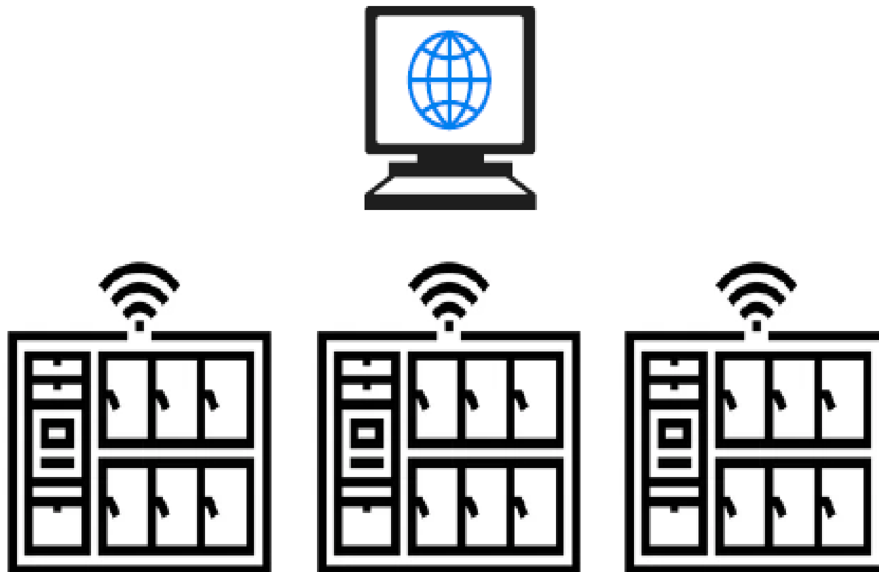

LockyCloudCore - CTT locker management infrastructure

MIGUEL MOREIRA LOPES SELEIRO

(Licenciado)

Trabalho de Projeto para obtenção do Grau de Mestre em

Engenharia Informática e de Computadores

Orientadores: Doutor José Simão
Doutor Nuno Datia

Júri:

Presidente: Doutor Nuno Miguel Machado Cruz
Vogais: Doutor Manuel Martins Barata
Doutor José Simão

Setembro de 2024

LockyCloudCore - CTT locker management infrastructure

MIGUEL MOREIRA LOPES SELEIRO

(Licenciado)

Trabalho de Projeto para obtenção do Grau de Mestre em

Engenharia Informática e de Computadores

Orientadores: Doutor José Simão, ISEL/IPL
Doutor Nuno Datia, ISEL/IPL

Júri:

Presidente: Doutor Nuno Miguel Machado Cruz, ISEL/IPL
Vogais: Doutor Manuel Martins Barata, ISEL/IPL
Doutor José Simão, ISEL/IPL

Setembro de 2024

To my family and friends.

Acknowledgements

First I would like to thank my thesis supervisors, Professor José Simão and Professor Nuno Datia, for all the guidance given throughout this project thesis. Your availability to review and give suggestions during this project gave me the strength and motivation to complete this thesis.

I would also like to thank Dr. Rogério Campos Rebelo, as the representative of the Correios de Portugal (CTT) Locky team, for the clarification of the functional requirements, providing business-perspective analysis of locker features, and for accommodating all of the students involved in the CTT Project.

I also leave a word of thanks to our Lisbon School of Engineering (ISEL), for providing all of the theoretical and practical knowledge that has lead me to this point, and for the trust in attributing a scientific research scholarship, with the reference *"Projeto CTT_188829/2024"*, to this project.

I thank my friends, who not only motivated me at difficult times, but were also great company for me to temporarily relax with and take my mind off the academic world every day.

Last but not least, I thank my parents, Cristina and João, for always being here for me, listening, and supporting me throughout my entire academic journey. I wouldn't have been able to achieve everything I have without the both of you.

Statement of integrity

I declare that this project work is the result of my personal and independent research. Its content is original, and all sources listed in the bibliographic references were consulted and are duly mentioned in the text. I further declare that all scientific and technical references relevant to the development of the work are duly cited and included in the bibliographic references.

The author

____

Lisbon, September 2024

LockyCloudCore - CTT locker management infrastructure

Copyright© MIGUEL MOREIRA LOPES SELEIRO, Instituto Superior de Engenharia de Lisboa, Instituto Politécnico de Lisboa.
The Instituto Superior de Engenharia de Lisboa and the Instituto Politécnico de Lisboa have the right, perpetual and without geographical boundaries, to file and publish this dissertation through printed copies reproduced on paper or on digital form, or by any other means known or that may be invented, and to disseminate through scientific repositories and admit its copying and distribution for non-commercial, educational or research purposes, as long as credit is given to the author and editor.

This document was created using the (pdf)LaTeX processor, based in the "iselthesis" template, developed at the DEETC of ISEL-IPL.

LockyCloudCore - CTT locker management infrastructure

Abstract

The concept of Edge Computing came has a way to address the concerns related to the integration of Internet of Things (IoT) in a cloud service infrastructure, such as resource limitations, performance requirements, and data safety and privacy. For developers wanting to create a such an infrastructure, an analysis on how to perform this integration need to be made, from choosing which architecture to use in the system, to how it will be deployed, and how to optimize the data processing at the edge.

Through the development of a management infrastructure for the CTT locker network, Locky, we demonstrate the factors that need to be taken into consideration when developing a distributed system with edge devices, as well as testing and comparing orchestration runtimes for edge devices.

Keywords: Cloud, Distributed Computing, Edge Computing, Orchestration, Metric Extraction

LockyCloudCore - CTT locker management infrastructure

Resumo

O conceito de Computação de Borda surgiu como uma forma de abordar as preocupações relacionadas à integração de Internet das Coisas (IoT) numa infraestrutura de serviços em nuvem, como limitações de recursos, requisitos de desempenho e a segurança e privacidade de dados. Para programadores que desejam criar tal infraestrutura, é necessário fazer uma análise de como realizar esta integração, desde a escolha de qual arquitetura utilizar no sistema, até como será orquestrado e como otimizar o processamento de dados na edge.

Através do desenvolvimento de uma infraestrutura de gestão para a rede de cacifos CTT, Locky, demonstramos os fatores que se devem ter em consideração no desenvolvimento de um sistema distribuído com dispositivos IoT, bem como testamos e comparamos tempos de execução de soluções de orquestração para dispositivos IoT.

Palavras-chave: Nuvem, Computação Distribuida, Computação de Borda, Orquestração, Extração de Métricas

Contents

List of Figures xix

List of Tables xxi

Listings xxiii

Glossary xxv

Acronyms xxvii

1 Introduction 1

1.1 Context 1
1.2 Motivation 2
1.3 Proposed Solution 3
1.4 Contributions 3
1.5 Document Structure 3

2 Background and Related Work 5

2.1 Distributed Architectures 5
2.1.1 Service-Oriented Architecture - SOA 6
2.1.2 Microservices Architecture 6
2.1.3 Event-Driven Architecture - EDA 7
2.1.4 Serverless Architectures 8
2.2 Edge Computing 9
2.2.1 Serverless Edge Paradigm 10
2.3 Orchestration Runtimes 12
2.3.1 Dapr - Distributed Application Runtime 12
2.3.2 Knative 13
2.3.3 OpenFaaS 14
2.3.4 SpringBoot 14
2.4 Observability 14
2.4.1 Prometheus 15
2.4.2 Graphite 15
2.4.3 Grafana 16

2.5 Related Work . . . 16
2.5.1 Lightweight Kubernetes Comparison . . . 16
2.5.2 Blueprint Toolchain . . . 17
2.5.3 Kubernetes k4.0s . . . 17
2.5.4 faas-sim - A trace-driven Function-as-a-Service simulator . . . 18
2.5.5 LaSS - Latency Sensitive Serverless . . . 18
2.5.6 Kappa - Serverless IoT Deployment . . . 18

3 Project Analysis **21**
3.1 Locky Infrastructure . . . 21
3.2 Problem Statement . . . 22
3.3 Metrics . . . 24

4 Evaluation of Frameworks for Service Distribution **27**
4.1 Dapr Runtime . . . 27
4.1.1 Locky with Dapr . . . 27
4.1.2 Testing Dapr in Raspberry Pi 3 . . . 28
4.2 OpenFaaS Framework . . . 29
4.2.1 Locky with faasd . . . 29
4.2.2 Testing faasd in Raspberry Pi 3 . . . 30

5 Development **33**
5.1 Back-end Locker Interactions . . . 33
5.2 Functions . . . 34
5.3 Relational Model . . . 36
5.4 Function Deployment . . . 37
5.5 Metrics UI . . . 40

6 Performance results and Project Discussions **43**
6.1 Results . . . 43
6.1.1 Test 1 - Memory Usage . . . 43
6.1.2 Test 2 - CPU Usage . . . 44
6.1.3 Test 3 - Response Latency . . . 45
6.2 Discussion . . . 46
6.2.1 Result analysis . . . 46
6.2.2 Ease of Coding, Deployment and Maintenance . . . 46
6.2.3 Observability of both frameworks . . . 47

7 Conclusions and Future Work **49**
7.1 Achievements . . . 49
7.2 Main Difficulties . . . 50

7.3 Future Work 51

Bibliography **53**

Appendices

A JMeter Latency test output **59**

A.1 faasd results 59

A.2 SpringBoot results 60

List of Figures

2.1 Microservices architecture scheme versus SOA architecture scheme . . . 7
2.2 Event-Driven Architecture scheme 8
2.3 Serverless Architecture scheme 9

3.1 Simplified design of the Locky infrastructure 22
3.2 Structure of the GQM Model (adapted from [12]) 24

4.1 Locker infrastructure with Dapr. 28
4.2 Service deployment effects on the available (free) RAM. 29
4.3 Locker infrastructure with OpenFaaS and faasd. 30
4.4 Function deployment effects on the available (free) RAM. 31

5.1 Locker workflow events. 34
5.2 Locker function environment and interactions. 35
5.3 Simplified relational model of the Locky system 38
5.4 Function deployment flow. 38
5.5 Function environment file structure 40
5.6 Metrics UI 41

6.1 Deployment effects on RAM (higher is better). 44
6.2 CPU usage for each chosen event. 45

List of Tables

3.1 Raspberry Pi 3/Raspberry Pi 4 comparison. 22

6.1 Response latency to HTTP Requests. 45

Listings

5.1 OpenFaaS function template. 36
5.2 OpenFaaS metadata template. 39

Glossary

distributed architecture	a singular, large computing network with its the business concerns separated along many services. xxv, 2, 3, 5, 6, 8, 9, 10, 21, 49
distributed computing	a field of computer science for system design based on a distributed architectures. 2, 7, 9, 12, 14, 15, 22, 27
framework	platform that provides a foundation for developing software applications. 2, 3, 5, 11, 12, 14, 16, 18, 23, 25, 27, 29, 31, 43, 46, 47, 48, 49, 51
function	a single-purpose service that is both stateless and ephemeral (used in serverless). xix, xxvi, 3, 8, 9, 10, 11, 13, 14, 18, 25, 29, 30, 31, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 50, 51
granularity	the breaking down of larger tasks into smaller ones. 6
loosely-coupled	weakly associated, can be changed without affecting the rest of the system. 6, 7, 8, 27
metric	a measure of a software characteristic that is quantifiable or countable. xix, 3, 12, 14, 15, 16, 22, 23, 24, 25, 27, 28, 29, 31, 33, 40, 41, 42, 43, 46, 47, 48, 50
microservices	an architectural and organizational approach to software development where software is composed of small independent services that communicate over well-defined APIs. xix, 2, 3, 6, 7, 9, 10, 14, 17, 27
monolithic	a singular, large computing network with one code base that couples all of the business concerns together. 2, 5, 6
runtime	an execution environment for computer programs written in a specific computer language. 2, 3, 5, 12, 13, 17, 18, 24, 25, 27, 28, 29, 36
scalability	the capacity to be changed in size or scale. 6, 7, 11

serverless	an application development and execution model that enables developers to build and run stateless functions without having to worry about deployment strategies and underlying infrastructure. xix, xxv, 2, 3, 8, 9, 10, 11, 12, 13, 14, 18, 23, 27, 29, 41, 43, 46, 49, 50, 51
service	a mechanism to enable access to one or more system functionalities, where the access is provided through an API. xix, xxv, 2, 3, 4, 5, 6, 7, 8, 10, 12, 13, 14, 15, 16, 17, 21, 22, 23, 25, 27, 28, 29, 30, 31, 34, 37, 43, 44, 45, 46, 47, 48, 49, 50
tightly-coupled	strongly associated, heavily relies on the rest of the system, can not be changed easily. 3

Acronyms

API Application Programming Interface xxv, xxvi, 6, 12, 13, 14, 22, 28

BaaS Backend-as-a-Service 8, 9

CDN Content Delivery Networks 9
CLI Command Line Interface 14
CNI Container Networking Interface 14
CPU Central Processing Unit 3, 22, 25, 43, 44, 46, 47
CTT Correios de Portugal 1, 5, 9

DB Database 21, 22, 33

EDA Event-Driven Architecture xix, 2, 6, 7, 8, 10, 13

FaaS Function-as-a-Service 8, 9, 10, 11, 18

GQM Goal Question Metric xix, 24, 43, 46, 49

HTTP Hypertext Transfer Protocol 12, 13, 21, 22, 25, 35, 36, 45, 48, 50

IoT Internet of Things 10, 11, 13, 18, 19, 23, 49, 51

RAM Random Access Memory xix, 3, 11, 16, 22, 25, 29, 30, 31, 44
REST REpresentational State Transfer 6
RPC Remote Procedure Call 6, 17

SOA Service-Oriented Architecture xix, 2, 6, 7, 10, 23, 24
SOAP Simple Object Access Protocol 6

VM Virtual Machine 9, 16

1 Introduction

In this chapter, the context of this work is presented along with the reason for its necessity. Then, the motivation behind the research and development is presented. A proposed solution is then summarized, and the contributions of this work are explained. Finally, the document structure is outlined, explaining the following chapters.

1.1 Context

During the COVID-19 pandemic, pandemic-induced shocks caused online retailers to increase their sales during the lockdown period [64]. In post-lockdown, the online retailers were able to maintain their sales, which opened a window of opportunity for couriers to strengthen their business and increase their market activity. However, to achieve sustainable growth, they need to be efficient, more environmentally friendly [18] and keep their business attractive to customers [51]. One solution that has proved successful is the use of smart lockers, distributed in many convenient locations, e.g. supermarkets, public places, and more [40]. These lockers address both delivery efficiency, ensuring that a scheduled order is always delivered successfully, and customer convenience and satisfaction, since customers can collect their online orders at a time that suits them.

In Portugal, Correios de Portugal (CTT) adopted this solution and created a new national smart locker network, named Locky [37]. Locky aims to complement CTT's national delivery network by reinforcing their positioning in the online retail market, and adding more flexibility and efficiency to their delivery processes, strengthening their relationship with customers.

With the continuous rise of online purchases, the demand for smart lockers will likely increase [64]. On the one hand, maintaining a large number of lockers purchased at different times poses some problems, such as the difference in hardware capabilities, due to the hardware replacement life-cycle, and also due to hardware shortages and

costs. On the other hand, there is the need to have a common software base for some common features, such as authentication, across different hardware versions and hardware solutions. These previous needs are driving a change in software architecture, away from a monolithic approach to a more microservices-centric one, forming a centrally coordinated IT infrastructure [11], with containerised services for portability. This is proven to be challenging as it requires the use of orchestration runtimes for the service containers, and hardware has memory limitation constraints, amongst others. Our lockers face the same issue, being the edge nodes of a monitoring infrastructure based on centralized services that are replicated to the lockers, most of the currently available frameworks do not take the locker hardware constraints into account. In order to decide how to implement this architecture, multiple factors need to be taken into account, such as resource management in the physical lockers, including per company locker allocation and communication between local and back-end.

As of early/mid 2024, Locky has already changed their locker architecture from monolithic to a more distributed architecture, deploying a solution built on top of SpringBoot. This solution is very inflexible however, relying on a launcher service to deploy the remainder of the locker services, which makes any maintenance and service upgrades cause an entire system downtime.

1.2 Motivation

In this project, we aim to demonstrate the capabilities of distributed computing on smart lockers as edge nodes in a monitoring infrastructure, via the conception, implementation and testing of a prototype management infrastructure for the Locky locker system. By developing a more flexible infrastructure, we aim to achieve the following:

- Provide the ability to scale the services seamlessly if required;
- Allow for faster implementation of business related changes without provoking large downtimes.
- Be comparable or better than the performance of the current system;

Although not specifically about smart lockers, a lot of research has been done for distributed computing in edge systems to support this project. Of the multiple distributed architectures, we place an emphasis on SOA [30], EDA [68], and microservices [28], as 3 architectures which have been highly studied as edge architectures in multiple scenarios, as well as serverless [8], which although more recent in expanding from cloud to edge, it shows great potential. Thus, we also aim to explore these architectures in our smart locker prototype in order to present new framework possibilities for edge nodes.

1.3 Proposed Solution

In this work, we discuss and present system proposals based on two main architectural solutions: (i) Event-Driven Microservices Architecture; (ii) Serverless Edge Architecture. As already mentioned, Locky's in-production system is built using the Spring framework and SpringBoot, but is very inflexible and complicated to work with and maintain due to tightly-coupled services. Thus, for Event-Driven Microservices, we present a system built on top of an open-source runtime system initially developed by Microsoft, named Dapr, and for Serverless Edge, we develop a function workflow on top of the newly developed OpenFaaS variant, *faasd*.

Considering that the lockers have limited resources, such as limited RAM, CPU capacity and storage space, the frameworks/runtimes of these proposals were first tested on our hardware, a Raspberry Pi 3 Model B, which is also used in the production lockers. From this test, we choose *faasd* to be the base for our software, which we then develop into a complete smart locker system prototype, with an event workflow, orchestration platform and metric web application, to run and manage our prototype infrastructure. Finally, we perform a comparison between *faasd* and SpringBoot using our prototype.

1.4 Contributions

The main contributions of the development of this project include:

- Exploring different distributed architectures in an edge system, namely containerization architectures and serverless, testing performance and resource occupation, while taking into account ease of coding and maintenance;
- Implementing a smart locker system prototype.
- Development of a research paper, accepted at *16th edition of CENTERIS - International Conference on ENTERprise Information Systems*, titled *"A Serverless Approach for Resource-constrained Smart Locker Networks"*.

1.5 Document Structure

The remainder of this document is organized in the following chapters:

- In Chapter 2, we detail the background of the work, including information about distributed architectures, the concept of Edge Computing, and the process of orchestration, finishing with presenting the related work.
- In Chapter 3, we analyse the project at hand, establish a problem statement, and define the testing environment for our solution.

- In Chapter 4, we present two solution proposals and early testing on our test environment to choose the most appropriate.
- In Chapter 5, we dive into the development process of our locker service prototype, including the database schema, services developed, workflow, and observability.
- In Chapter 6, we evaluate our prototype against a demo version of the aforementioned SpringBoot system, and discuss on the the differences between our prototype and SpringBoot.
- In Chapter 7, we make an overview on the work developed, main difficulties, points to improve and future work.

2 Background and Related Work

IT infrastructure is the base foundation of budgeted-for IT capability (both technical and human) shared throughout the business in the form of reliable services that are centrally coordinated [32, 67]. In this project, we will be developing, and implementing, a management infrastructure for CTT lockers, which consists of a back-end monitoring services, and a distributed network of lockers, each equipped with a local device. To create a powerful and resilient concept, we first must study existing architectures and strategies available to implement such an infrastructure, their pros and cons, and how they fare in our specific scenario. Thus, in this chapter, we: (i) Make an analysis of relevant distributed architectures that fit this project, which includes architectures that involve different programming approaches and challenges (section 2.1); (ii) Explore the concept of Edge Computing for the local devices (section 2.2); (iii) Inspect different frameworks and runtimes that can provide automation to configuration and deployment (section 2.3); (iv) Mention different tools that can provide observability as part of a monitoring infrastructure (section 2.4); (v) Finally, we look into investigations and projects similar to this work (section 2.5).

2.1 Distributed Architectures

With the advent of cloud computing, businesses have expanded, leading to a globalization of their operations and the implementation of highly available systems with a global reach [41]. With this, the traditional monolithic architecture is being replaced by more scalable and efficient distributed architectures. These architectures see multiple components, or nodes, communicating and coordinating with each other in order to achieve a common goal, fulfilling their roles isolated from the others. The implementation of these distributed architectures is a challenging task, as it requires both a different approach to software development, as well as considering new problems

related to containerization and service communication, such as Byzantine faults, concurrency and consistency. In this chapter, we will take a look at different distributed architectures, and how they can play a role in our project.

2.1.1 Service-Oriented Architecture - SOA

Service-Oriented Architecture (SOA) is a distributed architecture that focuses on the creation of independent, coarse-grained services. These services encapsulate one or more business capabilities and are designed to be loosely-coupled from each other, communicating with each other to obtain the necessary information or data. A key factor of communication in SOA is its interoperability, where information is shared seamlessly, and based on well known protocols such as SOAP and RPC over the usage of custom, incompatible communication. The development of SOA architectures is started from a business standpoint, where the needs of the business decide how the technical details will be used and developed, prioritizing long-term strategic business goals ahead of project-specific benefits [19].

Another fundamental aspect of SOA is flexibility. With business requirements being ever-changing, the services must be able to accommodate these changes without much effort or impact in the rest of the system, including adding, changing or removing any given capability of the service [35]. This aspect promotes the development of systems, that are built over SOA, to be built with the aim of evolutionary refinement of the system, and not to search for perfection of the system right from the start [19, 27]. These fundamental aspects, and others, have been written and agreed upon as the bases for SOA, contributing for its success.

Following these principles, a lot of different patterns were developed over SOA [58]. The event pattern, which became the Event-Driven Architecture, placed a bigger focus on asynchronous communications, and the Microservices Architecture, which is an evolution of SOA, provides more scalability and granularity.

2.1.2 Microservices Architecture

The microservices architecture, much like SOA, focuses on the creation of services as a representation of the business logic, but instead of being coarse-grained, these services are implemented to perform a specific business functionality, with the system being built as collections of these fine-grained, independently deployable, and highly scalable services [57], Figure 2.1. In the microservices architecture, each service is designed to have their own data storage if necessary, and the communication between services is made with protocols such as REST, with well defined and simple APIs.

Microservices has already established itself in the world of computer science due to how it addresses and solves many of the challenges of the monolithic architecture, and

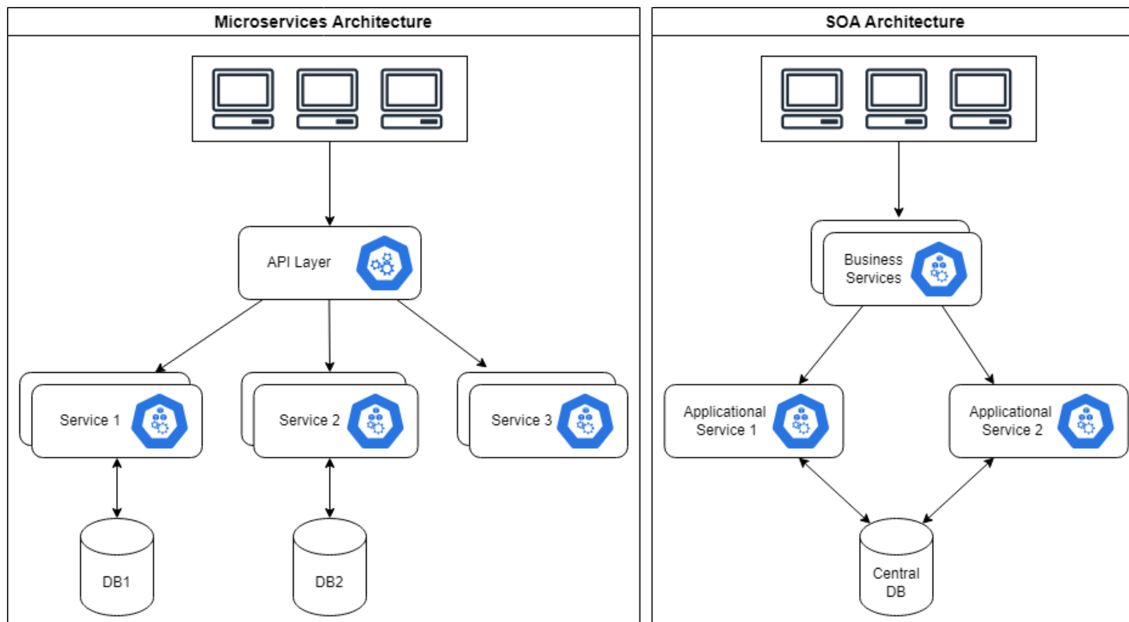

Figure 2.1: *Microservices architecture scheme versus SOA architecture scheme*

together with its high scalability and flexibility, it has seen more and more widespread adoption in the business world. In a survey conducted in 2020 by O'Reilly [38], 28% of respondents answered that their organizations have used microservices for at least 3 years, with an extra 61% answering that they have been using microservices for 1 year or more. For those who are still migrating their systems, over 50% are migrating to microservices. However, as mentioned in the introduction of this chapter, the implementation of distributed computing systems is a challenging and complex task. As such, in this same survey, when asked about the rate of success in the migration to microservices, 46% reported a mostly successful outcome and 9% a complete success, with 8% being no success at all. This rate success was highly attributed to features of microservices such as flexibility, responding quickly to changing technology and business requirements and better scalability, while lack of success, and the biggest challenges, came from culture mindset and the decomposition of the already developed systems into microservices.

2.1.3 Event-Driven Architecture - EDA

As previously established, the Event-Driven Architecture (EDA) is an extension of SOA using a set of relatively independent actors who communicate events amongst themselves [39]. In this architecture, services and applications communicate by publishing and consuming events, which are triggered when there is a change in the system. It is through these events that data is distributed across the system, leading to a real-time asynchronous data distribution inside the system, with its services being loosely-coupled.

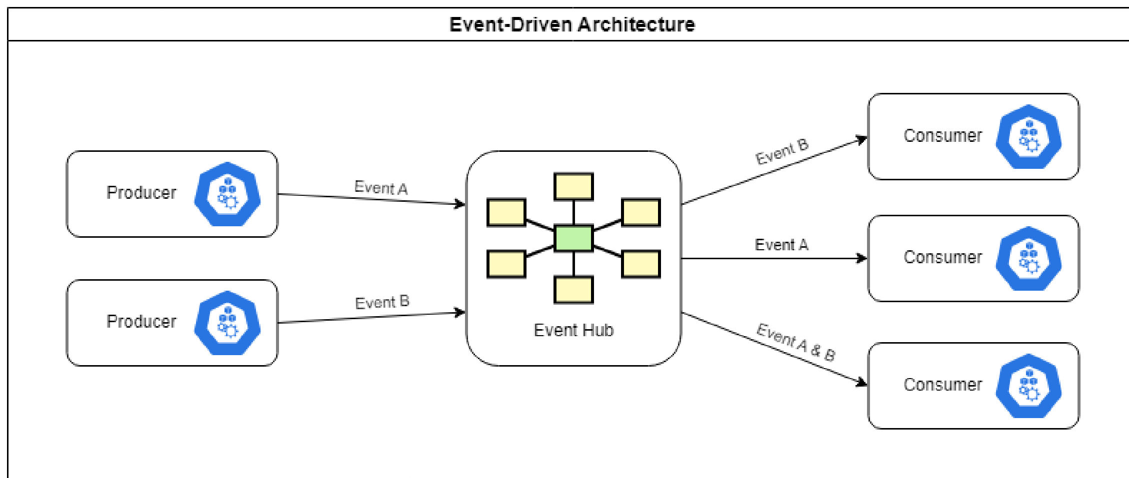

Figure 2.2: *Event-Driven Architecture scheme*

In EDA, Figure 2.2, the services of a system are categorized in 3 types, event produces, event hubs, and event consumers. The event producers are the services that know when certain changes in a system categorize as an event, and publish that event along with the necessary information and data to the event hub. The event hub receives the events sent out by the publishers and notifies the services that registered their interest in that event, those being the event consumers.

The advantages of using EDA include easier and real-time data processing from multiple sources, since that data is distributed across the system through events and event hubs, and the services are easier to scale due to being loosely-coupled.

2.1.4 Serverless Architectures

So far, we have talked about 3 different distributed architectures, talking about their principles, and the way they work. With the Serverless Architectures, the paradigm changes. While all of the above architectures started with local deployments, that require containers to isolate services and dependencies, and then expanded to cloud, the Serverless Architectures have their services directly hosted by a cloud provider, with automatic scaling [55]. Therefore, the developers write and deploy services, while the cloud providers host servers to run these services at any scale, including databases and storage systems. In short, the Serverless Architecture is designed to allow the developers to focus on building the services and business logic, relying on automation and cloud providers to handle the underlying infrastructure.can be grouped into the group of container architectures [45]

Serverless architecture, Figure 2.3, can be seen as a combination of Function-as-a-Service (FaaS) and Backend-as-a-Service (BaaS) [36]. In FaaS, the business logic is implemented as a set of discrete functions, each performing a specific task. These functions are triggered by events, and the cloud provider executes the functions in

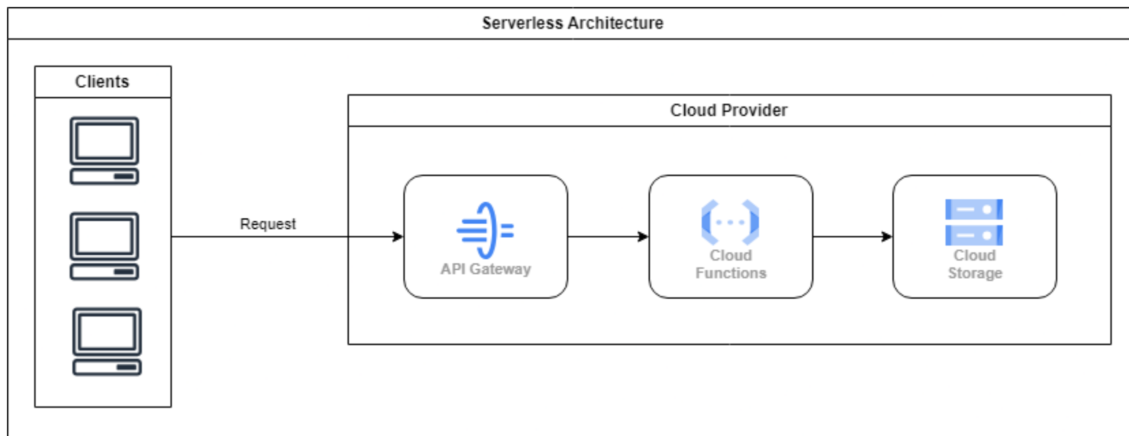

Figure 2.3: *Serverless Architecture scheme*

their provisioning servers, thus greatly reducing the costs of paying for servers or VMs, with instances of these functions being created and destroyed based on incoming traffic. In BaaS, only the back-end management infrastructure is abstracted to cloud retainers, such as databases, file storage and analytics. FaaS and BaaS can also be used independently, together with a microservices architecture for example, to form an hybrid architecture, depending on the system requirements.

2.2 Edge Computing

In order to build our concept of a management infrastructure for the CTT lockers, we first took a look at several distributed architectures, which will be the core architecture of our services. Thus, in this chapter, we will talk about the integration of the local devices to the core architecture, and its major challenges.

As presented previously, these local devices aren't just made to store information, they also process information and control their respective lockers, sending and receiving information daily, or even hourly, so they must be well integrated with the back-end infrastructure, and at the same time be optimized to use as little resources as possible, both in memory and in energy. This distributed computing paradigm is called Edge Computing.

In general, the purpose of Edge Computing is to bring the computation and data storage closer to the source of the data itself [60], in order to improve response times and save bandwidth. Edge Computing originated from Content Delivery Networks (CDN) [15]. These networks were created in order to distribute web and video content across the internet, but to do that effectively, the content was replicated across edge servers that were deployed closer to the end users. Though CDNs can be constructed using several different architectures, edge servers are included in all of them. In our project, the local devices of the management infrastructure are the edge servers,

located closer to the physical lockers, that need the data of, for example, the parcels, to operate. However, considering that the devices, or our edge servers, are distributed in multiple different locations, new challenges emerge.

The physical and geographical distribution of the devices implies that data has to be communicated to and from the back-end through the Internet, meaning that information needs to be encrypted with using special mechanisms, which in turn might increase latency in communication. Together with this, the reliability of the devices needs to be assured, in order to prevent both any major errors and keep the service alive during connection down-times. Finally, the efficiency of the code being executed and the way the data is stored on the device needs to be well designed, in order to prevent any memory or energy problems, which could lead to the device failing to communicate or completely shutting down. Planning the development of the edge architecture accordingly will increase the up-time and reliability of the complete infrastructure. Of the aforementioned distributed architectures, SOA, EDA and microservices are the most used distributed architecture in edge systems [28, 30, 63, 68], while serverless, being a cloud-oriented architecture, has seen less research up until recent years. In light of this, we will next look into the concept of serverless edge, and how it is a architectural possibility for our project.

2.2.1 Serverless Edge Paradigm

The discussion of serverless edge computing as a real use case, started to rise in 2017 [7], as the scenario where code could/should be executed outside the cloud data centre started to be considered. One of the main possibilities discussed, was to include serverless support in IoT devices and associated computation at the edge. With this motivation, efforts were made to extend the serverless paradigm to the world of edge computing.

Baresi, L. and Mendonça, D.F. [8] in 2019, picked up on this discussion and reflected on the possibility of a serverless platform for edge computing. The authors began by identifying possible characteristics of a serverless edge platform through reference scenarios. These include (i) Low-Latency Computation Offload for computation-intensive tasks; (ii) Inter-Platform Collaboration between multiple edge nodes running FaaS platforms; (iii) Latency Optimisation when creating function workflows; (iv) Opportunistic Data Analysis and processing at the edge, preventing large volumes of data to be sent to distant cloud servers. The authors identify key requirements and challenges for the development of these characteristics, proposing solutions for their resolution. This analysis served as a base for the authors to develop a prototype for serverless edge platforms, taking all the characteristics into account, built on top of

an adopted version of the open-source project Apache OpenWhisk [3], a FaaS platform. The authors performed two experiments, using Linux machines with 16 GB of RAM, and measured the memory footprint of all components in the prototype. The total memory footprint was measured to be about 2.2 GB in idle, which the authors reflect that still needs further optimisation. On a positive note, the authors say that overall, the obtained results corroborate the benefits of the serverless architecture in: (i) Drastically reducing the burden of infrastructure management; (ii) Allowing more functionality to be deployed on fog nodes with limited resource; (iii) Fulfilling the requirements of different application scenarios and heterogeneous deployments of edge nodes. The authors hope that this prototype will motivate future research into the development of serverless at the edge.

Similarly, in 2021 Aslanpour et al. [4] identified, through an extensive literature review of serverless and edge computing, the essential components of such an infrastructure, discussing how it is placed in the cloud and edge. This infrastructure is split into three layers of components – computation, execution and coordination. In the computation layer, we have storage, networking, and functions computing; the execution layer includes the execution engine and monitoring; and finally, the coordination layer is where the controller is located, together with schedulers and queues. In typical cloud serverless, the serverless infrastructure is housed in a single cloud data centre, handling all functions. However, in edge computing, each edge node has its own individual infrastructure, only executing functions related to whichever devices it controls. The authors describe how the serverless edge computing paradigm offers a wide range of opportunities for edge computing. These include the ability to handle greater amounts of workload and data by scaling functions up, save resources such as RAM and battery in low traffic scenarios by being able to scale functions down to zero, and a serverless, stateless life-cycle that is more compatible with the event-driven scenario typically seen alongside the edge. Conversely, the authors also highlighted several open issues that still need to be addressed, such as function cold starts which add a delay to response times, reliability and fault tolerance issues, and a lack of simulation tools to test the developed functions. Evidently, the implementation of a serverless approach in edge computing has its merits. Serverless seamlessly tackles resource constraint issues commonly seen in IoT and edge, and its model of stateless functions with variable scalability allows for plenty of flexibility in architectural design and resource management. However, before we can accept serverless as a sustainable edge solution, we must first tackle the many open issues serverless edge is facing. To work through these issues, many tools and frameworks have been developed to support serverless functions in edge environments, promoting serverless edge and advancing the research in the area.

These developments can be divided in two categories, cloud-hosted and self-hosted.

A cloud-hosted framework example is AWS Greengrass [5], a framework which can be used in conjunction with the serverless platform AWS Lambda [6], to achieve edge integration. However, while cloud-hosted solutions show consistent performance results, connection to the cloud is still required, which creates extra latency in operation and may lead developers into choosing more traditional edge solutions, e.g. Spring services, over serverless. Given that smart lockers require quick reactions to new events and interactions, self-hosted solutions are more fitting.

2.3 Orchestration Runtimes

When deploying distributed computing systems, we need to consider how they will be hosted, and how they are going to communicate with each other. Here is where Orchestration comes in. Orchestration, in system administration, is the automated configuring, coordinating, and managing of computer systems and software [20]. Some modern examples include SpringBoot, with automated configuration for Spring based applications, and Docker, for containerization and hosting. Orchestration frameworks have kept evolving in recent years, with frameworks such as Kubernetes and Ansible, which are able to further automate server configuration and orchestration, working over Docker and/or using YAML for better readability and comprehension. Recently, the development of these tools is reaching an even higher level of abstraction, allowing for the easy integration of components of any kind, be it for communication, data storage or metric gathering. In this section, we will analyse 3 tools for orchestration that are capable of supporting the requirements for our project. This analysis will also include the requirements to use these tools and other details, such as programming languages supported, for a comparison purposes.

2.3.1 Dapr - Distributed Application Runtime

Dapr, or Distributed Application Runtime, is a portable and event-driven runtime for building distributed computing applications on the cloud and edge. It provides developers with a set of APIs, named building blocks, that work to abstract away the complexity of creating distributed computing applications, such as service invocation and state management, and allowing the focus to stay on the development of the business logic [23]. The application code can access the APIs through the Dapr sidecar, that runs on a separate container or process, using HTTP or gRPC, supporting typed language. In total, Dapr provides the following building blocks for distributed computing applications:

- Service Invocation - which handles service-to-service communication through well-known endpoints, using HTTP or gRPC, using name resolution for service discovery;

- State Management - providing a key/value-based state preservation to the system;
- Publish/Subscribe - allowing for services to publish messages to a topic where subscribers can subscribe;
- Bindings - which is a bi-directional connection to an external service, such as databases and queues;
- Actors - that work as reusable code and data objects with single threaded execution;
- Secrets - which is an API to access secrets, such as the connection string to a database, in secret stores located in the cloud, locally or Kubernetes;
- Configuration - serving to retrieve and subscribe to configuration items;
- Distributed Lock, allowing multiple instances of an application to access the same resource without conflicts and guaranteed consistency;
- Workflows - enabling the orchestration of tasks within the system. The workflow API can also use other APIs, such as service invocation, if needed;
- Cryptography - providing the encryption and decryption of data.

Dapr also allows for the creation of pluggable components, which are components not included in the runtime. These can be developed in any gRPC-supported language, and are registered with Dapr via Unix Domain Sockets, using gRPC. The pluggable components need to be started manually, and will run in a distinct process, separated from Dapr itself. This adds value to the system by providing building blocks that depend on pluggable components [23]. The current supported APIs for pluggable components, which include a gRPC definition, are: (i) State stores; (ii) Pub/Sub; (iii) Bindings; (iv) Secret stores.

2.3.2 Knative

Knative [33] is a platform-agnostic solution for running serverless functions on top of Kubernetes. It is composed of 2 main components: serving and eventing. Knative serving defines a set of objects as Kubernetes Custom Resource Definitions (CRDs) which will be used to define and control the serverless workload. Serving can enable rapid deployment and automatic scaling of containers based on demand. Knative eventing is a collection of APIs that enable the use of Event-Driven Architecture in the serverless infrastructure. The APIs are used to create route events from producers to consumers, using HTTP requests for communication. The main components of Knative are compatible with edge and using Knative on top of a lightweight solution of Kubernetes, such as k3s, allows for the development of edge solutions in low-resource devices. Knative serving can scale functions to 0, granting a low-resource footprint, and Knative eventing makes the detection of events from IoT devices integrate seamlessly with the serverless functions running on the edge.

2.3.3 OpenFaaS

OpenFaaS is a framework that makes it easy for developers to create serverless functions [43]. The main component of OpenFaaS is its API gateway, which is where all the other components connect to. Access to the functions deployed on the orchestration engine, i.e. Kubernetes, is provided through the gateway, and from there, tools such as Prometheus can collect metrics on the function usage and performance, and automatic scaling can be enabled based on traffic by interacting with Kubernetes. In terms of function development, OpenFaaS provides a set of function templates for multiple languages, namely CSharp, Go, Java, Node, Python, Ruby and Rust, available through the OpenFaaS Command Line Interface (CLI) *faas-cli*. It is through this CLI that we create, list, deploy and remove our functions.

In order to increase support for edge solutions, OpenFaaS developed a variant of itself called faasd [21], which removes the cost and complexity of Kubernetes. Running on a single host, faasd uses the containerd system process and Container Networking Interface (CNI) alongside the same core components of OpenFaaS, turning the edge device it is running on into a lightweight server for hosting OpenFaaS functions, only losing out on the automatic scaling feature.

2.3.4 SpringBoot

As mentioned previously, SpringBoot is an orchestration framework for automatic configuration of Spring-based Java applications [61]. It follows a convention-over-configuration approach [14], providing default configurations to reduce the need for manual setup. To host services, SpringBoot works with embedded servers like Tomcat, Jetty, or Undertow, to allow services to run without needing an external server. Additionally, SpringBoot automatically configures services based on the dependencies that have been added through the use of Gradle. It also offers a number of additional features such as metric monitoring, health checks, and externalized configuration to help manage applications in a production environment. As for language support, SpringBoot supports both Java and Kotlin applications.

The Spring framework is, by design, a lightweight framework, made to work in both cloud-based applications and edge devices for microservices development. SpringBoot elevates the quality of Spring by providing automated configuration, making this an attractive framework for our project.

2.4 Observability

Another very important aspect of this work is Observability. Observability, in distributed computing systems, is determined by how much of the infrastructure, the services and

their interactions can be monitored [50]. Since our work involves the development of a monitoring infrastructure, the degree of observability we can reach in our services becomes a critical point of consideration. In fact, monitoring and the observability of distributed computing systems is not purely a technical issue but a strategic topic, critical to the success of a company which offers the services [42]. With this in mind, in this section we will analyse 2 tools for metric extraction, and 1 tool for displaying metrics, all open-source.

2.4.1 Prometheus

Prometheus [52] is an open-source monitoring toolkit for all environments, from Cloud to Edge. Prometheus main functionality is to collect and store time-series data, which is identified by metric names and key/value pairs, known as labels, from instrumented jobs we configure it to pull metrics from. To query and visualize this data, Prometheus provides an interactive interface, available in the /graph endpoint. Finally, for data storage, Prometheus keeps all of the currently in-use chucks data in memory, with long-lived memory allocation, with a default garbage collection target percentage [22] of 40%. Still, even with in-memory storing, Prometheus can handle around 200000 time-series simultaneously. The most up-to-date time-series scraped by Prometheus is available in `/metrics` and can either be fetched by advanced visualization tools to display the data in user-friendly graphs, or we can configure Prometheus to push these metrics to a remote endpoint.

2.4.2 Graphite

Graphite [25] is an open-source monitoring tool designed to track and graph time-series data. It consists of three main components: (i) Carbon, which receives and stores metric time-series data; (ii) Whisper, a database system for storing this data; (iii) Graphite-Web, a web application for querying and visualizing the data. To use Graphite, developers must write an application that collects numeric time-series data and send it to Graphite's processing backend, Carbon, which stores the data in Graphite's specialized database. The data can then be visualized through graphite's web interfaces. Unlike Prometheus, Graphite stores all data in disk, with each time-series being stored in a unique file. Under load, Graphite needs to perform a very significant amount of I/O operations, which would normally impact the real-time accuracy of the data being displayed in the web applications. To circumvent this issue, when a graph re-renders, Graphite fetches data from both the stored files and the pre-storage cache, ensuring that data displayed is always in real-time.

2.4.3 Grafana

Grafana [24] is an open-source advanced data-visualization platform. Much like Prometheus and Graphite, we can query, visualize, and explore metrics. However, in Grafana, we can merge data from various data sources into a single dashboard. These data-sources can be time-series databases, e.g. Prometheus and Graphite, SQL databases, and/or cloud services. The dashboards also have a higher level of customization, where we can add and remove graphs, setup more complex alerts, and create events. To use Grafana, we can be host it locally for custom configuration of the interface and endpoints, or in the cloud, which allows for the use of interface templates and automatic configuration of endpoint fetching, with Prometheus and Graphite integration.

2.5 Related Work

In this section, we detail several papers and tools produced to try and both integrate cloud with edge computing in a simpler way, as well as attempting to improve edge performance via testing and measuring metrics.

2.5.1 Lightweight Kubernetes Comparison

As containers gain widespread adoption as a method for software deployment, there is a growing interest in utilizing container orchestration frameworks not only within data centers and on back-end systems, but also on hardware with limited resources, such as Internet-of-Things devices. As a consequence, multiple distributions of lightweight Kubernetes have been developed, but choosing which distribution to use is always a challenge. Thus, this paper [34] focuses on a comparison between MicroK8s, k3s, k0s, and MicroShift, investigating their minimal resource usage as well as control plane and data plane performance in stress scenarios.

For this comparison, the authors first established a template with questions and metrics to be measured for each distribution, using Microsoft Azure VMs as a testbed for the distributions. 5 VMs were created, with: 3 running on Ubuntu for MicroK8s, k3s and k0s, each with a multi-cluster configuration; A 4th running on Red Hat for the MicroShift distribution because there wasn't any Ubuntu installation packages available. Additionally, since MicroShift doesn't have multi-cluster support, the measurements were performed this single node; And the final VM being the central, data gathering node, aggregating the metrics to evaluate the distributions.

In sum, the authors confirmed that all lightweight distributions tested were suited for edge devices with a minimum of 1-2 GB of RAM, asserting that, in stress scenarios,

k3s and k0s marginally showed the highest control plane throughput, while MicroShift achieved the highest data plane throughput.

2.5.2 Blueprint Toolchain

Blueprint [1] is an extensible compiler for microservices applications, along with a collection of off-the-shelf microservices benchmark applications. It was developed based on the need to iteratively configure, build and deploy microservices applications, most commonly seen when creating prototypes or experimenting with microservices architectures, with the central goal of reducing the amount of effort involved when changing and re-compiling microservices architectures.

Applications developed in Blueprint consist of two key components: workflow spec and wiring spec. A workflow spec defines the core business logic of an application, and the wiring spec instantiates services, connects them together, and specifies how they should be modified, configured, and deployed. The Blueprint compiler then implements RPC code for the services and other mechanisms for service-to-service communication if necessary, in a way similar to how orchestration runtimes deal with service-to-service communication. These components thus provide a more clear separation between code and infrastructure, facilitating the design, implementation and reconfiguration of microservices systems.

2.5.3 Kubernetes k4.0s

Kubernetes k4.0s [9] is a proposal of a Kubernetes-based implementation tailored for Industry 4.0 [62]. It is founded on the need for a mixed criticality [10] container orchestration system, due to the need to consolidate different functionalities on the same edge node with different levels of priority and requirements.

In k4.0s there are 2 types of nodes: master and worker. The master nodes form a control plane architecture, being able to decide actions for the worker nodes to perform, scheduling actions if necessary, and can track the global state of the worker nodes, including resources used during execution. The master nodes can work alongside each other by keeping a consensus on the system state. The worker nodes form a compute cluster architecture, running jobs and collecting health information about them and about the worker node itself, forwarding the information to the control plane.

The authors, with the creation of k4.0s, argue for the need of new monitoring strategies for workloads, taking into account the criticality of workloads due to the rise of Industry 4.0 and edge/fog computing.

2.5.4 faas-sim - A trace-driven Function-as-a-Service simulator

faas-sim [54] is a simulation framework tailored to serverless edge computing platforms. It can be used to develop, and evaluate the performance of operational strategies for such systems, like scheduling, auto-scaling, load balancing, and others.

The domain model of faas-sim focuses on 3 concepts, Functions, the FaaS system and Function simulators. A function is the highest level of abstraction and simply refers to some business functionality that is identified by a name, and can be invoked through it. The FaaS system is the high-level interface a client can interact with, being able to control any function created by deploying, invoking, removing and scaling, amongst other possibilities. Finally, a Function simulator encapsulates the simulation code for a function, using traces from real functions and workloads as basis for the simulation, in order to test the performance of a function and how it handles stress tests over the network, to the edge device.

2.5.5 LaSS - Latency Sensitive Serverless

LaSS [66] is a platform for managing Latency Sensitive Serverless computations in edge, built on top of Apache OpenWhisk. The authors present a principled approach for allocating edge resources to latency-sensitive serverless functions, using this approach to design and implement an algorithm to dynamically scale the resource allocation of each function up or down based on time-varying workload demands. When the system is not under load, LaSS dynamically allocates the necessary resources to each function based on its observed workload, and when it is under heavy load, LaSS uses a weighted fair-share resource allocation approach that guarantees a minimum fair share of the edge cluster resources to each function in proportion to its weight. The authors show that the models being LaSS can react quickly to load fluctuations, and that, when stopping containers, their resource deflation policies [59] can improve resource utilisation by 6% compared to the typical termination policy.

2.5.6 Kappa - Serverless IoT Deployment

Kappa [49] is a serverless deployment concept for IoT, developed on top of the distributed IoT framework Calvin [48] due to being fully distributed and not requiring the cloud for its execution, and tasks can be executed anywhere, including edge nodes. Calvin applications are based on the actor model, implemented as re-usable units and written in Python, and the dataflow model, expressed using the purely declarative CalvinScript language. Kappa is divided in 3 layers, these being the frontend, backend and IoT layer. In the frontend, the user can create, communicate and delete kappas. The backend contains the Calvin runtime, which interprets the user script from the frontend and compiles a new script with the current actor workflow for the Calvin runtime.

Finally, the IoT layer is where all the kappa actors are running. These actors, named kappas, have 4 categories: (i) input/output kappa; (ii) input only kappas; (iii) output only kappas; (iv) no input/output kappas. The categories are such so that the different events of an IoT system are executed by the most fitting kappa. For example, an event that are triggered by timers to execute some individual periodically task should be implemented in a no input/output kappa.

3 Project Analysis

In this chapter we will start presenting the infrastructure of the in-production Locky infrastructure, and how the services are organized across back-end and local devices. Next, we define a problem statement, detailing the primary objectives and challenges for the development of our infrastructure. Lastly, we establish an evaluation process to compare our infrastructure to a simplified version of the production infrastructure.

3.1 Locky Infrastructure

The Locky infrastructure consists of a central cluster, with multiple services for the different business requirements and a central database accessed by them. Locally, Locky includes a single device programmed to maintain the locker, with a local database containing only relevant and updated information about the parcels going in and out of the locker. The infrastructure is going to be deployed using a distributed architecture, which also contains the advantage of monitoring the network, values that could be useful for data analysis.

In the infrastructure presented in Figure 3.1, the local architecture on every device is the same, with the DB only containing the relevant data for the lockers the device controls. This data will be fragmented and replicated from the back-end cluster, with only each locker receiving the data it requires for operation. This is ideal for data manipulation purposes [13] because in our scenario, the locker system can be constantly updated with new parcels, and give updates back to the back-end accordingly as each parcel is delivered to the locker, and picked up by the customer, without collisions. The local architecture will also include a service for communication with the back-end and a service for peripheral interaction, such as the locker doors and the screen user interface. The communication between the back-end and the local device is done via HTTP requests to pre-assigned endpoints. The path of the request URL is used to specify the service, while the query option specifies the action. Requests are always

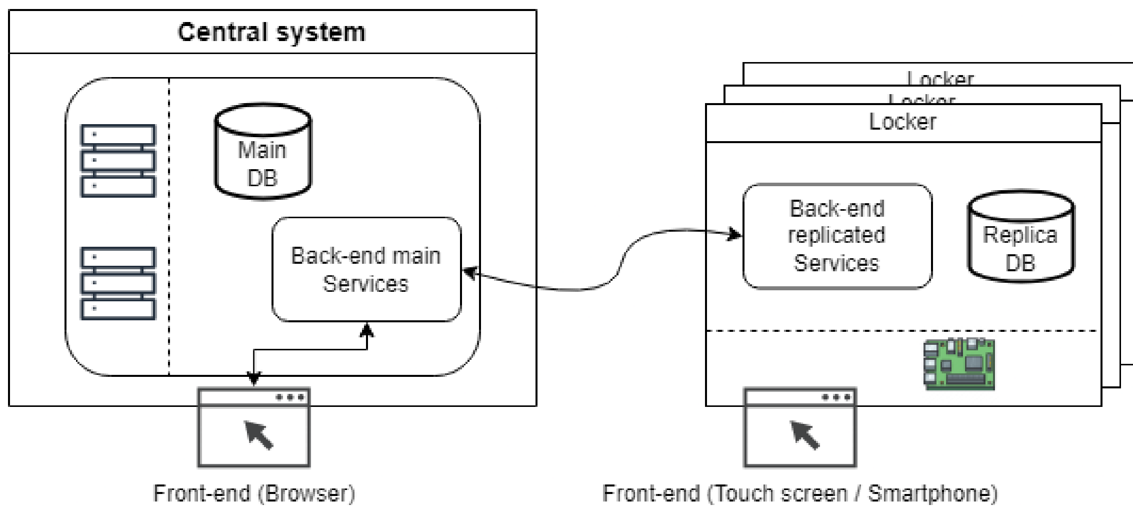

Figure 3.1: *Simplified design of the Locky infrastructure*

HTTP POST, with the body containing the the information to be processed. As for the back-end architecture, it has a single central DB, multiple services for the business requirements of the application, an API layer for downstream and upstream communication with the back-end cluster, and a monitoring web-application, which receives metrics such as CPU usage and memory from all of the lockers, and displays it in easy-to-read graphs.

During the making of this project, the hardware of each locker on the locker network was being updated from a Raspberry Pi 3 Model B to a Raspberry Pi 4 Model B. The most noticeable change coming from this upgrade is a RAM increase from 1 Gb to 2 Gb, which severely reduces the memory constrains for distributed computing frameworks. A full comparison between both hardwares can be found in Table 3.1. Additional specification can be found in the Raspberry Pi documentation [56].

Table 3.1: Raspberry Pi 3/Raspberry Pi 4 comparison.

	Raspberry Pi 3	Raspberry Pi 4
Processor	BCM2711	BCM2837
Clock Frequency	1.2Ghz	1.8Ghz
RAM	1 Gb	2 Gb
Bluetooth	4.1	5.0
Power Supply	2.5A	3A

3.2 Problem Statement

As stated previously, there are some problems that need to be addressed when developing a management infrastructure for distributed lockers, many of these being

IoT and edge computing inherent. The problems related to different hardware versions and solutions can be solved with the use of containerisation, which is used by many distributed and serverless frameworks. However, since the software base for essential locker functionalities needs to be executed locally in its entirety, the hardware resources, which are extremely limited, become even more problematic to manage, and with many frameworks not assuming these hardware constraints, the options become quite limited in choosing the supporting framework.

Besides the framework, it is also important to define the locker and its workflow during operation. The locker is composed by a set of several differently sized doors, each door being able to contain one (1) parcel at most. It has a touchscreen, which the mail courier and customer use to interact with the locker to deliver and retrieve parcels respectively. To separate mail courier from customer interactions, a set of three (3) keys corresponding to each parcel are generated during its creation, encrypted and sent to the locker. These are the delivery key, retrieval key, and emergency key. The delivery and emergency keys are sent in plain to the mail courier so they can deliver the parcels to the lockers and remove any parcels in case of a mistake, and the retrieval key is used by the customer to retrieve their package. Finally, to open any door, the locker must first verify if the key communicated from the screen matches any encrypted key from the local database. If a match is found, then the locker orders the opening of the door the parcel is associated with.

Another key information that needs to be stored and updated is the parcel status, as both the locker and the back-end need to keep track of the parcel status in order to ensure continuity of service and re-usability of doors. As such, we defined the parcel status to be one of the following: arriving, in storage and delivered. When the delivery key is used, if the operation is successful, then the parcel will change from the arriving status to in storage, and after the retrieval key is used, the parcel will go from in storage to delivered status. The emergency key is an exception, since the use case for this key is only when the mailman accidentally inserted the incorrect parcel into a specific door and needs to change it, so the parcel status will continue to be in storage. After every status change, the locker notifies the back-end so that both systems are synchronized.

Thus, the aim of this work is to develop a prototype, utilising a serverless approach to software development, that fulfils the locker requirements and functionalities. The prototype should (i) Communicate with the back-end cluster to receive the data it requires for service; (ii) Order the opening of doors if the codes inserted in the touchscreen are correct; (iii) Keep the back-end updated with parcel status; (iv) Have all the software base be executed in a lightweight serverless framework; (v) Be comparable or better to a traditional SOA approach in terms of performance. In order to fulfil the final requirement, we will measure a set of metrics of the business logic when supported by an orchestration framework, and compare these values against the same logic,

but using a SOA approach. Other values such as battery usage won't be measured because, as a part of this case study, the locker will always have access to a power source and a running 4G terminal for back-end communication.

System monitoring will be a feature connected to the running locker system, deployed in the back-end, that will monitor the lockers and display relevant information, such as performance, memory usage and deployment status. This data will allow the developers, data analysts and business managers to improve the infrastructure on sections that show poor results, as well as adapting quickly to any problems that might appear during execution.

3.3 Metrics

In order to evaluate the several orchestration runtimes we will be testing, we must first establish a set of goals to be achieved. Following the Goal Question Metric approach model [12], or GQM, we start by defining the goals, specifying purpose of measurement, object to be measured, issue to be measured, and viewpoint from which the measure is taken. Then, we refine each goal into a set of questions, breaking down the major components, and then refine each question into a set of metrics that can be measured, which can be reused in multiple questions. Figure 3.2 illustrates this model.

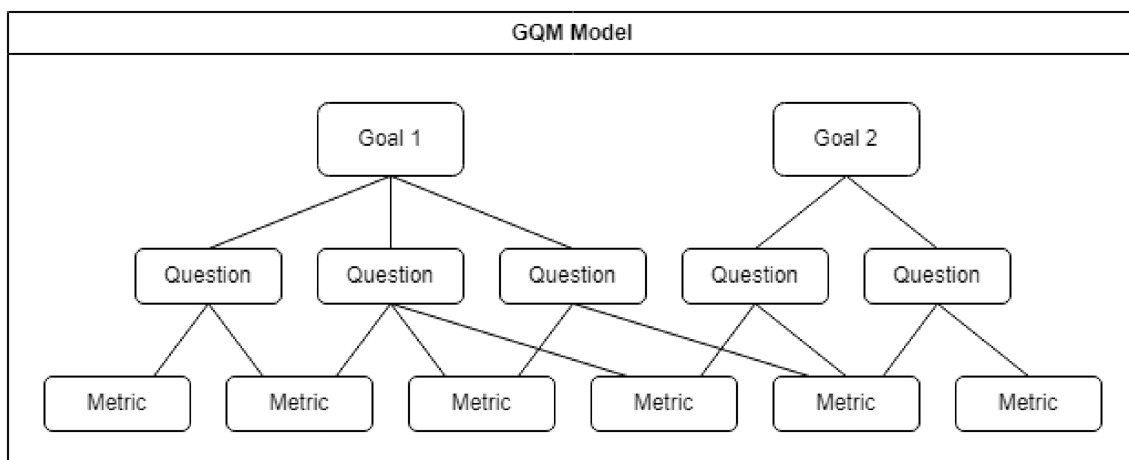

Figure 3.2: *Structure of the GQM Model (adapted from [12])*

Following this model, we created an hierarchical list to highlight the goal for measuring metrics, which will be the basis for the comparison between architectures and between orchestration runtimes.

- Goal: Compare the performance of our infrastructure, with our chosen runtime, against a SpringBoot alternative.
  - Q1: What is the resource usage our infrastructure and SpringBoot?
    - Q1_M1: Average Memory usage;

    * Q1_M2: Measure CPU usage.
  - Q2: What is the performance of our infrastructure and SpringBoot?
    * Q2_M1: Average latency for request resolution.

As already defined previously, the main goal for this evaluation is to compare the performance of our infrastructure, with our chosen runtime, against a SpringBoot alternative. We will achieve this by measuring the performance of each orchestration runtime involved, and compare them based on 2 questions. In Q1, we evaluate the resource usage of each runtime, in which we will both see how the CPU and RAM usage scale with activity, while in Q2, we will measure the performance of each infrastructure by evaluating the latency of request resolution. To define each metric: Memory usage consists of how much Random Access Memory (RAM) is still considered to be free after each individual service deployment, including the hosting frameworks. CPU usage is the amount of CPU resources are allocated for the services, measured during and after deployment for this project. Lastly, latency for request resolution, or response time, is the time it takes for the request to reach the server, the time for the request to be handled by the service/function, and the time for the response to be delivered back to the client.

To obtain these metrics, a Windows Machine was used with a six-core Ryzen 3600 and 16GB of RAM, connected to the same local network as the Raspberry Pi to reduce latency. Through an SSH connection to the Raspberry Pi, the free memory was directly measured from the system data after each individual deployment of a service/function stabilizes, and the CPU usage was measured using the *htop* command [29]. The HTTP requests are generated through Apache JMeter [2], calculating the arithmetic average response time after 10 requests. To access performance of both approaches, the database insertion logic is used as the main testing code for response latency taken by services/functions. The test environment hardware is a Raspberry Pi 3 Model B, with a 1.2GHz 64-bit quad-core ARM processor and 1GB of RAM, running Debian 11 bullseye, which will host all the services/functions.

4 Evaluation of Frameworks for Service Distribution

For the development of a new infrastructure, we must keep an open mind and analyse possible tools in order to find the most fitting ones for the project. With that being said, working with resource limited hardware poses heavy restrictions on which framework/runtime we can choose to support the business logic. In this chapter we present two solution proposals - a microservices approach, supported by the Dapr runtime; and a serverless approach, supported by *faasd*, an OpenFaaS variant. Additionally, we show performance measures both the frameworks on our test environment, and conclude with our thoughts on the compatibility of the infrastructures with this project.

4.1 Dapr Runtime

For our first proposal, we decided to follow the original plan of developing a microservices architecture, implementing similar services to the real system. Thus, we looked into distributed computing runtimes to host the services, while providing ease of code in service-to-service communication, and seamless access to metrics about the services status. Dapr fulfills all these criterias with its modular approach to components, including components for communication and observability, allowing us to remain focused on implementing only the business logic.

4.1.1 Locky with Dapr

When initially designing an infrastructure for the locker system to be supported by Dapr, we decided that the best course of action would be to first replicate the back-end database services, related to the Locky system, to the lockers. This would allow us to do minimal data manipulation, only fragmenting the data so that each locker receives only the information pertaining to itself. The remainder of the locker services, related to locker operation workflow, would be included in the deployment. In short, the locker system would have the following services: (i) N-services, loosely-coupled, one for

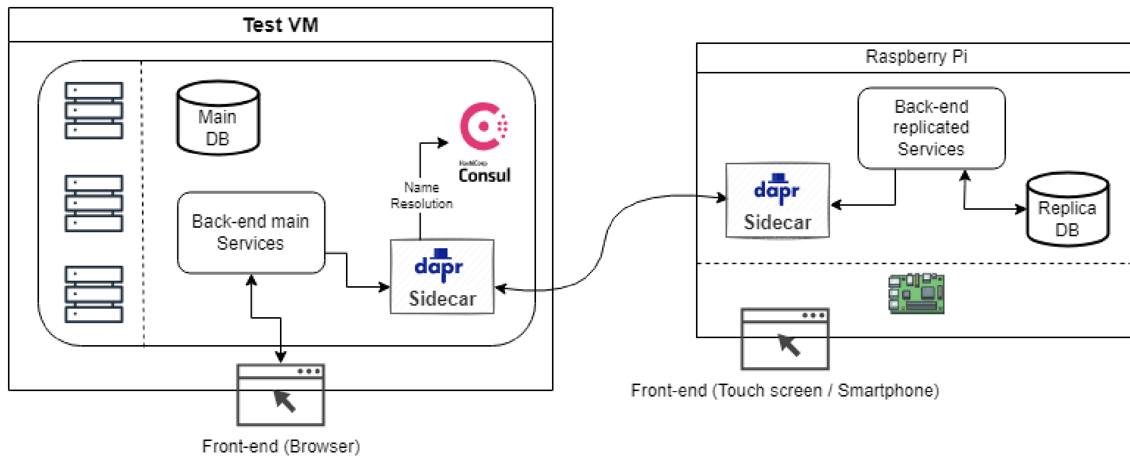

Figure 4.1: *Locker infrastructure with Dapr.*

each table in the database; (ii) One database system, e.g. PostgreSQL, to store the relevant data; (iii) One Dapr sidecar, with all components for service-to-service communication, and the database API; (iv) Any other service required for the locker workflow.

As we had previously discussed, the development of extra code for service-to-service communication, database interaction, and metric observability bloats the code considerably. To avoid this, we make use of Dapr's building blocks to connect the system. We used two building blocks, one to provide service invocation for service-to-service communication, and a building block with an agnostic API for the locker database, which allows us to use, or switch between, any database, without major code changes. Finally, to allow communication between the back-end system and the locker, we make use of the same service invocation building block, assisted by HashiCorp Consul [26] to discover remote services. The complete infrastructure can be viewed in Figure 4.1.

4.1.2 Testing Dapr in Raspberry Pi 3

To properly stress test Dapr in our Raspberry Pi 3, we decided to first test how the Dapr runtime affects the available memory in the system. The approach for this test was the following: ran the command *top* in remote console from the Raspberry Pi, measured the available memory, then started deploying services with Dapr one by one, measuring the memory in between deployments. What we were not expecting however, is that with each service deployment, Dapr launches a new sidecar with all the default Dapr components. This sidecar is not modifiable, so the available memory reduces very quickly, as we can see in Figure 4.2. After 3 services, the available memory was not enough to maintain system stability, with delayed reactions to commands inserted in the open remote console which was being used for deployments. After deploying one more service, the Raspberry Pi stopped responding entirely, ending the test.

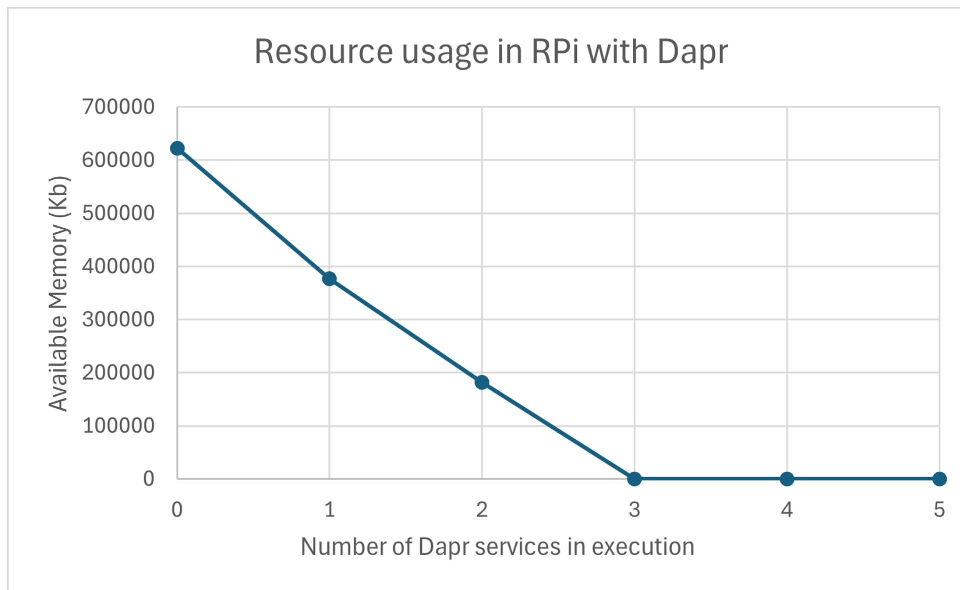

Figure 4.2: *Service deployment effects on the available (free) RAM.*

With this, we concluded that Dapr, and similar runtimes, are not a good fit for this project, as the available memory is not enough to support these. The idea of building blocks and ease of code was good however. Thus, we made the decision to find tools that facilitate the development and maintenance of the system, with a focus on tools developed for edge devices.

4.2 OpenFaaS Framework

For the second proposal, we had to consider an infrastructure that occupies less resources. Dapr failed due to a forced requirement to create a new sidecar per service instead of a single sidecar for all of the services. The current system, implemented in SpringBoot, dodges a similar issue of having one JVM per service by using a launcher services to start the rest of the services. OpenFaaS variant, faasd, on the other hand, hosts all functions in a single gateway, using containerd, saving a considerable amount of resources. Additionally, much like Dapr, faasd includes a component for metrics extraction, providing the same level of observability as Dapr.

4.2.1 Locky with faasd

To support the Locky infrastructure, and its execution in the test environment, with a serverless architecture, we chose faasd. This decision was based on analysis and comparisons performed on several serverless platforms and frameworks for edge, where OpenFaaS and faasd showed great results compared to similar frameworks [31] [46].

With faasd, as we can see in Figure 4.3, one can remotely deploy the functions to

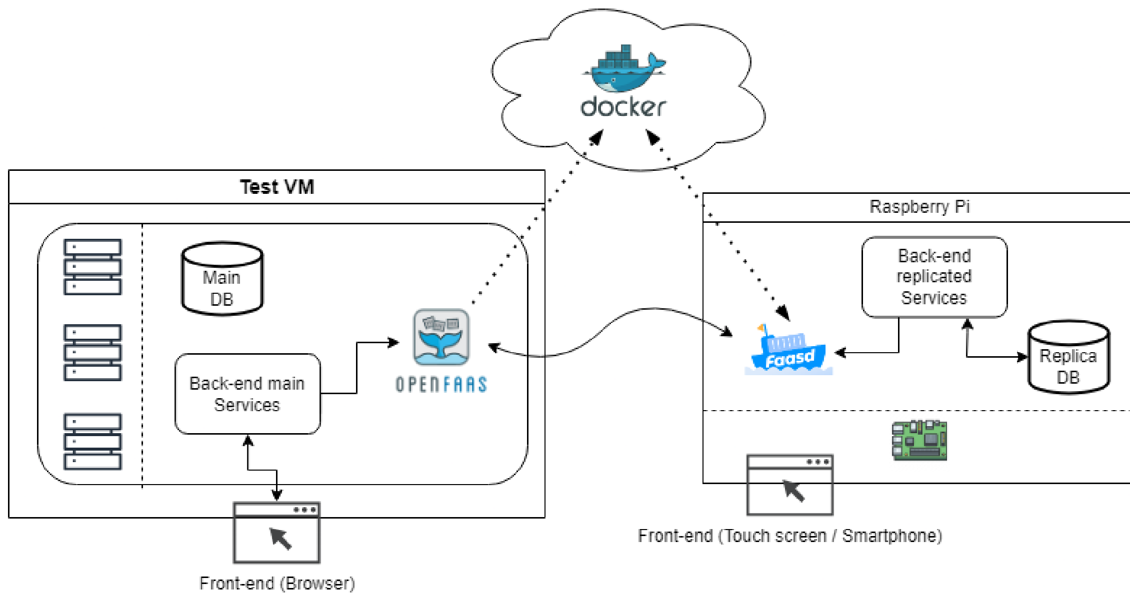

Figure 4.3: *Locker infrastructure with OpenFaaS and faasd.*

the Raspberry Pi through a gateway hosted on the device. The function image is first published in Docker Hub, and then, through a deploy function request, the remote gateway fetches the image from Docker Hub and hosts it on containerd, exposing the function endpoint on the gateway URL. We believe that this deployment strategy allows for more dynamic ways to manage and update the locky system, with a possibility to do quicker update roll-outs while maintaining system stability.

For function development, we decided to keep the original plan of replicating the back-end database services, but this time in a function format. Since all functions are hosted in the same endpoint, and listening on the same port, setting up function-to-function communication is an easy task. Besides the database functions, one or more functions will be developed for the rest of the locker workflow.

Lastly, since faasd runs directly off of containerd by creating its own CNI for networking, we are unable to use any other orchestration tool such as Docker or Podman. Thus, for a database system, we decided to choose MariaDB, as its a database system with minimal configurations required to setup as a core service, and that has been measured to occupy low amounts of system memory.

4.2.2 Testing faasd in Raspberry Pi 3

Much like in the case of Dapr, the most critical test to be made prior to the development of the complete system involves testing the effects that the deployment of services has on the available memory. In Figure 4.4, we can see the effects of faasd and the deployment of services in our 1 Gb RAM Raspberry Pi. The approach for this test was similar to Dapr: ran the command *top* in remote console from the Raspberry Pi, measured the available memory without faasd nor any functions, then started

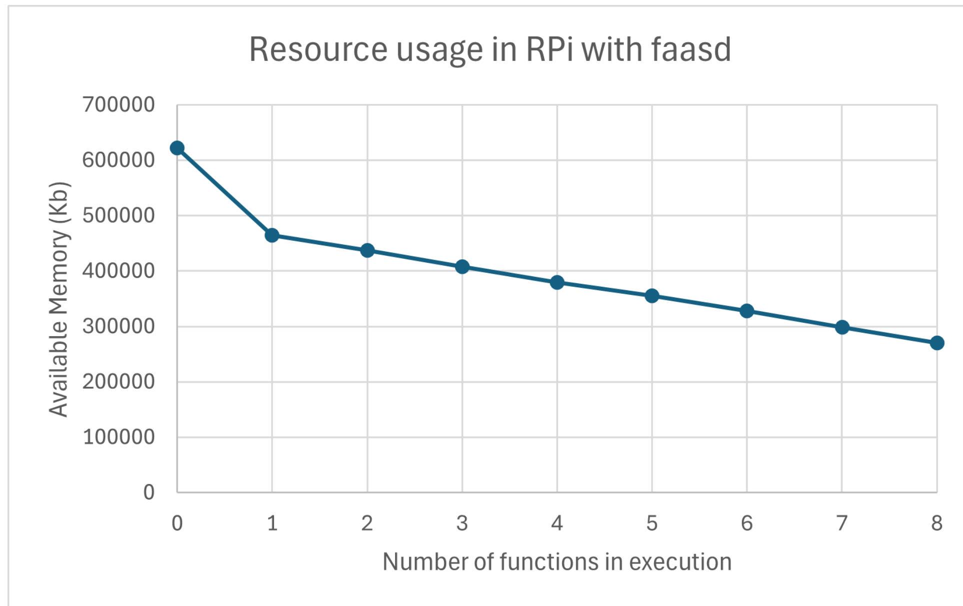

Figure 4.4: *Function deployment effects on the available (free) RAM.*

faasd and deployed the functions one by one, measuring the memory in between deployments.

Analysing the results, each function reduced the available memory by around 25 Mb of memory on average, with the faasd services reducing around 130 Mb. Overall, with 8 services deployed, the lowest measured value was available memory value was of 270 Mb. This is a very promising result, considering that we remain far from the available memory values that cause considerable system instability, which we saw in Dapr.

To conclude, this framework fits our vision of the Locky project very nicely. The remote deployment and update roll-out feature is much less restrictive than the production SpringBoot solution, and with the added bonus of having integrated metric extraction, faasd proves its worth in terms of framework features. The resources utilized by the framework also show great potential, which motivated us to develop a complete system prototype for the locker, including a web-app to track the Prometheus metrics and test the prototype.

5 Development

In this chapter, we will go over the development of the locker system, including a detailed explanation of the database schema, functions, locker workflow, and available metrics. Additionally, we will go over the challenges faced during this development. This chapter is organized as follows: In section 5.3 we present the relational model developed for this project and how it compares to the production model; In section 5.2, we go into detail on which functions will be included in our proposal, as well as their purpose in the system; In section 5.1 we explain how the functions interact, and how each system event is handled in its normal operation mode; Finally, in section 5.5 we present a React based web application that provides observability to our system, completing our smart locker management infrastructure.

5.1 Back-end Locker Interactions

In Figure 5.1 we present the workflow we designed for our prototype. There, we define the 3 main categories of functions inside the locker function environment. By color we have DB Access functions (purple), Main Controller functions (blue) and I/O functions (green), which we will go into detail later. As for the workflow events, they were split into two categories: Back-end events and Locker events. In the back-end events, the information sent by the back-end is defined as new parcel event, which is received by the database access functions, which support all CRUD operations to the local database, as mentioned previously; the update main DB event is created inside the main controller, triggered whenever a new locker interaction results in a database update, most notably when a reservation changes status. As for the locker events, the touchscreen interactions are defined as parcel request events, and which are handled by the screen controller, with the event containing the code typed by the customer; the lock open/close event is triggered by the lock controller whenever it receives a command to open, or wait for closure, of a specific door lock.

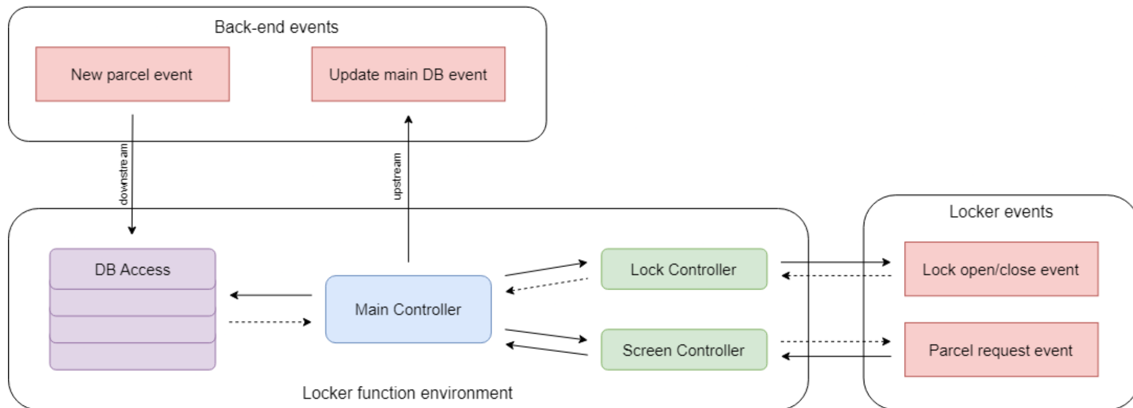

Figure 5.1: *Locker workflow events.*

There are two main entry points to this workflow, the new parcel event and the parcel request event. Because the downstream data does not need any extra processing at the edge, nor will it trigger any events by itself, the event gets redirected to the database-handling functions to store the incoming data. Meanwhile, touchscreen interactions will execute the locker business logic in the main controller. By making use of the data received from the back-end, the main controller will decide whether to open a door to the locker, or to display information on screen to convey that the key inserted is incorrect or invalid. If valid, the locker controller orders for a door to open, and is then requested to await for its closure, before the main controller disables the used code.

5.2 Functions

For function development, we first started by planning how many functions were needed for the system while keeping a service-oriented programming. To start, our plan already involved replicating the back-end database services to the locker system, and to translate this to functions, we created 1 function for each table in our database model. Next, we decided to create a function for each external component associated with the locker system, of which there are two - the touchscreen component and the door lock component. Lastly, we created one last function, which hosts the business logic related to the locker workflow, communicates updates to the back-end, and handles local errors. This brings our total to 7 functions, which can be viewed in Figure 5.2.

The database functions, named `reservation-manager`, `door-manager`, `partner-manager` and `-user-manager`, have all of the necessary CRUD operations for the locker system to receive data and maintain a working status. To communicate with the database, we make use of the `java.sql` package, with the `mariadb` database connector, to connect and make queries to the database, with the `PreparedStatement` class. All requests

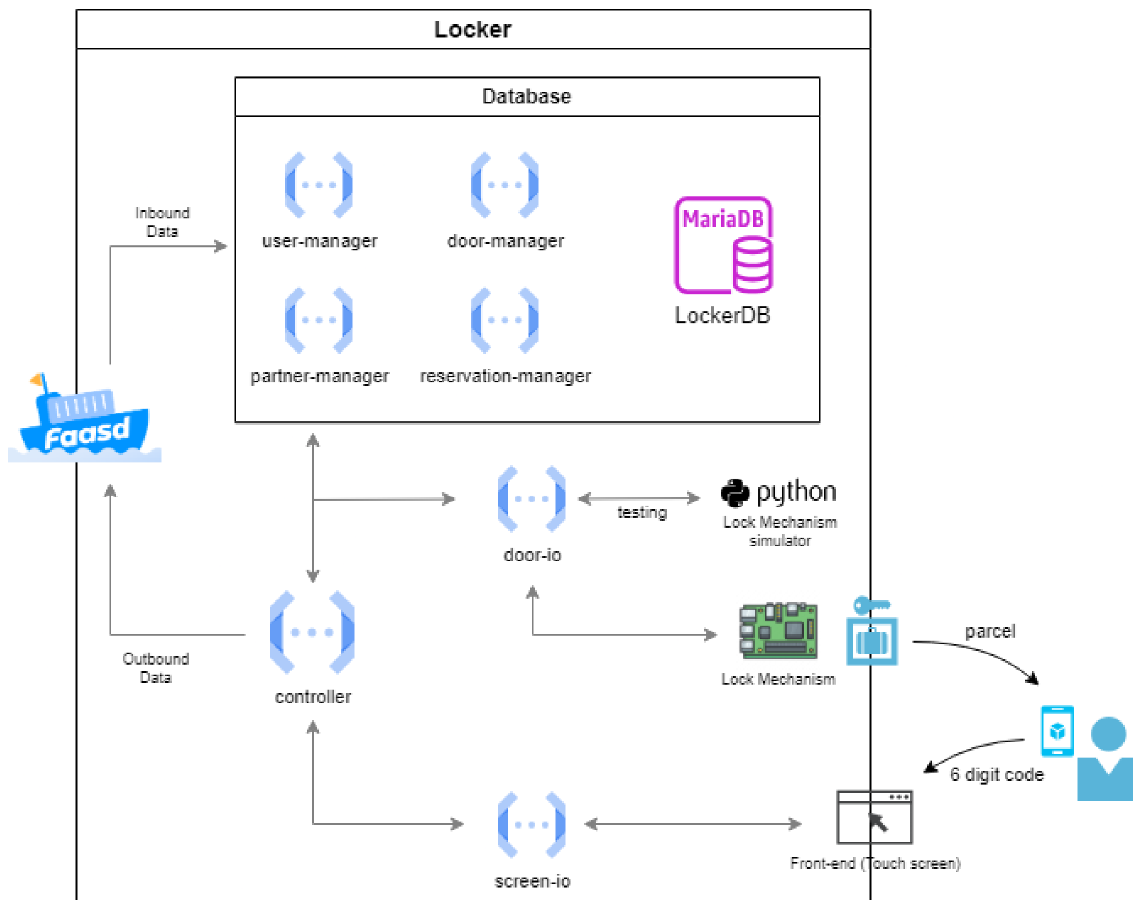

Figure 5.2: *Locker function environment and interactions.*

made to these functions must be done with HTTP POST, like all of the other functions in this system, and to specify the CRUD operation, we make use of the query string, with the format `?action=[insert/update/delete]`. Special functionalities such as validate or disable code also make use of the action query.

The touchscreen component is controller by the `screen-io` function. In our system, this function receives the code introduced in the touchscreen and send it out to the `controller` function. The code is designed so that when it receives a reply from the `controller`, after all the internal operations have concluded, the `screen-io` can update the display based on the reply information, but this functionality is not implemented, as we did not have a touchscreen to work with during the realization of this project.

The `door-io` function handles the lock mechanisms on the locker doors. It supports two possible actions, also described by the query string - `?action=[open/close]` - and receives the door identifier in the request body. For the purpose of this thesis, and testing these functionalities, we are using a legacy door simulator, written in Python, originally developed by the Locky team. This simulator also includes an interface, where the locker can be seen and interacted with. Just like the real lock system, this simulator automatically opens doors when it receives `open` requests, and waits for the

doors to be manually closed before replying to any `close` requests.

Lastly, the `controller` function is the heart of the locker system. It contains the main logic of the system, and handles most events of the system, with the exception of new back-end information, which goes directly to the database functions. This makes it so, if the workflow of the functions ever needs to be updated, only this function needs to be modified, simplifying the maintenance process.

As specified previously, all the communication handled by the functions is made with HTTP POST to specified paths in the faasd gateway. The body contents are in JSON, and only contain the minimal necessary information to perform the requested operation. We make use of the `Gson` library for JSON manipulation.

Developing functions with OpenFaaS and faasd is a simple and easy process to all the way through. With tools such as `faas-cli` and the template store, we are able to fetch project templates for all the programming languages supported by OpenFaaS, using `faas-cli new`, develop our functions, and automatically build them using `faas-cli build` or `faas-cli publish`. For this project, we made use of the default Java 11 template, showcased in Listing 5.1. The classes `AbstractHandler`, `Response`, `IRequest` and `IResponse`, are all from the `com.openfaas.model` package, which is imported by default.

Listing 5.1: OpenFaaS function template.

```java
public class Handler extends AbstractHandler {
    public IResponse Handle(IRequest req) {
        Response res = new Response();
        ...
        return res;
    }
}
```

func.java

5.3 Relational Model

The database model was a task that was tackled early on at the start of development, alongside the testing of the Dapr runtime. After an analysis of the model used in Locky's production system, and the objectives of this thesis, we arrived at the model that can be viewed in Figure 5.3. In our model, we have 4 tables - Reservations, Doors, Partners and Users:

- Reservations are the core of the locker system, representing each parcel that will, currently is, or has, interacted with this locker; a distinction that is indicated in the rsv_status column. They are associated to a door and a user.

- Doors are a digital representation of each physical door in the locker, having sequential IDs and a size type. Each door is also associated to a partner.
- Partners are, in essence, a way to allow Locky to collaborate with other online retailers. Lockers can have multiple doors allocated to certain retailers to guarantee space, and also have a custom paint for advertisement.
- Users are used internally for the purpose of testing the system - reservations made by customers are assigned a default system user when distributed on the system, while system tests are assigned a different users depending on the situation. For this project, we only use a default user.

Another important detail about the model involves the Reservations table. The 3 code fields - `rsv_dcode`, `rsv_lcode` and `rsv_ccode` - represent 3 different codes generated in the backend system. These codes are almost one time use, with a grace period of 10 minutes where it can be reused in case of a mistake, and they represent: (i) Courier deposit (dcode); (ii) Customer pick-up (lcode); (iii) Courier back-up code (ccode). The back-up code is for special occasions where the courier made a mistake during delivery. The codes are disabled by adding a special character at the start of the hash, preventing it from getting matched again.

Our model is a simplified version of the production model, removing characteristics such as door pattern designs that help with partner association, and reducing many of the tables used. With this simple model, we were able to focus on the management part of the system while also demonstrating its capabilities, and keeping almost all functionalities of the production system. Keeping the amount of tables low also allowed us to develop individual functions for each table, identical to Locky's production system and their services dedicated to each table.

5.4 Function Deployment

With *faas-cli* performing basic function configuration automatically, the process of remote function deployments becomes very simple and intuitive. In short, this process can be split into two phases: the compile phase and the deployment phase. In the compile phase, *faas-cli* compiles and publishes the functions onto Docker Hub, in order to be fetched by the remote Raspberry Pi later, performing cross-platform function compilation for the processor architecture of our choice, which in the Raspberry Pi we are using is *arm64*. This cross-compilation is achieved by using Docker buildx extension [17] and the buildkit project [16]. Then, in the deployment phase, we send a metadata file with all the functions we intend to deploy to the faasd gateway. The gateway will then fetch the functions from the Docker Hub one by one, notifying us back after each successful deployment. This interaction can be seen in Figure 5.4.

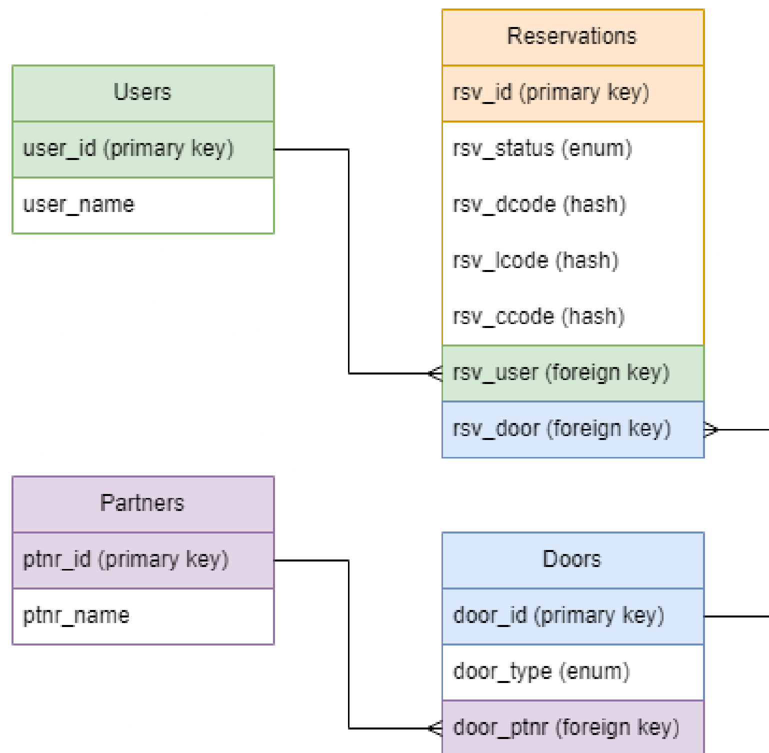

Figure 5.3: *Simplified relational model of the Locky system*

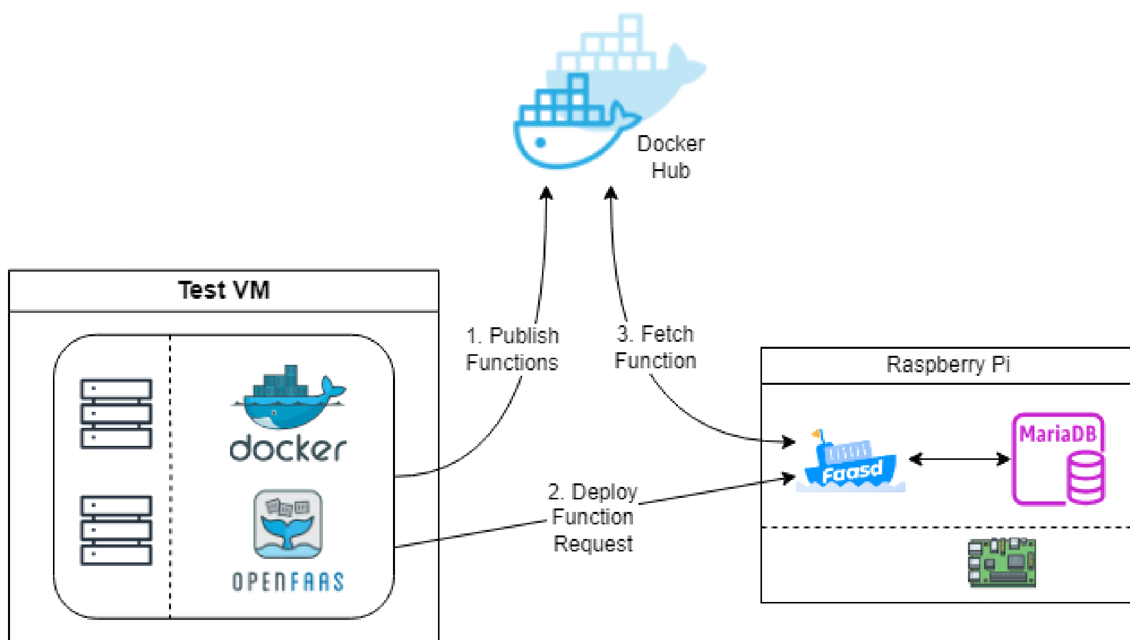

Figure 5.4: *Function deployment flow.*

Looking back at the function template we presented in Listing 5.1, when initially creating a function using *faas-cli*, a configuration file with the name of the function is also created, with the format `<function-name>.yml`, named by OpenFaaS as a "stack file"[44]. This file contains the function information, including the language, folder location and most notably, the Docker Hub image tag. The configuration file based on the function previously presented can be viewed in Listing 5.2.

Listing 5.2: OpenFaaS metadata template.

```
1 provider:
2   name: openfaas
3   gateway: <openfaas_gateway>
4
5 functions:
6   func:
7     lang: java
8     handler: ./func
9     image: func:latest
```

func.yml

Applying this to our project, when we created all the aforementioned functions with *faas-cli*, it generated the work environment seen in Figure 5.5. The automated function configuration by *faas-cli* allowed us to quickly and easily deploy each function individually for testing, so we could remain focused on code development. We also added extra configurations to include environment files for the functions that dealt with database connections and for function-to-function communication.

To simplify the deployment process further, we created an additional file called `stack.yml`, which is also recommended in the OpenFaaS documentation. In this file, we placed all of our different functions configurations into the `functions` list, which allowed us to compile and deploy all of the functions in one command each:

```
faas-cli publish -f stack.yml --platforms linux/arm64

faas-cli deploy -f stack.yml
```

A noteworthy aspect of deploying with *faas-cli* into faasd is that, even if deploying all of the functions at the same time in one file, faasd only fetches and deploys functions one at a time, assuring that each function is properly running before moving on to the next. We did not test what would happen if one of the functions in the middle of this process did not exist or simply failed to orchestrate the container, but since faasd replies with success for each function during deployment, we assume that it replies with an error code for a faulty function, and then continues on with the deployment.

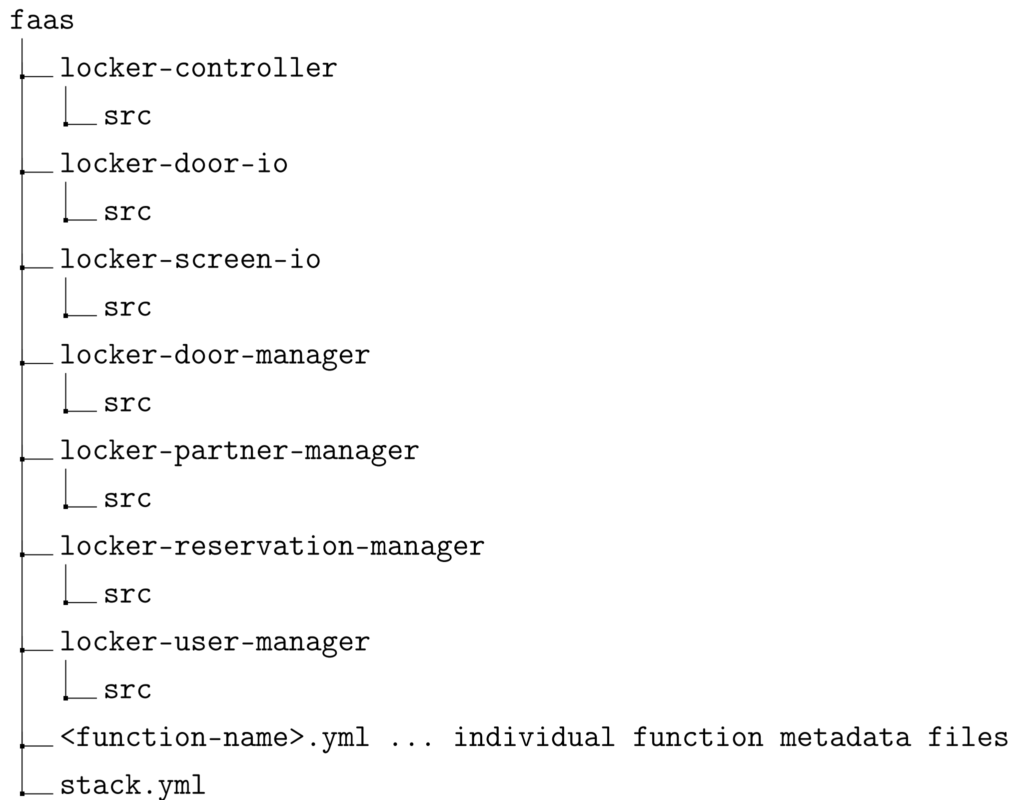

Figure 5.5: *Function environment file structure*

5.5 Metrics UI

As an effort to demonstrate faasd metrics and the observability of the system, we decided to develop a metrics user interface in the shape of web application (web-app). The objective was to develop a main page that shows the most relevant metrics of the built-in faasd Prometheus in charts, and include a series of buttons for system testing.

To complete this objective, we chose to develop a React based web-app. This decision was primarily due to React's component based programming, where we can create JavaScript functions that return HTML code, and use these functions as HTML components in the primary HTML script, simplifying the development of HTML code. Another notable aspect of React is its component interactivity. With it, we are able to dynamically update the information a component is displaying on the web page as a reaction to a user triggered event, or do timer based updates.

Before the web-app development however, we must first check how to properly fetch these metrics from the built-in Prometheus. In the OpenFaaS documentation, developers that want to create a web-app for metric observability are advised to not use nor modify the built-in Prometheus, and that this Prometheus has its query functionality

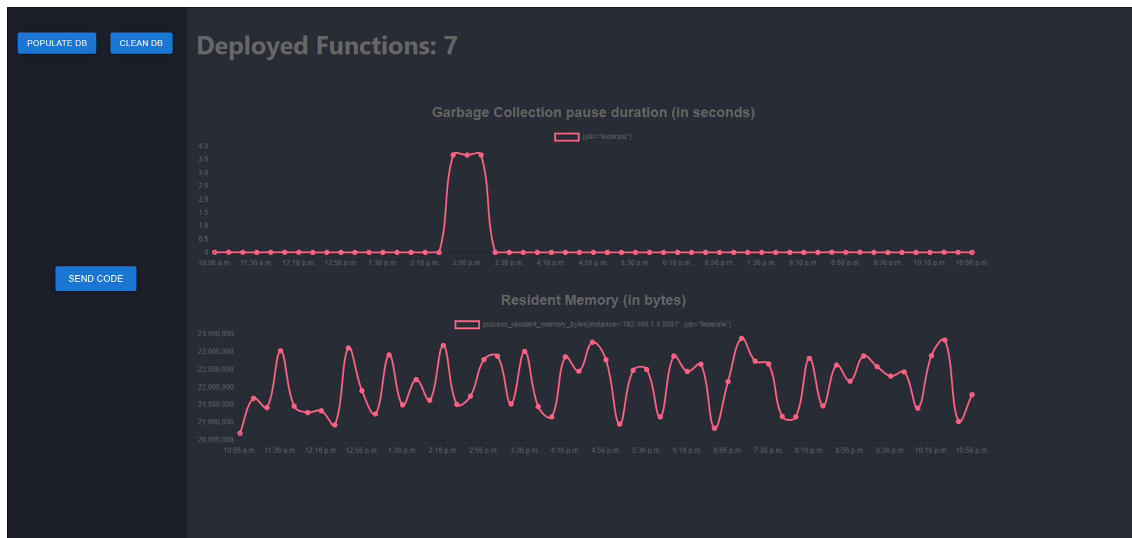

Figure 5.6: *Metrics UI*

disabled; instead developers are to create a different instance of Prometheus and configure this new instance to federate mode [53]. Federation allows a Prometheus server to scrape selected time series from other Prometheus servers, and in a distributed smart locker system, would allow us to develop a hierarchical federation model, with a back-end Prometheus instance scraping metrics from all of the different lockers and their built-in Prometheus. In the context of this web-app, this hierarchy is not as apparent, having only 1 Prometheus to scrape from.

The best way to show these metrics is by using time series graphs, as that is the inherent Prometheus data format. To create and display these graphs, seen in Figure 5.6, we made use of the `Chart.js` library, which is compatible with react and has a very complete documentation, with a Prometheus plugin support for automatic fetching and data conversion. The graphs rendered are configured to have metrics of the previous 12 hours, with displayed in intervals of 20 minutes. The graphs also automatically re-render every 5 minutes, which is the same rate of update as the Prometheus metric scrapes. To complete our web-app design, we made use of the `Material UI` (MUI) library for the locker system test buttons, which are configured to populate the locker databases with predefined tuples, test if the code opens a door in the simulator, and remove the test tuples from the database. To be able to send out these tests to the locker system, we set up a reverse proxy to redirect requests sent out to `/api/` path to the locker endpoint. Finally, the deployed functions text is pointing to a counter that is configured to fetch the amount of deployed functions from the locker system every 60 seconds, so that we can spot system failures very quickly.

Our metrics web application, alongside the developed serverless locker system, completes our proposal for a management smart locker infrastructure. We provided a set

of functions that assure the continuous operation of a locker, through a function work-flow, as well as a way to remotely maintain and update these functions. Alongside that, we developed a web application that is both able to track the stability of the locker with its available metrics and allows the developer to perform remote tests on the locker. A complete, crude list of the available metrics of the built-in Prometheus can be found in the project's GitHub [65]. However, to prove the reliability and usability of this proposal, we must first compare it to the currently in-use system and appropriately measure if trade-offs between ease-of-code and performance values exist.

6 Performance results and Project Discussions

In this chapter, we will be directly comparing our proposal to a demo system that uses the same approach as the production system in Locky, which as mentioned previously, was made on top of Spring Boot. In order for the Locky demo to be comparable to our serverless proposal, we implemented a single endpoint in each Spring service to have the same code as one of the functions. In this scenario, we will have the same amount of services in both approaches. The remainder of this chapter is organized as followed: Section 6.1 showcases the results we obtained, as well as the test environment and our methodology; In section 6.2 we discuss the results obtained and reflect on the usage of faasd as our chosen framework and Spring as the comparison target.

6.1 Results

To compare these two system infrastructures, faasd and SpringBoot, we will be following the Goal Question Metric model that was previously defined in Chapter 3, section 3.3. In short, our goal with this with this evaluation is to compare the performance of our system, with our chosen framework, against a SpringBoot alternative. To achieve this goal, we will be answering the following questions: (Q1) "What is the resource usage our system and SpringBoot?"; (Q2) "What is the performance of our system and SpringBoot?". And to accurately answer these questions, we measured the following metrics: (Q1_M1) Memory usage; (Q1_M2) CPU usage; (Q2_M1) Response Latency.

6.1.1 Test 1 - Memory Usage

Figure 6.1 shows eight deployments impact into Q1_M1 - memory usage. These results were measured after each service stabilized. Note, the first deployment, the faasd process, which hosts the function containers, occupies approximately 60000 kB less memory than a JVM + SpringBoot service. As more services are deployed, the gap between memory usage of faasd and Spring starts decreasing, with Spring

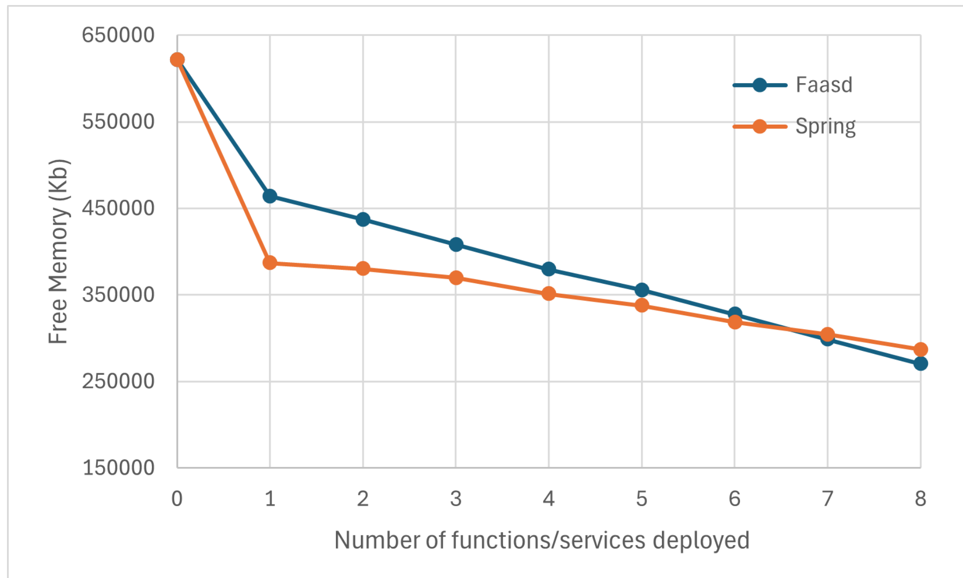

Figure 6.1: *Deployment effects on RAM (higher is better).*

eventually surpassing faasd when 7 or more services are deployed. This is due the multi-service deployment technique using only one JVM, saving a considerable amount of memory per extra service after the first one.

6.1.2 Test 2 - CPU Usage

As for Q1_M2 - CPU usage, we decided to measure how much CPU resources each infrastructure required on the most critical actions, these being services deployment (i) Before, during and after the deployment of the services; (ii) First service calls (cold-starts); (iii) Upon shutdown of the deployed services. The tool used, *HTOP*, displays the CPU usage per core, so we had access to the usage of all 4 cores of our Raspberry Pi during these actions. Thus, to display this data, we chose the average high and the average low core usages, in percent, for each infrastructure during all of the actions, as seen in Figure 6.2. During the entire deployment of services, Spring was utilizing almost all of the CPU resources, with 3 cores being at a constant 100% usage. Meanwhile with faasd, the highest core value measured was 65% during the entire function deployment process. This trend repeats itself in both the service cold-starts and shutdown process of the services, where faasd uses much less resources than Spring. Another noticeable aspect is that faasd was, at most, 50% faster than Spring during the realization of all actions, with the exception of shutdown which can be done instantly with Spring services. Overall however, faasd showed much better results than Spring, utilizing less CPU resources on all actions, with the exception of the deployment effects, after all functions were deployed, due to the *containerd* process requiring CPU resources to host the functions.

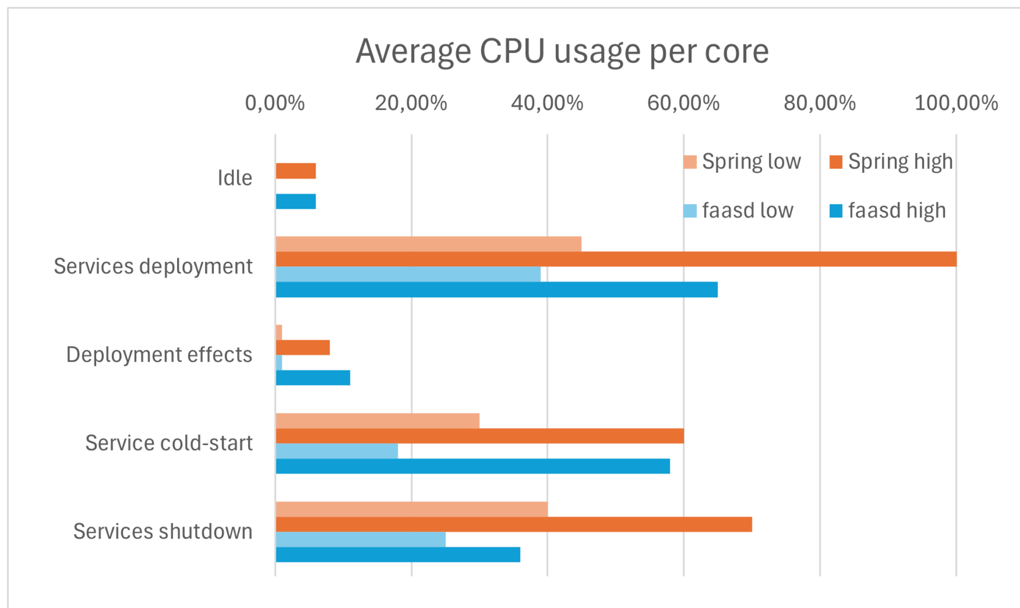

Figure 6.2: *CPU usage for each chosen event.*

6.1.3 Test 3 - Response Latency

In relation to Q2_M1 - response latency, we can observe in Table 6.1, that both architectural approaches suffer from cold starts, and the first call to either approach had a much higher response time than the following calls. In the case of the Spring service, both the get health status request and the post user request were affected by cold starts on the first call to each, whereas in faasd, only the first call to the function itself had a delayed response. This is due to the requests in faasd having to be implemented on the same endpoint, as mentioned previously. After allowing the services to stabilize, we ran a JMeter test of 100 requests to measure the average response latency. There, we measured that the Spring services replied faster to HTTP requests when compared to faasd, by about 28 ms. However, the highest value measured ended up being from the Spring services at 153 ms, while faasd takes 119 ms. A more comprehensive list of these tests can be seen in Appendix A.

Table 6.1: Response latency to HTTP Requests.

Architectural approach	First call (ms)	Cold start	Average (ms)	Highest (ms)
OpenFaaS/faasd functions	1990	Yes	79	119
Spring services	1682	Yes	51	153

6.2 Discussion

In this section, we will start by discussing the results obtained in the tests, relating them to the framework functionalities and usability. Then, we reflect on the development of our serverless prototype with faasd and the SpringBoot demo for testing, including the aspects of coding, deployment and maintenance. Finally, we touch on the concept of observability, and how each framework would adapt as part of a monitoring infrastructure.

6.2.1 Result analysis

These experimental results, show that faasd performs considerably well in edge when compared to Spring services. This can be attributed to the architectural approach of this framework, which consumes a lower amount of resources, as well as its native support of the ARM architecture. Alongside that, the containerization strategy of faasd allowed for a more optimized usage of system memory when compared to SpringBoot when using 6 or less services. This however, increased the response latency of faasd functions, with the requests having to be redirected to the function container, unlike Spring which is executed directly in a JVM. As for CPU usage, faasd requires much less CPU resources for all actions with high processing requirements when compared to Spring. This can be again partially attributed to the containerization of functions and native support for the ARM structure. SpringBoot showed a more optimized use of memory per service deployed, due to both Spring's lightweight nature and the multi-launcher optimization to only use one JVM. With this, we conclude the goal of our Goal Question Metric list, having measured and compared the most relevant performance metrics.

6.2.2 Ease of Coding, Deployment and Maintenance

In terms of software development and ease of code, both faasd and SpringBoot have their own advantages. *faasd*, through the usage of *faas-cli*, allows developers to create Java projects from function templates and focus on the function code immediately. No deployment configuration is required as it gets automatically generated alongside function creation, and it is later used by *faas-cli* to build and deploy the function. We created 7 functions, as mentioned previously, and to deploy all functions in one command we created a configuration file and copied all the generated configurations of the functions to the new file, which *faas-cli* is able to interpret and execute. Any third-party library we want to use needs to be included in the *gradle.build* file of the function. As for SpringBoot, it not only allows applications to only require minimal Spring configuration, but also makes the integration of third-party libraries a seamless process. In the scope of this project, we created a SpringBoot demo based on the

Spring multi-launcher technique and did not need to create any configuration files for our solution to work just as faasd. Since the database access code needed for this project was not too complex, we chose not use Spring Data JPA for this development, but it is another tool for Spring to simplify code development.

As for deployment, the function deployment model is much more effective than the Spring multi-launcher technique. Using faasd, we can deploy the functions we want at any point in time through *faas-cli*, using well documented commands. However, with Spring, since it uses a launcher service to start all other Spring services inside the same JVM, it lacks flexibility in deployment, which may cause issues during testing, such as only wanting to test a few of the services and being unable to. Similarly, to work around not having an autoscaling feature, with faasd one can deploy multiple similar functions and set up a reverse proxy for load balancing, which requires to be manually setup on the Spring multi-service launcher. During the preparation and execution of the tests above, we also noticed that faasd is considerably faster at service deployment than Spring, even if *faas-cli* deploys a function and waits for that function to be operational before moving on to the next.

For the maintenance aspect of the infrastructures, faasd once again displays its flexibility by allowing for update roll-outs on the deployed functions, allowing for minimal service downtime during updates and maintenance. As for Spring, again due to the multi-service launcher approach, to update one service, we are required to take down all the running services, deploy the update, and restart all of the services through the launcher. This sequence of actions required by Spring multi-launcher places more load on the CPU overall compared to faasd. This load disparity, over a long period of time, may lead to more battery consumption, but further testing would be required.

6.2.3 Observability of both frameworks

Last but not least, a very important topic for this project involves the observability and metric extraction of both frameworks, in order to properly design a central monitoring interface that manages all of the remotely deployed locker services.

Starting with faasd, as we have already introduced in chapter 5 section 5.5, faasd provides a built-in Prometheus for development of web applications. This Prometheus however, is not modifiable, and does not allow us to perform queries on the available data. So, in order to display the data we desire, we need to configure a new Prometheus instance, and configure it to federate the data from our faasd Prometheus. This solution works relatively well, with very flexible configurations that allow us to determine the interval between updates and which metric group we want to fetch from. Although not very explored in this project, Federating metrics allows the development of an hierarchical observation system, where a central back-end Prometheus instance

scrapes metrics from all the deployed locker systems, allowing us to build a more complex web application for monitoring distributed lockers.

As for SpringBoot, it includes a number of additional features to help us monitor and manage the locker services, including auditing, health checks, and metrics gathering, that can be automatically applied to our services. We can manage and monitor our services by using HTTP endpoints or with JMX. Although included by default, these features are enabled through a use of a SpringBoot Actuator, which is added as a dependency to the project. From there, we can configure Observation components in our services which gives us access to logs, metrics, tracing and auditing.

As the SpringBoot solution was only built for the purpose of comparing performance, the observability components were not developed as part of this project, but we must consider them as a point of comparison between our frameworks in the infrastructure. Overall the SpringBoot observability solution provides more flexibility in choosing which metrics we want to gather, as well as additional features such as observing logs and performing audits, which need to be included as components in each service. As for faasd, we can plan and build a hierarchical monitoring structure seamlessly through the use of Prometheus with no need for extra code, but we can only use the metrics available by default. In the case of observability, there is no better framework to chose from. Each have their own advantages and disadvantages, so it is up to the developer to decide which is best.

7 Conclusions and Future Work

To conclude this work, we will highlight the main achievements that our work accomplished and their importance. Then, we discuss the main difficulties encountered during the development of this work, and their impact. Finally, we discuss the points of improvement of our prototype and future work.

7.1 Achievements

Our literature review into distributed architectures and edge computing helped create a solid base for the development of this work. Most noticeably, the overview on the Serverless Edge Computing Paradigm clearly defined the problems and benefits of serverless in edge and IoT, which was of great help during the development of the serverless prototype. This overview also helped consolidate the main difficulties of service development in IoT devices, which helped accelerate the process of choosing the most appropriate framework to support our project.

The OpenFaaS framework proved to be a perfect fit for the context of this project. The OpenFaaS variant, faasd, was built for the purpose of serverless development in IoT and since it included all of the features of OpenFaaS, we were able to use automatic configuration and template generation to simplify the development process and remain focused on the development of the prototype.

Thus, a functional prototype was made, which serves as a proof-of-concept and baseline for further development. As mentioned previously, some simplifications were made, namely in terms of the database model, which also simplified the code developed, and making our prototype be far from its potential. With more complexity and physical locker testing, we believe that our prototype can become a reliable smart locker system.

Finally, the Goal Question Metric model developed in Chapter 3 helped us create a basis for testing and comparison, with well-defined goals, questions to answer and

metrics to extract from both faasd, our prototype, and the SpringBoot demo. With this, we were able to obtain very positive results for serverless development in edge devices, and thus further merit to extend investigations into the field of serverless edge.

All of the code developed during this work is publicly available on the GitHub [65].

7.2 Main Difficulties

At several points during this work, there were complications that hindered the development process and/or affected the prototype:

- **Initial Confusion** - There was some confusion at the start of development about the scope of this work. The initial idea for this work was the development of a monitoring infrastructure involving both locker services and a cloud back-end support. This idea was later changed to only the development of the locker services, which made us return back to the planning process, slowing down development.
- **Lack of Component Integration** - faasd does not include any default interfaces to simplify code development. Thus, for example, for database integration, so we had to manually write the code database access code, as well as injecting the database connector into the function packages.
- **Slow Metric Gathering** - As mentioned previously, the built-in, un-modifiable, Prometheus instance included in faasd, does not keep any past metric snapshots. Thus, if we want a days worth of metric snapshots from the built-in Prometheus, we need to have our own Prometheus instance scraping the current metrics and storing them as individual snapshots. Another aspect related to Prometheus is that the built-in Prometheus always displays its metrics with real-time values, meaning that we could be scraping and saving metric snapshots in intervals smaller than 20 minutes. We ended up not exploring faster scraping with our Prometheus instance due to the slow metric gathering.
- **Grafana Cloud Issues** - Although initially considered as an option for our monitoring web-application, we were unable to use Grafana Cloud to fetch the metrics from the built-in Prometheus. This was due to Grafana Cloud only accepting Prometheus endpoints with SSL/TLS certificates to perform HTTPS requests. As such, we decided to develop our own web application, using a React, and integrate additional functionalities, i.e. locker testing, in the monitoring interface. Unfortunately, since our focus was heavily on the prototype development for the locker services, our web application was not as complex as it could be and did not reach its full potential as a monitoring interface.

7.3 Future Work

In the following list we include some points for future work in both the development of this prototype, as well as future investigations:

- **Increased Complexity** - As mentioned previously, our prototype provides a baseline for development with a simple functionality. However, with increased complexity, our prototype can evolve into a ready-for-production infrastructure, without losing any of its current capacities.
- **More Testing** - Although our prototype showed positive results when compared to SpringBoot, more testing should be performed for a more complete evaluation, such as battery life and long-term deployments.
- **Serverless Edge Paradigm** - The field of serverless edge is still very recent, and the path of progress in investigation is still very vague, along with the best usability for serverless in IoT devices. We believe that our prototype helped in this regard by proving to be a great supporting framework, as shown by the comparative results, in a smart locker infrastructure built with function workflows, and hope that we see further investigations on the usability of serverless in edge systems.

Bibliography

[1] V. Anand, D. Garg, A. Kaufmann, and J. Mace. “Blueprint: A Toolchain for Highly-Reconfigurable Microservices”. In: (2023) (cit. on p. 17).

[2] *Apache JMETER*. https://jmeter.apache.org/. Accessed: Jul 2024 (cit. on p. 25).

[3] *Apache Openwhisk*. https://openwhisk.apache.org. Accessed: June 2024 (cit. on p. 11).

[4] M. S. Aslanpour, A. N. Toosi, C. Cicconetti, B. Javadi, P. Sbarski, D. Taibi, M. Assuncao, S. S. Gill, R. Gaire, and S. Dustdar. “Serverless edge computing: vision and challenges”. In: *Proceedings of the 2021 Australasian computer science week multiconference*. 2021, pp. 1–10 (cit. on p. 11).

[5] *AWS Greengrass*. https://aws.amazon.com/greengrass/. Accessed: June 2024 (cit. on p. 12).

[6] *AWS Lambda*. https://aws.amazon.com/lambda/. Accessed: June 2024 (cit. on p. 12).

[7] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, et al. “Serverless computing: Current trends and open problems”. In: *Research advances in cloud computing* (2017), pp. 1–20 (cit. on p. 10).

[8] L. Baresi and D. F. Mendonça. “Towards a serverless platform for edge computing”. In: *2019 IEEE International Conference on Fog Computing (ICFC)*. IEEE. 2019, pp. 1–10 (cit. on pp. 2, 10).

[9] M. Barletta, M. Cinque, L. De Simone, and R. Della Corte. “Introducing k4. 0s: a model for mixed-criticality container orchestration in industry 4.0”. In: *2022 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/-CyberSciTech)*. IEEE. 2022, pp. 1–6 (cit. on p. 17).

[10] A. Burns and R. Davis. “Mixed criticality systems-a review”. In: *Department of Computer Science, University of York, Tech. Rep* (2013), pp. 1–69 (cit. on p. 17).

[11] B. Butzin, F. Golatowski, and D. Timmermann. "Microservices approach for the internet of things". In: *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE. 2016, pp. 1–6 (cit. on p. 2).

[12] V. R. B. G. Caldiera and H. D. Rombach. "The goal question metric approach". In: *Encyclopedia of software engineering* (1994), pp. 528–532 (cit. on p. 24).

[13] B. Charron-Bost, F. Pedone, and A. Schiper. *Replication*. Springer, 2010 (cit. on p. 21).

[14] *Convention-over-Configuration*. https://en.wikipedia.org/wiki/Convention_over_configuration. Accessed: Sep 2024 (cit. on p. 14).

[15] A. Davis, J. Parikh, and W. E. Weihl. "Edgecomputing: extending enterprise applications to the edge of the internet". In: *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*. 2004, pp. 180–187 (cit. on p. 9).

[16] *Docker BuildKit*. https://docs.docker.com/build/buildkit/. Accessed: Sep 2024 (cit. on p. 37).

[17] *Docker Buildx Extension*. https://docs.docker.com/reference/cli/docker/buildx/. Accessed: Sep 2024 (cit. on p. 37).

[18] J. Edwards, A. McKinnon, T. Cherrett, F. McLeod, and L. Song. "The impact of failed home deliveries on carbon emissions: Are collection/delivery points environmentally-friendly alternatives". In: *14th Annual Logistics Research Network Conference*. 2009, p. M117 (cit. on p. 1).

[19] T. Erl. *The Annotated SOA Manifesto*. https://patterns.arcitura.com/wp-content/uploads/2019/03/Annotated_SOA_Manifesto.pdf. Accessed: 2024/01/23 (cit. on p. 6).

[20] T. Erl. *Service-oriented architecture: concepts, technology, and design*. Pearson Education India, 1900 (cit. on p. 12).

[21] *faasd - A lightweight and portable faas engine*. https://github.com/openfaas/faasd. Accessed: Sep 2024 (cit. on p. 14).

[22] *Garbage Collection Target Percentage reference (SetGCPercent))*. https://pkg.go.dev/runtime/debug. Accessed: Sep 2024 (cit. on p. 15).

[23] R. Gatev. *Introducing Distributed Application Runtime (Dapr)*. Springer, 2021 (cit. on pp. 12, 13).

[24] *Grafana*. https://grafana.com/. Accessed: Sep 2024 (cit. on p. 16).

[25] *Graphite*. https://graphiteapp.org/. Accessed: Sep 2024 (cit. on p. 15).

[26] *HashiCorp Consul*. https://www.consul.io/. Accessed: May 2024 (cit. on p. 28).

[27] J. Highsmith. *Adaptive software development: a collaborative approach to managing complex systems*. Addison-Wesley, 2013 (cit. on p. 6).

[28] M. D. Hossain, T. Sultana, S. Akhter, M. I. Hossain, N. T. Thu, L. N. Huynh, G.-W. Lee, and E.-N. Huh. "The role of microservice approach in edge computing: Opportunities, challenges, and research directions". In: *ICT Express* (2023) (cit. on pp. 2, 10).

[29] *htop - an interactive process viewer*. https://htop.dev/. Accessed: Sep 2024 (cit. on p. 25).

[30] V. Issarny, G. Bouloukakis, N. Georgantas, and B. Billet. "Revisiting service-oriented architecture for the IoT: a middleware perspective". In: *Service-Oriented Computing: 14th International Conference, ICSOC 2016, Banff, AB, Canada, October 10-13, 2016, Proceedings 14*. Springer. 2016, pp. 3–17 (cit. on pp. 2, 10).

[31] H. Javed, A. N. Toosi, and M. S. Aslanpour. "Serverless platforms on the edge: a performance analysis". In: *New Frontiers in Cloud Computing and Internet of Things*. Springer, 2022, pp. 165–184 (cit. on p. 29).

[32] P. G. Keen. *Shaping the future: Business design through information technology*. Harvard Business School Press, 1991 (cit. on p. 5).

[33] *Knative*. https://knative.dev/docs/. Accessed: Jul 2024 (cit. on p. 13).

[34] H. Koziolek and N. Eskandani. "Lightweight Kubernetes Distributions: A Performance Comparison of MicroK8s, k3s, k0s, and Microshift". In: *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering*. 2023, pp. 17–29 (cit. on p. 16).

[35] D. Krafzig, K. Banke, and D. Slama. *Enterprise SOA: service-oriented architecture best practices*. Prentice Hall Professional, 2005 (cit. on p. 6).

[36] M. Kumar. "Serverless architectures review, future trend and the solutions to open problems". In: *American Journal of Software Engineering* 6.1 (2019), pp. 1–10 (cit. on p. 8).

[37] *Locky - Quem Somos*. https://locky.pt/home/quem-somos/. Accessed: May 2024 (cit. on p. 1).

[38] M. Loukides and S. Swoyer. *Microservices Adoption in 2020*. https://www.oreilly.com/radar/microservices-adoption-in-2020/. Accessed: 2024/02/10 (cit. on p. 7).

[39] B. Malekzadeh. “Event-Driven Architecture and SOA in collaboration-A study of how Event-Driven Architecture (EDA) interacts and functions within Service-Oriented Architecture (SOA)”. MA thesis. 2010 (cit. on p. 7).

[40] R. Mangiaracina, A. Perego, A. Seghezzi, and A. Tumino. “Innovative solutions to increase last-mile delivery efficiency in B2C e-commerce: a literature review”. In: *International Journal of Physical Distribution & Logistics Management* 49.9 (2019), pp. 901–920 (cit. on p. 1).

[41] T. Muhammad, M. T. Munir, M. Z. Munir, and M. W. Zafar. “Elevating Business Operations: The Transformative Power of Cloud Computing”. In: *International Journal of Computer Science and Technology* 2.1 (2018), pp. 1–21 (cit. on p. 5).

[42] S. Niedermaier, F. Koetter, A. Freymann, and S. Wagner. “On observability and monitoring of distributed systems–an industry interview study”. In: *Service-Oriented Computing: 17th International Conference, ICSOC 2019, Toulouse, France, October 28–31, 2019, Proceedings 17*. Springer. 2019, pp. 36–52 (cit. on p. 15).

[43] *Openfaas*. https://www.openfaas.com/. Accessed: July 2024 (cit. on p. 14).

[44] *Openfaas YAML*. https://docs.openfaas.com/reference/yaml/. Accessed: Aug 2024 (cit. on p. 39).

[45] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi. “Cloud container technologies: a state-of-the-art review”. In: *IEEE Transactions on Cloud Computing* 7.3 (2017), pp. 677–692 (cit. on p. 8).

[46] A. Palade, A. Kazmi, and S. Clarke. “An evaluation of open source serverless computing frameworks support at the edge”. In: *2019 IEEE World Congress on Services (SERVICES)*. Vol. 2642. IEEE. 2019, pp. 206–211 (cit. on p. 29).

[48] P. Persson and O. Angelsmark. “Calvin–merging cloud and iot”. In: *Procedia Computer Science* 52 (2015), pp. 210–217 (cit. on p. 18).

[49] P. Persson and O. Angelsmark. “Kappa: serverless iot deployment”. In: *Proceedings of the 2nd International Workshop on Serverless Computing*. 2017, pp. 16–21 (cit. on p. 18).

[50] R. Picoreti, A. P. do Carmo, F. M. de Queiroz, A. S. Garcia, R. F. Vassallo, and D. Simeonidou. “Multilevel observability in cloud orchestration”. In: *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/-DataCom/CyberSciTech)*. IEEE. 2018, pp. 776–784 (cit. on p. 15).

[51] J. Pilawa, L. Witell, A. Valtakoski, and P. Kristensson. “Service innovativeness in retailing: Increasing the relative attractiveness during the COVID-19 pandemic”. In: *Journal of Retailing and Consumer Services* 67 (2022), p. 102962 (cit. on p. 1).

[52] *Prometheus*. https://prometheus.io/. Accessed: Aug 2024 (cit. on p. 15).

[53] *Prometheus Federate Documentation*. https://prometheus.io/docs/prometheus/latest/federation/. Accessed: Aug 2024 (cit. on p. 41).

[54] P. Raith, T. Rausch, A. Furutanpey, and S. Dustdar. “faas-sim: A trace-driven simulation framework for serverless edge computing platforms”. In: *Software: Practice and Experience* 53.12 (2023), pp. 2327–2361 (cit. on p. 18).

[55] R. A. P. Rajan. “Serverless architecture-a revolution in cloud computing”. In: *2018 Tenth International Conference on Advanced Computing (ICoAC)*. IEEE. 2018, pp. 88–93 (cit. on p. 8).

[56] *Raspberry Pi Documentation*. https://www.raspberrypi.com/documentation/computers/raspberry-pi.html. Accessed: Sep 2024 (cit. on p. 22).

[57] M. Richards. *Microservices vs. service-oriented architecture*. O’Reilly Media Sebastopol, 2015 (cit. on p. 6).

[58] A. Rotem-Gal-Oz. *SOA patterns*. Simon and Schuster, 2012 (cit. on p. 6).

[59] P. Sharma, A. Ali-Eldin, and P. Shenoy. “Resource deflation: A new approach for transient resource reclamation”. In: *Proceedings of the Fourteenth EuroSys Conference 2019*. 2019, pp. 1–17 (cit. on p. 18).

[60] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. “Edge computing: Vision and challenges”. In: *IEEE internet of things journal* 3.5 (2016), pp. 637–646 (cit. on p. 9).

[61] *SpringBoot Project*. https://spring.io/projects/spring-boot. Accessed: Sep 2024 (cit. on p. 14).

[62] A. A. Stavdas. “NETWORKED INTELLIGENCE A Wider Fusion of Technologies That Spurs the Fourth Industrial Revolution—Part I: Foundations”. In: 2022 (cit. on p. 17).

[63] H. Suryotrisongko and P. K. F. Ananto. “The potential of microservice architecture for internet of things (iot) in smart city, a literature review”. In: *Jurnal Ilmiah Kursor* 9.1 (2017) (cit. on p. 10).

[64] L. Szász, C. Bálint, O. Csíki, B. Z. Nagy, B.-G. Racz, D. Csala, and L. C. Harris. “The impact of COVID-19 on the evolution of online retail: The pandemic as a window of opportunity”. In: *Journal of Retailing and Consumer Services* 69 (2022), p. 103089 (cit. on p. 1).

[65] *TFM Locker Services Github*. https://github.com/MSeleiro/tfm-locker-services. Accessed: Sep 2024 (cit. on pp. 42, 50, 59).

[66] B. Wang, A. Ali-Eldin, and P. Shenoy. “Lass: Running latency sensitive serverless computations at the edge”. In: *Proceedings of the 30th international symposium on high-performance parallel and distributed computing*. 2021, pp. 239–251 (cit. on p. 18).

[67] P. Weill, M. Subramani, and M. Broadbent. “IT infrastructure for strategic agility”. In: *Available at SSRN 317307* (2002) (cit. on p. 5).

[68] Y. Zhang and J.-I. Chen. “Constructing scalable Internet of Things services based on their event-driven models”. In: *Concurrency and Computation: Practice and Experience* 27.17 (2015), pp. 4819–4851 (cit. on pp. 2, 10).

A JMeter Latency test output

The following lists contain the Timestamp, Bytes sent, and request Latency (in milliseconds) of the 100 requests performed during Test 3 to measure the response latency of our prototype and the SpringBoot demo. The complete and crude JMeter logs can be found in this project GitHub [65].

A.1 faasd results

Timestamp	Bytes	Latency (ms)	Timestamp	Bytes	Latency (ms)
1719668093502	273	119	1719668094994	273	82
1719668093656	273	87	1719668095077	273	82
1719668093733	273	97	1719668095150	273	84
1719668093801	273	74	1719668095224	273	81
1719668093865	273	68	1719668095295	273	95
1719668093924	273	70	1719668095381	273	78
1719668093994	273	86	1719668095449	273	93
1719668094060	273	72	1719668095533	273	85
1719668094122	273	82	1719668095618	273	73
1719668094195	273	75	1719668095691	273	61
1719668094260	273	87	1719668095753	273	68
1719668094318	273	71	1719668095821	273	69
1719668094379	273	86	1719668095890	273	83
1719668094435	273	70	1719668095953	273	79
1719668094496	273	74	1719668096013	273	92
1719668094561	273	73	1719668096075	273	78
1719668094624	273	74	1719668096133	273	66
1719668094689	273	90	1719668096200	273	63
1719668094770	273	83	1719668096263	273	73
1719668094843	273	79	1719668096337	273	81
1719668094913	273	91	1719668096773	273	83

Timestamp	Bytes	Latency (ms)	Timestamp	Bytes	Latency (ms)
1719668096836	273	62	1719668098475	273	89
1719668096899	273	89	1719668098534	273	73
1719668096958	273	75	1719668098597	273	89
1719668097023	273	71	1719668098676	273	74
1719668097084	273	72	1719668098740	273	105
1719668097147	273	72	1719668098836	273	78
1719668097209	273	70	1719668098904	273	71
1719668097270	273	99	1719668098965	273	75
1719668097329	273	72	1719668099030	273	80
1719668097391	273	70	1719668099100	273	74
1719668097452	273	72	1719668099155	273	98
1719668097514	273	75	1719668099213	273	79
1719668097580	273	73	1719668099283	273	84
1719668097644	273	75	1719668099357	273	84
1719668097710	273	80	1719668099431	273	74
1719668097780	273	71	1719668099495	273	73
1719668097842	273	71	1719668099558	273	80
1719668097903	273	71	1719668099628	273	75
1719668097964	273	74	1719668099693	273	74
1719668098028	273	75	1719668099758	273	99
1719668098093	273	78	1719668099847	273	78
1719668098161	273	79	1719668099916	273	81
1719668098230	273	70	1719668099987	273	93
1719668098291	273	73	1719668100050	273	73
1719668098354	273	97	1719668100113	273	70
1719668098411	273	73	1719668100172	273	82

A.2 SpringBoot results

Timestamp	Bytes	Latency (ms)	Timestamp	Bytes	Latency (ms)
1719668205228	269	153	1719668205964	269	57
1719668205385	269	44	1719668206001	269	76
1719668205429	269	52	1719668206058	269	60
1719668205471	269	64	1719668206098	269	61
1719668205516	269	55	1719668206140	269	67
1719668205552	269	58	1719668206187	269	59
1719668205591	269	45	1719668206227	269	60
1719668205627	269	49	1719668206268	269	54
1719668205667	269	51	1719668206312	269	45
1719668205739	269	62	1719668206358	269	46
1719668205782	269	56	1719668206395	269	48
1719668205818	269	48	1719668206433	269	51
1719668205846	269	53	1719668206474	269	42
1719668205880	269	65	1719668206517	269	58
1719668205926	269	58	1719668206555	269	57

Timestamp	Bytes	Latency (ms)	Timestamp	Bytes	Latency (ms)
1719668206602	269	59	1719668207847	269	45
1719668206642	269	65	1719668207872	269	47
1719668206703	269	60	1719668207899	269	47
1719668206743	269	50	1719668207926	269	35
1719668206784	269	57	1719668207951	269	36
1719668206831	269	53	1719668207977	269	44
1719668206875	269	56	1719668208001	269	53
1719668206911	269	51	1719668208034	269	54
1719668206943	269	48	1719668208068	269	46
1719668206971	269	50	1719668208095	269	37
1719668207001	269	51	1719668208132	269	45
1719668207033	269	55	1719668208167	269	44
1719668207068	269	57	1719668208191	269	44
1719668207105	269	52	1719668208215	269	44
1719668207137	269	55	1719668208239	269	48
1719668207172	269	63	1719668208267	269	49
1719668207216	269	55	1719668208296	269	46
1719668207261	269	48	1719668208323	269	46
1719668207289	269	48	1719668208350	269	56
1719668207318	269	52	1719668208387	269	55
1719668207361	269	48	1719668208422	269	45
1719668207389	269	35	1719668208448	269	45
1719668207424	269	38	1719668208473	269	46
1719668207462	269	49	1719668208499	269	44
1719668207491	269	50	1719668208523	269	34
1719668207521	269	48	1719668208547	269	43
1719668207550	269	49	1719668208570	269	48
1719668207580	269	52	1719668208599	269	40
1719668207612	269	53	1719668208629	269	47
1719668207645	269	53	1719668208656	269	31
1719668207678	269	58	1719668208687	269	49
1719668207716	269	46	1719668208716	269	42
1719668207752	269	36	1719668208739	269	44
1719668207788	269	46	1719668208763	269	43
1719668207815	269	52	1719668208786	269	39