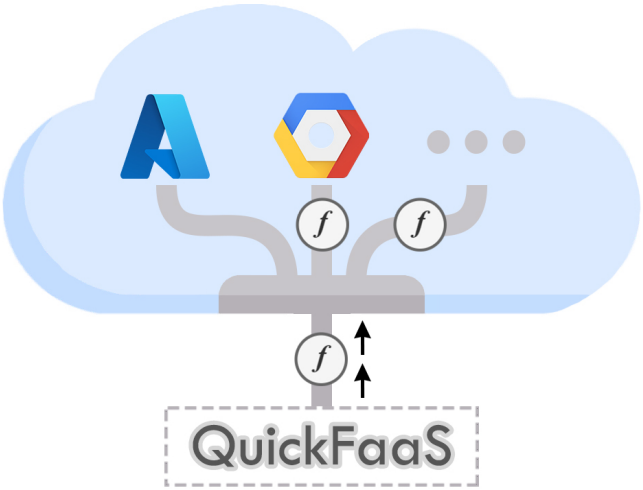
# INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

**Departamento de Engenharia de Electrónica e Telecomunicações e de Computadores**


# Characterizing and Providing Interoperability to Function as a Service Platforms


**Pedro Miguel Fialho Rodrigues**


Bachelor's degree

Dissertação para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores


Orientadores : Prof. Doutor Filipe Freitas
Prof. Doutor José Simão


Júri:

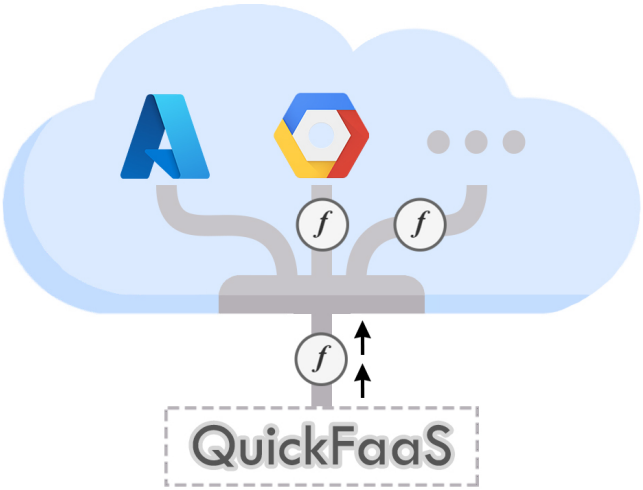
Presidente: Prof. Doutor Carlos Jorge de Sousa Gonçalves

Vogais: Prof. Doutor Manuel Martins Barata
Prof. Doutor Filipe Bastos de Freitas

**December, 2022**

# INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

**Departamento de Engenharia de Electrónica e Telecomunicações e de Computadores**


# Characterizing and Providing Interoperability to Function as a Service Platforms


**Pedro Miguel Fialho Rodrigues**


Bachelor's degree

Dissertação para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores


Orientadores : Prof. Doutor Filipe Freitas
Prof. Doutor José Simão


Júri:

Presidente: Prof. Doutor Carlos Jorge de Sousa Gonçalves

Vogais: Prof. Doutor Manuel Martins Barata
Prof. Doutor Filipe Bastos de Freitas

**December, 2022**

*Aos meus pais por toda a educação que me deram, ao meu irmão por ser a referência que é, e aos meus amigos pelos momentos inesquecíveis que passámos juntos.*

# Agradecimentos

As competências demonstradas num trabalho final de mestrado são o culminar de um percurso académico com vários anos de aprendizagem, esforço e dedicação. Contudo, tal não seria possível sem a ajuda e o apoio incondicional de um conjunto de pessoas.

Nesse sentido, gostaria de deixar, em primeiro lugar, um agradecimento especial aos meus orientadores, Exmo. Professor Doutor Filipe Freitas e Exmo. Professor Doutor José Simão, pela confiança que depositaram em mim desde o começo do projeto, e pela boa cooperação que mantivemos no decurso do mesmo. A sua entrega e disponibilidade diária deram-me as forças necessárias para me manter motivado e com uma ambição de autossuperação. Graças a eles, pude experienciar novos desafios na área da investigação e atingir metas que sem a sua ajuda dificilmente seriam alcançáveis.

Ao Instituto Superior de Engenharia de Lisboa, por ter sido a minha segunda casa nos últimos cinco anos. Irei para sempre recordar excelentes memórias da família iseliana, da qual me orgulho em fazer parte. Ao corpo docente da LEIC e do MEIC, não só pela transmissão de conhecimentos teórico-práticos, mas também por me terem ajudado a desenvolver um pensamento crítico direcionado para a resolução de problemas que transcende a área da engenharia informática. Aos não docentes, por cuidarem de nós, alunos, da instituição, e por serem, acima de tudo, ótimos profissionais. Deixo também uma palavra de agradecimento ao Instituto Politécnico de Lisboa, pela atribuição da bolsa de iniciação à investigação, com referência *IPL/2021/FaaS-IntOp_ISEL*, e por continuarem a apostar neste tipo de iniciativas.

Aos meus amigos, que nas etapas de maior ansiedade me permitiram descontrair e abstrair-me temporariamente do mundo académico.

Por último, mas não menos importante, agradecer aos meus pais, irmão e restante família, por nunca me terem falhado com nada ou colocado barreiras em quaisquer que fossem as minhas decisões académicas e profissionais.

# Abstract

Serverless computing hides infrastructure management from developers and runs code on-demand automatically scaled and billed during code's execution time. One of the most popular serverless backend services is called Function-as-a-Service (FaaS), in which developers are many times confronted with cloud-specific requirements. Function signature requirements, and the usage of custom libraries that are unique to cloud providers, were identified as the two main reasons for portability issues in FaaS applications. Such reduced control over the infrastructure and tight-coupling with cloud services amplifies various vendor lock-in problems.

In this work, we introduce QuickFaaS, a multi-cloud interoperability desktop tool targeting cloud-agnostic functions development and FaaS deployments. QuickFaaS substantially improves developers' productivity, flexibility and agility when creating serverless solutions to multiple cloud providers, without requiring the installation of extra software. The proposed cloud-agnostic approach enables developers to reuse their serverless functions in different cloud providers with no need to rewrite code. The solution aims to minimize vendor lock-in in FaaS platforms by increasing the portability of serverless functions, which will, therefore, encourage developers and organizations to target different providers in exchange for a functional benefit.

**Keywords:** cloud computing; serverless computing; Function-as-a-Service; vendor lock-in; cloud interoperability; cloud orchestration; cloud-agnostic; FaaS portability.

# Resumo

A computação sem servidor abstrai o controlo da infraestrutura dos programadores e executa código a pedido com escalonamento automático onde apenas se é cobrado pela quantidade de recursos consumidos. Um dos serviços mais populares da computação sem servidor é a Função como Serviço (*Function-as-a-Service* ou FaaS), onde os programadores são muitas vezes confrontados com requisitos específicos dos prestadores de serviços de nuvem. Requisitos de assinatura das funções, e o uso de bibliotecas exclusivas ao prestador de serviços, foram identificados como sendo as principais causas de problemas de portabilidade das aplicações FaaS. O controlo reduzido da infraestrutura e a elevada dependência para com o prestador de serviços dá origem a diversos problemas de aprisionamento tecnológico.

Neste trabalho, introduzimos o QuickFaaS, uma ferramenta para *desktop* de interoperabilidade *multi-cloud* com foco principal no desenvolvimento de funções agnósticas à nuvem e na criação das mesmas na respetiva plataforma. O QuickFaaS permite melhorar substancialmente a produtividade, flexibilidade e agilidade no desenvolvimento de soluções sem servidor para múltiplos prestadores de serviços, sem o requisito de instalar software adicional. A abordagem agnóstica à nuvem irá permitir que os programadores reutilizem as suas funções em diferentes prestadores de serviços sem terem a necessidade de reescrever código. A solução visa a minimizar o aprisionamento tecnológico nas plataformas FaaS através do aumento da portabilidade das funções sem servidor, incentivando assim programadores e organizações a apostarem em diferentes prestadores de serviços em troca de um benefício funcional.

**Palavras-chave:** computação na nuvem; computação sem servidor; Função como Serviço; aprisionamento tecnológico; interoperabilidade na nuvem; orquestração de nuvem; agnóstico à nuvem; portabilidade FaaS

# Contents

# List of Figures

# List of Listings

# Acronyms

**API** Application Programming Interface. 6, 7, 9, 10, 13, 17, 18, 19, 25, 33, 34, 35, 39, 41, 42, 46, 47, 61

**AWS** Amazon Web Services. 11, 12, 13, 19, 21, 22, 40, 61

**CI/CD** Continuous Integration / Continuous Delivery. 19

**CLI** Command-line. 9, 10, 12, 18, 19, 31

**CPU** Central Processing Unit. 19

**CRUD** Create, Read, Update and Delete. 8

**DB** Database. 8, 25

**EDA** Event-Driven Architecture. 8

**ERM** Entity-Relationship Model. 18, 26, 27, 29, 31, 32, 35, 37, 39, 59

**GCP** Google Cloud Platform. 8, 9, 10, 12, 13, 19, 21, 22, 35, 40, 41, 42, 49, 50, 51, 54, 55, 56, 57

**gRPC** Google Remote Procedure Call. 46

**GUI** Graphical User Interface. 18, 30

**HTTP** Hypertext Transfer Protocol. 7, 8, 14, 20, 25, 26, 30, 31, 32, 33, 34, 35, 42, 45, 46, 55, 56, 57

**I/O** Input/Output. 36, 51, 54

**IaC** Infrastructure as Code. 11, 12, 13

**JAR** Java Archive. 40
**JDK** Java Development Kit. 23, 31
**JRE** Java Runtime Environment. 23
**JSON** JavaScript Object Notation. 23, 31, 36, 37, 38, 42, 51, 54
**JVM** Java Virtual Machine. 23, 31, 48

**MB** Megabyte. 46, 55
**MiB** Mebibyte. A mebibyte equals to $2^{20}$ or 1,048,576 bytes. 46

**OS** Operating System. 7

**POM** Project Object Model. 42

**REST** Representational State Transfer. 46, 47
**RFC** Request For Comments. 60

**SQL** Structured Query Language. 6, 8, 25

**UCI** Unified Cloud Interface. 17
**UI** User Interface. 31, 33
**UML** Unified Modeling Language. 30, 31, 32
**URL** Uniform Resource Locator. 21, 33

**VM** Virtual Machine. 2, 7

**XML** Extensible Markup Language. 23

**YAML** Yet Another Markup Language. 12, 13

# Glossary

**auto-scaling** the system's ability to accommodate larger loads, either by making hardware stronger (scale-up), or by creating additional nodes (scale-out). 7, 9, 52, 53

**booting time** the time it takes for a device to be ready to operate after being turned on. This involves loading the startup instructions, followed by the operating system. 2, 48

**debug** to identify and remove errors from computer hardware or software. 60, 61

**elasticity** the ability to grow or shrink infrastructure resources dynamically as needed. 1

**garbage collector** an automatic process to free unreachable objects from heap memory, relieves programmers from having to mark objects to be deleted explicitly. 34, 52

**hardcoded** a software development practice of embedding data directly into the source code. 51

**heap space** in Java, the heap is utilized for dynamic memory allocation of objects and Java Runtime Environment (JRE) classes at runtime. 34, 52

**load balancing** a core solution used to distribute workload across multiple backend servers. 1

**programmatically** something that is feasible to be automated using source code, rather than via direct user interaction. 9, 42, 46

**reverse engineering** the process of analysing and understanding how an existing device, process, system, or piece of software works by breaking it down into its core components. 60

**serverful** an architectural model for building long-living backend services that are usually less cost-effective than serverless solutions. 5, 52

**stateless** the application doesn't rely on in-memory state set by previous interactions. 6, 7

**timestamp** a digital record of the time a particular event occurred. 55, 56

# 1

# Introduction

This chapter will start by contextualizing this work with the introduction of the serverless computing paradigm together with its most popular implementation, called Function-as-a-Service (FaaS). Then, a list of some major motivations behind its adoption is given. Moreover, the central problem concerning FaaS solutions is described. The main contributions of this work are then provided to emphasize its significance. Furthermore, a couple of research conferences in which we had the opportunity to participate are mentioned. Finally, the document outline is presented, where the contents of the remaining chapters are briefly described.

## 1.1 Context

Serverless computing was a major technological breakthrough that has been drawing interest from the industry as well as academic institutions, largely due to the recent shift of enterprise application architectures to containers and microservices [70].

Serverless potential is sustained by the great abstraction of server management challenges with low costs [67]. Function-as-a-Service, or simply FaaS, is known as the popular implementation of the serverless computing model, where developers can compose applications using arbitrary, event-driven functions to be executed on demand.

Cloud providers assume most of the responsibilities when it comes to serverless computing. Thus, the development of systems can be more focused on business logic and less on non-functional aspects, such as elasticity, redundancy and load balancing.

## 1.2 Motivation

The main idea behind cloud serverless computing is to mitigate the need for infrastructure management while keeping control of the system configurations. Summarized below, we point out some extra reasons to embrace serverless solutions:

- ***Faster deployment and delivery.*** Developers can easily deploy serverless applications without the requirement for server administration experience [69].
- ***Auto-scaling.*** Serverless platforms assume responsibility for scaling applications in case there's an increase in demand, but also scale them back to zero to reduce costs [4, 5]. In some platforms, the scaling boundaries can be specified.
- ***Cost efficiency.*** Follows the *pay as you go* pricing model [66], where customers only pay for the consumed computational resources. There's no need to pay for idle servers or the overhead of servers creation and destruction [65], such as VMs booting time.
- ***Greener computing.*** The usage of computational resources is more efficient, less computing power is wasted on idle state.

## 1.3 Problem

Developers are many times confronted with cloud-specific requirements when developing FaaS applications. The noticeable tight-coupling between providers and serverless function services amplifies various vendor lock-in problems that discourage developers and organizations to migrate or replicate their FaaS applications to different platforms. Cloud orchestration tools can as well be contributing to the increase of multi-cloud complexity by continuously adding more and more provider-specific modules. As mentioned by the Research Cloud group of the Standard Performance Evaluation Corporation (SPEC) [74]:

"There is a need for a vendor-agnostic definition of both the basic cloud-function and of composite functions, to allow functions to be cloud-agnostic."

Otherwise, migrating a FaaS application from one provider to another would imply rewriting all the functions that make up that application, causing an impact at operational level in addition to costs. This happens mainly due to function signature requirements and the usage of custom libraries that are unique to cloud providers, resulting in non-portable solutions.

## 1.4 Contributions

This work focuses on characterizing and describing high-level cloud-agnostic models that compose FaaS platforms. These models were materialized into a multi-cloud interoperability desktop tool named **QuickFaaS**, where users can develop and deploy cloud-agnostic functions to a set of cloud providers.

By adopting a cloud-agnostic approach, developers can provide better portability to their FaaS applications, and would, therefore, contribute to the mitigation of vendor lock-in in cloud computing.

## 1.5 Research papers

Throughout the course of this work, we managed to do several research contributions. The resulting papers were entitled "*QuickFaaS: Providing Portability and Interoperability between FaaS Platforms*", and are referenced below:

- Accepted for publication by MDPI in the peer-reviewed scientific journal *Future Internet*, within the special issue "*Distributed Systems for Emerging Computing: Platform and Application*" [60].
- Included in the proceedings of the $9^{th}$ *European Conference On Service-Oriented And Cloud Computing* (ESOCC), in the projects track, to be published by Springer in the Communications in Computer and Information Science (CCIS) book series [75]. The conference was hosted in an online format, where we got the chance to present this work and receive valuable feedback from research experts.
- Presented as a poster at the $17^{th}$ *Iberian Conference on Information Systems and Technologies* (CISTI), held in the Technical University of Madrid (UPM), Spain.

## 1.6 Document outline

The remainder of this document is organized into five main chapters as follows: Chapter 2 gives a general overview of how serverless computing operates today, and how cloud providers handle both the development and deployment of FaaS applications. It also provides some background to the problems that need to be addressed for the purposes of this work. In the last part of the chapter, several tools and related works

concerning multi-cloud interoperability and FaaS portability are described. Chapter 3 defines a cloud-agnostic approach to model FaaS applications using entity-relationship models, together with a number of challenges to be taken into account when doing so. This is followed by the illustration of a set of use cases that highlight the main benefits of cloud-agnostic functions. Chapter 4 introduces the desktop tool QuickFaaS and details the proposed solution to achieve interoperability and portability between FaaS platforms. This includes an explanation of different class diagrams that compose the uniform programming model. Chapter 5 evaluates different metrics to measure the impact of a cloud-agnostic approach on the function's performance by comparing it to a cloud-non-agnostic one. It goes into detail on what metrics were utilized, how were the tests performed, and an analysis of the obtained results is given. At the end of the chapter, the main adversities faced during the data collection process are described. Chapter 6 summarizes the project's primary achievements but also presents some drawbacks to the proposed solution. The future work concludes the chapter by addressing feasible improvements to the tool and its models.

# 2

# Background and Related work

For this chapter, we will be detailing how serverless computing operates today, with emphasis on Function-as-a-Service. We also survey how some of the most popular cloud providers deal with the development and deployment of FaaS applications. Moreover, we provide some background to the problems that need to be addressed for the purposes of this work. Finally, several existing tools and closest related works concerning multi-cloud interoperability and FaaS portability are described.

## 2.1 Serverless computing

Serverless computing is emerging as a new and successful paradigm that promotes total absence of control over the deployment and execution of services. Even though we call it server**less**, these platforms do run servers, but only for short periods of time. In contrast to *serverful* architectures, the entire lifespan of a server only exists within the life cycle of a given execution request.

It is also known for being very cost-efficient, where customers only pay for what it's consumed in terms of computational resources — *pay as you go* pricing model. When running small, simple, self-contained applications, it makes no sense to use our own hardware or implement our own provisioning for these types of applications. In serverless technologies, cloud providers take responsibility for dynamically allocating servers as demand spikes or drops, with the possibility of scaling them back to

zero. Because servers aren't always running, there's no need to pay for idle or inactive servers, unlike on-premises machines, shared servers, or rented virtual machines.

Nowadays, we can find all kinds of services based on serverless computing. For instance, Amazon DynamoDB and Google Cloud Firestore, are two of the most popular NoSQL database serverless services. Even full-stack serverless applications can be deployed in just a few minutes. Google Cloud provides a practical tutorial on how Cloud Run service and MongoDB come together to enable a completely serverless application development experience using the MEAN stack (*MongoDB–Express.js–AngularJS–Node.js*) [13].

This work focuses on the most basic and commonly used serverless service, called Function-as-a-Service. This service popularized the serverless computing paradigm, and it's explained in more detail in the upcoming subsection.

### 2.1.1 Function-as-a-Service

Serverless functions are the general purpose element in serverless computing today, and lead the way to a simplified programming model for the cloud. At the very start, everyone used to build monoliths, with N-tier architectures. Monoliths are heavyweight, less scalable, and have a tight coupling between components. We then broke these down into microservices. They are single-purpose services focused on being composable, fault-tolerant and easy to scale-up. Serverless functions are the next step in the evolution. These functions are usually lightweight, short-lived, and stateless.

Typically, serverless functions are deployed with the help of a popular cloud serverless service called Function-as-a-Service (FaaS), where applications and all their business logic can be constructed as a composition of multiple functions. These functions are executed in response to various types of events. FaaS provides high-level abstractions of distributed computing elements, reducing the need for users to be experts in distributed systems, or to manage complex microservice-based architectures themselves [22]. Some of the most popular use cases for FaaS are APIs for web and mobile applications, multimedia processing, data processing, and the Internet of Things (IoT).

In Figure 2.1, we present the generic FaaS architecture that reveals what mechanisms are involved between the occurrence of an event and the function getting executed.

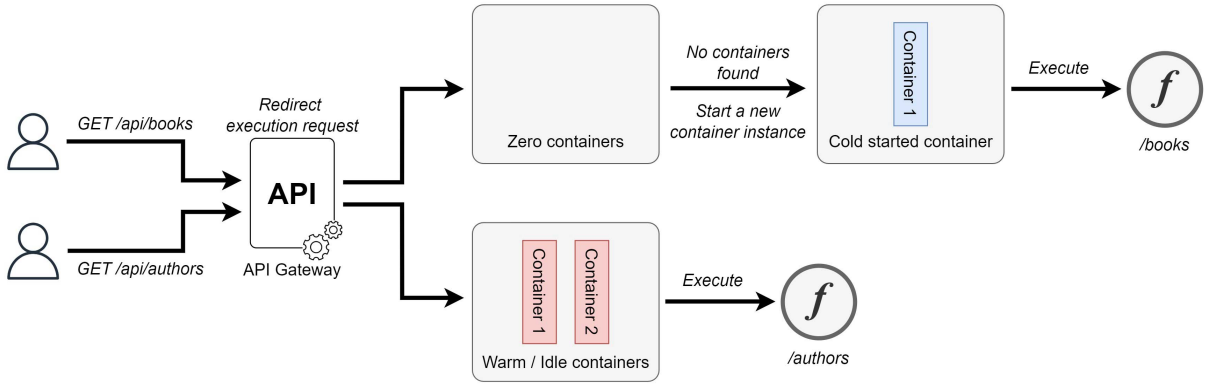



Figure 2.1: Generic FaaS architecture

Triggering events are often generated by user actions. In the above example, two HTTP requests were made. The first one, triggers the execution of the *books* function, while the second one, triggers the execution of the *authors* function. The API gateway component acts as a single entry point to provide an external route to the deployed FaaS resources. Then, a mechanism checks for the availability of warm/idle container instances. If there are none, a new container instance must be allocated to serve the recently arrived execution request [17].

Containers are self-contained sealed units of software that package everything that is necessary to run the code, this includes dependencies, runtimes, etc. Containers virtualize at the OS level and share the same kernel of the host computer, while VMs virtualize at the hardware level. They are also much more lightweight than VMs. The time between checking for existing warm containers, and starting up a new instance, including the preparation of the execution environment, is known as the cold start latency.

The *books* function goes through a cold start in order to get executed, while the *authors* function wastes no time in cold boots by reusing an existing container from recent executions. To reduce costs and resource wastage, container instances are offloaded after remaining idle for a certain period of time, meaning that the next invocation will result in a cold start. This is done by an auto-scaling mechanism responsible for monitoring the demand and supply by elastically adding or removing container instances.

The execution of a serverless function is intrinsically stateless — one function invocation should not rely on in-memory state set by a previous invocation — although a container may last for more than one invocation. This is due to the number of container instances, and the respective lifetime, being fully managed by the cloud infrastructure. If there's a need to share state across function invocations, then a secondary service dedicated to persist shared data should be accessed during execution.

### 2.1.2 Event-driven architecture

Event-Driven Architecture (EDA) is an architectural paradigm that supports the production, detection and reaction to a system state change, known as events. The execution of a serverless function can be triggered by the occurrence of many different types of events, most of them being generated within the cloud infrastructure. In the EDA system, events can be transmitted between components and services, such as FaaS and a Storage Service, whenever a storage triggering event occurs [21].

Defined below, are the more commonly used types of event triggers offered by cloud providers:

- ***HTTP.*** Serverless functions that rely on this type of trigger can be invoked using the primary HTTP methods (CRUD operations). These functions can require some sort of authentication since they can be invoked by different kinds of identities, originating in different places. Usually, this type of triggering event is not originated by the cloud infrastructure.
- ***Storage.*** A serverless function can be triggered in response to uploads, updates, or deletes of blobs in a storage resource. Useful blob information can then be read during execution.
- ***Database.*** Database events can also trigger the execution of serverless functions. Usually, these are NoSQL databases due to being less complex and easier to detect changes in a particular document (e.g. Cloud Firestore, Azure Cosmos DB).
- ***Pub/Sub.*** The publication of messages to Pub/Sub topics is one of the most used mechanisms to trigger serverless functions. This pattern uses asynchronous messaging, which is an effective way of decoupling senders from consumers. It also facilitates the implementation of *per-message parallelism*, by having multiple serverless function instances (subscribers) processing messages of the same topic.

Events can carry a state or data to be read using the function's signature parameters, thus the types of parameters can vary depending on the configured event.

## 2.2 FaaS providers

Every provider imposes its own prerequirements that need to be established before enabling the deployment of FaaS applications. In this section, we enumerate those requirements for two of the most popular cloud providers: MsAzure and GCP.

### 2.2.1 MsAzure – Azure Functions

The developer will be asked to provide details about the following items before being able to deploy a FaaS resource to MsAzure: (i) a Microsoft account, (ii) an Azure account, (iii) at least one active subscription associated with the Azure account, (iv) a *resource group* to hold various types of resources, and (v) a *storage account*, responsible for storing function sources. After all these requirements are met, the developer is finally able to create a *function app*, in a specified location, capable of holding multiple azure functions for a single runtime.

Most configurations used by azure functions are established during the creation of the *function app*. Some of these configurations are common between cloud providers, e.g., the runtime and respective version, the instance location, etc. While others are specific to MsAzure, such as the subscription ID, the hosting plan type (consumption plan, premium plan, dedicated plan), the option to enable the Azure Monitor Application Insights, and others. The event trigger is only specified when deploying an azure function to the *function app*.

The deployment environment for azure functions varies depending on the selected runtime. For instance, the deployment of azure functions for Node.js can be done directly on the Azure portal, while for Java or Python, this is not possible. To overcome these restrictions, the developer must install the Azure Functions Core Tools, together with the provider-specific tool Azure CLI [45]. The Azure Functions Core Tools includes a version of the same runtime that powers Azure Functions runtime, meaning that developers are able to test their azure functions locally, before being deployed. This can be done using the command prompt, or by installing the Azure Functions extension in Visual Studio Code.

### 2.2.2 Google Cloud Platform – Cloud Functions

In order to deploy cloud functions to GCP, developers will first need: (i) a Google account, (ii) a project responsible for managing all types of services, (iii) a billing account, that can be linked to multiple projects, and (iv) the Cloud Functions API enabled. The solution we propose will also require developers to enable the Cloud Resource Manager API [12], which allows to programmatically manage metadata for GCP resources.

Apart from the usual resource configurations, such as the name or the location, the developer can also specify how much memory the function can use, the timeout duration of the function, and the auto-scaling boundaries (maximum/minimum number

of instances). This time, serverless functions don't need to be aggregated in a parent resource, such as the *function app* from MsAzure.

Moreover, there are no restrictions to what runtimes should be used in order to develop and deploy cloud functions directly on the portal, which avoids the need to install extra software. GCP offers a provider-specific tool, called gcloud, that enables FaaS deployments through the command-line. However, it's not mandatory to be installed in order to do so, unlike Azure CLI for certain runtimes.

## 2.3 Vendor lock-in

Systems migration to cloud computing environments has been gaining attraction from organizations over the past few years. Cloud providers offer organizations proprietary cloud-based services that have different specifications from one provider to another, such as specific technology solutions, remote APIs, etc. [84]. As a consequence of this, clients become dependent (locked-in) on certain services and are unable to move seamlessly from one vendor to another [24, 83].

For the ongoing work, the effects of vendor lock-in are relevant to point out:

- Struggle when switching between cloud providers due to the impact at operational level, in addition to costs.
- If the provider's quality of service declines, or never meets the desired requirements to begin with, the client will have no choice other than to accept the conditions.
- A provider may impose price increases for the services, knowing that their clients are locked-in.

Multi-cloud solutions tend to attenuate some of these issues by distributing workloads and making them more independent of the underlying cloud infrastructure.

## 2.4 Multi vs. Poly cloud strategies

Organizations can provide better reliability to their services by adopting a multi-cloud strategy. In a scenario of a cloud provider going down, some services deployed in another cloud provider can still be available for usage [50], preventing single points of failure (SPOF).

Poly cloud is often mistaken for multi-cloud. In a poly cloud approach, the system benefits from services of different cloud providers that best suit a specific use case. It targets different cloud providers in exchange for a functional benefit [54, 55]. For some applications, adopting a poly cloud strategy is a must in order to get access to specific services that are exclusive to a single cloud provider. Also, when using poly cloud, organizations can pick the most affordable services from different providers, thus reducing costs.

Both strategies require developers to learn about multiple service providers and how they differ. Cloud professionals are in high demand, the recruitment process for cloud engineers with expertise in a single cloud provider is getting more and more competitive, so, it becomes even more problematic when the recruitment targets experts in multiple cloud environments. There's a need for developing solutions that can help in managing multi-cloud complexity, rather than amplifying it.

For the purposes of this work, we will consider both strategies in the same manner from now on, and use the term "multi-cloud" in application of either scenario.

## 2.5 Related work

The future evolution of serverless computing will presumably be guided by efforts to provide abstractions that simplify cloud programming [26]. To the best of our knowledge, no published work suggests a uniform approach to model FaaS applications or a way to characterize cloud-agnostic functions development and deployment, while avoiding the installation of provider-specific tooling.

There are already a couple of tools and studies concerning cloud orchestration that have an important role in providing developers a better management of multi-cloud environments by solving some of the problems discussed in this work. Provider-specific modeling tools, such as AWS Cloud-Formation [6], focuses solely on their own platform. As a result, additional software is required for the coordination and deployment on multi-cloud environments. The following subsections detail various related tools and mechanisms that facilitate multi-cloud usage.

### 2.5.1 Terraform

Terraform [76] is probably the developer's number one choice for an infrastructure as code (IaC) tool. It provides open-source software for cloud service management

with a consistent CLI workflow. However, when it comes to FaaS development and deployment, it is far from being cloud-agnostic.

Each cloud provider has its own dedicated configuration file (*.tf), that needs to be strictly followed in order to successfully deploy a serverless function to the cloud. Folder structures that contain the function's source code are also cloud-specific. No custom libraries or signatures are provided to facilitate the development of serverless functions either.

The authentication process also varies depending on the cloud provider we want to work with. For instance, Terraform suggests the installation of the CLI tool gcloud as the primary authentication method for GCP [78]. While for MsAzure, Terraform recommends the usage of Azure Active Directory to generate secret authentication tokens [77]. This makes developers having to adapt their systems to a specific authentication process every time there's a switch to a new cloud provider.

### 2.5.2 Serverless Framework

Another example of a cloud orchestration tool is Serverless Framework [68]. Just like in Terraform, this framework enables the automation of infrastructure management through code, this time using YAML syntax instead of Terraform language. Serverless Framework focuses on app-specific infrastructure, while Terraform allows the management of a full-fledged infrastructure (e.g. defining networking, servers, storage, etc.).

The framework supports deployments to AWS out of the box, deploying to other cloud providers requires the installation of extra plugins. Even though Serverless Framework is dedicated to managing serverless applications, the deployment models are not cloud-agnostic. The models describe provider-specific services, event types, etc., ending up sharing the same problems highlighted previously with Terraform [24].

### 2.5.3 Pulumi

Pulumi [56] is a modern infrastructure as code platform that allows developers to use familiar programming languages and tools to build, deploy, and manage cloud infrastructure. As a language-neutral IaC platform, Pulumi doesn't force developers to learn new programming languages, nor does it use domain-specific languages. Just like previous tools, it requires the installation of provider-specific software for authentication and deployment purposes.

Pulumi introduces a new approach for simplifying the development of serverless functions in the form of lambda expressions, they call it *Magic Functions* [59]. Still, both the function signatures and event trigger libraries that are required by *Magic Functions* are not cloud-agnostic.

Developers from Pulumi have also worked on a new framework named *Cloud Framework* [57], which lets users program infrastructure and application logic, side by side, using simple, high-level, cloud-agnostic building blocks. It provides a Node.js abstraction package called *@pulumi/cloud* [58], which defines common APIs to all providers. At the moment, the library is in preview mode, where only AWS is supported. MsAzure support is currently being worked on, and it's in an early preview state. They also intend to support GCP in the future.

Nonetheless, we found Pulumi's solution to be a step ahead of the previous IaC tools, in the sense that it makes a real attempt to provide a cloud-agnostic way to develop cloud applications, with *Cloud Framework*, as well as offering a flexible and simple path to serverless, using *Magic Functions*. The ability to specify every deployment configuration using familiar programming languages is also very convenient for developers, avoiding the need to use YAML configuration files.

### 2.5.4 OpenFaaS

OpenFaaS is a framework for building serverless functions on the top of containers through the use of docker and kubernetes [51]. With OpenFaaS, serverless functions can be managed anywhere with the same unified experience. This includes on the user's laptop, on-premises hardware, or by creating a cluster in the cloud.

Kubernetes do the heavy work by enabling developers to build a scalable, fault-tolerant, and event-driven serverless platform for applications. There's even a tutorial explaining how to install and use OpenFaas on an Azure Kubernetes Service (AKS) cluster [47]. GCP offers a similar service called Cloud Run, where OpenFaaS functions can be deployed as well [33]. This tool contrasts with the ones presented before, in the sense that it doesn't provide any sort of cloud orchestration mechanism to abstract the complexity of provider-specific tools.

Technically, OpenFaaS's solution enables serverless functions to run on multiple cloud environments without the need to change a single line of code, by providing custom function signatures and libraries. Function templates for various programming languages can be found in their *templates* GitHub repository [53].

However, despite supporting many different kinds of events [52], OpenFaaS functions cannot be directly triggered by the majority of events originating within the cloud infrastructure, such as storage or database events. Workarounds can be implemented, but wouldn't be as optimized as triggering a function using the default FaaS platform from cloud providers. Most OpenFaaS use cases rely on HTTP events for triggering serverless functions.

The OpensFaaS framework does not provide portability to serverless functions using the existing FaaS from cloud providers; it instead creates a custom FaaS platform on top of a different service to do so. Hence, that is the reason why we did not consider this to be a valid solution for the problem we intend to solve with this work. For instance, the usage of a kubernetes cluster service may be inappropriate for developers that start from scratch and want to build a less complex FaaS application.

### 2.5.5 SEAPORT method

Manual portability assessment is inefficient, error-prone, and requires significant technical expertise in the domains of serverless and cloud computing. To simplify this process, a work from the University of Stuttgart [63] specifies a method called SEAPORT (*SEreverless Applications PORtability assessmenT*) that automatically evaluates the portability of serverless orchestration tools with respect to a chosen target provider or platform. The method can be optimized over time by testing more and more heterogeneous serverless use case applications.

The SEAPORT method introduces a *CAnonical SErverless* (CASE) model, which is the result of transforming the obtained deployment model from a certain serverless orchestration tool into a provider-agnostic format. Yet, we find the represented CASE model far too abstract for our needs, making it non-reusable. For instance, the model doesn't detail the various composing elements of a function definition, it only specifies the function's programming language and the event category that triggers its execution. Furthermore, it doesn't illustrate the abstraction of different types of authentication mechanisms used by cloud orchestration tools in order to get access to resources from different providers. Lastly, the model represents some extra entities that are only convenient to facilitate the portability evaluation of a serverless application, making them unrelated to this work.

### 2.5.6 TOSCA modeling

Some cloud orchestration tools, such as Cloudify [10] and Alien4Cloud [1], follow the Topology and Orchestration Specification for Cloud Applications (TOSCA) standard [82], created by the industry group OASIS. TOSCA defines the interoperable description of services and applications hosted on the cloud, thereby enabling portability and automated management across cloud providers regardless of the underlying platform or infrastructure. TOSCA modeling could be an alternative to the entity-relationship models presented in this work, there is even a work that promotes the usage of TOSCA standard to build deployment models for serverless applications [71].

However, TOSCA models are usually focused on representing the interactions between technologies and services hosted in different cloud providers using normative relationship types, such as *connectsTo* or *hostedOn*, but also the operations that each node type may have: *create, configure, start, stop,* and *delete*. FaaS deployments don't usually require interaction with many different types of services, nor require communication between nodes hosted in different providers, making TOSCA modeling not very suitable to represent our cloud-agnostic solution.

## 2.6 Summary

This chapter provided a general overview of serverless computing, with a special focus on Function-as-a-Service. A description on how some cloud providers handle the development and deployment of FaaS applications was given. We also considered several closest related works and tools to check whether they propose a valid solution to the problems addressed in this work.

For the next chapter, we will be introducing some models that identify key characteristics for achieving interoperability and portability between FaaS platforms, together with some challenges and use cases.

# 3

# Models

The current chapter will start by giving an overview of the cloud-agnostic approach and respective modules that need to be unified in order to achieve interoperability and portability between FaaS platforms. Next, several challenges that are experienced with the adoption of FaaS applications are described. We then reveal and detail the various entities that model each module of the cloud-agnostic solution. The last section defines a number of use cases that exemplify the usage of cloud-agnostic functions in a practical scenario to highlight their main benefits.

## 3.1 Overview

From the analysis of cloud interoperability by Parameswaran and Chaddha [11], two major approaches have emerged to ensure the developer needs are met:

1. Creation of a unified cloud interface (UCI).
2. Creation of a cloud orchestration platform.

A UCI is an API about APIs, in the sense that it provides an interface which a developer can make use of, while under the hood it is interacting with unique APIs from different cloud providers to execute specific operations. The proposed solution is considered to be an UCI, since it abstracts the complexity of multiple cloud providers into a single tool.

We decided to design a few entity-relationship models (ERMs) so that we could have a visual starting point of the cloud-agnostic solution. An ERM represents the information structure of the problem domain in terms of entities and relationships. We adopted a top-down approach when designing the ERMs, the more general one can be found in Figure 3.1. This model illustrates the three main modules that compose the cloud-agnostic solution. Each module is then described in Section 3.3.

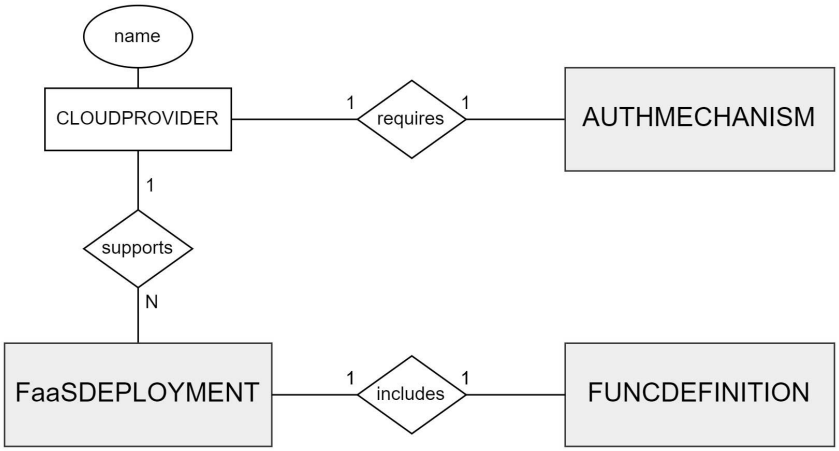



Figure 3.1: Abstract ERM of the cloud-agnostic solution

Achieving interoperability between FaaS platforms requires a certain number of permissions for accessing platform-specific APIs. These permissions can only be granted by the user, thus, an authentication mechanism needs to be provided before being able to interact with any of those APIs (*AUTHMECHANISM* entity).

Furthermore, each cloud provider uses its own strategies and services to enable FaaS deployments using a certain mechanism — manual GUI deployment, ZIP deployment, CLI deployment, etc. — hence the need to establish a deployment process for each platform, preferably a common one (*FaaSDEPLOYMENT* entity).

Finally, for every FaaS deployment, a function definition needs to be provided for execution (*FUNCDEFINITION* entity). This definition should be as independent as possible from the selected cloud provider in order to improve the portability of FaaS applications.

## 3.2 Challenges

Described below, are several challenges that are faced by developers when adopting FaaS solutions. These challenges need to be taken into account when designing a uniform model for FaaS applications as well:

- ***Custom function signatures.*** Every cloud provider imposes its own function signature depending on the programming language and trigger selected by the developer. The function's implementation can become deeply dependent on the provider's specific requirements [24], resulting in portability-related issues.

- ***Unique libraries.*** There are no common libraries shared between cloud providers that could attenuate portability issues when developing serverless functions to multiple platforms. Library dependencies are introduced not only for processing custom data types but also for interacting with provider-specific APIs [24]. Switching between cloud providers requires the developer to adapt and study new documentation, making him less productive.

- ***Provider-specific deployment environments.*** Each cloud provider decides where and how service deployments can be performed. For instance, up until this date, the cloud provider MsAzure does not support the deployment of a serverless function written in Java or Python directly on the Azure portal [9], while Google Cloud Platform does. The workaround requires the installation of the provider-specific tool named Azure CLI. Each provider offers its own CLI tool to manage cloud resources: AWS CLI, Azure CLI, gcloud from GCP, etc. The variety of tooling is also not ideal for CI/CD pipelines that require multi-cloud usage.

- ***Discrepancy in deployment configurations.*** During a service deployment configuration stage, developers are confronted with different payment plans and detailed hardware specifications to set up, such as the size of the allocated memory, or the number of CPU cores available [72]. The amount of providers and their intrinsic variability results in a discrepancy between configurations that do not facilitate multi-cloud usage [73].

- ***Different service naming terminologies.*** Service naming terminologies vary between cloud providers, even when they have the same resource type and offer similar features. Cloud developers may struggle to make correlations between services from different providers. In FaaS, the differences are not that noticeable, a function in GCP is called a *cloud function*, while in AWS they call it a *lambda function*. In other services, it can be a bit more unclear, a storage resource in MsAzure is called a *container* (associated with a *storage account*), while in GCP they call it a *bucket*.

To evidence the differences between function signatures, exemplified below, in Listings 3.1 and 3.2, are two simple use cases of serverless functions written in Java. Both

functions are triggered by HTTP requests and can be deployed to MsAzure and Google Cloud Platform, respectively. The end result will be the same for both functions.

Listing 3.1: Java "Hello world!" – Azure function, MsAzure

```
public class Function {
    @FunctionName("HttpExample")
    public HttpResponseMessage run(
            @HttpTrigger(name = "req", methods = {HttpMethod.GET},
                authLevel = AuthorizationLevel.ANONYMOUS)
            HttpRequestMessage<String> request,
            ExecutionContext context) {
        return request
                .createResponseBuilder(HttpStatus.OK)
                .body("Hello world!").build();
    }
}
```

Listing 3.2: Java "Hello world!" – Cloud function, GCP

```
public class Function implements HttpFunction {
    @Override
    public void service(HttpRequest request, HttpResponse response)
            throws IOException {
        BufferedWriter writer = response.getWriter();
        writer.write("Hello world!");
    }
}
```

By introducing such wrapping around the actual business logic, functions can become deeply dependent on provider's requirements.

## 3.3 Cloud-agnostic models

This section reveals the cloud-agnostic entities that compose each module of the abstract model illustrated previously in Figure 3.1. The models were separated into two subsections, cloud interoperability and FaaS portability. Both the *authentication mechanism* and the *FaaS deployment* models relate to cloud interoperability due to requiring the exchange of information between two systems, the application and the cloud provider. On the other hand, the *function definition* model represents the entities and respective attributes that determine the definition of a cloud-agnostic function, thus being related to the portability of FaaS applications.

### 3.3.1 Cloud interoperability

Establishing a proper authentication mechanism is usually the very first step towards achieving interoperability with cloud providers. In Figure 3.2, we represent the entities and respective attributes that model the authentication mechanism based on the OAuth 2.0 protocol. This protocol is commonly used by different types of applications to authenticate users in various platforms, such as the three most popular cloud vendors: AWS, GCP and MsAzure.

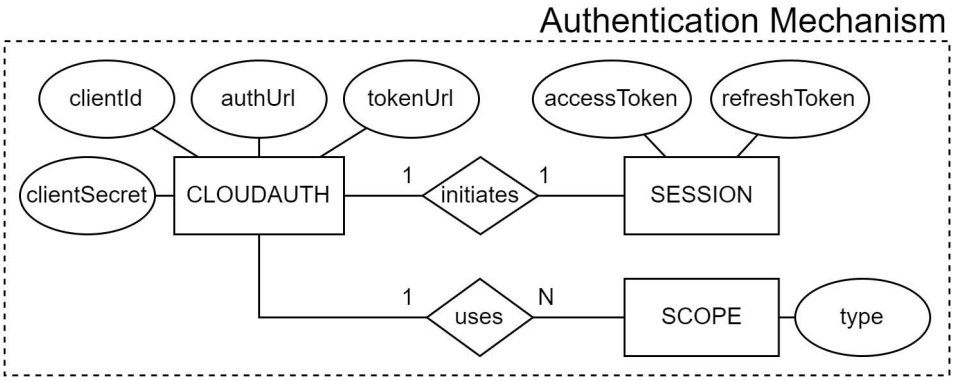



Figure 3.2: Authentication Mechanism ERM

The trust relationship between the tool and the provider's identity platform is established once the developer registers it as an application in the respective cloud provider. This trust is unidirectional, the application trusts the provider identity platform, and not the other way around. After registration, a client ID and a client secret are randomly generated (*clientId* and *clientSecret* attributes from *CLOUDAUTH* entity). The client ID is a public identifier for the application within the identity platform, while the client secret is confidential to the application and should be kept private. Otherwise, a malicious software could reuse the client secret to impersonate the application and gain access to the user's data [64].

With OAuth 2.0, an application can request one or more scopes (*SCOPE* entity), this information is also presented in the consent screen during the user authentication process. Once the user is successfully authenticated, an access token is requested and issued to the application (*accessToken* attribute from *SESSION* entity) using a specific token URL (*tokenUrl* attribute from *CLOUDAUTH* entity). The extent of the application's access is limited by the scopes granted. OAuth access tokens normally last for about an hour in both GCP and MsAzure [80, 81]. Once expired, a refresh token can be requested (*refreshToken* attribute from *CLOUDAUTH* entity). Refresh tokens usually have a much longer lifetime that can last for several days before expiring.

Facilitating cloud service deployments to multiple providers helps in managing multi-cloud complexity, thus contributing to cloud interoperability. Considering that this work focuses on characterizing interoperability to FaaS platforms, Figure 3.3 only illustrates cloud entities that participate in the deployment of a FaaS application.

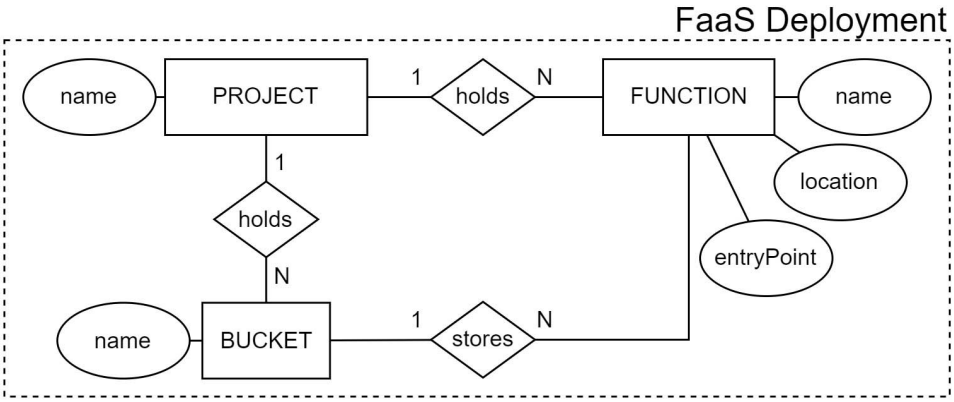



Figure 3.3: FaaS Deployment ERM

Usually, a function needs to be linked to a storage resource. We adopted the same naming terminology from AWS and GCP by calling it a bucket (*BUCKET* entity). These buckets can have various purposes: storing every version of the function's source code, storing execution logs, etc. The user must have at least one bucket created before being able to deploy the FaaS resource to a certain location (*location* attribute from *FUNCTION* entity). A resource location is wherever the resource resides, preferably as close as possible to the end user. As for the *entryPoint* attribute, from the *FUNCTION* entity, it specifies the entry point to the FaaS resource in the source code. This is the code that will be executed when the function runs. A project (*PROJECT* entity), is simply a holder for various types of resources from different cloud services.

When adopting this model, developers have to consider the differences in service naming terminologies. In MsAzure, a *PROJECT* corresponds to a *resource group*, a *FUNCTION* to a *function app*, and a *BUCKET* to a *container* associated with a *storage account*.

### 3.3.2 FaaS portability

The usage of provider-specific function signatures, as well as libraries, can be considered as the two main causes for portability issues that limit a serverless function to only work on a single cloud provider. To counter these problems, we propose a model for the development of cloud-agnostic functions, that is, functions that can be reused in multiple cloud providers with no need to change a single line of code.

The entities and respective attributes that model a cloud-agnostic function definition are represented in Figure 3.4.

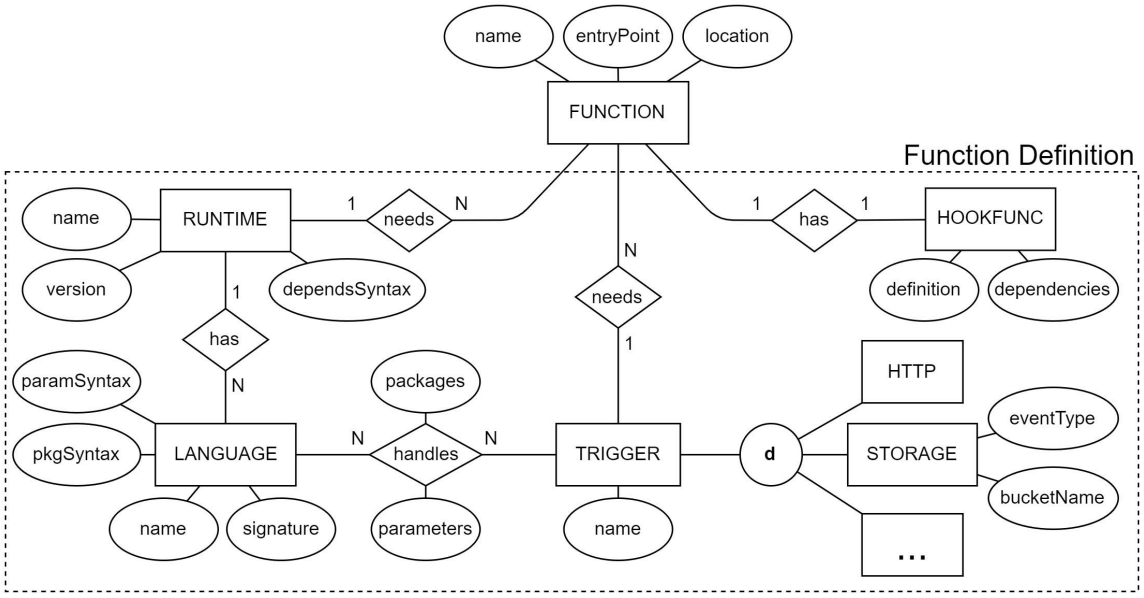



Figure 3.4: Function Definition ERM

The *FUNCTION* entity is the exact same entity as the one illustrated in Figure 3.3, and should not be confused with the *HOOKFUNC* entity, which is the cloud-agnostic function defined by the developer (*definition* attribute from *HOOKFUNC* entity).

Starting a new function instance involves loading a runtime environment (*RUNTIME* entity), capable of running code from multiple programming languages (*LANGUAGE* entity). For instance, Kotlin JVM and Java programming languages both use the Java Development Kit (JDK), which includes the Java Runtime Environment (JRE).

The developer has the option to specify a list of external dependencies (*dependencies* attribute from *HOOKFUNC* entity), which needs to be analyzed before every deployment to check whether there are any missing dependencies to be downloaded from a remote repository. These should follow the appropriate syntax when specified, which can vary depending on the chosen runtime (*dependsSyntax* attribute from *RUNTIME* entity). For instance, Maven dependencies for Java projects are specified in the POM file using XML, while Node.js dependencies are specified in the *package.json* file using JSON.

For every programming language, a cloud-agnostic function signature needs to be established (*signature* attribute from *LANGUAGE* entity). Defined below, in Listings 3.3 and 3.4, are two examples of cloud-agnostic function signature skeletons used for Java and JavaScript programming languages, respectively.

Listing 3.3: Java cloud-agnostic signature skeleton

```
<packages>

public class MyFunctionClass {
    public void myFunction(<parameters>) {
        <definition>
    }
}
```

Listing 3.4: JavaScript cloud-agnostic signature skeleton

```
<packages>

module.exports = function(<parameters>) {
    <definition>
}
```

The words between the angle brackets represent the mutable parts of a function, thus, they can change from one deployment to another. Both the parameters and packages are established based on the trigger and programming language selected by the developer (*packages* and *parameters* attributes from the $LANGUAGE \leftrightarrow TRIGGER$ relation). These should be referencing the cloud-agnostic libraries for a specific event trigger. Both of them are defined using the appropriate language-specific syntax (*paramSyntax* and *pkgSyntax* attributes from *LANGUAGE* entity). For instance, in Kotlin, function parameters are specified using the "*name: type*" syntax, while Java follows the "*type name*" syntax instead.

## 3.4 Use cases

We defined three use cases that exemplify the usage of cloud-agnostic functions in a practical scenario to highlight their main benefits. The first use case, illustrated in Figure 3.5, was based on a frequently-described example of an event-driven FaaS-based application, called the *thumbnail generation* [24].

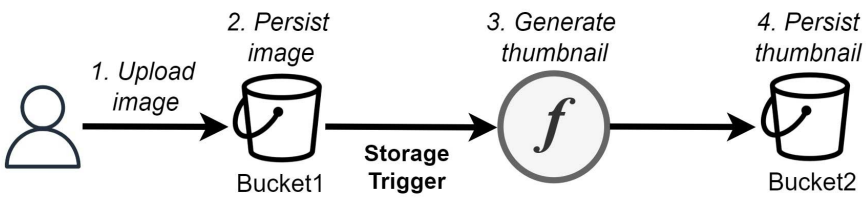



Figure 3.5: Use case 1 – *thumbnail generation*

The use case starts with the upload of an image file to be persisted in a storage bucket (*Bucket1*). The upload action triggers the execution of a cloud-agnostic function responsible for generating and storing a new image thumbnail in a second storage bucket (*Bucket2*). Prior to the thumbnail generation, the function makes a remote call to the provider's storage service to read the uploaded image bytes that triggered its execution. The thumbnail generation operation simply consists in cutting the image width in half using common Java libraries.

This first use case intends to demonstrate the high portability of a full cloud-agnostic function between different cloud providers (multi-cloud approach), even when being triggered by an event originated within the cloud infrastructure (storage event).

The second use case, represented in Figure 3.6, illustrates two cloud-agnostic functions that interact with each other in a poly cloud approach. The purpose of the use case is to translate and store short pieces of text into a database, we named it *store translation*.

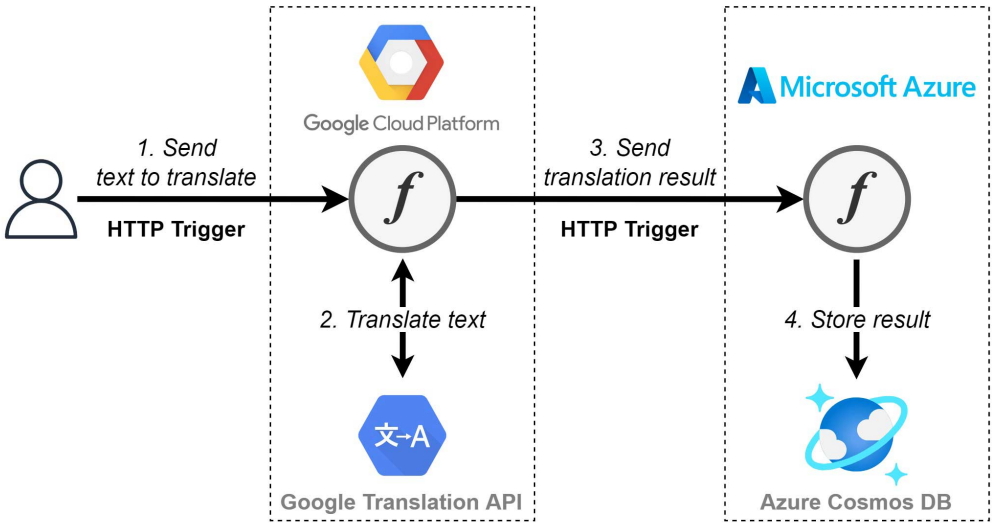



Figure 3.6: Use case 2 – *store translation*

The use case starts by sending an HTTP request with a piece of text to be translated by the first cloud-agnostic function deployed in Google Cloud Platform. Because we are already in Google's environment, no extra authentication is required when accessing the Cloud Translation API [14]. The translation result is then sent, once again, via HTTP, to a second cloud-agnostic function, this time deployed in MsAzure. The second function has the single task of storing the translation result into a NoSQL database, using the Azure Cosmos DB service.

Because both functions are using the same event trigger type, their function signatures will be the same as well. With this use case, we intend to demonstrate that even when

using a poly cloud approach, cloud-agnostic functions can help developers to focus more on business logic and less on provider-specific requirements, such as following a sophisticated function signature.

The third and last use case is called *search blobs*, represented in Figure 3.7. This use case was designed exclusively for the purpose of the performance testing evaluated in Chapter 5.

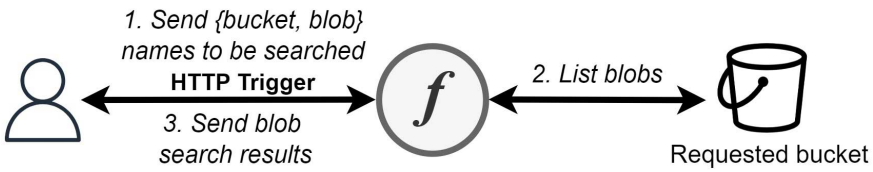



Figure 3.7: Use case 3 – *search blobs*

The cloud-agnostic function gets triggered via HTTP requests, in which the users must specify the blob name they want to search for, together with the bucket name. The *listBlobs* operation, found in the cloud-agnostic libraries, is then executed to retrieve a list of blobs contained in a given bucket, which requires a remote call to the storage service. Finally, a simple search is made to check whether any of the blob names include the requested *string*. The search results are then sent as a response to the execution request.

No other use case described in this work gets triggered by HTTP requests while being fully cloud-agnostic, making them inappropriate for performance testing. For instance, if the *thumbnail generation* use case was chosen instead, it would take an excessive amount of time to do multiple storage trigger executions when compared to a HTTP-triggered function. This happens as a result of the extra file upload time, plus the latency experienced during the communication between the two cloud services ($FaaS \leftrightarrow StorageService$).

## 3.5 Summary

In this chapter, we introduced several ERMs that compose the cloud-agnostic solution to achieve interoperability and portability between FaaS platforms. We also specified different challenges that need to be taken into account when doing so. To conclude this chapter, a number of use cases were presented with the purpose of highlighting the main benefits that cloud-agnostic functions can offer in a practical scenario.

The next chapter will be detailing the proposed solution by introducing a desktop tool as well as the uniform programming model for FaaS applications. The model comes as a result of the ERMs illustrated in this chapter.

# 4

# Implementation

For this chapter, we will be detailing the proposed cloud-agnostic solution to achieve interoperability and portability between FaaS platforms, by introducing the desktop tool QuickFaaS. It will start by describing the system's architecture and technologies. Furthermore, the uniform programming model materialized by QuickFaaS is characterized and divided into three smaller class diagrams. Each diagram corresponds to a specific ERM, previously illustrated in Section 3.3. The full programming model can be found in Appendix A.2

## 4.1 QuickFaaS

QuickFaaS is a multi-cloud interoperability desktop tool targeting cloud-agnostic functions development and FaaS deployments. Our mission, with QuickFaaS, is to substantially improve developers' productivity, flexibility and agility when developing serverless computing solutions to multiple providers. The cloud-agnostic approach allows developers to reuse their serverless functions in different cloud environments, with the convenience of not having to rewrite code. This solution aims to minimize vendor lock-in in FaaS platforms while promoting interoperability between them.

Initially, the tool will support automatic cloud-agnostic function packaging and FaaS deployments to MsAzure and Google Cloud Platform. The expansion to other cloud providers is feasible. For the following subsections, we describe the system's architecture, as well as the technologies utilized by QuickFaaS.

### 4.1.1 Architecture

An overview of the system's architecture can be visualized in Figure 4.1. This type of diagram is known as *deployment diagram*, typically designed using UML, which is a visual language for specifying, constructing, and documenting the artifacts of systems. *Deployment diagrams* are often used for describing the hardware components where software is deployed. In our case, the diagram focuses on revealing the main software components and technologies behind QuickFaaS, and how these components interact with each other and with remote services.

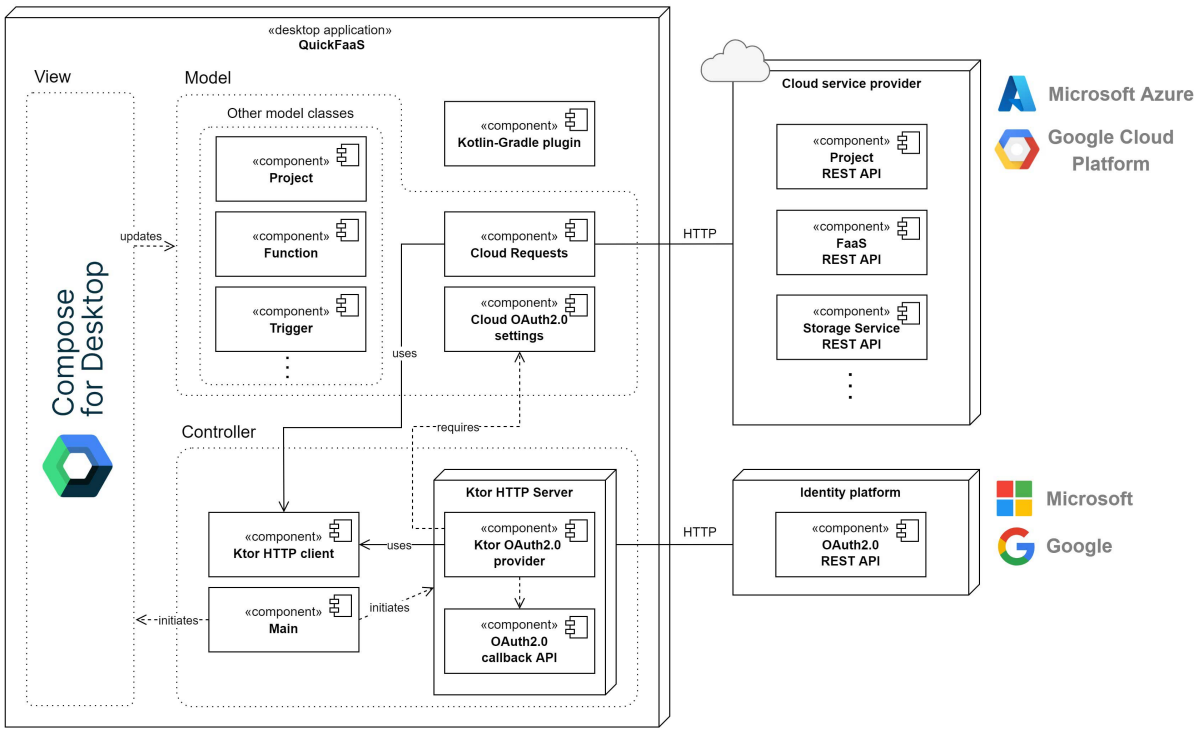



Figure 4.1: Deployment diagram

We adopted a design pattern that is commonly used in various types of GUI applications, called Model-View-Controller (MVC). An effective use of this pattern follows the separation of concerns (SoC) principle, meaning that the domain model, the presentation, and the controller are divided into three separate components. In QuickFaaS, the Controller manages the flow of the application mainly during the start-up process, where the HTTP server, as well as the View, are initialized. The Model manages the behaviour and data of the application domain, responds to requests for information about its state (usually from the View), and responds to instructions to change state. The View simply defines how the data should be displayed to the user.

### 4.1.2 Technologies

As regards to technologies, the Kotlin-Gradle plugin from JetBrains was chosen to compile the Kotlin code, targeted to JVM. Gradle also offers a highly-customizable resolution engine for dependencies specified in the *build.gradle* file. The majority of dependencies required by QuickFaaS are Ktor dependencies. This framework allowed us to instantiate the HTTP server on a certain port. It also includes an asynchronous HTTP client to make requests and handle responses. We extended the dependencies functionalities with the addition of a few extra plugins provided by Ktor, such as the authentication plugin to the server [40], and the JSON serialization to the client [39].

Finally, the View was developed using Compose for Desktop [16], a relatively new UI framework that provides a declarative and reactive approach to create desktop user interfaces with Kotlin. Even though the JDK 11 is the minimum required version to develop Compose for Desktop applications, JDK 15 or later must be used in order to do a native distribution packaging. This is due to the *jpackage* packaging tool only becoming available starting from JDK 14, and considered to be a permanent feature in JDK 16 [35].

Notice that no extra provider-specific CLI tools nor cloud orchestration frameworks are required by QuickFaaS for authentication or deployment purposes. QuickFaaS relies solely on the HTTP protocol to establish communications and exchange data with cloud providers. However, developers should still install the necessary runtime-related software for their functions, such as Maven together with JDK 11, to build Java projects, or *npm* to download Node.js modules, which is not yet supported.

## 4.2 Uniform programming model

The following subsections map out the structure of the application through class diagrams designed using UML. Class diagrams tend to model software applications that follow an object-oriented programming approach, just like QuickFaaS does.

We divided the full programming model into three smaller class diagrams. Each diagram is individually detailed in a dedicated subsection and has a corresponding ERM already described in Section 3.3. The full programming model, connecting all class diagrams together, can be found in Appendix A.2. The standard way of designing a class block is to have the class name at the top, attributes in the middle, and operations or methods at the bottom.

Classes named with the word *Cloud*, are actually interfaces in QuickFaaS that are mandatory to be implemented for a cloud provider to be considered as being supported. We instead represented these as common classes so that we could exemplify how a neutral cloud provider would implement them. For the time being, QuickFaaS will only support interoperability to MsAzure and Google Cloud Platform, meaning that for each *Cloud* class/interface, two implementation classes were developed and are missing from the programming model.

Some operations are marked with Kotlin's *suspend* modifier. These operations can only be executed from a *coroutine* or within another *suspend* function. A *coroutine* [36], is a feature of Kotlin that enables the developer to write asynchronous sequential code to manage potential long-running tasks in background threads, such as performing HTTP requests. QuickFaaS also takes advantage of *Data Classes* [37], which is a concept introduced by Kotlin whose main purpose is to simply hold data and make model classes cleaner and more readable.

### 4.2.1 Authentication mechanism

Both MsAzure and Google Cloud Platform share similar implementations of services based on the industry-standard protocol for authorization OAuth 2.0: Google Identity and Microsoft Identity Platform. QuickFaaS benefits from this common mechanism to authenticate its users in those providers, avoiding the need to require the installation of extra software for authentication purposes.

The programming classes that resulted from the ERM previously illustrated in Figure 3.2, are now represented in Figure 4.2. To better understand the usage of the OAuth 2.0 protocol, we designed a sequence diagram using UML, which can be found in Figure 4.3. Sequence diagrams are primarily used for representing the interactions between entities/objects in the sequential order that those interactions occur.

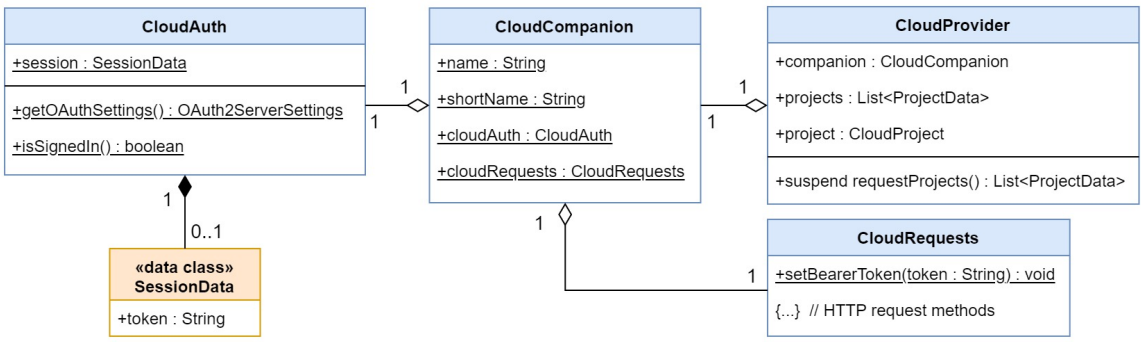



Figure 4.2: Authentication Mechanism class diagram

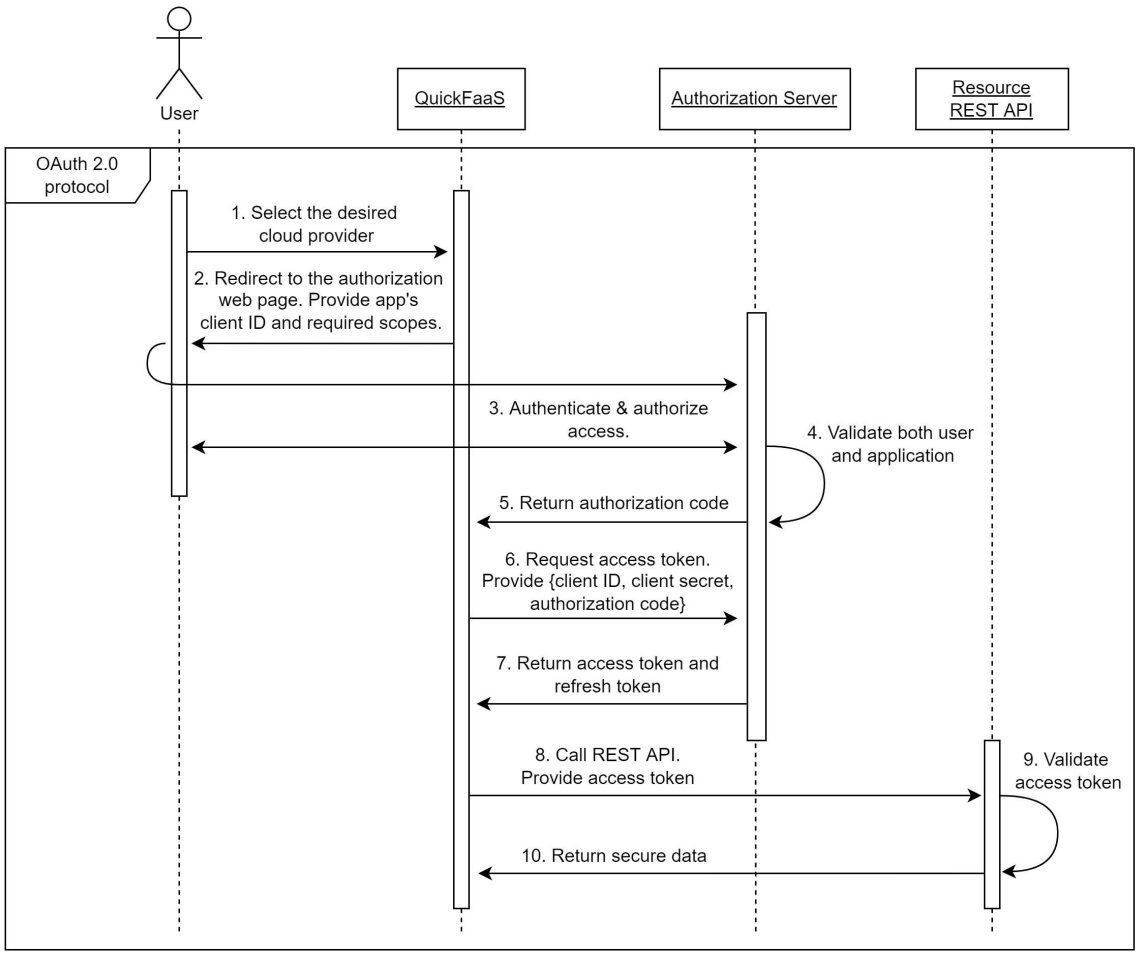



Figure 4.3: OAuth 2.0 sequence diagram

The authentication process starts once the user decides which cloud provider he wants to work with. After selecting one of the available cloud providers through QuickFaaS's UI, the user is redirected to the provider's authentication web page. In order to receive tokens sent by providers, QuickFaaS launches an HTTP server that starts listening for requests to the callback API locally, on a certain port.

A few OAuth 2.0 server settings need to be specified before being able to establish a connection between the application and the vendor's identity platform. The settings are defined in the *CloudAuth* class, using the *getOAuthSettings* operation. This operation returns an object of type *OAuth2ServerSettings*, which is a class provided by the Ktor authentication plugin. These settings include the authorization page URL, the access token request URL, the scopes, and the application's client ID and client secret. These last two are randomly generated during the application registration process, usually done in the provider's platform.

The operation *setBearerToken*, from the *CloudRequests* class, is invoked once the access token arrives to the callback API after the user authentication. *CloudRequests* classes define several other methods responsible for making HTTP requests to the respective cloud APIs. They all hold a reference to the same HTTP client instance defined in the Controller.

The *CloudCompanion* class holds static properties that are necessary for the authentication mechanism to work properly. Some of these properties are also used by the View to show basic information about the cloud provider, such as its name. This technique avoids the creation of unnecessary *CloudProvider* instances that would remain unused in heap space for long periods of time.

By the time the application supports more and more cloud providers, it's expected that most users won't work with the majority of them. For that reason, *CloudProvider* classes only get instantiated at the start of the function creation process, which is enabled once the user completes a successful authentication. This same instance will be automatically handled by the garbage collector whenever the user decides to exit the function creation process, usually after getting it deployed.

### 4.2.2 Function definition

Serverless functions need to follow provider-specific signatures in order to get triggered by the occurrence of events, so how can cloud providers handle the execution of cloud-agnostic functions? To overcome this constraint, we adopted the behavioural design pattern, identified by Gamma, called the *Template Method Pattern* [19, 20]:

"The *Template Method* lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure."

In our case, the template method/function is the entry point of the deployed FaaS, that is, the function that follows a provider-specific signature and gets triggered by a certain event. The cloud-agnostic function, developed by the QuickFaaS user, corresponds to the *hook* function, specified in the *Template Method Pattern*. The term "*hook*", is applied to functions that are invoked by template functions at specific points of the algorithm. Template functions are predefined in QuickFaaS, and don't require any modification by the user. Template functions, unlike *hook* functions, are specific to a cloud provider due to requiring the usage of custom function signatures and unique libraries.

QuickFaaS provides a built-in code editor for the development of *hook* (cloud-agnostic) functions, which can be visualized in Figure A.2.

Below, in Listings 4.1 and 4.2, we exemplify the usage of the *Template Method Pattern* for the provider GCP, by implementing both the template function and a *hook* function, respectively, using the Java programming language. The template function follows the provider-specific signature that enables it to be triggered by HTTP requests.

Listing 4.1: GCP template function – HTTP trigger

```
import ...
import quickfaas.triggers.http.HttpRequestQf;
import quickfaas.triggers.http.HttpResponseQf;

public class GcpHttpTemplate implements HttpFunction {
    @Override
    public void service(HttpRequest request, HttpResponse response) {
        HttpRequestQf reqQf = new GcpHttpRequest(request);
        HttpResponseQf resQf = new GcpHttpResponse(response);
        // Calls hook function
        new MyFunctionClass().myFunction(reqQf, resQf);
    }
}
```

Listing 4.2: Hook (cloud-agnostic) function – HTTP trigger

```
import quickfaas.triggers.http.HttpRequestQf;
import quickfaas.triggers.http.HttpResponseQf;

public class MyFunctionClass {
    public void myFunction(HttpRequestQf req, HttpResponseQf res) {
        res.send(200, "Hello world!");
    }
}
```

The developer should embrace the libraries provided by QuickFaaS when writing a fully cloud-agnostic function definition. In the above example, the HTTP event classes *HttpRequestQf* and *HttpResponseQf* are bundled into the *quickfaas-triggers.jar* file. Even though the libraries may look cloud-agnostic from the user's perspective, under the hood they are interacting with unique APIs from cloud providers to execute specific operations.

The following class diagram was based on the ERM previously illustrated in Figure 3.4.

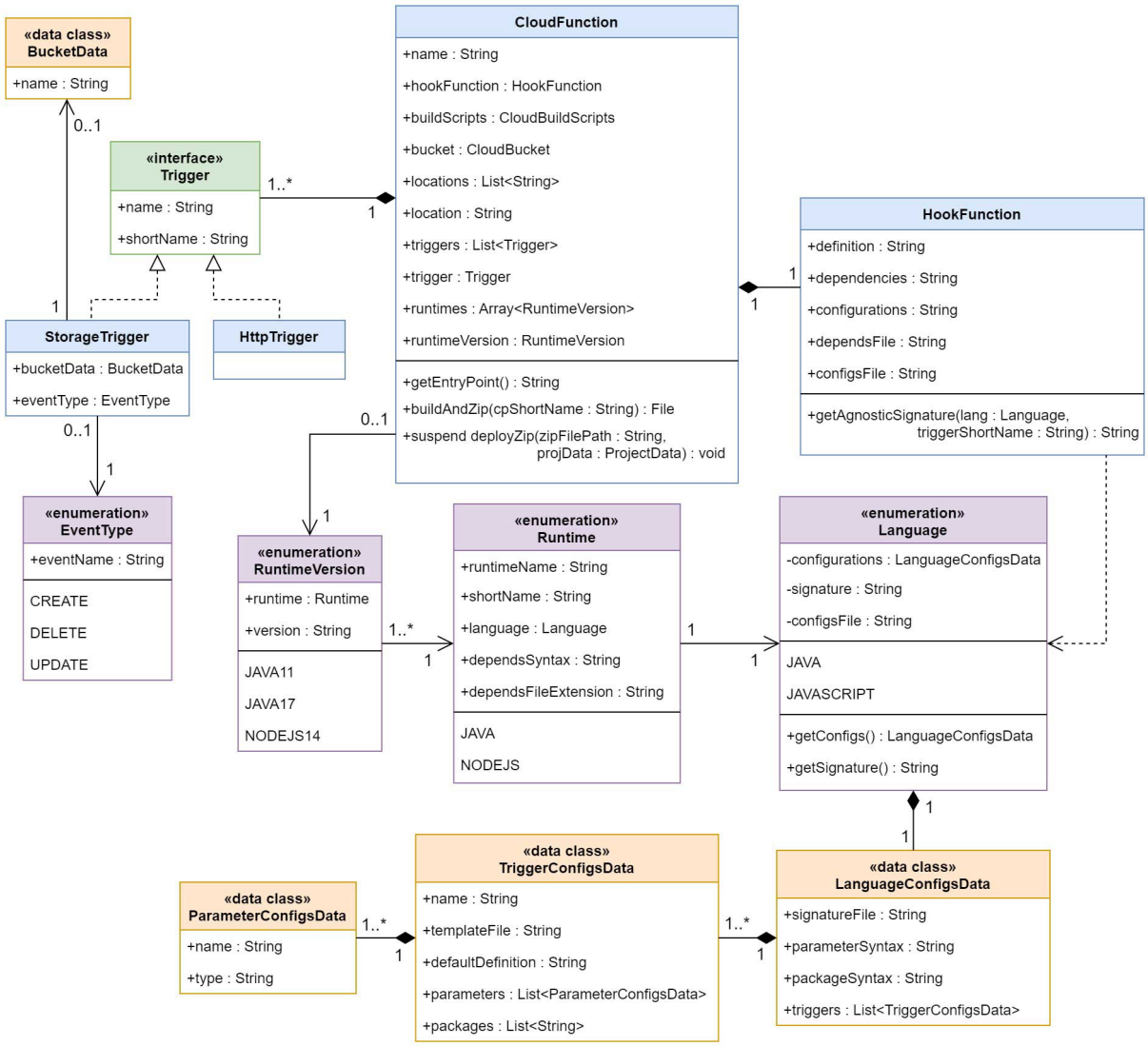



Figure 4.4: Function Definition class diagram

To start the cloud-agnostic function development, the user must first choose which runtime environment he wants to work with. For now, we only support function deployments for Java runtime, despite Node.js being defined as a constant in the *Runtime* enumeration class.

Once a runtime is selected, two file I/O operations are executed. The first one reads the respective language configurations file, written in JSON (*configsFile* from the *Language* enumeration class), using the *getConfigs* operation from the *Language* enumeration class. This operation populates an instance of the *LanguageConfigsData* class together with the rest of the subclasses using Ktor's serialization plugin. The second file I/O operation obtains the function signature skeleton by reading the language signature file (*signatureFile* from the *LanguageConfigsData* class), using the *getSignature* operation from the *Language* enumeration class. Signature files contain a definition similar

to the ones defined in Listings 3.3 and 3.4. Then, the *defaultDefinition*, the *packages* and the *parameters*, are determined according to the selected trigger. All of them are defined in a *TriggerConfigsData* class instance, that by this point is already populated. The *defaultDefinition* is usually only one or two lines of code.

Although an object of type *CloudBucket* is being referenced in the *CloudFunction* class, the *CloudBucket* class is missing from the illustrated model. This is due to the *bucket* entity not being a part of the function definition ERM, so the *CloudBucket* class is instead included in the FaaS deployment programming model, which is detailed in the next subsection.

The user can also define useful JSON configurations and extra dependencies to be downloaded right before deployment (*configurations* and *dependencies* properties from the *HookFunction* class, respectively). The configurations file allows users to specify JSON properties that can be accessed during function's execution time. In some cases, a few configurations are mandatory to be specified, otherwise, certain cloud-agnostic libraries won't work properly. For instance, a bucket access key needs to be provided in configurations when using the cloud-agnostic storage libraries for MsAzure. An example of these configurations can be found below in Listing 4.3, written in JSON.

Listing 4.3: Configurations file content example

```
{
  "resources": [
    {
      "id": "my-bucket-name",
      "type": "storage",
      "properties": {
        "accessKey": "my-private-key"
      }
    }
  ],
  "fruits": ["apple", "banana", "orange"]
}
```

The *accessKey* property is mandatory for accessing the respective bucket deployed in MsAzure, while the *fruits* array exemplifies a custom property that can be added by the user. Configurations that are required by certain libraries need to be marked as mandatory in QuickFaaS's documentation. The *fruits* array can be obtained using the *getConfiguration* static operation from the *ConfigurationsQf* class. This operation receives the property name as an argument and returns an instance of the respective JSON element, in this case, of type *JsonArray*.

QuickFaaS takes advantage of the Gson library from Google when doing these types of operations, which is a Java serialization/deserialization library to convert Java objects into JSON and back [34]. For the time being, the configurations file is always read during cold starts, regardless of whether the JSON properties are used or not. The impact of this operation on the function's performance is evaluated in Section 5.3.

Finally, to demonstrate a more complete use of the cloud-agnostic libraries, we provide the implementation of the *thumbnail generation* use case in Listing 4.4.

Listing 4.4: Use case 1 function definition – *thumbnail generation*

```
1  import ...
2  import quickfaas.resources.storage.BucketQf;
3  import quickfaas.resources.storage.StorageQf;
4  import quickfaas.triggers.storage.BlobQf;
5  import quickfaas.triggers.storage.BucketEventQf;
6
7  public class MyFunctionClass {
8      public void myFunction(BucketEventQf event, BlobQf blob) {
9          BucketQf bucket1 = StorageQf.newBucket(event.getBucketName());
10         byte[] source = bucket1.readBlob(blob.getName());
11         byte[] thumbnail = generateThumbnail(source, "jpeg");
12         BucketQf bucket2 = StorageQf.newBucket("bucket2thumbnails");
13         bucket2.createBlob("thumbnail-" + blob.getName(), thumbnail, "image/jpeg");
14     }
15     public byte[] generateThumbnail(byte[] source, String type) {...}
16 }
```

There are two cloud-agnostic operations that remote call the provider's storage service, these are the *readBlob* and the *createBlob* operations from the *BucketQf* class (lines 10 and 13, respectively). The first one reads all the bytes from the specified blob stored in the referenced bucket, while the second one creates a blob in the referenced bucket. The blob name, as well as the content of bytes, need to be provided as arguments to the blob creation operation. The *getBucketName* operation, from the *BucketEventQf* class (line 9), returns the bucket name where the storage event occurred. It's also worth mentioning that the *newBucket* static operation, from the *StorageQf* class (lines 9 and 12), returns a reference to an existing bucket, it doesn't create a new one.

The *generateThumbnail* operation definition was omitted due to being irrelevant in this context (line 15), but it simply consists in cutting the image width in half using common Java libraries.

### 4.2.3 FaaS deployment

FaaS deployments can be challenging when dealing with multiple cloud providers that require setting up different types of configurations and environments. Even when two providers offer an identical type of service configuration, it doesn't necessarily imply that the available values associated with that configuration are the same in both providers. For instance, not all cloud providers support the deployment of a service to the same locations nor support the same runtimes for executing serverless functions.

QuickFaaS tries to overcome these types of adversities by only enabling the set-up of fundamental configurations that are present in most cloud providers. Nevertheless, QuickFaaS allows the modification of some provider-specific configurations, but only those that are considered relevant or may affect the function's performance, e.g., the configuration to specify the memory allocated for each function is not available in every provider.

Similarly to the authentication process previously detailed, QuickFaaS benefits once again from a common mechanism available in the supported cloud providers, avoiding the need to install extra provider-specific tooling for deployment purposes. The deployment mechanism works by uploading a ZIP archive using provider-specific HTTP APIs. The following model, represented in Figure 4.5, shows what programming classes are involved during the function's source code build process and subsequent deployment of the ZIP archive. These are the resulting programming classes of the ERM previously illustrated in Figure 3.3.

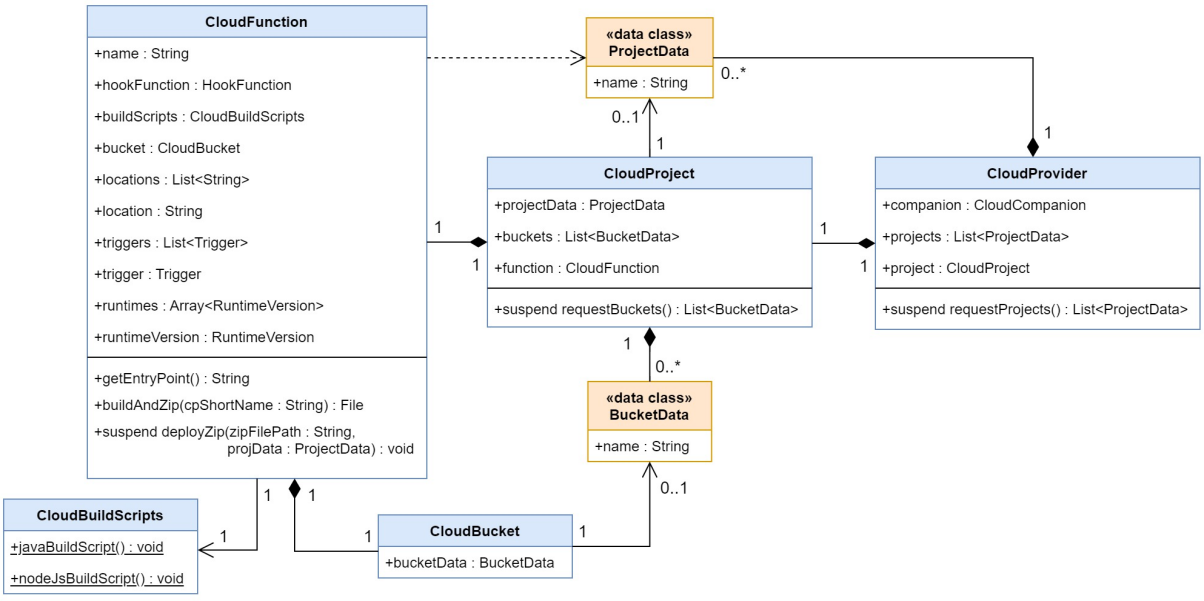



Figure 4.5: FaaS Deployment class diagram

QuickFaaS requires most deployment configurations to be established by the user before enabling the cloud-agnostic function development. The standard configurations include: (i) the project that manages the FaaS resource (*project* property from the *CloudProvider* class), (ii) the function name, (iii) the resource location, and (iv) the bucket used to store function related files, such as the function's source code, execution logs, etc., (*name*, *location* and *bucket* properties from the *CloudFunction* class, respectively).

However, as stated before, cloud providers can require extra and unique configurations to be established by the user. This happens with the azure subscription ID field, which is mandatory to be specified when deploying resources for the majority of services in MsAzure, including FaaS resources (*function apps*). Cloud-specific properties are declared in cloud-specific data classes. For instance, the subscription ID is declared in the *MsAzureProjectData* class, which extends the *ProjectData* class. This class is missing from the diagram due to declaring cloud-specific properties.

The ZIP deployment strategy requires the implementation of a dedicated deployment script for each of the supported cloud providers. Functions are packaged and deployed differently, e.g., AWS allows having multiple functions in one package, whereas MsAzure allows only one function per package. Described below, are the two main operations, from the *CloudFunction* class, responsible for the function's deployment to a FaaS platform:

1. ***buildAndZip***
   This operation starts by building the function's source code, if needed, into an executable file. The template function file, together with the created *hook* (cloud-agnostic) function file, make up the function's source code. We used Maven when building Java-based projects. As for JavaScript sources, no build tool would be necessary, only a package manager tool, such as *npm*, to download the required node modules.

   The executable JAR file, which results from the Maven build, is then packaged together with the downloaded dependencies. These packages should be organized using the provider's specific folder structure, which varies depending on the function's runtime [7, 29]. Runtime build scripts are defined per cloud provider, by implementing the *CloudBuildScripts* operations. This modular approach allows QuickFaaS to expand and integrate new environments efficiently. Since Node.js is not yet supported, only the *javaBuildScript* operation is implemented for both GCP and MsAzure cloud providers.

   The *buildAndZip* operation terminates once everything is zipped and ready to be deployed.

2. ***deployZip***

   The last operation of the process is the ZIP archive deployment to the FaaS platform. Cloud providers offer different APIs and services to enable FaaS deployments. For instance, MsAzure requires the deployment of a *function app* first [85], which is where the ZIP archive gets deployed afterwards. Additionally, MsAzure ZIP archives can contain multiple azure functions to be deployed to a single *function app* at once, while in GCP there can only be one function per FaaS deployment. Because *function apps* can only support one runtime at a time, QuickFaaS reuses existing *function apps*, that were configured with the same runtime, when deploying new azure functions, resulting in faster deployment times. The Kudu API was used to perform ZIP deployments to *function apps* [41, 42].

   As for GCP, we first used the Cloud Storage API to initiate a *resumable upload* of the ZIP archive to the selected storage bucket [62]. The *create* operation, from the Cloud Functions API, is then invoked to deploy the FaaS resource [18]. The ZIP sources are automatically loaded during deployment using the *sourceArchiveUrl* property, provided in the request body. This property specifies the exact location of the ZIP archive within the storage bucket.

As shown in Figure 4.6, the ZIP archive contains all the necessary artifacts to successfully launch the serverless function in the cloud. In this example, we illustrate the deployment process of a cloud-agnostic function written in Java to GCP, where Maven is used as the build tool.

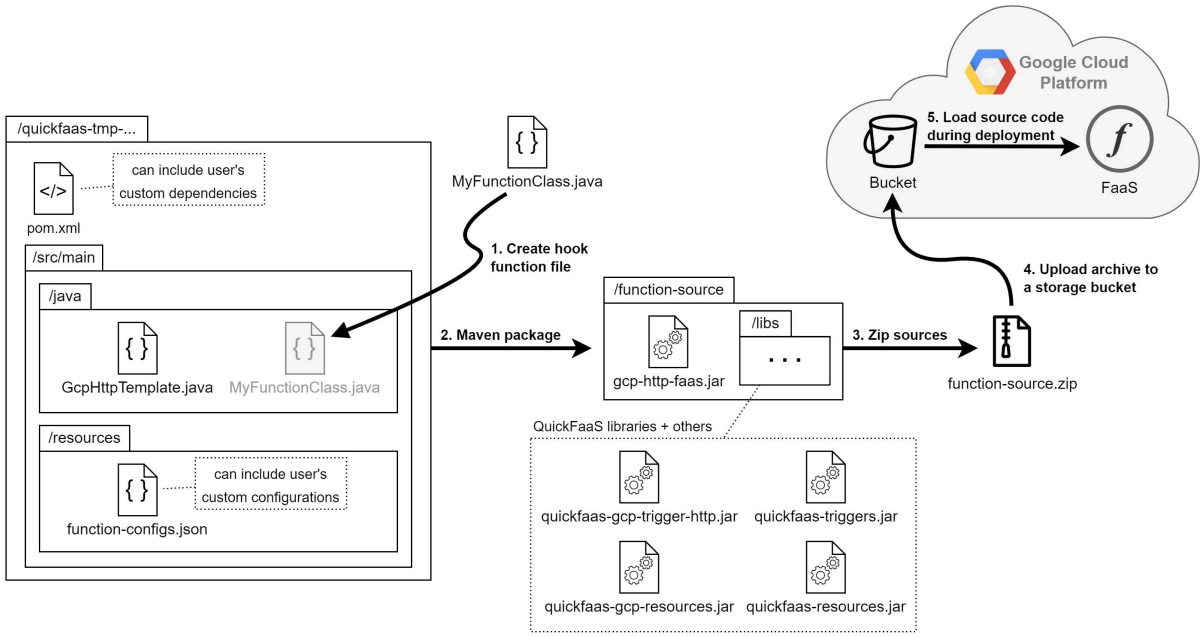



Figure 4.6: FaaS deployment to GCP for Java runtime

Below, we describe the purpose of each artifact that is bundled into the ZIP archive:

- ***pom.xml*** – includes information about the Java project and configuration details used by Maven to build the project, such as the build directory, source directory, dependencies, etc. The only modification allowed to the POM file is the addition of custom dependencies.

- ***MyFunctionClass.java*** – contains the cloud-agnostic function defined by the developer (*hook* function). This file is created during the *buildAndZip* operation.

- ***GcpHttpTemplate.java*** – contains the template function to be triggered by the occurrence of events. In the above example, the template function is triggered by HTTP requests. Template function files are predefined in QuickFaaS and use the following file naming syntax.

  [*cloudProvider*][*eventTrigger*]Template[.*languageFileExtension*]

- ***function-configs.json*** – contains user defined JSON properties to be accessed during function's execution time, using QuickFaaS's libraries.

- ***quickfaas-triggers.jar*** – establishes event trigger contracts between cloud-agnostic classes and provider-specific implementation classes.

- ***quickfaas-gcp-trigger-http.jar*** – implements cloud-agnostic HTTP trigger contracts using provider-specific event libraries. In this case, Google Cloud Platform event libraries are used for implementation.

- ***quickfaas-resources.jar*** – establishes contracts between cloud-agnostic classes and provider-specific implementation classes for interaction with common cloud resources and services.

- ***quickfaas-gcp-resources.jar*** – implements cloud-agnostic resource contracts using provider-specific libraries of services from Google Cloud Platform.

For GCP in particular, QuickFaaS requires developers to enable the Cloud Resource Manager API [12]. This will allow QuickFaaS to programmatically manage resource metadata.

## 4.3 Summary

This chapter covered the proposed solution to achieve interoperability and portability between FaaS platforms, including the system and software design decisions taken when developing QuickFaaS. We then went into detail to describe the mechanisms as well as the uniform programming model materialized by QuickFaaS.

For the next chapter, we will be presenting several metrics to evaluate and compare the execution performance of a use case implemented using both a cloud-agnostic and a cloud-non-agnostic approach.

# 5

# Evaluation

Measuring the performance of computer systems is a challenging task, especially when dealing with distributed systems managed by cloud providers. Different design aspects from FaaS infrastructures need to be taken into account when planning the evaluation experiments.

This chapter introduces different metrics that measure the impact of a cloud-agnostic approach on the function's performance, by comparing it to a cloud-non-agnostic one. To do this, we made several deployments and executions of the *search blobs* use case in MsAzure and Google Cloud Platform. The *search blobs* use case, represented in Figure 3.7, was written in Java, and it's the only one out of the three described in this work that gets triggered by HTTP requests, while at the same time being fully cloud-agnostic. Being triggered through HTTP, helped in the development of automated tests, whose purpose is to automatically generate and collect data for evaluation.

The automated tests were developed using the Kotlin language, together with JUnit 5 testing framework and Gradle build tool [38]. For each of the given tests, we describe the data collection methodology and do some analysis of the obtained results.

## 5.1 Metrics definition

Having established the appropriate use case for evaluation, we now had to decide what were the key metrics that could best characterize the performance of the function's deployment and execution.

The **execution time** is commonly recognized as the primary metric for measuring a function's performance. When cold started, the function's execution time includes an extra latency derived from the container's startup process, thus producing higher execution times than warm starts. The execution time was measured in milliseconds, while the second performance metric, the function's **memory usage**, was measured in megabytes (MB). The memory usage refers to the total amount of memory consumed during the function's execution.

When measuring for Google Cloud Platform, both metrics can be programmatically obtained using the *MetricService* from the Cloud Monitoring gRPC API [43]. Within this service, the operation *ListTimeSeries* can then be used to capture sets of metrics data that match certain filters, for a given time frame. The following filters need to be specified when capturing the execution time and the memory usage, respectively: *function/execution_times* and *function/user_memory_bytes* [31].

An extra cloud service is also required when capturing the function's execution time in MsAzure, called the Application Insights. Application Insights is a feature of Azure Monitor that provides extensible application performance management and monitoring for live applications, including *function apps*. We created as many Application Insights resources as *function apps* deployed. The *Query* operation, from the Application Insights REST API [3], can then be invoked to request a set of execution times (*FunctionExecutionTimeMs*) within a given time frame. This is done by sending the following log query as the body of the HTTP request:

*requests | project timestamp, customDimensions['FunctionExecutionTimeMs']*

Unfortunately, the measurement of memory usage per function execution isn't a metric currently available through Azure Monitor. There are, however, several other related metrics [8]:

- ***Working set*** – the current amount of memory used by the app (*function app*), in mebibytes (MiB).
- ***Private bytes*** – the current size, in bytes, of memory that the app process has allocated that can't be shared with other processes. Useful for detecting memory leaks.
- ***Function execution units*** – a combination of execution time and memory usage, measured in MB-milliseconds.

Both the *Working set* and the *Private bytes* consist of measuring the app's memory as a whole, they are not exclusive to serverless functions, making them inadequate metrics for this study. Additionally, any sort of interaction with the app via its REST APIs, or through the Azure Portal, can cause memory spikes, even when no functions were recently executed. As a consequence, the data collection process would have difficulties in distinguishing accurately the memory consumed during the function's execution time from the one spent in processing secondary app operations.

The *Function execution units* is the one out of the three specified metrics that best meets our needs, for different reasons: (i) gets measured for every function execution, (ii) doesn't include memory consumption from secondary app operations, and (iii) the calculation formula is defined in official documentation [25]. The *Function execution units* per function execution are calculated according to Equation 5.1.

$$FunctionExecutionUnits = execution\_time \times \boldsymbol{memory\_usage} \tag{5.1}$$

Even though this formula is being applied once every function execution by Azure Monitor, the units are presented as a time-based aggregation, meaning that they can't be obtained individually for a particular function execution. To be more specific, Azure Monitor aggregates units of function executions made within the same minute, so that it can be obtained as an average, minimum, maximum, sum, or count.

A set of *FunctionExecutionUnits* averages, for a particular time frame, can be requested using the *Metrics* operation from the Application Insights REST API [2]. The time frame and the *Average* aggregation are specified using the *timespan* and the *aggregation* query parameters, respectively. Then, using the formula specified above, the average memory consumption is determined by simply doing the average of the set of *FunctionExecutionUnits* averages, divided by the average of the respective set of execution times. Because cold starts evaluation only requires the measurement of one execution per FaaS resource, the two generated sets only include a single value each, so the division can be applied straight away. The *count* property was also useful to keep track of the number of executions each *FunctionExecutionUnits* average value corresponded to.

An article published in CODE Magazine, detailing various aspects regarding Azure Functions, also follows the same approach when measuring the function's memory usage [15]. In their case, the memory usage is relevant for the calculation and comparison of consumption costs between different azure pricing models.

Because the *FunctionExecutionTimeMs* is not recognized as a metric by Azure Monitor in the same way as *FunctionExecutionUnits*, the execution of a log query had to be used as an alternative to the *Metrics* operation.

## 5.2 Function execution environment

The specifications for the execution environments are presented below in Table 5.1. Apart from the location and the runtime, these are the default values recommended by providers when deploying a serverless function [30, 49].

| Specification | Google Cloud Platform | Microsoft Azure |
|---|---|---|
| Location | europe-west1 (Belgium) | west europe (Netherlands) |
| Runtime | Java 11 | Java 11 |
| Memory allocated | 256 MB | 1.5 GB |
| Min/Max instance count | 0 – 3000 | 0 – 200 |
| Operating system | Ubuntu 18.04 | Windows |

Table 5.1: Function execution environment specifications

The aim of the current study is to evaluate and compare the performance of different functions hosted within the same cloud provider. For that reason, we didn't make an effort in configuring machine specifications as similar as possible for the two providers.

## 5.3 Cold starts

Every time a function's triggering event occurs, a gateway component checks whether there's already a container instance in an idle (warm) state that could serve the recently arrived execution request. If there's no idle container, the gateway must allocate a new one and direct the execution request to the respective machine [17]. The process of starting up a new container, together with the preparation of the function's execution environment that serves the newly arrived request, is called a cold start.

Cold starts are one of the most critical performance challenges in FaaS applications due to its overwhelmingly expensive latency caused by the booting time, which can easily dominate the function's total execution time [65]. While languages such as JavaScript use an interpreter, Java requires a more complex environment to be set up in the container with JVM, leading to a higher latency most of the time.

The time it takes for an inactive container to be deallocated varies depending on the cloud provider. For instance, MsAzure offers three types of hosting plans for azure

functions that affect the frequency of cold starts: consumption plan, premium plan and dedicated (App Service) plan [46]. The consumption plan is the default hosting plan where clients only pay for compute resources when functions are running. As for the premium plan, *function apps* are running continuously, or nearly continuously, avoiding cold starts with perpetually warm instances. Finally, the dedicated plan is best for long-running scenarios and billing is made regardless of how many *function apps* are running in the plan. This last one differs from the other two hosting plans, that have consumption-based cost components. The hosting plan that best suits our needs is the consumption plan, given that the *search blobs* use case doesn't require heavy operations for execution, and we also don't want to avoid cold starts for the purposes of this evaluation like the premium plan does. When using the consumption plan, container resources are deallocated after roughly 20 minutes of inactivity, meaning that the next invocation will result in a cold start [44].

Cold starts can also be avoided in Google Cloud Platform by setting a minimum number of instances greater than zero [32]. We kept the minimum number of instances at zero during the evaluation. Even though no official documentation specifies for how long a container stays in an idle state (*idle timeout*) before being offloaded, some studies have argued that container instances are recycled after 15 minutes of inactivity [28].

This evaluation will enable us to verify whether the usage of cloud-agnostic libraries, in addition to the execution of a few extra operations in template functions, have an evident impact on the function's performance when cold started.

### 5.3.1 Measurement methodology

The established measurement methodology consists in generating and collecting 300 cold start execution records of both the cloud-agnostic and the cloud-non-agnostic functions, in each cloud provider. This methodology requires a waiting time of at least 15 to 20 minutes after the latest execution to get a single data record. This interval allows the container to have enough time to transition from an idle to a stopped (cold) state. Which means that if we were to trigger a FaaS resource to obtain a single cold start execution record at a time, each function test would take around 100 hours to retrieve the 300 data records ($300 \times 20$ *minutes*).

To overcome this issue, we first deployed 100 FaaS resources in each cloud provider, that is, 100 cloud functions in GCP and 100 *function apps* in MsAzure. We then used three batches of 100 invocations to trigger the 100 deployed functions one at a time in each cloud provider. The invocation batches were separated by an interval of about an hour to ensure that cold starts would occur.

In MsAzure, cold starts happen per *function app*, meaning that once an azure function is cold started, any other function within the same app can be warm started if executed shortly after the cold start. Therefore, the fastest way of collecting 100 cold start execution records is by cold starting 100 different *function app* instances, each one containing a single azure function. Due to some unanticipated problems in cold starting each *function app*, to be discussed in Section 5.6, we ended up only doing 50 executions per batch in MsAzure for 50 *function apps*. This issue caused the process of data collection in MsAzure to take the double the amount of time of GCP. Nevertheless, we were still able to retrieve the 300 records for each function.

### 5.3.2 Measurement analysis

Given the nature of cold starts, not all 300 records per function definition were considered when determining the average execution time. Each function bar, illustrated in Figure 5.1, only takes into account the best 75% of the total measured records, in other words, the 225 lowest execution times, to perform the average calculation. This reduces the probability of presenting misleading results, known as outliers, that occur frequently in cold starts due to latency issues.

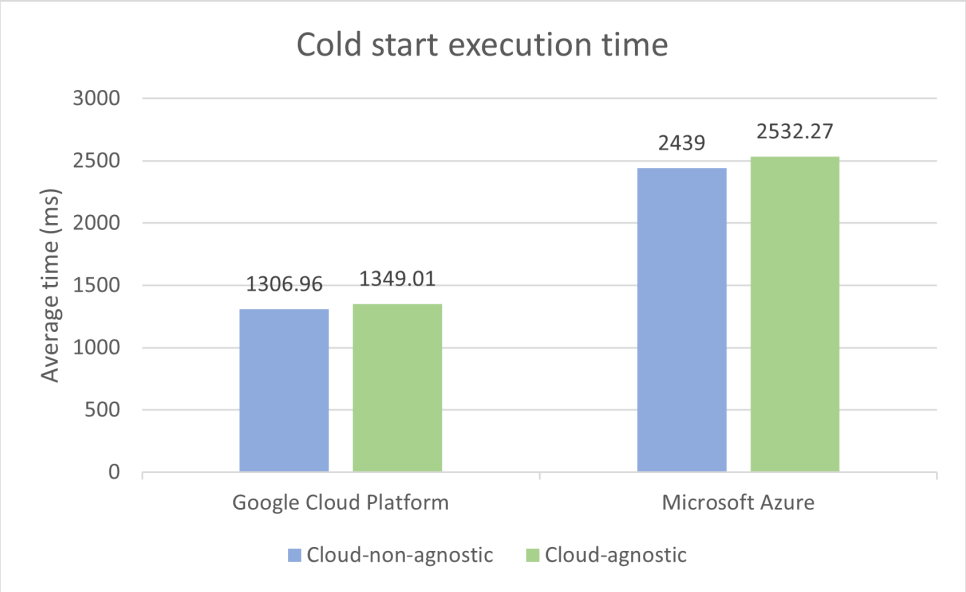



Figure 5.1: Cold start execution time

From the results illustrated above, we can conclude that the cloud-agnostic function definition had an increase in execution time of 3.2% in Google Cloud Platform and 3.8% in Microsoft Azure, when compared to the cloud-non-agnostic one. The slight increase in execution times was inevitable for cold starts, given that template functions start by

reading the configurations JSON file, ending up causing some overhead due to being a file I/O operation. The configurations file allows users to specify JSON properties that can be accessed during function's execution time. The only configuration specified by the *search blobs* use case is the bucket's access key when deployed to MsAzure, which is a requirement for accessing the storage service. The same access key is *hardcoded* in the respective cloud-non-agnostic definition so that we can have a clear perception of the impact on execution time when performing the file I/O operation.

For the time being, the configurations file is always read in cold starts, regardless of whether the JSON properties are used or not. No extra authentication is required to cloud functions deployed to GCP in order to interact with other services, meaning that the execution time could be improved by not reading the configurations file, since no storage access key is needed.

We can also point out that around 17% of the total number of cold start execution times measured in MsAzure (cloud-agnostic and cloud-non-agnostic), with no exclusions, were above three seconds. While in Google Cloud Platform, no record reached the two-second mark. This can be attributed to different hardware, but also to the underlying operating system and virtualization technology. Perhaps Ubuntu containers, from GCP, have faster start-up times on average when compared to Windows containers from MsAzure [86].

As for the memory usage, represented in Figure 5.2, no records were ignored when determining the average memory consumption.

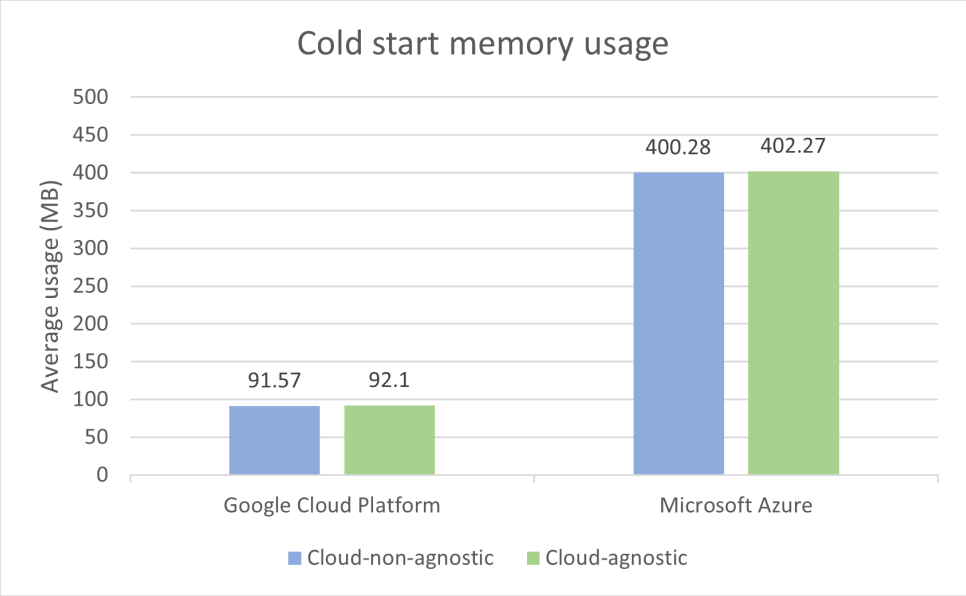



Figure 5.2: Cold start memory usage

The difference in memory usage between the two function definitions is practically non-existent in both providers. This difference was expected to be close to zero, since there are no extra heavy allocated objects or operations executed by cloud-agnostic libraries that could cause the memory usage to increase drastically.

## 5.4 Warm starts

The time wasted in cold booting new containers is considered to be the main drawback of serverless computing. Warm starts have an important role in keeping execution times balanced and as close as possible to *serverful* services. Without them, the computational costs of processing a batch of several consecutive execution requests would be much higher, considering that it would require the start-up of a new container for every newly arrived request.

A warm start happens whenever an existing execution environment (container) is reused in subsequent executions of the same function. A container is considered to be in an idle (warm) state after having recently served one or more execution requests prior to the current one. To reduce costs, FaaS containers are offloaded after remaining idle for a certain period of time. The lifetime of a warm instance varies depending on the cloud provider.

Some developers decide to force their FaaS containers to stay warm for long periods of time by warm starting them frequently. Their goal is to prevent cold starts from happening, and it's often recommended for FaaS applications that experience a substantial amount of latency in execution times when cold booted (>10 seconds). Keeping containers alive can, however, consume valuable computational resources from servers, and will most likely increase costs [23]. Additionally, cold starts can still occur when the function is auto-scaling to handle capacity, causing a new instance to be created.

By measuring multiple consecutive warm starts, we will be able to evaluate whether QuickFaaS's cloud-agnostic libraries produce any kinds of memory leaks as the executions go by. A memory leak is a scenario where objects present in the heap space are no longer needed, but the garbage collector is unable to remove them from memory because they're still being referenced, therefore, they're unnecessarily maintained. As for execution times, it's expected the difference between the two function definitions to be close to zero milliseconds.

### 5.4.1 Measurement methodology

The established measurement methodology consists in generating and collecting 200 warm start execution records of both the cloud-agnostic and the cloud-non-agnostic functions, in each cloud provider. The 200 records are a combination of two batches of 100 execution records, each batch being measured at separate times. The very first function execution of each batch originates a cold start, and for that reason, it was not considered as one of the data records.

To be able to retrieve the 100 records for each batch, a randomly selected FaaS resource from the previous test was triggered around 400 consecutive times ($4 \times 100$ *records*), with a delay of 10 seconds between invocations. The reason for the high number of executions, as well as the delay between them, is explained in Section 5.6. As a result, the warm starts test was the one that took the longest amount of time to be completed.

### 5.4.2 Measurement analysis

We experienced considerably fewer outliers while measuring execution times from warm starts. Warm starts are less likely to have high latency issues when compared to cold starts, since they don't have to deal with cold booting operations as often. Therefore, the percentage of best records to be taken into consideration when calculating the average was increased from 75% to 85%. Because we're triggering the same function multiple times, cold boots can still occur due to auto-scaling, which causes a new instance to be created. The warm starts test results can be found in Figure 5.3.

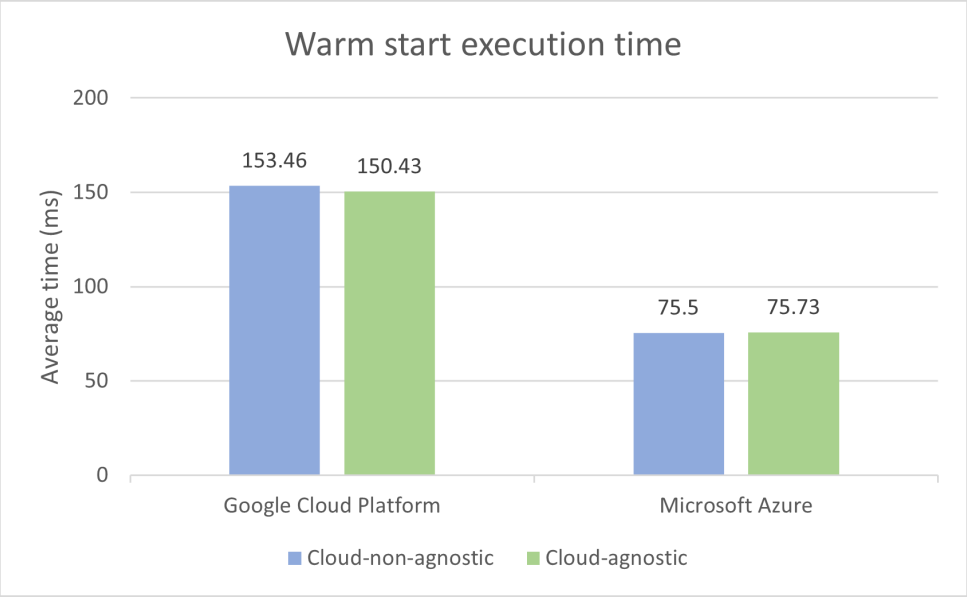



Figure 5.3: Warm start execution time

The test results show minor differences in execution times, despite the greater abstraction provided by cloud-agnostic libraries, which also simplifies code complexity. Even though there's no guarantee that the state of serverless functions is preserved between consecutive invocations, the execution environment can be often recycled during warm starts. QuickFaaS libraries take advantage of this characteristic by caching certain objects that may be expensive to recreate on each function invocation [79]. For instance, whenever a cold start happens, the JSON text of the configurations file is cached in a global variable, avoiding the need to repeat the file I/O operation in subsequent warm starts. By adopting this strategy, we are able to close the gap in execution times between the two function definitions, which was much higher in cold starts.

It's also worth mentioning the difference in execution times between cloud providers. GCP functions took approximately the double the amount of time of MsAzure functions to finish execution. One of the factors that may have contributed to this time difference is the amount of memory allocated to MsAzure containers, which is much higher than in GCP. Low memory capacity can increase the function's execution time.

As shown in Figure 5.4, there was also a considerable difference in memory usage. No records were excluded this time when determining the average memory consumption.

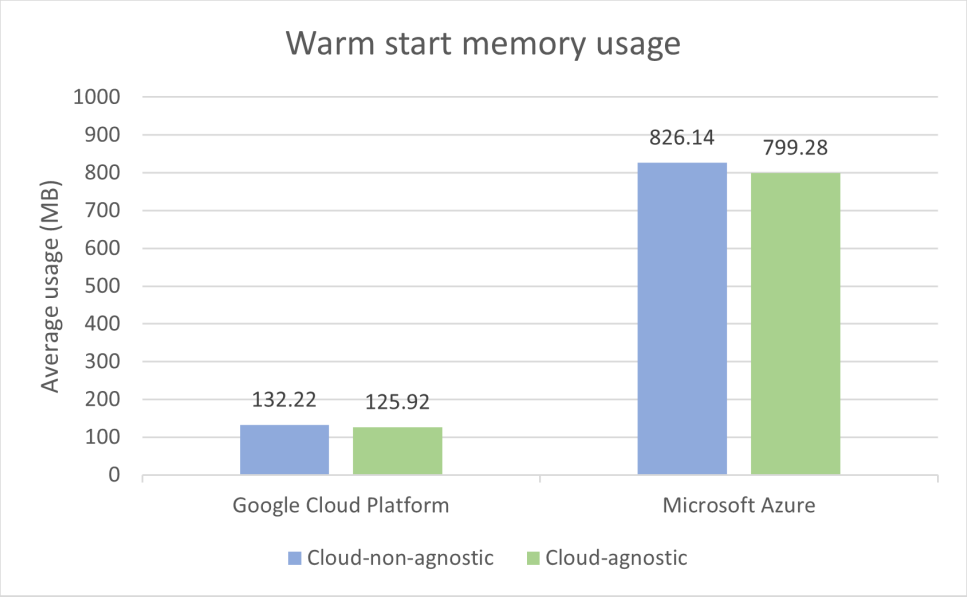



Figure 5.4: Warm start memory usage

The measurements reveal that cloud-agnostic functions made better use of the available memory when compared to cloud-non-agnostic ones. We were unable to find a reasonable answer to justify these differences in memory consumption. Despite the

difference, both functions from GCP reached a maximum of 145 MB in memory consumption during testing. The same analysis can't be made for azure functions, given that *Function execution units* are presented as a time-based aggregation and not individually for a particular execution.

Despite the noticeable increase in memory usage when compared to cold starts, there were no signs of memory leaks. This increase is a natural consequence of warm starting the same container instance several times.

## 5.5 ZIP deployments

To conclude the evaluation, we will do a comparison between ZIP deployment times to check whether the usage of QuickFaaS's libraries have a negative impact in this regard. The collected records derive from the deployments made during the previous tests. The evaluation consists in comparing ZIP sizes and deployment times of the *search blobs* use case in both cloud providers.

### 5.5.1 Measurement methodology

The established measurement methodology consists in collecting data from 100 FaaS resource deployments of both the cloud-agnostic and the cloud-non-agnostic functions, in each cloud provider. The same deployment scripts developed for QuickFaaS were reused by the automated tests. The deployment time was determined based on the formula specified in Equation 5.2.

$$DeploymentTime = zip\_upload\_time + resource\_deployment\_time \tag{5.2}$$

The *zip_upload_time* corresponds to the HTTP request duration that is responsible for the upload of the ZIP archive to the cloud provider. While the *resource_deployment_time*, indicates the time it took for the FaaS resource to be available for access after being requested to be deployed.

In Google Cloud Platform, a cloud function is ready to be accessed once the resource's *updateTime* attribute is defined with the deployment timestamp [27]. As for MsAzure, a *function app* is considered to be successfully deployed once a record with the *Create* value attached to the *changeType* attribute appears in the activity logs [48]. By accessing this record, we can then retrieve the deployment timestamp.

Each timestamp is then used to calculate the *resource_deployment_time*, by subtracting the respective HTTP deployment request timestamp, stored previously.

### 5.5.2 Measurement analysis

The 100 deployments of each function were made in a sequential order, with an average upload speed of 21 megabits per second (mbps). For the next chart, two distinct data types are included in each function bar. The first one being the average ZIP deployment time in seconds (s), at the top of the function bar, and the second one being the ZIP archive size in kilobytes (KB), at the middle of the function bar.

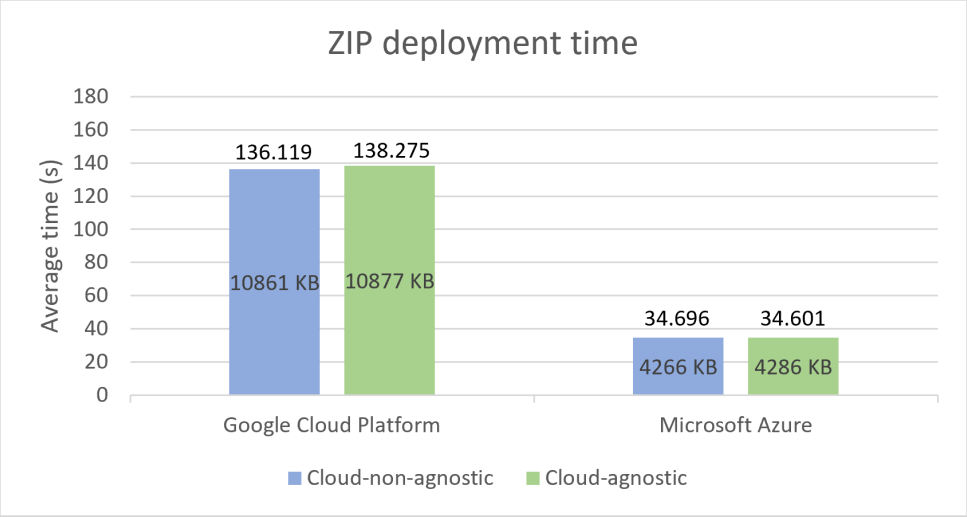



Figure 5.5: ZIP deployment time

As shown in Figure 5.5, cloud-agnostic libraries only add a few extra kilobytes of space to ZIP archives, 16 KB for GCP and 20 KB for MsAzure to be more precise. As expected, the size difference had almost no impact on the function's deployment time.

Nonetheless, as the libraries become more and more complete over time, ZIP archives will become larger in size, resulting in higher deployment times. By that time, Quick-FaaS should be capable of minimizing the number of its own dependencies as much as possible, not only to lower deployment times, but most importantly, to optimize cold starts.

## 5.6 Adversities

Described below, are the main adversities we came across while collecting performance metrics data, together with an explanation on how we managed to overcome them.

They are sorted based on the time period in which they occurred:

1. ***Function apps unavailable.***

   The fastest way of collecting 100 cold start execution records in MsAzure is by cold starting 100 different *function app* instances. However, while triggering each azure function, we noticed that the last 30 or so functions were responding with a 503 HTTP status code, indicating that the *function app* service was unavailable. The *function apps* were redeployed a few times, but the same problem persisted. We also verified that the consumption plan allowed a maximum number of 100 *function apps* [49], meaning that the limit wasn't being surpassed. Unfortunately, we were unable to determine the exact reason behind this problem.

   To overcome this issue, we decided to only cold start the first 50 azure functions twice, causing the process of data collection in MsAzure to take the double the amount of time of GCP.

2. ***Unregistered executions.***

   The warm starts evaluation included a total of 800 warm start execution records, meaning that we had to do at least 200 consecutive invocations for each function definition. It turns out that not all warm start executions are registered by Google Cloud Platform metrics. For instance, out of 100 consecutive function invocations, less than 10 got registered by the metrics service.

   We tried to mitigate this issue by adding a ten-second delay between consecutive function invocations. The purpose of this delay is to give some extra time to the metrics service to register execution metrics. With this strategy, we were able to reduce the number of necessary invocations from $10\times$ to $4\times$ the number of desired records. Therefore, in order to generate the 200 execution records, each function had to be warm started around 800 times ($4 \times 200$ *records*). If no delay was applied while generating the 200 execution records, each function would have to be warm started around 2000 times ($10 \times 200$ *records*).

   On the other hand, the Application Insights, from MsAzure, is capable of registering every function execution time. However, a zero-second delay for warm starts could no longer be used as well. Otherwise, most execution times with a zero-second delay would be much lower than the ones registered using a ten-second delay, which would lead to unfair comparisons between the two providers. To verify this analysis, we made an extra test in MsAzure using a zero-second delay between warm start invocations of the cloud-agnostic function. We came to the

conclusion that most execution times ranged from 20 to 40 ms, while with a ten-second delay, as shown in Figure 5.3, the average execution was around 75 ms, with most execution times ranging from 60 to 80 ms.

3. ***Measurements inconsistency.***

   Latency can't be ignored when evaluating the performance of FaaS applications. The highest latency is usually experienced in cold starts, during the preparation of the execution environment. For the *search blobs* use case in particular, the remote calls to the storage service also contribute with some network latency.

   All things considered, execution times can therefore be volatile, causing the results to be inconsistent when measured at separate times. To increase the reliability of measurements, various tests had to be repeated multiple times on different days during off-peak hours (i.e., between 2 p.m. and 6 p.m.), in an effort to avoid network congestion.

## 5.7 Summary

This chapter described different metrics to measure the impact of a cloud-agnostic approach on the function's performance, by comparing it to a cloud-non-agnostic one. For every test that was developed, we specified the purpose of its metrics, followed by a detailed explanation on how it was performed, and lastly, an analysis of the obtained results was provided. We also described the main adversities that were faced during the data collection.

The sixth and final chapter summarizes the key points of this work and explains how do these contribute to solving the problems addressed before. We also point out some drawbacks to the proposed solution. Finally, the future work is presented to briefly discuss in what ways we believe QuickFaaS can be improved.

# 6

# Conclusions

This chapter concludes this work by mentioning its main achievements and explaining why are these significant to solve or minimize the problems addressed. We also point out some drawbacks to the proposed solution. The future work sums up this chapter by briefly discussing in what aspects we believe QuickFaaS can be improved, along with our expectations for the future regarding cloud computing.

## 6.1 Achievements

The overall goal of this work was to characterize a solution to provide portability and interoperability between FaaS platforms. We started by designing a few ERMs to give an overview of the cloud-agnostic approach to the problem. A uniform programming model was then established as a result of the ERMs. The programming model was later materialized into a multi-cloud interoperability desktop tool, which we called QuickFaaS, developed using Kotlin.

QuickFaaS targets the development of cloud-agnostic functions as well as FaaS deployments to multiple cloud environments, without requiring the installation of extra provider-specific software. The proposed cloud-agnostic approach enables developers to reuse their serverless functions in multiple cloud providers, with the convenience of not having to change a single line of code. The solution aims to minimize vendor lock-in issues in FaaS platforms, and will, therefore, encourage developers and organizations to target different cloud providers in exchange for a functional benefit.

We also provided an evaluation to validate the proposed solution by measuring the impact of a cloud-agnostic approach on the function's performance, when compared to a cloud-non-agnostic one. The study has shown that the cloud-agnostic approach doesn't have a significant impact neither in the function's execution time nor in memory usage.

The main contributions of this work were made publicly available on a GitHub repository [61]. In terms of code development, the repository includes the uniform programming model for authentication and FaaS deployments, and finally, the cloud-agnostic libraries and respective documentation in the *wiki* page. The data supporting the reported results for evaluation is also included in the form of Excel spreadsheets, together with the implementation of the *search blobs* use case using a cloud-agnostic and a cloud-non-agnostic approach. Being an open-source project will allow us to receive feedback or even accept new contributions from the community.

## 6.2 Drawbacks

Throughout the course of the project, we were able to identify a number of drawbacks to the proposed cloud-agnostic solution provided by QuickFaaS. These can be found below:

- ***Unauthorized access to OAuth 2.0 client secrets.***

  QuickFaaS tries to obfuscate OAuth 2.0 client secrets through the usage of cryptographic keys stored in a protected format to prevent unauthorized access. However, trying to obfuscate a secret in installed applications can be seen as a futile effort, since it can always be recovered using the abundance of reverse engineering and debugging tools.

  More details regarding OAuth 2.0 threat model and security considerations are described in RFC 6819 [64].

- ***OAuth 2.0 not ideal for automation.***

  Despite being a popular industry-standard authorization protocol, OAuth 2.0 comes with the trade-off of being hard to integrate with automation scripts. Continuous Integration (CI) pipelines are often adopted by organizations when delivering cloud-native apps to clients. The automation using QuickFaaS's deployment scripts is possible, but the authentication process is not ideal. The workaround would probably require the usage of provider-specific mechanisms, which would go against the idea of providing a uniform authentication model.

- ***Unsupported operations by cloud-agnostic libraries.***

  Serverless functions have access to a wide variety of libraries offered by cloud providers. QuickFaaS's cloud-agnostic libraries only provide a uniform access to operations that are considered relevant or that are commonly found in most provider-specific libraries for a particular resource or event trigger. Meaning that operations that are unique to a specific provider will probably be left out of QuickFaaS's libraries for the sake of providing a full cloud-agnostic usage.

  Additionally, if for some reason a certain cloud-agnostic operation can't be implemented for a particular provider, the documentation should show a warning specifying that the operation is incompatible with that cloud provider.

- ***Cloud-agnostic functions can be even harder to debug.***

  Because QuickFaaS's cloud-agnostic libraries are exposing an abstract layer over provider-specific APIs, errors can become even harder to understand when they originate from provider-specific operations. Cloud-agnostic libraries try to prevent this from happening by handling exceptions and informing users of what went wrong during execution.

  There's also no feature to check whether cloud-agnostic functions will run as expected before being deployed. Azure Functions Core Tools, for instance, offers this feature.

## 6.3 Future work

As part of the future work, we plan on publishing a scientific journal article to continue to raise awareness concerning the issues identified with FaaS platforms and to explain how QuickFaaS attempts to solve them.

Further development of QuickFaaS will consist on supporting more cloud providers, runtimes, and adding extra features. At the moment, Amazon Web Services (AWS) is the cloud provider with the highest priority. As for new function runtimes, Java is the only one supported for now, we intend to support Node.js next. New possible features include enabling the automation of FaaS deployments through the command-line and providing users a way to test their cloud-agnostic functions before being deployed.

We strongly believe that this work will inspire other developers to create their own solutions that could somehow improve the portability of cloud applications for any kind of service. Contributions resulting from new projects will be fundamental to help us take major steps towards the mitigation of vendor lock-in in cloud computing.

# References


[1] "Alien4Cloud – TOSCA". Last accessed 17/07/2022, [Online]. Available: https://alien4cloud.github.io/#/documentation/2.0.0/concepts/tosca.html.

[2] "Metrics – Application Insights REST API". Last accessed 03/07/2022, [Online]. Available: https://docs.microsoft.com/en-us/rest/api/application-insights/metrics/get.

[3] "Query – Application Insights REST API". Last accessed 30/06/2022, [Online]. Available: https://docs.microsoft.com/en-us/rest/api/application-insights/query/execute.

[4] Mariano Ezequiel Mirabelli, Pedro García-López, and Gil Vernik, "Bringing Scaling Transparency to Proteomics Applications with Serverless Computing", in *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, 2020, pages 55–60. DOI: 10.1145/3429880.3430101.

[5] Samuel Ginzburg and Michael J. Freedman, "Serverless Isn't Server-Less: Measuring and Exploiting Resource Variability on Cloud FaaS Platforms", in *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, 2020, pages 43–48. DOI: 10.1145/3429880.3430099.

[6] "AWS CloudFormation". Last accessed 17/07/2022, [Online]. Available: https://aws.amazon.com/cloudformation/.

[7] "Folder structure of an Azure Functions Java project". Last accessed 05/01/2022, [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/functions-reference-java?tabs=bash%2Cconsumption#folder-structure.

[8] "Supported metrics with Azure Monitor". Last accessed 01/07/2022, [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-monitor/essentials/metrics-supported#microsoftwebsites.

[9] "Language support details". Last accessed 28/12/2021, [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/functions-create-function-app-portal#language-support-details.

[10] "Cloudify – TOSCA". Last accessed 17/07/2022, [Online]. Available: https://cloudify.co/tosca/.

[11] A. Parameswaran and Asheesh Chaddha, "Cloud Interoperability and Standardization", *SETLabs Briefings*, vol. 7, 2009.

[12] "Cloud Resource Manager API". Last accessed 26/08/2022, [Online]. Available: https://cloud.google.com/resource-manager/reference/rest.

[13] "Serverless MEAN stack". Last accessed 01/08/2022, [Online]. Available: https://cloud.google.com/blog/topics/developers-practitioners/serverless-with-cloud-run-mongodb-atlas.

[14] "Cloud Translation API". Last accessed 20/07/2022, [Online]. Available: https://cloud.google.com/java/docs/reference/google-cloud-translate/latest/com.google.cloud.translate.

[15] "Digging into Azure Functions: It's Time to Take Them Seriously". Last accessed 02/07/2022, [Online]. Available: https://www.codemag.com/article/1711071/Digging-into-Azure-Functions-It%E2%80%99s-Time-to-Take-Them-Seriously.

[16] "Compose for Desktop". Last accessed 19/07/2022, [Online]. Available: https://www.jetbrains.com/lp/compose-desktop/.

[17] David Bermbach, Ahmet-Serdar Karakaya, and Simon Buchholz, "Using Application Knowledge to Reduce Cold Starts in FaaS Services", in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, 2020, 134–143. DOI: 10.1145/3341105.3373909.

[18] "Create cloud function". Last accessed 28/07/2022, [Online]. Available: https://cloud.google.com/functions/docs/reference/rest/v1/projects.locations.functions/create.

[19] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[20] Dirk Riehle, "Design Pattern Density Defined", *SIGPLAN Notices*, vol. 44, no. 10, pages 469–480, 2009, ISSN: 0362-1340. DOI: 10.1145/1639949.1640125.

[21] Lina Lan, Fei Li, Bai Wang, Lei Zhang, and Ruisheng Shi, "An Event-Driven Service-Oriented Architecture for the Internet of Things", in *Proceedings of the IEEE Asia-Pacific Conference on Services Computing (APSCC)*, 2014, pages 68–73. DOI: 10.1109/APSCC.2014.34.

[22] Erwin van Eyk, Johannes Grohmann, Simon Eismann, André Bauer, Laurens Versluis, Lucian Toader, Norbert Schmitt, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup, "The SPEC-RG Reference Architecture for FaaS: From Microservices and Containers to Serverless Platforms", *IEEE Internet Computing*, vol. 23, no. 6, pages 7–18, 2019. DOI: 10.1109/MIC.2019.2952061.

[23] Alexander Fuerst and Prateek Sharma, "FaasCache: keeping serverless computing alive with greedy-dual caching", in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pages 386–400. DOI: 10.1145/3445814.3446757.

[24] Vladimir Yussupov, Uwe Breitenbücher, Frank Leymann, and Christian Müller, "Facing the Unplanned Migration of Serverless Applications: A Study on Portability Problems, Solutions, and Dead Ends", in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, 2019, pages 273–283. DOI: 10.1145/3344341.3368813.

[25] "Consumption plan costs". Last accessed 01/07/2022, [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/functions-consumption-costs?tabs=portal#consumption-plan-costs.

[26] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J. Yadwadkar, Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica, and David A. Patterson, "What Serverless Computing is and Should Become: The next Phase of Cloud Computing", *Communications of the ACM*, vol. 64, no. 5, pages 76–84, 2021, ISSN: 0001-0782. DOI: 10.1145/3406011.

[27] "Cloud Function resource". Last accessed 12/07/2022, [Online]. Available: https://cloud.google.com/functions/docs/reference/rest/v1/projects.locations.functions#CloudFunction.

[28] "Cold Starts in Google Cloud Functions". Last accessed 11/06/2022, [Online]. Available: https://mikhail.io/serverless/coldstarts/gcp/.

[29] "Structuring source code for Java". Last accessed 05/01/2022, [Online]. Available: https://cloud.google.com/functions/docs/writing#structuring_source_code.

[30] "Cloud Functions – Memory limits". Last accessed 08/07/2022, [Online]. Available: https://cloud.google.com/functions/docs/configuring/memory.

[31] "Cloud function metrics". Last accessed 20/06/2022, [Online]. Available: https://cloud.google.com/monitoring/api/metrics_gcp#gcp-cloudfunctions.

[32] "Cloud Functions – Using minimum instances". Last accessed 04/08/2022, [Online]. Available: https://cloud.google.com/functions/docs/configuring/min-instances.

[33] "Run OpenFaaS Functions on Cloud Run". Last accessed 05/08/2022, [Online]. Available: https://www.openfaas.com/blog/openfaas-cloudrun/.

[34] "Gson library". Last accessed 26/07/2022, [Online]. Available: https://github.com/google/gson.

[35] "Significant Changes in JDK 16 Release". Last accessed 20/07/2022, [Online]. Available: https://docs.oracle.com/en/java/javase/16/migrate/significant-changes-jdk-release.html#GUID-327C39ED-C3FD-4637-906A-36C6697E85D5.

[36] "Kotlin coroutines". Last accessed 27/07/2022, [Online]. Available: https://kotlinlang.org/docs/coroutines-overview.html.

[37] "Kotlin Data classes". Last accessed 24/09/2022, [Online]. Available: https://kotlinlang.org/docs/data-classes.html.

[38] "Test code using JUnit in JVM". Last accessed 10/06/2022, [Online]. Available: https://kotlinlang.org/docs/jvm-test-using-junit.html.

[39] "Ktor – JSON serializer". Last accessed 20/07/2022, [Online]. Available: https://ktor.io/docs/serialization-client.html#register_json.

[40] "OAuth – Ktor". Last accessed 20/07/2022, [Online]. Available: https://ktor.io/docs/authentication.html#oauth.

[41] "Kudu API". Last accessed 27/07/2022, [Online]. Available: https://github.com/MicrosoftDocs/azure-docs/blob/main/articles/app-service/deploy-zip.md#kudu-api.

[42] "Deploying from a zip file or url – Kudu". Last accessed 27/07/2022, [Online]. Available: https://github.com/projectkudu/kudu/wiki/Deploying-from-a-zip-file-or-url.

[43] "MetricService". Last accessed 19/06/2022, [Online]. Available: https://cloud.google.com/monitoring/api/ref_v3/rpc/google.monitoring.v3#metricservice.

[44] "Understanding serverless cold start". Last accessed 11/06/2022, [Online]. Available: https://azure.microsoft.com/en-us/blog/understanding-serverless-cold-start/.

[45] "Azure Functions Core Tools". Last accessed 11/03/2022, [Online]. Available: https://docs.microsoft.com/en-gb/azure/azure-functions/functions-run-local?tabs=v4%2Cwindows%2Ccsharp%2Cportal%2Cbash#prerequisites.

[46] "Azure Functions – Overview of plans". Last accessed 11/06/2022, [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-functions/functions-scale#overview-of-plans.

[47] "Using OpenFaaS on AKS". Last accessed 05/08/2022, [Online]. Available: https://docs.microsoft.com/en-us/azure/aks/openfaas.

[48] "Get resource changes". Last accessed 12/07/2022, [Online]. Available: https://learn.microsoft.com/en-us/azure/governance/resource-graph/how-to/get-resource-changes.

[49] "Azure Functions – Service limits". Last accessed 13/06/2022, [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale#service-limits.

[50] Dana Petcu, "Multi-Cloud: expectations and current approaches", in *Proceedings of the International Workshop on Multi-Cloud Applications and Federated Clouds*, 2013, pages 1–6. DOI: 10.1145/2462326.2462328.

[51] "OpenFaaS". Last accessed 05/08/2022, [Online]. Available: https://www.openfaas.com/.

[52] "Triggers – OpenFaaS". Last accessed 05/08/2022, [Online]. Available: https://docs.openfaas.com/reference/triggers/#cloudevents.

[53] "Templates – OpenFaaS". Last accessed 06/08/2022, [Online]. Available: https://github.com/openfaas/templates.

[54] "Polycloud". Last accessed 03/12/2021, [Online]. Available: https://www.thoughtworks.com/radar/techniques/polycloud.

[55] Volker Ziegler, Peter Schneider, Harish Viswanathan, Michael Montag, Satish Kanugovi, and Ali Rezaki, "Security and Trust in the 6G Era", *IEEE Access*, vol. 9, pages 142 314–142 327, 2021. DOI: 10.1109/ACCESS.2021.3120143.

[56] "Pulumi". Last accessed 17/07/2022, [Online]. Available: https://www.pulumi.com.

[57] "Cloud Framework (Preview)". Last accessed 23/09/2022, [Online]. Available: https://www.pulumi.com/docs/tutorials/cloudfx/.

[58] "@pulumi/cloud". Last accessed 23/09/2022, [Online]. Available: https://www.npmjs.com/package/@pulumi/cloud.

[59] "Magic Functions in Pulumi". Last accessed 12/03/2022, [Online]. Available: https://www.pulumi.com/blog/lambdas-as-lambdas-the-magic-of-simple-serverless-functions/#magic-functions.

[60] Pedro Rodrigues, Filipe Freitas, and José Simão, "Quickfaas: Providing portability and interoperability between faas platforms", *Future Internet*, vol. 14, no. 12, 2022, ISSN: 1999-5903. DOI: 10.3390/fi14120360.

[61] "QuickFaaS Essentials repository". Last accessed 18/10/2022, [Online]. Available: https://github.com/Pexers/quickfaas-essentials.

[62] "Perform resumable uploads". Last accessed 28/07/2022, [Online]. Available: https://cloud.google.com/storage/docs/performing-resumable-uploads.

[63] V. Yussupov, U. Breitenbücher, A. Kaplan, and F. Leymann, "SEAPORT: Assessing the Portability of Serverless Applications", in *Proceedings of the 10th International Conference on Cloud Computing and Services Science*, 2020, pages 456–467. DOI: 10.5220/0009574104560467.

[64] "OAuth threats – Obtaining Client Secrets". Last accessed 23/07/2022, [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6819#section-4.1.1.

[65] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot, "Benchmarking, Analysis, and Optimization of Serverless Function Snapshots", in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pages 559–572. DOI: 10.1145/3445814.3446714.

[66] S. Eismann, J. Grohmann, E. van Eyk, N. Herbst, and S. Kounev, "Predicting the Costs of Serverless Workflows", in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, 2020, pages 265–276. DOI: 10.1145/3358960.3379133.

[67] Adam Eivy and Joe Weinman, "Be Wary of the Economics of "Serverless" Cloud Computing", *IEEE Cloud Computing*, vol. 4, no. 2, pages 9–11, 2017. DOI: 10.1109/MCC.2017.32.

[68] "Serverless Framework". Last accessed 17/07/2022, [Online]. Available: https://www.serverless.com.

[69] Hai Duc Nguyen, Chaojie Zhang, Zhujun Xiao, and Andrew A. Chien, "Real-Time Serverless: Enabling Application Performance Guarantees", in *Proceedings of the 5th International Workshop on Serverless Computing*, 2019, pages 1–6. DOI: 10.1145/3366623.3368133.

[70] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski, "The Rise of Serverless Computing", *Communications of the ACM*, vol. 62, no. 12, page 44, 2019, ISSN: 0001-0782. DOI: 10.1145/3368454.

[71] M. Wurster, U. Breitenbücher, K. Képes, F. Leymann, and V. Yussupov, "Modeling and Automated Deployment of Serverless Applications using TOSCA", in *Proceedings of the IEEE 11th Conference on Service-Oriented Computing and Applications (SOCA)*, 2018, pages 73–80. DOI: 10.1109/SOCA.2018.00017.

[72] Pascal Maissen, Pascal Felber, Peter Kropf, and Valerio Schiavoni, "FaaSdom: A Benchmark Suite for Serverless Computing", in *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems*, 2020, pages 73–84. DOI: 10.1145/3401025.3401738.

[73] Eelco Dolstra, Martin Bravenboer, and Eelco Visser, "Service Configuration Management", in *Proceedings of the 12th international workshop on Software configuration management*, 2005, pages 83–98. DOI: 10.1145/1109128.1109135.

[74] Erwin van Eyk, Alexandru Iosup, Simon Seif, and Markus Thömmes, "The SPEC cloud group's research vision on FaaS and serverless architectures", in *Proceedings of the 2nd International Workshop on Serverless Computing*, 2017, 1–4. DOI: 10.1145/3154847.3154848.

[75] "ESOCC 2022 Conference proceedings". Last accessed 22/12/2022, [Online]. Available: https://link.springer.com/book/9783031232992.

[76] "Terraform". Last accessed 17/07/2022, [Online]. Available: https://www.terraform.io.

[77] "Authenticating using a Service Principal with a Client Secret". Last accessed 04/12/2021, [Online]. Available: https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/guides/service_principal_client_secret#creating-a-service-principal.

[78] "Terraform configuration reference". Last accessed 04/12/2021, [Online]. Available: https://registry.terraform.io/providers/hashicorp/google/latest/docs/guides/provider_reference#authentication.

[79] "Use global variables to reuse objects in future invocations". Last accessed 07/07/2022, [Online]. Available: https://cloud.google.com/functions/docs/bestpractices/tips#use_global_variables_to_reuse_objects_in_future_invocations.

[80] "Token types". Last accessed 21/09/2022, [Online]. Available: https://cloud.google.com/docs/authentication/token-types#access.

[81] "Configurable token lifetimes in the Microsoft identity platform". Last accessed 21/09/2022, [Online]. Available: https://learn.microsoft.com/en-us/azure/active-directory/develop/active-directory-configurable-token-lifetimes#access-tokens.

[82] Paul Lipton, "Escaping Vendor Lock-in with TOSCA, an Emerging Cloud Standard for Portability", *CA Technology Exchange 4*, pages 49–55, 2013.

[83] "Vendor lock-in and cloud computing". Last accessed 07/11/2021, [Online]. Available: https://www.cloudflare.com/en-gb/learning/cloud/what-is-vendor-lock-in/.

[84] Justice Opara-Martins, Reza Sahandi, and Feng Tian, "Critical review of vendor lock-in and its impact on adoption of cloud computing", in *Proceedings of the International Conference on Information Society (i-Society 2014)*, 2014, pages 92–97. DOI: 10.1109/i-Society.2014.7009018.

[85] "Web Apps – Create Or Update". Last accessed 28/07/2022, [Online]. Available: https://docs.microsoft.com/en-us/rest/api/appservice/web-apps/create-or-update.

[86] Garrett McGrath and Paul R. Brenner, "Serverless Computing: Design, Implementation, and Performance", in *Proceedings of the IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, 2017, pages 405–410. DOI: 10.1109/ICDCSW.2017.36.

# Appendix A

## A.1 QuickFaaS screenshots

The following Figures A.1, A.2 and A.3 illustrate screenshots of the desktop application in action.

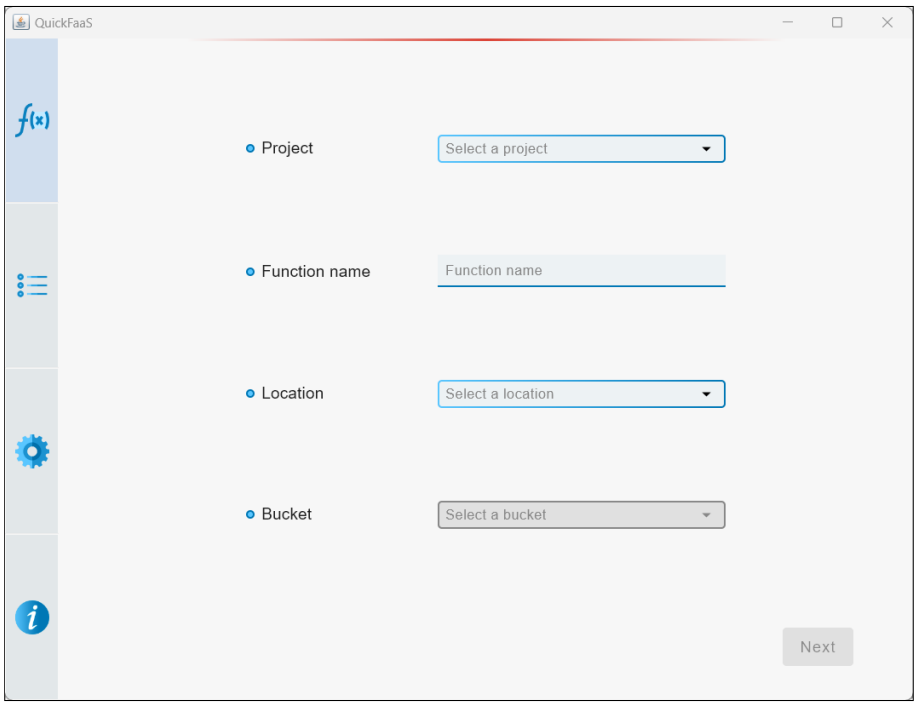



Figure A.1: Resource configuration screenshot

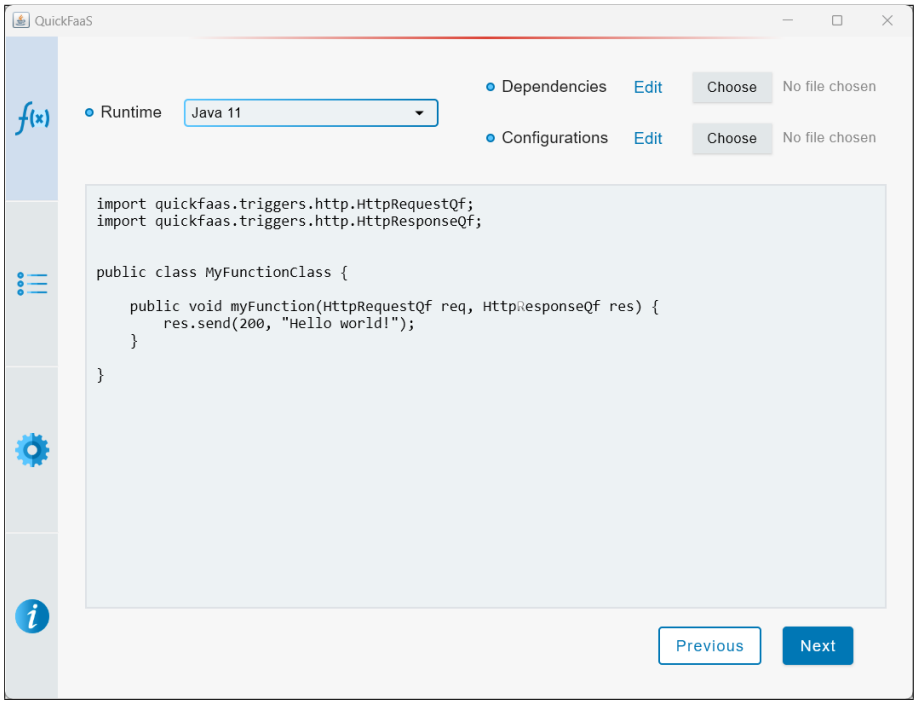



Figure A.2: Cloud-agnostic function definition screenshot

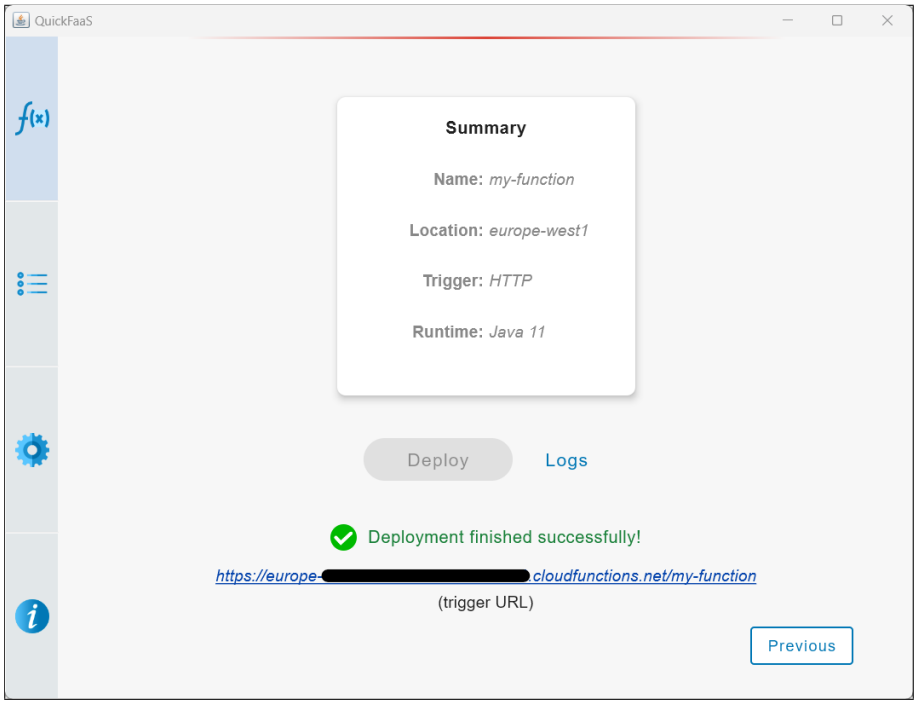



Figure A.3: FaaS deployment screenshot

## A.2 Complete uniform programming model

For this section, the full uniform programming model is provided in Figure A.4.

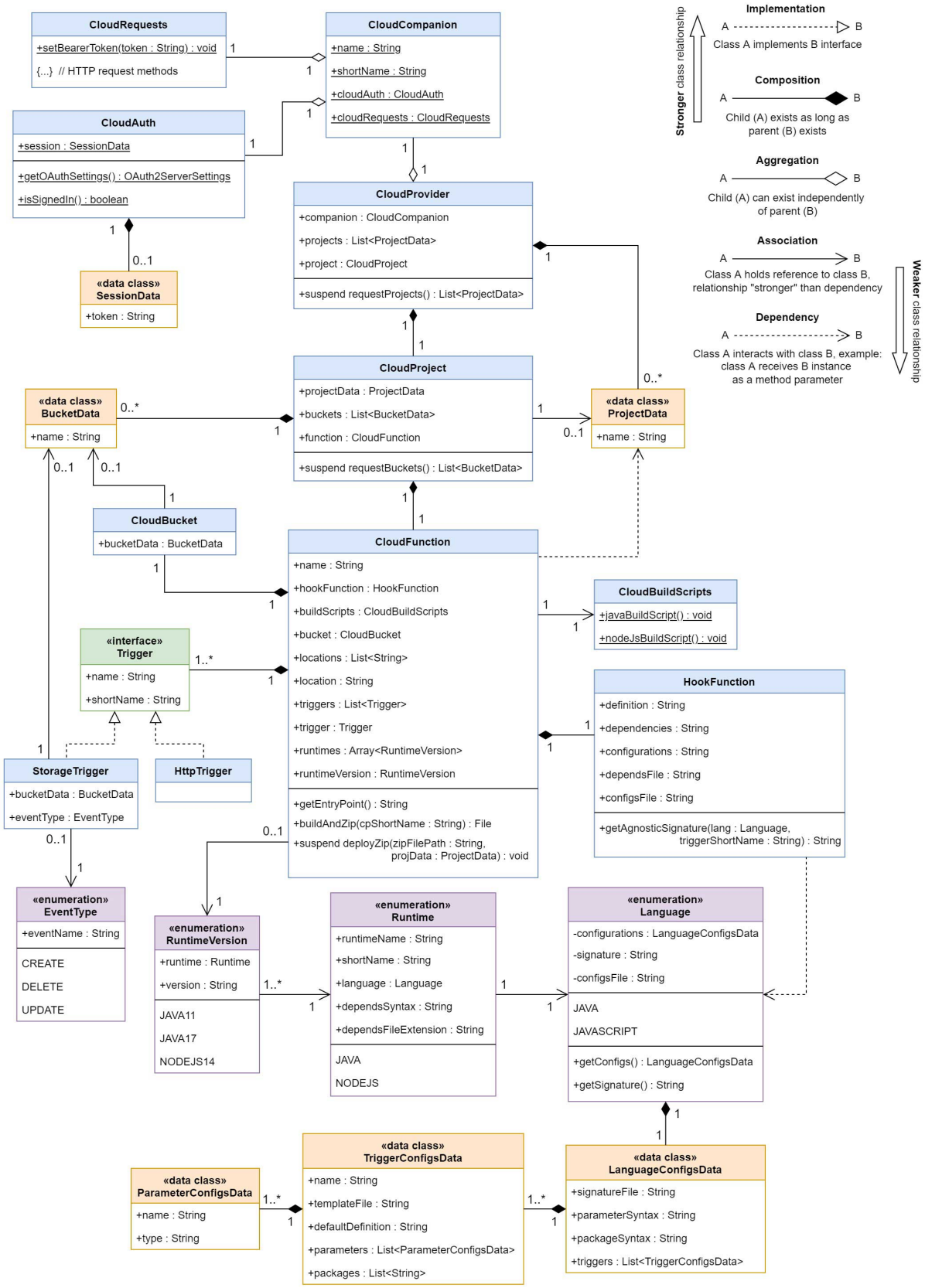



Figure A.4: Uniform programming model