Degree Project in Computer Science
Second cycle, 30 credits

Dynamic container orchestration for a device-cloud continuum

CAMILO-ALFONSO RODRIGUEZ-GARZON
Dynamic container orchestration for a device-cloud continuum

CAMILO-ALFONSO RODRÍGUEZ-GARZÓN

Master’s Programme, ICT Innovation, 120 credits
Date: October 22, 2023

Supervisors: Feridun Tütüncüoglu (KTH), Yu Liu A (Ericsson), Aitor Hernandez Herranz (Ericsson)
Examiner: György Dán
School of Electrical Engineering and Computer Science
Host company: Ericsson AB
Abstract

Edge computing has emerged as a paradigm to support the growing demand for real-time processing of data generated at the edge of the network. As the devices at the edge are constrained, one of the challenges in the area is how to schedule workloads. The scheduling problem is difficult to tackle due to the multitude of sources from which variables originate, diverse algorithms and execution methods, and tasks involving information dissemination and action execution. This project aims to explore the problem and implement a system that simplifies the construction of a scheduler for the edge computing to reduce the cognitive load on developers that work on the area and focus their attention on their expertise area. To construct the solution, a literature review is conducted, a set of functional and non functional requirements are proposed, an implementation using a Kubernetes operator and Python application is performed, and an evaluation and validation of the solution against the requirements and an use case and test case are performed. The results demonstrate that the system generates customized instances capable of receiving any number of inputs, outsources the execution of the logic and interacts with different outputs. This allows developers to rapidly deploy instances for their own needs, focusing on their domain of expertise.

Keywords

Edge computing, kubernetes operator, dynamic scheduling
**Sammanfattning**


**Nyckelord**

Edge computing, Kubernetes-operatör, dynamisk schemaläggning
Acknowledgments

I would like to thank those who have made it possible to finish this thesis project:

Aitor and Yu (Ericsson supervisors), thank you for the opportunity, your time, the discussions and advice, and in general for the supervision during the project.

Feridun (KTH supervisor) and György (KTH examiner). Thank you for your proofreading and project and document guidance.

Eileen. Thanks for your patience, wise words and advice. We made it!

Pa, Ma, Mapila and little Lia. Despite being 9672 km away, I always have felt you next to me. Thank you so much for everything.

A todos, muchas gracias.

Stockholm, October 2023
Camilo-Alfonso Rodríguez-Garzón
iv | Acknowledgments
# Contents

## 1 Introduction
- 1.1 Background ......................................................... 1
- 1.2 Problem .......................................................... 2
- 1.3 Purpose ............................................................ 3
- 1.4 Objective .......................................................... 3
- 1.5 Research Methodology ............................................. 3
- 1.6 Ethics and sustainability ......................................... 4
- 1.7 Delimitations ....................................................... 4
- 1.8 Structure of the thesis ............................................ 5

## 2 Background
- 2.1 Cloud, Edge and Fog Computing and the device-cloud continuum ......................................................... 6
- 2.2 Containerization .................................................... 8
- 2.3 Container orchestration .......................................... 9
  - 2.3.1 Kubernetes ..................................................... 9
  - 2.3.2 Kubernetes (K8S) Scheduling .............................. 11
    - 2.3.2.1 Manual Assignment .................................. 12
    - 2.3.2.2 Descheduler ........................................... 12
  - 2.3.3 K8S extensions .............................................. 12
- 2.4 Functional and non functional requirements ................ 16
- 2.5 Related work ..................................................... 16
  - 2.5.1 Gaps in the literature ................................... 20

## 3 Design of the Solution
- 3.1 Requirements .................................................... 23
  - 3.1.1 Non-functional ............................................ 23
  - 3.1.2 Functional .................................................. 24
- 3.2 Design ............................................................ 25
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.1</td>
<td>Operator</td>
<td>25</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Optimizer</td>
<td>29</td>
</tr>
<tr>
<td>4</td>
<td>Implementation</td>
<td>31</td>
</tr>
<tr>
<td>4.1</td>
<td>Steps for end users</td>
<td>33</td>
</tr>
<tr>
<td>5</td>
<td>Evaluation and validation</td>
<td>35</td>
</tr>
<tr>
<td>5.1</td>
<td>Evaluation of requirements</td>
<td>35</td>
</tr>
<tr>
<td>5.2</td>
<td>Environment Configuration</td>
<td>38</td>
</tr>
<tr>
<td>5.3</td>
<td>Use Case</td>
<td>38</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Design</td>
<td>38</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Scenarios</td>
<td>41</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Results</td>
<td>43</td>
</tr>
<tr>
<td>5.4</td>
<td>Test Case</td>
<td>44</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Design</td>
<td>44</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Results</td>
<td>44</td>
</tr>
<tr>
<td>5.5</td>
<td>Discussion</td>
<td>45</td>
</tr>
<tr>
<td>6</td>
<td>Conclusions and Future work</td>
<td>48</td>
</tr>
<tr>
<td>6.1</td>
<td>Conclusions</td>
<td>48</td>
</tr>
<tr>
<td>6.2</td>
<td>Future work</td>
<td>49</td>
</tr>
<tr>
<td>References</td>
<td></td>
<td>50</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Cloud, Edge, and Fog computing and its importance in the device-cloud continuum. ............................................... 7
2.2 Kubernetes Architecture ......................................................... 10
2.3 Taints, tolerations and node affinity .................................................. 13
2.4 Example of a Custom Resource Definition (CRD) and its instantiation .............................................. 14
2.5 Creation of a CRD and its instantiation. ............................................. 15
2.6 Graphical summary of the systematic literature review .................................................... 22
3.1 Design of the solution ............................................................... 25
3.2 Kubernetes objects of the Optimizer .................................................. 26
3.3 Sequence diagram of the CRD object creation in K8S ........................................ 27
3.4 Design of the Optimizer ............................................................... 29
3.5 Kubernetes objects of the Optimizer .................................................. 30
4.1 Output log of a running Operator ................................................... 33
5.1 Used devices ............................................................................... 39
5.2 Architecture of the application case study ............................................... 40
5.3 Deployment scenarios. ................................................................. 42
5.4 Graphical comparison of the three metrics (Mean Average Precision (mAP), mAP runtime and latency) in scenarios 1, 2 and 3. ................................................................. 43
5.5 Migration on realtime ................................................................. 45
List of Figures
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Namespaces enabling container isolation.</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Used parameters for the systematic literature review.</td>
<td>17</td>
</tr>
<tr>
<td>2.3</td>
<td>Summary of articles.</td>
<td>19</td>
</tr>
<tr>
<td>5.1</td>
<td>Justification of Non-Functional Requirement (NFR).</td>
<td>37</td>
</tr>
<tr>
<td>5.2</td>
<td>Functional requirements.</td>
<td>38</td>
</tr>
<tr>
<td>5.3</td>
<td>Averaged results for the 3 metrics in every scenario.</td>
<td>43</td>
</tr>
</tbody>
</table>
List of Tables
List of Algorithms

1  Operator’s reconcile algorithm  . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
Listings

2.1 YAML Ain’t Markup Language (YAML) file with the definition of a two-pod deployment of an open source web server (nginx) ........................................ 11
4.1 YAML file example with the CRD of the Optimizer kind. . . . 31
# List of acronyms and abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACM</td>
<td>Association for Computing Machinery</td>
</tr>
<tr>
<td>AP</td>
<td>Average Precision</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>cgroup</td>
<td>Control Group</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CRD</td>
<td>Custom Resource Definition</td>
</tr>
<tr>
<td>FR</td>
<td>Functional Requirement</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute of Electrical and Electronics Engineers</td>
</tr>
<tr>
<td>IO</td>
<td>Input/Output</td>
</tr>
<tr>
<td>IoT</td>
<td>Internet of Things</td>
</tr>
<tr>
<td>IPC</td>
<td>Interprocess Communication</td>
</tr>
<tr>
<td>IT</td>
<td>Information Technology</td>
</tr>
<tr>
<td>K8S</td>
<td>Kubernetes</td>
</tr>
<tr>
<td>mAP</td>
<td>Mean Average Precision</td>
</tr>
<tr>
<td>NFR</td>
<td>Non-Functional Requirement</td>
</tr>
<tr>
<td>NIST</td>
<td>National Institute of Standards and Technology</td>
</tr>
<tr>
<td>OD</td>
<td>Object Detection</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PID</td>
<td>Process Identification Number</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>RDMA</td>
<td>Remote Direct Memory Access</td>
</tr>
<tr>
<td>ROS</td>
<td>Robot Operating System</td>
</tr>
<tr>
<td>Acronym</td>
<td>Abbreviation</td>
</tr>
<tr>
<td>---------</td>
<td>--------------</td>
</tr>
<tr>
<td>SDK</td>
<td>Software Development Kit</td>
</tr>
<tr>
<td>UTS</td>
<td>Unix Time Sharing</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>YAML</td>
<td>YAML Ain’t Markup Language</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This chapter describes the specific problem that this project addresses, details the context of the problem and the goals of this project, and outlines the structure of the thesis.

1.1 Background

A current trend in the Information Technology (IT) space involves providing processing capacity, memory and storage to common objects such as refrigerators, light bulbs or cars. This trend has placed a pressure of enabling faster and more reliable services. As a consequence, the applications that run on top of them have specific Quality of Service (QoS) requirements and demand a proper allocation and coordination of components, to ensure certain performance guarantees.

The deployment of the applications mentioned before has changed thoroughly with the introduction of cloud native technologies. Initially, applications operated on physical servers with Operating Systems (OSes). Subsequently, as the physical servers increased their capacity, an abstraction layer, namely the hypervisor, was created between the OS and the physical layer to abstract the hardware and enable several OSes or Virtual Machines (VMs) to host different applications in parallel. Finally, containerization technology emerged as a lightweight virtualization alternative and has proven to be more advantageous to VMs[1].

The minimum unit in the containerization technology is container. In small deployments, administering containers is simple but as the number of deployed containers grows, the administration of those containers becomes
tedious. Due to this fact container administration tools have been created. The de-facto technology of achieving this is Kubernetes (K8S). It allows to orchestrate, i.e., to coordinate, the execution of containers and provide the possibility of having more objects that support them to build highly scalable and resilient applications.

Despite being highly customizable, the K8S scheduling mechanism cannot fully meet the requirements of specific applications and therefore it still is an open issue [2, 3, 4]. The importance of scheduling considering external metrics and requirements affects how well the application and infrastructure can perform. For example, in a context where applications are containerized and running in heterogeneous environments, the inner logic of an application could request the consideration of external metrics and/or the interaction with external Application Programming Interfaces (APIs) to modify characteristics that affect the external environment of the application such as the network bandwidth. This project seeks to devise a framework in K8S that enables developers to ease the construction of schedulers based on different inputs, an external logic, and a way to generate outputs.

1.2 Problem

Due to its underlying infrastructure, the device-cloud continuum* poses a new set of challenges for cloud providers. One of them is the proper coordination of workloads and infrastructure components during the whole execution of the deployed applications. One notable example of the aforementioned challenge lies in K8S. By default, K8S lacks of mechanisms for workload rescheduling, external metric consideration for scheduling, runtime application characteristic adjustments, and the ability to execute third-party scheduling logic.. The research question of the project is:

How to dynamically orchestrate the workloads in a container orchestration platform that is across the device-cloud continuum, by taking inputs from predefined application and system metrics and manipulating the underlying configuration of one or more entities (application, orchestrator, platform, infrastructure, network, or other) to fulfill certain QoS?

*As shown by [5] there is a multitude of definitions of the concept. The one used in this project comes from the same reference and it is defined as “an extension of the traditional Cloud towards multiple entities (e.g., Edge, Fog, Internet of Things (IoT)) that provide analysis, processing, storage, and data generation capabilities.”
In order to answer the main question, the following two research questions are proposed:

- **RQ1** - What are the requirements for developing a framework that is able to receive inputs, a logic, and outputs to ease the construction of a scheduler for the device-cloud continuum?
- **RQ2** - How can a software implementation be devised to fulfill the identified set of requirements?

### 1.3 Purpose

Taking advantage of the extensibility of Kubernetes, the purpose of this project is to extend the functionality of Kubernetes for a device-cloud continuum scenario. The expected value comes from constructing a configurable framework for engineers and developers that work with container orchestration in IoT. The framework is envisioned to not only perform scheduling and rescheduling of containerized workloads, but also handle the characteristics of the deployed containers and applications.

### 1.4 Objective

The main objective of this thesis is to build a framework for Kubernetes to reschedule workloads and modify features from the application, platform, network, or infrastructure. To achieve this, three key objectives are proposed:

1. **Subgoal 1** – Conduct a comprehensive review of the state-of-the-art of container scheduler mechanisms and assess their technology readiness levels, strengths, and weaknesses.

2. **Subgoal 2** – Evaluate the most feasible approach for dynamic workload orchestration based on Ericsson’s existing testbed.

3. **Subgoal 3** – Validate the proposed solution by implementing it with a selected application and assessing its performance and efficacy.

### 1.5 Research Methodology

This thesis project consists of three main phases targeted to fulfill the main objective of it.
• **Literature review.** It is a research method that identifies and classifies data into relevant research [7]. In this thesis project it is used to get the state-of-the-art and identify the gaps and requirements in the existing solutions, if any.

• **Design and implementation.** This phase involves defining the requirements, designing the solution and implementing a solution that fits the design and requirements.

• **Evaluation and validation.** The final step consists of an assessment of the implemented solution in comparison to its current status. This step includes the validation of requirements and a use and test case to assess the validation.

### 1.6 Ethics and sustainability

According to the International Energy Agency, in 2020 data centers and data transmission networks accounted for the 0.6% of total greenhouse gas emissions in the world [8]. To keep global warming to no more than 1.5°C, those emissions should be reduce by 45% by 2030 and reach net zero by 2050 [9]. Given this project’s context, it is not uncommon to find schedulers aiming to decrease energy consumption. For instance, the authors of [4] and [10] consider energy factors, addressing both environmental concerns and practical device limitations in the device-cloud continuum, offering a dual benefit. However, the goals of this project do not encompass any direct energy optimization effort aimed at lowering greenhouse emissions. Nevertheless, the system developed in this project leaves the door open for potential implementation of schedulers that could consider energy metrics or interact with energy matters, just like the journal articles presented before.

### 1.7 Delimitations

This thesis project focuses on a framework that will enable developers to customize metrics, inputs, outputs, and model to ease the deployment of dynamic schedulers while ensuring the QoS levels for a containerized platform. Specifically, its boundaries are described as follows:

• The algorithm that decides which outputs are generated and how the system is modified is out of the scope of this project. Developers will be
responsible for the implementation of the algorithm, and the framework facilitates this capability as one of the main features.

- Despite several technologies exist for container creation and orchestration, the framework proposed in this project is limited to Docker and Kubernetes.
- The intended implementation of this project is not a final product but rather serves as a proof of concept.

1.8 Structure of the thesis

The outline of this document is as follows. Chapter 2 presents relevant background information about the main topics, including the device-cloud continuum, containerization, container Orchestration, and related works. Chapter 3 discusses the design of the solution. Chapter 4 describes how the implementation of the solution was done. Chapter 5.1 shows the results of the implementation. Chapter 6 discusses the conclusions and future work of this project.
Chapter 2

Background

To get a better understanding of the concepts and interactions of the components proposed in this project, it is key to gain an understanding of some basic concepts. This chapter describes the principles behind the device-cloud continuum along with the role of containerization in it, and its challenges and possibilities.

2.1 Cloud, Edge and Fog Computing and the device-cloud continuum

From having static, on-premise and limited resources, cloud computing opened the door to dynamically and rapidly scaling resources based on current and future demands [11]. According to National Institute of Standards and Technology (NIST) cloud computing consists of four characteristics [12]: First, On-demand self-service enables consumers to automatically and unilaterally provision computing resources. Second, Broad network access allows consumers to access the provisioned resources over the network following standard practices. Third, resource pooling is the multitenant architecture offering of the cloud provider, where the resources are shared unbeknownst to the customers. Finally, measured service to the metering of the consumption of services for its later payment.

As embedded systems become more powerful, popular, and widespread under the umbrella of IoT, the limits of where to execute computation have been eliminated. Services enabled by cloud native technologies are no longer limited to servers running in the cloud, but have evolved to include intermediate and even end devices capable of using spare resources to run
Cloud computing workloads [11]. Figure 2.1 is a depiction of the locations and relationships between the Edge, Fog and Cloud Computing and the device-cloud continuum. There are several characteristics that differentiate them such as the number, heterogeneity or distribution of the devices. Edge devices are built to interface with end users and the physical world as their application fields range from military devices such as drones or satellites, to Smart Grid applications such as smart metering devices for solar panels, and beyond, encompassing scenarios such as smart cities, entertainment or eHealth.

Figure 2.1: Cloud, Edge, and Fog computing and its importance in the device-cloud continuum. Own elaboration based on [14, 15].

The characteristics of edge computing differ from those of the cloud. For example, edge devices are often energy constrained, vastly differ in hardware specs, and provide sufficient but limited amount of resources in terms of Central Processing Unit (CPU), memory, storage, and sensors. By 2022, more than 13 billion of devices were connected to the Internet and a twofold is expected circa 2028 [16]. Another characteristic is the diversity the scenarios in which edge computing devices can be deployed. Military, e-health, smart city and even entertainment scenarios require extreme amounts of processing.

*The world health organization defines eHealth as the cost-effective and secure use of information and communications technologies in support of health and health-related fields [13]
in almost real time with low response times[17], which makes the QoS, in terms of low latency and high bandwidth, a crucial aspect to consider [18].

2.2 Containerization

One of the methods for sharing computing resources is virtualization [18]. The idea behind it is to abstract the physical hardware by creating a pool of resources, such as CPU, memory or disk to create isolated instances that make use of the resources. An example of an isolated instance is a VM. Nonetheless, VMs come at a cost of high resource overheads [18]. A lightweight alternative of virtualization is containerization. A container is an isolated executable unit of software that contains the application code along with the required libraries and dependencies [19]. As opposed to VMs, containers make use of several characteristics of the kernel of the OS where it is executed, making them lightweight. Because the executable contains all the libraries and dependencies, containers are also portable and platform-independent. The characteristics of portability and lightweight make containers energy, resource, and storage efficient and exceedingly quick during boot up, which make containers a good option to be used in the device-cloud continuum [15, 20, 21].

To gain an in-depth understanding of how containers function as cohesive entities, it is imperative to comprehend two fundamental concepts inherent to the Linux kernel, which facilitate the isolation and autonomous behavior of containers:

• **Control Groups (cgroups).** To control the consumption of processes in the Linux kernel, cgroups allow to set limits on resources for processes. According to the official kernel documentation [22], a cgroup is a “a mechanism to organize processes hierarchically and distribute system resources along the hierarchy in a controlled and configurable manner”. cgroups offer resource limiting, prioritization, accounting and control over a number of characteristics: CPU, Memory, Input/Output (IO), Process Identification Number (PID), Device, Remote Direct Memory Access (RDMA), miscellaneous and others. Every container instance is assigned a unique cgroup, therefore all the processes that are part of the container are resource controlled by the same cgroup.

• **Namespace.** A namespace is an abstraction in the Linux Kernel that makes a process running within a namespace to have its own isolated
instance of the global resource [23]. The group of namespaces that form a container consists of eight different namespaces [24], which are described in Table 2.1, making a container an isolated instance.

<table>
<thead>
<tr>
<th>Namespace</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mount</td>
<td>It handles the filesystem of the container</td>
</tr>
<tr>
<td>Unix Time Sharing (UTS)</td>
<td>It gives the container its own host and domain names</td>
</tr>
<tr>
<td>Interprocess Communication (IPC)</td>
<td>It isolates the message queue system of the container</td>
</tr>
<tr>
<td>PID</td>
<td>It separates the processes running in the container</td>
</tr>
<tr>
<td>Network</td>
<td>It allows the container to have its own network devices</td>
</tr>
<tr>
<td>User</td>
<td>It isolates users and groups from the container</td>
</tr>
<tr>
<td>cgroup</td>
<td>It hides the identity of the cgroup of which the process is a member</td>
</tr>
<tr>
<td>Time</td>
<td>It allows a container to have its own unique clock offsets</td>
</tr>
</tbody>
</table>

Table 2.1: Namespaces enabling container isolation.

2.3 Container orchestration

Every container is only a part of a more complex system. In order to deploy resilient, distributed, and scalable applications, a virtual system that manages containers and its underlying services are required. For that purpose, a family of container orchestration platforms exist, which include products like Kubernetes*, SwarmKit† or Rancher‡. Some of the key capabilities of orchestrators include scalability, fault-tolerance and availability, resource management and scheduling, and networking [25, 26, 15].

2.3.1 Kubernetes

One of the most used container orchestration technologies is Kubernetes. Figure 2.2 displays the Kubernetes architecture. The core component, the API server, exposes an Hypertext Transfer Protocol (HTTP) API that can be consumed using a client, e.g., kubectl§. This API allows developers to manipulate the state of the different objects in Kubernetes (e.g., pods, namespaces or secrets) that are stored in the etcd database. To achieve the state defined in the database, the paradigm used in Kubernetes consists in control loops (controllers) that watch the state of the cluster and perform different actions to attain the desired state [27].

*Available at https://github.com/kubernetes/kubernetes
†Previously known as Docker swarm. Available at https://github.com/moby/swarmkit
‡Available at https://github.com/rancher/rancher
§Available at https://github.com/kubernetes/kubectl
On the worker node(s) side three components are essential. The first one, *kubelet*, is the controller in charge of keeping the communication with the Master node and managing the containers on the work node. The *container runtime*, is the platform in charge of running the container as such. Finally, the Kubernetes Service Proxy, or *kubeproxy*, manages connections from the end users to the pods.

![Kubernetes Architecture](image)

Figure 2.2: Kubernetes Architecture. Own elaboration based on [28] and [29, Figure 1.9].

One of the K8S controllers is the Scheduler Controller [30]. Because the process of allocating a pod to a node to be run depends on many factors (such as the availability of ports, *CPU* or memory, or specific requirements of the pod) when a pod is created it does not have a node assigned. The process of listing, filtering and scoring nodes is performed by the Scheduler Controller. Another controller, the Controller Manager [31], looks for specific objects that are part of an application’s deployment such as Deployments or Namespaces and perform actions to maintain the state specified.

The way to deploy applications in Kubernetes is by the use of objects, classified in K8S resource types named *kinds*. The most important *kinds* are namespaces, deployments, services, ingresses, configmaps, volumes and secrets. The specification of all of them can be found in the official reference documentation*. The first step to instantiate any object consists of defining

*For version 1.22 it is available in [https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.22/](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.22/)
a YAML Ain’t Markup Language (YAML) file with the specification of the object. Code 2.1 shows an example of a deployment object. Once applied, the API stores the desired state in the database and then the controllers create the necessary components on the worker nodes to fulfill the desired state.

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 80
---
```

Listing 2.1: YAML file with the definition of a two-pod deployment of an open source web server (nginx)

### 2.3.2 K8S Scheduling

In Kubernetes, scheduling makes reference to the process of matching nodes to pods [30]. In the default scenario, the process starts when kube-scheduler, which is the default scheduling controller, notices an unassigned pod. The controller maintains a queue of pods called `podQueue` that continuously listens to the API server for new pods. Once it has found an unbounded pod it is first added to the queue. Another process is watching over the queue and once it finds a pod on it, two main cycles are triggered: the Scheduling Cycle and the Binding Cycle [32]. The first cycle consists of two sub-steps. The first one filters nodes that comply with certain requirements established by the pod definition. For example, if a node can provide the requirements asked by the pod such as CPU, memory and availability of port(s), the node is pre-selected. The second sub-step ranks the pre-filtered nodes based on how well the nodes
would perform. In the case of a tie, the node is selected randomly. The second big step, the Binding cycle, applies that decision to the cluster.

### 2.3.2.1 Manual Assignment

One way to force a pod to run in a specific node is by the use of taints, tolerations and node affinity. Taints and tolerations work together to mark both the nodes and pods with corresponding labels for pods to tolerate certain nodes. For example, setting a blue taint on a node will only let blue labeled pods to tolerate and start running on that node (Figure 2.3a). But it will not limit tainted blue pods to run on untainted nodes, as they do not block them, so it is a solution that attracts pods to some nodes. The approximation of node affinity is similar yet does the opposite restricting pods to run on some nodes. For example, setting a blue label on a node will make blue pods to only run on those nodes. However, a restriction of this model is that it will not limit any unlabeled pod to run on a blue node (Figure 2.3b). Consequently the combined use of taints & tolerations and node affinity (Figure 2.3c) permits the exclusive assignment of pods to nodes ensuring that (i) specific pods will not run on untainted nodes and (ii) unlabeled pods will not run on specific nodes.

### 2.3.2.2 Descheduler

The descheduler project∗ is an open source implementation for Kubernetes that extends the default scheduler. As opposed to the scheduler, which assigns nodes to pods, the mechanism behind the descheduler consists only of finding and evicting pods based on a set of preestablished rules set on a policy. Typical scenarios when the project can be used include moments when nodes are under/over utilized, removed, or provisioned; or when the status of the taints, nodes or labels change.

### 2.3.3 K8S extensions

Because of the extensibility of Kubernetes, it also offers the possibility to create new objects and their APIs [6]. In order to achieve it, the first part consists of defining and applying a Custom Resource Definition (CRD) object to instantiate objects and interact with them. For example, the left box in Figure 2.4 shows an object of CustomResourceDefinition kind that after being applied (Figure 2.5a) will create a new object kind, pizza, in the Kubernetes

∗Available at https://github.com/kubernetes-sigs/descheduler
(a) Taints and toleration

(b) Node affinity

(c) Taints, tolerations and node affinity

Figure 2.3: Manual assignment in Kubernetes using taints, tolerations and node affinity. Own elaboration.

API with its specification (Figure 2.5b). An instantiation of this object, shown in the right box of Figure 2.4 shows the kind of the object, pizza, along with the previously defined specification. The application of this definition will create an object of kind pizza with some characteristics, in the Kubernetes API (Figure 2.5c).

A closer look shows that the CRD and the possibility of creating instances will not do much more than creating a structured object in a database. To insert logic and the possibility of creating native Kubernetes objects, the creation of an application controller, a.k.a. operator, is required. Operators are control loops that make use of custom resources to manage applications and their components [33]. They are applications that contact the Kubernetes API to watch over specific events that are generated from specific CRDs. They can be written in any language, but a preference exists over those languages that already have Kubernetes client libraries such as Go, Python or Java [34]. The creation of an operator is not as trivial as the CRD as there are different considerations such as role based access controls, monitoring or deployment that tend to slow the creation of one. To ease those problems, frameworks such
as kubebuilder* or Operator-Software Development Kit (SDK)† have been created.

Figure 2.4: Example of a CRD and its instantiation. On the left, the Pizza CRD file configuration. On the right, a pizza file object instantiation. The relationship between the two is displayed. Own elaboration. Based on [35].

---

*Available at https://github.com/kubernetes-sigs/kubebuilder
†Available at https://github.com/operator-framework/operator-sdk
(a) Application of pizza CRD.

(b) Description of pizza kind.

(c) Apply, get and describe an object instance of the pizza CRD named meatsp

Figure 2.5: Example of the creation of a CRD and its instantiation. Reproduced from [35].
2.4 Functional and non functional requirements

To facilitate the construction of a software, a developer must know the intended software functionality. Software engineering undertakes this process of formulating the planned functionality of a software, thus facilitating the extraction of value. The specific concept designed to encapsulate the manner in which a software behaves is formally termed a requirement. The definition of a requirement has been stated in various manners by different scholars. Notably, in the international standard ISO/IEC/IEEE 24765:2017 [36], which establishes a common vocabulary for engineering work, the concept of requirement is defined using four distinct definitions. Nonetheless, the classification of requirements into **Functional Requirements (FRs)** and **Non-Functional Requirements (NFRs)** has shed light on their fundamental nature.

**FRs** address the question of the software’s intended actions, whereas **NFRs** answers the question of how the system executes its functions [37]. This comparison creates a clear demarcation between the operational behaviour of software and its quality attributes. **FRs** may be constructed as the tangible outcomes that a software shall produce while **NFRs** relate on the mechanisms governing the realization of those outcomes [36]. Informally put, **NFRs** can be mostly related with characteristics often denoted as “-ilities” or “-ities” such as usability or integrity [38]. Consequently, it is conceivable for two distinct software products to yield identical results; however, differences in user experience, performance or cost can arise due to disparities in their respective **NFRs**.

2.5 Related work

A literature review was performed in order to conduct a comprehensive review of the state-of-the-art of container scheduler mechanisms. The used parameters are shown in Table 2.2. The searching process in each database is different; in Scopus, the search was done using the keyword **TITLE-ABS-KEY** which represents a combined field that searches in abstracts, keywords and document titles. In both **Institute of Electrical and Electronics Engineers (IEEE)** and **Association for Computing Machinery (ACM)** databases the **Author Keyword(s)** parameter was used, which is self-explanatory. Figure 2.6 shows the PRISMA flow diagram [39], which describes the process of
identification, screening, and inclusion or exclusion of the material extracted from the databases. As shown in the same Figure, a total of 23 articles were finally selected to be included.

Table 2.3 summarizes the results of the analysis of each of the 23 articles. For each article, a short description of the objective, methodology and results is presented. Inspired by [2, 3] five taxonomies are created to analyze every selected article: Algorithm, Application type, Setup, Container orchestration used and Metrics.

<table>
<thead>
<tr>
<th>Inclusion criteria</th>
<th>Characteristic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Research article</td>
</tr>
<tr>
<td>Time</td>
<td>2018-2023</td>
</tr>
<tr>
<td>Databases</td>
<td>Scopus, IEEE, ACM</td>
</tr>
<tr>
<td>Language</td>
<td>English</td>
</tr>
</tbody>
</table>

Table 2.2: Used parameters for the systematic literature review.

From the results it is possible to infer several aspects. Of the total 23 articles, 5 articles [2, 3, 40, 41, 42] recompile information from the literature to analyze the state of the art and the 18 remaining perform experiments, simulations, or specific descriptions, which shows interest on the topic. None of the articles exemplify scheduling in the robotic applications domain. Simulations and experiments are performed, each one with its own pros and cons.

There is a body of work that deals with application placement but does not make use of container orchestration. [43] acknowledges how orchestration tools such as K8S provide minimal ability to influence the placement and utilization of microservices* part of an application. Hence, they create an adaptation mechanism to manage the placement of the microservices, taking into account affinity, and most importantly, resource usage history. The authors in [44] call for a reexamination of service placement mechanisms in Edge Computing to take into consideration the interdependencies within

---

*A microservice is defined as a decoupled and autonomic software that has a specific functionality and it is context-bounded [43].
<table>
<thead>
<tr>
<th>Document Title</th>
<th>Authors</th>
<th>Publication Year</th>
<th>Objective</th>
<th>Methodology</th>
<th>Results</th>
<th>Taxonomy</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CEC: A Containerized Edge Computing Framework for Dynamic Resource Provisioning</strong></td>
<td>Hu, S., Shi, W., Li, G.</td>
<td>2022</td>
<td>To respond to varying workload, a task prediction algorithm is created and</td>
<td>A physical testbed (1 Macbook plus 4 Raspberry Pi) is used to compare with and without the scheduling algorithm</td>
<td>The implementation of the algorithm results in lower service latency and higher resource utilization</td>
<td>Both simulation and physical system experiments are executed.</td>
</tr>
<tr>
<td><strong>COSCO: Container Orchestration Using Co-Simulation and Gradient Based Optimization for Fog ComputingEnvironments</strong></td>
<td>S. Tuli, S. R. Poujara, S. N. Sirimaru, G. Casale, N. R. Jennings</td>
<td>2022</td>
<td>A back-propagation gradient algorithm, digital twin model framework and Quality of Service simulation are created to deal with volatile environments and low scheduling overheads.</td>
<td>A full panorma (algorithm, framework, simulation testbed) is run using a cloud provider.</td>
<td>The model reduce energy consumption, response-time, SLO violations and scheduling times compare to state-of-the-art schedulers</td>
<td>Gradient based-back-propagation algorithm, Testbed using cloud, Own implementation, Energy consumption, response time, execution time, SLO Violations, Fairness, Migration Time, Scheduling Time, Wait Time</td>
</tr>
<tr>
<td><strong>Custom Scheduling in Kubernetes: A Survey on Common Problems and Solution Approaches</strong></td>
<td>Rejith, Z., Chamaramana, J.</td>
<td>2022</td>
<td>A survey about scheduling in kubernetes is performed.</td>
<td>A literature review along with a classification and analysis are performed</td>
<td>In the literature it is missing: evaluation with larger clusters, leverage the scheduling framework, common repository of schedules, historic scheduling, delays on the edge</td>
<td>NA, NA, NA, Kubeernetes, NA</td>
</tr>
<tr>
<td><strong>Dynamic Scheduling of Contenally Categorised Internet of Things Services in Fog Computing Environment</strong></td>
<td>Kervic, P., Pasuk, M., Cuvark, I., Slavce, P.</td>
<td>2022</td>
<td>Create a rescheduler based on QoS parameters.</td>
<td>A model and specification are proposed. An implementation is created and using a simulation tested against 3 scenarios.</td>
<td>The meshscheduler works against the proposed scenarios, yet it has to be triggerd manually to redistribute the loads.</td>
<td>Linear-search algorithm, NA, Simulation, Own implementation, Fulfilment of case scenarios,</td>
</tr>
<tr>
<td><strong>Kubernetes as a Standard Container Orchestrator - A Bibliometric Analysis</strong></td>
<td>Carrión, C.</td>
<td>2022</td>
<td>A state of the art and bibliometric analysis of Kubernetes.</td>
<td>5 stages are performed: Formulation of research questions, search, analysis, trends and conclusions. The bibliometric analysis shows the countries, areas of interest, related topics, and most cited authors for kubernetes.</td>
<td>NA, NA, NA, Kubeernetes, NA</td>
<td></td>
</tr>
<tr>
<td><strong>Kubernetes Scheduling: Taxonomy, Ongoing Issues and Challenges</strong></td>
<td>Carrión, C.</td>
<td>2022</td>
<td>A state of the art of Kubernetes Scheduling.</td>
<td>A literature review is performed.</td>
<td>A taxonomy is made for k8s scheduling. According to the taxonomy, ongoing challenges are described.</td>
<td>NA, NA, Kubeernetes, NA</td>
</tr>
<tr>
<td><strong>Quality of Service Aware Orchestration for Cloud – Edge Continuum Applications</strong></td>
<td>Orriu, A., Aguirre, A., Truong, H.-L., Sarabjara, L., Marcos, M.</td>
<td>2022</td>
<td>A distributed scheduling architecture for a generic infrastructure is built to support a previously constructed algorithm.</td>
<td>The architecture is built atop Kubernetes and then tested using a real implementation. Two different applications are deployed.</td>
<td>Results show a QoS improvement of the applications.</td>
<td>NA, Trains, Real scenarios, K8S, Allocation success</td>
</tr>
<tr>
<td><strong>Resource Management for Latency-Sensitive IoT Applications With Satisfiability</strong></td>
<td>Avavalki, C., Tsilkanos, C., Douduri, S.</td>
<td>2022</td>
<td>A decentralized algorithm and framework are created using latency SLAs and resource preferences of nodes with a focus on containerized microservices</td>
<td>Experiments are performed using single-board computers as edge devices. Task allocation times of the decentralized resource management are around 500 ms.</td>
<td>NA, App as a collection of tasks, Activites, face recognition, public safety, team-building, Small physical testbed, Not used, Successful mapping, time, SMF Symbols</td>
<td></td>
</tr>
<tr>
<td><strong>A delay-sensitive resource allocation algorithm for a container cluster in edge computing environment</strong></td>
<td>Guo, S., Zhang, K., Gong, B., Hu, W., Qu, X.</td>
<td>2021</td>
<td>A model to analyze end-to-end delay is proposed and using the model a delay-sensitive algorithm is proposed to allocate resources.</td>
<td>The model and algorithm are first created and then evaluated. A simulation in one machine is created to test the feasibility of the model.</td>
<td>Results show that the combination of the system and algorithm reduce the end-to-end packet delay which imporve the throughput of the system.</td>
<td>Time-series prediction algorithm, NA, Simulation, Own implementation, End-to-end packet delay and throughput</td>
</tr>
<tr>
<td><strong>Container Placement and Migration in Edge Computing: Concept and Scheduling Models</strong></td>
<td>O. Olugbe</td>
<td>2021</td>
<td>State of the art article about container placement and migration in edge server.</td>
<td>Literature review about frameworks and algorithms regarding the container placement problem.</td>
<td>The container placement problem can be abstracted using multi-objective optimization or graph network models. Several algorithms can be used to solve the models.</td>
<td>Heuristic algorithms, Several industries, NA, Kubeernetes and Docker Swarm, NA</td>
</tr>
<tr>
<td><strong>Fault-tolerant adaptive scheduling with dynamic qos-awareness in edge containers for delay-sensitive tasks</strong></td>
<td>Wang, R., Chen, N., Yao, X., Hu, L.</td>
<td>2021</td>
<td>A fault-tolerant adaptive scheduling mechanism with dynamic quality of service (QoS) awareness is proposed.</td>
<td>Using EdgeCloudSim the mechanism is constructed and simulation.</td>
<td>Results show that execution of tasks reduces about 12%. There is also a task latency reduction and improvement of the system resource utilization.</td>
<td>Adaptive algorithm, NA, Simulation, Kubeernetes, Task guarantee rate</td>
</tr>
<tr>
<td>Document Title</td>
<td>Authors</td>
<td>Publication Year</td>
<td>Objective</td>
<td>Methodology</td>
<td>Results</td>
<td>Taxonomy</td>
</tr>
<tr>
<td>----------------</td>
<td>---------</td>
<td>------------------</td>
<td>-----------</td>
<td>-------------</td>
<td>---------</td>
<td>----------</td>
</tr>
<tr>
<td>Improving Container Deployment in Edge Computing Using the Infrastructure Aware Scheduling Algorithm</td>
<td>A. Dias, A. Knob, C. H. Kuprit, T. Ferrets</td>
<td>2021</td>
<td>Using a infrastructure aware algorithm on a reduction in the total time needed to deploy application is expected.</td>
<td>The scheduler is presented and a simulation tested.</td>
<td>Compared with default behaviour, the proposed solution decrease the deployment latency by more than 52%.</td>
<td>Linear-search algorithm</td>
</tr>
<tr>
<td>On the optimality of Concurrent Container Clusters Scheduling over heterogeneous smart environments</td>
<td>A. Maier Bruni, X. Garcia, J. Sánchez, S.</td>
<td>2021</td>
<td>A formulation of a scheduling problem is presented along with a heuristic approach to solve it.</td>
<td>A Concurrent Container Clusters Scheduling (C-CSV) problem is presented and an heuristic solution described in different scenarios.</td>
<td>The problem can be utilized to validate new container scheduling algorithms. The proposed problem is tested with an infrastructure aware architecture.</td>
<td>Heuristic algorithm</td>
</tr>
<tr>
<td>Polaris Scheduler: Edge Sensitive and SLO Aware Workload Scheduling in Cloud-Edge-IoT Clusters</td>
<td>S. Naval, T. Prost, A. Merlinetta, V. Crepaz, P. D. De N.</td>
<td>2021</td>
<td>Using Software Level Objectives, a framework algorithmic (heuristic) is proposed</td>
<td>Using research challenges (requirements) the framework architecture is proposed.</td>
<td>Each of the requirements are fulfilled. More work is required for testing of the framework.</td>
<td>Multi-criteria decision making (MCDA) online scheduling algorithm</td>
</tr>
<tr>
<td>Quasi-based service-time scheduling in the infra-cloud</td>
<td>M. R. F. M. S. T. B.</td>
<td>2021</td>
<td>A service time aware scheduler for the edge environment is proposed and tested.</td>
<td>Experiments are run to test the scheduler.</td>
<td>Experiments shows the algorithm outperformed other two methods, yet as network metrics were considered</td>
<td>Ant colony optimization algorithm</td>
</tr>
<tr>
<td>Quality of service provision in fog computing: Network-aware scheduling of containers</td>
<td>A. Musgrove, J. Bruns, T. Kim</td>
<td>2021</td>
<td>Using the status of the network, a scheduling algorithm is implemented as an extension to the Kubernetes default scheduler.</td>
<td>Using a physical server a k8s cluster is created consisting of 4 nodes.</td>
<td>A 30% improvement against other proposals is obtained.</td>
<td>Linear-search algorithm</td>
</tr>
<tr>
<td>Scalable fog computing orchestration for reliable cloud task scheduling</td>
<td>J. Lin</td>
<td>2021</td>
<td>For end user devices that offload tasks to cloud devices, a mechanism to orchestrate the tasks between edge servers to enable end user mobility is created.</td>
<td>Using a urban mobility software, a simulation is performed to test the scheduling algorithm.</td>
<td>Despite adding network traffic, the system shows less package lost and improves migration times.</td>
<td>(Own) task scheduling algorithm</td>
</tr>
<tr>
<td>Ultra-Reliable and Low-Latency Computing in the Edge with Kubernetes</td>
<td>R. Tolka</td>
<td>2021</td>
<td>A scheduler that considers both network latency and applications latency requirements is built.</td>
<td>With a simulation, the proposed algorithm is tested extending Kubernetes scheduler.</td>
<td>A proactive scheduler that considers latency is tested. Results show effectiveness in terms of application delay, amount of provided services and scalability.</td>
<td>Linear-search algorithm</td>
</tr>
<tr>
<td>ElasticFog: Elastic Resource Provisioning in Container-Based Fog Computing</td>
<td>N. D. Nguyen, L. A. Phuc, D. H. D. Kim</td>
<td>2020</td>
<td>Using network traffic information an elastic resource provisioning is created for containerized applications in fog computing.</td>
<td>Two IoT scenarios (3 fog nodes and 5 locations, 5 fog nodes) are used to test the algorithm.</td>
<td>Results shows improvements in terms of throughput and network latency vs default mechanisms in k8s.</td>
<td>Linear-search algorithm</td>
</tr>
<tr>
<td>Smart containers schedulers for microservices provision in cloud-fog-IoT networks: challenges and opportunities</td>
<td>R. De Paula, R. P. Garcia-Galán, S. Meliote-Exposito, I. Martinez, A. Ruiz-Reyes, N.</td>
<td>2020</td>
<td>Do a review on intelligent scheduling strategies, its benefits and impact</td>
<td>Scientific Literature review</td>
<td>The evolution of scheduling solutions is towards state-aware mechanisms that incorporate autonomic capabilities.</td>
<td>NA</td>
</tr>
<tr>
<td>Container-based cluster orchestration systems: A taxonomy and future directions</td>
<td>M. A. Rodríguez, A. Buiya, R.</td>
<td>2016</td>
<td>A state of the art is performed along with a taxonomy, survey and future directions regarding container orchestration systems.</td>
<td>A reference architecture, taxonomy and survey and future directions are presented.</td>
<td>A taxonomy for orchestration systems is created: Application, scheduling, infrastructure and management</td>
<td>Linear-search algorithm</td>
</tr>
<tr>
<td>Resource provisioning in fog computing: From theory to practice</td>
<td>W. Santos, L. W. V. Martins, B. Silva, F. D.</td>
<td>2019</td>
<td>A network aware scheduling (NAS) implementation in Kubernetes is tested.</td>
<td>The NAS algorithm is implemented as an extension of the default scheduler. A method is implemented and the algorithm tested.</td>
<td>The implementation decide based on the current status of the network infrastructure achieving a 30% reduction in network latency.</td>
<td>Linear-search algorithm</td>
</tr>
<tr>
<td>Dynamic Scheduling for Seamless Computing</td>
<td>F. Kevitkrikk, A. Seet, L. Mittnerle, H. Müller, B. Bouge</td>
<td>2018</td>
<td>Provide a time-continuous context-aware scheduler for the edge in Kubernetes</td>
<td>A set of functional and non-functional requirements are specified and a solution is constructed based on them.</td>
<td>The rescheduler meets the specific requirements in two industrial-specific scenarios.</td>
<td>Linear-search algorithm</td>
</tr>
</tbody>
</table>

Table 2.3: Background of articles.
microservices. Factors in the complexity of Edge computing, such as user movement, are taken into account to propose a Reinforcement Learning mechanism for service placement and migration. Finally and to offer an example that minimizes energy consumption under resource and latency constraints, the authors in [45] consider a joint resource dimensioning and placement problem for microservices. An algorithm is proposed and benchmarked against two greedy algorithms, showing a reduction of up to 30 percent in energy consumption.

2.5.1 Gaps in the literature

After reading the 23 articles that resulted from the literature review there are five main aspects that differentiate this project. The first one is how the scheduling process is seen by most of the authors as a one time process that only happens during a workload is instantiated. One of the concepts used in this project, similarly referred by [46] as dynamic scheduling, is dynamic orchestration. Dynamically orchestrate refers to the possibility of rescheduling components between nodes at runtime [46] and offer the possibility of acting in more ways than only assigning workloads to nodes, e.g., calling an API that modifies external components such as the network QoS or publishing a topic for consumption.

The next aspect that differentiates this project is how the authors create specific schedulers for specific scenarios. [47] work creates an infrastructure aware scheduling algorithm. [48] considers network latency and applications’ latency requirements for their scheduler whereas [49] uses the status of the network to schedule containers. The underlying concept of this project consists in the construction of a framework that can be used for any of those specific use cases i.e. the implementation of a construction tool. The design and implementation hereby presented feeds on inputs, relays on an external execution of a logic, and outputs directly on the orchestrator and/or external APIs to change not only the location of the workloads, but possibly the configuration of the infrastructure or application.

The third aspect which was not seen in the revision is that none of the authors worked with robotic applications. There is a specific field in robotic applications, that uses Object Detection (OD) applications that require powerful machines. With the advent of the edge computing, those tasks are now executed closer to the end devices, which are resource constrained.
Therefore better models or offloading strategies are topics under research. Those offload strategies could be implemented using the system proposed in this project.

The fourth aspect is the use of functional and non-functional requirements. Only [46] uses requirements to implement their application. As the proposed solution in this project consists in a software solution, rather than focusing on optimizing a specific scheduling problem, the use of both functional and non-functional requirements creates a specification over how the software should behave, centering the attention on a software solution and its specification to deal with scheduling problems.

The final aspect not explicitly mentioned by any of the authors but considered in this project is the time each of them took to implement their solution. The proposed solution in this project leverages the use of K8S operators to automate the deploy of instances that work as schedulers. This decreases the cognitive load of developers, which allows them to deploy instances quickly and focus their resources not into learning about K8S but into their expertise area e.g., robotics. The possibility of a quick and smooth deploy, from developing code to delivering value is under the umbrella concept of Continuous Integration and Continuous Deployment (CI/CD) [50] which intends to automatically build, test and integrate development changes into shared repositories to then deploy code changes to customer directly, facilitating the work for developers.
Figure 2.6: Graphical summary of the systematic literature review. Own elaboration.
Chapter 3

Design of the Solution

The purpose of this chapter is to show how the design of the solution in this project was made.

3.1 Requirements

In the following lines, the description of the non-functional and functional requirements required by the solution are presented.

3.1.1 Non-functional

• **NFR 1. Usability.** The solution should provide an easy way for the users to interact with it. No additional expertise should be needed to operate the solution. The solution aims to simplify the deployment of different optimizers (which represent the component that receives inputs, process them and produce outputs) for the application developers without the need to interact and implement different collectors (inputs) and actuators (outputs).

• **NFR 2. Extensibility.** The solution must be modularized in such a way that the addition of a new feature requires minimum effort and no -or minimum- re-architecture. Features like new type of input and outputs, or support for execution of new logic should be added without effort.

• **NFR 3. Adaptability.** The solution should provide a way to (re)configure the system and update any of the inputs, logic of the optimizer, or outputs in a declarative way.
• **NFR 4. Scalability.** The solution should be scalable, i.e., it must be possible to have one or more different instances that process a set of inputs-logic-outputs for one or many applications.

• **NFR 5. Observability.** The solution should provide information about how the systems behaves, any errors that occur and the performance of the different components. The information should be exposed according to state-of-the-art observability and monitoring tools.

• **NFR 6. Consistency.** The information should be the same across nodes at a given time.

• **NFR 7. Resiliency.** The system should be able to maintain an acceptable level of functionality and performance in the presence of faults and errors.

### 3.1.2 Functional

• **FR 1.** The solution must incorporate a component that, based on a specification, is capable of constructing instances within a finite time that can handle the input-processing-output workflow within a definitive timeframe.

• **FR 2.** The specification should contain the description of one or more inputs, an execution pattern, and one or more outputs.

• **FR 3.** Every input describes a source of information that will be utilized by the solution to collect relevant data, e.g., platform, network or application metrics.
  
  – **FR 3.1.** The collection of inputs could be push or pull based.

• **FR 4.** The processing workflow represents the description of the logic that is executed. The description includes an executable and execution pattern that will be used by the solution.
  
  – **FR 4.1.** The execution pattern can be time or event based. In case of time based, it would require a frequency time.

• **FR 5.** Every output describes how the solution should act based on the given outputs defined in the specification after the processing workflow has been executed. For example platform, network or application configurations.
3.2 Design

The design of the implementation is visually represented in Figure 3.1. The system comprises two key components: the K8S Operator and the Optimizer. The Operator takes the responsibility of receiving the Optimizer’s specification in the form of a CRD. Based on the specification, the Operator proceeds to construct all the necessary objects within the Kubernetes environment to enable the functioning of the Optimizer. Once deployed, the Optimizer initiates its operations from a preexisting image that encompasses all the essential libraries and code required for its functioning. The following sections provide a comprehensive description of the design of each individual object.

Figure 3.1: Design of the solution. The diagram displays the system consisting of the Operator and Optimizer within the dotted lines.

3.2.1 Operator

The main job of the Operator is to create, edit, and delete the required objects for every instance of the Optimizer kind that is created, edited or deleted. In order to achieve it a set of Kubernetes objects are defined, showed in Figure 3.2. A namespace is the logical separation for the objects that are created. A deployment containing the pod that executes the code is defined. The CRD of the Optimizer is also part of the objects that come with the Operator. As the Operator will directly interact with Kubernetes, a service account that is binded both to an internal and external role is defined.
Concerning the operation of the Operator, the sequence diagram depicted in Figure 3.3 illustrates the interaction between an actor and an instance of the Optimizer kind. This interaction encompasses the creation, modification, or deletion of a CRD instance. Every time an action is performed, the K8S API modifies the state of the database, thereby generating an event. The Operator listens for these events and, based on their content and the existing state of the K8S environment, initiates a sequence of actions aimed at complying the status of the database. The aforementioned actions are logically grouped and executed within an algorithm denoted as Reconcile.

The Reconcile algorithm follows a declarative paradigm where the algorithm aims to move the current state of the cluster to the desired state by any means. The algorithm watches for events that come from the K8S API regarding the Optimizer kind and acts accordingly. Algorithm 1 shows the pseudocode implemented. After a new event arrives, the first validation checks that the CRD object that triggered the event exists (Line 3). If the CRD object does not exist, that means the CRD object was deleted, so the objects that make the Optimizer i.e., the deployment, service, and configmap, are deleted (Lines 8, 13 and 18). If the CRD object exists, it means that either the CRD object was created or modified. In the case of a CRD object creation, the accompanying objects are not possible to retrieve and therefore are created (Lines 21, 26 and 31). In the case of an object modification, the supporting objects are possible to retrieve and should be modified (Lines 24, 29 and 34).
Figure 3.3: Sequence diagram of the **CRD** object creation in **K8S**.
Algorithm 1: Operator’s reconcile algorithm

```plaintext
while true do
  if new event arrives then
    if CRD is not found then
      if deployment is not found then
        Do nothing
      else
        deleteDeployment()
      end
    if service is not found then
      Do nothing
    else
      deleteService()
    end
    if Configmap is not found then
      Do nothing
    else
      deleteConfigmap()
    end
  else
    if deployment is not found then
      createDeployment()
    else
      modifyDeployment()
    end
    if service is not found then
      createService()
    else
      modifyService()
    end
    if configmap is not found then
      createConfigmap()
    else
      modifyConfigmap()
    end
  end
end
```
3.2.2 Optimizer

The Optimizer is in charge of receiving the different inputs, pass them to an external executable, receive the results and output the results. The Optimizer consists of several modules that handle the different functions performed, shown in Figure 3.4. The input module receives all the inputs required for the Optimizer to work. Some of those inputs could come from a metric collector. The logic modules takes the inputs and passes them to the external logic to be executed and get the results. The last step of the flow is done by the output module, which works as a proxy to redirect the results to the different instances. An advantage of this model is decoupling the logic for developers so they can focus on developing scheduling algorithms and leave aside the administration of inputs, outputs, and the logic of Kubernetes.

![Figure 3.4: Design of the Optimizer.](image)

The definition of Kubernetes objects required to create one instance of kind Optimizer is shown in Figure 3.5. A namespace containing the deployment, service and two configmaps is defined. The deployment contains the pod and volumes which contain the application. One of the volumes contains the logic that the code will call to get the results. The other volume contains the configuration needed for the application to operate, like the number and type of inputs and outputs. All the objects that are part of an Optimizer, are automatically created by the Operator, defined in Section 3.2.1.
Figure 3.5: Kubernetes objects of the Optimizer.
Chapter 4
Implementation

The purpose of this chapter is to show how the implementation of the solution in this project was made.

The construction of the system is divided into two main parts. The first part consists of the implementation of the Operator in Go language. Using kubebuilder the initial sketch was created. Then, the necessary functions to create the objects that create the Optimizer are implemented. As it is possible to configure objects in Kubernetes using a declarative approach [51], the main function of the Operator, called Reconcile, aims to move the current state of the cluster to the desired state. The function watches for events that come from the K8S API regarding the Optimizer kind and acts accordingly. In order to create an Optimizer, the Operator receives the specification of one of these objects. The YAML file shown in Listing 4.1 shows an example of the specification required to construct an object of the Optimizer kind with two inputs, two outputs and the logic in a configmap.

```yaml
apiVersion: mygroup.my.domain/v1alpha1
kind: Optimizer
metadata:
  labels:
    app.kubernetes.io/part-of: operator1
    app.kubernetes.io/created-by: operator1
  name: optimizer-sample1000
spec:
  image: optimizer:2.0
  reconciliation_freq: 10
  port: 31234
  inputs:
```
The second part of the construction of the system consists of the implementation of the Optimizer in Python language. As part of the specification of the Optimizer comes from the CRD, it is necessary to make sure that the information exists beforehand. Therefore, all the specification is previously mapped from the CRD to a configuration file in the Optimizer. Using the information in that file, the Optimizer knows about the inputs, logic and outputs. As the Optimizer handles several loops at the same time, a multi-threading approach is used. A main thread that constructs three threads is defined. The main thread calls the other three threads, runs a HTTP server using the Flask library⁰ and publishes metrics using an HTTP library to export metrics called prometheus_flask_exporterⁱ. The first thread, makes requests to the Metric Collector in an asynchronous way with the help of both the asyncio library⁲ and the requests library⁳. The second thread handles the information that is exchanged with Robot Operating System (ROS) that follows a publish–

---

⁰Available at https://github.com/pallets/flask
ⁱAvailable at https://github.com/rycus86/prometheus_flask_exporter
⁲Available at https://docs.python.org/3/library/asyncio.html
⁳Available at https://github.com/psf/requests
subscribe pattern with the help of the rclpy library\(^*\). The third and final thread is in charge of periodically executing the python logic.

Once the Operator and Optimizer have been implemented, container images of both objects are created and placed in a repository. Then, the Operator is deployed and started in Kubernetes. Figure 4.1 shows the output log of one operator running at two moments. Once the Operator is up and running, instances of the Optimizer kind are possible to be created.

Figure 4.1: Output log of a running Operator. Figure (a) shows the output during the startup whereas Figure (b) shows the moment when the operator creates an object of the Optimizer kind. In the Figure it is possible to see the event coming and the following moments when the deployment, service, and configmap are created.

4.1 Steps for end users

Once the system is running, there is a procedure required for a end user to make use of the system:

Prerequisites:

- Create a namespace in K8S.

- The logic needs to be an executable file in python that receives a number of inputs and produces a number of outputs. The number of inputs and outputs must correspond to those defined in the configuration file.

\(^*\)Available at [https://github.com/ros2/rclpy](https://github.com/ros2/rclpy)
Procedure:

1. Upload the logic developed into a configmap with the command `kubectl create configmap <nameConfigmap> -from-file =exefile=<locationFile> -n <namespace>`. The only characteristic required is that the logic needs to receive a predefined number of inputs and produce a predefined number of outputs.

2. Create the configuration file.

3. Apply the configuration file to K8S. The general syntax using `kubectl` is `kubectl apply -f <locationFile>`
Chapter 5

Evaluation and validation

The purpose of this chapter is to evaluate how the set of NFR and FR are expressed in the solution implemented. Also, a description and results of an use and test case are presented to validate the software implementation.

5.1 Evaluation of requirements

Tables 5.1 and 5.2 presents a justification of how the solution complies with each requirement proposed in Section 3.1.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Justification</th>
</tr>
</thead>
<tbody>
<tr>
<td>NFR Usability 1.</td>
<td>As shown in Section 4.1, a final user needs to execute three steps in order to have an instance of the optimizer up and running. Without the solution presented, to have the same results as before, an user would have to execute two main procedures. The first one would be the development of an application with several functions such as the interaction with K8S, with ROS, the ability of publishing metrics or the use of threads. The second step would consist in the deployment of the previous application in K8S. For this, a definition of five YAML files (namespace, service, deployment, two configmaps) would follow their application in K8S. These two steps require time and knowledge from the developer, who should be focusing on its own tasks, e.g. optimizing the OD tasks and in general those of the robotics knowledge area.</td>
</tr>
</tbody>
</table>

(continues on the next page)
<table>
<thead>
<tr>
<th>Requirement</th>
<th>Justification</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NFR 2. Extensibility</strong></td>
<td>As shown in Section 4, the application is developed using functions and threads, and it is deployed as different objects in K8S. If a future requirement contains the addition/modification/deletion of a new type of input, execution pattern or output, there should be only a matter of adding/modifying/deleting a thread, function, or object in K8S. If current deployments are required to have those new changes, a new image should be created and uploaded to the repository and the configuration files of the optimizers pointed towards it (see Line 9 in Listing 4.1).</td>
</tr>
<tr>
<td><strong>NFR 3. Adaptability</strong></td>
<td>To deploy an Optimizer the creation and configuration of a CRD is required. In this file, shown in Listing 4.1, it is possible to add any number of inputs, the logic of the optimizer, and any number of outputs. The redeployment of one of this file in K8S would create a new optimizer with new characteristics. The act of proposing the desired system (and not the instructions of how to construct one) comply with the declarative approach.</td>
</tr>
<tr>
<td><strong>NFR 4. Scalability</strong></td>
<td>The application in K8S of one CRD creates one Optimizer. The procedure is possible to do it as many times as desired. The important is to consider the name of the object being deployed (see Line 7 in Listing 4.1), as it is not possible to have two objects in K8S with the same name. Also, it is important to be sure that the inputs and outputs do not overlap between Optimizers as it could bring problems within the Optimizers deployed.</td>
</tr>
<tr>
<td><strong>NFR 5. Observability</strong></td>
<td>In order to see the internal behaviour and status of the application, two characteristics were deployed. The first one involves the publishing of metrics by the application, as shown by the use of the prometheus_flask_exporter library mentioned in Section 4. The second one is the continuous output to console from the application to get information about moments and status of the execution of the application.</td>
</tr>
<tr>
<td><strong>NFR 6. Consistency</strong></td>
<td>The objects that both the Operator and Optimizer use are stored in the native database that comes with K8S called etcd [52]. etcd is a strongly consistent, distributed key-value store for distributed systems [53]. Using etcd to store all the objects from the Operator and Optimizer, the consistency of the solution is guaranteed.</td>
</tr>
</tbody>
</table>

*(continues on the next page)*
Resiliency

Compared to a deployment where a daemon runs, and if it crashes, nothing will run again, the advantage of deploying in K8S is that K8S will try to fulfill the configured state as long as it runs. For example, if a node is disconnected from the K8S cluster and it had 2 deployments running on it (not tied to that node, as shown in Section 2.3.2.1), K8S will try to restart the deployments in other nodes. This characteristic adds an acceptable level in the face of faults and error at a low cost. Complex configurations, tests and systems could increase the resiliency of the system, nonetheless, the topic itself could be a single area where to focus a project.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Justification</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NFR 7.</strong> Resiliency</td>
<td>Compared to a deployment where a daemon runs, and if it crashes, nothing will run again, the advantage of deploying in K8S is that K8S will try to fulfill the configured state as long as it runs. For example, if a node is disconnected from the K8S cluster and it had 2 deployments running on it (not tied to that node, as shown in Section 2.3.2.1), K8S will try to restart the deployments in other nodes. This characteristic adds an acceptable level in the face of faults and error at a low cost. Complex configurations, tests and systems could increase the resiliency of the system, nonetheless, the topic itself could be a single area where to focus a project.</td>
</tr>
<tr>
<td><strong>FR 1.</strong></td>
<td>As shown in Section 3.2, the solution is composed of two parts. The first one, the Operator, based on the specification described in the CRD creates instances of Optimizers, which handle the input-processing-output workflow of the solution.</td>
</tr>
<tr>
<td><strong>FR 2.</strong></td>
<td>The CRD used to create instances of Optimizers, is a configuration file used to specify the characteristics required to create the instances and for their internal operation. In the configuration file, it is possible to place one or more inputs, the description of an execution pattern and one or more outputs.</td>
</tr>
<tr>
<td><strong>FR 3. &amp; 3.1</strong></td>
<td>Currently, two types of inputs are defined. The first one, is of type <em>ros2topic</em> and represents a push way to get information. The model of communication in ROS uses topics. Participants subscribe to topics and get the information that is published by the producers. In this sense, the model represents a push-model that starts when the publishers create and broadcast the information for the listening members (subscribers). On the contrary, the second type of input is of type <em>Prometheus</em>. This way of getting the information extracts (and does not receive, as in the first model) the information periodically. The petition that is sent to the source is a specially crafted HTTP GET request built using information from the CRD that contains the query (see line 20 in Listing 4.1).</td>
</tr>
</tbody>
</table>
At present, the execution of the logic occurs as a consequence of a system timeout reaching its expiration, which represents a time-based execution model in the solution. An event-based execution model would require attention on the moment when the inputs get to the system to trigger the execution. It is essential to establish a coherent relationship among all system components (inputs, execution, and outputs), given that the event source may originate from either an input or an external source. In the same sense, a clear definition of the outputs is required if an event-based workflow is to be implemented.

Just as the inputs are of two different kinds, the outputs are as well of two kinds. The first kind, and in coherence with ROS, is defined as ros2topic and represents in the system a publication of information in a ROS topic. The second type of output is defined as kubernetes and represents the possibility of interacting directly with K8S, as shown in the Test case (Section 5.4.2).

Table 5.2: Functional requirements.

## 5.2 Environment Configuration

The use and test cases described in the next two sections are deployed in a testbed. The use case was tested with final users, whereas the test case was tested internally. The environment consists of two devices (shown in Figure 5.1): an NVIDIA Jetson NX device (shown in Figure 5.1a), that represents the user device and an NVIDIA Jetson AGX device (shown in Figure 5.1b), as the Edge device. A lightweight distribution of Kubernetes, called K3S*, is deployed over the devices.

## 5.3 Use Case

### 5.3.1 Design

With the advent of mobile devices and robotics, a move towards executing OD tasks in resource constrained devices is on demand. The main purpose of

*Available at [https://k3s.io/](https://k3s.io/)
OD tasks consists of locate and classify objects in a frame. Yet, OD tasks are computational expensive so there is a concern on making either OD algorithms more accurate or creating offload strategies to move the OD tasks to nearby devices. Currently, Ericsson is exploring OD strategies and applications to offload tasks from resource constrained devices to nearby devices such as those located on the edge computing. In consequence the intention of this use case is to validate the feasibility of the solution.

One of the internal applications developed by Ericsson that performs OD, runs simultaneously lightweight and heavyweight OD processes in corresponding devices, as shown in Figure 5.2. Both the lightweight and heavyweight processes output device detections to a detection failure module that combines them in certain way. Then, a scheduler feeds on that information to decide for a master process between the lightweight and heavyweight processes in order to locate and classify objects in frames.

The OD Scheduler module (shown inside a blue box in Figure 5.2) that runs on the lightweight device holds significant importance for this project. The OD Scheduler module is expected to be replaced by the Optimizer, defined in Section 3.2.2. The default operation of the OD Scheduler module has one input, a simple logic that consists of an if-else structure, already ported to a single executable file, and 3 outputs.

To evaluate the performance of the application, three metrics are used. The first one is the Mean Average Precision (mAP). To understand how Average Precision (AP) is obtained it is good to understand first some basic metrics.
Figure 5.2: Architecture of the application case study.

such as Precision and Recall. Precision is the ability of a model to identify only relevant objects. It is the percentage of correct positive predictions. Recall is the ability of a model to find all relevant cases i.e., the percentage of correct positive predictions among all given ground truths [54]. The metrics are defined as:

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

Where $TP$ represents a true positive i.e., a correct detection of a ground truth, $FP$ represents a false positive i.e., an incorrect detection of a nonexistent object, and $FN$ represents a false negative i.e., an undetected ground truth.

By estimating the area under the curve of the precision $\times$ recall relationship it is possible to get $AP$. The justification of this is exactly described by [54]:

The precision $\times$ recall curve can be seen as a trade-off between precision and recall for different confidence values associated to the bounding boxes generated by a detector. If the confidence of a detector is such that its $FP$ is low, the precision will be high. However, in this case, many positives may be missed.
yielding a high $FN$, and thus a low recall. Conversely, if one accepts more positives, the recall will increase, but the $FP$ may also increase, decreasing the precision. However, a good object detector should find all ground-truth objects ($FN = 0$ high recall) while identifying only relevant objects ($FP = 0$ high precision). Therefore, a particular object detector can be considered good if its precision stays high as its recall increases, which means that if the confidence threshold varies, the precision and recall will still be high. Hence, a high area under the curve (AUC) tends to indicate both high precision and high recall.

The second variable is the real-time $mAP$ ($mAP$ runtime). The only difference against $mAP$ is the image with which the detection results are compared to. For $mAP$ the comparison is performed against the ground truth image whereas for $mAP$ runtime the comparison uses the image that that is perceived by the sensor in real-time. The third and final variable is the end-to-end latency (latency), which represents the time in seconds since the frame is read until the detection results are received in the device platform.

### 5.3.2 Scenarios

In order to compare the results, three scenarios are devised (Figure 5.3). **Scenario 1**, shown in Figure 5.3a is the baseline. This scenario represents the status quo of the OD application. It is run as a python executable directly on the devices without any virtualization layer. The executable in the lightweight device runs three processes. The OD process itself, the detection failure module and the OD Scheduler. The Heavyweight device runs only the OD process that corresponds to it, i.e., the heavyweight OD process.

**Scenario 2**, shown in Figure 5.3b, adds a K8S layer to the Scenario 1. After installing and configuring the K8S layer, the python executable on every device is converted into a docker image that runs atop of K8S. The purpose of this scenario consists in determining whether the addition of the virtualization layer degrades the system. Finally, **Scenario 3**, shown in Figure 5.3c, extends the second scenario. The modification in this scenario consists of making use of the Optimizer proposed in the project. For this, the inbuilt OD Scheduler is replaced by the Optimizer (described in Section 3.2.2) and deployed by the Operator (described in Section 3.2.1).
(a) Scenario 1.

(b) Scenario 2

(c) Scenario 3.

Figure 5.3: Deployment scenarios for the use case application.
5.3.3 Results

According to Ericsson’s subject matter experts of the OD application, the recommendation is to run for every scenario a set of fifty (50) predefined videos three (3) different times. In consequence, and for every of the scenarios defined in subsection 5.3.2, three datasets are collected and averaged. General averages for every of the three metrics are collected. Table 5.3 shows the results and Figure 5.4 shows a graphical representation of the results. The discussion of the results is presented in Section 5.5.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Scenario 1</th>
<th>Scenario 2</th>
<th>Scenario 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>mAP</td>
<td>0.417</td>
<td>0.374</td>
<td>0.376</td>
</tr>
<tr>
<td>mAP runtime</td>
<td>0.385</td>
<td>0.277</td>
<td>0.282</td>
</tr>
<tr>
<td>Latency (s)</td>
<td>0.196</td>
<td>0.193</td>
<td>0.192</td>
</tr>
</tbody>
</table>

Table 5.3: Averaged results for the 3 metrics in every scenario.

![Graphical comparison of the three metrics](image)

(a) mAP  
(b) mAP runtime  
(c) Latency

Figure 5.4: Graphical comparison of the three metrics (mAP, mAP runtime and latency) in scenarios 1, 2 and 3.
5.4 Test Case

5.4.1 Design

In order to test the functionality of rescheduling a workload in K8S, the following scenario is proposed. There is a deployment of a web server (nginx) running in K8S on the two devices from Figure 5.1. The deployment contains a label and depending on the value of the label, the deployment will run on a specific device. Additionally, there are two external objects that produce values that change every 30 seconds. The Optimizer should input those values into the system and use them to execute a previously provisioned external python logic. The logic adds the values together and outputs the result. Depending on the value of the output, the Optimizer will reschedule the deployment containing the nginx server to one device or the other. If the Optimizer works as described, it means that the system is capable of receive inputs, use the values to execute an external logic and depending on the output of the logic, reschedule a workload in K8S. The previously functioning would show the system’s ability to reschedule a running workload in K8S depending on the value that the external logic outputs, which was previously loaded from two external values.

5.4.2 Results

No quantitative variable, such as time taken, is considered; instead, a status validation is performed to confirm that the deployment is rescheduled onto a new server after the value of the logic changes. After configuring the environment and running the test case, the moment when the rescheduling happens is hereby presented. Figure 5.5 illustrates the moment in time when the rescheduling happens. The figure displays two console logs: the left console shows the Optimizer running, while the right one, shows commands that display the location of the deployed workload. In the left console, the red box indicates the result of executing logic. Inside the red box, the value changes from 13 to 12. When this change occurs (highlighted inside the green box), the Optimizer redeploy a workload from one of the nodes to the other node. The movement is visible in the yellow bubble that highlights the column where the workload is running. The node where the workload is running changes from lab-nx-4 to lab-agx-4.
5.5 Discussion

After creating the requirements, making a design, implementing a solution, and testing it against two cases, several considerations are worth discussing. The first one relates to the process of eliciting requirements, which demands a deep comprehension of the problem. The solution does not solve a specific problem directly; instead, it serves as a tool to construct instances that solve particular problems. This abstraction places the solution one level higher, requiring careful consideration of NFRs. Characteristics such as usability, extensibility, or adaptability, to name a few, deserve higher attention since the value provided by the solution depends on them.

The metrics mAP and mAP runtime degrade in scenarios 2 and 3 (K8S and K8S+Solution). However, the noticeable difference is observed from the moment Kubernetes is used (Scenario 1 vs Scenario 2 & 3), indicating that the additional layer from K8S is the primary cause, as there is not a noticeable difference between scenarios 2 and 3. On the other side, it is also worth noting that the purpose in this project is not to improve the current solution of a problem but rather to offer a software product that fulfills certain set of software requirements to ease the cognitive load of developers and enable them to focus on their area of expertise, which in the case of Ericsson is robotics.

The use of an additional and external system for the core functioning of an application could have yielded in significant degradation. Yet, the impact of the proposed solution in the metrics is measured in hundredths of a second. Accepting or rejecting the delay depends on the use case being evaluated. For example, in an OD application used to classify and locate pedestrians from a moving car at 100 km/h, the added delay could hold considerable importance.
Conversely, the added difference in classifying and locating cattle in smart agriculture might be negligible.

During the configuration of the application an effort was made in decoupling the scheduler mechanism. This situation is probably common to all the applications that would like to make use of the proposed solution, unless they are created from scratch and modularized in such a way to fit with the proposed solution. Nonetheless, once an application adopts the solution, and as more applications follow suit, the specific features of each application could add value to the rest. Consequently, the more widely the solution is implemented, the more diverse features it can offer. The previous idea, and from the standing point of the cloud and infrastructure department of a company, also shows how the solution is a tool that could help developers automating and integrating software development with IT deployment and operations.

Building an operator in K8S is not an easy task, but once it is accomplished it could bring automation to future deployments. One of the parts of the solution, the operator, continuously monitors events from a predefined kind of object in K8S. An added value from this project, directly to the area in Ericsson where it was developed, is the design, construction and deployment of a K8S operator. There is the possibility to either create a K8S operator from scratch, with the previous advantage of having created one, or branch the source code from the current implementation, modify it and create a new Operator. The introduction of the operator object offers the potential to facilitate future deployments of applications created in K8S, thereby contributing to Ericsson’s Cloudification strategy (see [55]).

After the previous discussion, it is now possible to specifically answer the research questions that were proposed. To answer RQ1, *What are the requirements for developing a framework that is able to receive inputs, a logic, and outputs to ease the construction of a scheduler for the device-cloud continuum?*, after conducting the literature review and analyzing the situation with the stakeholders a set of NFR and FR has been created and described in Section 3.1. The decision to construct NFR arises from the importance of providing qualities in how the system behaves. As the system is intended to be a tool that fits many use cases (for developers who are not expert in cloud technologies such as K8S), characteristics such as usability, extensibility, adaptability, or scalability are taken into heavy consideration to provide a better developer experience.
To answer RQ2, *How can a software implementation be devised to fulfill the identified set of requirements?*, it is possible to state that a software implementation based on a K8S Operator and a custom application written in Go, which satisfies the set of requirements is hereby constructed. The design of the solution is described in Section 3.2 and the implementation details are described in Chapter 4. To test the solution against the requirements, an evaluation of each requirement is provided in Section 5. Finally, to assess the solution, an use case (Section 5.3.1) and a test case (Section 5.4.1) have been designed and validated against the software solution to check the feasibility of the software implementation created.
Chapter 6
Conclusions and Future work

In this chapter, conclusions and future work of the project are discussed.

6.1 Conclusions

First and foremost it is important to mention that the purpose of this project was not to optimize a specific problem but rather to build a solution capable of adapting to a predefined set of problems. The project’s focus centered on how a set of non-functional and functional requirements could describe the desired characteristics and conditions that a solution should have. The main problem addressed in this project is the lack of a simple and extensible mechanism for dynamically scheduling workloads. A comprehensive literature review on the topic provided insights into the characteristics of the problems and guided the creation of the requirements.

Once a stable version of the requirements were made, a design and implementation were created to show a feasible solution for the problem. One solution that satisfies the requirements is described as a composition of two objects: a K8S operator and a custom application. By combining both and utilizing test cases, it is shown how the solution works for two rather different scenarios, highlighting its usability, extensibility and adaptability. Nonetheless, the introduction of a solution of this kind brings additional considerations. On one side, the architecture and components of an application that would like to make use of the proposed solution require certain amount of work in order to decouple the scheduler mechanism, which would bring a cost. Yet, modularizing an application could lead the application towards a microservices architecture, offering some advantages and preparing the application
for cloud-native technologies such as Kubernetes.

Besides the added characteristics of the NFRs, the value of the solution hereby presented lies in its abstracting model of a scheduling problem. Typically, the metrics required to make a decision come from different sources such as routers, antennas, physical servers, monitoring solutions, IoT devices, and applications, etc. The possibility of using any number of those metrics by modelling and characterizing the inputs, logic, and outputs, provides significant flexibility for anyone facing a scheduling problem to leverage the solution. For instance, to solve an energy optimization problem, one can easily add or remove various metrics like energy consumption, requests per second, available size, or uptime of any component to analyze their impact on the outputs. The outsourcing of the logic gives the possibility to focus on the algorithm rather than implementation details with the inputs or outputs, such as the use of specific libraries or implementation of functions. For example, if a logic is based on a neural network that receives two inputs, but there’s a desire to compare it with a fuzzy logic approach, only the logic itself needs to be changed without requiring an entire re-architecturing, redesign, and re-implementation of the entire system.

6.2 Future work

During the execution of the project, several aspects were identified as possible future considerations to continue. The following list presents a description of each one.

- **New and different scenarios.** With the built solution, any number of different scenarios could be tested. As a tool to make quick and easy deployments, it is possible to test the feasibility and obtain quick results of scheduling algorithms.

- **Performance.** As described in the document, the purpose in this project was not to optimize metrics of any kind but rather to provide a framework to ease dynamic rescheduling. A next step could make use the solution to make measurements and introduce changes to improve the performance of scheduling algorithms.

- **Infrastructure metrics.** In hand with the previous item, a measurement and modification of infrastructure metrics such as CPU or memory could be implemented to see the impact and usage of them. In
Kubernetes, it is relatively easy to set limits on memory and CPU (see [56]).

- **Statefulness.** The solution and the way it was designed considers only real-time data that is passed to the logic and produces an instantaneous result. It would be highly valuable, to design a mechanism and implement it, that considers historical data. Nonetheless this would increase the complexity of the system as it would be necessary to implement a database to store the data, a mechanism to process and execute it, and the definition of thresholds that would impact the decision as how much in the past would be useful to store and how much computational power would be required to process the data. And not only the processing of historical data would be interesting to consider. With the same information forecasting mechanisms that predict consumption trends could be extremely valuable.

- **Implementation of new execution patterns.** The current execution pattern executes in console a python file every predefined amount of time. Another mechanism could obtain the same results making use of HTTP requests. Which would be required to be implemented. Other mechanisms could be like the python execution but with different languages and their considerations such as Go.

- **Security considerations.** More than ever, securing applications and the underlying infrastructure has gained importance. One of these considerations is inserting since the conception of the products security features as it is easier, more effective and cost less. Besides the previously mentioned, the devices used in the device-cloud continuum impact directly the physical world hence the compromise of these devices could have more impact. No security contemplations were made during this project. Consequently, much work is still left to be done.
References


