POLITECNICO DI TORINO

DEPARTMENT OF CONTROL AND COMPUTER ENGINEERING (DAUIN)

Master's Degree in Mechatronic Engineering

Master's Degree Thesis

AWS Services for Cloud Robotics Applications

Supervisor

Prof. Alessandro RIZZO

Co-supervisors

Dr. Stefano PRIMATESTA

Ing. Roberto ANTONINI

Candidate

Matteo SARTONI

Academic Year 2021-2022

Summary

Nowadays robots are equipped with a great number of sensors, in order to increase their level of autonomy. Since analyzing the data coming from such devices is very resource demanding, cloud robotics was born to offload some of the heavy tasks, such as computing, memory and storage, to the cloud. Furthermore the availability of computational resources and the effort to manage these infrastructures is minimal. However, not all the software can be unloaded to the cloud due to latency requirements and to maintain a local autonomy. In fact, the robot may be affected by disconnections during an operation and, therefore, a minimum level of autonomy must be guaranteed on-board to maintain an adequate level of safety, as well as avoiding it being stuck in place.

In this thesis, an in-dept study of cloud technologies, such as Docker and Kubernetes, has been done. Subsequently some of the AWS services have been analyzed and used in order to design a cloud architecture for robotic fleet management. This architecture is made by three different parts: the single robot development, the multi-robotic algorithm and a REST API to visualize the robots' status.
Concerning the single robotic algorithm the attention is focused on AWS RoboMaker, a service that facilitates the creation of a robotic application through ROS and its simulation via the Gazebo simulator. The multi-robotic algorithm, instead, will be developed with AWS Cloud9, an IDE that supports different programming languages. Its containerized image will be uploaded on Amazon ECR, a container registry, and deployed in a Kubernetes' Pod, provided by Amazon EKS. The interaction between the Pod and the simulated robots will be possible via a NoSQL database offered by Amazon DynamoDB.
Lastly, the different robot status are shown using a REST API, developed through the integration of a Python script, that retrieves the status from the database, AWS Lambda and Amazon API Gateway.

In conclusion a different architecture, in case of availability of physical robots, will be exposed. Furthermore eProsima Fast DDS will be analyzed as a connection

between ROS2 nodes and other nodes stored in a Kubernetes' Pod.

Acknowledgements

I would like to thank my family for the constant support and enthusiasm while helping me following this journey, my mother Valentina and my father Alessandro for their patience, for always being present when I needed help, for their confidence on my capabilities and choices, and my brother Sandro for his advice.
Special thanks also to my friends, a second family to whom I have shared unforgettable moments, dreams and hopes.
Lastly I want to thank prof. Alessandro Rizzo, Dr. Stefano Primatesta and Dr. Roberto Antonini for having given me the opportunity to develop this thesis work and for their support throughout these months, and Dr. Fulvio Cambiotti for the technical assistance.

Table of Contents

List of Tables

List of Figures

Acronyms

AMI

Amazon Machine Image

API

Application Programming Interface

AWS

Amazon Web Services

CI/CD

Continuous Integration / Continuous Delivery

CLI

Command Line Interface

CPU

Central Processing Unit

CRI

Container Runtime Interface

DNS

Domain Name System

DDS

Digital Data Service

ECR

Elastic Container Registry

ECS

Elastic Container Service

EC2

Elastic Compute Cloud

EFS

Elastic File System

EKS

Elastic Kubernetes Service

FaaS

Function as a Service

GB

Gigabyte

GPU

Graphics Processing Unit

GUI

Graphical User Interface

HTTP

HyperText Transfer Protocol

IaaS

Infrastructure as a Service

IAM

Identity and Access Management

IDE

Integrated Development Environment

IP

Internet Protocol address

LAN

Local Area Network

OCI

Open Container Initiative

OMG

Object Management Group

OS

Operating System

PaaS

Platform as a Service

RAM

Random Access Memory

RBAC

Role-based Access Control

REST

Representational State Transfer

RTPS

Real-time Publish-Subscribe Protocol

SaaS

Software as a Service

SAM

Serverless Application Model

SDK

Software development kit

SQL

Structured Query Language

SSD

Solid-State Drive

SSH

Secure Shell

S3

Simple Storage Service

TCP

Transmission Control Protocol

UDP

User Datagram Protocol

VPC

Virtual Private Cloud

VM

Virtual Machine

WAN

Wide Area Network

Chapter 1

Introduction

1.1 Cloud Computing

In the last years cloud storage has grown its popularity among individuals who need larger storage space, and for entities seeking an efficient off-site data back-up solution, instead of keeping files on a hard drive or in a local storage device.

Cloud computing is the on-demand delivery of host services, like databases, storage and servers over the Internet, with pay-as-you-go pricing. It takes all the heavy tasks involved in processing the data away from the user's device. It lowers the operating costs, runs the infrastructure more efficiently and scales when changes are needed.

The hosted services can be either public and private. Public ones are available online for a fee, through an account that can be accessed by anyone, and offered by cloud providers that handle and control all the hardware, software and the general infrastructure. Private services instead are hosted on a private network to specific clients. Furthermore a hybrid solution is also available, that combines public and private clouds allowing data and applications to be shared between them. This type of approach gives greater flexibility, more deployment options and it helps to optimize the existing infrastructure, security and compliance.

Cloud computing is primarily composed of three services, as depicted in Figure 1.1:

- **Infrastructure as a Service (IaaS):** it involves a method for delivering everything from Operating System (OS) to servers and storage through Internet Protocol address (IP) based connectivity. Costumers can avoid to purchase software or servers and instead procure them in an outsourced, on-demand service.

- **Software as a Service (SaaS):** it provides the license of a software application to customers. Licenses are typically provided through a pay-as-you-go model or on-demand.

- **Platform as a Service (PaaS):** considered as the most complex of the three layers of cloud-based computing, it shares some similarities with SaaS. The main difference is that, instead of delivering software online, it is a platform for creating software that is delivered via Internet.

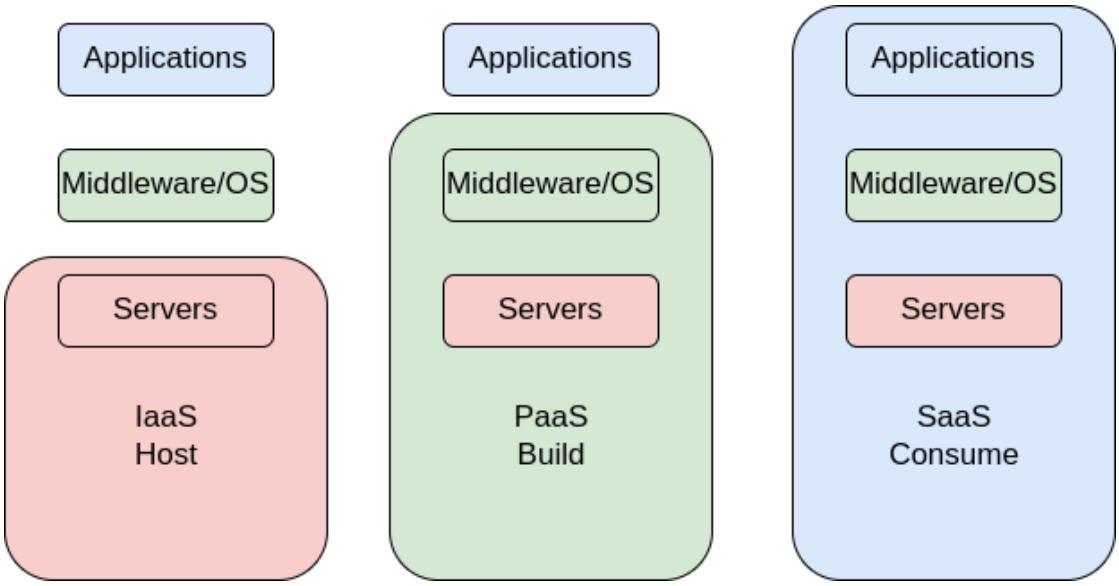

Figure 1.1: IaaS, PaaS and SaaS structure

Cloud-based solutions offer multiple benefits including the ability to use software from any device, either via a native application or a browser. As a consequence users can seamlessly carry their files and settings over to other devices. It also makes possible to back up their files, ensuring that they are immediately available in case of a hard drive crash.
Cloud technology has gained lots of popularity also because it offers businesses huge cost-savings potential. Before it became a feasible alternative, companies were required to purchase, construct and maintain costly management technology and infrastructure.
Lastly, cloud computing lets users upgrade their software more quickly, because software companies can offer their products via web rather than through more traditional methods, involving disks or flash drives.

1.2 Cloud Robotics

The term **"Cloud Robotics"** was coined by J. Kuffner in 2010 [1]. It is a field of robotics that aims to apply cloud computing resources to improve robotic system's collective learning, computational speed, collective memory and interconnection. This can be done since lots of resources, such as computing, storage and memory are provided by external data centers in the cloud, which can process and share information from multiple robots or agents.

Modern robots are equipped with a great number of sensors, to increase their level of autonomy. Since analyzing all the data coming from such devices can be very resource demanding, the cloud can be used to offload some of these heavy tasks, with advantages such as the wide availability of storage and computational resources, and a minimal management effort for provision. Other Internet-related features of robotics, like online sharing of open-source hardware and software, crowd sourcing of robotics funding, telepresence and human based computation may also fall under the definition of Cloud Robotics.

A cloud robotics platform includes secure servers that host vast databases. The stored data controls every aspect of the robotics machinery. Cloud robotics architectures typically comprise the following six components:

- **Global library of images, maps and object data**: it generally includes geometry and mechanical properties, expert systems and knowledge base.
- **Parallel computation on-demand:** to allow sample-based statistical modeling and motion planning, task planning, multi-robot collaboration, scheduling and coordination.
- **Shared information:** such as outcomes, trajectories and control policies as well as robot learning support.
- **Open-source code, data and designs:** for easy programming, experimentation and hardware construction.
- **On-demand human guidance and assistance:** for evaluation, learning and error recovery.
- **Augmented human-robot interaction**.

Nevertheless not all the tasks can be offloaded to the cloud because of latency constraints, but also to ensure local autonomy. In fact mobile robots may, for a given time interval, be away from the router that allows the communication with the cloud itself. It follows that distributed solutions are necessary, where the computation and storage are spread among the robot and the cloud.

1.3 Related Work

The integration of robotic systems together with the powerful resources offered by cloud computing is a topic that is becoming always more important. Off-loading different tasks to external data centers is not only useful for data management and analysis, but also from an economic perspective since it allows to mount less complex hardware systems on board, hence the overall unit cost for a given device decreases.

Several papers published during the last years focus on the development of a cloud robotic infrastructure to manage and control a fleet of autonomous robots. The work [2], for instance, provides a generic type of cloud robotic architecture usable by builders as reference, and a detailed panorama of all the required components. The authors retrieved a list of utilized components to devise a generic architecture, such as robots, I/O devices, service oriented architecture, data processing and control. This work has been done because authors realized that there was no agreed consensus on the tools to be used in the build-up of this type of systems.

In recent years new types of distributed solutions are also being developed, named Fog and Dew robotics.
Fog robotics is a system that efficiently distributes both computation and memory between edge, gateway and cloud devices, in order to access privacy and security. This approach can escalate the dexterity of robots by shoving the data closer to the device. Furthermore, it can ensure a more responsive human-robot interaction. Dew robotics, instead, is a paradigm that efficiently distributes computation and memory between edge, gateway and cloud devices to address privacy and security. This technique can improve scalability, since the tasks are distributed among a large number of heterogeneous devices. In this way it is possible to build distributed applications without the need of central nodes. For what concerns this second approach, an implementation based on ROS is described in [3], where a cloud platform has been developed to coordinate a fleet of drones and a rover for a search and rescue mission. The three ROS nodes, depicted in Figure 1.2, can communicate one with another through wireless network technology, using the rover as an access point. The *Reader* node is deployed on the drone, that captures an input video from a generic device, and sends the frames to the second tier of the platform. The other nodes, named *Encoder* and *Annotator*, are deployed on the ground rover so that they can use its resources to perform medium-weight tasks.

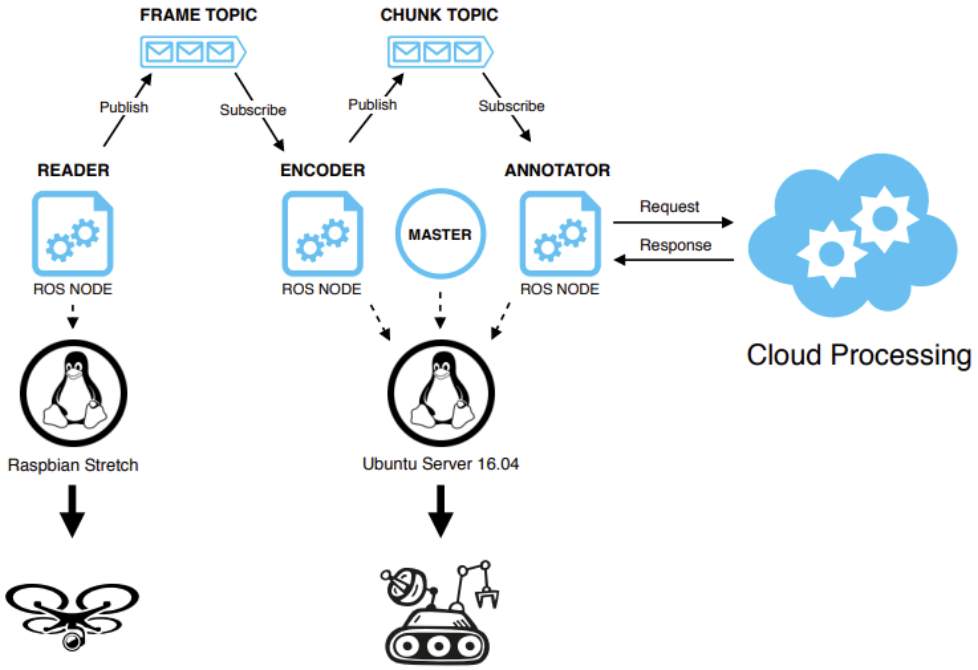

Figure 1.2: Dew Robotics platform based on ROS developed in [3]

1.4 Thesis structure

The main goal of this thesis work is to design a cloud robotic architecture for fleet management, using services provided by Amazon Web Services (AWS). The robots, simulated through the Gazebo simulator, are managed by a multi robot algorithm deployed in a Kubernetes' Pod. A non-relational database is used as the bridge linking the Pod and the different robots. It is also used to store the devices' status, that will be viewable in a web page through a Representational State Transfer Application Programming Interface (REST API).

In the next chapters the adopted technologies and the proposed cloud architecture will be analyzed. The second chapter gives an overview on Docker, Kubernetes and non-relational databases. Chapter 3 focuses on AWS and its services that have been used in this work, while the successive one describes the developed architecture and its implementation. Lastly further developments are presented, such as the design of a cloud infrastructure in case of availability of physical robots, and the connection, through eProsima Fast DDS, between ROS2 nodes deployed on a robot with others contained in a Kubernetes' Pod.

Chapter 2

Adopted Technologies

2.1 Docker

Founded as DotCloud in 2008 by Solomon Hykes, what now is known as Docker started out as a PaaS, before pivoting in 2013 to focus on democratizing the underlying software container its platform was running on. Hykes first presented a demo of Docker at PyCon, the biggest annual conference on the Python programming language. Here he explained that Docker was created because developers kept asking for the underlying technology powering DotCloud.

Docker is an open source software platform that uses OS-level virtualization to deliver software in containers. Containers are small and lightweight execution environments that make shared use of the OS kernel, but run in isolation from one another. The isolation and security given by containers allow the user to run many of them on a given host.
While containers have been used in Linux and Unix systems for some time, Docker spread this technology by making it easier for developers to package their software. Docker packages, provisions and runs containers, whose technology is available through the OS. A container packages the application service with all of its libraries, configuration files, dependencies and the other parts needed to operate.

Differently from Virtual Machines (VMs), which encapsulate an entire OS with executable code, Docker uses resources isolation on the operating system kernel to run multiple containers on the same OS.

2.1.1 VM vs Container

Containers have become the compute units of modern cloud-native applications, since they are more portable and resource-efficient than VMs.

VMs, whose architecture is depicted in Figure 2.1, are an abstraction of physical hardware turning one server into multiple ones. The hypervisor allows multiple VMs to run on a single machine, where each machine includes a full copy of an OS, libraries, the application and necessary binaries, taking up a lot of Gigabytes (GBs). Because of this they can be slow to boot.

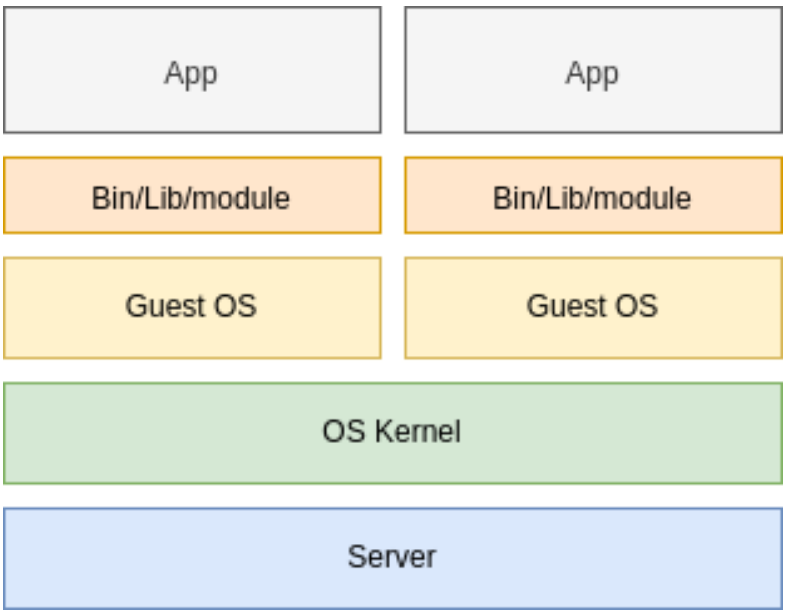

Figure 2.1: VM architecture

Containers and VMs have similar isolation and allocation benefits, but function differently because containers virtualize the OS instead of the hardware.

A container, whose structure is reported in Figure 2.2, is instead an OS-level virtualization in which the software or application can be moved and run consistently, in any environment and infrastructure, without launching a VM for each application. It is independent of the environment itself and the infrastructure's OS. Containers are an alternative way to coding on one platform or OS, which made moving the applications difficult since the code might not be compatible with the new environment.

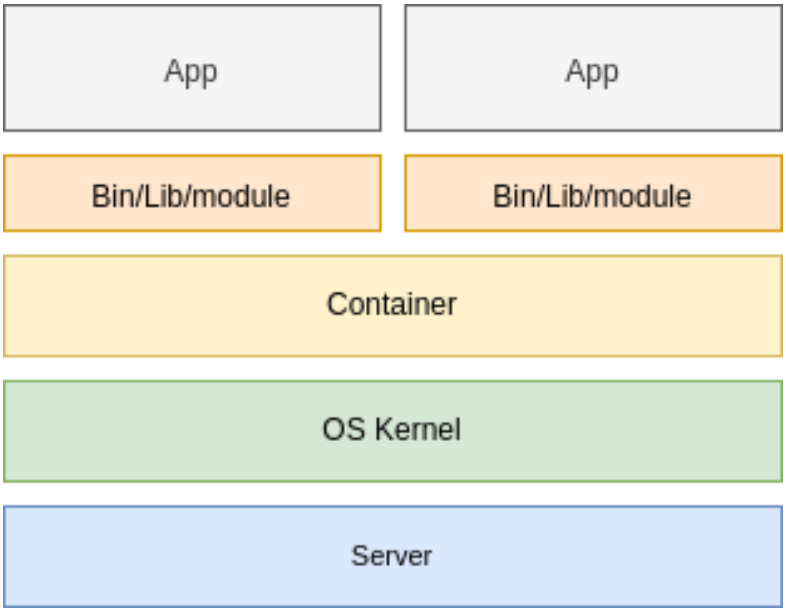

Figure 2.2: Container structure

Each container consists of one or more running processes which are isolated from the rest of the system. However when a container is not running, it exists only as a saved file, the **container image**. It is a lightweight executable, a package of the application source code, binaries, files and other dependencies.

A container image can run on various types of infrastructure like bare metal, within VMs and in the cloud. When a containerized application starts, the contents of a container image are copied before they are spun up into a container instance. Each image can be used to instantiate any number of container and, for this reason, it can be considered as a container blueprint.

Container images can only be shared with others via a public or private container registry. To promote sharing and compatibility, images are typically create in the industry-standard Open Container Initiative (OCI) format.

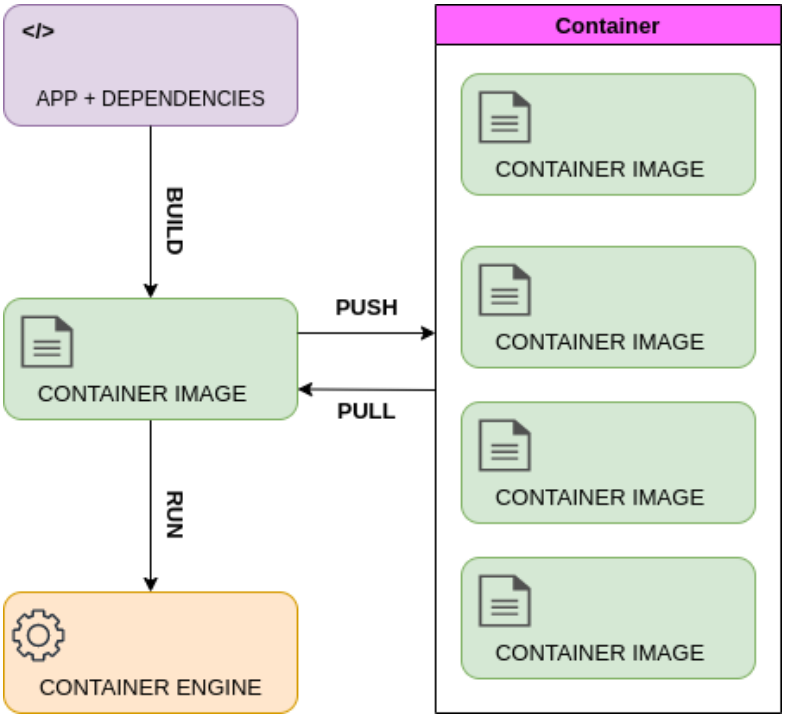

Figure 2.3: Container workflow

The **container engine** is the software components that enables the host OS to act as a container host. It accepts user commands to build, start and manage container through client tools, and also provides an API that allows external program to make similar requests. The most important aspects of a container engine's functionality are performed by its core component, the **container runtime**. It is responsible for creating the standardized platform on which apps can run, for running containers, and for handling container's storage needs on the local system.

A container is executed by a container runtime engine. Among the ones available in the market, Docker is the most adopted one.

2.1.2 Docker architecture

Docker, whose architecture is reported in Figure 2.5, uses a client-server architecture. Broken down, it comprises six different component parts.

Dockerfile

The **Dockerfile** is a text file that provides a series of instructions to build a Docker image, such as the OS, languages, environmental variables, file locations, network

ports and any other required component.

```
# syntax=docker/dockerfile:1
FROM node:12-alpine
RUN apk add --no-cache python2 g++ make
WORKDIR /app
COPY . .
RUN yarn install --production
CMD ["node", "src/index.js"]
EXPOSE 3000
```

Figure 2.4: Dockerfile example, source [4]

It is made by different instructions, the most important are:

- **FROM:** is the first instruction in a Dockerfile, it defines the image we want to use.
- **RUN:** is an image build step, the state of the container after this command will be committed to the container image. A Dockerfile may have different RUN steps that layer on top of one another to build the image.
- **COPY:** allows to copy local files on the specified image.
- **CMD:** is the command the container executes by default when the user launches the built image. A Dockerfile will only use the final **CMD** defined.
- **EXPOSE:** exposes a particular port with a specified protocol inside a Docker container. The ports can be either Transmission Control Protocol (TCP) or User Datagram Protocol (UDP), but the former is selected by default.
- **VOLUME:** specifies a mount point in the container. It will be mapped to a location on the host that is either specified when the container is created or, if not, chosen automatically from a directory created in */var/lib/docker/volumes*.
- **ENTRYPOINT:** is used to set executables that will always run when the container is initiated. Differently from CMD, this command cannot be ignored or overridden.
- **ENV:** defines the environment variables of the user's container.

Docker image

A **Docker image** is a portable, read-only executable file containing the instructions for creating a container and the specifications for which software component the container will run and how. The user can create the images through a Dockerfile or use only those created by others and published in a registry. When the developer changes the Dockerfile and rebuilds the image, only those layers which have changed are rebuilt.

Docker Hub

Also known as Docker registry, it is a public repository where container images can be stored, shared and managed. Docker is configured to look for images on **Docker Hub** by default.

Docker Engine

It is the core of Docker. **Docker Engine** is the underlying client-server technology that creates and runs the container. It includes a long-running daemon process, called **dockerd**, for managing container, APIs that allow program to communicate with the daemon and a Command Line Interface (CLI). The Docker client and Docker daemon can run on the same system, or the user can connect a client to a remote daemon. They communicate using a REST API, over UNIX sockets or a network interface.

Docker Compose

This component is a command-line tool that uses YAML files to define and run multi-container Docker applications. It allows to create, start, stop and rebuild all the services from the user's configuration and view the status and log output of all running services.

Docker Desktop

It is an application for Desktop to work with the Docker Engine. It provides a user-friendly way to build and share containerized applications and microservices.

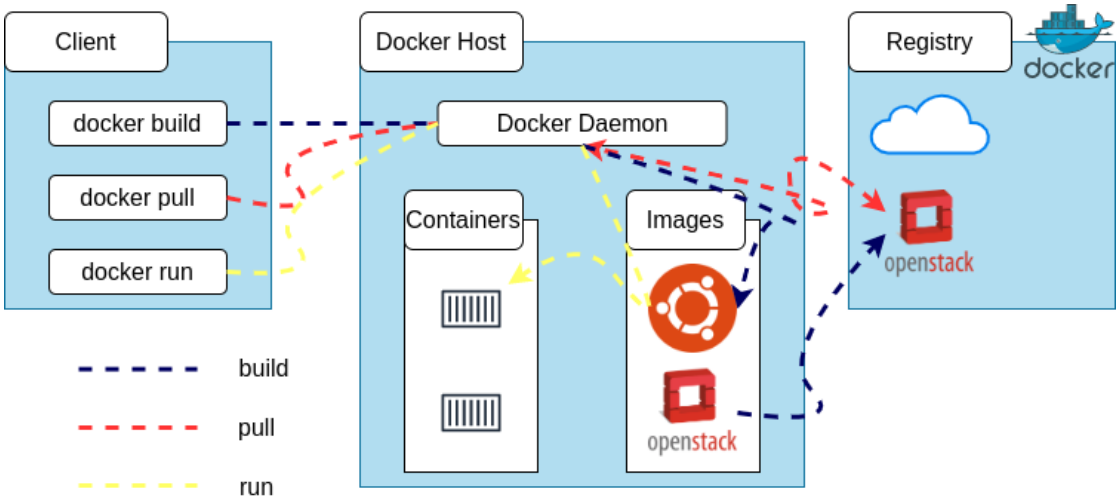

Figure 2.5: Docker architecture

2.2 Kubernetes

Google was one of the early contributors to Linux container technology and in 2004 launched Borg, an internal cluster management system. With this platform the company generated more than 2 billion container deployments a week. Given the relevance of this technology Google decided to present, 10 years after, Kubernetes as an open-source version of Borg.

Containers are a good way to bundle and run applications. However, in a production environment, there is the need to manage the containers that run the applications and ensure there is no downtime.
This is where Kubernetes comes in handy. As depicted in Figure 2.6, it is the most used open-source container orchestration platform that runs distributed systems resiliently, automating many of the processes involved in deploying, managing and scaling containerized applications.

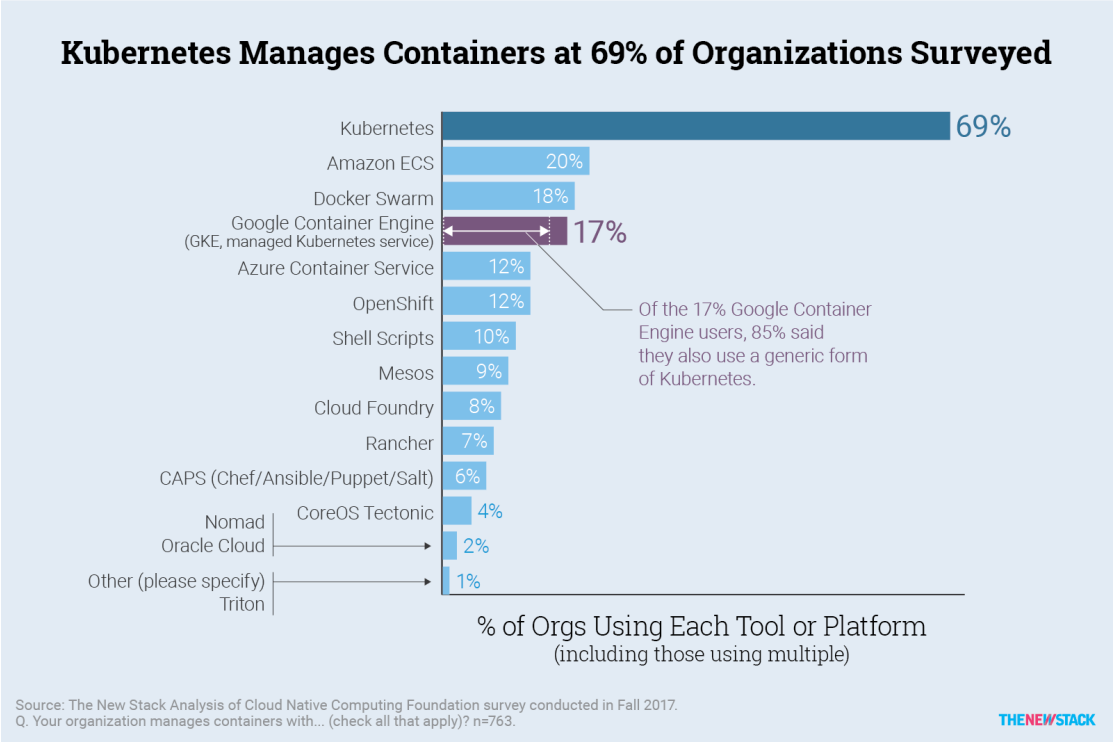

Figure 2.6: Market share of container management systems, source [5]

2.2.1 Kubernetes features

Kubernetes provides the user with:

- **Service discovery and load balancing:** it exposes a container via the IP address or the Domain Name System (DNS) name. In case in which the traffic to a container is high, Kubernetes load balances and distributes the network traffic to ensure deployment stability.
- **Automated rollouts and rollbacks**: the user can describe the desired state of the deployed containers via Kubernetes, and change them to a desired one at a controlled rate.
- **Automatic bin packing:** when working with Kubernetes the user creates a cluster of nodes that are used to run containerized applications, underlying how much Central Processing Unit (CPU) and Random Access Memory (RAM) each container needs. Kubernetes will fit containers onto the nodes to make the best use of the resources.
- **Storage orchestration:** Kubernetes allows the user to automatically mount

the storage system that prefers, such as local storage, public cloud providers and more.

- **Secret and configuration management:** used to store and manage sensitive information like passwords, OAuth tokens and Secure Shell (SSH) keys. The user can deploy and update secrets and application configuration without rebuilding his/hers container images and exposing secrets in the stack configuration.

- **Self-healing:** Kubernetes restarts containers that fail, replaces them, kills the ones that do not respond to user-defined health check and doesn't advertise them to clients until they're ready.

2.2.2 Kubernetes components

The core of Kubernetes is its cluster, represented in Figure 2,7. It contains a set of worker machines, called **nodes**, that run containerized applications, and the **control plane**, that manages the nodes. Every Kubernetes cluster contains at least one node.

The worker nodes host the **Pods**, the smallest execution unit in Kubernetes, which encapsulate one or more applications. The control plane usually runs across multiple computers. A Kubernetes cluster generally contains multiple nodes in order to provide availability and fault-tolerance.

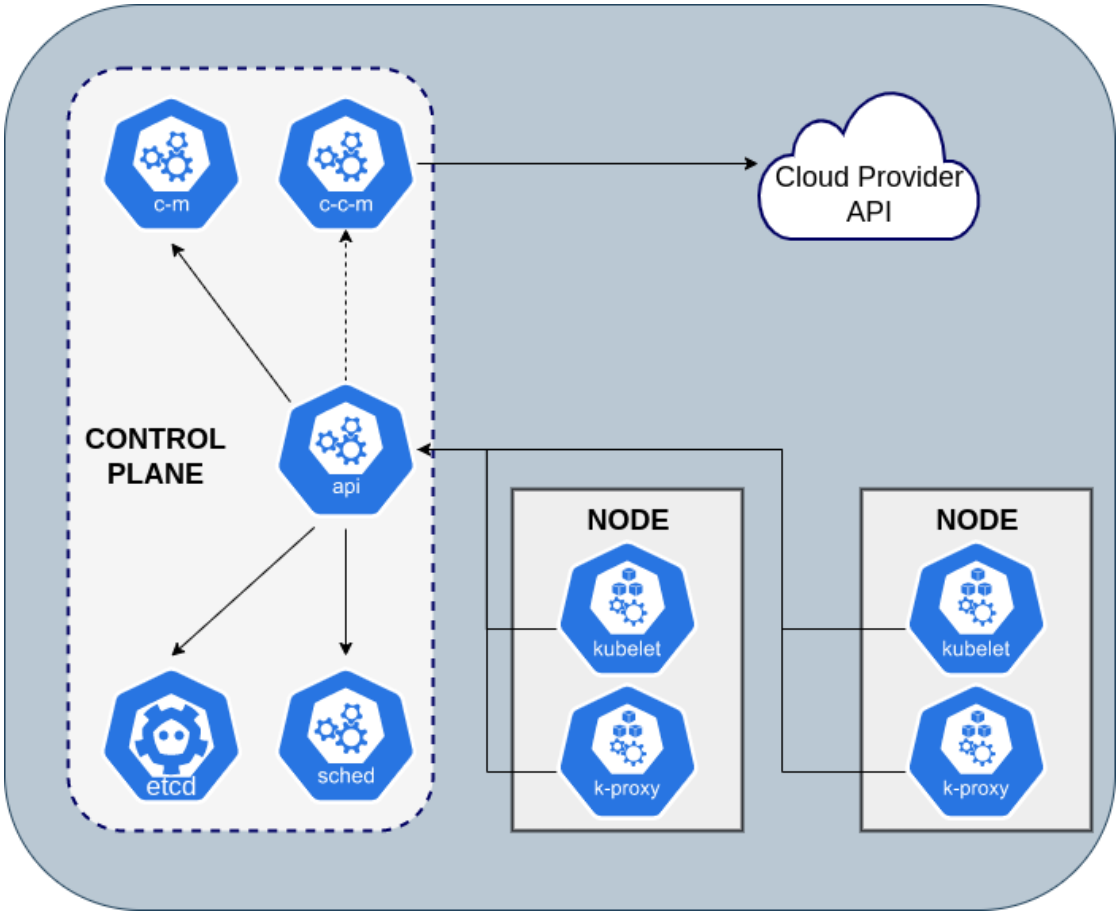

Figure 2.7: Kubernetes cluster

2.2.3 Control plane components

The control plane makes global decision about the cluster, detects and responds to cluster events. Its components can be run on any machine in the cluster. For simplicity, set up scripts typically start all these components on the same machine, in which the user's containers do not run.

kube-apiserver (api)

The **kube-apiserver** is the front end of the control plane itself. It is a component that exposes the Kubernetes API. It is designed to scale horizontally, deploying more instances. In this way the user can run different instances of kube-apiserver and balance traffic between them.

etcd

It is a highly-available key value store, used by Kubernetes as a backing store for all the cluster data. If the Kubernetes cluster uses **etcd**, it is required to have a back up plan for those data.

kube-scheduler (sched)

The **kube-scheduler** is a control plane component that looks for newly generated Pods with no assigned node, and selects a node for them to run on.
Factors taken into consideration for scheduling are: individual and collective resource requirements, affinity and anti-affinity specifications, hardware/software/policy constraints, data locality, inter-workload interference and deadlines.

kube-controller-manager (c-m)

The **kube-controller-manager** is the control plane component that runs controller processes.
Logically each controller is a separate process but, to decrease complexity, they are all compiled into a single binary and run in a single process.

The basic types of controllers are:

- **Node controller:** notices and responds when nodes go down.
- **Job controller:** looks up for job object that represent one off-tasks, then creates Pods to run those tasks to completion.
- **Endpoints controller:** populates the Endpoints object.
- **Service Account & Token controllers**: create default accounts and API access tokens for new namespaces.

cloud-controller-manager (c-c-m)

The **cloud-controller-manager** is a component that embeds cloud-specific control logic. It allows the user to link the cluster into the cloud provider's API, and separates out the components that interact with that cloud platform from the ones that only interact with the cluster. It only runs controllers that are specific to the cloud provider.
Similarly to the kube-controller-manager, the cloud-controller-manager combines several independent control loops into a single binary that the user runs in a single process. It is possible to scale horizontally in order to improve performance and to help tolerate failures.

The controllers that can have cloud provider dependencies are:

- **Node controller:** to check the cloud provider to determine if a node has been deleted in the cloud after it stops responding.
- **Route controller:** to set up routes in the underlying cloud infrastructure.
- **Service controller:** to create, update and delete cloud provider load balancers.

2.2.4 Node components

The node components run on every node, maintaining running pods and bringing the Kubernetes runtime environment.

kubelet

It is an agent that makes sure that containers are running in a Pod.
The **kubelet** takes a set of **PodSpecs**, files that describe a version of a Pod library, and ensures the containers described by them are running and healthy. It receives command from the API server and instructs the container runtime to start or stop containers as needed.
It does not manage containers that were not created by Kubernetes.

kube-proxy (k-proxy)

The **kube-proxy** is a network proxy that runs on each Kubernetes node and it is responsible for maintaining network rules on each worker machine. These rules allow communication to the user's Pods from network sessions inside or outside from the user's cluster. Kube-proxy can forward traffic itself or use the OS packet filter layer.

Container runtime

It is the software layer responsible for running the containers.
There are different **container runtimes** supported by Kubernetes such as containered, Container Runtime Interface plus OCI (CRI-O), Docker and other Kubernetes CRI implementations.

2.2.5 Addons

Addons use Kubernetes resources, to implement cluster features. Since these are providing cluster-level features, namespaced resources for addons belong within the kube-system namespace.

DNS

Among all the available addons, the only one that is required is the cluster **DNS**. It is a DNS server, in addition to others in the user's environment, which serves DNS records for Kubernetes services.
Containers started by Kubernetes automatically include this DNS server in their DNS searches.

2.3 NoSQL Database

Carl Strozz introduced the "Not only Structured Query Language" (NoSQL) Database concept back in 1998. The term **"NoSQL"** refers to non-relational databases that store data in a different format with respect to relational ones. They can be queried using idiomatic language APIs, declarative structured query languages and query-by example languages.

These databases are widely used in real-time applications and big data, since they are scalable, flexible, highly available, and capable of responding to modern data management demands. They are also the favourite choice of developers, as they lend themselves to an agile development paradigm by adapting to changing requirements. NoSQL databases allow the data to be stored in more intuitive ways with fewer transformations required. They can also use cloud to deliver zero downtime.

The four most popular types of NoSQL database are:

- **Key-value stores:** group associated data in collections with records, identified through unique keys for easy retrieval. The application has complete control over what is stored in the value, making it the most flexible NoSQL model. Data is partitioned and replicated across a cluster to guarantee scalability and availability. Because of this, they do not often support transactions. **Key-value stores** have enough structure to mimic the value of relational databases while keeping the advantages of NoSQL.
- **Document databases:** typically store self-describing JSON, XML and BSON documents. They are similar to key-value stores but, in this case, a value is

a single document that stores all data related to a given key. Field in the document can be indexed by the user to provide fast retrieval without knowing the key. Each document can have the same or different structure.

- **Wide-column databases:** use the tabular format of relational databases, but allow a wide variance in how data is named and formatted in each row. A query can retrieve related data in a single operation, since only the columns associated with the query are retrieved. Like key-value stores, they have some basic structure while preserving flexibility.

- **Graph databases:** use graph structures to store, map and query relationships. They are useful for identifying patterns in semi-structured and unstructured information.

2.3.1 Relational vs Non-Relational Database

SQL databases are relational, while NoSQL ones are not. The relational database management system is the basis for structured query language and allows the user to access and manipulate data in structured tables. In NoSQL databases the data access syntax can be different for each one of them.

The data in an relational databases is stored in tables. A table is defined as a collection of related data entries, made by rows and columns. This type of databases requires defining the schema upfront, i.e., all the columns and their associated data-types must be known in advance, so applications can write data to the database. They store information linking different tables through various keys, thus creating a relationship across multiple tables.

In a NoSQL one instead, data can be stored without defining the schema upfront, meaning that the user has the ability to get moving and iterate quickly. Until recently relational databases were the most used models. Nowadays the variety, velocity and volume of data sometimes requires a very different database. This is when the NoSQL approach comes in handy, thanks to its ability to scale out horizontally and quickly.

2.3.2 NoSQL Database Advantages

To summarize, non-relational databases offer important advantages with respect to relational ones, including:

- **Scalability:** NoSQL approach uses horizontal scaling to add or reduce capacity quickly and non-disruptively with commodity hardware. This has the ability

SQL	NoSQL
Relational Database management system	Distributed Database management system
Vertically scalable	Vertically and horizontally scalable
Fixed or predefined Schema	Dynamic Schema
Not suitable for hierarchical data storage	Best suitable for hierarchical data storage
Can be used for complex queries	Not good for complex queries

Table 2.1: SQL vs NoSQL databases

to support increased traffic in order to meet demand with zero downtime. Scalability eliminates the cost and complexity of manual sharing that is necessary when trying to scale relational databases.

- **Performance:** users can increase performance with non-relational databases by adding commodity resources. This allows organization to continue to deliver reliably fast user experience with a return on investment for adding resources.
- **High Availability:** they are generally designed to ensure high availability and avoid the complexity of relational architecture, that relies on primary and secondary nodes. Some distributed NoSQL databases use a masterless architecture that automatically distributes data equally among different resource. Because of this the application remains available for read and write operations even in case of node failure.
- **Global availability:** non-relational databases can minimize latency and ensure a consistent application experience wherever users are located. To achieve this, they automatically replicate data across multiple servers, data centers or cloud resources.
- **Flexibility:** NoSQL has the ability to implement fluid and flexible data models. Users can leverage the data types and query options that are the most natural fit to the specific application use case. This results in a simpler interaction between the application and the database, and a faster, more agile development.

Chapter 3

Amazon Web Services

Amazon Web Services is a subsidiary of Amazon launched in July 2002, when it opened the *Amazon.com* platform, with its first web services, to all developers. **AWS** is the world's most adopted cloud platform, as depicted in Figure 3.1, and provides scalable and cost-effective on-demand cloud computing. It also comes up with APIs on a metered pay-as-you-go basis, either on a per-hour or per-second usage. Fees are based on a combination of usage, hardware, OS, software, networking and service options. If a costumer cannot afford the cost, the AWS Free Tier allows users to access up to 60 services in one of three different options: always free, 12 months free and trials.

AWS offers more services, and more features within them, than any other cloud provider, including a mixture of IaaS, PaaS and packaged SaaS. It provides the widest variety of databases that are purpose-built for different types of applications, therefore the user can choose the right tool for the job. All of this makes it faster, easier and more economical to move the user's existing applications to the cloud.

This cloud provider offers services from dozens of data centers spread across availability zones in regions throughout the world. A single availability zone contains multiple physical data centers, and multiple ones in a geographic proximity, connected by low-latency network links, is called region. The user can choose one or multiple zones for different reasons, such as compliance and proximity to the end costumers. A user can also spin up VMs and replicate data in different regions to ensure reliability and resistance to failures of individual servers or of an entire data center.

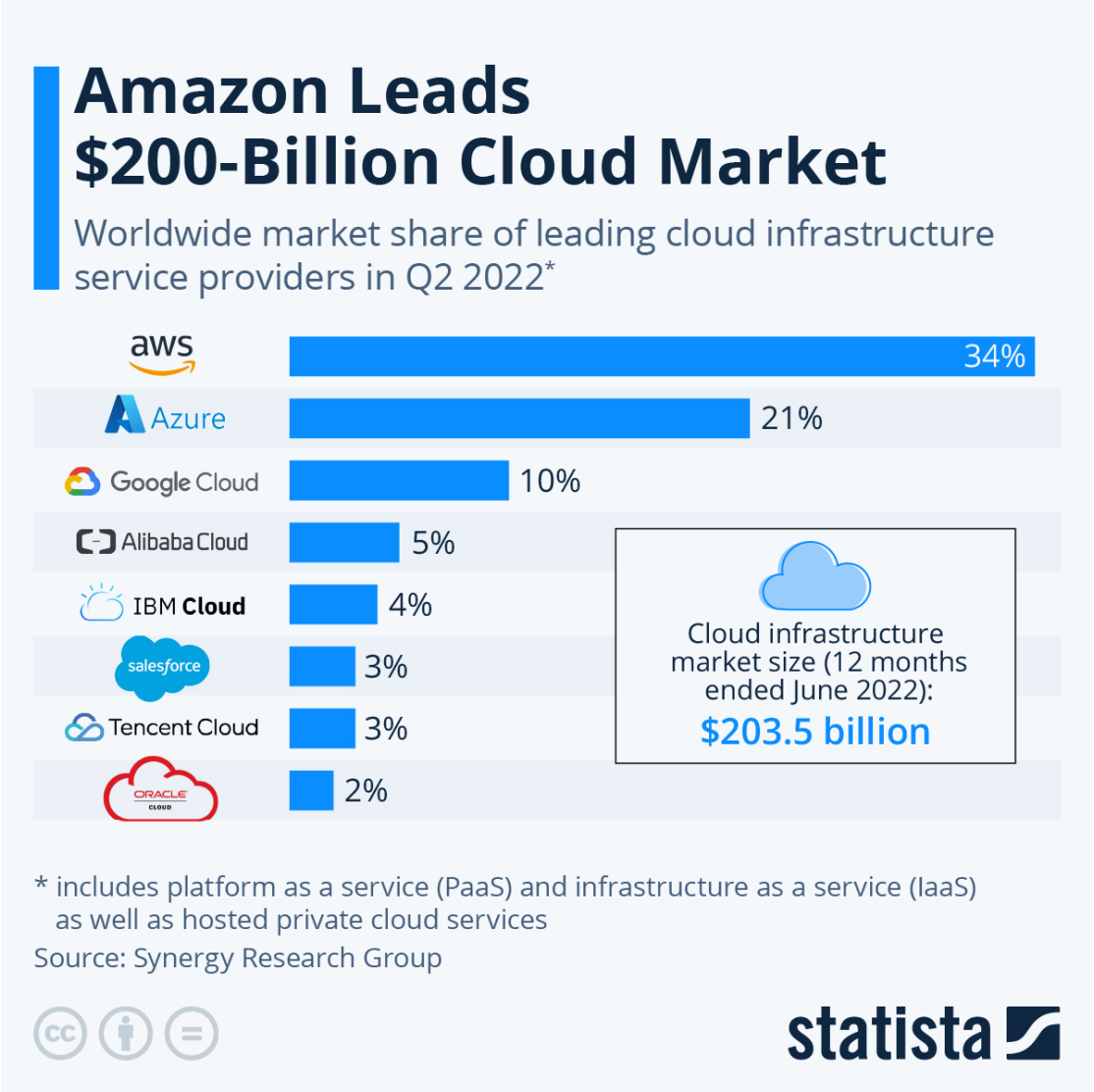

Figure 3.1: Cloud providers market share, source [6]

3.1 Security

AWS offers lots of different tools that can be used, by individuals and companies, in data centers in up to 190 countries. Through the support of 98 different security standards and compliance certifications, the core infrastructure of AWS satisfies the security requirements for the military, banks and other high-sensitivity organizations.

This model has been recognized by Gartner, a technological research and consulting firm, as the recommended approach for running enterprise applications that require high availability. In fact Gartner positioned AWS in the Leaders quadrant of the *2021 Magic Quadrant for Cloud Infrastructure Platform Services (CIPS)*. CIPS are defined, in this context, as "standardized, highly automated offerings, in which infrastructure resources (e.g., compute, networking and storage) are complemented by integrated platform services". [7]

Figure 3.2: Magic Quadrant for Cloud Infrastructure & Platform Services, source [8]

3.2 Services

AWS is made up by more than 100 services, including those for computing, databases, application development and many others. Each one of them can be configured in different ways based on the user's needs. Among the services provided by AWS, Amazon Elastic Compute Cloud (EC2) allows users to have at their disposal a virtual cluster of computers, always available through the Internet. These virtual computers emulate most of the attributes of a real one, including hardware CPUs and Graphics Processing UnitS (GPUs) for processing, local/RAM memory, hard disk/Solid-State Drive (SSD) storage, a choice of OS, networking and many others.
Most services are not exposed directly to end users, but instead offer functionality through APIs for developers to use in their applications. AWS offerings are accessed over HTTP, using the REST architectural style and JSON for newer APIs.

This chapter contains an overview on the main features and advantages of all the services that have been used for this thesis work. First of all the services employed for the multi robot application are analyzed, i.e, AWS Cloud9, Amazon Elastic Container Registry (ECR) and Amazon Elastic Kubernetes Service (EKS). Subsequently AWS RoboMaker, used to simulate the robots, is presented. Lastly AWS Lambda and Amazon API Gateway, useful for the implementation of the REST API displaying the robots' status, are analyzed.

3.3 AWS Cloud9

AWS Cloud9 is a cloud-based Integrated Development Environment (IDE) that lets developers write, run and debug their code with just a browser. It includes a code editor, a debugger and a terminal. This service is prepackaged with essential tools for over 40 programming languages, including Python, Java, C++ and many others. Therefore users do not need to install files and configure their development machine when starting new projects. The cloud-based nature of this IDE allows user to work on their projects everywhere they want, using just an internet-connected machine.

3.3.1 Features

An interesting feature of this service is the quickly sharing of the development environment with colleagues, characteristic that facilitates the collaboration. It updates in real time, so a programmer can see the code entered, edited and deleted from another member of the team as it happens. Colleagues can also chat with one another through a terminal.

The Cloud9 IDE also offers default support for Git, the defacto standard of source control management, and comes with a pre-installed Serverless Application Model (SAM) local environment. The local testing of SAM applications is enabled by Docker, already available in an AWS Cloud9 instance.

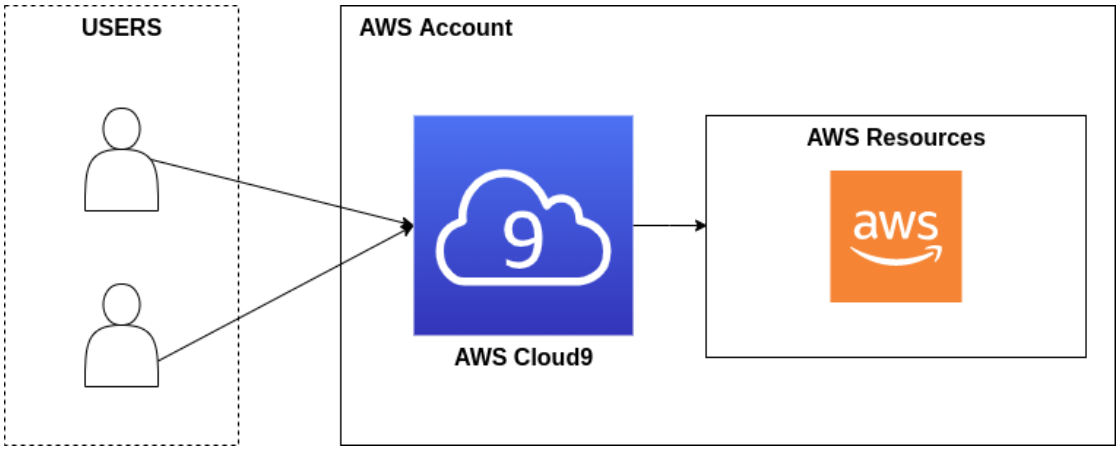

Figure 3.3: AWS Cloud9 workflow

3.3.2 Benefits and security

AWS Cloud9 is useful for developing serverless applications. It preconfigures the development environment with the required Software Development Kits (SDKs), libraries and plug-ins for serverless development. It can be also integrated with other AWS services. Developers can, through Cloud9, build, edit and debug AWS Lambda functions, a service described in Section 3.7. It can run on managed Amazon EC2 instances, or on any SSH-supported Linux server.

The **AWS shared responsibility model** applies to data protection in Cloud9. The cloud provider is responsible for protecting the global cloud infrastructure, while the users are responsible for maintaining control over their content.

3.4 Amazon ECR

Amazon ECR is a fully-managed Docker container registry useful for developers to store, manage and deploy Docker container images. The users do not need to operate their own container repositories or to scale the underlying infrastructure. This service allows developers to save configurations and move them quickly into a production environment, reducing the overall workloads.

ECR provides a CLI and APIs to manage the repositories and its integrated services such as Amazon Elastic Container Service (ECS), which installs and manages the infrastructure for the containers. The main difference between ECR and ECS is that the former comes up with the repositories to store all the code that has written and wrapped up in a Docker image, while the latter takes these files and use them in the deployment of applications.
The developer can use the Docker CLI to push or pull container images to or from an AWS region. Amazon ECR can be used wherever a Docker container service is running.

3.4.1 Features

Amazon ECR writes and packages code in the form of a Docker image. Subsequently it compresses, encrypts and manages access to these images and controls their lifecycle. Lastly Amazon ECS pulls the Docker images from ECR to be used in the deployment of applications and continues to manage containers anywhere.

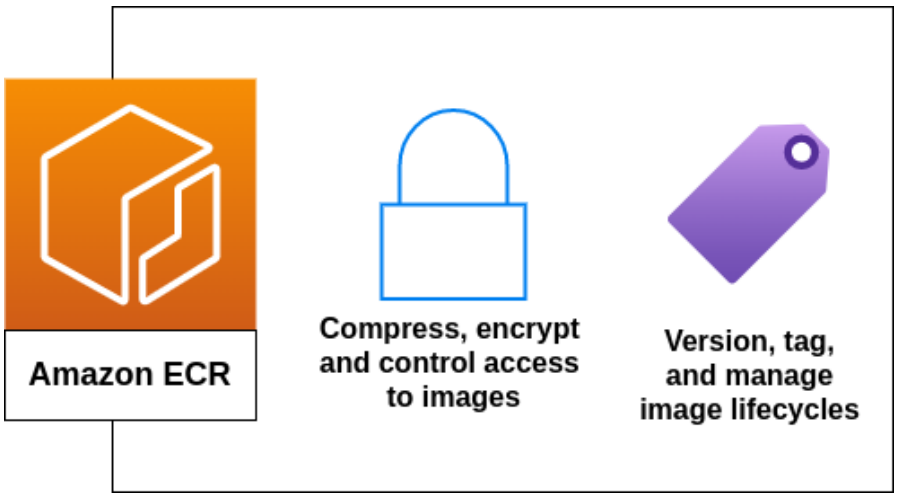

Figure 3.4: Amazon ECR workflow

Amazon ECR is made by the following components:

- **Docker images:** they can be pushed or pulled to the user's repositories. The developers can use them on their local development system or Amazon ECS task definitions and Amazon EKS Pod definitions.
- **Repository:** stores the user's Docker images, OCI images and compatible artifacts. Developers can push and pull images to the repository. ECR lifecycle policies allows to specify the lifecycle management of images inside a repository.

- **Repository policy:** used by developers to manage the access to the repositories and the images within them.
- **Registry:** an Amazon ECR private registry is provided to every AWS account. Developers can create one or more repositories in their registry and store images in them.
- **Authorization token:** before it can push and pull images, the Docker client must be recognized as an AWS account holder. The AWS CLI *get-login* command provides the user with authentication credentials to pass to Docker.

3.4.2 Benefits and security

The main benefits of ECR are:

- **High availability:** ECR architecture is scalable, durable and redundant. Therefore the Docker images are easily available and accessible, and users can dependably deploy new containers for their application.
- **Streamlines workflow:** the integration with Amazon ECS and the Docker CLI allows users to simplify their development and production process by facilitating Continuous Integration/Continuous Delivery (CI/CD) in ECS. Container images can be pushed to Amazon ECR through the Docker CLI and Amazon ECS can easily pull the images directly and use them for production deployments.
- **Fully managed**: Amazon ECR does not include software that needs to be installed and managed, or an infrastructure that must be scaled.

An important trait of Amazon ECR is its increased security. It encrypts images at rest with Amazon S3 server-side encryption, an AWS service that provides object storage through a web service interface. ECR also allows administrators to use AWS Identity and Access Management (IAM), an AWS product that specifies who or what can access services in AWS, to create restrictions to limit the access for the repositories. AWS security groups, virtual firewalls at the instance level, can be selected for the interface that controls whether each host is allowed to interact with that interface.

3.5 Amazon EKS

Amazon EKS is a cloud-based container management service useful to run Kubernetes on AWS without the need to install, control and maintain the control

plane and the nodes. It automatically manages and scales clusters of infrastructure resources.

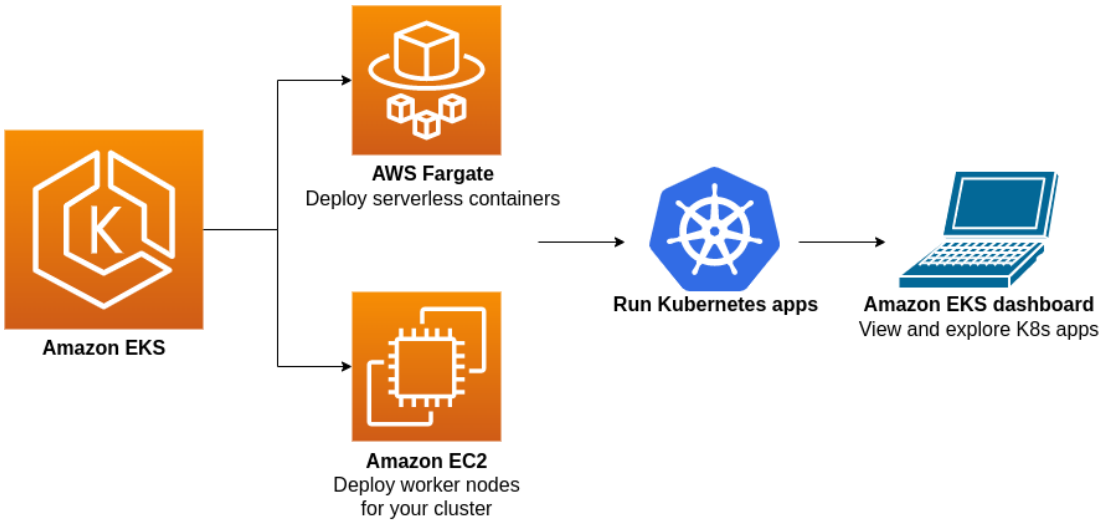

Figure 3.5: Amazon EKS workflow

3.5.1 Components

An EKS cluster is made of a control plane and multiple nodes.

EKS Control Plane

The control plane, that consists of at least two API servers and three etcd instances, runs on a set of EC2 instances in an AWS account. It uses three load-balanced master nodes arranged in a high-availability configuration. They are managed by AWS and carry the functionalities needed to implement Kubernetes, including access to the EKS API.
Data on *etcd* is encrypted using Amazon KMS, a service that lets the user create, manage, and control cryptographic keys across the applications and more than 100 AWS services. Kubernetes master nodes are distributed across different availability zones. Role-Based Access Control (RBAC) policies can be used to authorize a cluster to view or receive communication from other clusters or other AWS accounts. The user can create an EKS cluster for each application or use one cluster for multiple applications. The second case requires the use of IAM security and Kubernetes namespaces to isolate applications inside the cluster.

EKS Nodes

Worker nodes are created by EKS users via EC2 instances. They host pods of containers that compose container-based applications. Nodes are in general arranged in node groups, and different node groups can be created inside the same cluster. The worker nodes communicate with the control plane via the EKS API, RBAC and Amazon Virtual Private Cloud (VPC).

Self-Managed Nodes

Nodes are organized into node groups. Every EC2 instance in a node group must have the same Amazon instance type, Amazon Machine Image (AMI) and IAM role. An AMI is a supported and maintained image that provides the information required to launch a given instance. The user can have different node groups in a cluster, each one representing a different type of instance or instances with different role.

Managed Node Groups

Amazon EKS provides managed node groups with automated lifecycle management. This allows the user to create, update or shut down nodes with only one operation. When the user terminates nodes, EKS drains them to make sure there is no interruption of service.
Managed nodes are operate using EC2 Auto Scaling groups, that are managed by the Amazon EKS service. The user can define in which availability zones the groups should run on.

3.5.2 Features and benefits

A developer that works with EKS will provide the worker nodes and link them to the Amazon EKS endpoints. AWS handles all the management tasks for the Kubernetes control plane such as patches, updates and security configurations. The user must create an IAM role, a VPC and a security group for its cluster. The customer should create different VPCs per cluster in order to increase network isolation.

Kubernetes uses pods, or group of containers, to orchestrate and scale servers. Amazon EKS automatically replicates master schedulers across three availability zones in each AWS region to ensure higher availability. It also checks for unhealthy control plane instances and automatically replaces them, restarting them across the zones within the AWS Region as needed.

EKS provides a series of features that focus on several operational areas such as:

- **Cluster management:** EKS provides a managed control plane and node groups, a hosted Kubernetes console, AWS service integrations and support for a large library of Kubernetes add-ons.
- **Network management:** it controls networking and security through support for IPv6, service discovery, IAM authentication and compliance with a series of regulatory requirements.
- **Load balancing:** EKS supports load balancing via the Application Load Balancer, Network Load Balancer and Classic Load Balancer.
- **Serverless computing:** it allows serverless computing using AWS Fargate, a service that can be integrated with Amazon ECS to run containers without having to manage clusters of Amazon EC2 instances or servers.
- **Logging:** EKS uses AWS CloudTrail and Amazon CloudWatch for logging and analysis of its environment. The former helps the users enable operational and risk auditing, governance, and compliance of their AWS account, while the latter is a metrics repository.
- **Updating:** EKS supports easy updates, allowing to rapidly update to the latest Kubernetes version without significant disruption to existing clusters or applications. Users can select the desired Kubernetes version through the CLI, SDK or AWS Console.
- **Eksctl:** the eksctl CLI automates the creation and control of EKS clusters, and other individual tasks.

Amazon EKS also joins different Kubernetes-based container services available on the market, including Google Kubernetes Engine, Microsoft Azure Kubernetes Engine and RedHat OpenShift. Scripts, plugins and cluster configurations can be easily moved among these platforms since they rely on the same orchestration layer.

3.6 AWS RoboMaker

Developer teams design code to cover a wide range of deployments situations, integrate the code and test the application on robotics hardware in real life scenarios. This manual approach wastes a lot of time, it necessitates expensive technology and it slows program update. To overcome these problems, AWS launched RoboMaker

back in 2020.

AWS RoboMaker is a cloud-based simulation service that makes it easy to develop, test and simulate robotics application at scale without managing any infrastructure. By means of this service, users can cost-effectively scale and automate simulation workloads, run large-scale and parallel simulations with a single API call and create 3D virtual environments. It is capable of automated testing within a CI/CD pipeline, training reinforcement models and connecting multiple synchronous simulations to the fleet management software for testing.
AWS RoboMaker also extends ROS, described in Appenidx A, with connectivity to different cloud services like Amazon Kinesis Video, a service designed to process large-scale data streams from different services in real-time, and Amazon Lex, that builds conversational interfaces for applications using voice and text. The integration of RoboMaker with machine learning, monitoring and analytics services enables the robot to stream data, communicate and navigate. It comes up with a development environment for application development and a simulation service for application testing.

3.6.1 Features

AWS RoboMaker comes up with the following features:

- **Development:** RoboMaker provides a customized environment in AWS Cloud9 for robotics development, configured with ROS/ROS2 and integrated with other capabilities. The user can manage build configurations, create simulation jobs and interact with running simulation via graphical tools. The provided CLI supports the Gazebo simulator, rviz, the visualization engine for ROS, and rqt, a QT-based framework for ROS Graphical User Interfaces (GUIs).

- **Creating 3D world with Simulation WorldForge:** Simulation WorldForge automatically creates hundreds of pre-defined and randomized simulation worlds, that implements real-world conditions without managing world generation infrastructure.

- **Simulation:** a fully managed service that allows the user to run simulation jobs without provisioning or managing any infrastructure. RoboMaker supports large-scale and parallel simulations, and it automatically scales them depending on the complexity of the tested scenario. The simulation can be used to run the robot software and simulator choice such as ROS, Gazebo, Unreal and Unity.

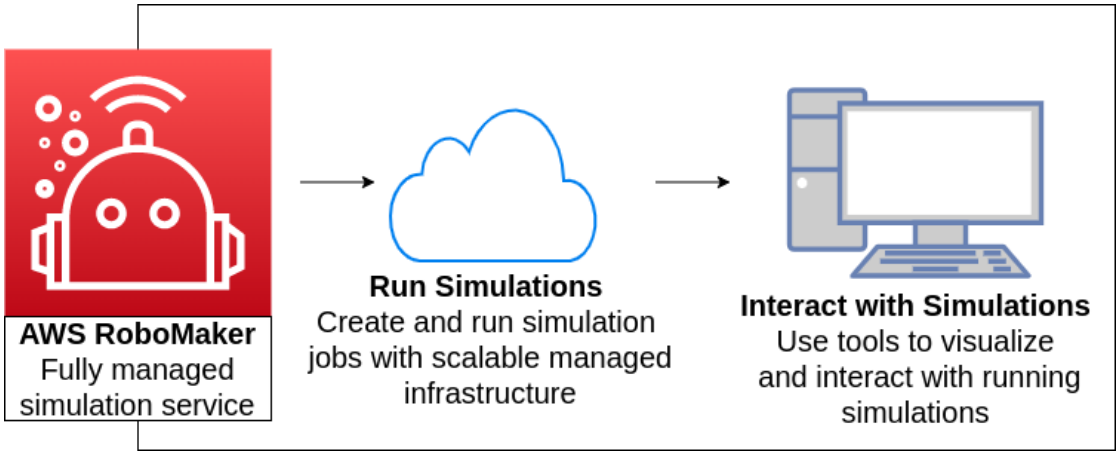

Figure 3.6: AWS RoboMaker features

AWS RoboMaker can also be integrated with Amazon CloudWatch and Amazon Simple Storage Service (S3) for job monitoring and logging of simulation data.

3.7 AWS Lambda

AWS Lambda is an event-driven computing cloud service that allows users to program functions on a pay-per-use basis. Developers are not required to provision storage or compute resources to support their functions. This model is also referred to as Function as a Service (FaaS). Lambda runs the users' code on a high-availability compute infrastructure and performs the resources administration, including server and OS system maintenance, capacity provisioning, automatic scaling and logging. Users can list, delete, update and monitor functions via the dashboard, CLI or SDK. AWS Lambda also performs infrastructure-focused activities, such as servers and OS maintenance, patch deployment and logging through AWS CloudWatch. It also supports third-party logging APIs and developers can connect custom APIs endpoint to Lambda through Amazon API Gateway.

3.7.1 Lambda Functions

Developers organize their code into Lambda functions. This service runs the user's function only when needed and scales automatically, from few daily requests to thousands per second.

A function is a piece of programming that implement a specific task. Developers use this service to code and run functions in response to specific events in

other AWS services. Each Lambda function runs inside an isolated computing environment with its resources and view of the file system. The storage and compute resources for a given function spin up automatically as a metered service as soon as that function is called. Through AWS Lambda the user can run code for any type of application, it takes care of everything required to run and scale the code with high availability. The developer can set up the code to automatically trigger from other AWS services or call it directly from a mobile or web app.

Users can use AWS Lambda to:

- process streaming data stored in Amazon Kinesis.
- Build data-processing triggers for AWS services such as Amazon S3 and Amazon DynamoDB.
- Create their own back-end that operates at AWS scale, performance and security.

3.7.2 Features

The following features help users to develop scalable, secure and easily extensible Lambda applications:

Concurrency and scaling controls

Concurrency and scaling controls give the users fine-grained control over the scaling and responsiveness of their production applications.

Code signing

Code signing provides trust and integrity controls that let user verify that only unaltered code, approved by developers, is deployed as Lambda function.

Functions defined as container images

Developers can use any container image tooling, workflow and dependencies to build, test and deploy Lambda functions.

Lambda extensions

Lambda extensions can be used to augment the user's Lambda functions. An example can be to use extensions to easily integrate this service with tools for monitoring, observability, security and governance.

File systems access

The user can configure a function to mount an Amazon Elastic File System (EFS), a file storage service for applications and workloads that run in the AWS public cloud, to a local directory. With this service the code can access and modify shared resources safely and at high concurrency.

Function blueprints

This feature provides sample code showing how to use Lambda with other AWS services or third-part applications. Blueprints include sample code and function configuration presets for Python and Node.js runtime.

Database access

A database proxy manages a pool of database connections and relays queries from a function. This permits functions to reach high concurrency levels without exhausting database connections.

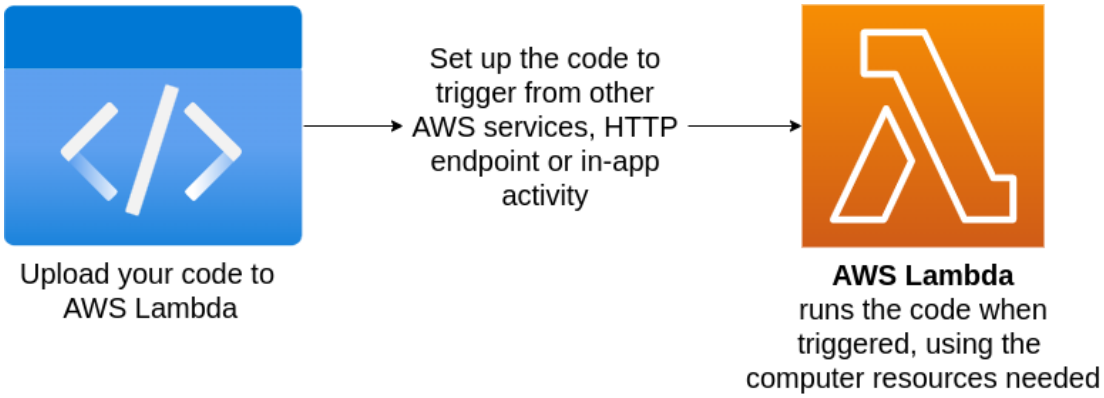

Figure 3.7: AWS Lambda architecture

3.7.3 Benefits

One of the main benefits of AWS Lambda is that it manages the servers, allowing the developers to focus more on writing application code. Because Lambda manages these resources, the user cannot log in to compute instance or customize the OS on provided runtimes. It supports code written in different programming languages, such as Node.js, C, Java and Python. Users can also integrate this service with compiler tools like Maven or Gradle, and packages to build functions.

3.8 Amazon API Gateway

An API Gateway is the middleware that sits between an API endpoint and backend services, sending client requests to an appropriate service of an application. It accepts and processes concurrent API call, which take place when APIs submit requests to a server. An API Gateway can also handle any type of interaction between the user's website, web or mobile application, IoT devices and microservices.

Amazon API Gateway is a closed-source SaaS used to create, publish, maintain, monitor and secure Hypertext Transfer Protocol (HTTP), REST and WebSocket APIs. It enables developers to connect non-AWS application to AWS back-end resources. Users can build APIs that access AWS, other web services and data stored in the AWS Cloud. It supports containerized and serverless workloads, as well as web applications. This service manages traffic, authorized end users and monitors performance. Amazon API Gateway can be connected to other AWS services like AWS Lambda.

By means of few clicks in the AWS Management Console, the user can create an API, a front door for applications to access data, business logic or functionality from the user's back-end services.

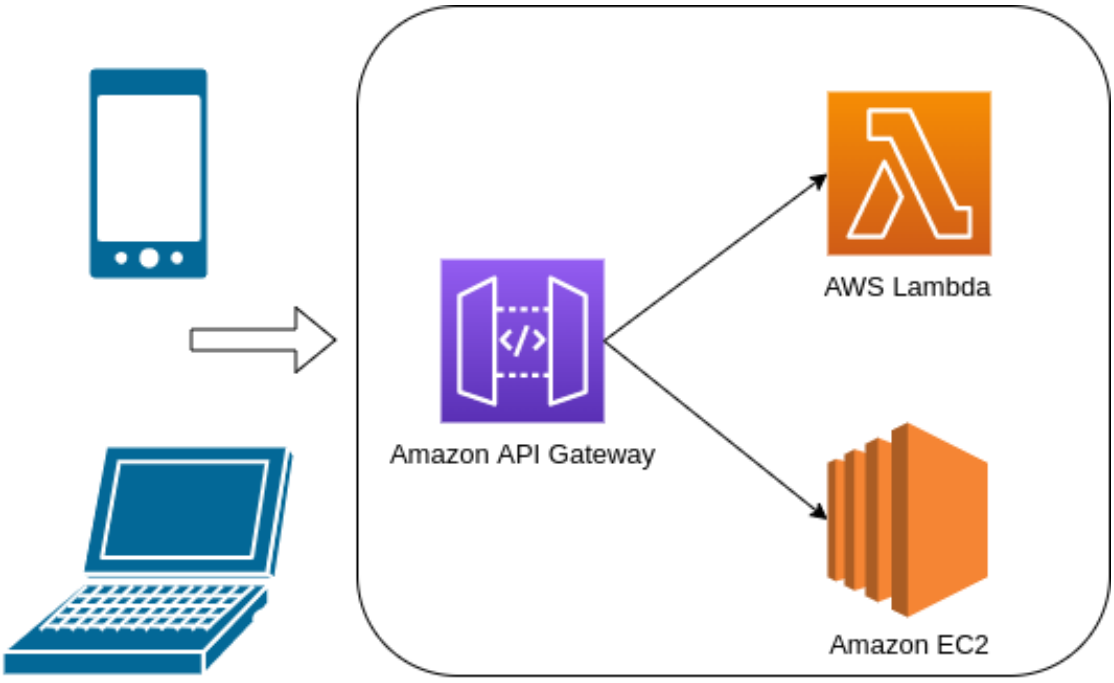

Figure 3.8: Amazon API Gateway workflow example

3.8.1 Features

When creating an API, the user must define its name, an HTTP function, the integration between the API and other services and how requests and transfers are handled. Through an SDK the developer can integrate with software that calls the APIs. When the user makes API calls to API Gateway, the service triggers a Lambda function, which sends back the response to Amazon API Gateway.
This service accepts all payload sent over HTTP and users can monitor API calls through a dashboard. They can also retrieve errors, access and debug logs from Amazon CloudWatch.
This service handles API call traffic in two main ways:

- **Throttling:** limits the number of API call per day or per hour. It helps maintain the performance of calling applications during unexpected spike in calls, which can happen when many users use an application at the same time.
- **Caching:** different API calls use the same information and return the same result. Caching provides common API responses, rather than performing all the process required to produce a given result. It also reduces the number of API calls and improves the performance of calling applications.

Amazon API Gateway support REST and WebSocket APIs.

REST APIs

They communicate with a server via HTTP methods, such as GET, POST, PUT and DELETE, the same method used to access web pages and create a resource. With Amazon API Gateway, REST APIs are used for serverless workload and HTTP back-ends using HTTP APIs. HTTP APIs are the best choice for building APIs that require only API proxy functionality.

WebSocket API

It enables a full-duplex communication channel, between client and server, over a single TCP connections. It facilitates client-server communication in real-time applications. Differently from HTTP APIs, that rely on the client to initiate communication, this API allows the server to send messages to the client without any request.

3.8.2 Management components

API management platforms support the API infrastructure and the underlying data. They incorporate several components in a layered architecture. Other API management components are:

- **API developer portal:** lets users access documentation and other information related to an API in one place. It contains resources and tool like SDK and API keys.
- **Reporting and analytics** tracks usage metrics, including data such as requests per second, request volume, latency and throughput.
- **API lifecycle management:** describes the steps in the development and maintenance of an API. This process begins with the design of the API and continues with the development, testing and deployment. It ends with the retirement of the API.
- **API policy manager:** controls the policies that define and manage APIs. The policies go through the same lifecycle as the API. They control API traffic, security and performance. Management platform often provide policies that can be implemented without new code or changes to back-end services.

3.8.3 Security and benefits

Amazon API Gateway provides security through access keys to control API access. This service work with AWS IAM and Amazon Cognito, a service that controls user authentication and access for mobile applications on internet-connected devices, to authorize access to APIs. It also supports AWS Signature Version 4 to add authentication information to AWS API requests sent by HTTP, creating access keys for every API call. As an alternative security measure, OAuth tokens can also be passed to running workload.

This service lets developers operate multiple versions of an API simultaneously. In this way they can build and deploy new APIs while existing applications use previous versions. It provides the end user with the lowest possible latency for API requests and responses.
The user does not have to worry about having Autoscaling groups responding to API requests, since API Gateway scales automatically.

3.9 Amazon DynamoDB

Amazon DynamoDB is a fully managed NoSQL database service known for low latency and its scalability. It makes easy to store and retrieve data, as well as serve any level of traffic. All data is stored on SSD which provides high I/O performance and can handle high-scale requests in an efficient manner. Users can interact and work with this service by means of the AWS dashboard or the DynamoDB API.

3.9.1 Features

The DynamoDB Triggers feature integrates with AWS Lambda to allow users to code actions based on updates to items on a DynamoDB table. The developer associates a Lambda function with the stream on a DynamoDB table, subsequently AWS Lambda read updates to a table from a stream and executes the function. DynamoDB provides two consistency options when reading data:

- **Eventually consistent:** in this case, when the user reads data from a table, the response might not give back the result of a recently completed write operation. The response might include some stale data. If the users repeats the request after a short amount of time, the response should throw back the latest data.

- **Strongly consistent:** with this option, DynamoDB returns a response with the most up-to-date data, returning the updates from all prior write operations that were successful. However this consistency might not be available if there is a network delay, and the reads may have higher latency than eventually consistent reads. Furthermore the strongly consistent reads use more throughput capacity than eventually consistent ones.

3.9.2 Benefit and security

This service execute replication across three availability zones for high availability, durability and read consistency. Developers can also go for cross-region replication, that creates a backup copy of a DynamoDB table in one or more global geographic locations.

Amazon DynamoDB comes up with Fine-Grained Access Control for an administrator to secure data in a table. The admin or the table owner can specify which users can access which items and what action that person can perform. This control feature is based on AWS IAM service.

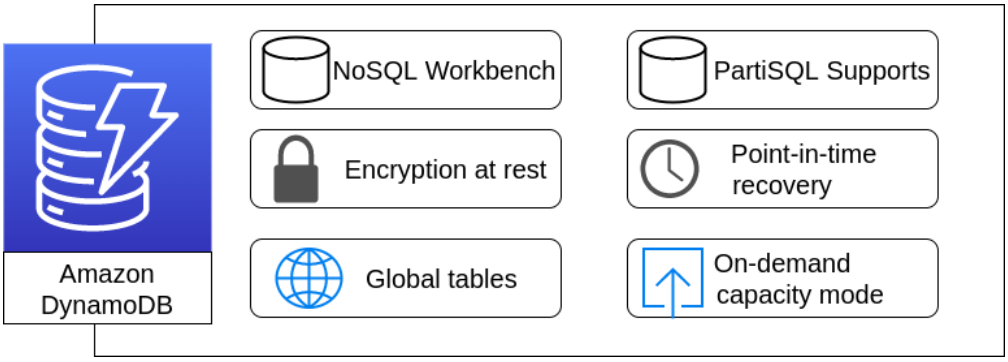

Figure 3.9: Amazon DynamoDB features

Chapter 4

Cloud Robotics Architecture

This chapter contains a description of the developed cloud robotics architecture, represented in Figure 4.1.

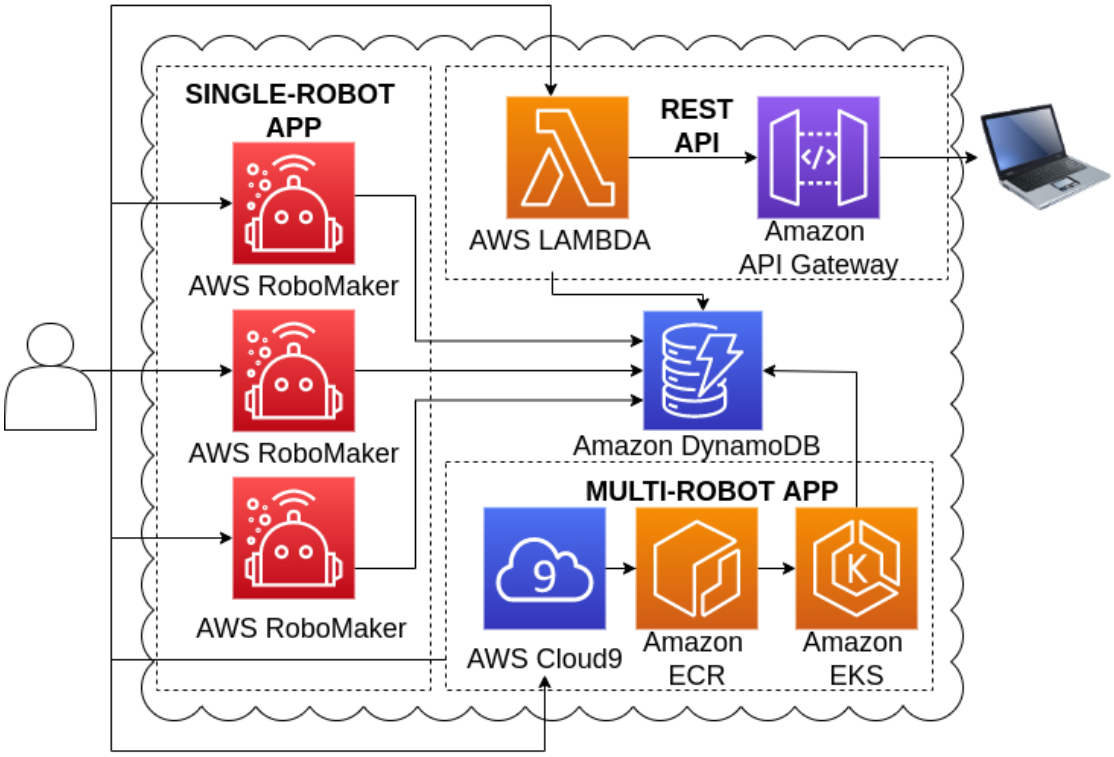

Figure 4.1: Cloud robotics architecture

It is made by three main components: the single-robot application, the multi-robot management algorithm and a REST API that shows the robots' status. This

different sections are connected one another through a non-relational database, the focal point of the proposed solution.

4.1 Database configuration

The NoSQL database provided by Amazon DynamoDB, depicted in Figure 4.2, plays a crucial role in this architecture, since it is the bridge between the fleet management software and the simulated robots. It also stores the devices' status, that are displayed in a web page through a REST API.

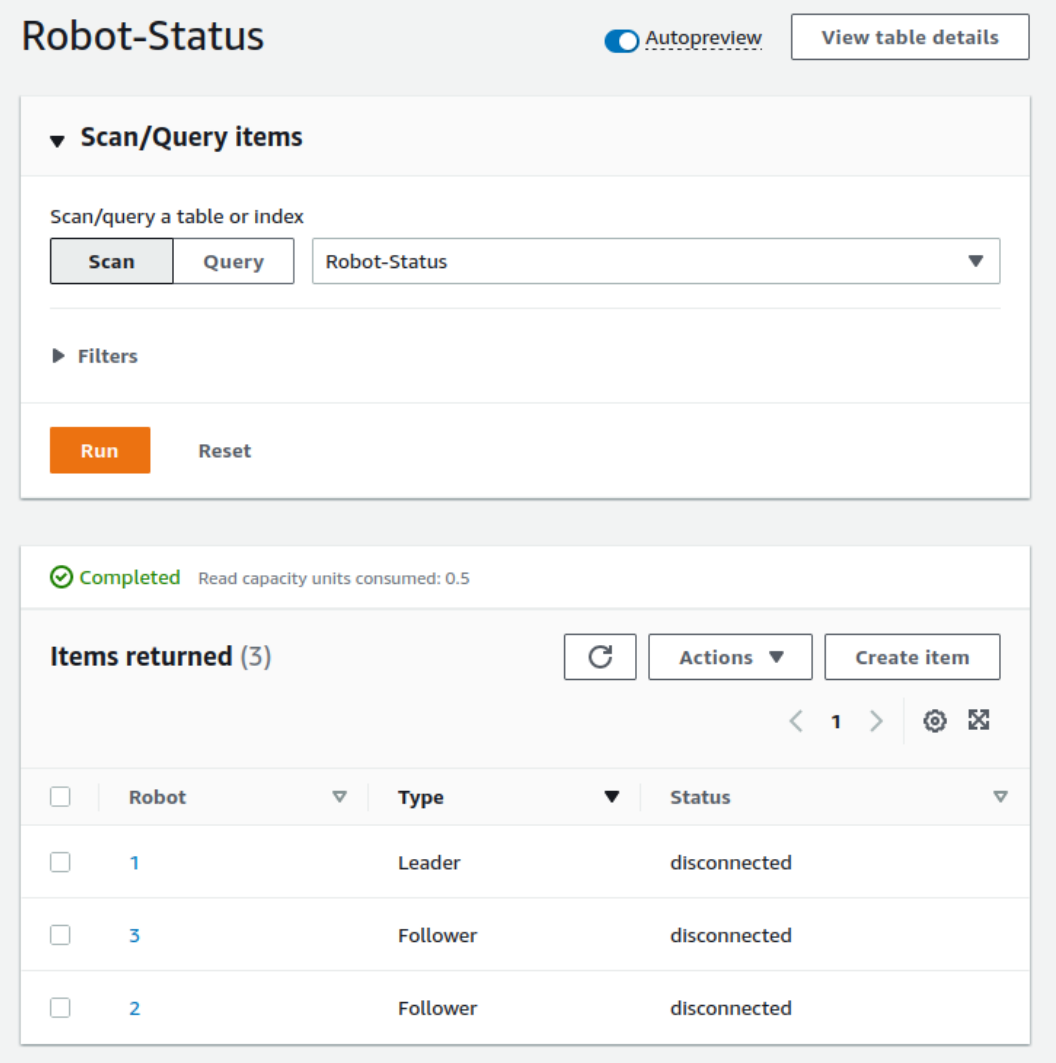

Figure 4.2: Robot-Status Database

This choice was done because of its ability to scale horizontally, therefore it can efficiently handle an increasing number of robots.

In this thesis work the database uses a partition key named *Robots* of type *Number*, to properly distinguish the different devices, and a sort key named *Type* of type *String*, representing the kind of robots, i.e. leader or follower. This distinction was done since the robots are controlled via a "Leader Follower Approach", described in section 4.2, that allows the followers to complete their tasks if and only if the leader is being simulated.

An item was created for each robot, using also the attribute *Status* of type *String*. This last parameter was set to denote the different conditions of the devices:

- **connected:** that is when the simulation is running,
- **disconnected:** once the simulation is stopped,
- **disconnecting:** when the multi robot algorithm updates the database to stop the followers simulation.

This attribute was set as disconnected by default for every item, and they were subsequently updated by the simulated robots and the fleet management software.

4.2 Multi robot application

The multi robot algorithm controls the robots using the **Leader Follower Approach**. One device, the leader, can complete its task independently of the followers' status. The followers instead can run the simulation if and only if the leader is connected.

To implement the control logic, the Kubernetes' Pod containing this algorithm constantly performs a query operation on the leader's status and, in case in which it reads *disconnected*, it updates the followers' status to *disconnecting*. As a further security level, these robots constantly read their status and, once it is *disconnecting*, update it to *disconnected* and exit the simulation. The sequence diagram describing this control logic is depicted in Figure 4.3.

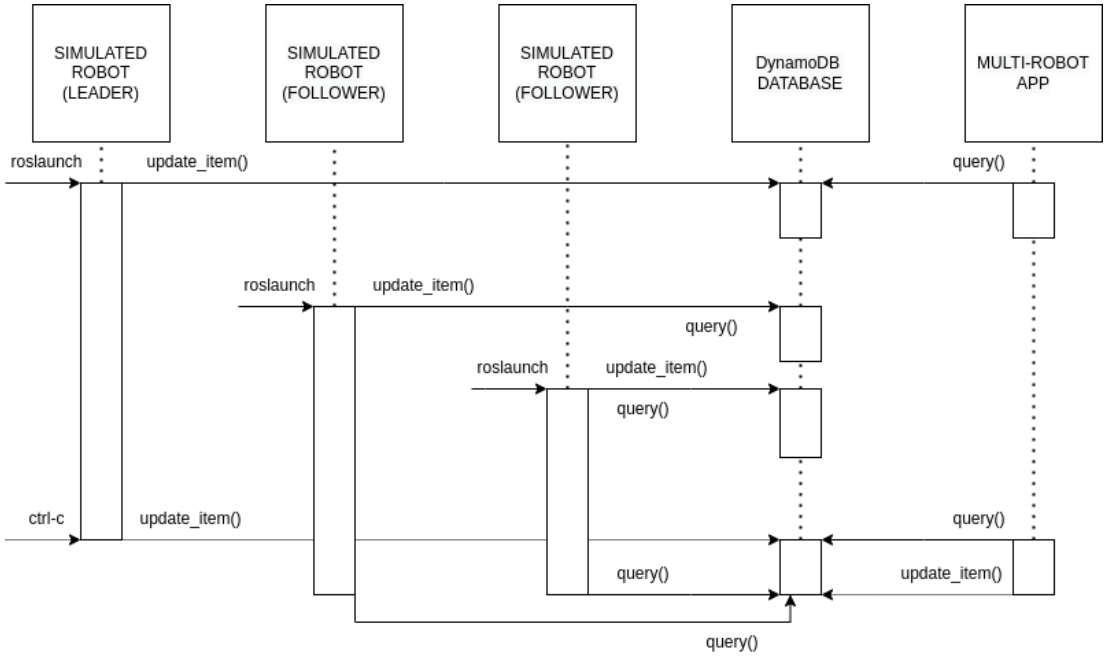

Figure 4.3: Fleet management sequence diagram

The fleet management software was developed through the following Python script, using an AWS Cloud9 instance.

```
#!/usr/bin/env python3

import boto3
from boto3.dynamodb.conditions import Key
import time

while True:

    TABLE_NAME = "Robot-Status"

    # Creating the DynamoDB Client
    dynamodb_client =
    boto3.client('dynamodb',region_name="eu-central-1",
    aws_access_key_id="xxxx",aws_secret_access_key="xxxx")

    # Creating the DynamoDB Table Resource
    dynamodb=boto3.resource('dynamodb',region_name="eu-central-1",
    aws_access_key_id="xxxx",aws_secret_access_key="xxxx")
    table=dynamodb.Table(TABLE_NAME)
```

```
# Use the DynamoDB client get item method to get a single item
response1=dynamodb_client.query(
    TableName=TABLE_NAME,
    KeyConditionExpression='Robot = :Robot',
    ExpressionAttributeValues={
    ':Robot': {'N': str(1)}
    }
)
print(response1['Items'][0]['Status']['S'])
# Use the DynamoDB client get item method to get a single item
response2=dynamodb_client.query(
    TableName=TABLE_NAME,
    KeyConditionExpression='Robot = :Robot',
    ExpressionAttributeValues={
    ':Robot': {'N': str(2)}
    }
)

print(response2['Items'][0]['Status']['S'])
# Use the DynamoDB client get item method to get a single item
response3=dynamodb_client.query(
    TableName=TABLE_NAME,
    KeyConditionExpression='Robot = :Robot',
    ExpressionAttributeValues={
    ':Robot': {'N': str(3)}
    }
)
print(response3['Items'][0]['Status']['S'])

if response1['Items'][0]['Status']['S'] == "disconnected" and
response2['Items'][0]['Status']['S'] == "connected":
    response2 = table.update_item(
        Key={
            'Robot': 2,
            'Type':'Follower'
            },
            UpdateExpression = "set #st = :s",
            ExpressionAttributeValues={":s" : "disconnecting"},
            ExpressionAttributeNames={"#st":"Status"},
            ReturnValues="UPDATED_NEW"
        )

if response1['Items'][0]['Status']['S'] == "disconnected" and
response3['Items'][0]['Status']['S'] == "connected":
```

```
        response3 = table.update_item(
            Key={
                'Robot': 3,
                'Type':'Follower'
                },
                UpdateExpression = "set #st = :s",
                ExpressionAttributeValues={":s" : "disconnecting"},
                ExpressionAttributeNames={"#st":"Status"},
                ReturnValues="UPDATED_NEW"
            )
    time.sleep(0.5)
```

In this script the AWS access key id and secret access key were overshadowed for security reasons. They were needed for the containerized image to access the NoSQL database described in the previous section. The "while True:" condition instead was added in order to have the multi robot image always running inside a Kubernetes' Pod.
This script was then containerized through the following Dockerfile, and uploaded into an Amazon ECR's private repository.

```
FROM python:3
ADD multirobot.py /
RUN pip install boto3
CMD [ "python", "./multirobot.py"]
```

Subsequently, to ensure high availability and scalability, the created image was offloaded inside a Kubernetes' Pod, provided by Amazon EKS, through the following YAML file.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: multirobotapp
  namespace: multi-robotic-application
  labels:
    app: aws-ecr
spec:
  replicas: 3
  selector:
    matchLabels:
      app: aws-ecr
  template:
```

```
    metadata:
      labels:
        app: aws-ecr
    spec:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: kubernetes.io/arch
                operator: In
                values:
                - amd64
                - arm64
      containers:
      - name: aws-ecr
        image:
   060472927758.dkr.ecr.eu-central-1.amazonaws.com/robomaker-helloworld-
        robot-app:latest
        ports:
        - name: http
          containerPort: 80
        imagePullPolicy: IfNotPresent
      nodeSelector:
        kubernetes.io/os: linux
    )
```

A service was also created, through the following YAML file, to access all replicas through the same IP address or name.

```
apiVersion: v1
kind: Service
metadata:
  name: multirobotservice
  namespace: multi-robotic-application
  labels:
    app: aws-ecr
spec:
  selector:
    app: aws-ecr
  ports:
    - protocol: TCP
      port: 80
```

```
    targetPort: 80
```

The EKS Kubernetes cluster, and its resources, cannot access the DynamoDB table by default. Therefore the following permission was added to the node group containing the fleet manager software.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "VisualEditor0",
            "Effect": "Allow",
            "Action": [
                "dynamodb:Query",
                "dynamodb:UpdateItem"
            ],
            "Resource": "arn:aws:dynamodb:*:060472927758:table/*"
        }
    ]
}
```

4.3 Single robot application

The robots used for the simulation were the **TurtleBot3 Waffle Pi**, depicted in Figure 4.4. This device is a small, affordable, ROS-based mobile robot used in education, research and product prototyping. Its goal is to reduce the size of the platform and lower the price without affecting its functionality and quality.

The robots were developed and simulated through different AWS RoboMaker instances, a service that provides an IDE integrated with ROS and the Gazebo simulator.

Since June the 27th however, AWS ended the support for the AWS RoboMaker development environment feature. Hence the remaining instances, used to develop and simulate the followers, were manually set up following the steps suggested in [9]. A relevant aspect to emphasize is that, if the user is creating an instance in a region different from us-east-1, the ImageID of the document **SSM for ROS Melodic** has to be changed. To overcome the former problem the ImageID *ami-0f3f0cd0c5fc276ae*, retrieved from [10] by setting eu-central-1 as region and 18.04LTS as Ubuntu version, was used.

For this thesis work, Melodic was chosen as ROS distribution, while the Gazebo9 simulator was used to interact with the simulations.

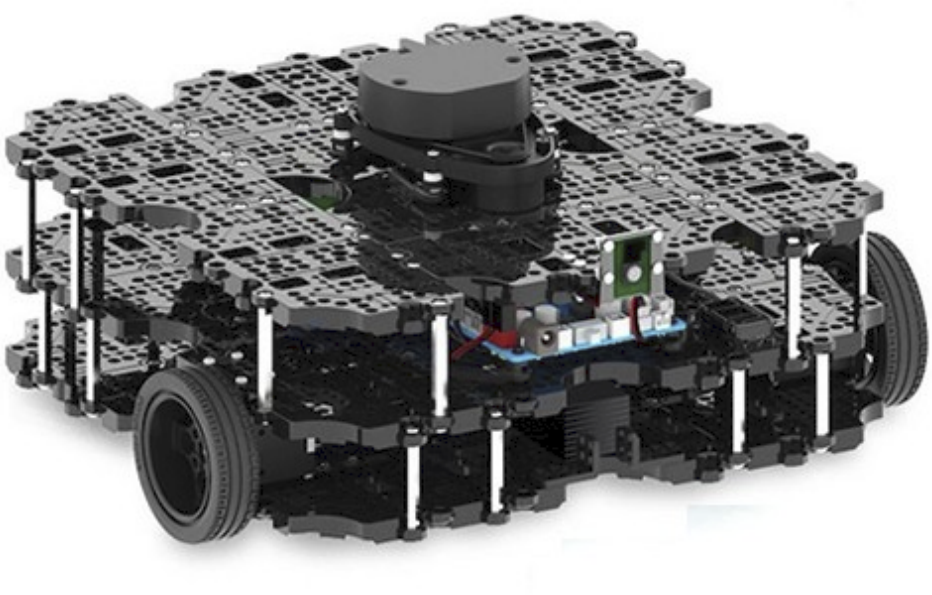

Figure 4.4: TurtleBot3 Waffle Pi, source [11]

The single robot algorithm is made by a ROS node that makes the robot rotate around the vertical axis, the blue one depicted in Figure 4.4, with a constant angular velocity. Once the algorithm is launched, the robot also updates its respective status to *connected* using the UpdateItem API of boto3, the SDK of Amazon DynamoDB. The stopper function instead makes the robot update back its status to *disconnected* once the simulation is stopped.

The Python script used to simulate the leader is the following.

```
#!/usr/bin/env python

from geometry_msgs.msg import Twist
import rospy
import boto3
from boto3.dynamodb.conditions import Key
```

```
import signal
import time


TABLE_NAME = "Robot-Status"


class Rotator():

    # Creating the DynamoDB Table Resource
    dynamodb=boto3.resource('dynamodb', region_name="eu-central-1")
    table=dynamodb.Table(TABLE_NAME)


    def __init__(self):
        self._cmd_pub = rospy.Publisher('/cmd_vel', Twist, queue_size=1)

        # Use the DynamoDB client update the status of Robot 1
        response=self.table.update_item(
            Key={
                'Robot': 1,
                'Type':'Leader'
            },
            UpdateExpression = "set #st = :s",
            ExpressionAttributeValues={":s" : "connected"},
            ExpressionAttributeNames={"#st": "Status"},
            ReturnValues="UPDATED_NEW"
        )

    def rotate_forever(self):
        self.twist = Twist()

        r = rospy.Rate(10)
        while not rospy.is_shutdown():
            self.twist.angular.z = 0.1
            self._cmd_pub.publish(self.twist)
            rospy.loginfo('Rotating robot: %s', self.twist)
            r.sleep()

    def stopper(self):
        response=self.table.update_item(
        Key={
            'Robot': 1,
            'Type':'Leader'
        },
```

```
        UpdateExpression = "set #st = :s",
        ExpressionAttributeValues={":s" : "disconnected"},
        ExpressionAttributeNames={"#st": "Status"},
        ReturnValues="UPDATED_NEW"
    )
        self.twist = Twist()

        r = rospy.Rate(10)
        while not rospy.is_shutdown():
            self.twist.angular.z = 0
            self._cmd_pub.publish(self.twist)
            rospy.loginfo('Rotating robot: %s', self.twist)
            r.sleep()


def main():
    rospy.init_node('rotate')

    def handler(*args):
        rotator.stopper()
        exit(1)

    try:
        rotator = Rotator()
        signal.signal(signal.SIGINT, handler)
        rotator.rotate_forever()
    except rospy.ROSInterruptException:
        pass


if __name__ == '__main__':
    main()
```

The previous code properly implements the leader robot, since it completes its task independently from the other devices. However it is not sufficient to develop the followers, since it does not handle the multi robot algorithm.

Therefore the following script was used to design the second robot. For the remaining follower, the same script has been adopted, with the only difference that the Query and UpdateItem API refer to the third one.

```
#!/usr/bin/env python
```

```
from geometry_msgs.msg import Twist
import rospy
import boto3
from boto3.dynamodb.conditions import Key
import signal
import time
import threading

TABLE_NAME = "Robot-Status"


class Rotator():

    # Creating the DynamoDB Client
    dynamodb_client = boto3.client('dynamodb',
   region_name="eu-central-1")

    # Creating the DynamoDB Table Resource
    dynamodb=boto3.resource('dynamodb', region_name="eu-central-1")
    table=dynamodb.Table(TABLE_NAME)

    def disconnecter(self):
            while True:
                response=self.dynamodb_client.query(
                    TableName=TABLE_NAME,
                    KeyConditionExpression='Robot = :Robot',
                    ExpressionAttributeValues={
                        ':Robot': {'N': str(2)}}
                )
                time.sleep(1.5)

                if response['Items'][0]['Status']['S']=='disconnecting':
                    response=self.table.update_item(
                        Key={
                                'Robot': 2,
                                'Type':'Follower'
                            },
                            UpdateExpression = "set #st = :s",
                            ExpressionAttributeValues={":s" :
   "disconnected"},
                            ExpressionAttributeNames={"#st": "Status"},
                            ReturnValues="UPDATED_NEW"
                    )
                    self.twist = Twist()
```

```
                r = rospy.Rate(10)
                while not rospy.is_shutdown():
                    self.twist.angular.z = 0
                    self._cmd_pub.publish(self.twist)
                    rospy.loginfo('Rotating robot: %s', self.twist)
                    r.sleep()
                exit(1)
            time.sleep(2)

def __init__(self):
    self._cmd_pub = rospy.Publisher('/cmd_vel', Twist, queue_size=1)

    # Use the DynamoDB client update the status of Robot 1
    response=self.table.update_item(
        Key={
            'Robot': 2,
            'Type':'Follower'
        },
        UpdateExpression = "set #st = :s",
        ExpressionAttributeValues={":s" : "connected"},
        ExpressionAttributeNames={"#st": "Status"},
        ReturnValues="UPDATED_NEW"
    )
    thread=threading.Thread(target=self.disconnecter, args=())
    thread.start()


def rotate_forever(self):
    self.twist = Twist()
    r = rospy.Rate(10)

    while True:
        response=self.dynamodb_client.query(
            TableName=TABLE_NAME,
            KeyConditionExpression='Robot = :Robot',
                ExpressionAttributeValues={
                    ':Robot': {'N': str(2)}}
            )
        if response['Items'][0]['Status']['S']=='connected':
            self.twist.angular.z = 0.1
            self._cmd_pub.publish(self.twist)
            rospy.loginfo('Rotating robot: %s', self.twist)
            r.sleep()
        time.sleep(2)
```

```
    def stopper(self):
        response=self.table.update_item(
        Key={
            'Robot': 2,
            'Type':'Follower'
        },
        UpdateExpression = "set #st = :s",
        ExpressionAttributeValues={":s" : "disconnected"},
        ExpressionAttributeNames={"#st": "Status"},
        ReturnValues="UPDATED_NEW"
    )
        self.twist = Twist()

        r = rospy.Rate(10)
        while not rospy.is_shutdown():
            self.twist.angular.z = 0
            self._cmd_pub.publish(self.twist)
            rospy.loginfo('Rotating robot: %s', self.twist)
            r.sleep()

def main():
    rospy.init_node('rotate')

    def handler(*args):
        rotator.stopper()
        exit(1)

    try:
        rotator = Rotator()
        signal.signal(signal.SIGINT, handler)
        rotator.rotate_forever()
    except rospy.ROSInterruptException:
        pass

if __name__ == '__main__':
    main()
```

These robots were simulated in an empty world. A simulation output is reported in Figure 4.5.

Figure 4.5: Simulation output

4.4 REST API

The web page displaying the robots' status was created through the integration of AWS Lambda and Amazon API Gateway. Once the REST API was created using the latter service, a **Resource** with the GET **Method** was added to it. This allowed the API to invoke the desired Lambda function, reported below.

```
import json
import boto3
```

```
TABLE_NAME = 'Robot-Status'

# Creating the DynamoDB Client
dynamodb_client = boto3.client('dynamodb', region_name="eu-central-1")

def lambda_handler(event, context):
    response1 = dynamodb_client.query(
        TableName=TABLE_NAME,
        KeyConditionExpression='Robot = :Robot',
        ExpressionAttributeValues={
        ':Robot': {'N': str(1)}
        }
    )

    response2 = dynamodb_client.query(
        TableName=TABLE_NAME,
        KeyConditionExpression='Robot = :Robot',
        ExpressionAttributeValues={
        ':Robot': {'N': str(2)}
        }
    )

    response3 = dynamodb_client.query(
        TableName=TABLE_NAME,
        KeyConditionExpression='Robot = :Robot',
        ExpressionAttributeValues={
        ':Robot': {'N': str(3)}
        }
    )
    string = f"Robot 1 status is
   {response1['Items'][0]['Status']['S']}\nRobot 2 status is
   {response2['Items'][0]['Status']['S']}\nRobot 3 status is
   {response3['Items'][0]['Status']['S']}"
    return{
        'statusCode': 200,
        'body': string
    }
```

This code connects to the DynamoDB table *Robot-Status*, and retrieves the different status through the GetItem API of boto3. The output is then displayed into a web page, as depicted in Figure 4.6.

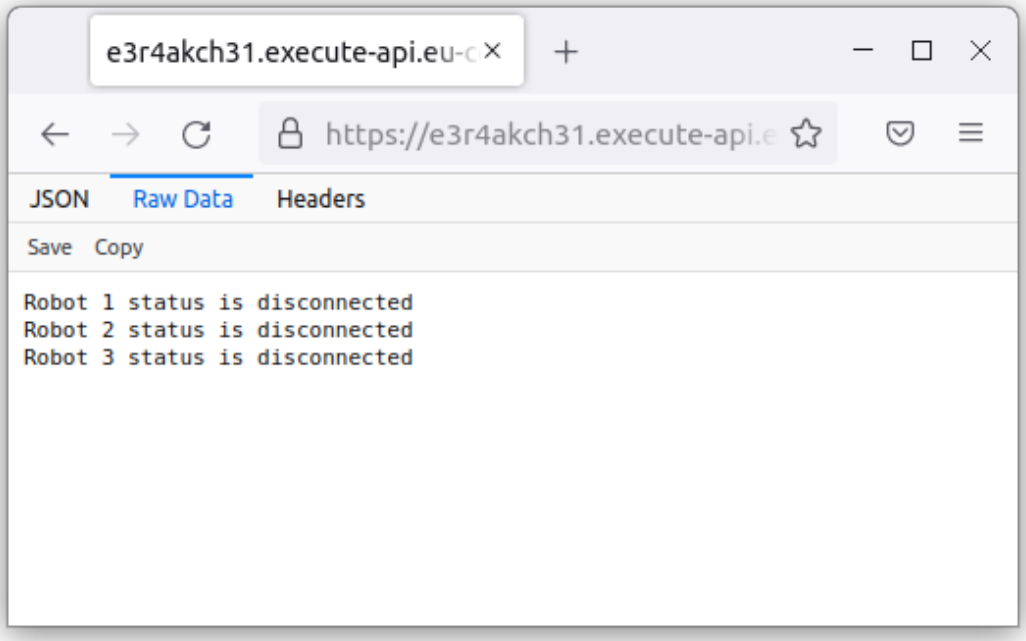

Figure 4.6: Default REST API

Chapter 5

Results

The proposed architecture has been validated through different tests:

- **Single robot simulation:** in which the robots were simulated one at a time.

- **Leader-Followers simulation:** during the which the leader was simulated together with the followers.

These trials have been done to properly evaluate the correct implementation of the fleet manager algorithm, and to check if the robots correctly update their own status.

Single robot simulation

The aim of this test was to verify that the followers could perform the rotation, and update their status to *connected*, if and only if the leader was being simulated, which instead can complete its tasks independently of the other devices. As expected, when the robot 1 was launched, it updated its status and began rotating, as depicted in Figure 5.1.

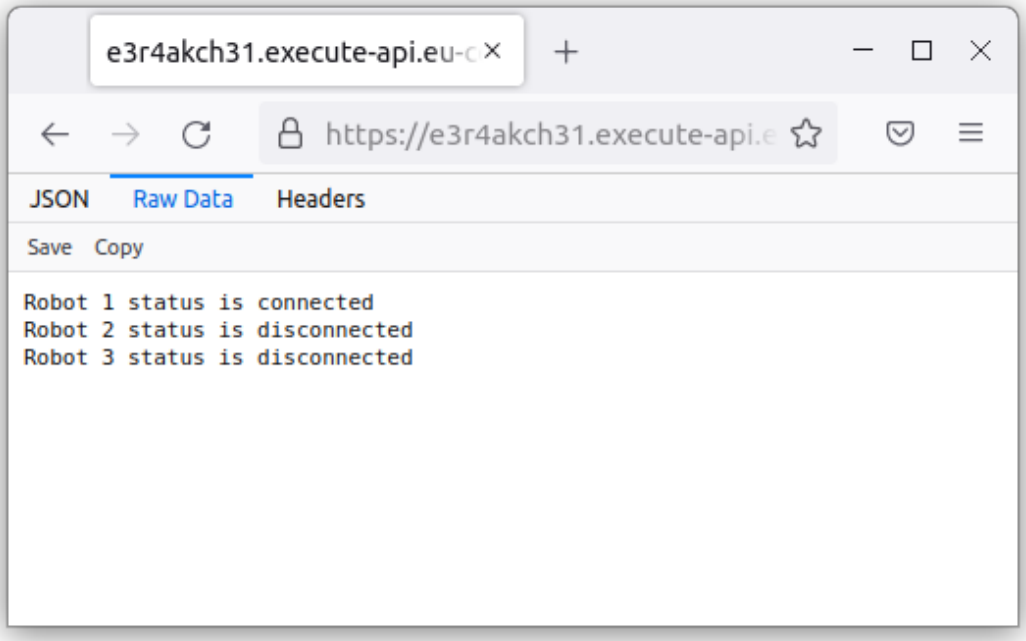

Figure 5.1: Leader simulation output

Once the leader simulation was stopped and a follower was launched, instead, it has been verified that this device did not perform the rotation. Therefore its status was update first to *connected* by the robot, then to *disconnecting* from the multi robot application, as reported in Figure 5.2 for the robot 2, and then to *disconnected* from the robot itself, reaching the default configuration depicted in Figure 4.6.

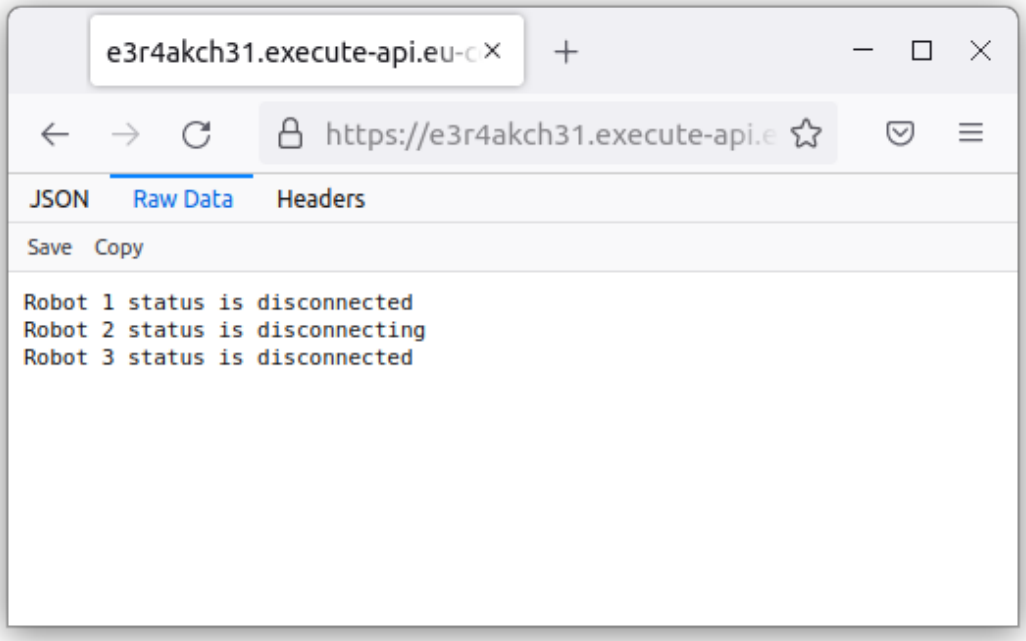

Figure 5.2: Single follower simulation output

Leader-Followers simulation

The goal of this simulation was to evaluate the last feature of the fleet manager algorithm, which forces the followers to stop once the leader is stopped. First of all the robots were simulated, and their status have been updated and displayed in a web page, as reported in Figure 5.3.

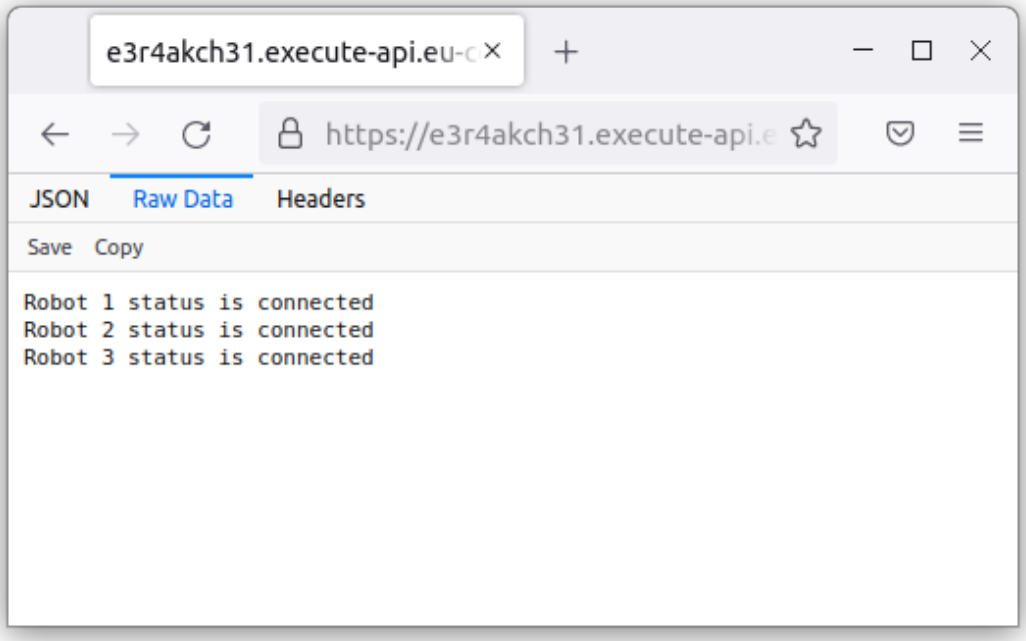

Figure 5.3: Leader Followers simulation

Afterwards the leader simulation was interrupted, and it was verified that the followers automatically stopped their simulation. The multi robot application updated their status to *disconnecting*, as reported in Figure 5.4, and the followers updated again their status to *disconnected*, reaching the default configuration.

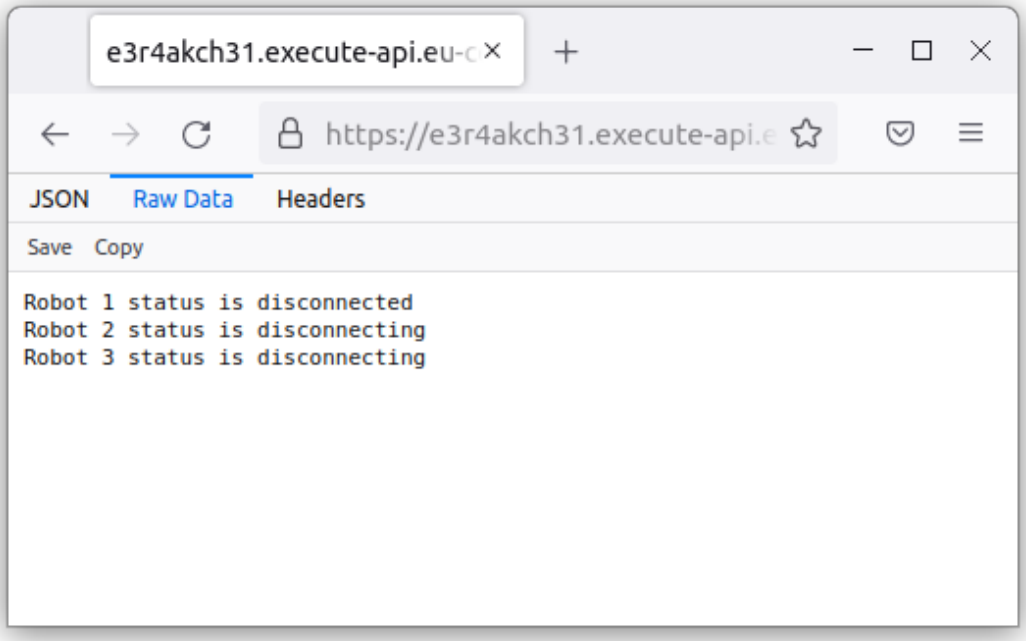

Figure 5.4: Leader disconnection

Chapter 6

Discussion and future developments

6.1 AWS Cloud architecture for physical robots

The cloud architecture presented in the previous chapter was designed to manage a fleet of simulated robots. Therefore, after the testing phase, the developed algorithms can be applied to physical devices. The schema depicted in Figure 6.1 represents an AWS cloud architecture to manage a fleet of real robots.

The main differences from the previous scheme regard the single robot application and the development of a dashboard for the end-user.

6.1.1 Single robot application

The single robot application can be developed using only one AWS RoboMaker instance per robot type, i.e. leader and follower. The robotic application can then be deployed to the physical devices using **AWS IoT Greengrass 2.0**. This service comprises two shared resources, named **Components** and **Developments**, to respectively import the application the user wants to deploy and register the core devices. The application can be defined through a recipe, that is a YAML or JSON file describing the component's details, dependencies, compatibility, and lifecycle. The physical robots instead can be registered to this service individually or as a group, and the developer can choose which component will be deployed to which deployment.

6.1.2 End-user Dashboard

Instead of setting up a REST API, as proposed in chapter 4, via AWS Lambda and Amazon API Gateway, AWS IoT Greengrass 2.0 can be easily integrated with **AWS IoT Device Management**. This service provides a dashboard through which the user can visualize the status of each deployment. It allows also to filter them on the basis of different parameters, such as the battery percentage and location. The developer can also select a trigger that activates to some user-defined conditions and IoT Device Management can automatically alert users via email once the specified conditions are not met.

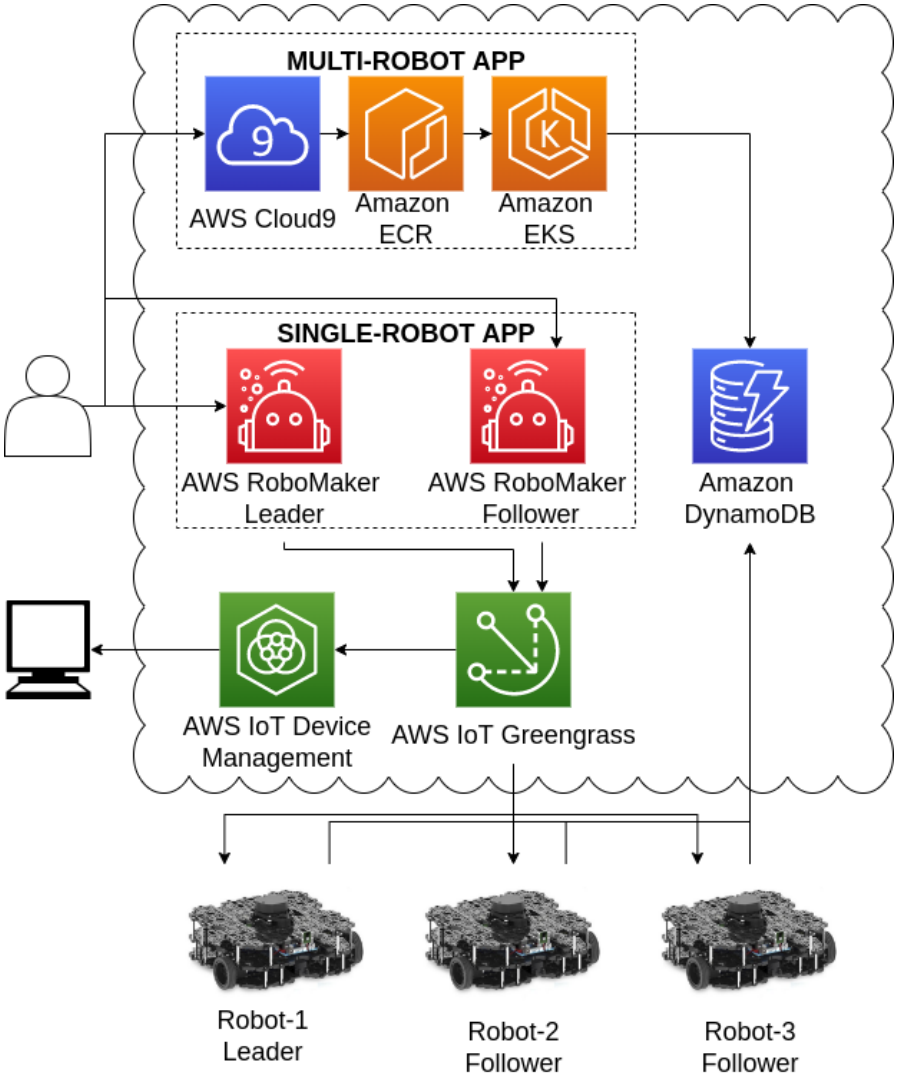

Figure 6.1: Cloud robotics architecture for physical devices

6.2 eProsima Fast DDS

In this thesis work the multi robot algorithm, that was deployed in a Kubernetes' Pod, was implemented through a Python script, and a NoSQL database acted as the bridge between this application and the simulated devices. However, for more complicated tasks, it may be necessary to carry out the fleet management software using ROS or ROS2 and to set a direct communication between the robots and the Kubernetes' Pod, through a Digital Data Service (DDS). Therefore a study has been made on the use of **eProsima Fast DDS** to connect ROS2 nodes deployed on a physical robot and others deployed in a Pod.

ROS2 was preferred to ROS because it is easier to containerize and because of its masterless nature. This last property helps the user since it is not required to deal with multi master communication problem.

eProsima Fast DDS is a free and open source middleware implementation providing both the Object Management Group (OMG) DDS 1.4 and the OMG Real-Time Publish-Subscribe (RTPS) 2.2 wire-protocol standards. This framework generates the Publisher/Subscriber code from the topic definition through an Interface Definition Language, allowing users to focus on the application development without minding about the networking.

In this example, depicted in Figure 6.2, a Kubernetes network and a local environment are set up to enable communication between a pair of ROS2 nodes, one sending messages from a Local Area Network (LAN) (talker) while the other (listener) receiving them in the cloud.

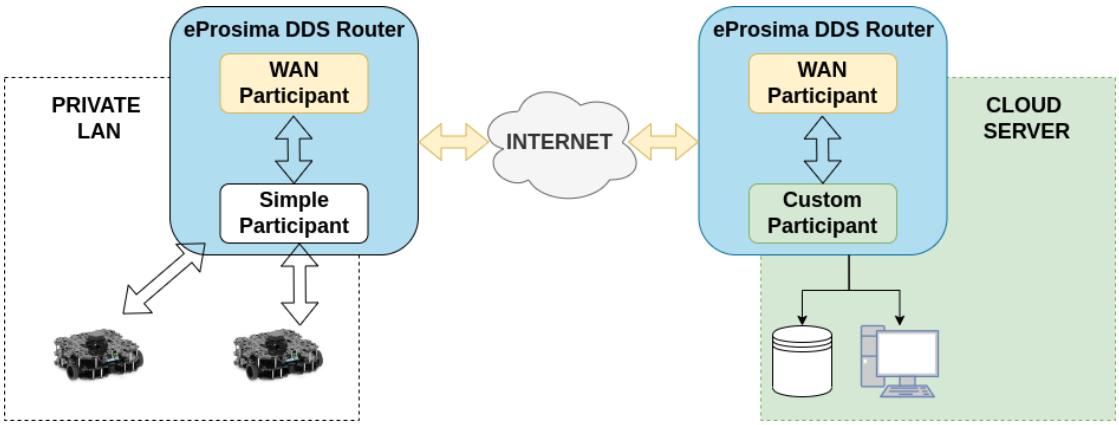

Figure 6.2: Robot - Kubernetes communication

6.2.1 Local setup

The local instance of the DDS router only requires the following two components:

- **Simple Participant:** which discovers all Participants deployed in its own local network in the same domain via multicast communication. It communicates with the ones that share publication or subscription topics.
- **WAN Participant:** which communicates with other Wide Area Network (WAN) Participants in different networks. This Participant acts as a bridge for every Participant working locally in the LAN and any other LAN having DDS with an active WAN Participant.

These two components acknowledge each other's existence through a Simple DDS discovery mechanism. The Simple Participant receives messages published by the ROS2 talker node and it forwards them to the WAN Participant.
Subsequently these messages are sent to another Participant in a Kubernetes' Pod to which it connects via WAN communication over UDP/IP.

6.2.2 Kubernetes setup

The cloud instance of the DDS Router consists on the following two Participants:

- **WAN Participant:** that receives the messages coming from the LAN via the UDP communication channel.
- **Local Discovery Server:** which forwards these messages to the ROS2 listener node, hosted in a third Pod.

In this instance the Local Discovery Server was preferred with respect to the Simple Participant because of the difficulty on enabling multicast communication in cloud environments.

Chapter 7

Conclusion

Cloud robotics is an emergent field of robotics that uses cloud technologies to improve robotic system's functionalities. Once connected to the cloud, these devices can take advantage of the powerful computation, storage, and communication resources offered by cloud computing. However not everything can be offloaded to the cloud because of latency constraints and to ensure local autonomy.

This thesis work provides an implementation of a cloud architecture to manage a fleet of simulated mobile robots, through different services provided by AWS. The robots were simulated with different AWS RoboMaker instances. The communication with the multi robotic application, stored in an Amazon EKS' Pod, was being possible through a NoSQL database provided by Amazon DynamoDB. Lastly it was demonstrated how to create and expose a REST API using AWS Lambda and Amazon API Gateway.

In order to validate the proposed solutions, two different types of test have been performed: the *Single robot simulation*, in which the robots were simulated one at a time; and the *Leader-Followers* one, where all the devices were tested simultaneously. These simulations have verified all the features of the multi robotic application. The former confirmed that the leader can perform its task independently of the followers, which instead cannot be simulated. The latter, instead, verified that the followers automatically stop their task once the leader simulation is stopped. Given the flexibility and high availability of the used technologies, the proposed solution can be applied also to an increasing number of devices and more sophisticated algorithms.

As further analysis, a cloud architecture was proposed also in case of availability of physical devices. Furthermore eProsima Fast DDS was described as a link between ROS2 nodes stored in a robot with others inside a Kubernetes' Pod.

Appendix A

Tools

A.1 ROS

ROS is an open-source, meta operating system to develop robotics software. It delivers services the user would expect from an OS like hardware abstraction, low-level device control and package management. ROS also comes up with tools and libraries to obtain, build, write and run code across different computers.
It is a middleware, responsible for handling the communication between programs in a distributed system, with a publisher-subscriber approach. The software codes are separated into packages, called nodes, which communicate with one other using one of the following tools:

- **Topics:** used for sending and receiving data streams between nodes. Every node declares in which topic publish its message and/or to which topic subscribe to.
- **Services:** useful to create a synchronous client/server communication between nodes. It is used to change settings in a robot or ask for a specific action.
- **Actions:** based on topics, they provide the developer with an asynchronous client/server architecture, where the client can send a request that takes a long time. The client can asynchronously monitor the state of the server, and cancel the request anytime.

Robotics code in ROS can be developed in many different languages like C++, Python, JavaScript, MATLAB and many others. A peculiarity of this tools is that ROS is language agnostic, meaning that the application can have nodes written in different programming languages, since the communication layer is below the language layer.

Bibliography

[1] J. Kuffner. «Cloud Enable Robots». In: *IEEE-RAS International Conference on Humanoid Robotics* (2010) (cit. on p. 3).

[2] Viraj Dawarka and Girish Bekaroo. «Building and evaluating cloud robotic systems: A systematic review». In: *Robotics and Computer-Integrated Manufacturing* 73 (2022), p. 102240. ISSN: 0736-5845. DOI: https://doi.org/10.1016/j.rcim.2021.102240. URL: https://www.sciencedirect.com/science/article/pii/S0736584521001216 (cit. on p. 4).

[3] Alessio Botta, Jonathan Cacace, Riccardo De Vivo, Bruno Siciliano, and Giorgio Ventre. «Networking for Cloud Robotics: The DewROS Platform and Its Application». In: *Journal of Sensor and Actuator Networks* 10.2 (2021). ISSN: 2224-2708. DOI: 10.3390/jsan10020034. URL: https://www.mdpi.com/2224-2708/10/2/34 (cit. on pp. 4, 5).

[4] Docker.com. «Sample Application | Docker». In: (). URL: https://docs.docker.com/get-started/02_our_app/ (cit. on p. 10).

[5] The New Stack. «What the Data Says about Kubernetes Deployment Patterns». In: (). URL: https://thenewstack.io/data-says-kubernetes-deployment-patterns/ (cit. on p. 13).

[6] statista. «Amazon Leads $200-Billion Cloud Market». In: (). URL: https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/ (cit. on p. 22).

[7] Gartner. «Browse Extreme Compute Reviews». In: (). URL: https://www.gartner.com/reviews/vendor/extreme-compute (cit. on p. 23).

[8] Gartner. «Gartner Report - 2021 Magic Quadrant for Cloud Infrastructure Platform Services». In: (). URL: https://aws.amazon.com/resources/analyst-reports/gartner-mq-cips-2021/?nc1=h_ls (cit. on p. 23).

[9] «Build and Simulate Robotics Applications in AWS Cloud9». In: (). URL: https://aws.amazon.com/blogs/robotics/robotics-development-in-aws-cloud9/ (cit. on p. 47).

[10] «Amazon EC2 AMI Locator». In: (). URL: https://cloud-images.ubuntu.com/locator/ec2/ (cit. on p. 47).

[11] SmartRobotWorks. «ROBOTIS TurtleBot3 Waffle Pi». In: (). URL: https://www.smartrobotworks.com/TurtleBot3-Waffle-Pi.asp (cit. on p. 48).