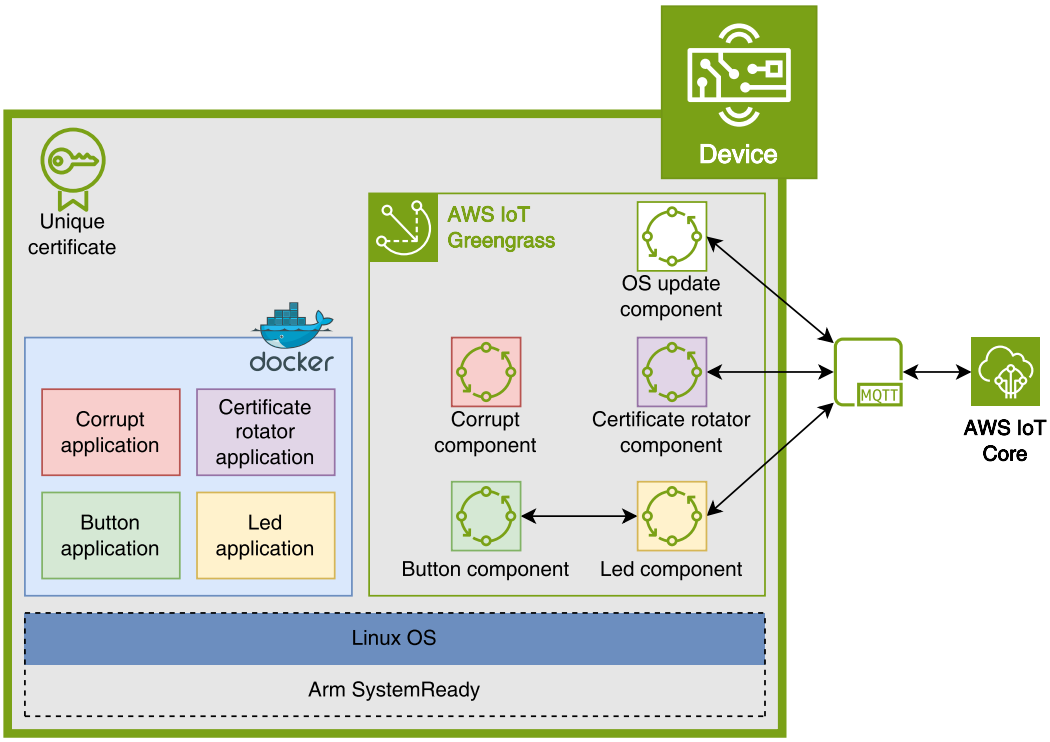


Figure 3.3 Applications overview

3.3.1 Certificate rotation

On embedded systems communicating with AWS, unique certificates are used to establish secure connections via the protocols. Certificates have expiry dates for security reasons. The use of certificates with an expiry date makes it possible to limit the time during which a certificate is considered valid. This helps to reduce the risks associated with compromised private keys and ensures that certificates are regularly renewed, incorporating the latest security standards. It is therefore crucial to update certificates on embedded systems before their expiry date. If a certificate expires, secure connections with cloud services may be interrupted, leading to communication problems. To avoid this, this application must allow certificates to be managed automatically. It works with the AWS IoT Device Defender service, which alerts the application when it is time to change a certificate.

This application runs in a Docker container. An activity diagram illustrates its behaviour in figure 3.4.

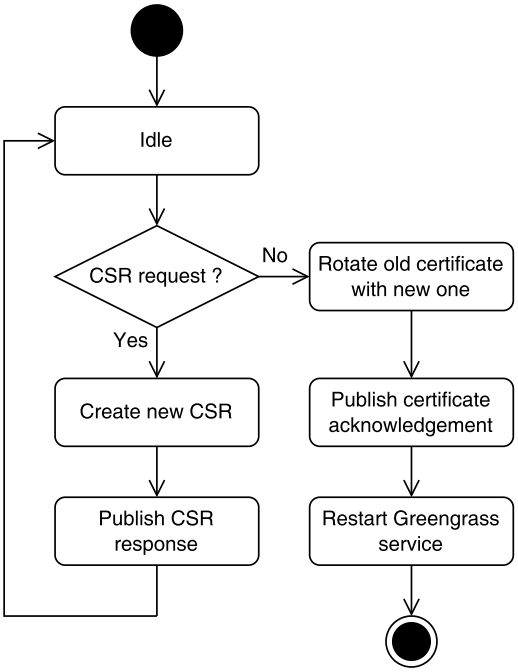


Figure 3.4 Certificate rotator application activity diagram

The general idea is that it is alerted by the AWS IoT Device Defender service when a certificate change is required. The application then creates a new private key and makes a certificate creation request (**C**ertificate **S**igning **R**equest (CSR)) to the Amazon root certificate authority. A signed certificate is returned. This will replace the old certificate and the old private key with the new one. In order for the AWS IoT Greengrass Core service to take the new certificate into account during future communications, this service must be restarted on the device. An overview of the certificate rotation is shown in figure 3.5. It shows the links between the AWS services and the application on the IoT device.

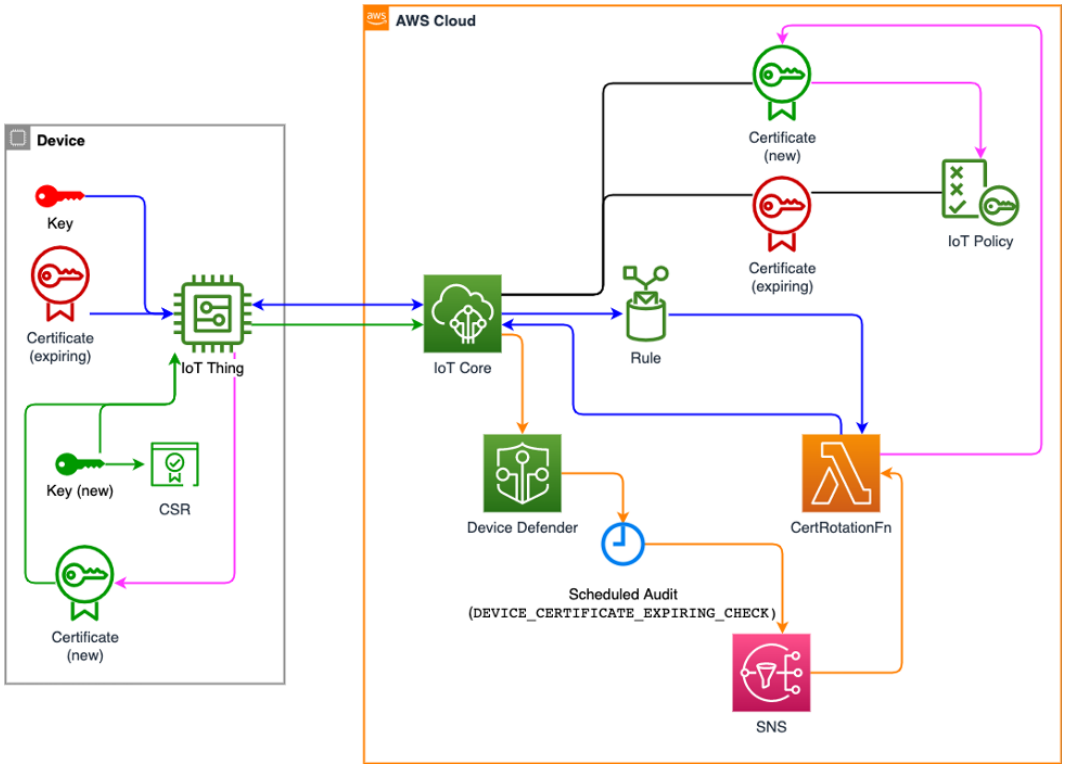


Figure 3.5 Certificate rotation overview [72]

3.3.2 Led application

This application is not essential to the reference architecture. It is used as a demonstration for MQTT communication with AWS IoT. It is an application in a Docker container that blinks a led on an embedded system. The blinking frequency can be adjusted from an application associated with AWS IoT using the MQTT protocol. Blinking can also be enabled or disabled from the button application. Two pieces of data, the blink frequency and the blink state, are transmitted to AWS IoT so that they can be viewed on an interface. This is shown in figure 3.6.

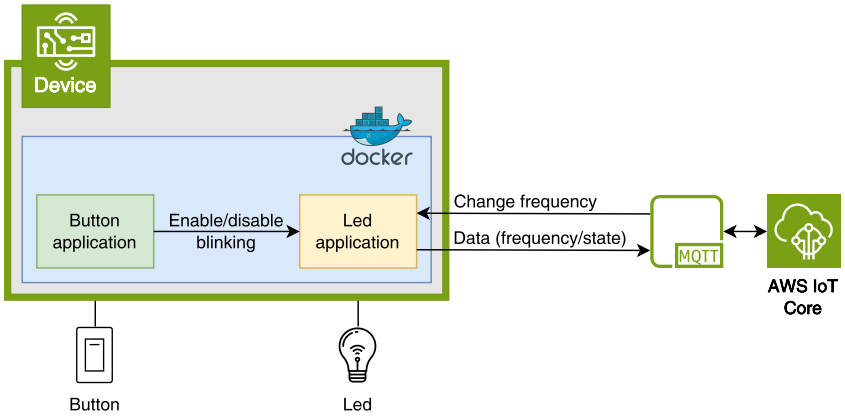


Figure 3.6 Led and button applications interactions

3.3.3 Button application

This application is connected to the led application while retaining its independence. It is designed as a demonstration of local MQTT communication. Also running in a Docker container, it simply interacts with the led application to activate or deactivate the blinking of the led. The user presses a button built into the device to reverse the blinking state. A high-level view is shown in figure 3.6.

3.3.4 Corrupt application

A corrupted application is also developed to examine the behaviour of the device with AWS services. The approach is to run a simple Docker application by deliberately providing an incorrect environment for launching the container.

3.3.5 OS update

The application used to update **O**perating **S**ystem features is essential to the reference architecture. Keeping Linux up to date is essential for security, stability, performance and compatibility. Security updates correct vulnerabilities, minimising the risk of exploitation by malicious software. Bug fixes and performance optimisations improve system stability and efficiency. Updating also ensures compliance with security requirements and adaptation to new features and standards. By keeping a Linux OS up to date, users can ensure that their system runs at optimum performance while keeping pace with technological developments and industry requirements.

This application does not run in a Docker container, but directly in the Greengrass component. The operation of the application must be kept simple. An activity diagram in figure 3.7 shows this.

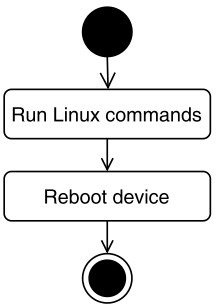


Figure 3.7 OS update application activity diagram

When the component starts up the first time the device is launched or after it has been updated, it will execute the Linux commands that have been written. To enable a complete update, the device will restart automatically. The component executes the Linux commands only once for each new version of the component.

4 Implementation

The implementation sets out the concrete realisation of the reference architecture, detailing the tools used to automate the deployment of the cloud infrastructure and the integration of the embedded systems. Particular attention is paid to the in-depth description of the automated pipeline. The various applications that accompany this architecture are also highlighted.

Contents

4.1 Reference architecture

4.1.1 Cloud infrastructure

An Infrastructure as Code (IaC) tool was selected to orchestrate the efficient deployment of the cloud infrastructure on AWS, and Pulumi was chosen because of its choice of programming language, in this case Python. The use of the same programming language for the implementation of applications and the description of AWS resources provides consistency within the project.

Integrating Pulumi is simple. The resources required are described in Python files. The deployment is managed by the Pulumi CLI, which records the state of the infrastructure in an Amazon S3 bucket. This bucket is located in the same AWS account as the infrastructure, using Amazon S3 to store the state rather than the default Pulumi Cloud platform, thus eliminating external dependencies. Figure 4.1 shows an overview of this part of the implementation.

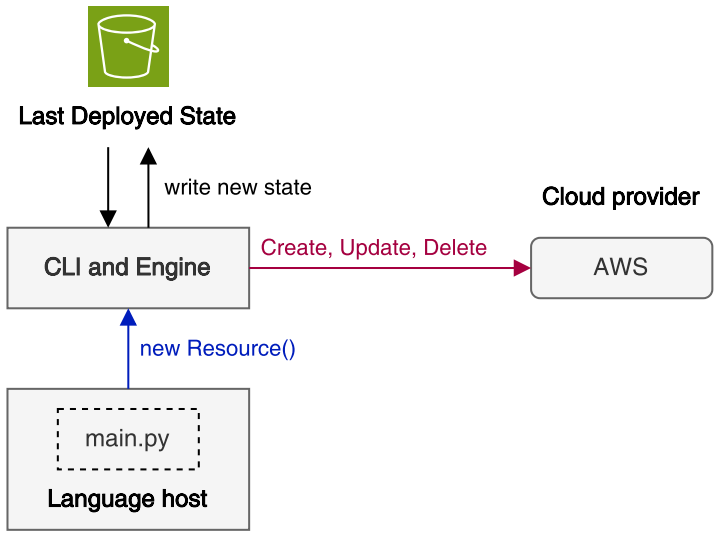


Figure 4.1 Overview of Pulumi's integration into the reference architecture

This cloud infrastructure part is implemented in a specific folder as follows :

```
cloud-infrastructure
    ├── Pulumi.yaml
    ├── Pulumi.dev.yaml
    ├── Pulumi.prod.yaml
    ├── main.py
    ├── iam.py
    └── requirements.txt
```

The main configuration file for the Pulumi tool, *Pulumi.yaml*, includes crucial information such as the name of the stack and the programming language used, in this case Python.

This implementation aims to facilitate the deployment of two distinct infrastructures, namely the development environment and the production environment. Therefore, two additional configuration files, *Pulumi.dev.yaml* and *Pulumi.prod.yaml*, are present to

differentiate these two environments. *Pulumi.dev.yaml* contains parameters such as the AWS development account ID and the deployment region, while *Pulumi.prod.yaml* includes the same parameters with production-specific values. The identifiers of the AWS accounts must be different.

The Python files *main.py* and *iam.py* detail the description of the resources to be deployed. *iam.py* focuses on IAM roles and policies, while *main.py* mainly covers resources related to IoT and other resources.

Finally, the *requirements.txt* file lists the Pulumi libraries to be installed, including the main Pulumi library, which is essential for all IaC projects.

In terms of the Pulumi libraries used, AWS Native, a new preview, uses the AWS Cloud Control API to manage and provision AWS resources, generally aligning with the latest AWS features as they are released. The resources available in this library are based on those defined in the AWS CloudFormation registry. In addition, AWS Classic is another library that is used to fill in resources that are not yet available in AWS Native. This library uses the AWS SDK to manage and provision these resources.

4.1.2 Embedded systems integration

4.1.2.1 AWS IoT Greengrass

Integrating embedded systems with the AWS ecosystem is crucial to reaping the full benefits of cloud computing. To this end, AWS IoT Greengrass was chosen, a powerful solution that facilitates the execution of local processing on IoT devices, while enabling seamless interaction with AWS cloud services. AWS defined this solution as follows :

> *AWS IoT Greengrass is an open source **I**nternet **o**f **T**hings (IoT) edge runtime and cloud service that helps you build, deploy and manage IoT applications on your devices. You can use AWS IoT Greengrass to build software that enables your devices to act locally on the data that they generate, run predictions based on machine learning models, and filter and aggregate device data. AWS IoT Greengrass enables your devices to collect and analyze data closer to where that data is generated, react autonomously to local events, and communicate securely with other devices on the local network. Greengrass devices can also communicate securely with AWS IoT Core and export IoT data to the AWS Cloud. You can use AWS IoT Greengrass to build edge applications using pre-built software modules, called components, that can connect your edge devices to AWS services or third-party services. You can also use AWS IoT Greengrass to package and run your software using Lambda functions, Docker containers, native operating system processes, or custom runtimes of your choice. [73]*

In this implementation, AWS IoT Greengrass Core software is deployed on embedded systems, acting as an intelligent hub that manages interactions with the AWS cloud. It adapts perfectly to Linux OS. Greengrass components, encapsulating the necessary code, dependencies and resources, are then deployed automatically using mechanisms such as AWS IoT Greengrass Deployment.

A component must be installed. *Greengrass nucleus* (aws.greengrass.Nucleus) is a mandatory component and the minimum requirement for running AWS IoT Greengrass Core software on a device. It is configurable to customize and update the AWS IoT Greengrass Core software **O**ver-**T**he-**A**ir (OTA).

4.1.2.2 Communication

In this project, components running on an embedded system use the AWS IoT Greengrass Core **I**nter**P**rocess **C**ommunication (IPC) library, available in the AWS IoT Device SDK, to exchange data with the AWS IoT Greengrass nucleus and other Greengrass components. The IPC interface supports the **M**essage **Q**ueuing **T**elemetry **T**ransport communication protocol. MQTT offers lightweight, asynchronous communication.

Publish/subscribe messaging allows messages to be sent and received in topics. Components can publish messages in topics to communicate with other components, and those who have subscribed to the topic can act on messages received. In this case, the communication is local to the IoT device.

The IPC interface also facilitates the sending and receiving of MQTT messages between AWS IoT Greengrass and AWS IoT Core. Components can publish messages to AWS IoT Core and subscribe to topics to react to MQTT messages from other sources.

4.1.2.3 OTA update

All devices with AWS IoT Greengrass Core software can obtain OTA updates via MQTT.

AWS IoT Greengrass Core software update
This functionality is built into the AWS IoT Greengrass Core software. It is made possible by the *Greengrass nucleus* component and other optional components. The following information is taken from the AWS documentation [40].

A few prerequisites must be met before an update can be carried out. Firstly, the Greengrass device must be connected to the AWS cloud in order to receive the deployment. It must be properly configured with certificates and authentication keys to interact with both the AWS IoT Core and AWS IoT Greengrass. Finally, the AWS IoT Greengrass Core software must be configured and run as a system service.

There are a few things to bear in mind when upgrading. The Greengrass nucleus stops. This stops all the other components present. When the nucleus component is shut down, the device's connection to AWS is lost.

The following table summarises the behaviour of Greengrass updates :

Action	Deployment configuration	Nucleus update behavior
Add new devices to a thing group targeted by an existing deployment without revising the deployment.	The deployment does not directly include Greengrass nucleus. The deployment directly includes at least one AWS-provided component, or includes a custom component that depends on an AWS-provided component or on the Greengrass nucleus.	On new devices, installs the latest patch version of nucleus that meets all component dependency requirements. On existing devices, does not update the installed version of the nucleus.
	The deployment directly includes a specific version of the Greengrass nucleus.	On new devices, installs the specified nucleus version. On existing devices, does not update the installed version of the nucleus.
Create a new deployment or revise an existing deployment.	The deployment does not directly include Greengrass nucleus. The deployment directly includes at least one AWS-provided component, or includes a custom component that depends on an AWS-provided component or on the Greengrass nucleus.	On all targeted devices, installs the latest patch version of the nucleus that meets all component dependency requirements, including on any new devices that you add to the targeted thing group.
	The deployment directly includes a specific version of the Greengrass nucleus.	On all targeted devices, installs the specified nucleus version, including any new devices that you add to the targeted thing group.

In this reference architecture, the version of *Greengrass nucleus* is not specified. However, there is one component provided by AWS (*aws.greengrass.Cli*) and custom components which depend on several components provided by AWS.

Greengrass components update
For custom or public AWS components, the AWS IoT Greengrass Deployment service enables OTA updates.

The prerequisites for an update are the same as for the software update. When an update is deployed, only components with a new version are updated. These components will then be stopped and restarted automatically after being updated. If a deployment error occurs, a rollback is performed and the components resume their course with the old version.

The following table explains two main actions:

Action	Components behaviour
Add new devices to a group of devices targeted by an existing deployment without modifying the deployment.	All the components and dependencies present in the deployment are automatically deployed with their latest version on the new devices. On existing devices, nothing happens.
Create a new version of one or more Greengrass components.	Only components with a new version will be deployed on all devices in the group of devices targeted for deployment.

4.1.2.4 Fleet provisioning

With AWS IoT fleet provisioning, it is possible to securely set up AWS IoT to generate and distribute X.509 device certificates and private keys to each device when they first connect to AWS IoT [74]. These client certificates, signed by the Amazon Root Certificate Authority, are provided by AWS IoT. It is very flexible to define specific device groups, device types and permissions for Greengrass Core devices to be provisioned using fleet provisioning. A provisioning template is used to describe the provisioning process for each device, detailing the device, policy and certificate resources to be created during provisioning. This template is created from Pulumi.

AWS IoT Greengrass offers a dedicated AWS IoT fleet provisioning plugin (*aws.greengrass.FleetProvisioningByClaim*), which facilitates the installation of AWS IoT Greengrass Core software using the resources created by AWS IoT fleet provisioning. This plugin uses claim-based provisioning, where devices use a provisioning claim certificate and private key to obtain a unique X.509 device certificate, along with a private key, enabling regular operations. The claim certificate and private key are integrated as soon as the Linux OS image is created, enabling devices to be activated later when they are connected. The same claim certificate and private key can be used for multiple devices.

To deploy AWS IoT Greengrass Core software with AWS IoT fleet provisioning, configuration of resources in the AWS account is required. These resources include a provisioning template, a claim certificate and a token exchange IAM role. Once these elements have been created from Pulumi, apart from the certificate from the AWS CLI, they can be reused to provision several embedded systems within the same fleet. The token exchange IAM role authorises calls to AWS services from the IoT device.

In this architecture, devices are registered by their serial number to ensure that each one has a unique name.

4.1.2.5 OS image

To enable the devices to be provisioned to integrate automatically on their first boot, a Linux OS image is created specifically for this purpose. The base image used for this architecture is *Raspberry Pi OS Lite* based on the Debian distribution. A directory dedicated to the creation of the image is present as follows:

```
os-image
    ├── raspios-lite.json
    ├── set_os.sh
    ├── provision.sh
    └── config.yaml
```

Image creation is managed by Packer [75], an open source tool for automating image creation. Packer follows a process defined in a configuration file JSON (*raspios-lite.json*).

It works as follows :

1. **Packer configuration** (raspios-lite.json) : The Packer configuration file describes all the steps required to create the image. It specifies the constructors (in this case, the Arm constructor), the actions to be performed, the base image sources and other parameters.
2. **Running Packer** : In this project, Packer runs in a Docker environment. When the container is launched, the tool starts the build process following the specified configuration.
3. **Packer builder ARM plugin** : A plugin [76] to the Packer tool is used to build images specific to the Arm architecture. It uses the Arm build model to create a custom image based on the specification.
4. **Actions** : The actions in the configuration file are carried out to configure the image. This may involve installing software, configuring system parameters, etc. In this case, some system parameters are configured using the *set_os.sh* script. The provisioning script *provision.sh* used to install the necessary software is saved in the image and configured to run the first time the OS is booted. Finally, the claim certificate and its private key are saved in the image.
5. **Build complete** : Once all the build and configuration steps have been successfully completed, Packer creates a custom image in the specified format (.img in this case).

The provisioning file *provision.sh* is a bash script. When it first runs on a device, it first retrieves the device's serial number, so that the device is registered in AWS IoT under that name. The script then installs the Docker Engine software, paving the way for future deployment of Greengrass components in containers. After this step, AWS IoT Greengrass Core is installed with the plugin specially dedicated to fleet provisioning. Finally, the provisioning command is executed using the *config.yaml* configuration file, which contains various parameters such as the device serial number, as well as the paths to locate the claim certificate and private key.

4.1.3 Security

At AWS, security is a top priority [77]. Security is a shared responsibility between AWS and the developer :

- **Security of the cloud** : AWS is responsible for protecting the infrastructure that runs AWS services in the AWS cloud.
- **Security in the cloud** : The developer's responsibility is determined by the AWS service that he uses.

When using AWS IoT Greengrass, the developer is also responsible for securing these devices, the connection to the local network and the private keys.

4.1.3.1 Authentication

Greengrass devices use X.509 certificates for authentication. X.509 certificates are digital certificates that adopt the X.509 public key infrastructure standard to bind a public key to the identity specified in the certificate. These X.509 certificates are issued by a trusted entity known as a **C**ertificate **A**uthority (CA). The CA holds one or more specific certificates, called CA certificates, which it uses to issue X.509 certificates. Only this CA has the privilege of accessing the certificates. In this process, the authority is Amazon Root **C**ertificate **A**uthority. When provisioning a device, the fleet provisioning plugin requests a unique client certificate from this authority. [77]

Only devices authorised to be provisioned will receive a certificate. In this implementation, a list allowing the serial numbers of authorised devices to be inserted has been created in a dedicated file (*allowlist.txt*). When the provisioning plugin attempts to provision a device, it will first trigger a Lambda function which will check the list to see if the device is authorised. If so, it will receive its unique certificate. If not, it will not receive it and will not be provisioned.

Greengrass devices store certificates in the default Greengrass root folder (*/greengrass/v2/*).

4.1.3.2 Authorization

Greengrass devices rely on AWS IoT policies for authorization. These AWS IoT policies determine the scope of operations permitted for AWS IoT devices. They specify, in detail, permissions or denials of access to AWS IoT Core and AWS IoT Greengrass data plane operations, including actions such as publishing MQTT messages. In this architecture, an AWS IoT policy is created directly from Pulumi. [77]

4.1.3.3 Data protection

Greengrass devices frequently collect data for transmission to AWS services for further processing. AWS IoT Greengrass uses encryption to protect data in transit (via the internet or local network) and at rest (stored in the AWS cloud). [77]

AWS IoT Greengrass offers two distinct modes of communication for data in transit :

- **Data in transit over the internet** : Communication over the internet between a device and AWS IoT is encrypted using the **T**ransport **L**ayer **S**ecurity (TLS) protocol. TLS is a security protocol that guarantees the confidentiality and integrity of data when it is exchanged over a network. All data sent to the AWS cloud therefore uses the MQTT protocol based on TLS.
- **Data on the device** : Communication between components on the Greengrass device is not encrypted, as the data does not leave the device.

When it comes to data at rest, AWS IoT Greengrass adopts different approaches :

- **Data at rest in the AWS cloud** : Customer data stored in the AWS cloud is encrypted by the AWS IoT Greengrass service using AWS KMS keys under its management.
- **Data at rest on the Greengrass device** : Data at rest on the device is protected by Unix file permissions and full disk encryption (if enabled, not enabled in this implementation). The user is responsible for securing the file system and the device.

4.2 CI/CD pipeline

A CI/CD process derived from the DevOps methodology has been put in place. It automates several workflows by integrating all the tools and software already described. The entire reference architecture has been developed in a public GitHub repository [78].

4.2.1 GitHub Actions

GitHub Actions is the **C**ontinuous **I**ntergration (CI) and **C**ontinuous **D**elivery (CD) service integrated directly into the GitHub platform and used in this project. It automates various aspects of the software development lifecycle, from building code to deploying it.

GitHub Actions works like this :

1. **Definition of workflows** : These describe the specific steps to be carried out during particular events, such as code validations, new version releases, etc. Several workflows can be run sequentially or in parallel.
2. **Triggering workflows** : These are triggered automatically in response to specific events defined by the developers. For example, a workflow can be triggered every time a new branch is created.
3. **Execution of steps** : Each workflow is made up of steps (jobs) that are executed sequentially or in parallel. These steps can include predefined actions, custom scripts, tests, deployments, etc.
4. **Runtime environments** : GitHub Actions provides runtime environments managed by GitHub. In this context, the Ubuntu-based Linux environment is used.
5. **Notifications and reports** : The results of each stage of the workflow are recorded and presented in the GitHub interface. Developers can receive notifications of successful or unsuccessful execution of the workflow.
6. **Integration with other services** : GitHub Actions can be integrated with other services and tools, such as cloud deployment services, notification services, secret management tools, etc.

7. **Actions ecosystem** : GitHub Actions offers a rich ecosystem of pre-built actions that developers can use in their workflows. These actions are reusable units of functionality (tests, deployments, notifications, etc.).

All workflows are implemented in a specific folder as follows :

```
.github/workflows
    ├── deploy-infra.yml
    ├── setup-provisioning.yml
    ├── build-image.yml
    ├── test-apps.yml
    ├── build-apps.yml
    ├── deploy-apps.yml
    └── destroy-infra.yml
```

4.2.2 Access to AWS resources

GitHub Actions integrates seamlessly with the AWS cloud. It uses the **O**pen**ID** **C**onnect (OIDC) credentials protocol to access resources when deploying or updating the cloud infrastructure. OIDC allows workflows to interact with AWS cloud service provider resources without having to permanently store credentials as GitHub secrets. Before deploying the infrastructure, the AWS account configuration is adjusted to recognise the GitHub OIDC as a trusted federated identity. In addition, this identification is based on IAM roles. This limits the possible actions on resources to the minimum required. Everything must be configured before the CI/CD process is launched.

To enable this connection to be used, AWS provides a pre-built action which is integrated into the flows (*aws-actions/configure-aws-credentials*).

4.2.3 Infrastructure deployment

This initial flow is triggered each time a new modification is pushed into Git. Before deploying the cloud infrastructure, the decision was taken to generate a claim certificate and store it in a dedicated S3 bucket. This operation is carried out only once, ensuring that the same claim certificate is always preserved, even if the infrastructure is destroyed and redeployed at a later date. This ensures that devices ready to be provisioned can always do so using the same claim certificate.

A second S3 bucket is specifically created to record the state of the infrastructure. Pulumi connects to it, orchestrating the deployment of the configured infrastructure, whether for the development or production environment.

Successful completion of this first stage simultaneously triggers two other processes: provisioning configuration and testing of the developed applications. If deployment fails, the entire CI/CD process is interrupted.

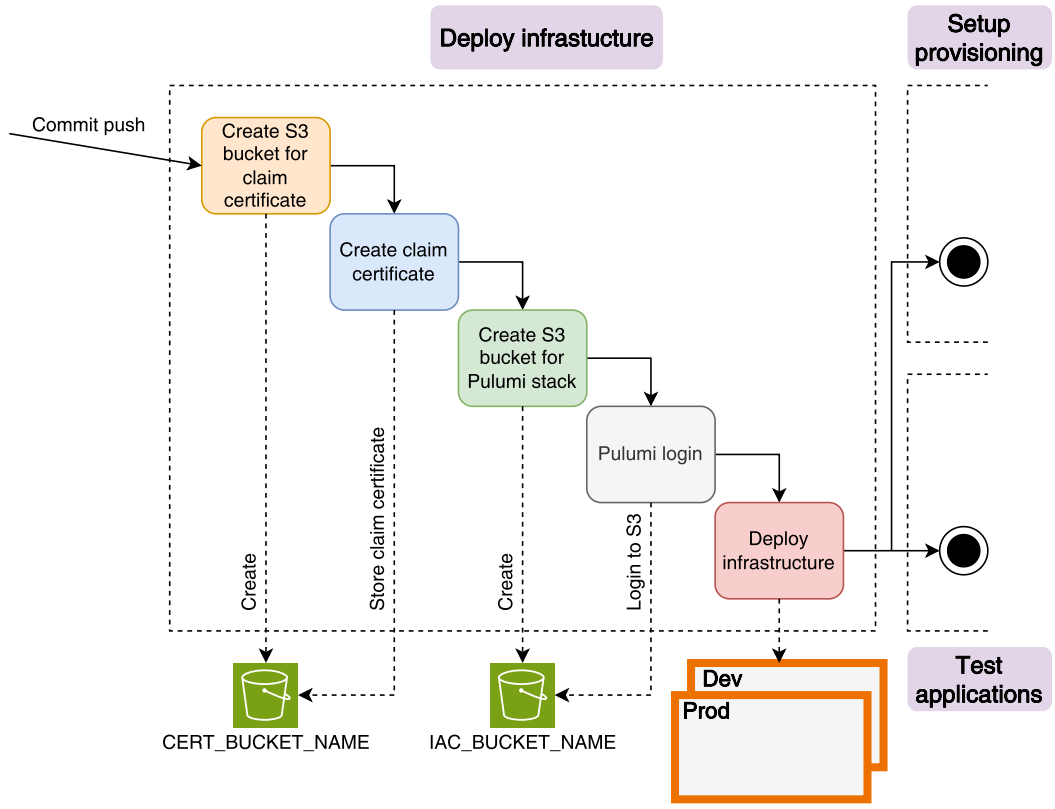


Figure 4.2 Infrastructure deployment workflow

4.2.4 Provisioning setup

This flow manages the preparation of configuration files for provisioning embedded systems, as well as checking devices that have already been provisioned. Two tasks are performed simultaneously. The first checks the serial numbers of devices in a dedicated list. If previously provisioned devices are no longer in the list, they are removed from the AWS cloud. Authorised devices are reassigned to the new infrastructure and their unique certificate is reassociated with the AWS IoT policy for actions in the cloud.

The second task, related to configuration, retrieves the IoT endpoints for future communications between the devices and AWS IoT. Next, the parameters are saved in a configuration file required for provisioning. This file is stored, along with the list of authorised devices, in the same S3 bucket as the claim certificate.

Only if this flow is successful will the creation of the Linux image begin.

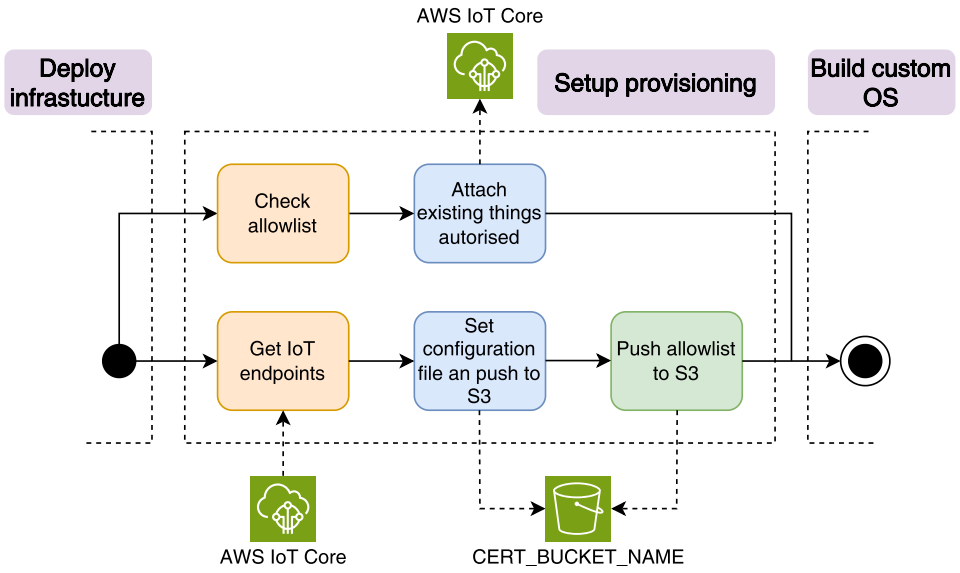


Figure 4.3 *Provisioning setup workflow*

4.2.5 OS image building

This process begins by retrieving the configuration file and the claim certificate, along with its associated private key, which have previously been saved in the S3 bucket. The image is then created using the Packer tool and its plugin. The image is created in a Docker container preconfigured with the necessary environment. Once the image has been generated, it is saved in its corresponding S3 bucket.

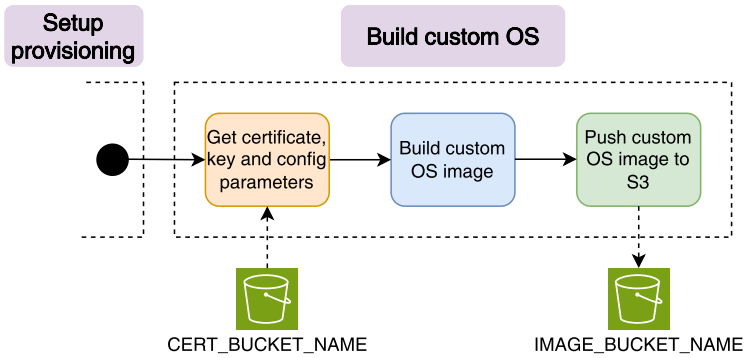


Figure 4.4 *OS image building workflow*

4.2.6 Applications testing

Simultaneously with the two previous workflows, this stage aims to test all the applications developed. It runs three tests concurrently. The first tests the linting of each application, helping to improve the quality of the code. The second performs static security tests for each application. Finally, unit tests are carried out using test code developed in parallel with the applications. The tools used in this task will depend on the programming language used.

If all the tests have been passed, the applications can be built.

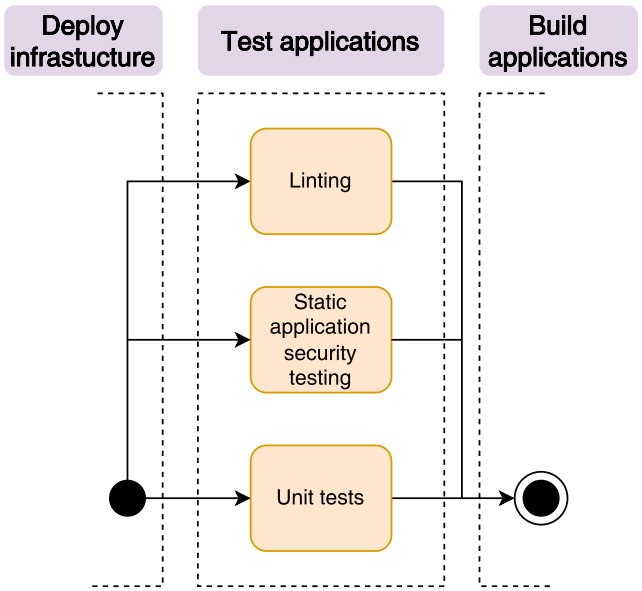


Figure 4.5 Applications testing workflow

4.2.7 Applications building

This process is divided into two stages. The first involves containerising applications using Docker, with this operation applied only to new applications or those that have undergone modifications. The resulting images are then stored in the dedicated Amazon ECR service.

The second part of the process focuses on creating a Greengrass component for each application. When a component is updated, a new version is created. However, only applications that have undergone modifications will see a new version of their component generated. It is also possible for a component's parameters to change, causing it to be updated. Each time a new version of a component is released, it is published in AWS IoT Greengrass, orchestrated by the AWS Greengrass Development Kit (GDK).

Once this workflow has been successfully completed, the Greengrass components are deployed.

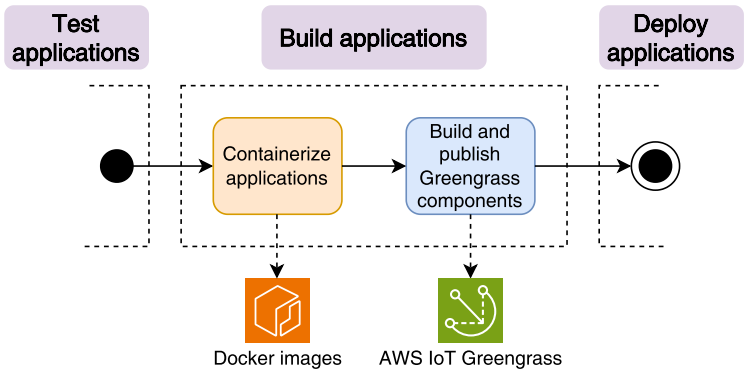


Figure 4.6 Applications building workflow

4.2.8 Applications deployment

This workflow supports the deployment of Greengrass components on a group of devices. It does this using the AWS CLI. It doesn't matter whether the components have been updated or not, deployment takes place either way. AWS IoT Greengrass Deployment is smart enough to see if a new version of a component has been released. If this is the case, only the component with the new version will be deployed in the device group.

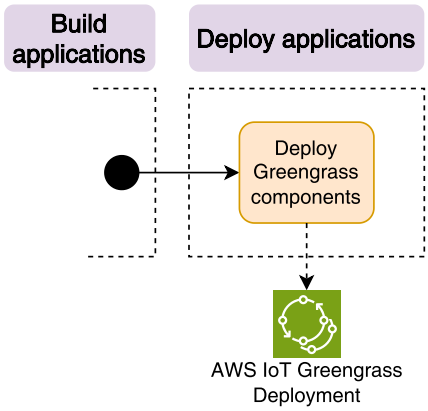


Figure 4.7 Applications deployment workflow

4.2.9 Infrastructure destruction

This last workflow is optional and has the task of dismantling the AWS cloud infrastructure. To enable Pulumi to delete the S3 bucket containing the OS image, it must first be emptied. The OS image is then deleted. The bucket containing the configuration file and the list of authorised devices is emptied, with the exception of the claim certificate and its private key. Embedded systems integrated into the AWS cloud retain their unique certificate in AWS, but lose their AWS IoT policy, dissociating them from any action. Finally, Pulumi connects to the S3 bucket containing the stack to completely remove the infrastructure. Once all the operations have been completed, the S3 bucket containing the stack is also deleted.

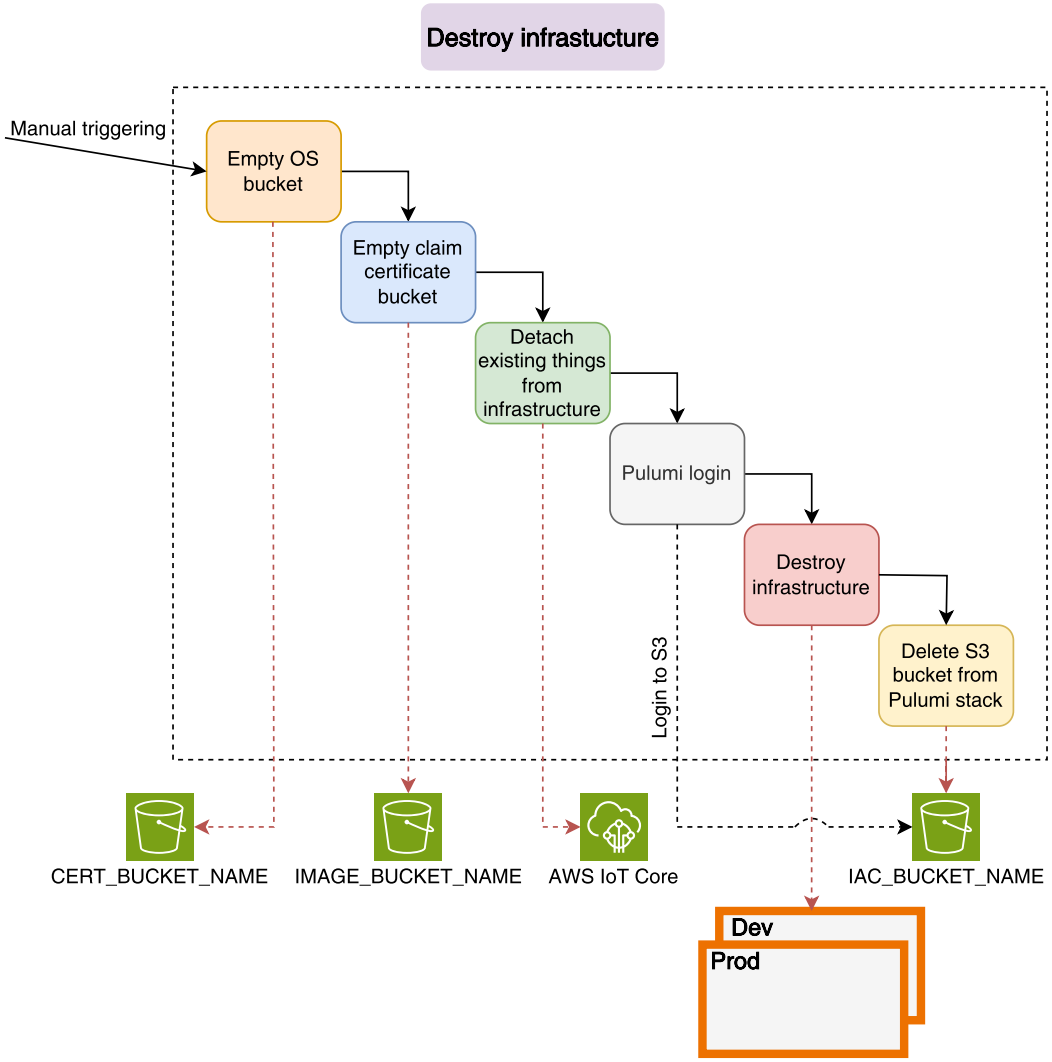


Figure 4.8 Infrastructure destruction workflow

4.3 Applications

4.3.1 Greengrass components

Greengrass components are software modules that are deployed on devices containing the AWS IoT Greengrass Core program. Components can be applications, runtime installers, libraries, or any code that is executable on a device. Components may depend on other components. For example, one component installs Python and other components depend on it to run Python applications. When components are deployed on a fleet of devices, AWS IoT Greengrass deploys only the software modules that the devices need. [79]

Here are some key points about Greengrass components :

- **Greengrass component definition** : A component can be defined as a combination of code (for example, a Python application), software dependencies and system resources. These may include libraries, configuration files, executables, etc.

- **Component management** : AWS IoT Greengrass enables centralised management of components. Components can be created, published and managed in AWS IoT Greengrass and deployed to groups of Greengrass devices.
- **Types of components** : Two types of component are used
  - Nucleus (aws.greengrass.nucleus): The Greengrass nucleus is the mandatory installed component that provides the minimum functionality for the operation of the AWS IoT Greengrass Core software.
  - Generic (aws.greengrass.generic): The Greengrass nucleus executes lifecycle scripts for a generic component. This type is the default for custom components.
- **Deployment** : Components are deployed on groups of Greengrass devices to run device-level applications. Deployments are orchestrated and managed from the AWS IoT Greengrass Deployment service.
- **Versioning** : Components can be versioned, enabling different versions of the software to be managed and updates to be rolled out in a controlled manner.

In this project, a component is defined in a folder as follows :

```
component_1
    ├── gdk-config.json
    └── recipe.yaml
```

The *gdk-config.json* file is specific to the AWS GDK tool, used to generate and publish a component to AWS IoT Greengrass. It only includes parameters such as the component name, for example.

The *recipe.yaml* file describes the component, its required dependencies, its lifecycle and the location of the executable. An example of this file is shown below.

```yaml
RecipeFormatVersion: "2020-01-25"
ComponentName: COMPONENT_NAME
ComponentVersion: COMPONENT_VERSION
ComponentDescription: COMPONENT_DESCRIPTION
ComponentPublisher: AUTHOR
ComponentConfiguration:
    DefaultConfiguration:
        var1: 1
        var2: "frequency"
        accessControl:
            aws.greengrass.ipc.mqttproxy:
                com.app:mqttproxy:1:
                    policyDescription: "Allows access to subscribe to input
                    ↪  topics from AWS IoT."
                    operations:
                        - "aws.greengrass#SubscribeToIoTCore"
                    resources:
                        - "topic1"
                com.app:mqttproxy:2:
                    policyDescription: "Allows access to publish to output
                    ↪  topics to AWS IoT."
                    operations:
                        - "aws.greengrass#PublishToIoTCore"
                    resources:
                        - "topic2"
ComponentDependencies:
```

```
      aws.greengrass.DockerApplicationManager
  Manifests:
  - Platform:
      os: linux
      architecture: aarch64
    Lifecycle:
      run:
          RequiresPrivilege: true
          Script: docker run DOCKER_IMAGE@DIGEST
    Artifacts:
      - URI:
        ↪  "docker:AWS_ACCOUNT_ID.dkr.ecr.AWS_REGION.amazonaws.com/DOCKER_IMAGE"
```

Starting at the top of the example, information about the component is presented. Next, the configuration (*ComponentConfiguration*) is detailed, allowing the creation of variables and the configuration of access controls (*accessControl*). The latter is used to define authorised operations, typically using the MQTT protocol, such as the topics to which the component can publish or subscribe. Next, the dependent components are specified (*ComponentDependencies*). Finally, the lifecycle (*Lifecycle*) is detailed, along with the specification of the environment platform. In this example, a Docker application is launched and its image has been retrieved from the Amazon ECR service via its URI.

4.3.2 Containerization

The Python applications developed in this project are run in Docker containers. The creation of the application image requires the use of a Dockerfile. This specifies the architecture of the embedded system on which the container is to run, as well as the base image used. Finally, the Dockerfile copies the directory containing the application into its environment, then simply executes a bash script. This bash script installs the necessary Python libraries and launches the Python application.

```
docker_app_1
    ├── app_1
    │   ├── start.sh
    │   ├── requirements.txt
    │   └── main.py
    └── Dockerfile
```

4.3.3 Certificate rotator application

This application, which is essential to ensure that IoT devices work properly, has been implemented in Python. Figure 4.9 illustrates the associated class diagram.

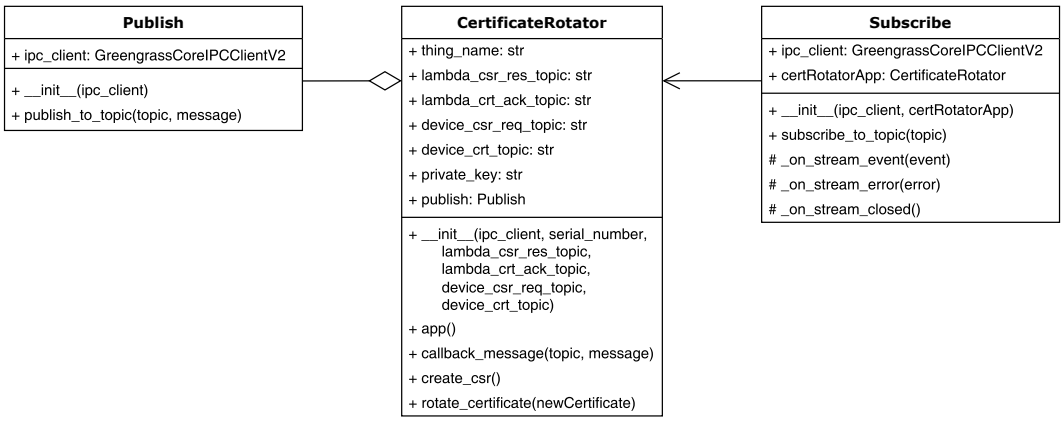


Figure 4.9 *Certificate rotation application class diagram*

Below is a description of the various classes :

- *CertificateRotator* : Main class that includes the program's main loop. It generates CSR and private keys. It also changes the old certificates to the new ones received.
- *Publish* : Class that publishes MQTT messages to AWS IoT.
- *Subscribe* : Class that receives MQTT messages from AWS IoT.

A sequence diagram illustrating how the application works is available in figure 4.10.

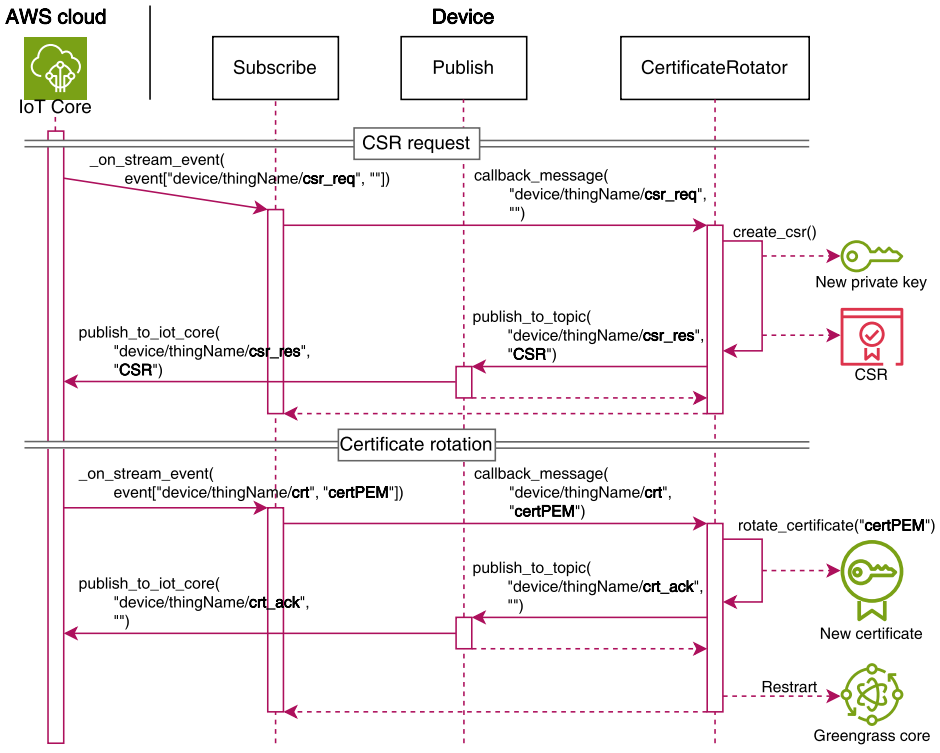


Figure 4.10 *Certificate rotation application sequence diagram*

The sequence diagram 4.10 highlights the two main actions of the application : the **C**ertificate **S**igning **R**equest (CSR) and the certificate replacement. When the MQTT broker integrated into AWS IoT Core transmits a message relating to the creation of the

CSR, the corresponding method is triggered. This method then sends the CSR to AWS IoT Core, which responds by providing a new certificate, thus initiating the rotation process.

Another sequence diagram is shown in Figure 4.11, detailing the process of integrating the application into the IoT device in collaboration with AWS services.

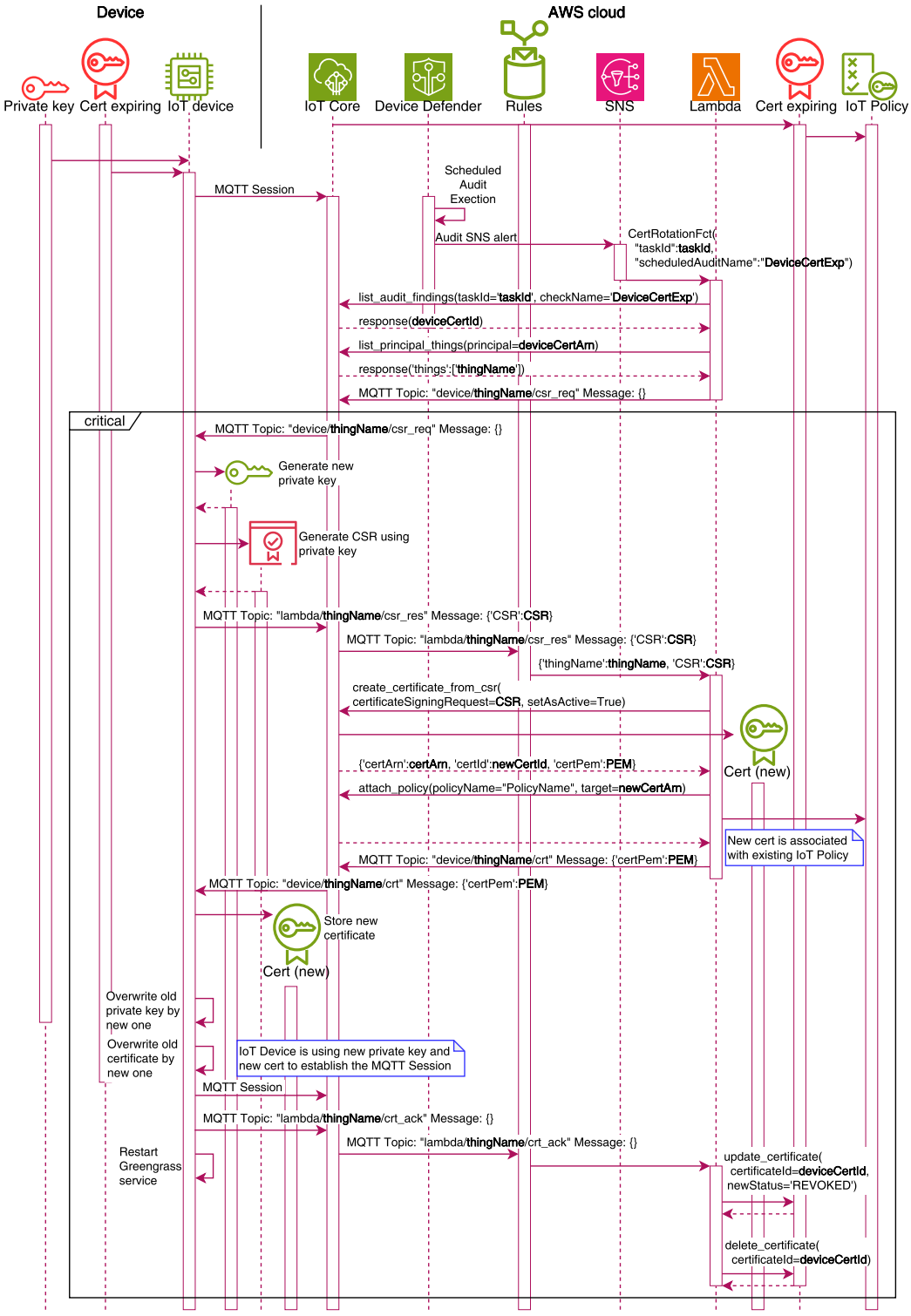


Figure 4.11 *Certificate rotation sequence diagram with AWS services [72]*

When a device's certificate expires, AWS IoT Device Defender sends a notification to the device via the Amazon SNS service and a Lambda function. The device then creates a CSR, which it sends to AWS IoT. This service then generates a new unique certificate by signing it with the Amazon Certificate Authority. The new certificate is then sent to the device, which replaces it with the old one. Finally, confirmation of the rotation is sent to the cloud. AWS IoT then triggers a Lambda function that revokes and deletes the old certificate. Meanwhile, the Greengrass service on the IoT device is restarted to support the new certificate for all new communications.

4.3.4 Led application

The led application was developed in Python using the class diagram shown in figure 4.12.

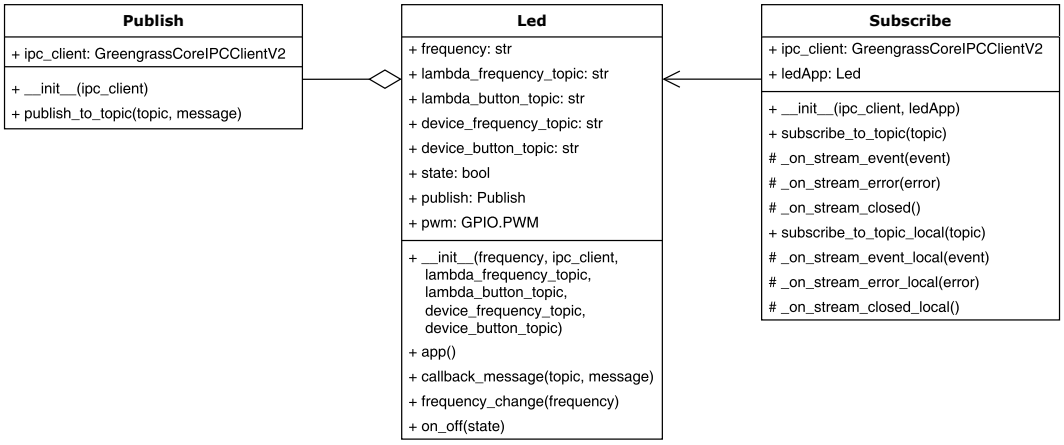


Figure 4.12 *Led application class diagram*

Below is a description of the various classes :

- *Led* : Main class that includes the program's main loop. It changes the blinking frequency of the led according to the messages received. It also manages the activation and deactivation of the blinking.
- *Publish* : Class that publishes MQTT messages to AWS IoT.
- *Subscribe* : Class that receives MQTT messages from AWS IoT and the button application at local level.

The application's behaviour is shown in figure 4.13 by a sequence diagram.

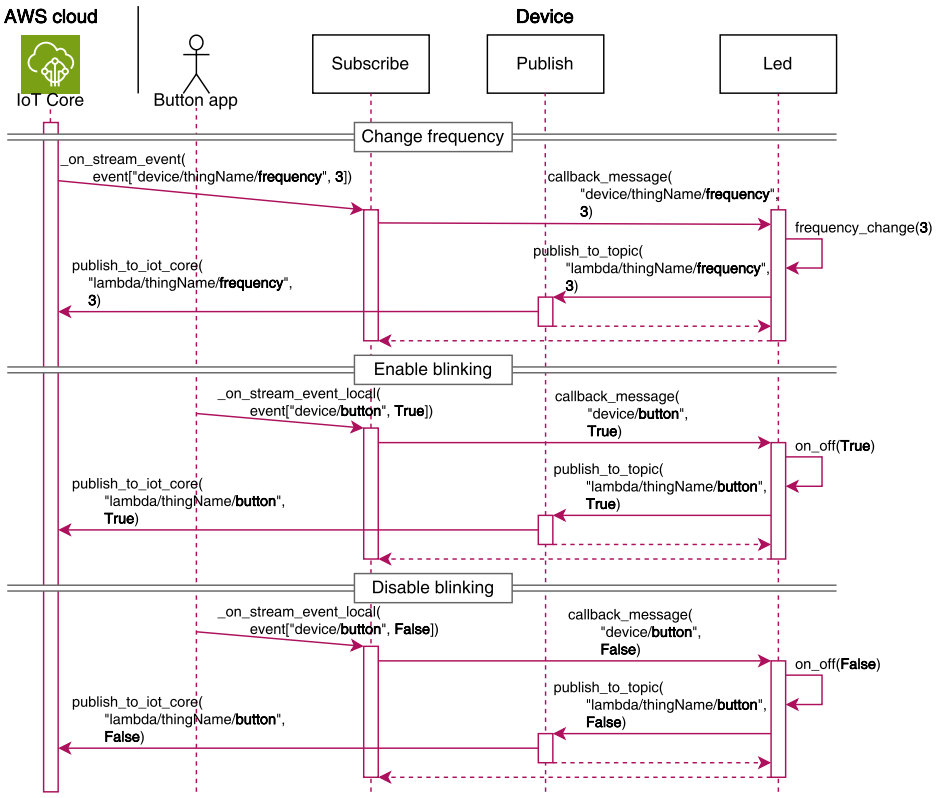


Figure 4.13 Led application sequence diagram

The frequency is modified from an interface associated with AWS IoT Core, which sends messages asynchronously. The message is routed to the core class, which responds by sending an MQTT message to AWS IoT to confirm receipt and optionally logs this data in an interface. Activation of the blinking is controlled by a button on the device, managed by the button application. Depending on the events, a local MQTT message is transmitted to the main class. The *Led* class sends a confirmation to AWS IoT with the status of the blinking, so that it can be logged if necessary. When the external button is pressed a second time, the status is logically reversed.

4.3.5 Button application

The button application has been implemented using the following class diagram 4.14. The programming language remains Python.

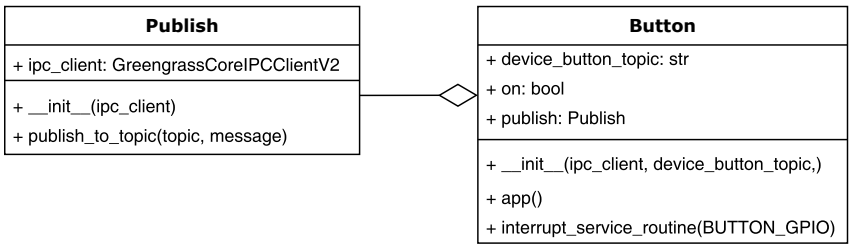


Figure 4.14 Button application class diagram

Here is a description of the different classes :

- *Button* : Main class that contains the program's main loop. This is the class that manages the interrupts each time the external button is pressed. MQTT messages are prepared by this class.
- *Publish* : Class which publishes local MQTT messages to the led application.

The sequence diagram 4.15 describes the internal workings of the application.

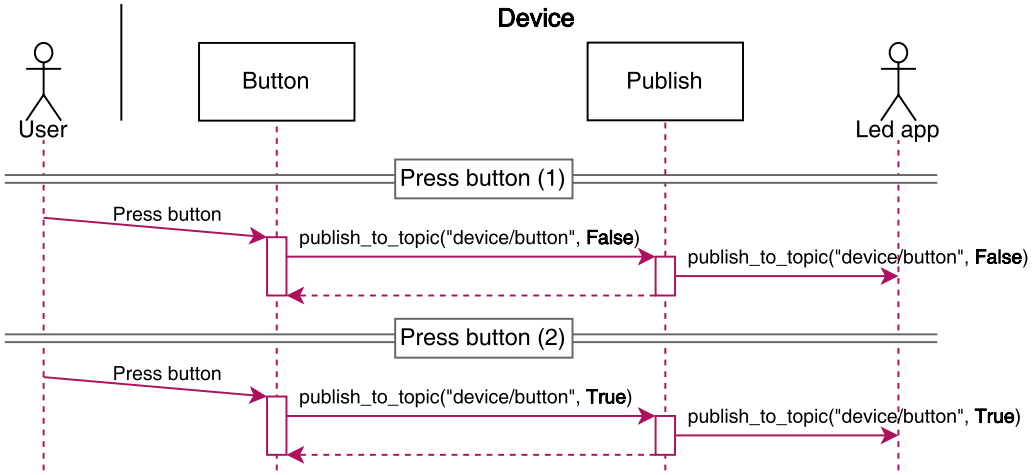


Figure 4.15 *Button application sequence diagram*

The user can press the external button connected to the IoT device at any time. An interrupt is triggered in the main class, reversing the current state of the blinking. In the sequence diagram scenario, blinking was initially enabled. The first time it is pressed, the blinking is deactivated. An MQTT message is then sent to the led application. When the button is pressed again, the status is reversed and the blinking should reactivate.

4.3.6 Corrupt application

As mentioned in the section on containerization 4.3.2, it is possible to specify the processor architecture on which the Docker container should run. On Arm SystemReady certified embedded systems, the Arm processor architecture is *arm64*. To make this application ineligible to run, an incorrect architecture is specified. In this case, the *amd64* architecture has been deliberately specified in the Dockerfile, as follows :

```
FROM --platform=linux/amd64 python:latest

CMD ["echo", "Corrupt Docker container"]
```

The Greengrass component itself is correctly implemented and will run normally. However, when executing the Docker command with the incompatible image, an error occurs and the application is stopped.

4.3.7 OS update

This application simply consists of command lines. These are written directly in the structure of the Greengrass component. The corresponding file, *recipe.yaml*, is constructed as follows :

```
RecipeFormatVersion: "2020-01-25"
ComponentName: "COMPONENT_NAME"
ComponentVersion: "{COMPONENT_VERSION}"
ComponentDescription: "This is a component to update OS."
ComponentPublisher: "{COMPONENT_AUTHOR}"
ComponentConfiguration:
DefaultConfiguration:
    update_no: "./update_1"
Manifests:
    - Platform:
        os: linux
        architecture: aarch64
      Lifecycle:
        run:
            RequiresPrivilege: true
            Script: |-
                if [ ! -f {configuration:/update_no} ] ; then
                    touch {configuration:/update_no}

                    # Write your commands
                    apt-get update
                    shutdown -r now
                else
                    echo "Already updated."
                fi
```

Each time the OS needs to be updated, a new version must be assigned to the *update_no* variable. This represents the name of a file with its path, ensuring that the update is only carried out once. In the script section, a condition checks whether this file has already been created. If so, this means that the OS update has already been performed. If not, the file is created and the commands specified under the comment "*# Write your commands*" are executed sequentially. These commands, chosen by the developer, are responsible for updating the functionality of the OS and are executed with full administrator privileges. The file created keeps a record that the update has already taken place when the device reboots to ensure the update.

5 Validation

This chapter outlines the tests carried out, along with their results, to demonstrate the proper functioning of the reference architecture as a whole. In addition, it examines the notion of cost linked to the cloud infrastructure.

Contents

5.1 Project configuration

Before pushing this reference architecture into a new GitHub repository, a few project configurations have been made. Firstly, it is essential to add the GitHub Actions user as a trusted identity so that AWS can grant him access to its resources. In the AWS account on which the infrastructure is deployed, the GitHub identity provider is specified (figure 5.1).

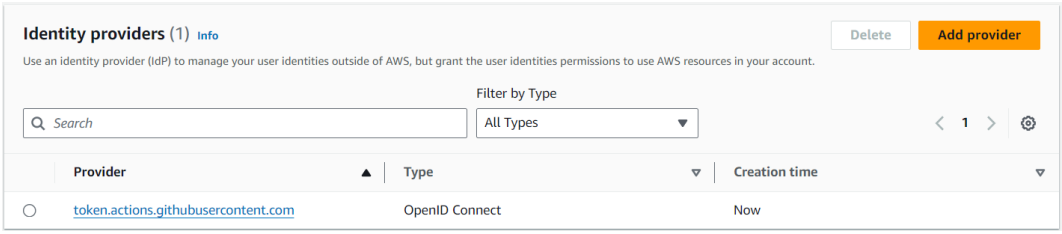


Figure 5.1 GitHub identity provider

An IAM role has been set up to authorise only the project's GitHub repository to access resources. A policy is linked to limit the actions allowed on the resources (figure 5.2).

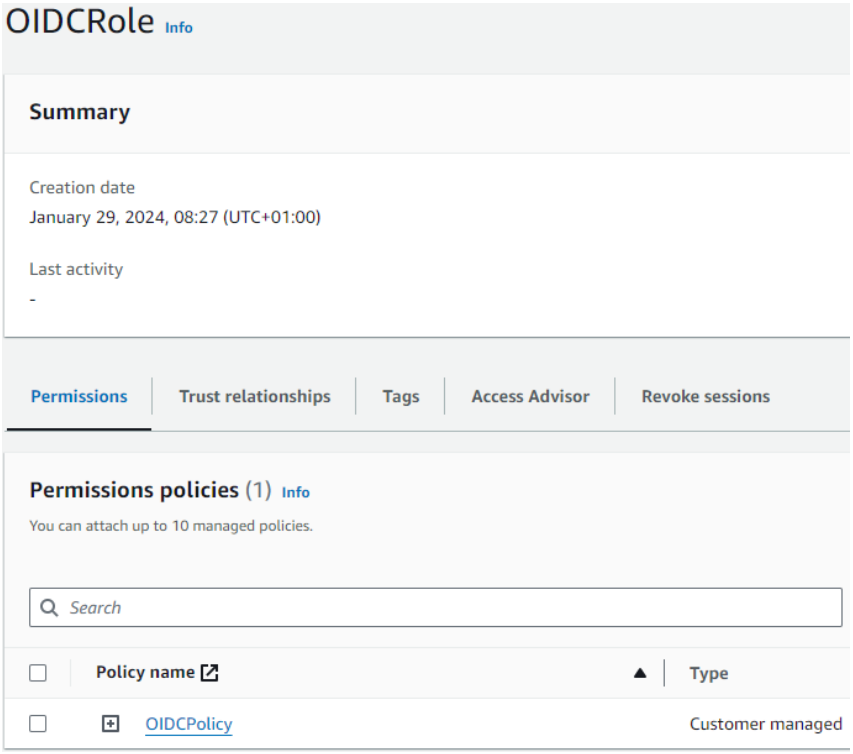


Figure 5.2 IAM OIDC role

The secret and visible variables that need to be configured have been added to the GitHub repository (figures 5.3 and 5.4).

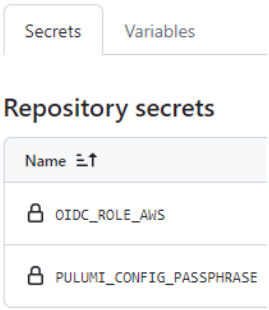


Figure 5.3 GitHub secret variables

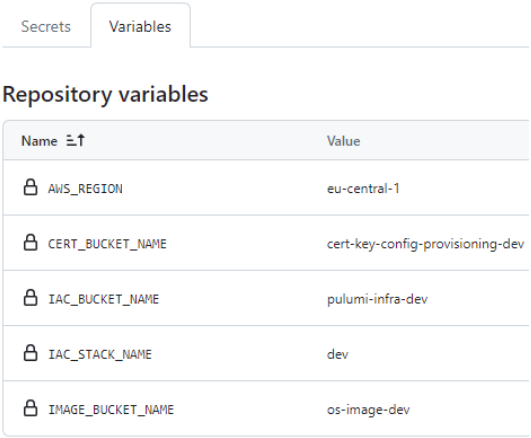


Figure 5.4 GitHub visible variables

The configuration file for the Pulumi tool was configured by specifying the AWS account ID and the region (figure 5.5).

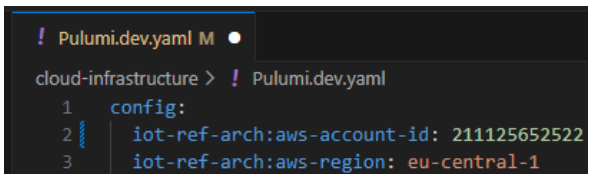


Figure 5.5 Pulumi configuration file (development environment)

Finally, the serial numbers of the devices authorised to be provisioned have been listed. In this case, only a Raspberry Pi 4 is authorised (figure 5.6).

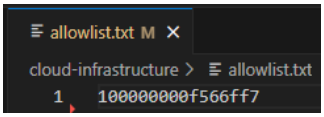


Figure 5.6 Embedded systems allowlist

5.2 CI/CD pipeline

After the previous configuration, the project was pushed into the GitHub repository. The CI/CD pipeline started automatically, beginning with the deployment of the cloud infrastructure. All tasks were completed successfully. The different times for each workflow can be seen on the right-hand side of figure 5.7. Total process time is just over 10 minutes.

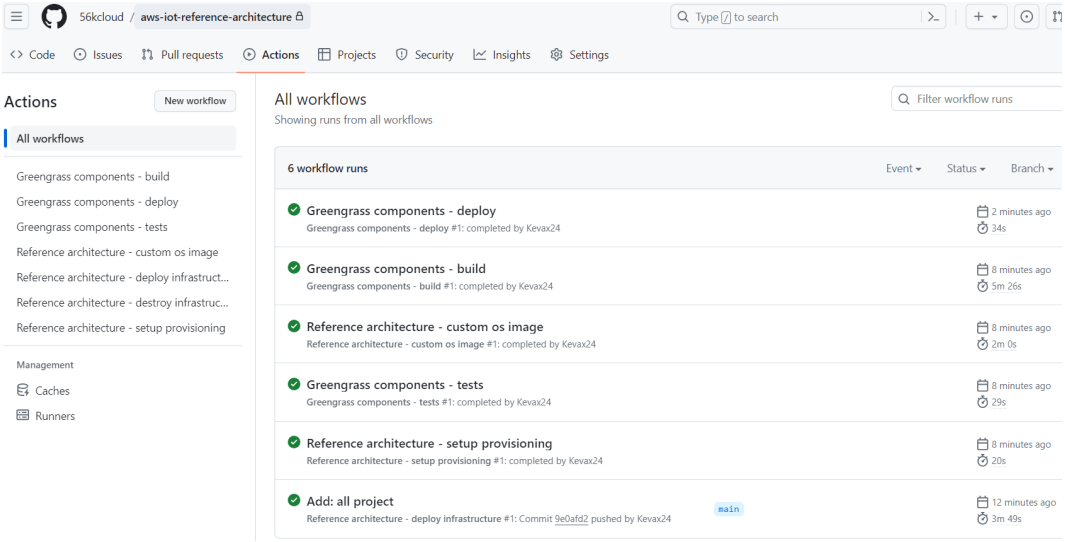


Figure 5.7 Successful workflows

The cloud infrastructure has been deployed on the development account specified and in the *eu-central-1* region. The configuration for the provisioning of the embedded systems was carried out correctly with the creation of the OS image. The base image used is *Raspberry Pi OS Lite* based on the Debian distribution.

At the same time, the applications were tested, including the led application, the button application and the certificate rotation application. Five Greengrass components were created and published in AWS. These components were also deployed.

5.3 Cloud infrastructure deployment

All the resources described in Pulumi have been created correctly. In this sequel, they are not all mentioned and presented in images due to the sheer number of resources. An example is shown in figure 5.8 where the S3 buckets have been created. There is the storage space for the state of the infrastructure, another for the provisioning configuration and finally one where the OS image is stored.

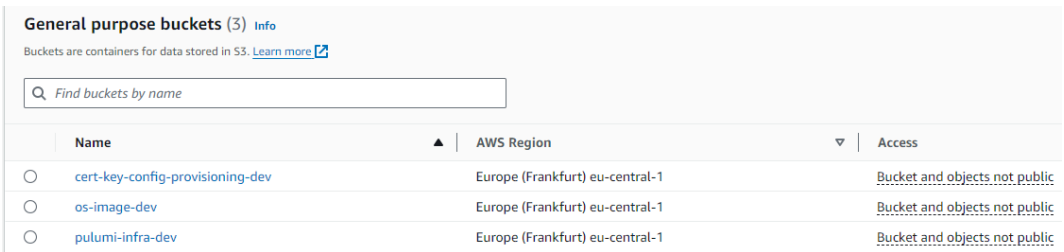


Figure 5.8 S3 buckets

The IoT device group has been created under the *GreengrassGroup* name. Two policies were created for provisioning and for interactions once provisioned (figure 5.9). The claim certificate has also been prepared and linked to the provisioning policy.

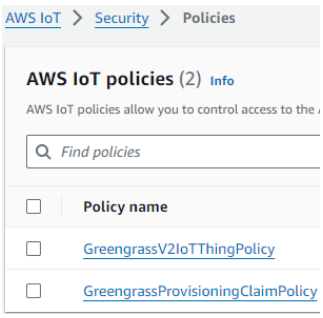


Figure 5.9 IoT policies

The Greengrass components have been successfully published (figure 5.10). Docker images of the applications are also available in the Amazon ECR service.

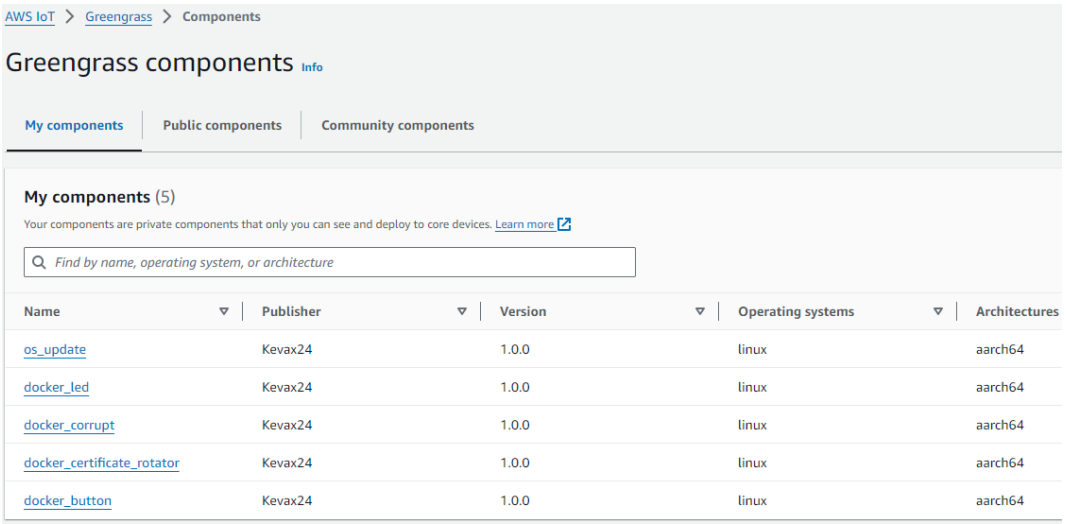


Figure 5.10 Greengrass components

A deployment service is ready to deploy applications as soon as devices are provisioned. It targets the group created (figure 5.11).

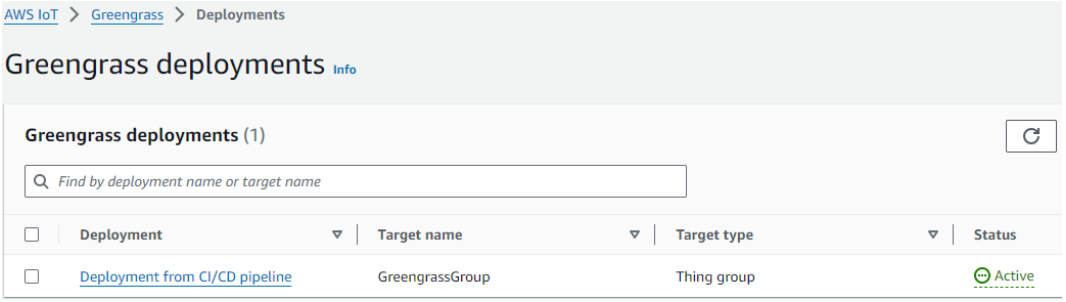


Figure 5.11 AWS IoT Greengrass Deployment

5.3.1 Second environment deployment

Above, the development environment has been deployed. This section deals with testing the deployment of the production infrastructure. To do this, the variables configured in GitHub were adjusted (figure 5.12). A new OIDC connection has been set up. By specifying the environment to the *IAC_STACK_NAME* variable, it will call the associated Pulumi configuration file (*Pulumi.prod.yaml*).

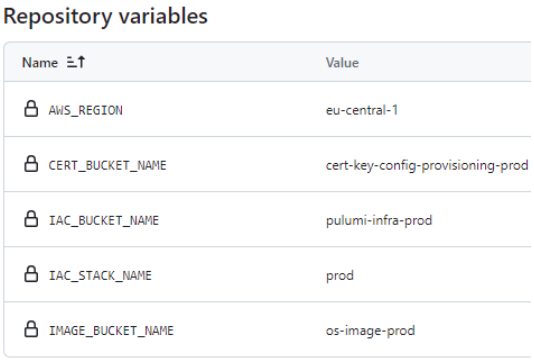


Figure 5.12 GitHub visible variables for the production environment

The CI/CD pipeline was launched manually from the GitHub interface. All workflows passed. Figure 5.13 shows part of the deployment of the production infrastructure. The S3 buckets are named in accordance with the variables defined in GitHub. In addition, figure 5.13 is a screenshot from an AWS account separate from the development account, specifically dedicated to production.

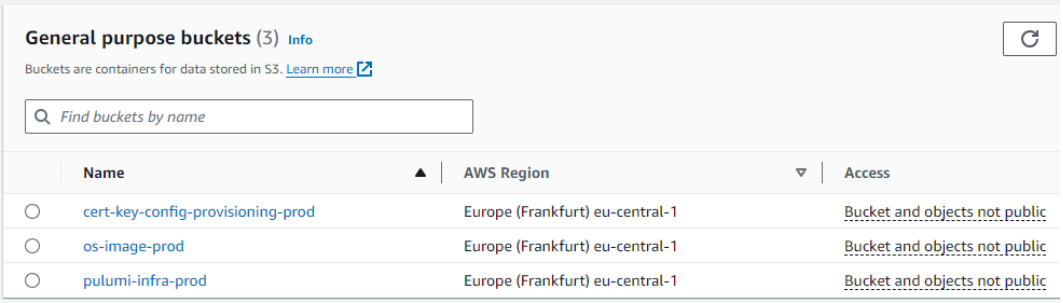


Figure 5.13 S3 buckets in the production environment

5.4 Provisioning a Raspberry Pi 4

The integration of an IoT device was a success. A Raspberry Pi 4 was provisioned. The OS image was first downloaded from the S3 bucket. It was then flashed into an SD card (figure 5.14).

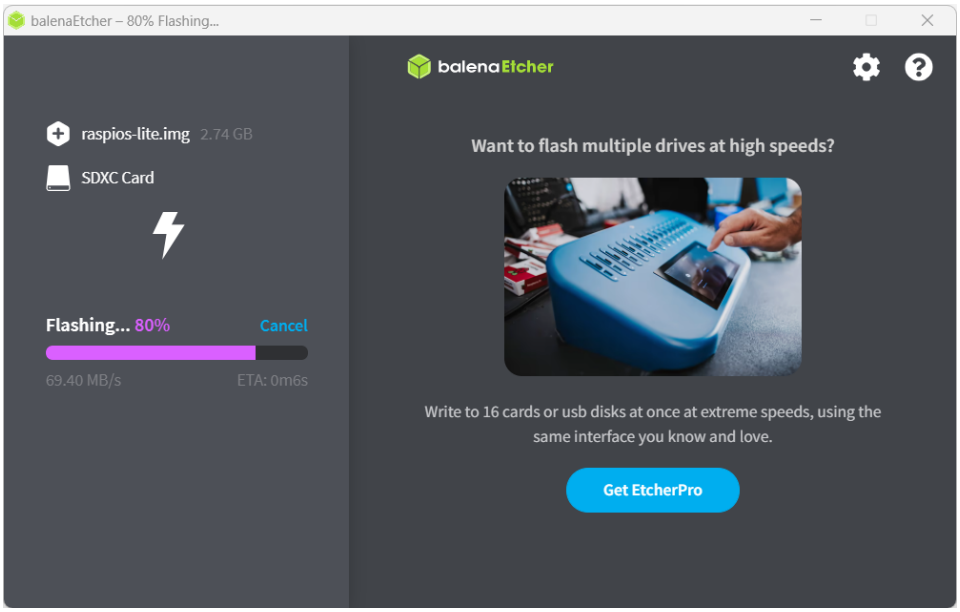


Figure 5.14 *OS image flashing*

The Raspberry Pi 4 was then powered up and booted for the first time. All the software was installed and provisioning was successfully completed (figure 5.15).

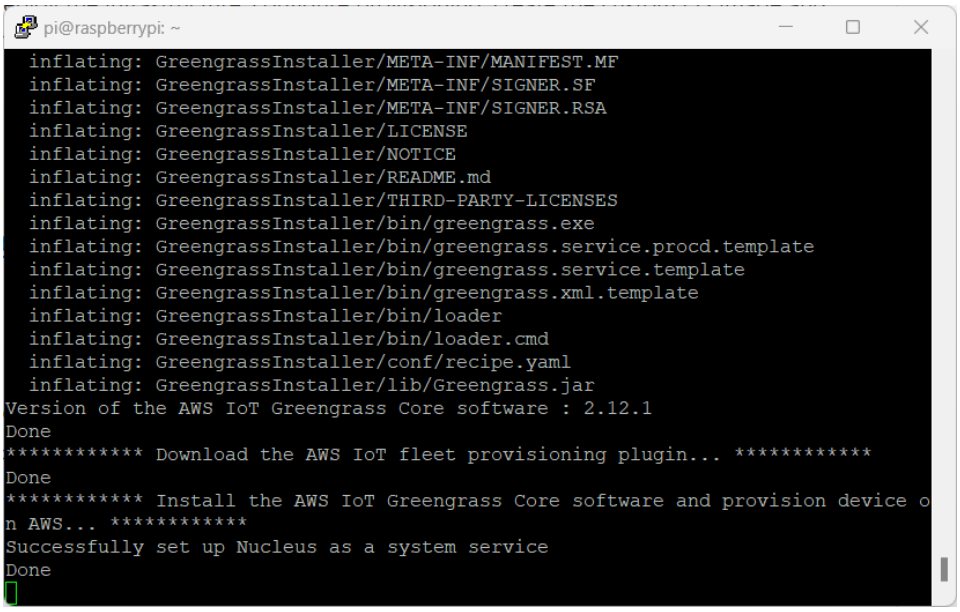


Figure 5.15 *End of provisioning on the device*

Provisioning is confirmed in figure 5.16. AWS has created a digital twin of this device on the AWS IoT service. The name corresponds to its serial number. It is linked to the group created earlier. A unique X.509 certificate has been provided.

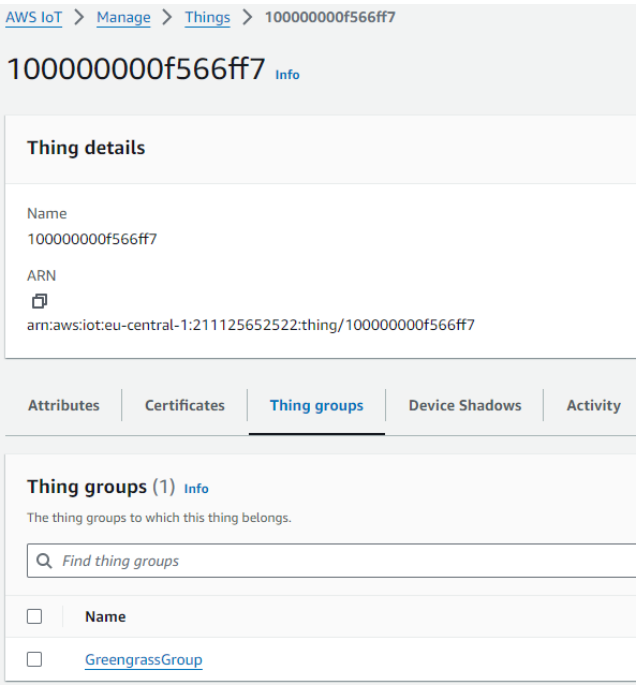


Figure 5.16 Confirmed provisioning on AWS

It should be noted that the power supply to the Raspberry Pi 4 was removed several times. Each time it was reconnected, the device reconnected to AWS IoT Core.

5.4.1 Adding a second Raspberry Pi 4

A second Raspberry Pi 4 has been successfully provisioned. Its serial number has been added to the list of authorised devices. A proof of its digital twin can be seen in figure 5.17.

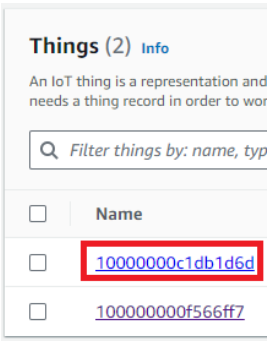


Figure 5.17 Second provisioning confirmed on AWS

The components have been deployed correctly. A message exchange can be seen on the AWS MQTT client interface (figure 5.18). A new blinking frequency was sent to the second Raspberry Pi 4 and a message from it was returned to AWS IoT.

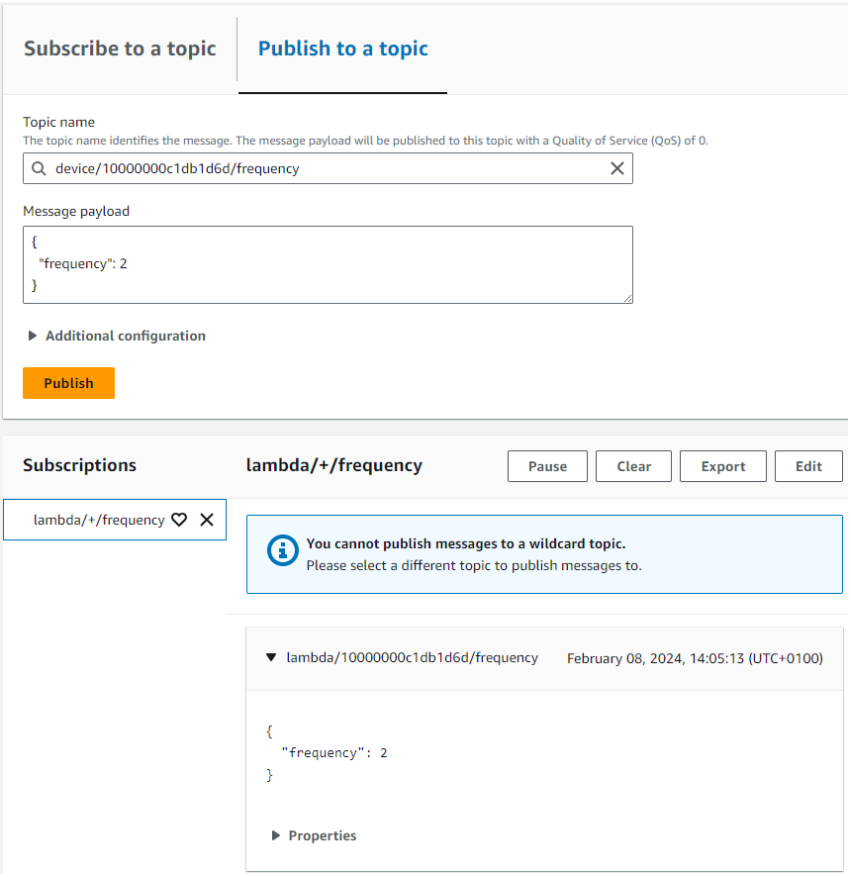


Figure 5.18 Exchanging MQTT messages with the second Raspberry Pi 4

5.5 Applications

All the applications were deployed on the Raspberry Pi 4. As soon as it was provisioned, the deployment started. This can be seen in figure 5.19.

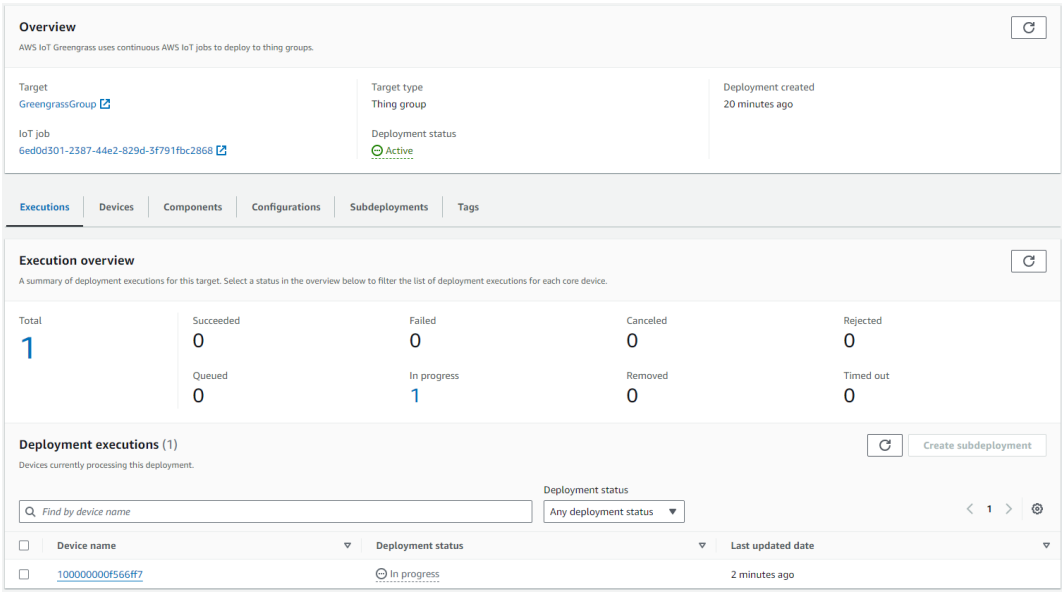


Figure 5.19 Deployment of Greengrass components in progress

Deployment took just a few minutes. The Greengrass components acting as intermediaries for the applications can be seen in figure 5.20. The dependencies are also installed in addition to the Greengrass nucleus *aws.greengrass.Nucleus*.

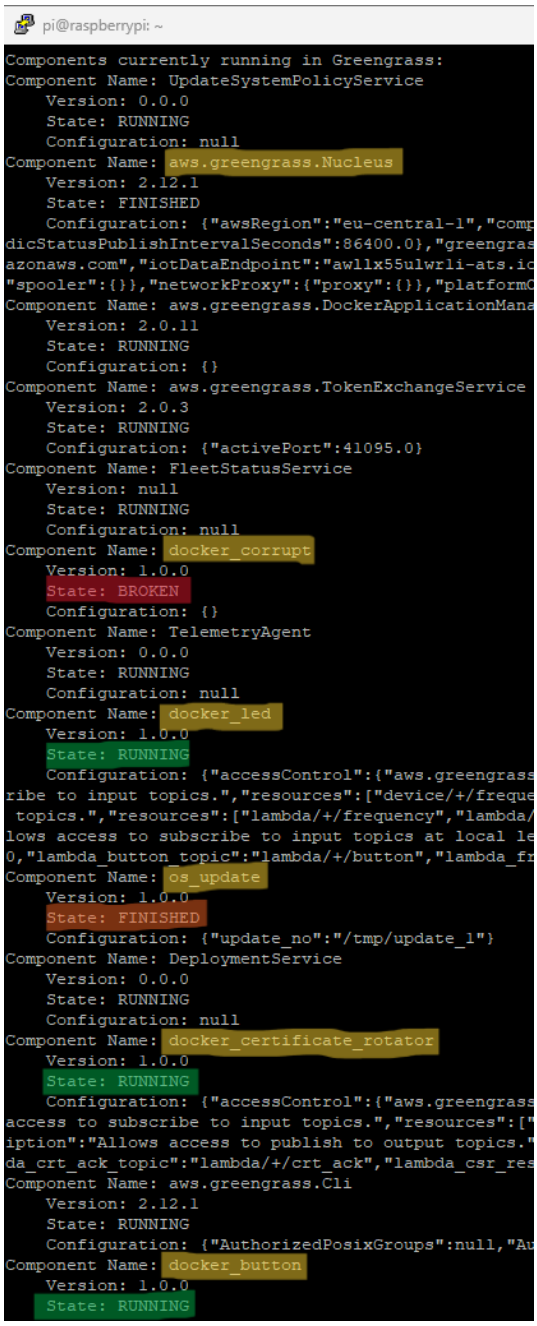


Figure 5.20 Greengrass components deployed

It is possible to see an overview of the Raspberry Pi 4 on AWS as shown in figure 5.21. Its state of health is poor because a component failed when it was launched. In fact, the corrupted component was unable to start up correctly and was interrupted. The other components are in a state of execution or completion.

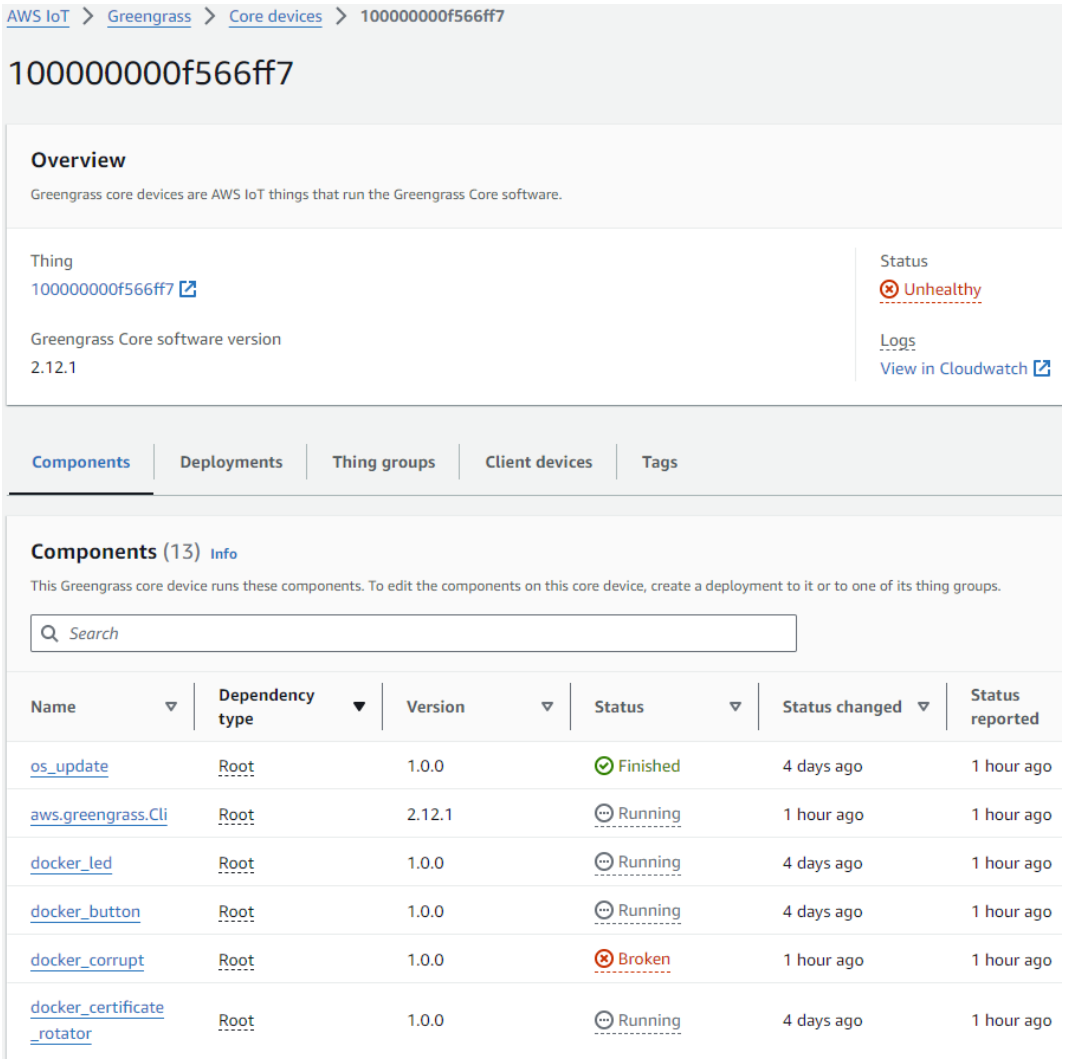


Figure 5.21 Overview of the Raspberry Pi 4 from AWS

5.5.1 Applications interaction

The applications were then tested. The first was the led application. The AWS web interface was used to send MQTT messages and view the data received. A message is sent to the Raspberry Pi 4 to change its led blinking frequency. A response was sent back to AWS IoT in another topic including the new frequency. A proof can be found in figure 5.22.

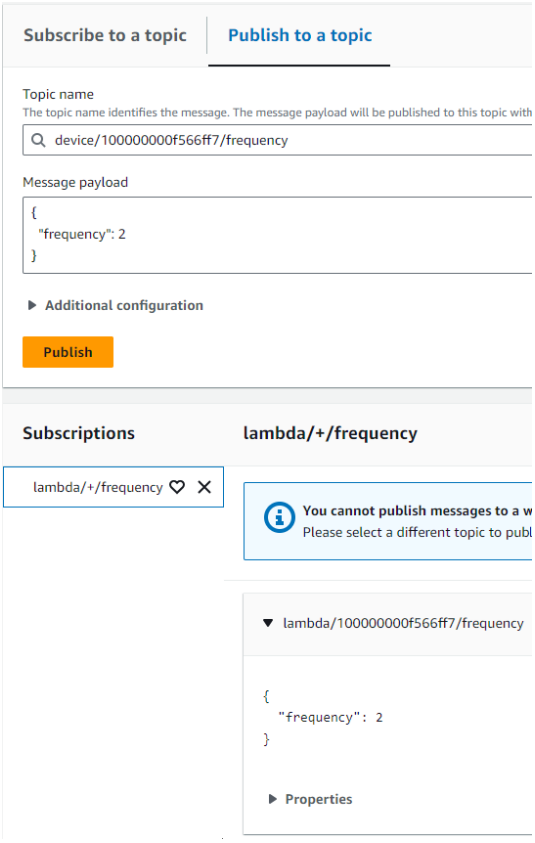


Figure 5.22 *Frequency change on AWS*

Figure 5.23 shows the led application output message thread. It is possible to view a frequency received on the Raspberry Pi 4.

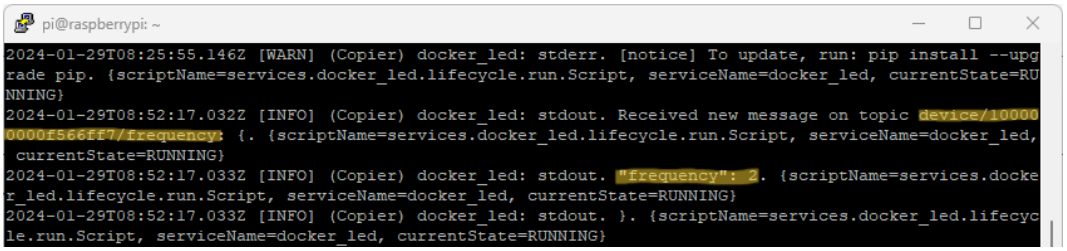


Figure 5.23 *Frequency change on the Raspberry Pi 4*

The button application was also tested. The external button on the Raspberry Pi 4 was pressed twice. MQTT messages were transmitted to the led application locally and then published to AWS IoT. The messages can be seen in figure 5.24. The first press deactivates the blinking of the led and the second reactivates it.

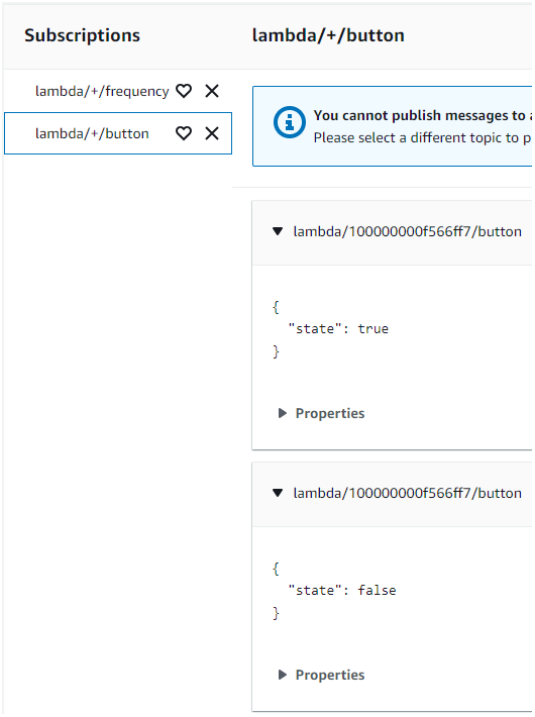


Figure 5.24 *Blink status changes on AWS*

Figure 5.25 shows the output message thread of the led application. It is possible to view the MQTT messages received from the button application on the Raspberry Pi 4.

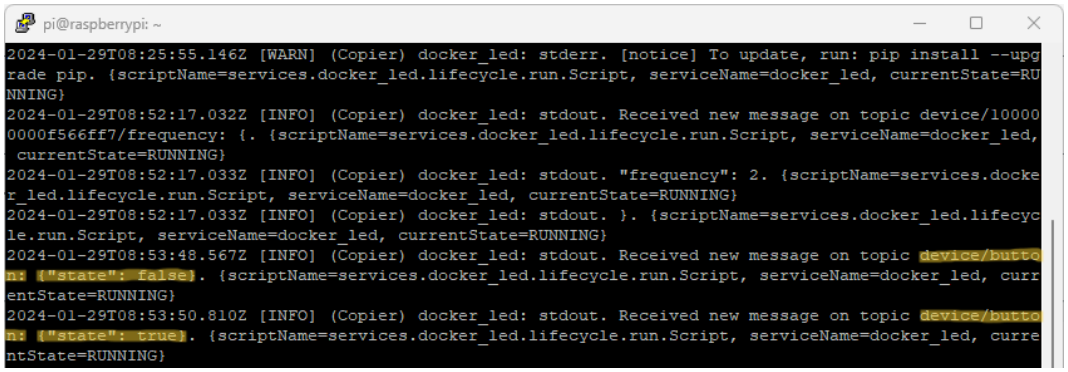


Figure 5.25 *Blink status changes on the Raspberry Pi 4*

Finally, the certificate was rotated. The triggering of a change alert was simulated from the AWS MQTT interface by requesting the creation of CSR on the Raspberry Pi 4. Several MQTT messages were exchanged. These are partly visible from the AWS interface in figure 5.26. When analysing the digital twin, the certificate identifier was changed, proving that the rotation had indeed taken place. The Greengrass service was also restarted to take the new certificate into account.

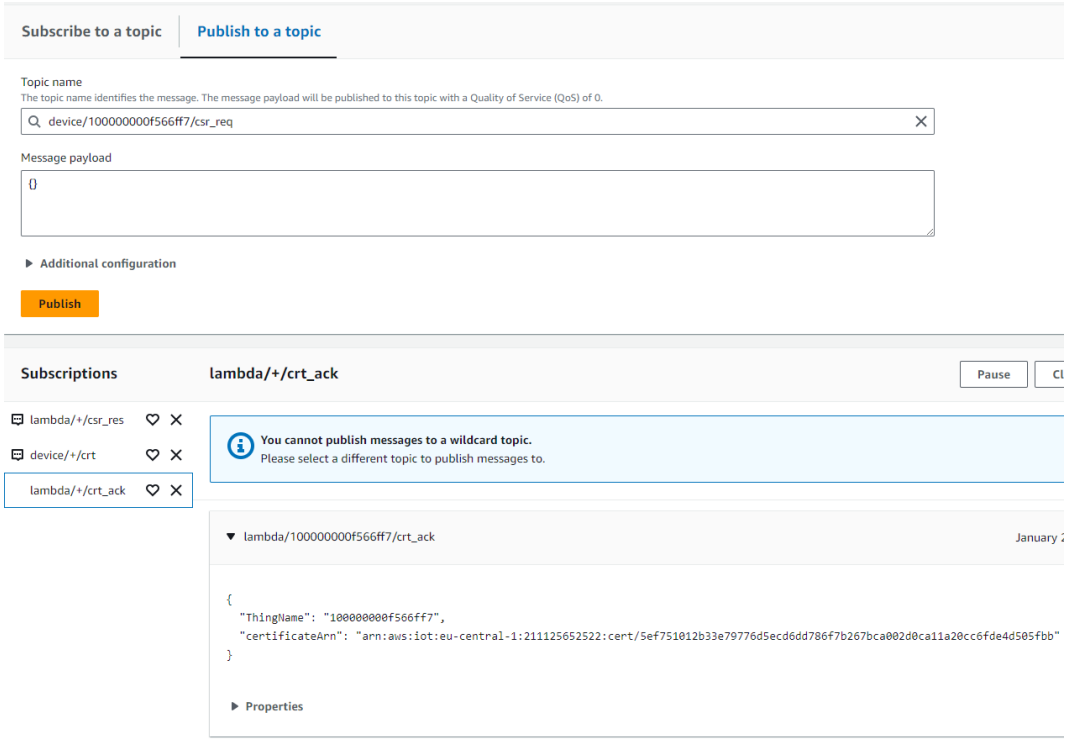


Figure 5.26 *Certificate rotation on AWS*

Figure 5.27 shows the output message thread of the certificate rotation application. It is possible to view the MQTT messages received from AWS. A request for CSR has been received as well as the new certificate.

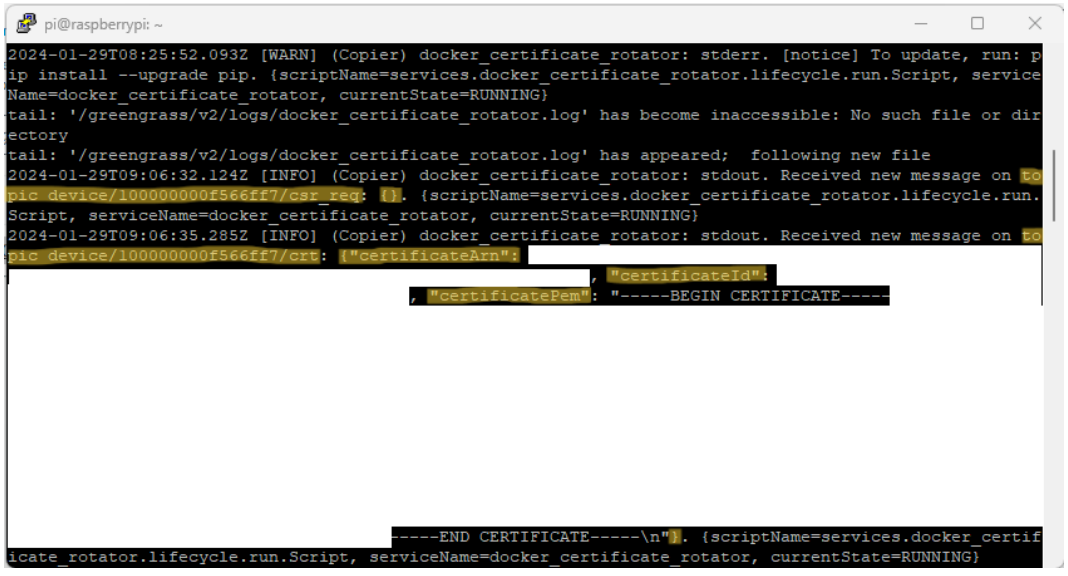


Figure 5.27 *Certificate rotation on the Raspberry Pi 4*

Due to the incorrect platform specified for the corrupted application, it was unable to start up correctly. This is clearly shown in figure 5.28. Greengrass tries to start the application three times before stopping.

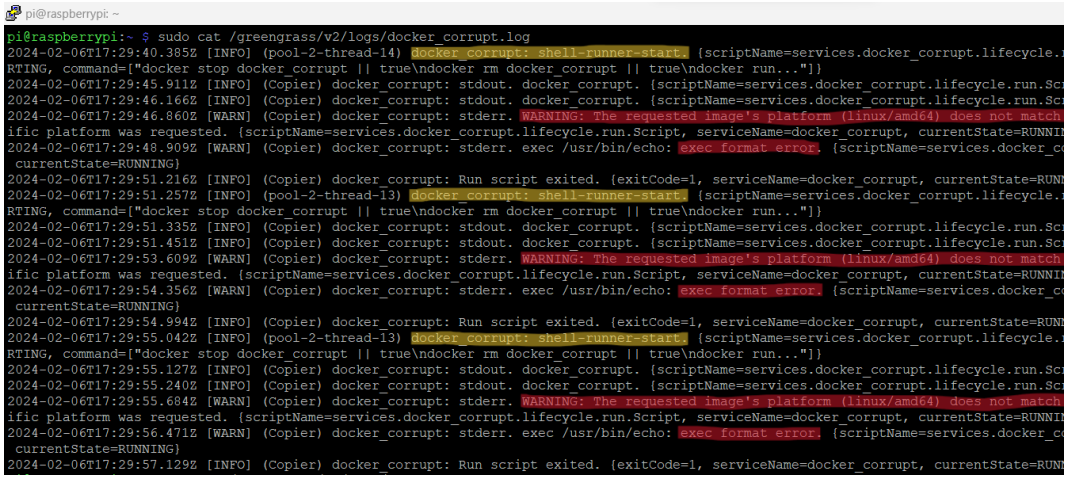


Figure 5.28 Launching the corrupt application on the Raspberry Pi 4

During testing of the components, an additional observation was made. If a component is corrupted and subsequently deployed with updates to other components, these components will retain the old version. As long as the corrupted application is not corrected, all the components remain frozen on the last version deployed, and not on the last version of the component.

5.5.2 Updating an application

In order to test the deployment of an update, the corrupted application was corrected by specifying the correct architecture. The Greengrass component has been updated since it has changed version (figure 5.29).

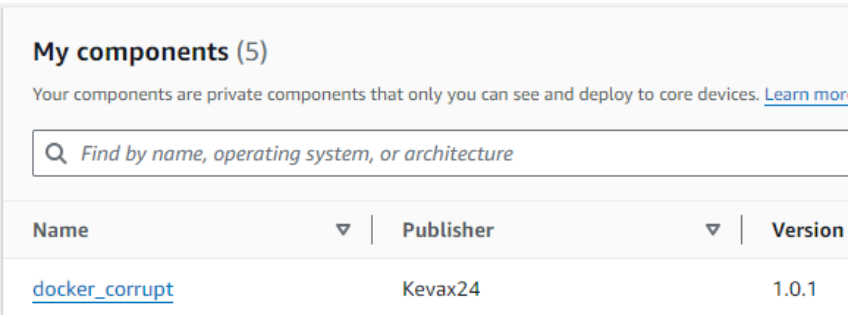


Figure 5.29 New version of the corrupted application on AWS

Looking at the Raspberry Pi 4 console in figure 5.30, the application ran correctly. A text was printed as agreed.

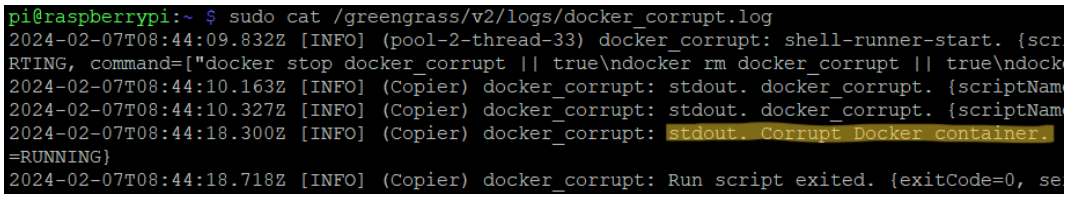


Figure 5.30 Preview of the corrupted application on the Raspberry Pi 4

Now that no components have been corrupted, the health of the IoT device is good. This was noted on the AWS console.

5.5.3 OS update

An update simulation was tested. The commands written were executed correctly and the Raspberry Pi 4 rebooted automatically without restarting the command lines. Evidence of this can be seen in figure 5.31. The first time it was run, the *apt-get update* command was issued, followed by the restart command. When the Greengrass component is launched a second time, it is noted that the update has already taken place.

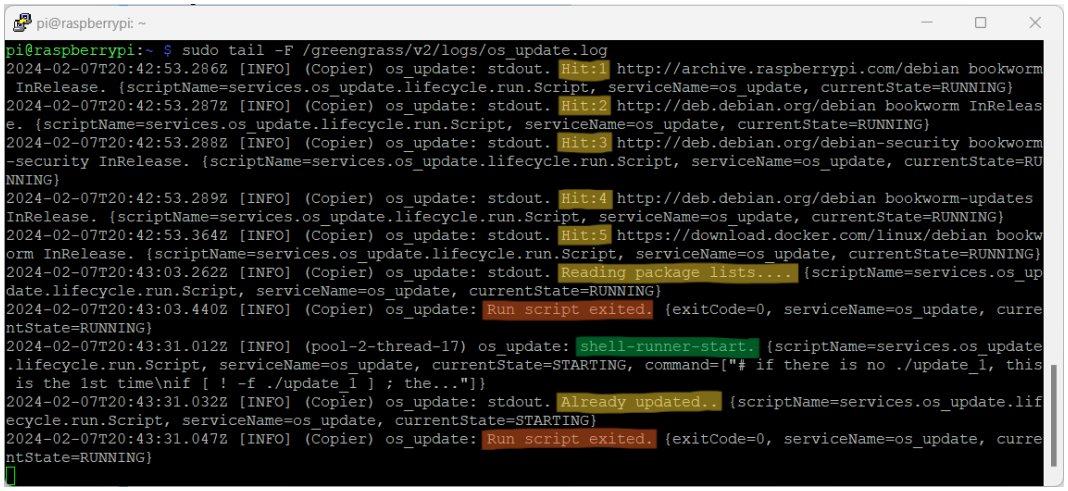


Figure 5.31 Behaviour of the OS update application

5.6 Cloud infrastructure destruction

The cloud infrastructure destruction was tested. To do this, the corresponding workflow was launched manually from the GitHub Actions console and passed successfully (figure 5.32). It takes about 4 minutes. All the resources created by the Pulumi tool have been deleted. The Raspberry Pi 4 remains provisioned with its unique certificate. The claim certificate remains available in the S3 bucket with its private key. The Greengrass components have not been removed either.

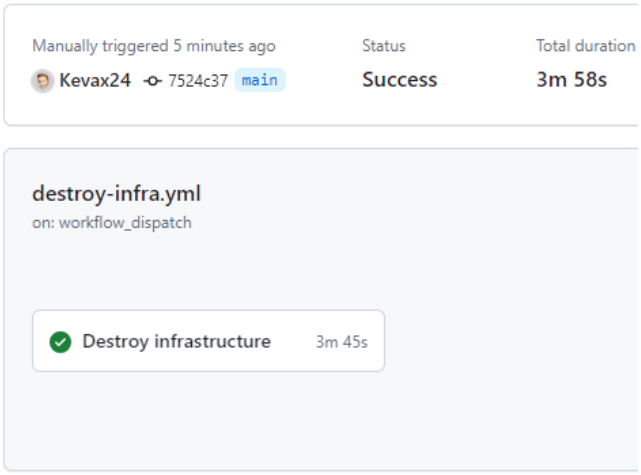


Figure 5.32 *Cloud infrastructure destruction successfully completed*

The Raspberry Pi 4's digital twin is still contained within AWS IoT. However, it has lost any policy for interacting with the cloud. The MQTT connection is therefore interrupted except locally on the device between the components. This was proven by pressing the external button on the Raspberry Pi 4 just once. The led stopped blinking and no status message was received on AWS IoT.

5.7 Management of authorised embedded systems

5.7.1 Infrastructure deployment with a provisioned embedded system

Following the logic of the tests, the infrastructure was redeployed with the Raspberry Pi 4 already provisioned. Its digital twin was attached to the group of devices and to an IoT policy. MQTT communication with the cloud is now back up and running. Figure 5.33 shows the attachment of the device to the new cloud infrastructure in the GitHub Actions console.

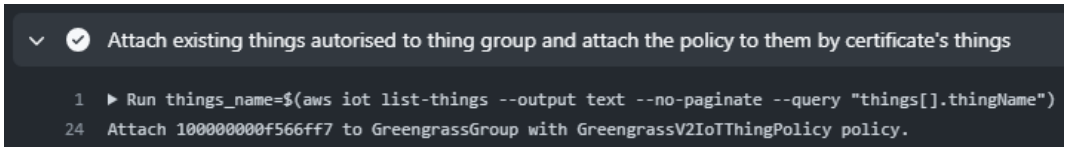


Figure 5.33 *Integration of Raspberry Pi 4 already provisioned*

5.7.2 Deleting a provisioned embedded system

It is possible to remove a device from the AWS cloud by deleting its serial number from the list of authorised devices. The serial number of the provisioned Raspberry Pi 4 has therefore been removed and this change has been pushed into Git. By following the GitHub Actions console in figure 5.34, it is possible to confirm that the device has been deleted. By browsing the AWS console, its digital twin no longer exists and no MQTT message reaches AWS IoT.

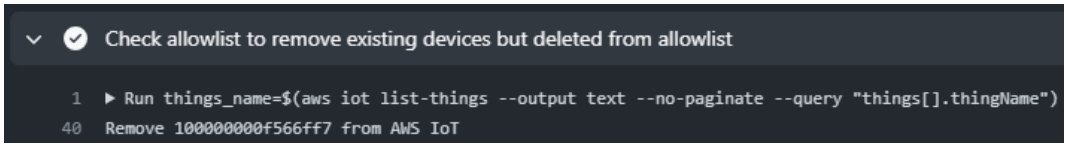


Figure 5.34 *Removal of the provisioned Raspberry Pi 4*

For this Raspberry Pi 4 to be provisioned again, it must be authorised again and its SD card must be flashed again with the OS image.

5.7.3 Unauthorised provisioning of an embedded system

This same Raspberry Pi 4 was tested for reintegration into the cloud infrastructure without being authorised. Its serial number was therefore not included in the list of authorised devices. Its SD card was flashed with the same OS image as for all the other tests carried out. Both software, Docker Engine and AWS IoT Greengrass, were successfully installed. During the provisioning attempt, the Lambda function, triggered at each attempt, detected that this Raspberry Pi 4 was not authorised to be provisioned.

Its certificate, which had been prepared and was supposed to be delivered to it, was therefore left waiting to be activated. After a while, the certificate disappears. This can be seen in figure 5.35.

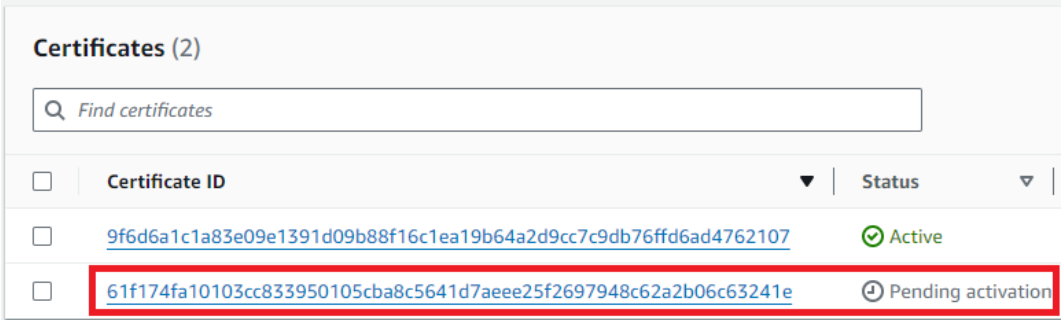


Figure 5.35 Certificate pending activation

5.8 Provisioning of other Arm SystemReady certified embedded systems

The aim was to extend the provisioning capability to other Arm SystemReady certified embedded systems, beyond the Raspberry Pi 4. Unfortunately, this attempt was not successful. Another device tested was the I-Pi SMARC IMX8M Plus board, which is Arm SystemReady IR certified. Despite flashing the same OS image as the Raspberry Pi 4 onto the SD card of this new device, booting failed. It seems that the *Raspberry Pi OS Lite* base image is specific to Raspberry devices. It is likely that each manufacturer prepares separate OS images based on the same Linux distribution for their devices, even with Arm SystemReady certification. Another custom OS image would surely have had to be created specifically for the I-Pi SMARC IMX8M Plus board to boot. Each image can be adapted using the Packer tool and its plugin for Arm architectures.

As a partial workaround to this failure, a virtual Raspberry Pi 4 was tested on an Arm development tool called **A**rm **V**irtual **H**ardware (AVH) [80]. AVH enables IoT development kits, Arm-based processors and cloud-based systems to be virtualised, speeding up the development of IoT software. In this type of tool, it was not possible to integrate the custom OS image. Among the options available, the basic *Raspberry Pi OS* image was selected. By connecting via SSH, the provisioning files were manually copied to the virtual device (figure 5.36). An arbitrary serial number was specified and added to the list of authorised devices.

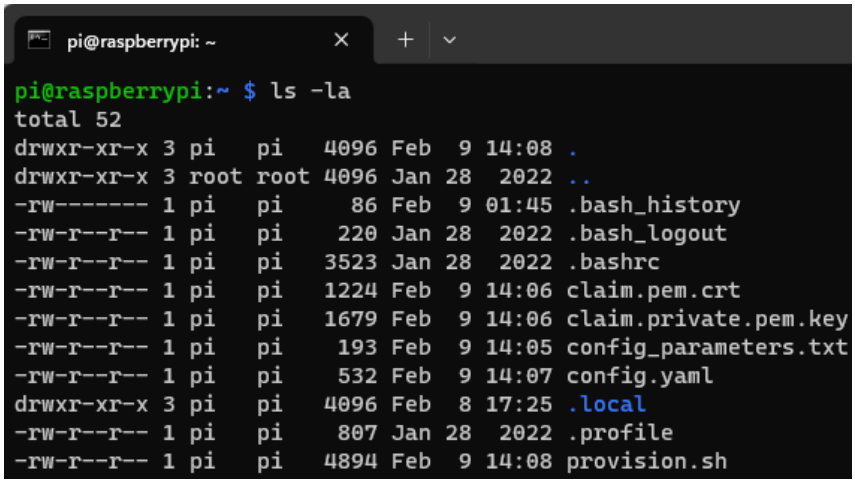


Figure 5.36 *Access to the Raspberry Pi 4 on AVH*

The *provision.sh* script was run manually as administrator. The installation of Docker Engine and AWS IoT Greengrass Core was successful, as was the provisioning. Its digital twin was created in AWS IoT, as shown in figure 5.37.

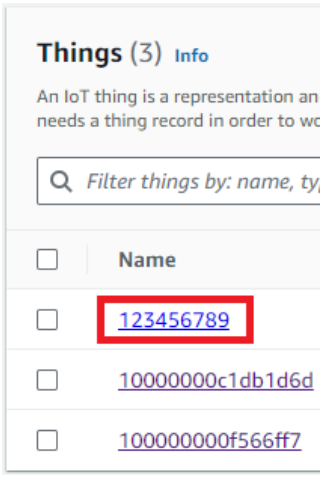


Figure 5.37 *Virtual device provisioning confirmed on AWS*

Finally, the Greengrass components were successfully deployed (figure 5.38). The health status of the device returned to AWS is not good due to the led and button applications not launching correctly. This is due to the GPIO library reserved exclusively for Raspberry that is not available on AVH. It is nevertheless remarkable that the certificate rotation application is running. In addition, the OS update application completed correctly and the device rebooted as expected. The corrupted application was executed correctly since in an earlier test its problem had been solved.

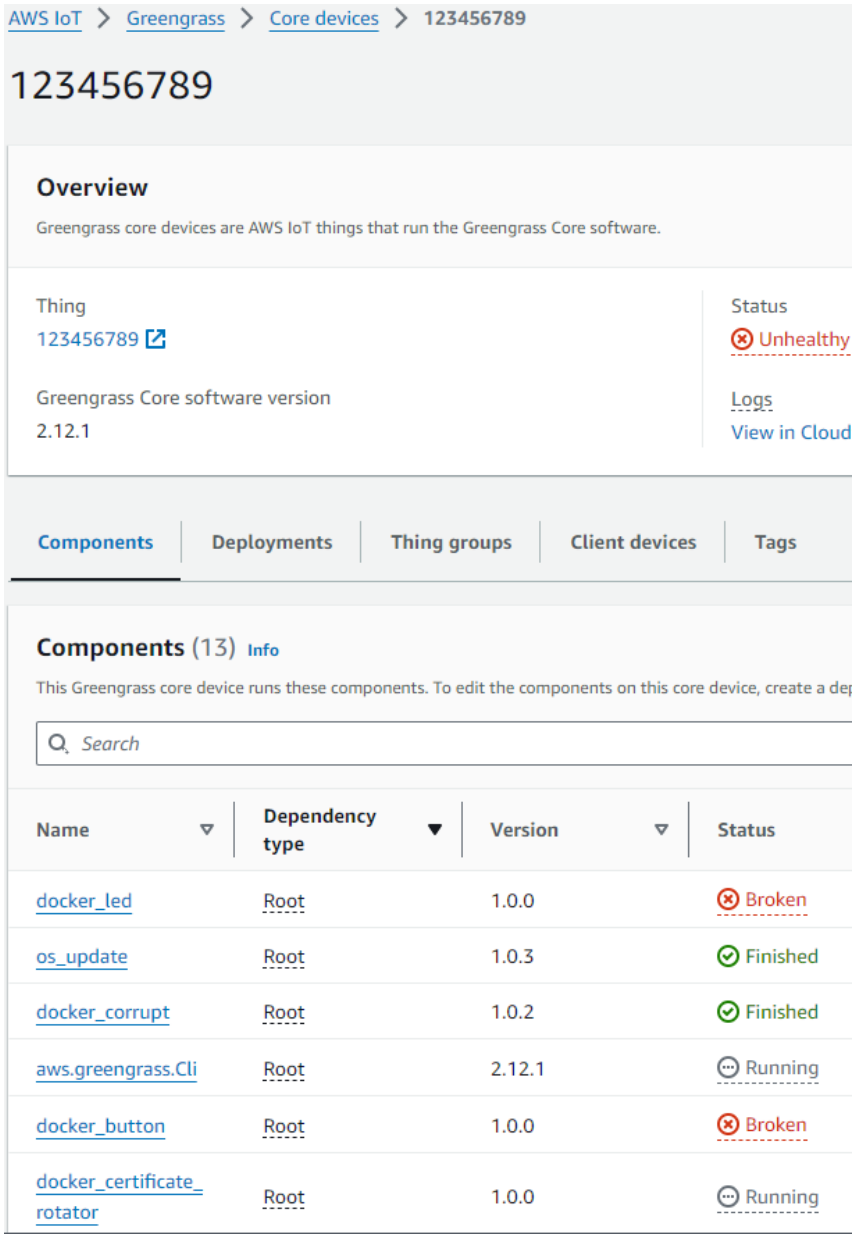


Figure 5.38 Deploying Greengrass components on the virtual device

5.9 Cloud infrastructure cost

An assessment of the costs of the AWS cloud infrastructure was carried out in order to understand the monthly and annual expenses to be expected. This estimate was carried out using the AWS Pricing Calculator. The estimate is shown in figure 5.39 and in appendix B.

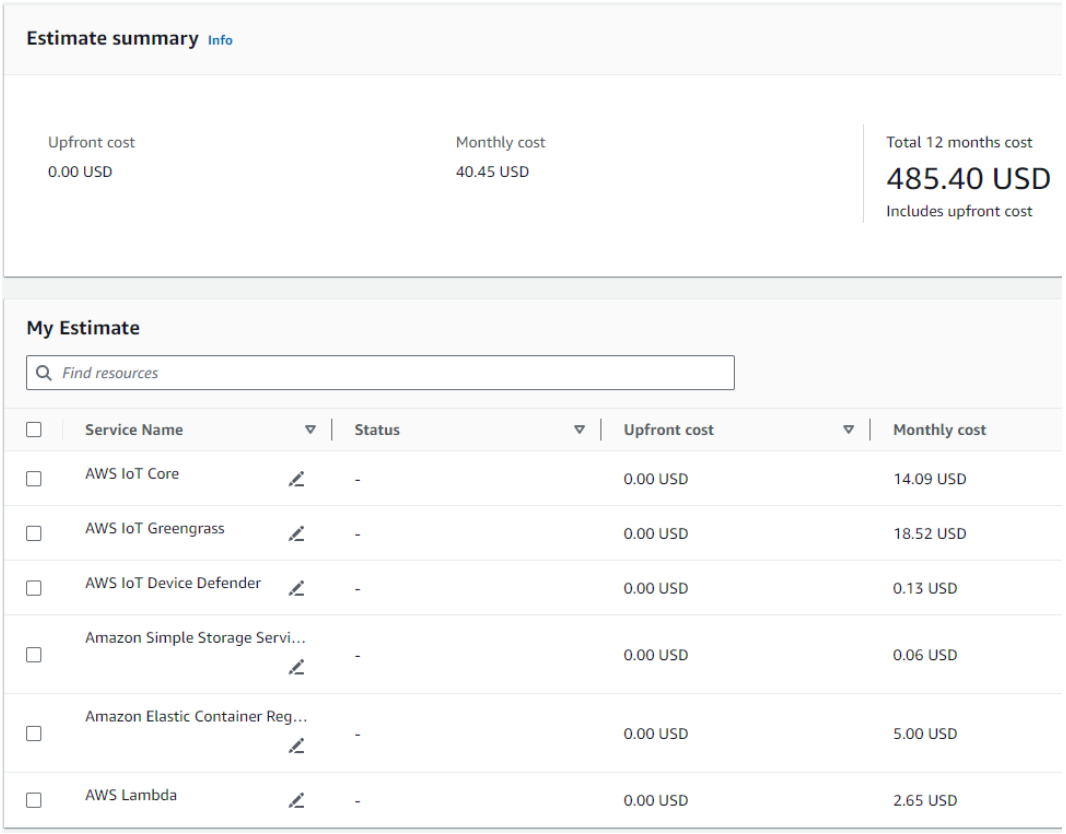


Figure 5.39 *Cloud infrastructure cost estimate*

The evaluation is based on a number of factors, assuming that it is the cost of the infrastructure in the production environment. The region chosen is Frankfurt (*eu-central-1*). A fleet of 100 embedded systems is considered.

AWS IoT Core is a high-cost service. Here is a list of factors taken into account :

- Number of devices : 100
- Protocol : MQTT
- Average size per message : 10 kB
- Number of messages per month per device : 43'800 (1 per minute)
- Number of rules triggered per month per device : 43'800

Each device sends one 10 kB MQTT message per minute. Each message received in AWS IoT Core triggers a rule that diverts the message to another service. For example, Lambda functions can be called by these rules depending on the message heading.

AWS IoT Greengrass is the most expensive service. Here are the factors taken into consideration:

- Number of devices : 100
- Activity period per month : 43'800 minutes (100% activity)
- Number of MQTT topics : 1000

With 1000 MQTT topics, each device can have 10 unique topics each.

The AWS IoT Device Defender service is used to perform a daily audit by monitoring a single aspect, namely the expiry of certificates. The cost is virtually insignificant.

The AWS Lambda service is used to execute operations when a function is triggered by a received message. It is estimated that each message received triggers a function, with a total of 4'380'000 messages sent per month for the fleet of 100 embedded systems. The average execution time for a Lambda function is one second.

Amazon **E**lastic **C**ontainer **R**egistry (ECR) is used to store the Docker images generated for the applications developed. A monthly storage limit of 50 GB has been defined. None of the application images in this project exceeds 400 MB. Therefore, there can be up to 125 images of this size to reach the 50 GB limit.

Amazon S3 is also used for storage, with three S3 buckets in use. The bucket containing the OS image for this reference architecture weighs 2.6 GB. The total of the three buckets remains at 2.6 GB, indicating that the Pulumi files on the state of the infrastructure, in addition to the provisioning files, do not exceed 100 MB.

The Amazon SNS service used in the reference architecture is not taken into account, as it is triggered every so many years (a certificate has a lifetime of several years by default). Consequently, its cost is negligible.

5.10 Observations

During this study, deployment incidents were occasionally noted when using the Pulumi tool. It was observed that certain resources, which are dependent on one another, can sometimes be called up before their creation is complete, despite this dependency being specified in the infrastructure description. This leads to errors during deployment. To solve this problem, simply restart the deployment workflow to allow the infrastructure to create the last missing resources.

Another observation concerns AWS regions. It should be noted that this reference architecture cannot be deployed in all the regions offered by AWS. Some recent resources, such as AWS IoT Greengrass, are not yet available in all regions. Development has focused exclusively on the Frankfurt region (*eu-central-1*).

6 Project methodology

This chapter covers all aspects of project management. A project plan is outlined, describing how the research was carried out. The Agile and KanBan methodologies are presented. Finally, the chapter concludes with a description of the management tools used and an account of the training received.

Contents

6.1 Project plan

At the start of the project, a research plan was drawn up. It is presented in figure 6.1 in the form of work packages.

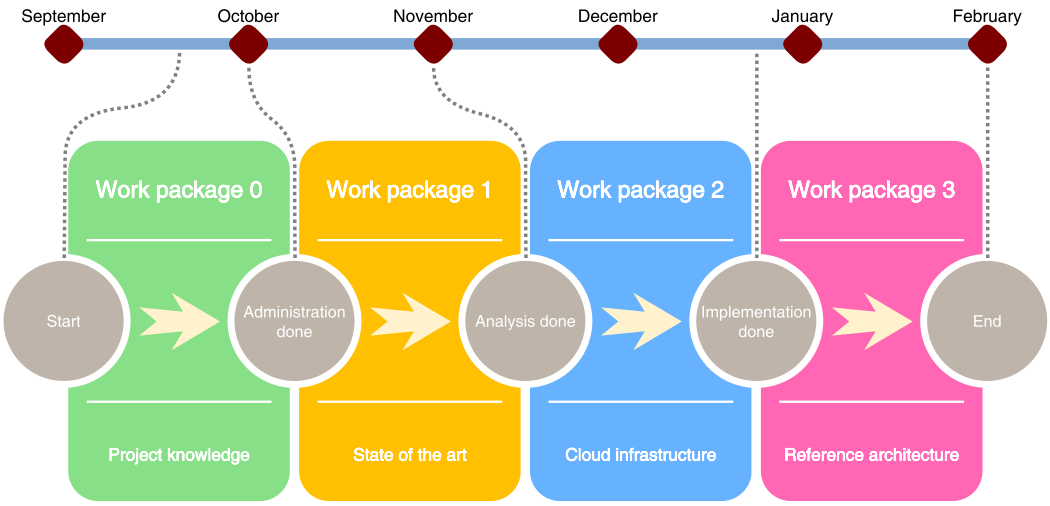


Figure 6.1 Project plan

The first work package covers all administrative activities. The aim of this initial phase was to gain an in-depth understanding of the project, become aware of the issues and clearly define the specifications. It lasted around two weeks.

About a month was spent researching existing solutions that were closest to the project through a state of the art, involving consultation of scientific articles and other documentation.

The third work package focuses on the cloud infrastructure, with around two months dedicated to this implementation phase. This part includes the creation of the CI/CD process.

In the latest work package, completion of the reference architecture was scheduled with just over a month remaining. It is at this stage that the final functionalities are developed. This is also the phase when the whole package must undergo testing and validation before being released as open source on a shared repository.

6.2 Research methodology

The research methodology adopted in this thesis began with a state of the art survey of the existing solutions most relevant to this project. Next, an overview of the reference architecture was established to facilitate implementation. A judicious choice of development tools was also made. In parallel, the implementation was carefully designed, followed by validation of the results through a proof of concept.

Throughout the process, agile methodology, in particular the Scrum method, was used, enabling iterative project management. Weekly meetings were scheduled with the professor responsible for the work, as well as with certain company staff on an optional

basis. To ensure that the project progressed efficiently, the KanBan method was used. This approach uses virtual cards to represent each task, organised in a table to clearly track progress. It is also used to break the project down into smaller parts to more accurately estimate the time required throughout the process.

6.3 Literature search

Literature search has focused on academic search engines. The following search engines were used :

- Google Scholar
- IEEE Xplore
- ScienceDirect

The sources were mainly conference papers. Some information was found on websites. The following keywords were used to find sources that met the requirements of this analysis:

- integration
- reference architecture
- embedded systems
- IoT
- Cloud-Native
- cloud
- cloud computing
- cloud providers
- cloud platforms
- **I**nfrastructure **a**s **C**ode
- IaC tools
- Arm SystemReady

6.4 Agile methodology

Agile methodology is the main project management method used in this work. It enables the project to be managed from the specifications to the final product. The idea for this method was conceived in the 70s or even before [81]. The idea was to change the development process in software engineering by using iterative techniques. Traditional methods worked with sequential techniques, often called Waterfall. Figure 6.2 shows the difference between these two approaches.

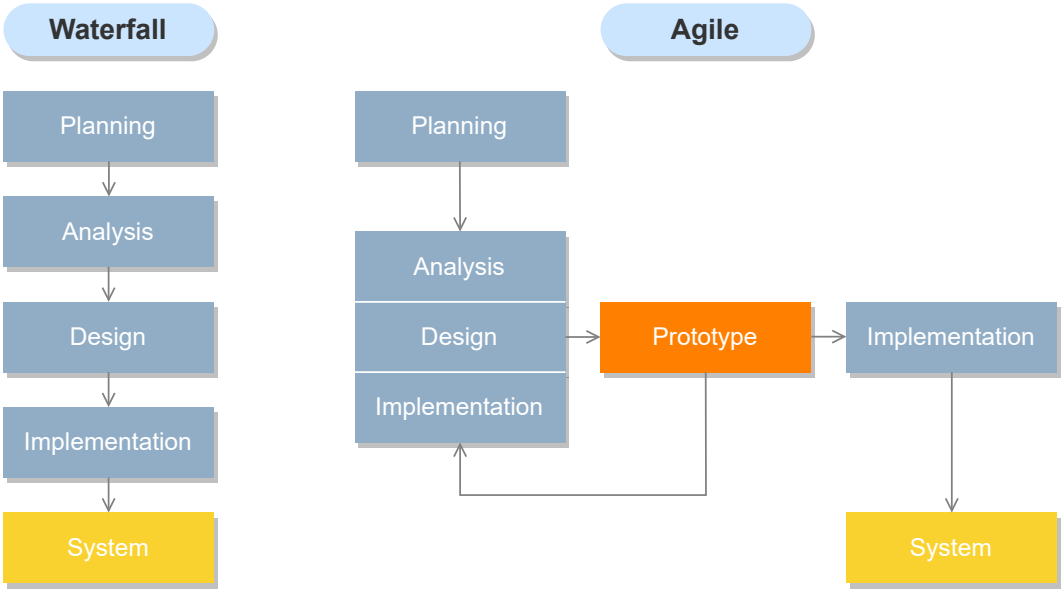


Figure 6.2 Sequential and iterative process [82]

The Waterfall methodology is considered cumbersome. There are deliverables after each phase. Approval is required before moving on to the next phase. It is difficult to go backwards (by moving up the phases). Professors explained this very well during a Masters course given at the HES-SO [82]. They added that this method is best used in very large projects where it is difficult to split teams into small groups to work iteratively.

These same teachers [82] described the agile methodology as follows:

> *"AGILE" is about values and principles, not practices, but many practices support them. It's not about doing agile, it's about being agile.*

Several agile process frameworks were created before the 2000s. Examples include Scrum, XP, RUP, etc. Between the 11th and the 13th of February 2001, 17 people from these different frameworks met to find an alternative to cumbersome, documentation-driven software development processes. They created the agile software development manifesto [83] :

> *Individuals and interactions over processes and tools*
> *Working software over comprehensive documentation*
> *Customer collaboration over contract negotiation*
> *Responding to change over following a plan*

The central principle of agile is to have very rapid iterations to build the first business values. The customer is at the centre of the process thanks to frequent communication with the development team [82]. The process framework chosen for this project is Scrum. It is very well suited to small projects with small teams. It enables the first software prototypes to be delivered quickly. It adapts easily to changes, unlike a sequential method. It is also based on experience. The idea is to learn continuously through

iterations. This is an important point in terms of learning from new developments in the project. This is one of the most popular agile methods. This approach was launched in 1995 by Ken Schwaber and Jeff Sutherland [84].

6.4.1 Scrum roles

Figure 6.3 shows the different roles in a Scrum team.

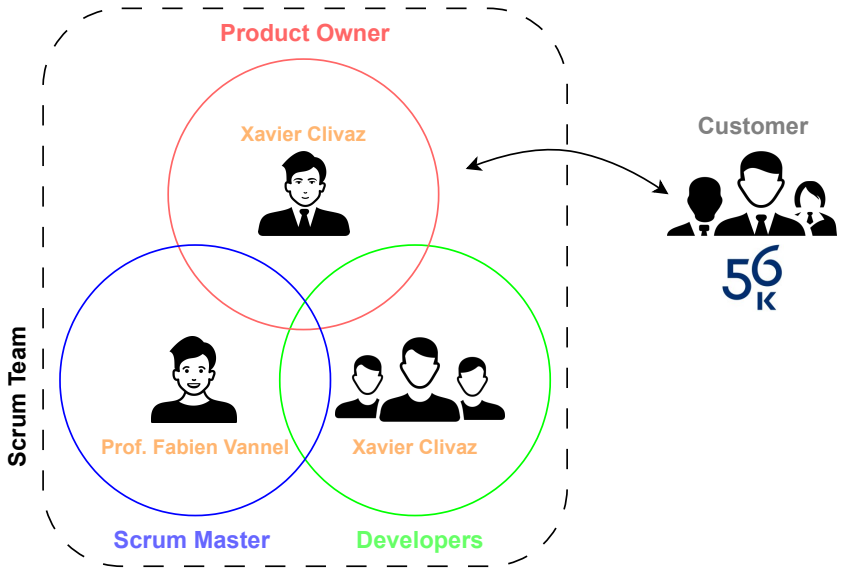


Figure 6.3 Scrum roles

A Scrum guide has been written by Jeff Sutherland and Ken Schwaber to explain the rules [85]. First of all, there is the **P**roduct **O**wner (PO). This is the customer's spokesperson. It is he who defines the product's functionalities. They are recorded in a Product Backlog in the form of tasks. He must manage these tasks by prioritising some of them. He is also responsible for the value and return on investment (ROI) of the product. The development team is generally a small one (around seven people [82]). It is capable of self-management. There are no specific roles or positions. Developers have to deal with tasks that are chosen from the Sprint Backlog. Finally, the Scrum Master is required to be of service to the team. He accompanies the team of developers and prevents obstacles from getting in the way of progress. It is the Scrum Master who establishes the practices and rules of Scrum.

6.4.2 Scrum process

The Scrum process is relatively simple. It follows the theory described in the official Scrum guide [85]. The life cycle is shown in figure 6.4.

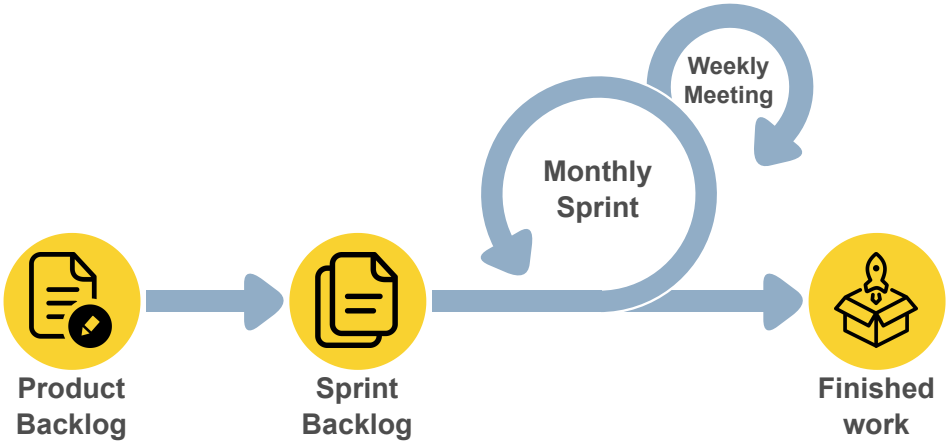


Figure 6.4 *Scrum life cycle*

The process is made up of multiple elements. It begins on the left of the diagram with the Product Backlog. This is a backlog that is filled by the PO. Since the Product Backlog is generally filled with many functionalities divided into tasks, only one part needs to be selected to be placed in the Sprint Backlog. A Sprint generally corresponds to a period of one month during which the tasks in the Sprint Backlog must be completed. During a Sprint, team meetings are organised every day. These are called Daily Meetings and usually last 15 minutes. A Daily Meeting ensures that, at least once a day, the entire team is available to get support on any problems encountered. In this project, only weekly meetings are held, depending on the distance separating the Scrum Master and the team of developers. Before each Sprint, a Sprint Planning is carried out to determine the tasks from the Product Backlog to be put into the Sprint Backlog. At the end of a Sprint, an inspection is made of how the Sprint went, with the aim of improving quality and efficiency. This is called a Sprint Retrospective. Finally, it's important to remember that at the end of each Sprint, one stage of the final product is completed. Sometimes this is called an increment.

6.5 KanBan methodology

The KanBan methodology is very well integrated into Scrum. Its aim is to manage the workflow as efficiently as possible. According to the KanBan guide produced by Scrum.org [86], several fundamental metrics should be taken into account for the flow:

- **Work In Progress (WIP)** : the number of tasks/items started and not completed.
- **Cycle Time** : the time elapsed between the start and end of a task.
- **Work Item Age** : the time elapsed between the start of the task and now.
- **Throughput** : the number of tasks completed per unit of time.

The average cycle time can be predicted using Little's Law [87]:

$$average\ cycle\ time = \frac{average\ WIP}{average\ throughput} \tag{6.1}$$

It explains that the more tasks there are in progress, the longer it will take to complete them. A cycle time in KanBan can be correlated to a Sprint in the Scrum methodology. Using this methodology, it is therefore possible to define the right number of tasks for each Sprint Planning. Note that the more you divide a feature into smaller tasks, the easier it will be to predict the cycle time.

KanBan does more than just limit the WIP. It provides a transparent display of the workflow. Tasks can be represented in a table, as shown in figure 6.5. In this case, the table is virtual and can be viewed from anywhere by the whole Scrum team.

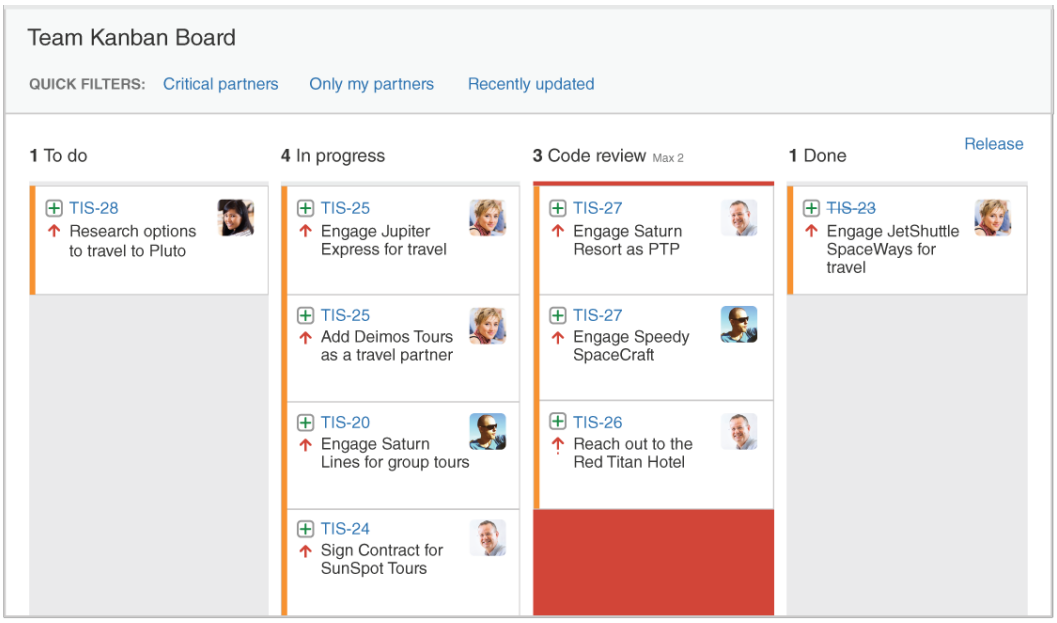


Figure 6.5 Example of a virtual KanBan table [88]

Each label in the table corresponds to a task. A task can contain a priority level and is often assigned to a member of the Scrum team. All this is defined during Sprint Planning. The labels are categorised in columns to show their progress.

6.6 Project management tools

6.6.1 Version management

Version management, embodied by tools such as Git, remains an essential part of the software development ecosystem. This practice offers a structured approach to tracking and maintaining changes to a project's code. One of the major benefits lies in its ability to keep a complete history of changes, facilitating collaboration between team members working on the same project.

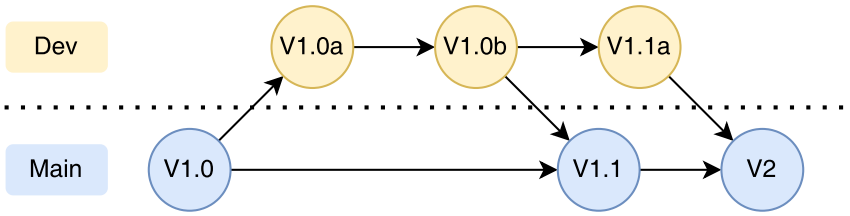


Figure 6.6 Example of version management with two branches

An example of a visual representation of a project is shown in figure 6.6, comprising several distinct versions and branches. Branches allow developers to work on alternative versions of the source code, facilitating parallel development, feature isolation and efficient change management without affecting the main branch.

When part of the development is considered complete, it becomes common practice to create a new version. This version freezes the current state of the project at a specific point in time, providing a clear and stable reference. This ability to create versions means that it is possible to return to previous states of the project if problems arise, whether identified by members of the development team or by external stakeholders, such as customers.

In short, version management, embodied by Git, transcends its simple function of tracking changes to become a central pillar of collaborative working, risk management and preserving the integrity of a software project. Its widespread adoption across the industry is testament to its continuing importance in delivering robust, scalable software development projects.

6.6.2 Continuous Intergration (CI) and Continuous Delivery (CD)

CI/CD (**C**ontinuous **I**ntergration/**C**ontinuous **D**elivery) integrates synergistically with version management, and in particular with the DevOps framework, as illustrated in figure 6.7. The aim of the DevOps method is to automate software development processes, thereby encouraging collaboration between development teams (grey area) and operational teams (pink area). Two major components of CI/CD have emerged from this approach, namely **C**ontinuous **I**ntergration (CI) and **C**ontinuous **D**elivery (CD), which have become fundamental pillars for developers.

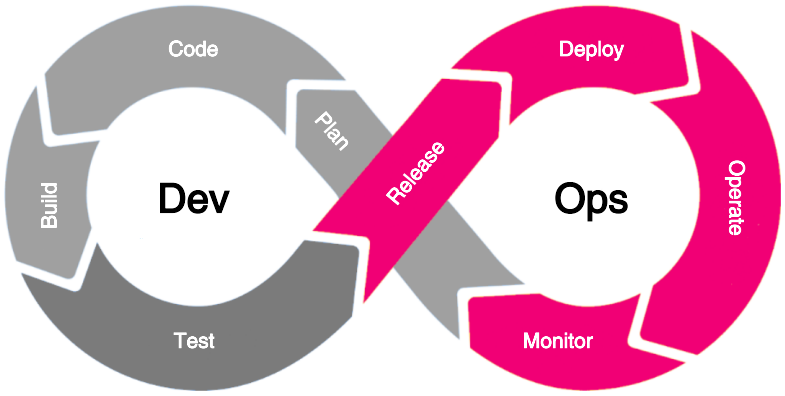


Figure 6.7 DevOps method

CI involves automating various processes with each new software release. These processes typically include building, testing and deploying the software design, as shown in figure 6.7. The main motivation for inventing the CI was to reduce the time needed between launching a project and bringing it to market, a concept known as time-to-market. This automation takes advantage of the constant evolution of machines, contrasting with

the physical and moral limits of human beings. The benefits of CI also extend to team collaboration, simplified debugging, reduced costs, improved software quality, and much more.

CD goes hand in hand with the CI, aimed at automating the deployment of software to production environments. It eliminates the tedious manual steps associated with deployment, ensuring fast, reliable and frequent delivery of new features and software updates.

CI/CD represents a modern and essential approach to the software development lifecycle, offering efficiency gains, improved team collaboration, accelerated time-to-market and an overall improvement in the quality of software products.

6.6.3 Work packages

Work packages are specific, delimited elements of a project which are assigned to a team or an individual for completion. Each work package represents a well-defined task or set of tasks, often associated with concrete deliverables. They facilitate the management and monitoring of the project by allowing a clear division of responsibilities and helping to assess progress. In short, work packages are manageable units that contribute to the achievement of an overall project.

6.7 Training

6.7.1 AWS Certified Cloud Practitioner

AWS offers a number of certifications designed to validate the skills and knowledge of IT professionals working on the AWS platform. These certifications are designed to cover a variety of areas, from cloud architecture and operations management to security, databases, application development and more.

These certifications are widely recognised in the industry and can help professionals demonstrate their skills and progress in their cloud careers. It is important to note that the AWS certification programme is evolving, and new certifications may be added as new services and features are introduced to the AWS platform.

The certification achieved as part of this project was as follows : **AWS Certified Cloud Practitioner**. This is the first certification to give a general overview of a cloud platform to professionals with no previous experience in this field. AWS defines it as follows :

> *The AWS Certified Cloud Practitioner validates foundational, high-level understanding of AWS Cloud, services, and terminology. This is a good starting point on the AWS Certification journey for individuals with no prior IT or cloud experience switching to a cloud career or for line-of-business employees looking for foundational cloud literacy. [89]*

Online lessons were taken at the start of the project and, within a month, an official exam was successfully passed.

Figure 6.8 AWS Certified Cloud Practitioner

7 Conclusions

7.1 Project summary

In summary, this project has resulted in the creation of an infrastructure that enables the fluid, automated and secure integration of IoT devices into an AWS cloud environment. The reference architecture aims to simplify the integration of embedded systems within an AWS cloud infrastructure dedicated to the IoT. The devices used in this IoT network feature Arm architecture and are Arm SystemReady certified. To improve this integration, a DevOps approach has been put in place thanks to a continuous integration and distribution process orchestrated by GitHub Actions. The central tool for deploying the infrastructure is Pulumi, which enables AWS cloud infrastructure to be deployed on various environments. Python applications have been specially developed to demonstrate how integration works via MQTT communication between devices and AWS IoT. In the CI/CD process, these applications are tested, published in AWS and then deployed to a fleet of devices. During these deployments, the applications run in Docker containers, ensuring greater portability. A custom OS image, based on Raspberry Pi OS Lite, has been developed, incorporating Docker for applications and AWS IoT Greengrass Core, which is essential for provisioning and integration. This OS image is downloaded, flashed onto an SD card and inserted into an embedded system, which is automatically provisioned in a few minutes when it is first booted up. The successful integration of a first Raspberry Pi 4 has been extended to a second Raspberry Pi 4, demonstrating the scalability of the architecture. To guarantee network reliability, security mechanisms are built into the architecture, making communications within the IoT system even more robust and protected.

7.2 Comparison with the initial objectives

The project was an overall success, achieving the main objective of designing a reference architecture to facilitate the deployment of a cloud infrastructure while automating the provisioning of a fleet of embedded systems. Now available as open source, this achievement provides a solid foundation for engineers looking for similar solutions. Detailed documentation accompanies the project to make it easier to understand and use.

The Cloud-Native approach and the DevOps methodology, with continuous integration and continuous distribution, played an essential role in the development of the architecture. These choices reinforced the robustness of the architecture, simplifying its use and enabling the integration of missing elements specific to each solution.

The AWS cloud infrastructure is totally modular, offering the possibility of adding the additional services required for the product. The use of the Pulumi IaC tool, integrated into this architecture, facilitates this flexibility.

The integration of the embedded systems has been carried out in such a way as to facilitate the deployment, updating and interaction of the applications, while guaranteeing the security of the IoT network. The use of Docker, well accepted by the central AWS IoT Greengrass Core service, enables applications to be launched in containers, offering

greater portability. What's more, Docker images are easily accessible in a dedicated registry in the cloud. Data flows seamlessly between IoT devices and the AWS cloud. It can be viewed from the AWS client console.

However, a challenge remains in terms of compatibility with embedded systems. Although Arm and Arm SystemReady certified devices were taken into account, booting from the same OS image was only possible with the Raspberry Pi 4, restricting the scope of the architecture. The integration of the Packer tool nevertheless offers a solution for modifying the creation of the OS image, enabling the base distribution and its configuration to be changed.

In terms of project management, the use of established methodologies such as Scrum and KanBan ensured that deadlines were met. Some deviations were observed, notably in the confusion of milestones. The term "proof of concept" turned out to be the reference architecture itself, encompassing the entire project. Despite some disparities between the forecast and actual schedules, the renamed milestones accurately reflect the different phases of the project. One notable difference is the amount of time devoted to process automation, which is taking almost as long as the implementation of the cloud infrastructure. The detailed schedules can be consulted in the appendix (A).

7.3 Encountered difficulties

In the course of this work, several challenges emerged. Firstly, the definition of the reference architecture. This proved difficult due to the vague nature of the term, which can vary from one domain to another. Positioning the components to be implemented to create a coherent reference architecture was complex and the overall vision was not always clear. However, by starting with the implementation of a few elements such as applications, it was possible to better define the objective to be achieved.

Next, the complexity of embedded systems compatibility was another difficulty to overcome. Creating an OS image compatible with various embedded systems proved to be a major challenge. The lack of time at the end of the project limited the possibility of finding a satisfactory solution.

On the whole, however, the tasks progressed without encountering any major obstacles.

7.4 Future perspectives

Various horizons are opening up for the future development of this project. The reference architecture is now freely available to any developer wishing to adopt it, accompanied by comprehensive documentation to make it easier to understand and use.

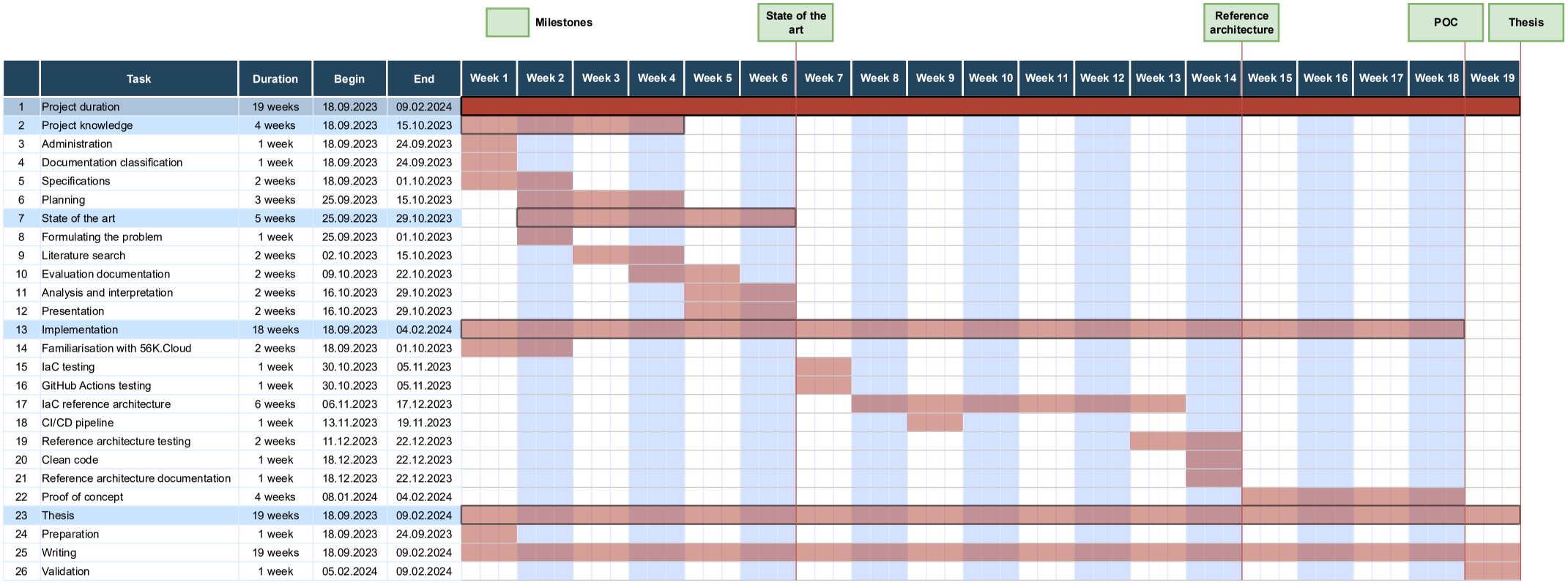
The main prospect would be to invest in research and development to extend the architecture's compatibility with a wider range of embedded systems, taking into account the Arm architecture and Arm SystemReady certification. One possible approach would be to modify the OS image creation process to design a generic version compatible with the entire range. If this is not possible, the creation of several OS images, listed by compatible device model, could be an alternative.

The integration of new cloud services is a promising prospect. Continued evolution of the architecture would mean incorporating cloud services and functionalities. Furthermore, applications developed for IoT devices could evolve to adapt to new emerging technologies, such as artificial intelligence.

Finally, the security of the cloud infrastructure could be strengthened. The use of additional AWS services could be explored to improve the monitoring of traffic on the infrastructure, in particular by integrating threat detection functionalities. This enhanced approach would help to ensure the security of data and connected IoT devices.

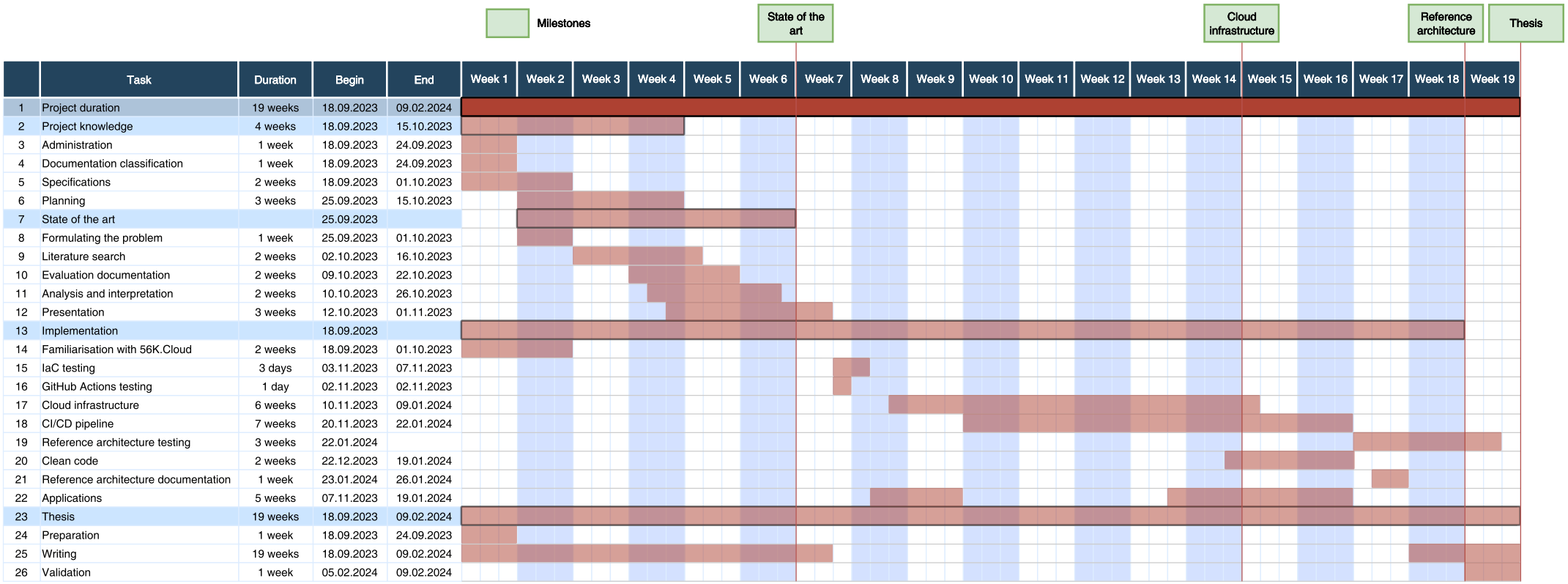
A | Plannings

A.1 Forward planning



	Task	Duration	Begin	End	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7	Week 8	Week 9	Week 10	Week 11	Week 12	Week 13	Week 14	Week 15	Week 16	Week 17	Week 18	Week 19
1	Project duration	19 weeks	18.09.2023	09.02.2024																			
2	Project knowledge	4 weeks	18.09.2023	15.10.2023																			
3	Administration	1 week	18.09.2023	24.09.2023																			
4	Documentation classification	1 week	18.09.2023	24.09.2023																			
5	Specifications	2 weeks	18.09.2023	01.10.2023																			
6	Planning	3 weeks	25.09.2023	15.10.2023																			
7	State of the art	5 weeks	25.09.2023	29.10.2023																			
8	Formulating the problem	1 week	25.09.2023	01.10.2023																			
9	Literature search	2 weeks	02.10.2023	15.10.2023																			
10	Evaluation documentation	2 weeks	09.10.2023	22.10.2023																			
11	Analysis and interpretation	2 weeks	16.10.2023	29.10.2023																			
12	Presentation	2 weeks	16.10.2023	29.10.2023																			
13	Implementation	18 weeks	18.09.2023	04.02.2024																			
14	Familiarisation with 56K.Cloud	2 weeks	18.09.2023	01.10.2023																			
15	IaC testing	1 week	30.10.2023	05.11.2023																			
16	GitHub Actions testing	1 week	30.10.2023	05.11.2023																			
17	IaC reference architecture	6 weeks	06.11.2023	17.12.2023																			
18	CI/CD pipeline	1 week	13.11.2023	19.11.2023																			
19	Reference architecture testing	2 weeks	11.12.2023	22.12.2023																			
20	Clean code	1 week	18.12.2023	22.12.2023																			
21	Reference architecture documentation	1 week	18.12.2023	22.12.2023																			
22	Proof of concept	4 weeks	08.01.2024	04.02.2024																			
23	Thesis	19 weeks	18.09.2023	09.02.2024																			
24	Preparation	1 week	18.09.2023	24.09.2023																			
25	Writing	19 weeks	18.09.2023	09.02.2024																			
26	Validation	1 week	05.02.2024	09.02.2024																			

A.2 Effective planning



	Task	Duration	Begin	End	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7	Week 8	Week 9	Week 10	Week 11	Week 12	Week 13	Week 14	Week 15	Week 16	Week 17	Week 18	Week 19
1	Project duration	19 weeks	18.09.2023	09.02.2024																			
2	Project knowledge	4 weeks	18.09.2023	15.10.2023																			
3	Administration	1 week	18.09.2023	24.09.2023																			
4	Documentation classification	1 week	18.09.2023	24.09.2023																			
5	Specifications	2 weeks	18.09.2023	01.10.2023																			
6	Planning	3 weeks	25.09.2023	15.10.2023																			
7	State of the art		25.09.2023																				
8	Formulating the problem	1 week	25.09.2023	01.10.2023																			
9	Literature search	2 weeks	02.10.2023	16.10.2023																			
10	Evaluation documentation	2 weeks	09.10.2023	22.10.2023																			
11	Analysis and interpretation	2 weeks	10.10.2023	26.10.2023																			
12	Presentation	3 weeks	12.10.2023	01.11.2023																			
13	Implementation		18.09.2023																				
14	Familiarisation with 56K.Cloud	2 weeks	18.09.2023	01.10.2023																			
15	IaC testing	3 days	03.11.2023	07.11.2023																			
16	GitHub Actions testing	1 day	02.11.2023	02.11.2023																			
17	Cloud infrastructure	6 weeks	10.11.2023	09.01.2024																			
18	CI/CD pipeline	7 weeks	20.11.2023	22.01.2024																			
19	Reference architecture testing	3 weeks	22.01.2024																				
20	Clean code	2 weeks	22.12.2023	19.01.2024																			
21	Reference architecture documentation	1 week	23.01.2024	26.01.2024																			
22	Applications	5 weeks	07.11.2023	19.01.2024																			
23	Thesis	19 weeks	18.09.2023	09.02.2024																			
24	Preparation	1 week	18.09.2023	24.09.2023																			
25	Writing	19 weeks	18.09.2023	09.02.2024																			
26	Validation	1 week	05.02.2024	09.02.2024																			

B | Estimated cost of cloud infrastructure

Contact your AWS representative: Contact Sales

Export date: **07.02.2024 15:11:59** Language: **English**

Estimate URL: **https://calculator.aws/#/estimate?id=10cd9e0e4cd44fece32b23298f60ee2e7f7c399f**

Estimate summary

Upfront cost	Monthly cost	Total 12 months cost
0.00 USD	**40.45 USD**	**485.40 USD** Includes upfront cost

Detailed Estimate

Name	Group	Region	Upfront cost	Monthly cost
AWS IoT Core	No group applied	EU (Frankfurt)	0.00 USD	14.09 USD
Status: - **Description**: **Config summary**: Number of devices (MQTT) (100), Average size of each message (10 KB), Average size of each record (1 KB), Average size of each record (1 KB), Average number of actions executed per rule (1), Average size of each message (10 KB), Number of messages for a device (43800), Number of rules triggered (43800)				
AWS IoT Greengrass	No group applied	EU (Frankfurt)	0.00 USD	18.52 USD
Status: - **Description**: **Config summary**: Number of greengrass core devices (100), Activity Period in minutes per month (43800), MQTT topics with cloud as source (optional) (1000)				
AWS IoT Device Defender	No group applied	EU (Frankfurt)	0.00 USD	0.13 USD
Status: - **Description**: **Config summary**: Disconnect Duration (Not Included), Number of devices (100), Number of devices (Rules) (100), Number of metrics (Rules) (1), Number of times metric datapoint(s) are reported daily (Rules) (1)				

Amazon Simple Storage Service (S3)	No group applied	EU (Frankfurt)	0.00 USD	0.06 USD

Status: -
Description:
Config summary: S3 Standard storage (2.6 GB per month), PUT, COPY, POST, LIST requests to S3 Standard (100), GET, SELECT, and all other requests from S3 Standard (100), Data returned by S3 Select (0 GB per month), Data scanned by S3 Select (0 GB per month) DT Inbound: Internet (0 TB per month), DT Outbound: Not selected (0 TB per month)

Amazon Elastic Container Registry	No group applied	EU (Frankfurt)	0.00 USD	5.00 USD

Status: -
Description:
Config summary: Amount of data stored (50 GB per month)

AWS Lambda	No group applied	EU (Frankfurt)	0.00 USD	2.65 USD

Status: -
Description:
Config summary: Invoke Mode (Buffered), Architecture (Arm), Architecture (x86), Number of requests (4380000 per month), Amount of ephemeral storage allocated (512 MB)

Acknowledgement

AWS Pricing Calculator provides only an estimate of your AWS fees and doesn't include any taxes that might apply. Your actual fees depend on a variety of factors, including your actual usage of AWS services. Learn more

C | Open source project GitHub repository

GitHub repository name : *56kcloud/aws-iot-reference-architecture*

Link : https://github.com/56kcloud/aws-iot-reference-architecture

Bibliography

[1] *56K.Cloud*. Let's Start your Cloud Journey. URL: https://www.56k.cloud (visited on 09/21/2023).

[2] *Freepik*. Images by macrovector. URL: https://www.freepik.com/author/macrovector (visited on 09/29/2023).

[3] Peter Mell and Timothy Grance. "The NIST Definition of Cloud Computing". en. In: 2011. DOI: 10.6028/NIST.SP.800-145.

[4] *RedHat - IaaS vs. PaaS vs. SaaS*. URL: https://www.redhat.com/en/topics/cloud-computing/iaas-vs-paas-vs-saas (visited on 10/13/2023).

[5] *Docker*. URL: https://www.docker.com (visited on 10/12/2023).

[6] *Kubernetes*. URL: https://kubernetes.io (visited on 10/12/2023).

[7] *Cloud Native Computing Foundation*. MAKE CLOUD NATIVE UBIQUITOUS. URL: https://www.cncf.io (visited on 10/12/2023).

[8] *Cloud Native Computing Foundation*. CNCF Charter. URL: https://github.com/cncf/foundation/blob/main/charter.md (visited on 10/12/2023).

[9] Nane Kratzke and Peter-Christian Quint. "Understanding cloud-native applications after 10 years of cloud computing - A systematic mapping study". en. In: 2017. DOI: 10.1016/j.jss.2017.01.001.

[10] *Rancher - The History of Cloud Native*. URL: https://www.suse.com/c/rancher_blog/the-history-of-cloud-native/ (visited on 10/12/2023).

[11] Simson Garfinkel. "An Evaluation of Amazon's Grid Computing Services: EC2, S3 and SQS". en. In: 2007.

[12] *Benjamin Black — EC2 Origins*. URL: http://blog.b3k.us/2009/01/25/ec2-origins.html (visited on 10/31/2023).

[13] *IBM*. What is lift and shift? URL: https://www.ibm.com/topics/lift-and-shift (visited on 10/12/2023).

[14] Vasilios Andrikopoulos, Christoph Fehling, and Fran Leymann. "DESIGNING FOR CAP - The Effect of Design Decisions on the CAP Properties of Cloud-native Applications". In: *Proceedings of the 2nd International Conference on Cloud Computing and Services Science - CLOSER*. INSTICC. SciTePress, 2012, pp. 365–374. ISBN: 978-989-8565-05-1. DOI: 10.5220/0003931503650374.

[15] Sergio García-Gómez et al. "4CaaSt: Comprehensive Management of Cloud Services through a PaaS". In: 2012, pp. 494–499. DOI: 10.1109/ISPA.2012.72.

[16] Nane Kratzke. "A Brief History of Cloud Application Architectures". en. In: Applied Sciences, 2018, p. 1368. DOI: 10.3390/app8081368.

[17] Manuel Díaz, Cristian Martín, and Bartolomé Rubio. "State-of-the-art, challenges, and open issues in the integration of Internet of things and cloud computing". en. In: Journal of Network and Computer Applications, 2016, pp. 99–117. DOI: 10.1016/j.jnca.2016.01.010.

[18] Mohammad Aazam et al. "Cloud of Things: Integrating Internet of Things and cloud computing and the issues involved". en. In: 2014, pp. 414–419. DOI: 10.1109/IBCAST.2014.6778179.

[19] Nakita Oza. "Integration of Cloud with Embedded Systems for IoT devices". en. In: International Journal of Latest Transactions in Engineering and Science, 2018.

[20] Tetsuo Furuichi and Kunihiro Yamada. "Next Generation of Embedded System on Cloud Computing". en. In: Procedia Computer Science, 2014, pp. 1605–1614. DOI: 10.1016/j.procs.2014.08.244.

[21] Cloud Standards Customer Council. "Cloud Customer Architecture for IoT". en. In: (2016).

[22] *Zigbee*. The Full-Stack Solution for All Smart Devices. URL: https://csa-iot.org/all-solutions/zigbee/ (visited on 10/20/2023).

[23] *Bluetooth*. URL: https://www.bluetooth.com (visited on 10/29/2023).

[24] *LoRa Alliance*. URL: https://lora-alliance.org (visited on 10/29/2023).

[25] *Thread*. URL: https://www.threadgroup.org (visited on 10/29/2023).

[26] *Connectivity Standards Alliance - Matter*. URL: https://csa-iot.org/all-solutions/matter/ (visited on 10/29/2023).

[27] Christos Stergiou et al. "Secure integration of IoT and Cloud Computing". In: *Future Generation Computer Systems* (2018), pp. 964–975. ISSN: 0167-739X. DOI: 10.1016/j.future.2016.11.031.

[28] Moataz Soliman et al. "Smart Home: Integrating Internet of Things with Web Services and Cloud Computing". en. In: *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*. 2013, pp. 317–320. DOI: 10.1109/CloudCom.2013.155.

[29] *MQTT*. The Standard for IoT Messaging. URL: https://mqtt.org (visited on 10/29/2023).

[30] Zach Shelby, Klaus Hartke, and Carsten Bormann. *The Constrained Application Protocol (CoAP)*. RFC 7252. 2014. DOI: 10.17487/RFC7252. URL: https://datatracker.ietf.org/doc/rfc7252.

[31] Jiehan Zhou et al. "CloudThings: A common architecture for integrating the Internet of Things with Cloud Computing". en. In: CSCWD, 2013, pp. 651–657. DOI: 10.1109/CSCWD.2013.6581037.

[32] Matthias Kovatsch, Martin Lanter, and Simon Duquennoy. "Actinium: A RESTful runtime container for scriptable Internet of Things applications". In: *2012 3rd IEEE International Conference on the Internet of Things*. 2012, pp. 135–142. DOI: 10.1109/IOT.2012.6402315.

[33] Sergio Martin et al. *Arm Mbed – AWS IoT System Integration*. en. Research Reports or Papers. 2018. URL: https://eprints.gla.ac.uk/157277/.

[34] *AWS IoT Core - MQTT*. URL: https://docs.aws.amazon.com/iot/latest/developerguide/mqtt.html (visited on 10/29/2023).

[35] Euripides G. M. Petrakis et al. "Internet of Things as a Service (iTaaS): Challenges and solutions for management of sensor data on the cloud and the fog". In: *Internet of Things* (2018), pp. 156–174. ISSN: 2542-6605. DOI: 10.1016/j.iot.2018.09.009.

[36] Maria Júlia Berriel de Sousa et al. "Over-the-air firmware update for IoT devices on the wild". In: *Internet of Things* (2022). ISSN: 2542-6605. DOI: 10.1016/j.iot.2022.100578.

[37] *Amazon Elastic Load Balancer*. URL: https://aws.amazon.com/en/elasticloadbalancing/ (visited on 10/30/2023).

[38] Yeongho Choi and Yujin Lim. "Optimization Approach for Resource Allocation on Cloud Computing for IoT". en. In: *International Journal of Distributed Sensor Networks* (2016). ISSN: 1550-1329. DOI: 10.1155/2016/3479247.

[39] *Comment établir la coexistence entre IPv4 et IPv6*. URL: https://community.fs.com/fr/article/how-to-achieve-ipv4-and-ipv6-coexistence-dual-stack-or-mpls-tunnel.html (visited on 10/30/2023).

[40] *Update the AWS IoT Greengrass Core software (OTA)*. URL: https://docs.aws.amazon.com/greengrass/v2/developerguide/update-greengrass-core-v2.html (visited on 10/30/2023).

[41] *Check Greengrass core device status*. URL: https://docs.aws.amazon.com/greengrass/v2/developerguide/device-status.html (visited on 10/30/2023).

[42] *What is Device Update for IoT Hub?* URL: https://learn.microsoft.com/en-us/azure/iot-hub-device-update/understand-device-update (visited on 10/30/2023).

[43] *Monitor device connection status*. URL: https://learn.microsoft.com/en-us/azure/iot-hub/monitor-device-connection-state (visited on 10/30/2023).

[44] Brendan Moran et al. *A Firmware Update Architecture for Internet of Things*. 2021. DOI: 10.17487/RFC9019.

[45] Alessio Botta et al. "Integration of Cloud computing and Internet of Things: A survey". In: *Future Generation Computer Systems* (2016), pp. 684–700. ISSN: 0167-739X. DOI: 10.1016/j.future.2015.09.021.

[46] Mohammad Aazam and Eui-Nam Huh. "Fog Computing and Smart Gateway Based Communication for Cloud of Things". In: *2014 International Conference on Future Internet of Things and Cloud*. 2014, pp. 464–470. DOI: 10.1109/FiCloud.2014.83.

[47] Muhammad Ilyas, Muneeb Ahmad, and Sajid Saleem. "Internet-of-Things-Infrastructure-as-a-Service: The democratization of access to public Internet-of-Things Infrastructure". en. In: *International Journal of Communication Systems* (2020). DOI: 10.1002/dac.4562.

[48] *Raspberry Pi*. URL: https://www.raspberrypi.com (visited on 10/20/2023).

[49] Fung Po Tso et al. "The Glasgow Raspberry Pi Cloud: A Scale Model for Cloud Computing Infrastructures". In: *2013 IEEE 33rd International Conference on Distributed Computing Systems Workshops*. 2013, pp. 108–112. DOI: 10.1109/ICDCSW.2013.25.

[50] Pekka Abrahamsson et al. "Affordable and Energy-Efficient Cloud Computing Clusters: The Bolzano Raspberry Pi Cloud Cluster Experiment". en. In: IEEE, 2013, pp. 170–175. ISBN: 978-0-7695-5095-4. DOI: 10.1109/CloudCom.2013.121.

[51] Ranjana Sikarwar, Pradeep Yadav, and Aditya Dubey. "A Survey on IOT enabled cloud platforms". In: *2020 IEEE 9th International Conference on Communication Systems and Network Technologies (CSNT)*. 2020, pp. 120–124. DOI: 10.1109/CSNT48778.2020.9115735.

[52] *Gartner - Cloud Infrastructure and Platform Services Reviews and Ratings*. URL: https://www.gartner.com/reviews/market/cloud-infrastructure-and-platform-services (visited on 10/26/2023).

[53] *Amazon Web Services*. URL: https://aws.amazon.com/ (visited on 10/23/2023).

[54] Yohanes Yohanie Fridelin Panduman, Sritrusta Sukaridhoto, and Anang Tjahjono. "A Survey of IoT Platform Comparison for Building Cyber-Physical System Architecture". In: *2019 International Seminar on Research of Information Technology and Intelligent Systems (ISRITI)*. 2019, pp. 238–243. DOI: 10.1109/ISRITI48646.2019.9034650.

[55] Alexandra A. Kirsanova, Gleb I. Radchenko, and Andrei N. Tchernykh. "Fog computing state of the art: concept and classification of platforms to support distributed computing systems". In: (2021). DOI: 10.48550/arXiv.2106.11726.

[56] *Microsoft Azure*. URL: https://azure.microsoft.com/ (visited on 10/23/2023).

[57] *Google Cloud*. URL: https://cloud.google.com/ (visited on 10/23/2023).

[58] *Akenza AG - Google IoT Core is Shutting Down*. URL: https://blog.akenza.io/google-iot-core-shutdown (visited on 10/25/2023).

[59] Claus Pahl and Brian Lee. "Containers and Clusters for Edge Cloud Architectures – A Technology Review". In: *2015 3rd International Conference on Future Internet of Things and Cloud*. 2015, pp. 379–386. DOI: 10.1109/FiCloud.2015.35.

[60] Matti Valkeinen. "Cloud Infrastructure Tools For Cloud Applications : Infrastructure management of multiple cloud platforms". en. In: (2022). URL: https://trepo.tuni.fi/handle/10024/137424.

[61] *AWS CloudFormation*. URL: https://aws.amazon.com/en/cloudformation/ (visited on 10/26/2023).

[62] *AWS CDK*. URL: https://aws.amazon.com/en/cdk/ (visited on 10/26/2023).

[63] *Azure Resource Manager*. URL: https://azure.microsoft.com/en-gb/get-started/azure-portal/resource-manager (visited on 10/26/2023).

[64] *Terraform*. URL: https://www.terraform.io (visited on 10/26/2023).

[65] *Pulumi*. URL: https://www.pulumi.com (visited on 10/26/2023).

[66] *GitHub - hashicorp/terraform*. URL: https://github.com/hashicorp/terraform/tags (visited on 10/26/2023).

[67] *GitHub - pulumi/pulumi*. URL: https://github.com/pulumi/pulumi (visited on 10/26/2023).

[68] *SystemReady Certification Program*. URL: https://www.arm.com/architecture/system-architectures/systemready-certification-program (visited on 10/27/2023).

[69] *Google, Oracle's Ampere VMs get Arm's SystemReady seal of approval*. URL: https://www.theregister.com/2022/11/02/google_oracles_cloud_arm/ (visited on 10/27/2023).

[70] *SystemReady Pre-Silicon Reference Guide BSA integration and compliance*. BSA and SBSA overview. URL: https://developer.arm.com/documentation/102858/0100/BSA-and-SBSA-overview (visited on 10/27/2023).

[71] *GitHub - ARM-software/bbr-acs*. URL: https://github.com/ARM-software/bbr-acs (visited on 10/27/2023).

[72] *How to manage IoT device certificate rotation using AWS IoT*. URL: https://aws.amazon.com/fr/blogs/iot/how-to-manage-iot-device-certificate-rotation-using-aws-iot/ (visited on 02/03/2024).

[73] *What is AWS IoT Greengrass?* URL: https://docs.aws.amazon.com/greengrass/v2/developerguide/what-is-iot-greengrass.html (visited on 01/31/2024).

[74] *Install AWS IoT Greengrass Core software with AWS IoT fleet provisioning*. URL: https://docs.aws.amazon.com/greengrass/v2/developerguide/fleet-provisioning.html (visited on 02/01/2024).

[75] *Packer by HashiCorp*. URL: https://www.packer.io (visited on 02/01/2024).

[76] *GitHub - mkaczanowski/packer-builder-arm*. URL: https://github.com/mkaczanowski/packer-builder-arm (visited on 02/01/2024).

[77] *Security in AWS IoT Greengrass*. URL: https://docs.aws.amazon.com/greengrass/v2/developerguide/security.html (visited on 02/01/2024).

[78] *GitHub - 56kcloud/aws-iot-reference-architecture*. URL: https://github.com/56kcloud/aws-iot-reference-architecture (visited on 02/01/2024).

[79] *Greengrass components*. URL: https://docs.aws.amazon.com/en_en/greengrass/v2/developerguide/greengrass-components.html (visited on 02/03/2024).

[80] *Arm Virtual Hardware*. URL: https://www.arm.com/products/development-tools/simulation/virtual-hardware (visited on 02/08/2024).

[81] Noura Abbas, Andrew M. Gravell, and Gary B. Wills. "Historical Roots of Agile Methods: Where Did "Agile Thinking" Come From?" en. In: 2008. ISBN: 978-3-540-68255-4. DOI: 10.1007/978-3-540-68255-4_10.

[82] Elena Mugelini and Omar Abou Khaled. *MSE university course - IT Project Management*. PHASE III: execute the IT project - choosing a SDLC methodologies. 2022.

[83] *Manifesto for Agile Software Development*. URL: https://agilemanifesto.org/iso/en/manifesto.html (visited on 10/05/2023).

[84] *Scrum Guides*. Jeff Sutherland. URL: https://scrumguides.org/jeff.html (visited on 10/05/2023).

[85] Ken Schwaber and Jeff Sutherland. *The Scrum Guide*. 6th ed. 2020.

[86] Scrum.org, Daniel Vacanti, and Yuval Yeret. *The Kanban Guide for Scrum Teams*. 2021.

[87] *Wikipedia - Little's law*. URL: https://en.wikipedia.org/wiki/Little%27s_law (visited on 10/06/2023).

[88] *Atlassian - Kanban*. URL: https://www.atlassian.com/agile/kanban (visited on 10/06/2023).

[89] *AWS - AWS Certified Cloud Practitioner*. URL: https://aws.amazon.com/certification/certified-cloud-practitioner/ (visited on 01/29/2024).

Glossary

Arm Arm is a British company that designs processors with RISC-type (Reduced Instruction Set Computer) architectures. viii, ix, 3, 5, 8, 13, 15, 21, 22, 24, 25, 37, 52, 55, 72, 73, 79, 87, 88

AWS AWS stands for Amazon Web Services. It is a company that provides a cloud platform with over 200 services. The resources are spread over more than 32 regions around the world. x, xi, 1, 3, 4, 9, 10, 13, 14, 15, 17, 18, 20, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 43, 44, 45, 46, 48, 49, 50, 51, 56, 57, 58, 60, 61, 62, 64, 65, 66, 67, 68, 69, 70, 71, 73, 74, 75, 76, 77, 85, 86, 87, 88, 89

cloud The term cloud refers to a remote infrastructure made up of servers that can be accessed from anywhere in the world via an internet connection. v, viii, x, xi, 1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 22, 24, 25, 26, 27, 32, 33, 34, 38, 39, 40, 41, 44, 50, 71, 77, 79, 85, 86, 87, 88, 89

cloud infrastructure A cloud infrastructure is the set of computing resources in the cloud that make up an IT environment. This includes servers, storage, network and security. viii, ix, x, xi, 1, 2, 3, 4, 6, 7, 10, 11, 15, 17, 18, 19, 23, 24, 25, 31, 32, 40, 44, 55, 58, 59, 70, 71, 74, 75, 78, 87, 88, 89, 95

HES-SO The HES-SO is a Swiss university based in French-speaking Switzerland. It offers a wide range of specialisations in different fields such as technology, health, social work, economics and art. It has several buildings spread across the French-speaking cantons. 80

KanBan KanBan is a project management tool designed to efficiently manage a workflow over a given period. It is widely adopted in agile methodology. xi, 77, 79, 82, 83, 88

provisioning Provisioning means the automated creation and installation of an IT infrastructure. Device provisioning involves creating the resources needed to integrate it into an IT infrastructure and integrating it into the existing infrastructure. viii, x, xi, 10, 17, 25, 26, 31, 36, 37, 38, 41, 42, 55, 59, 60, 61, 62

Scrum Scrum is a process framework based on agile methodology for software engineering. It is suitable for small development teams. It ensures good communication between the customer and the team. It enables the first versions of a software product to be delivered quickly.. xi, 77, 78, 80, 81, 82, 83

work package In project management, work packages are commonly used to define a plan. The plan provides an overview of the progress of the project. 77, 78, 85

Acronyms

API **A**pplication **P**rogramming **I**nterface. 16, 18, 20

ARM **A**zure **R**esource **M**anager. 18, 20

AVH **A**rm **V**irtual **H**ardware. xi, 72, 73

BBR **B**ase **B**oot **R**quirements. 21

BLE **B**luetooth **L**ow **E**nergy. 13

BSA **B**ase **S**ystem **A**rchitecture. 21

BSD **B**erkeley **S**oftware **D**istribution. 21, 22

CA **C**ertificate **A**uthority. 38

CD **C**ontinuous **D**elivery. viii, x, 3, 4, 23, 25, 31, 39, 40, 41, 43, 55, 58, 60, 77, 78, 84, 85, 87

CDK **C**loud **D**evelopment **K**it. x, 17, 18, 20

CI **C**ontinuous **I**ntergration. viii, x, 3, 4, 23, 25, 31, 39, 40, 41, 43, 55, 58, 60, 77, 78, 84, 85, 87

CLI **C**ommand **L**ine **i**nterface. 19, 32, 36, 44

CoAP **Co**nstrained **A**pplication **P**rotocol. 13

CSR **C**ertificate **S**igning **R**equest. 28, 48, 49, 50, 67, 68

DevOps **D**evelopment and **O**perations. xi, 2, 39, 84, 87

ECR **E**lastic **C**ontainer **R**egistry. 24, 43, 47, 59, 76

HTTP **H**yper**T**ext **T**ransfer **P**rotocol. 13

IaaS **I**nfrastructure **a**s **a** **S**ervice. 6, 10, 18

IaC **I**nfrastructure **a**s **C**ode. 5, 8, 17, 18, 19, 20, 25, 32, 33, 79, 87

IoT **I**nternet **o**f **T**hings. viii, x, xi, 1, 2, 5, 8, 11, 12, 13, 14, 15, 16, 17, 22, 24, 25, 26, 27, 28, 29, 33, 34, 35, 36, 37, 38, 39, 41, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 59, 60, 61, 62, 65, 66, 70, 71, 72, 73, 75, 76, 79, 87, 88, 89

IPC **I**nter**P**rocess **C**ommunication. 34

JSON **J**ava**S**cript **O**bject **N**otation. 13, 18, 37

MQTT **M**essage **Q**ueuing **T**elemetry **T**ransport. xi, 13, 24, 29, 30, 34, 38, 39, 47, 48, 50, 51, 52, 62, 63, 65, 66, 67, 68, 71, 75, 87

NAS **N**etwork **A**ttached **S**torage. 15

NAT **N**etwork **A**ddress **T**ranslation. 14

OIDC **O**pen**ID** **C**onnect. x, 40, 56, 60

OS **O**perating **S**ystem. x, xi, 21, 22, 23, 24, 25, 26, 30, 31, 33, 36, 37, 42, 44, 52, 53, 55, 58, 60, 61, 70, 71, 72, 73, 76, 87, 88

OTA **O**ver-**T**he-**A**ir. 14, 34, 35

PaaS **P**latform **a**s **a** **S**ervice. 6, 10, 16, 18

PO **P**roduct **O**wner. 81, 82

QoS **Q**uality **o**f **S**ervice. 14

REST **RE**presentational **S**tate **T**ransfer. 13

ROI **R**eturn **O**n **OI**nvestment. 81

SaaS **S**oftware **a**s **a** **S**ervice. 6, 18

SDK **S**oftware **D**evelopment **K**it. 20

SOA **S**ervice-**O**riented **A**rchitecture. 13

TCP **T**ransmission **C**ontrol **P**rotocol. 13

TLS **T**ransport **L**ayer **S**ecurity. 39

UDP **U**ser **D**atagram **P**rotocol. 13

WiFi **Wi**reless **Fi**delity. 13

WIP **W**ork **I**n **P**rogress. 82, 83