future internet

MDPI

Article

QuickFaaS: Providing Portability and Interoperability between FaaS Platforms

Pedro Rodrigues [1,*], Filipe Freitas [1,*] and José Simão [1,2,3]

[1] Instituto Superior de Engenharia de Lisboa (ISEL), Politécnico de Lisboa (IPL), 1959-007 Lisboa, Portugal
[2] INESC-ID, 1000-029 Lisboa, Portugal
[3] Future Internet Technologies, Instituto Superior de Engenharia de Lisboa (ISEL), 1959-007 Lisboa, Portugal
* Correspondence: pedro-rodri@outlook.com (P.R.); filipe.freitas@isel.pt (F.F.)

Abstract: Serverless computing hides infrastructure management from developers and runs code on-demand automatically scaled and billed during the code's execution time. One of the most popular serverless backend services is called Function-as-a-Service (FaaS), in which developers are often 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, leading to various vendor lock-in problems. In this work, we define three cloud-agnostic models that compose FaaS platforms. Based on these models, we developed QuickFaaS, a multi-cloud interoperability desktop tool targeting cloud-agnostic functions and FaaS deployments. The proposed cloud-agnostic approach enables developers to reuse their serverless functions in different cloud providers with no need to change code or install extra software. We also provide an evaluation that validates 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 shows that a cloud-agnostic approach does not significantly impact the function's performance.

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

Citation: Rodrigues, P.; Freitas, F.; Simão, J. QuickFaaS: Providing Portability and Interoperability between FaaS Platforms. *Future Internet* **2022**, *14*, 360. https://doi.org/10.3390/fi14120360

Academic Editors: Xu Wang, Bin Shi and Yili Fang

Received: 22 October 2022
Accepted: 28 November 2022
Published: 30 November 2022

1. Introduction

Serverless computing was a major technological breakthrough that has been drawing interest both from the industry and research, largely due to the recent shift of enterprise application architectures to containers and microservices [1–3]. Serverless potential is sustained by the great abstraction of server management challenges with low costs [4,5]. 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.

The main idea behind cloud serverless computing is to mitigate the need for infrastructure management while keeping control of the system configurations [6]. There are some extra reasons to embrace serverless solutions: (a) ***Faster deployment and delivery***—developers can easily deploy serverless applications without the requirement for server administration experience [7]; (b) ***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 [8]; (c) ***Cost efficiency***—follows the *pay as you go* pricing model [9–11], 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 [12], such as VMs booting time; (d) ***Greener computing***—the usage of computational resources is more efficient, less computing power is wasted on idle state.

Major cloud providers [13,14], such as Microsoft Azure, Amazon Web Services (AWS) and Google Cloud Plataform (GCP), have available widely adopted solutions for deploying

functions as a service. This is being used in many use cases in different areas, from video, image, text processing [15–17] to receiving data from IoT and edge applications [18–20].

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. Some effects of vendor lock-in are relevant to point out [15,21]: (a) Struggle when switching between cloud providers due to the impact at operational level, in addition to costs; (b) 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; (c) A provider may impose price increases for the services, knowing that their clients are locked-in.

As mentioned by the Research Cloud group of the Standard Performance Evaluation Corporation (SPEC) [22]:

> "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 due to function signature requirements and the usage of libraries that are unique to cloud providers, resulting in non-portable solutions.

This work focuses on characterizing and describing three cloud-agnostic models that compose FaaS platforms. The *Authentication mechanism* and the *FaaS deployment* models relate to cloud interoperability due to requiring the exchange of information between the application and the cloud provider. The third model is the *Function definition*, which represents the entities and respective attributes that determine the definition of a cloud-agnostic function, thus being related to the portability of FaaS applications.

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 with no need to install extra software. 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. The tool is capable of doing full deployments of serverless functions automatically, in different providers, without having to rewrite the function code. Despite being built using the JVM ecosystem, the models were designed to be independent of the language where the functions will be implemented. We also provide an evaluation that validates 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 shows that the cloud-agnostic approach does not have a significant impact on the function's performance (execution time, memory usage) and package size. 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.

The remainder of this paper is organized as follows: Section 2 discusses recent related works concerning multi-cloud interoperability and FaaS portability. Section 3 reveals and details the three modules that compose the cloud-agnostic solution, together with a number of use cases to exemplify the usage of cloud-agnostic functions. Section 4 introduces the desktop tool QuickFaaS by presenting its architecture and technologies, and finally, the uniform programming model is characterized. Section 5 evaluates different metrics to measure the impact of a cloud-agnostic approach on the function's performance. Section 6 concludes this work by mentioning its main achievements and by briefly discussing in what aspects we believe QuickFaaS can be improved as a future work.

2. Related Work

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 [23], focus 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.

2.1. Terraform

Terraform [24] 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) 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.

2.2. Serverless Framework

Another example of a cloud orchestration tool is Serverless Framework [25]. 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 [15].

2.3. Pulumi

Pulumi [26] 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, provider-specific software is required 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* [27]. 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* [28], 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* [29], 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 is 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.

2.4. OpenFaaS

OpenFaaS [30] is a framework for building serverless functions on the top of containers through the use of docker and kubernetes. 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.

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 [31]. However, despite supporting many different kinds of events [32], OpenFaaS functions cannot be directly triggered by the majority of events originated within the cloud infrastructure, such as storage or database events. Workarounds can be implemented, but would not 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. Lambada

Automated code deployment and transparent code offloading to FaaS platforms are interesting new workflows in cloud application software engineering and testing scenarios. A work from the Zurich University of Applied Sciences [33] introduces Lambada, a tool to automate this workflow by shifting Python functions and methods along with module-internal code dependencies into hosted function services. The work shares similarities with automated transformation approaches for Java [34].

However, the proposed *FaaSification* process of converting a code structure into an executable FaaS format will produce serverless functions that are only compatible with AWS. These functions rely on cloud-specific signatures and libraries, making Lambada not a valid solution for the problems addressed in this work. Moreover, the *lambdafication* process of rewriting code, as stated in the paper, introduces significant overhead in execution time.

2.6. 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 [35] 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 use cases.

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 does not 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 does not illustrate the abstraction of different types of authentication mechanisms used by cloud orchestration tools in order to obtain 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.

3. Cloud-Agnostic Models

We decided to use entity-relationship models (ERMs) to represent the main entities and relations of the cloud-agnostic solution. We adopted a top-down approach when designing the ERMs, the more general one can be found in Figure 1. This model illustrates the three main modules that compose the cloud-agnostic solution.

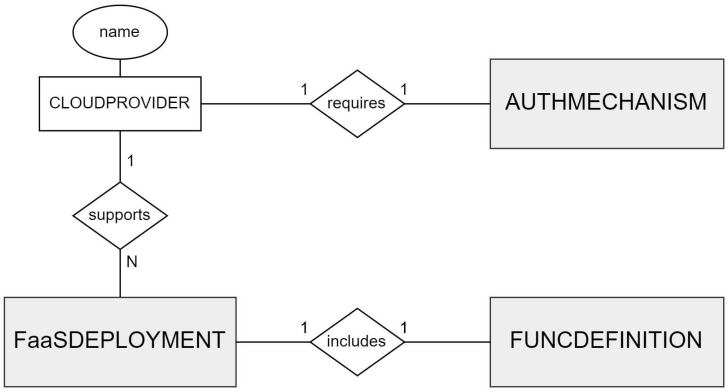

Figure 1. Abstract ERM of the cloud-agnostic solution.

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.

3.1. 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 [15], resulting in portability-related issues.
- ***Unique libraries.*** There are no common libraries shared between cloud providers that could attenuate portability issues when developing solutions to multiple platforms. Library dependencies are introduced not only for processing custom data types, but also for interacting with provider-specific APIs [15]. Switching between cloud providers requires the developer to adapt, 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 [36], while Google Cloud Platform does. The workaround requires the installation of the provider-specific tool named Azure CLI. The variety of tooling is not ideal for solutions 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 [37]. The amount of providers and their intrinsic variability results in a discrepancy between configurations that do not facilitate multi-cloud usage [38].

To evidence the differences between function signatures, exemplified below, in Listings 1 and 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 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 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 the provider's requirements.

3.2. Cloud Interoperability

Establishing a proper authentication mechanism is usually the very first step towards achieving interoperability with cloud providers. In Figure 2, we represent the entities and respective attributes that model the authentication mechanism based on the OAuth 2.0 protocol.

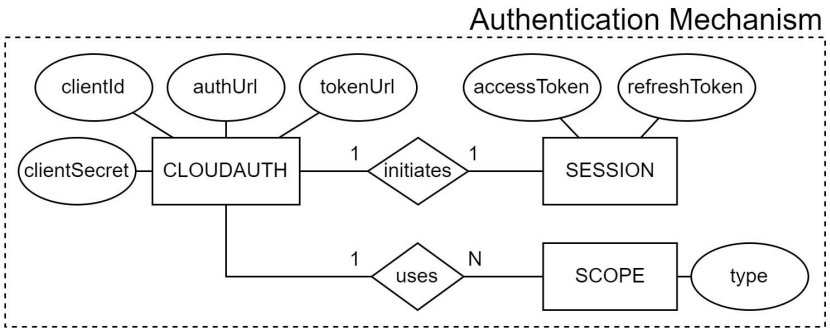

Figure 2. Authentication mechanism ERM.

This protocol is commonly used by different types of applications to authenticate users in various platforms, so developers can easily find documentation on how to adapt this model to their provider of choice. 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). Trying to obfuscate client secrets in installed applications is one of the main challenges in adopting this protocol, since they can be always recovered using the abundance of reverse engineering and debugging tools [39].

With OAuth 2.0, an application can request one or more scopes (*SCOPE* entity), and 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. Access tokens normally last for about an hour in both GCP and MsAzure [40,41].

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 only illustrates cloud entities that participate in the deployment of a FaaS application.

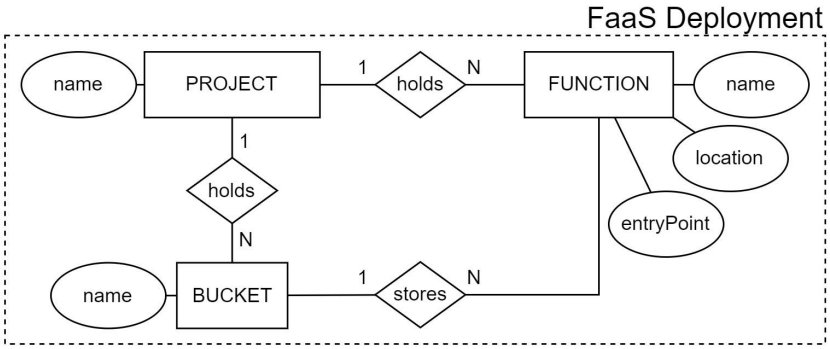

Figure 3. FaaS deployment ERM.

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 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 naming terminologies for services and resources. For instance, 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. 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.

Serverless functions have access to a wide variety of libraries offered by cloud providers. Cloud-agnostic libraries should provide uniform access to operations 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 for the sake of providing a full cloud-agnostic usage. Additionally, if for some reason a certain cloud-agnostic operation cannot be implemented for a particular provider, the documentation should warn developers specifying that the operation is incompatible with that cloud provider.

The entities and respective attributes that model a cloud-agnostic function definition are represented in Figure 4.

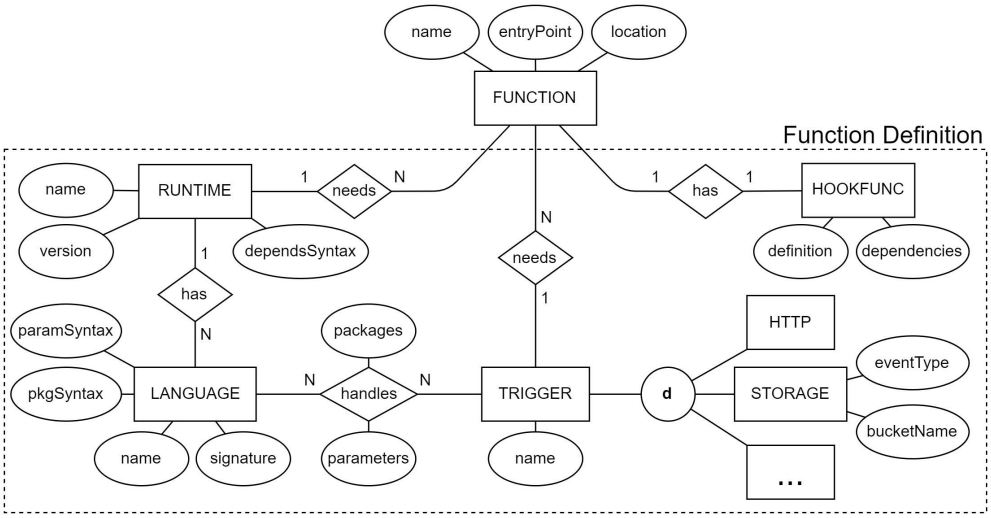

Figure 4. Function definition ERM.

The *FUNCTION* entity is the exact same entity as the one illustrated in Figure 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 the runtime environment (*RUNTIME* entity), capable of running code from a certain programming language (*LANGUAGE* entity). The developer has the option to specify a list of external dependencies (*dependencies* attribute from *HOOKFUNC* entity) to be downloaded from a remote repository if needed. These should follow the appropriate syntax when specified, that 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 and 4, are two examples of cloud-agnostic function signature skeletons used for Java and JavaScript programming languages, respectively.

Listing 3. Java cloud-agnostic signature.

```
<packages>

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

Listing 4. JavaScript cloud-agnostic signature.

```
<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).

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 5, was based on a frequently-described example of an event-driven FaaS-based application, called the *thumbnail generation* [15].

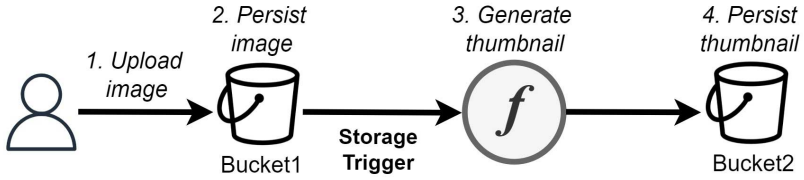

Figure 5. Use case 1—*thumbnail generation*.

The use case starts with the upload of an image file to persist 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. 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 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, and we named it *store translation*.

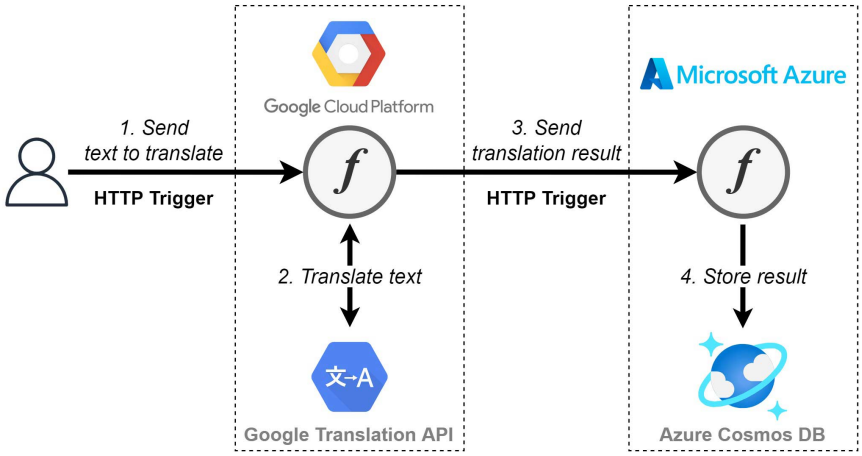

Figure 6. Use case 2—*store translation*.

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 [42]. 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 develop-

ers 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 7. This use case was designed exclusively for the purpose of the performance testing evaluated in Section 5.

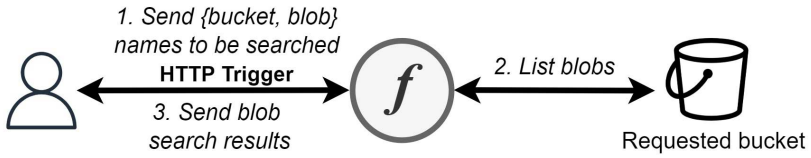

Figure 7. Use case 3—*search blobs*.

The cloud-agnostic function is triggered via HTTP requests, in which the users must specify the blob name that 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.

4. 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 change a single line of 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 next subsections, we describe the system's architecture, as well as the technologies utilized by QuickFaaS.

4.1. Architecture

An overview of the system's architecture can be visualized in Figure 8. This type of diagram is known as a *deployment diagram*, typically designed using UML. These 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.

We adopted a design pattern that is commonly used in various types of GUI applications, called Model-View-Controller (MVC). 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 behavior and data of the application domain, responds to requests for information about its state, and responds to instructions to change state. The View simply defines how the data should be displayed to the user.

4.2. 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 as well as using 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 [43], and the JSON serialization to the client [44]. Finally, the View was developed using Compose

for Desktop [45], a relatively new UI framework that provides a declarative and reactive approach to create desktop user interfaces with Kotlin.

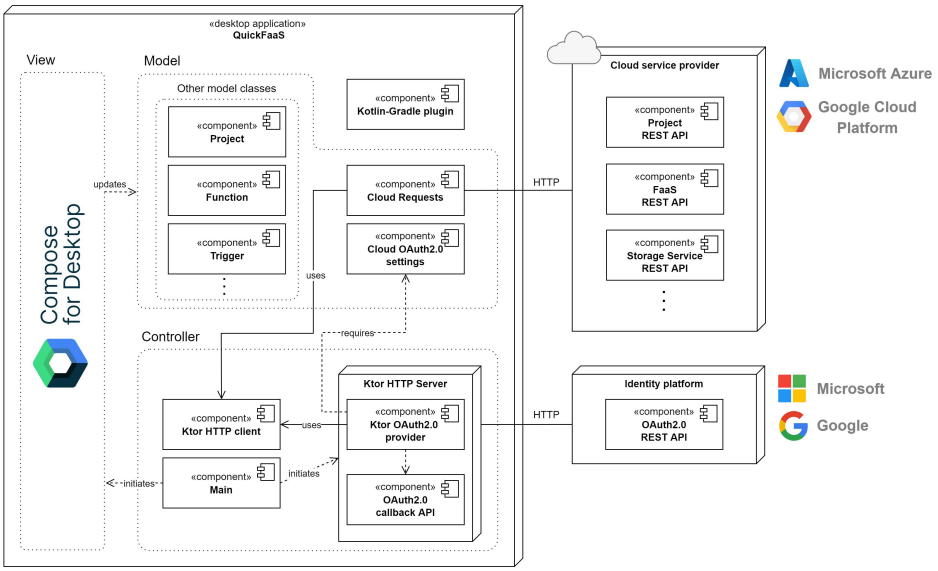

Figure 8. Deployment diagram.

Notice that no extra software is 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 are not yet supported.

4.3. 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 detailed in a dedicated subsection and has a corresponding ERM already described in Section 3. The full programming model, connecting all class diagrams together, can be found in Appendix A.1. 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 implement 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.

4.3.1. 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.

Automated pipelines are often adopted by organizations when delivering cloud-native apps to clients. Despite being a popular industry-standard authorization protocol, OAuth 2.0 comes with the trade-off of being hard to integrate with automation scripts. Usually, developers have to rely on provider-specific tools (e.g., gcloud, Azure CLI, etc.) and programming interfaces to access OAuth services. QuickFaaS abstracts the developer from these problems and provides a uniform interface to each provider's authentication service.

The programming classes that resulted from the ERM previously illustrated in Figure 2 are now represented in Figure 9.

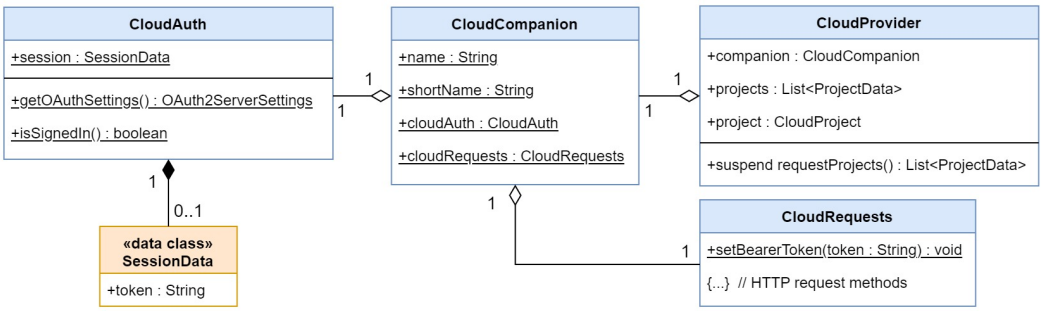

Figure 9. Authentication Mechanism class diagram.

The authentication process starts once the user decides which cloud provider 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. 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 (*OAuth2ServerSettings*). These settings include the authorization page URL, the access token request URL, the scopes, and the application's client ID and client secret. *CloudRequests* classes define several other methods responsible for making HTTP requests to the provider APIs.

4.3.2. Function Definition

Serverless functions need to follow provider-specific signatures in order to be 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 behavioral design pattern, identified by Gamma, called the *Template Method Pattern* [46,47]:

> "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 is 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 do not 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 A2. Below, in Listings 5 and 6, 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.

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 a JAR file. Even though the libraries may look cloud-agnostic from the user's perspective, they are interacting with unique APIs from providers to execute specific operations.

Figure 10 is based on the ERM previously illustrated in Figure 4. The use of a hook function is an easy concept to be implemented in other languages such as C-Sharp, JavaScript, or Python. This expansion is possible by adding new constants to runtime-related enumeration classes and configuration files for data classes (*ConfigsData*).

Listing 5. 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);
        new MyFunctionClass().myFunction(reqQf, resQf); // Calls hook function
    }
}
```

Listing 6. 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!");
    }
}
```

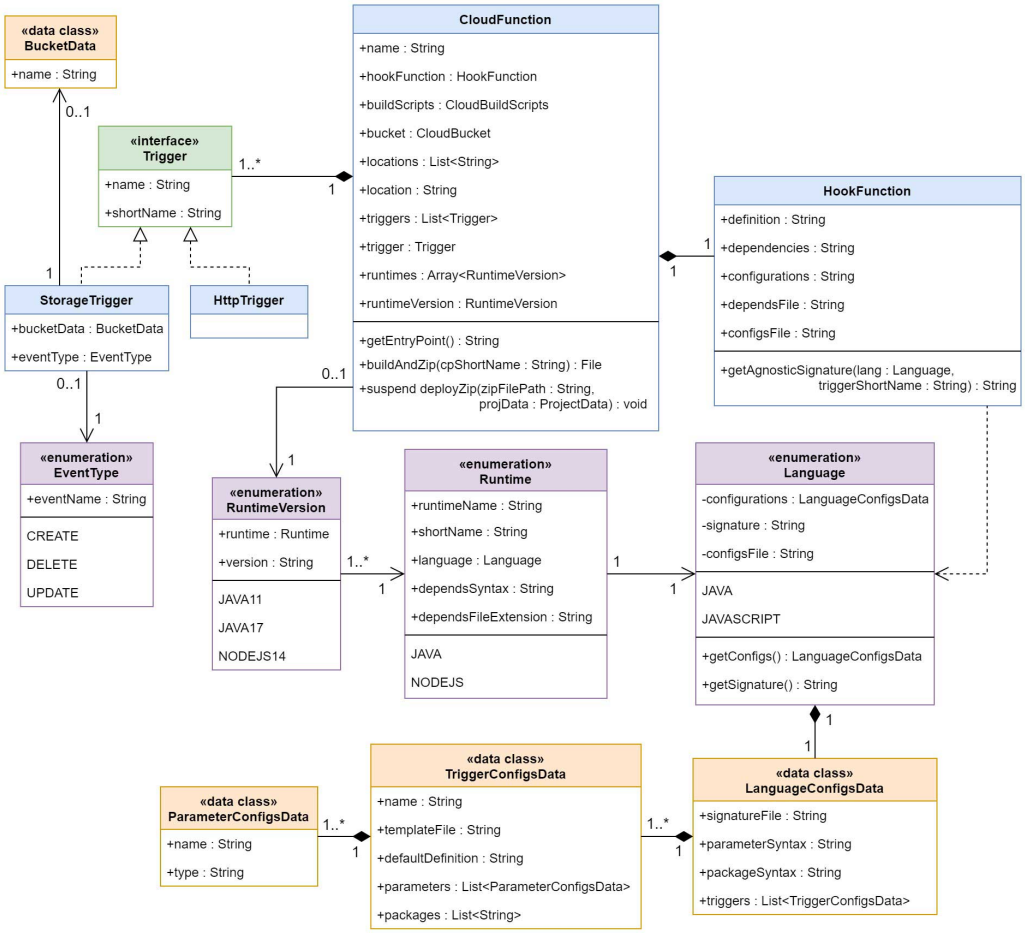

Figure 10. Function Definition class diagram.

To start the cloud-agnostic function development, the user must first choose which runtime environment 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. 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 the function's execution time. In some cases, a few configurations are mandatory to be specified, otherwise, certain cloud-agnostic libraries will 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 7, written in JSON.

Listing 7. 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. 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 [48]. 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 8.

Listing 8. Use case 1 function definition—*thumbnail generation*.

```
import ...
import quickfaas.resources.storage.BucketQf;
import quickfaas.resources.storage.StorageQf;
import quickfaas.triggers.storage.BlobQf;
import quickfaas.triggers.storage.BucketEventQf;

public class MyFunctionClass {
    public void myFunction(BucketEventQf event, BlobQf blob) {
        BucketQf bucket1 = StorageQf.newBucket(event.getBucketName());
        byte[] source = bucket1.readBlob(blob.getName());
        byte[] thumbnail = generateThumbnail(source, "jpeg");
        BucketQf bucket2 = StorageQf.newBucket("bucket2thumbnails");
        bucket2.createBlob("thumbnail-" + blob.getName(), thumbnail, "image/jpeg");
    }
    public byte[] generateThumbnail(byte[] source, String type) {...}
}
```

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 *getBucketName* operation, from the *BucketEventQf* class (line 9), returns the bucket name where the storage event occurred. It is also worth mentioning that the *newBucket* static

operation, from the *StorageQf* class (lines 9 and 12), returns a reference to an existing bucket, it does not create a new one. The *generateThumbnail* operation definition was omitted due to being irrelevant in this context (line 15).

4.3.3. FaaS Deployment

FaaS deployments can be challenging when dealing with multiple cloud providers that require the usage of different types of environments. QuickFaaS benefits once again from a common mechanism available in the supported cloud providers that avoids the need to install extra provider-specific tooling. The deployment mechanism works by uploading a ZIP archive using provider-specific APIs. The following model, represented in Figure 11, shows the programming classes that resulted from the ERM illustrated in Figure 3.

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, 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.

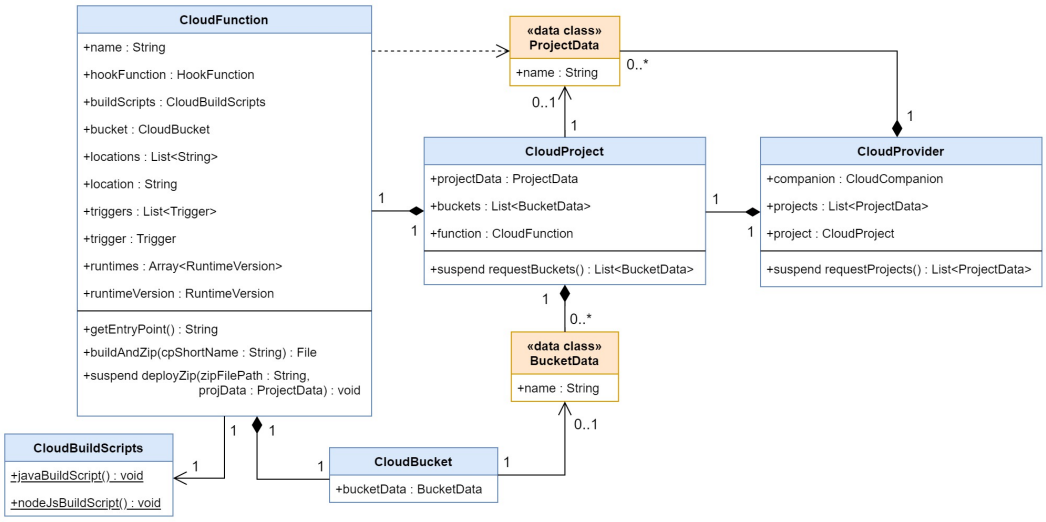

Figure 11. FaaS Deployment class diagram.

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, no build tool would be necessary, only a package manager tool, such as *npm*, to download the required modules.
   The executable JAR file that 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 [49,50]. 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.
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. Cloud providers offer different APIs and services to enable FaaS deployments. For instance, MsAzure requires the deployment of a *function app* first [51], which is where ZIP archives are 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 resource. 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* [52,53].
As for GCP, we first used the Cloud Storage API to initiate a *resumable upload* of the ZIP archive to the selected storage bucket [54]. The *create* operation, from the Cloud Functions API, is then invoked to deploy the FaaS resource [55]. The ZIP sources are automatically loaded during deployment using the *sourceArchiveUrl* property that specifies the exact location of the ZIP archive within the storage bucket.

As shown in Figure 12, 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.

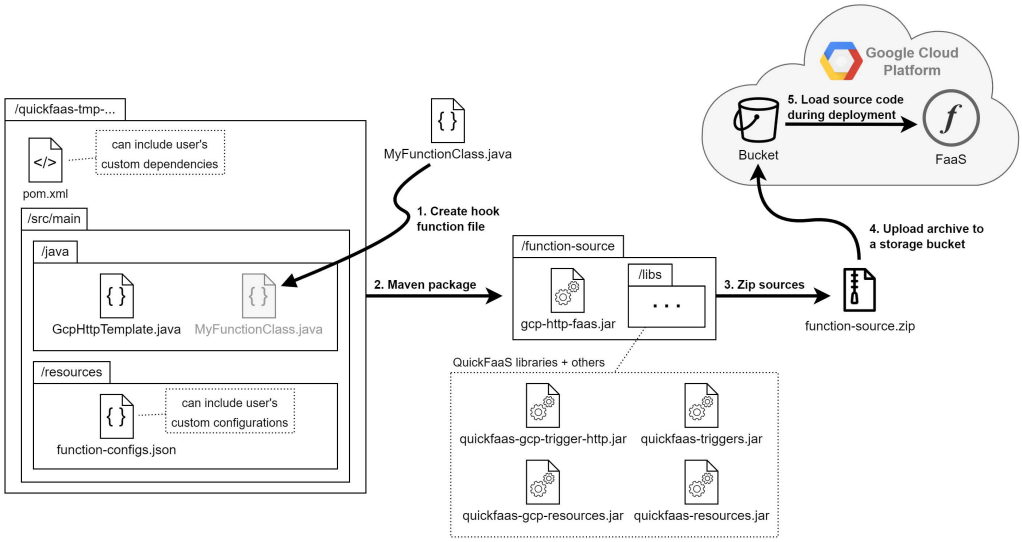

Figure 12. FaaS deployment to GCP for Java runtime.

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 [56]. This will allow QuickFaaS to programmatically manage resource metadata.

5. Evaluation

Measuring the performance of computer systems is a challenging task, specially when dealing with distributed systems managed by cloud providers. This section 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 7, was written in Java, and it is 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 a JUnit 5 testing framework and Gradle build tool [57]. For each of the given tests, we describe the data collection methodology and analyze the obtained results.

5.1. Metrics Definition

Having established the appropriate use case for evaluation, we now have 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 [58]. 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* [59].

An extra cloud service is also required when capturing the function's execution time in MsAzure, called the Application Insights. This 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 [60], 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 is not a metric currently available through Azure Monitor. There are several other related metrics [61]:

- ***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 cannot 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 accurately distinguishing 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) does not include memory consumption from secondary app operations, and (iii) the calculation formula is defined in official documentation [62]. The *Function execution units* per function execution are calculated according to Equation (1):

$$FunctionExecutionUnits = execution\_time \times \textbf{\textit{memory\_usage}}, \quad (1)$$

even though this is being applied once every function execution by Azure Monitor, the units are presented as a time-based aggregation, meaning that they cannot 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 [63]. 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 [64]. In their case, the memory usage is relevant for the calculation and comparison of consumption costs between different azure pricing models.

5.2. Function Execution Environment

The specifications for the execution environments are presented below in Table 1. Apart from the location and the runtime, these are the default values recommended by providers when deploying a serverless function [65,66].

Table 1. Function execution environment specifications.

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

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 did not make an effort in configuring machine specifications as similar as possible for the two providers.

5.3. Cold Starts

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 [12].

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 [67]. The hosting plan that best suits our needs is the consumption plan, given that the *search blobs* use case does not require heavy operations for execution, and we also do not want to avoid cold starts for the purposes of this evaluation, like the premium plan does. When using the consumption plan, containers are deallocated after roughly 20 min of inactivity, meaning that the next invocation will result in a cold start [68].

Even though there is no official documentation from Google Cloud Platform specifying 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 min of inactivity [69].

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 min after the latest execution to obtain a single data record. This interval allows the container to have enough time to transition from an idle to a stopped (cold) state. This 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 h to retrieve the 300 data records ($300 \times 20\ min$). 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 13, only takes into account the best 75% of the total measured records 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.

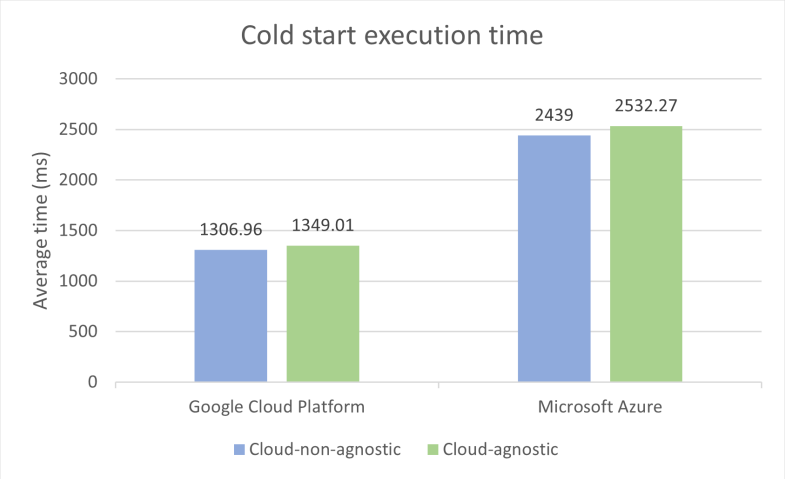

Figure 13. Cold start execution time.

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 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. As for GCP, the execution time could be improved by not reading the 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 [70]. Perhaps Ubuntu containers, from GCP, have faster start-up times on average when compared to Windows containers from MsAzure [71].

As for the memory usage, represented in Figure 14, no records were ignored when determining the average memory consumption.

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.

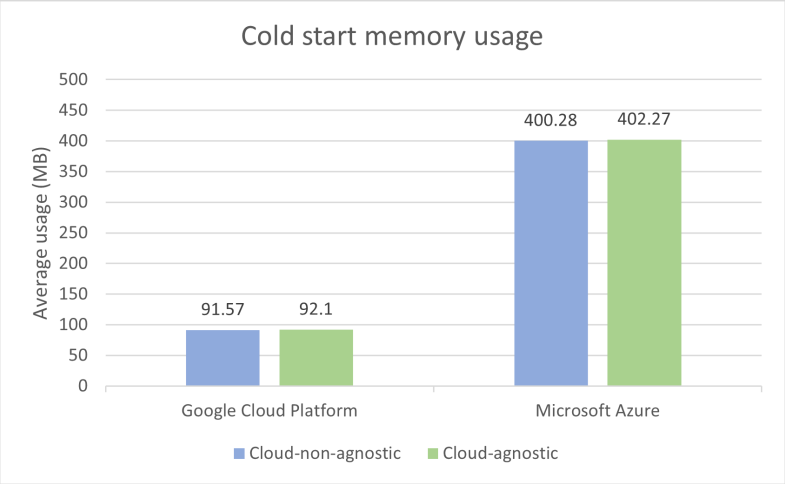

Figure 14. Cold start memory usage.

5.4. Warm Starts

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.

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. As for execution times, it is 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 (4×100 *records*), with a delay of 10 s between invocations. The reason for the high number of executions and 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

Warm starts are less likely to have high latency issues when compared to cold starts, since they do not 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 are 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 15.

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 is 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 [72]. 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.

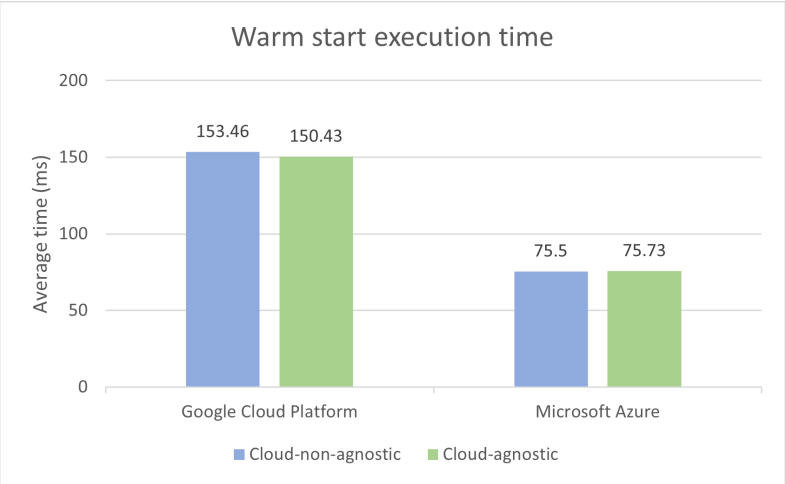

Figure 15. Warm start execution time.

It is 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 16, there was also a considerable difference in memory usage. No records were excluded this time when determining the average memory consumption.

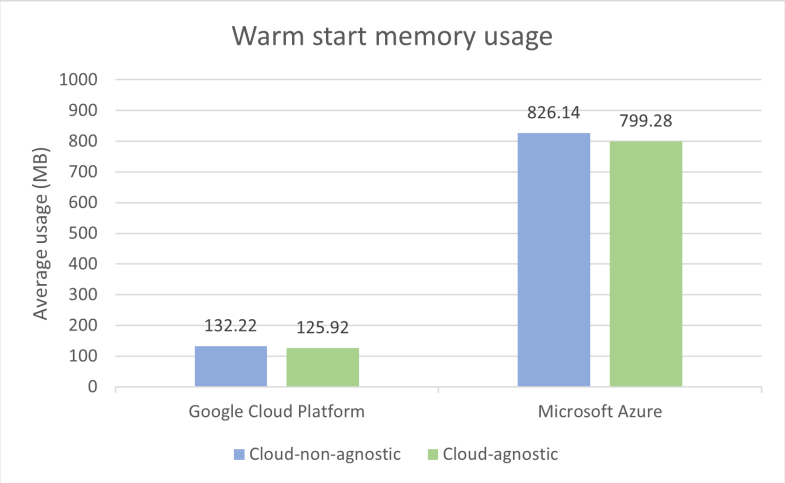

Figure 16. Warm start memory usage.

The measurements reveal that cloud-agnostic functions made a better use of the available memory when compared to the 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 cannot 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 has a negative impact in this regard. 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 deployment time was determined based on the formula specified by Equation (2):

$$DeploymentTime = zip\_upload\_time + resource\_deployment\_time, \quad (2)$$

where 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 [73]. 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 [74]. 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.

As shown in Figure 17, 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, QuickFaaS 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 [75].

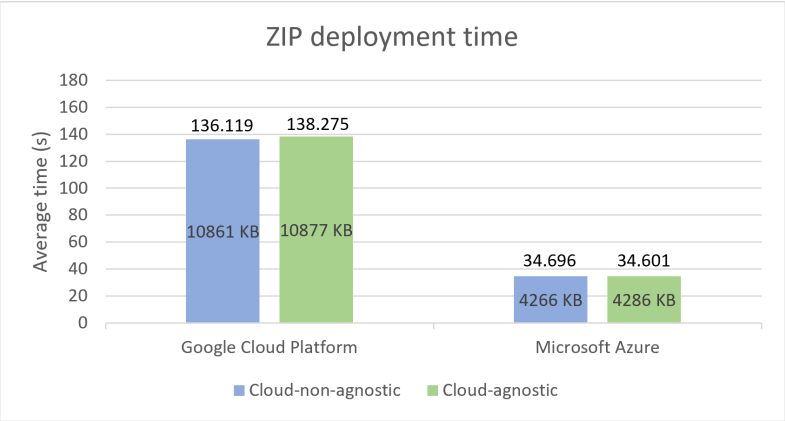

Figure 17. ZIP deployment time.

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* [65], meaning that the limit was not 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 (4×200 *records*). If no delay was applied while generating the 200 execution records, each function would have to be warm started around 2000 times (10×200 *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 15, the average execution was around 75 ms, with most execution times ranging from 60 to 80 ms.
3. ***Measurements inconsistency.***
   Latency cannot 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.

6. Conclusions and Future Work

The overall goal of this work was to characterize a solution to provide portability and interoperability between FaaS platforms. To achieve this, we defined three cloud-agnostic models that represent the main building blocks of FaaS platforms. These models were

materialized into a multi-cloud interoperability desktop tool named QuickFaaS that 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 approach enables developers to reuse their serverless functions in multiple cloud providers, with the convenience of not having to change code.

This work was an extension of a short position paper presented at the Projects Track of the 9^{th} *European Conference On Service-Oriented And Cloud Computing* (ESOCC) [76], where an initial view of the models was given. We now present a more complete view of the cloud-agnostic models, as well as a detailed characterization and description of a new uniform programming model for FaaS applications. Architectural and software design decisions taken when implementing the tool and its models were also described. Finally, a quantitative evaluation of the tool was discussed and compared with non-agnostic cloud solutions. The study has shown that the cloud-agnostic approach does not 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 [77]. 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 are 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.

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, and 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. Currently, QuickFaaS does not support agnostic monitoring of the deployed serverless functions. This feature is also considered for future work in order to minimize the need to manually interact with different platforms in a multi-cloud solution.

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.

Author Contributions: Conceptualization, P.R., F.F. and J.S.; methodology, P.R.; software, P.R.; validation, P.R., F.F. and J.S.; formal analysis, P.R.; investigation, P.R.; resources, P.R., F.F. and J.S.; data curation, P.R.; writing—original draft preparation, P.R.; writing—review and editing, P.R., F.F. and J.S.; visualization, P.R.; supervision, F.F. and J.S.; project administration, P.R., F.F. and J.S.; funding acquisition, F.F. and J.S. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by national funds through FCT, Fundação para a Ciência e a Tecnologia, under project UIDB/50021/2020, and by Instituto Politécnico de Lisboa under project IPL/2021/FaaS-IntOp_ISEL.

Data Availability Statement: The data supporting the reported results can be found in the form of excel spreadsheets at https://github.com/Pexers/quickfaas-essentials, accessed on 21 October 2022.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A

Appendix A.1

In this subsection, the full uniform programming model is provided. This type of diagram is known as class diagram, and it is designed using UML.

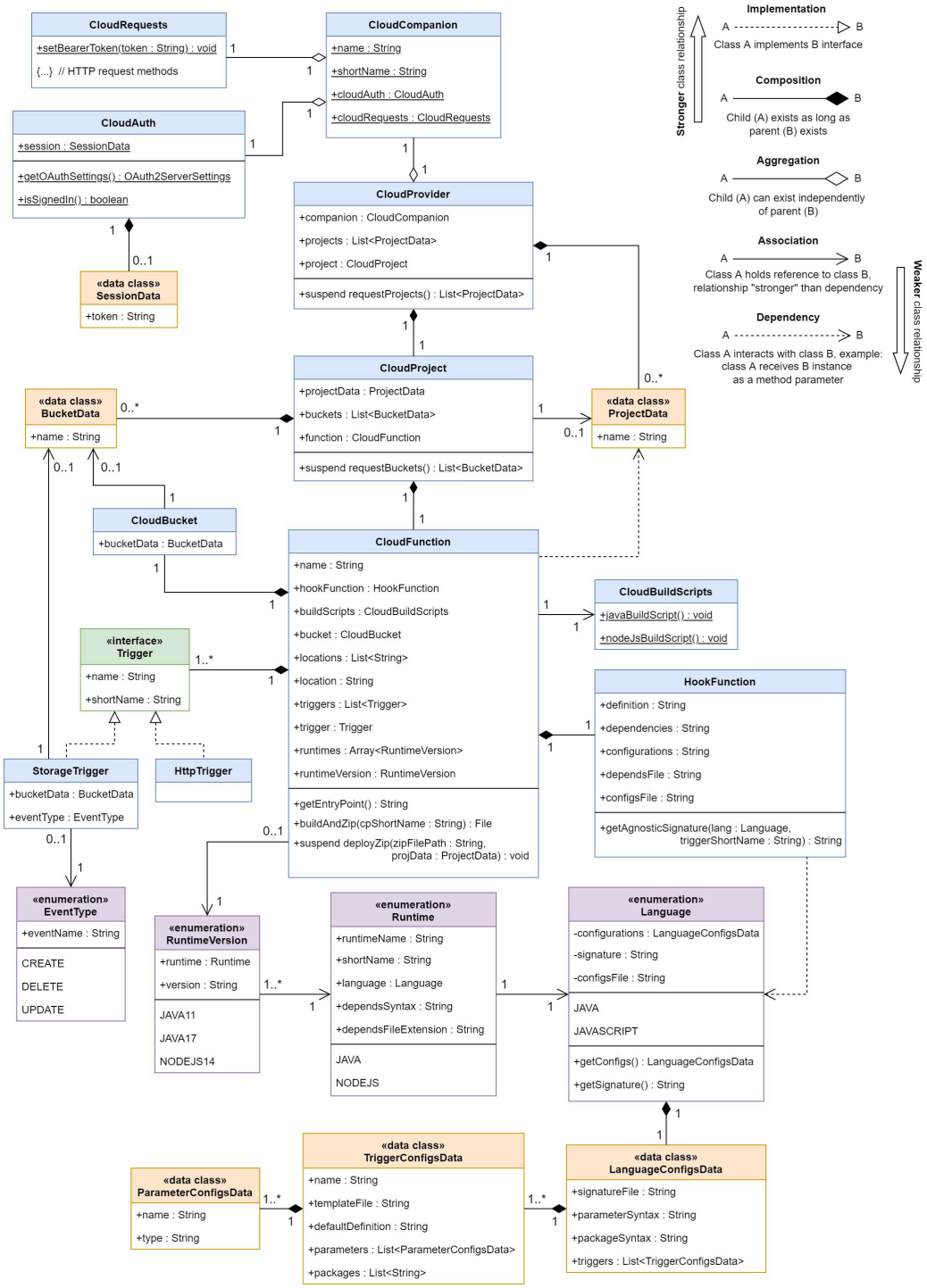

Figure A1. Uniform programming model.

Appendix A.2

The following Figure A2 illustrates a screenshot of the desktop application in action.

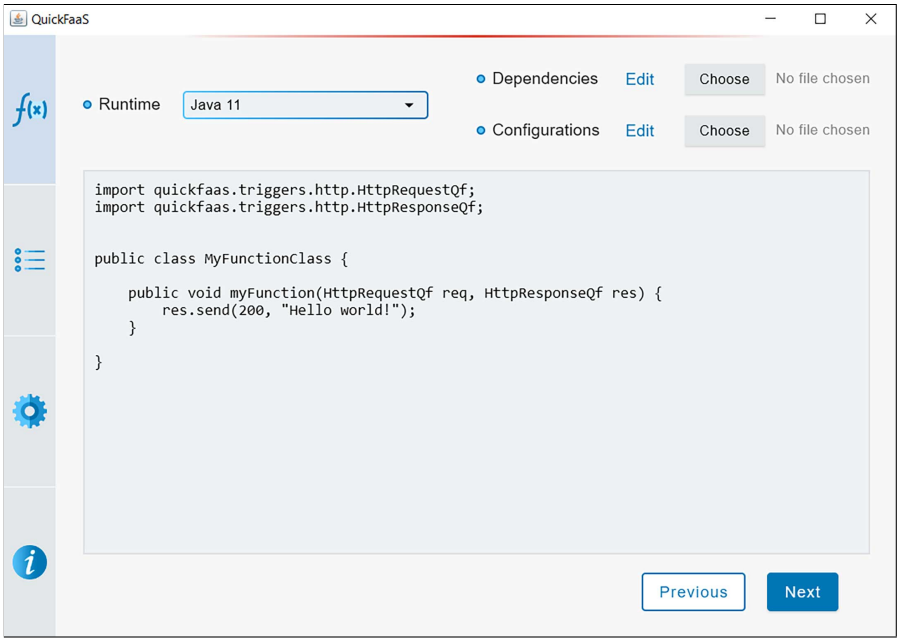

Figure A2. Cloud-agnostic function definition screenshot.

References

1. Baldini, I.; Castro, P.; Chang, K.; Cheng, P.; Fink, S.; Isahagian, V.; Mitchell, N.; Muthusamy, V.; Rabbah, R.; Slominski, A.; et al. *Serverless Computing: Current Trends and Open Problems*; Springer: Singapore, 2017. [CrossRef]
2. Castro, P.; Ishakian, V.; Muthusamy, V.; Slominski, A. The Rise of Serverless Computing. *Commun. ACM* **2019**, *62*, 44. [CrossRef]
3. Ivan, C.; Vasile, R.; Dadarlat, V. Serverless Computing: An Investigation of Deployment Environments for Web APIs. *Computers* **2019**, *8*, 50. [CrossRef]
4. Eivy, A.; Weinman, J. Be Wary of the Economics of "Serverless" Cloud Computing. *IEEE Cloud Comput.* **2017**, *4*, 9–11. [CrossRef]
5. Hsu, P.; Ray, S.; Li-Hsieh, Y.Y. Examining cloud computing adoption intention, pricing mechanism, and deployment model. *Int. J. Inf. Manag.* **2014**, *34*, 474–488. [CrossRef]
6. Aske, A.; Zhao, X. Supporting Multi-Provider Serverless Computing on the Edge. In Proceedings of the 47th International Conference on Parallel Processing Companion, Eugene, OR, USA, 13–16 August 2018; pp. 1–6. [CrossRef]
7. Nguyen, H.D.; Zhang, C.; Xiao, Z.; Chien, A. Real-Time Serverless: Enabling Application Performance Guarantees. In Proceedings of the 5th International Workshop on Serverless Computing, Davis, CA, USA, 9–13 December 2019; pp. 1–6. [CrossRef]
8. Mirabelli, M.E.; García-López, P.; Vernik, G. Bringing Scaling Transparency to Proteomics Applications with Serverless Computing. In Proceedings of the 2020 Sixth International Workshop on Serverless Computing, Delft, The Netherlands, 7–11 December 2020; pp. 55–60. [CrossRef]
9. Eismann, S.; Grohmann, J.; van Eyk, E.; Herbst, N.; Kounev, S. Predicting the Costs of Serverless Workflows. In Proceedings of the ACM/SPEC International Conference on Performance Engineering, Edmonton, AB, Canada, 25–30 April 2020; pp. 265–276. [CrossRef]
10. Elgamal, T.; Sandur, A.; Nahrstedt, K.; Agha, G. Costless: Optimizing Cost of Serverless Computing through Function Fusion and Placement. In Proceedings of the IEEE/ACM Symposium on Edge Computing (SEC), Seattle, WA, USA, 25–27 October 2018; pp. 300–312. [CrossRef]
11. Simão, J.; Veiga, L. Partial Utility-Driven Scheduling for Flexible SLA and Pricing Arbitration in Clouds. *IEEE Trans. Cloud Comput.* **2016**, *4*, 467–480. [CrossRef]
12. Ustiugov, D.; Petrov, P.; Kogias, M.; Bugnion, E.; Grot, B. 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, Virtual, 19–23 April 2021; pp. 559–572. [CrossRef]
13. Saraswat, M.; Tripathi, R. Cloud Computing: Comparison and Analysis of Cloud Service Providers-AWs, Microsoft and Google. In Proceedings of the 9th International Conference System Modeling and Advancement in Research Trends (SMART), Moradabad, India, 4–5 December 2020; pp. 281–285. [CrossRef]
14. Jonas, E.; Schleier-Smith, J.; Sreekanti, V.; Tsai, C.C.; Khandelwal, A.; Pu, Q.; Shankar, V.; Carreira, J.; Krauth, K.; Yadwadkar, N.; et al. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *arXiv* **2019**, arXiv:1902.03383.
15. Yussupov, V.; Breitenbücher, U.; Leymann, F.; Müller, C. 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, Auckland, New Zealand, 2–5 December 2019; pp. 273–283. [CrossRef]

16. Kuhlenkamp, J.; Werner, S.; Borges, M.; Tal, K.; Tai, S. An Evaluation of FaaS Platforms as a Foundation for Serverless Big Data Processing. In Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing, Auckland, New Zealand, 2–5 December 2019; pp. 1–9. [CrossRef]
17. Hassan, H.B.; Barakat, S.A.; Sarhan, Q.I. Survey on serverless computing. *J. Cloud Comput.* **2021**, *10*, 39. [CrossRef]
18. Shi, W.; Cao, J.; Zhang, Q.; Li, Y.; Xu, L. Edge Computing: Vision and Challenges. *IEEE Internet of Things J.* **2016**, *3*, 637–646. [CrossRef]
19. Fortier, P.; Le Mouël, F.; Ponge, J. Dyninka: A FaaS Framework for Distributed Dataflow Applications. In Proceedings of the 8th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems, Chicago, IL, USA, 18 October 2021; pp. 2–13. [CrossRef]
20. George, G.; Bakir, F.; Wolski, R.; Krintz, C. NanoLambda: Implementing Functions as a Service at All Resource Scales for the Internet of Things. In Proceedings of the IEEE/ACM Symposium on Edge Computing (SEC), San Jose, CA, USA, 12–14 November 2020; pp. 220–231. [CrossRef]
21. Vendor Lock-In and Cloud Computing. Available online: https://www.cloudflare.com/en-gb/learning/cloud/what-is-vendor-lock-in/ (accessed on 7 November 2021).
22. van Eyk, E.; Iosup, A.; Seif, S.; Thömmes, M. The SPEC cloud group's research vision on FaaS and serverless architectures. In Proceedings of the 2nd International Workshop on Serverless Computing, Las Vegas, NV, USA, 11–15 December 2017; pp. 1–4. [CrossRef]
23. AWS CloudFormation. Available online: https://aws.amazon.com/cloudformation/ (accessed on 17 July 2022).
24. Terraform. Available online: https://www.terraform.io (accessed on 17 July 2022).
25. Serverless Framework. Available online: https://www.serverless.com (accessed on 17 July 2022).
26. Pulumi. Available online: https://www.pulumi.com (accessed on 17 July 2022).
27. Magic Functions in Pulumi. Available online: https://www.pulumi.com/blog/lambdas-as-lambdas-the-magic-of-simple-serverless-functions/#magic-functions (accessed on 12 March 2022).
28. Cloud Framework (Preview). Available online: https://www.pulumi.com/docs/tutorials/cloudfx/ (accessed on 23 September 2022).
29. @pulumi/cloud. Available online: https://www.npmjs.com/package/@pulumi/cloud (accessed on 23 September 2022).
30. OpenFaaS. Available online: https://www.openfaas.com/ (accessed on 5 August 2022).
31. Templates–OpenFaaS. Available online: https://github.com/openfaas/templates (accessed on 6 August 2022).
32. Triggers–OpenFaaS. Available online: https://docs.openfaas.com/reference/triggers/#cloudevents (accessed on 5 August 2022).
33. Spillner, J. Transformation of Python Applications into Function-as-a-Service Deployments. *arXiv* **2017**, arXiv:1705.08169.
34. Spillner, J.; Dorodko, S. Java Code Analysis and Transformation into AWS Lambda Functions. *arXiv* **2017**, arXiv:1702.05510.
35. Yussupov, V.; Breitenbücher, U.; Kaplan, A.; Leymann, F. SEAPORT: Assessing the Portability of Serverless Applications. In Proceedings of the 10th International Conference on Cloud Computing and Services Science, Online, 7–9 May 2020; pp. 456–467. [CrossRef]
36. Language Support Details. Available online: https://docs.microsoft.com/en-us/azure/azure-functions/functions-create-function-app-portal#language-support-details (accessed on 28 December 2021).
37. Maissen, P.; Felber, P.; Kropf, P.; Schiavoni, V. FaaSdom: A Benchmark Suite for Serverless Computing. In Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems, Montreal, QC, Canada, 13–17 July 2020; pp. 73–84. [CrossRef]
38. Dolstra, E.; Bravenboer, M.; Visser, E. Service Configuration Management. In Proceedings of the 12th International Workshop on Software Configuration Management, Lisbon, Portugal, 5–6 September 2005; pp. 83–98. [CrossRef]
39. OAuth Threats–Obtaining Client Secrets. Available online: https://datatracker.ietf.org/doc/html/rfc6819#section-4.1.1 (accessed on 23 July 2022).
40. Token Types. Available online: https://cloud.google.com/docs/authentication/token-types#access (accessed on 21 September 2022).
41. Configurable Token Lifetimes in the Microsoft Identity Platform. Available online: https://learn.microsoft.com/en-us/azure/active-directory/develop/active-directory-configurable-token-lifetimes#access-tokens (accessed on 21 September 2022).
42. Cloud Translation API. Available online: https://cloud.google.com/java/docs/reference/google-cloud-translate/latest/com.google.cloud.translate (accessed on 20 July 2022).
43. OAuth–Ktor. Available online: https://ktor.io/docs/authentication.html#oauth (accessed on 20 July 2022).
44. Ktor–JSON Serializer. Available online: https://ktor.io/docs/serialization-client.html#register_json (accessed on 20 July 2022).
45. Compose for Desktop. Available online: https://www.jetbrains.com/lp/compose-desktop/ (accessed on 19 July 2022).
46. Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*; Addison-Wesley Longman Publishing: Boston, MA, USA, 1995.
47. Riehle, D. Design Pattern Density Defined. *SIGPLAN Not.* **2009**, *44*, 469–480. [CrossRef]
48. Gson Library. Available online: https://github.com/google/gson (accessed on 26 July 2022).
49. Folder Structure of an Azure Functions Java Project. Available online: https://docs.microsoft.com/en-us/azure/azure-functions/functions-reference-java?tabs=bash%2Cconsumption#folder-structure (accessed on 5 January 2022).

50. Structuring Source Code for Java. Available online: https://cloud.google.com/functions/docs/writing#structuring_source_code (accessed on 5 January 2022).
51. Web Apps–Create or Update. Available online: https://docs.microsoft.com/en-us/rest/api/appservice/web-apps/create-or-update (accessed on 28 July 2022).
52. Kudu API. Available online: https://github.com/MicrosoftDocs/azure-docs/blob/main/articles/app-service/deploy-zip.md#kudu-api (accessed on 27 July 2022).
53. Deploying from a Zip File or url–Kudu. Available online: https://github.com/projectkudu/kudu/wiki/Deploying-from-a-zip-file-or-url (accessed on 27 July 2022).
54. Perform Resumable Uploads. Available online: https://cloud.google.com/storage/docs/performing-resumable-uploads (accessed on 28 July 2022).
55. Create Cloud Function. Available online: https://cloud.google.com/functions/docs/reference/rest/v1/projects.locations.functions/create (accessed on 28 July 2022).
56. Cloud Resource Manager API. Available online: https://cloud.google.com/resource-manager/reference/rest (accessed on 26 August 2022).
57. Test Code Using JUnit in JVM. Available online: https://kotlinlang.org/docs/jvm-test-using-junit.html (accessed on 10 June 2022).
58. MetricService. Available online: https://cloud.google.com/monitoring/api/ref_v3/rpc/google.monitoring.v3#metricservice (accessed on 19 June 2022).
59. Cloud Function Metrics. Available online: https://cloud.google.com/monitoring/api/metrics_gcp#gcp-cloudfunctions (accessed on 20 June 2022).
60. Query–Application Insights REST API. Available online: https://docs.microsoft.com/en-us/rest/api/application-insights/query/execute (accessed on 30 June 2022).
61. Supported Metrics with Azure Monitor. Available online: https://docs.microsoft.com/en-us/azure/azure-monitor/essentials/metrics-supported#microsoftwebsites (accessed on 1 July 2022).
62. Consumption Plan Costs. Available online: https://docs.microsoft.com/en-us/azure/azure-functions/functions-consumption-costs?tabs=portal#consumption-plan-costs (accessed on 1 July 2022).
63. Metrics–Application Insights REST API. Available online: https://docs.microsoft.com/en-us/rest/api/application-insights/metrics/get (accessed on 3 July 2022).
64. Digging into Azure Functions: It's Time to Take Them Seriously. Available online: https://www.codemag.com/article/1711071/Digging-into-Azure-Functions-It%E2%80%99s-Time-to-Take-Them-Seriously (accessed on 2 July 2022).
65. Azure Functions Hosting Options–Service Limits. Available online: https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale#service-limits (accessed on 13 June 2022).
66. Cloud Functions–Memory Limits. Available online: https://cloud.google.com/functions/docs/configuring/memory (accessed on 8 July 2022).
67. Azure Functions Hosting Options—Overview of Plans. Available online: https://learn.microsoft.com/en-us/azure/azure-functions/functions-scale#overview-of-plans (accessed on 11 June 2022).
68. Understanding Serverless Cold Start. Available online: https://azure.microsoft.com/en-us/blog/understanding-serverless-cold-start/ (accessed on 11 June 2022).
69. Cold Starts in Cloud Functions. Available online: https://mikhail.io/serverless/coldstarts/gcp/ (accessed on 11 June 2022).
70. Figiela, K.; Gajek, A.; Zima, A.; Obrok, B.; Malawski, M. Performance evaluation of heterogeneous cloud functions. *Concurr. Comput. Pract. Exp.* **2018**, *30*, e4792. [CrossRef]
71. McGrath, G.; Brenner, P.R. Serverless Computing: Design, Implementation, and Performance. In Proceedings of the IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW), Atlanta, GA, USA, 5–8 June 2017; pp. 405–410. [CrossRef]
72. Use Global Variables to Reuse Objects in Future Invocations. Available online: https://cloud.google.com/functions/docs/bestpractices/tips#use_global_variables_to_reuse_objects_in_future_invocations (accessed on 7 July 2022).
73. Cloud Function Resource. Available online: https://cloud.google.com/functions/docs/reference/rest/v1/projects.locations.functions#CloudFunction (accessed on 12 July 2022).
74. Get Resource Changes. Available online: https://learn.microsoft.com/en-us/azure/governance/resource-graph/how-to/get-resource-changes (accessed on 12 July 2022).
75. Manner, J.; Endreß, M.; Heckel, T.; Wirtz, G. Cold Start Influencing Factors in Function as a Service. In Proceedings of the IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), Zurich, Switzerland, 17–20 December 2018; pp. 181–188. [CrossRef]
76. Rodrigues, P.; Freitas, F.; Simão, J. QuickFaaS: Providing Portability and Interoperability between FaaS Platforms. In Proceedings of the 9th European Conference on Service-Oriented And Cloud Computing (ESOCC), Wittenberg, Germany, 22–24 March 2022.
77. QuickFaaS Essentials Repository. Available online: https://github.com/Pexers/quickfaas-essentials (accessed on 18 October 2022).