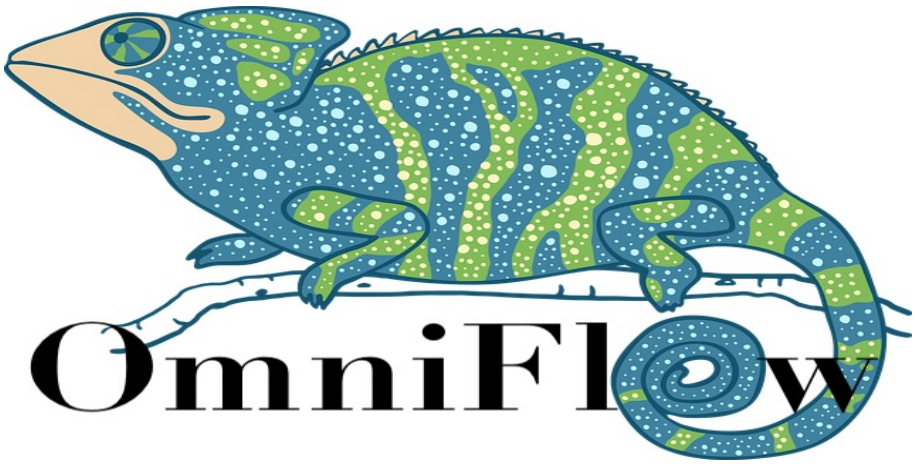
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Departamento de Engenharia de Electrónica e Telecomunicações e de Computadores



Function composition in Function-as-a-Service Platforms

Bernardo José Mateus Costa

Bachelor's degree

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

Orientadores : Prof. Doutor Filipe Bastos de Freitas
Prof. Doutor José Manuel de Campos Lages Garcia Simão

Júri:

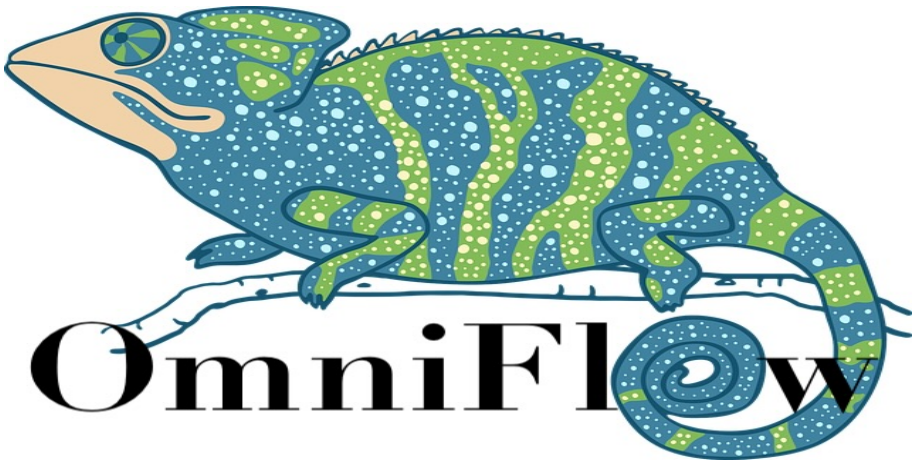
Presidente: Prof. Doutor Nuno Miguel Soares Datia

Vogais: Prof. Doutor José Manuel de Campos Lages Garcia Simão
Prof. Doutor Manuel Martins Barata

October, 2023

INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Departamento de Engenharia de Electrónica e Telecomunicações e de Computadores



Function composition in Function-as-a-Service Platforms

Bernardo José Mateus Costa

Bachelor's degree

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

Orientadores : Prof. Doutor Filipe Bastos de Freitas
Prof. Doutor José Manuel de Campos Lages Garcia Simão

Júri:

Presidente: Prof. Doutor Nuno Miguel Soares Datia

Vogais: Prof. Doutor José Manuel de Campos Lages Garcia Simão
Prof. Doutor Manuel Martins Barata

October, 2023

Aos meus pais pela educação, apoio, incentivo e inspiração durante todas as etapas, boas e menos boas, à minha irmã por ser a referência que é, à minha avó pelo carinho, ao meu cunhado pela boa disposição, e ao meus amigos por todos os momentos inesqueciveis que passámos juntos.

Acknowledgments

I would like to express my deepest gratitude to all those who played a fundamental role in the realisation of this project and my growth as a student and as an individual.

Firstly, to my supervisors Professor Filipe Freitas and Professor José Simão, I would like to thank you for your guidance and availability. Your support and vision were keys for the success of this work and for my academic development. The availability and trust you have given me daily have kept me motivated, giving me the pleasure of carrying out this project.

I would like to thank the Instituto Superior de Engenharia de Lisboa (ISEL) for all the high-quality education and learning opportunities it has given me. The solid foundations I acquired at this institute were fundamental to my formation and professional growth.

To my friends, who have always been by my side, sharing moments of joy, challenges and discoveries. Your words of encouragement and our camaraderie have made this academic journey lighter and more memorable.

To my work colleagues, who not only shared my daily life but also motivated and supported me throughout this journey, my most sincere thanks. The inspiring conversations and collaborative working environment were crucial to my motivation.

And last but definitely not least, to my family. Thank you for all the love, unconditional support and lessons you have taught me throughout my life, which has made me the person I am today. Your constant encouragement and for always being by my side has been the driving force behind my endeavours and achievements.

You have all played a unique and invaluable role in my academic and personal fulfilment. I am deeply grateful for all the moments shared, lessons learnt and support. This work is dedicated to all of you and is the result of our collaboration and unity.

Abstract

Serverless computing is a paradigm that allows developers to focus on writing code without the need to manage the underlying infrastructure. Code is executed on-demand, automatically scaled, and billed during its execution time. Function-as-a-Service and cloud function composition, also known as cloud workflows, are among the most popular serverless backend services, but they often require developers to meet cloud-specific requirements. This can lead to vendor lock-in issues in workflow applications due to custom workflow specifications and deployment requirements.

In this work, we created OmniFlow, a programming library that targets function composition and deployment in different cloud providers. The proposed library aims to improve developer's productivity, flexibility, and agility when creating serverless solutions, for multiple cloud providers, without requiring the installation of additional software. Our approach enables developers to reuse their serverless workflows in different cloud providers without needing to rewrite them. This library was developed in Kotlin and evaluated using two major cloud providers, Amazon and Google. We found some limitations but our solution can translate the workflow definition with a small overhead.

Keywords: cloud computing; serverless computing, Function-as-a-Service; vendor lock-in; function composition; cloud orchestration; workflow.

Resumo

A computação *serverless* é um paradigma que permite aos programadores um maior foco no desenvolvimento de código, sem a necessidade de gerir a infra-estrutura subjacente. O código é executado a pedido, escala automaticamente, e apenas é cobrado pelo tempo de execução. Composição de funções também conhecido como fluxos de trabalho, e Function-as-a-Service são serviços populares no contexto de serviços de *backend serverless*. No entanto, muitas vezes exigem que os programadores conheçam os seus requesitios específicos. Isto pode levar a uma elevada dependência do fornecedor, em aplicações com processos de trabalho, devido a especificações dos fluxos de trabalho e requesitos de *deployment*.

Neste trabalho, criamos uma biblioteca de programação designada Omniflow, que visa fazer *deploy* de composição de funções em diferentes fornecedores da nuvem. A biblioteca proposta tem como objetivo melhorar a produtividade, flexibilidade, e agilidade do desenvolvedor na criação de soluções *serverless*, em múltiplas plataformas da nuvem, sem a necessidade de instalar *software*. Para além disso, permite uma maior reutilização de processos de trabalho *serverless*, sem a necessidade de reescrevê-los, em diferentes fornecedores da nuvem. O Omniflow foi desenvolvido em Kotlin e aplicado em duas grandes plataformas, Amazon e Google. Encontrá-mos algumas limitações, mas a nossa solução consegue traduzir fluxos de trabalho de forma eficiente.

Palavras-chave: computação na nuvem, computação *serverless*, Function-as-a-Service, dependência de fornecedores, composição de funções, orquestração na nuvem, fluxos de trabalho

Contents

List of Figures

List of Tables

List of Listings

Acronyms

AI Artificial Intelligence. 11

API Application Programming Interface. 6, 10, 11, 12, 16, 18, 19, 21, 27, 29, 45

AWS Amazon Web Services. 6, 7, 10, 15, 16, 19, 23, 36, 38, 54

BaaS Backend-as-a-Service. 19

CPU Central Processing Unit. 6

DSL Domain Specific Language. 3, 6, 19, 20, 23, 24, 25, 28, 29, 30, 31, 35, 37, 42, 43, 45, 46, 53, 54

EaaS Evaluation-as-a-Service. 19

FaaS Function-as-a-Service. 2, 3, 5, 6, 7, 10, 11, 17, 19, 21, 24, 25, 26, 53

GCP Google Cloud Platform. 6, 10, 11, 15, 21, 23, 38, 47, 54

HTTP Hypertext Transfer Protocol. 5, 8, 13, 14, 15, 16, 26, 28, 29, 32, 41, 42, 46, 47, 50

IaaS Infrastructure-as-a-Service. 10

IDE Integrated Development Environment. 11

JMH Java Microbenchmark Harness. 45

JSON JavaScript Object Notation. 16, 17, 20, 35, 50, i

JVM Java Virtual Machine. 45

PaaS Platform-as-a-Service. 10

REST Representational State Transfer. 16

SDK Software Development Kit. 19, 20, 21, 30, 37, 42, 54

TCK Technology Compatibility Kit. 20

URL Uniform Resource Locator. 14, 15, 16, 21, 50

VM Virtual Machine. 19

YAML YAML Ain't Markup Language. 12, 20, 50

Glossary

CI/CD A set of practices that involve automatically testing and deploying code changes frequently and consistently. 55

deployment Includes all the steps, processes and activities required to create or update a software system and make it available to the user. 2, 3, 17, 18, 19, 20, 23, 24, 25, 26, 29, 30, 35, 36, 37, 38, 39, 41, 42, 43, 45, 46, 47, 49, 51, 53, 54, 55

function composition The act of bringing together multiple functions to accomplish a more complex goal. 3, 5, 17, 22, 23, 24, 26, 38, 39, 53

microservices An architectural and organizational approach to software development where software is composed of small independent services that communicate over well-defined contracts. 1, 21, 27

orchestration The coordination and management of multiple functions or services to complete a workflow or business process. 6, 14, 15, 18, 20, 21

RESTful Corresponds to the use of REST principles to develop solutions or services. 16, 18

serverful Architectural model for building software services with complete control of the infrastructure, where servers have to be maintained. 1

stateless Function or component that does not retain status from previous interactions. 5, 6

1

Introduction

The serverless concept, also referred to as serverless computing, is gaining more popularity as it shows promise. This approach is a way of deploying applications to the cloud, which has seen a uptake because companies have recently made a shift to infrastructures that use containers and microservices [3]. The cloud computing execution models traditionally used are serverful, where infrastructure maintenance is involved, which is, in contrast to serverless computing, where the infrastructure is abstracted away from the developers [31]. From a cloud provider's point of view, serverless computing offers an opportunity for developers to concentrate solely on code development, without the need to set up infrastructure resources. Therefore serverless computing reduces the level of effort required to create and manage applications in the cloud, lowering operational costs [3].

Separation of responsibilities is recommended when it comes to code development, instead of having just one function with all the logic, it should be separated into multiple functions with different behaviours to achieve the end goal. This way, functions can be reused and are smaller and easier to understand. With separation, functions are executed in a chain, where the result of invoking the first function is the input for the second function, and so on [4]. This concept, where one function aggregates the behaviour of multiple functions with responsibilities and objectives, is called a workflow. During industrialisation, the concept of workflow gained affluence with manufacturing and office processes, due to the fact that they could be separated into tasks, roles, rules and procedures. This concept originated from a study about how to improve the

efficiency of tasks that are repeated on a daily basis. Initially, when processes began to be used, they were carried out by humans using physical objects, tools, paper and more. With the advent of technology, it is possible to automate these processes and improve their efficiency in order to achieve better and faster delivery in a work environment. A common example is humans using computers, with programmes or tools to carry out their work [12]. Therefore, with the rapid evolution of technology, it is easier to identify patterns and cyclical work in various areas, such as forecasting, health, science, management, call centres, and more.

Cloud platforms allow workflows to be run in the cloud, using the serverless paradigm, where users can automate complex tasks. A cloud workflow is made up of multiple cloud functions, each designed to carry out specific tasks when triggered. It should be noted that the deployment of well-structured workflows can achieve a reduction in costs, time and control over processes.

The serverless computing paradigm is centred on cloud providers that manage the infrastructure, where developers can deploy event-driven code. Solutions such as Function-as-a-Service (FaaS) and Workflows are within the serverless domain and are provided and managed by cloud platforms. However, the benefit of having no control over the infrastructure comes at the cost of high platform coupling. The platforms have different perspectives on the problem and end up providing specific and different solutions to the same problem. As cloud providers have their own languages and ways of using their services, the risk of vendor lock-in increases. In addition, it is possible to combine multiple services within the platform, such as workflows with FaaS, which emphasises this problem [39]. An inhibiting factor for the adoption of serverless solutions in practice is the difficulty of estimating the expected costs of serverless functions and workflows [8].

In the software industry when applications take advantage of cloud solutions for development, there may be cases where it is necessary to migrate them to a different cloud provider by factors such as pricing, termination of a contract, or other circumstances. Typically, cloud platforms supply tailored solutions that include specific services, configurations, schemas, and other proprietary components that are coupled to the provider. Such particularities engender complications and makes the transition arduous for developers, as they are obligated to familiarise themselves with the new requirements and models of the alternative provider. Cloud workflows are not an exception, from the issues associated with provider migration. In the event of transitioning to an alternative cloud provider, the user must undertake the task of rewriting the entire workflow to follow to the standards imposed by the new provider, resulting in operational disruptions and additional costs. This is predominantly attributed to the

distinct schemas and requirements that are exclusive to each cloud provider, giving rise to a non-portable solution that is widely recognised as the lock-in problem.

1.1 Contributions

The main contribution was to build a solution to support the developers in defining and deploying function based workflows to different FaaS platforms. A model was established and integrated into a library, using a new Domain Specific Language (DSL), allowing users to define and deploy function composition on a range of cloud platforms. The library translates the custom DSL to the specific language of each provider. This approach promotes flexibility and enable users to take full advantage of the benefits of multi-cloud deployments.

1.2 Document outline

This document consists of five chapters. The first chapter introduces and contextualises the topic, outlining the problem posed by the serverless paradigm and how it was approached throughout the work. Chapter two provides an overview of the current state of serverless computing, and demonstrates how to develop and deploy functions and workflows on multiple platforms. In addition, are described several tools and related works that focus on solving the vendor lock-in problem. Chapter three provides the approach to the problem and the entire development of the solution. It also discusses the set of challenges that arose during implementation, and what structure was used at both software and DSL level. The fourth chapter refers to the evaluation of the library, identifying the metrics to be considered and the results obtained. Finally, the fifth chapter contains all the conclusions drawn during the development of the work, future work and objectives to be considered.

2

Background and Related Work

This chapter presents how FaaS platforms work, and how the problem of vendor lock-in is highlighted. In addition, the execution patterns most frequently supported by the platforms are mentioned, as well as examples and demonstrations of function and function composition in three distinct cloud providers, Amazon, Azure and Google. It concludes with related work, such as platforms, frameworks and tools used in a multi-cloud environment.

2.1 Background

The main purpose behind the concept of serverless is to abstract away all the server configurations from software developers, which means, that the actual servers remain present in the scope. However, all the infrastructure requirements regarding server management, such as provisioning, scaling, and maintenance, are in charge of the cloud providers, and the developers do not need to attend to this duty. The FaaS follows a serverless architecture therefore the applications are triggered via specific events, e.g., database update operations or Hypertext Transfer Protocol (HTTP) requests; and are composed of multiple, short-lived, and, typically, stateless functions. With that in mind, the cloud providers embrace an optimised cost model, where the charges only apply when the functions have been executed, instead of, also paying for idle times [39].

FaaS platforms perform deploys, monitor, and manage the cloud functions in a way that is not perceptible to the tenants, and release them from operational concerns such as resource auto-scaling, traffic routing, and log aggregation. In any case, the FaaS user still has some responsibilities in the operational part, because the user can specify parameters and configurations, such as suggested memory size or number of Central Processing Units (CPUs) of the underlying function host, which influence the operation of the deployed cloud function [34].

On the other hand, FaaS brings disadvantages, such as the so-called start problem or limited function execution time. The restriction time for tasks to run, is a major drawback, primarily for more complex use cases. The partition of a large function into a chain of smaller functions, respecting the same workflow, can be a possible solution. Function orchestrators were designed for that purpose, to execute workflows based on multiple FaaS-hosted functions, e.g., Microsoft Azure, offered by cloud providers that each one has its own feature set. The majority of Function orchestrators are structured to support more complex workflows like chaining, branching, and parallel execution of functions, to facilitate the modelling of long execution tasks and make FaaS more reliable [39]. A big challenge that FaaS brings is the Lock-in problem where the users are locked into the requirements and features of a specific provider. Bringing complexity in multiple different migration contexts, for example, the need to migrate one cloud application from one provider to another. Every cloud provider has its requirements, features, and configurations to be filled in to start using it, including data formats, remote Application Programming Interfaces (APIs), or even custom DSLs and programming languages. Choosing which provider should be used is not easy, because the users become tightly coupled with the corresponding service, and this makes it significantly harder costs and efforts-wise to switch cloud providers, e.g., organisational restructuring or cost optimisation. Furthermore, cloud applications are typically not built with a general structure, and by choosing a specific cloud provider, the applications become restricted to the features of the chosen provider [39].

However, in some real-world scenarios sometimes is required complex serverless applications that have multiple functions that need to communicate with each other, and the cloud functions platforms are not aware of the data dependency between those functions. This dependency may lead to poor performance of communication. For this sub-optimal functional relationship, serverless providers come up with new orchestration frameworks to compose workflows, that consists of a pool of cloud functions [5]. FaaS platforms, such as Amazon Web Services (AWS) Lambdas, Google Cloud Platform (GCP) Cloud Functions, or Azure Functions enable the serverless execution of stateless, ephemeral compute functions [27]. In order to implement intricate business

processes, individual functions can be combined to form serverless workflows through the utilisation of platforms including AWS Step Functions, Google Cloud Workflow, or Azure Durable Functions [36]. At present, leading cloud providers utilise a uniform cost model for serverless functions, whereby the expense of a function execution is contingent upon three factors: the response time of a function, which is approximately 100 milliseconds; the memory allocated to the function; and a fixed fee for each invocation [1]. Although numerous organisations have reported substantial cost reductions through the transition from traditional hosting options to serverless solutions, the practical adoption of serverless solutions is impeded by the challenge of accurately predicting the costs of serverless functions and workflows [1, 35]. As a result, the expense and response time of functions encompassed within a workflow may be unpredictable, creating difficulty in accurately forecasting the cost of the overall serverless workflow.

Researchers from diverse fields, including high-energy physics and astronomy, are creating extensive applications in the shape of workflows that contain numerous jobs with precedence constraints. Scientific workflows frequently evolve into sophisticated systems characterised by a high volume of jobs, a significant quantity and magnitude of input and output data, and precedence constraints between the various jobs [16]. The objective of serverless cloud workflows is to simplify the construction of applications that need to pass the output of one function as the input to other. Thus, functions with different purposes, are inherently event-driven and independent and execute in a specific order to produce more complex workflows [5]. A most popular and straightforward use case is an image editing application that applies a set of techniques to the image involving a chain of FaaS functions to re-size, greyscale, and add watermark to the input image, see Figure 2.1.

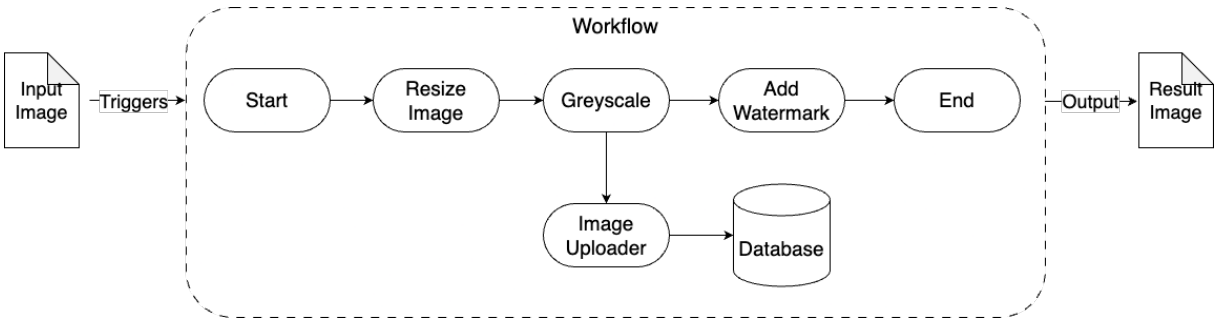


Figure 2.1: Image processing workflow

2.2 Patterns

There are multiple practical patterns for each provider; however, the simplest ones are supported by most of the popular providers. More complex and specific patterns may only be supported on some platforms. The first pattern is **function chaining**, where a sequence of HTTP requests triggers cloud functions in a specific order, and the output of one function is used as the input of the next. Another design pattern is **concurrent execution**, where a branch or a loop can be defined to be executed concurrently. This pattern is intended to assist with concurrent long-running operations and reduce the total execution time. The execution is completed when all concurrent tasks have finished successfully or by an exception. The third execution flow is the **iteration**, used to iterate over a sequence of numbers or a collection of data, that can walk through every item in the collection and process it. Finally, the **conditional** pattern is a selection mechanism that enables the value of an expression to determine the control flow of the workflow's execution. If the condition is met, the statement is executed, otherwise, another execution flow is followed. These are the fundamental patterns and execution flows that may be supported in this project. It is possible to observe the patterns, function chaining, concurrent execution, iteration, and condition in the Figures 2.2, 2.3, 2.4, and 2.5 adapted from [39].

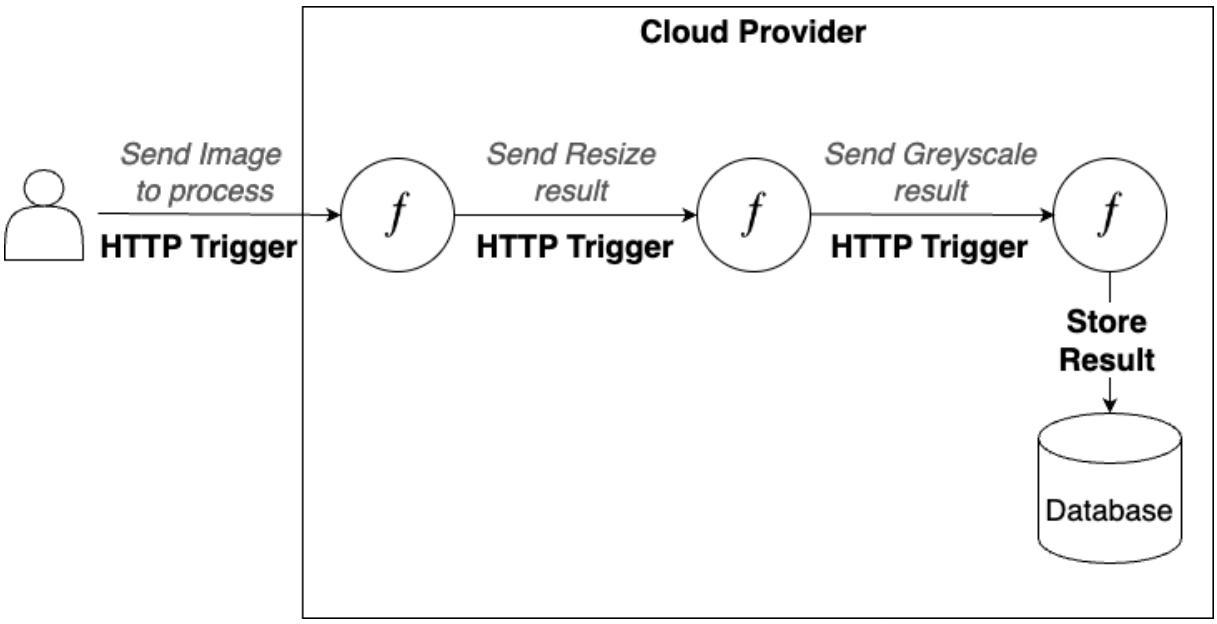


Figure 2.2: Function chaining

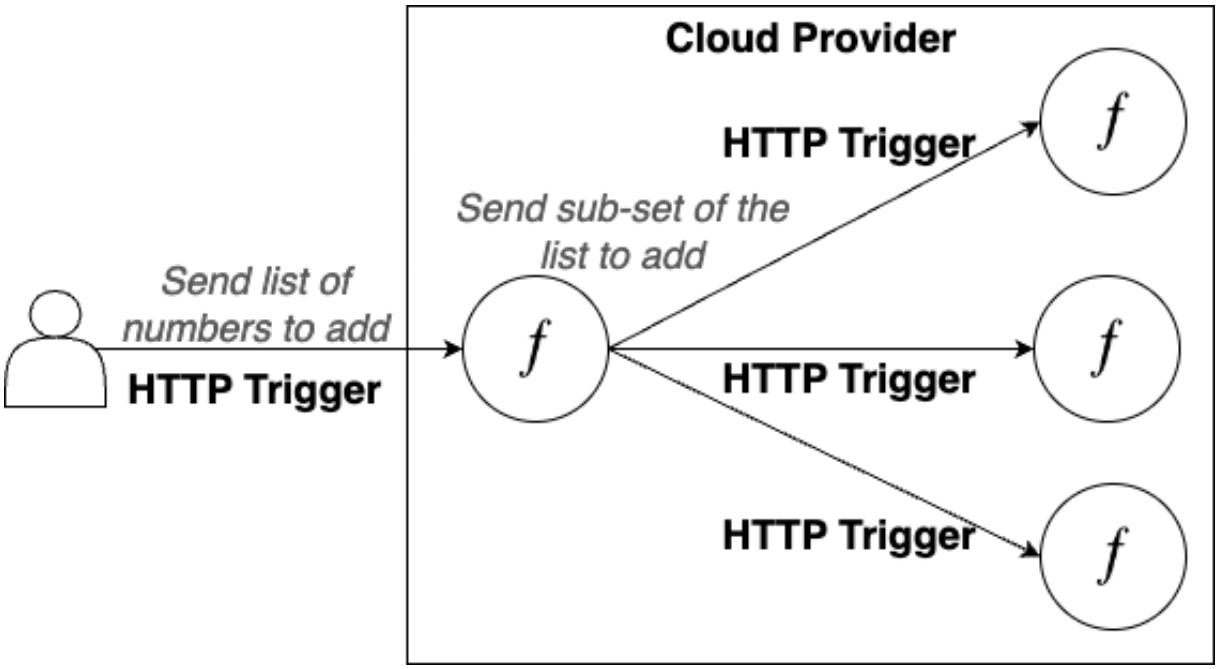


Figure 2.3: Concurrent execution

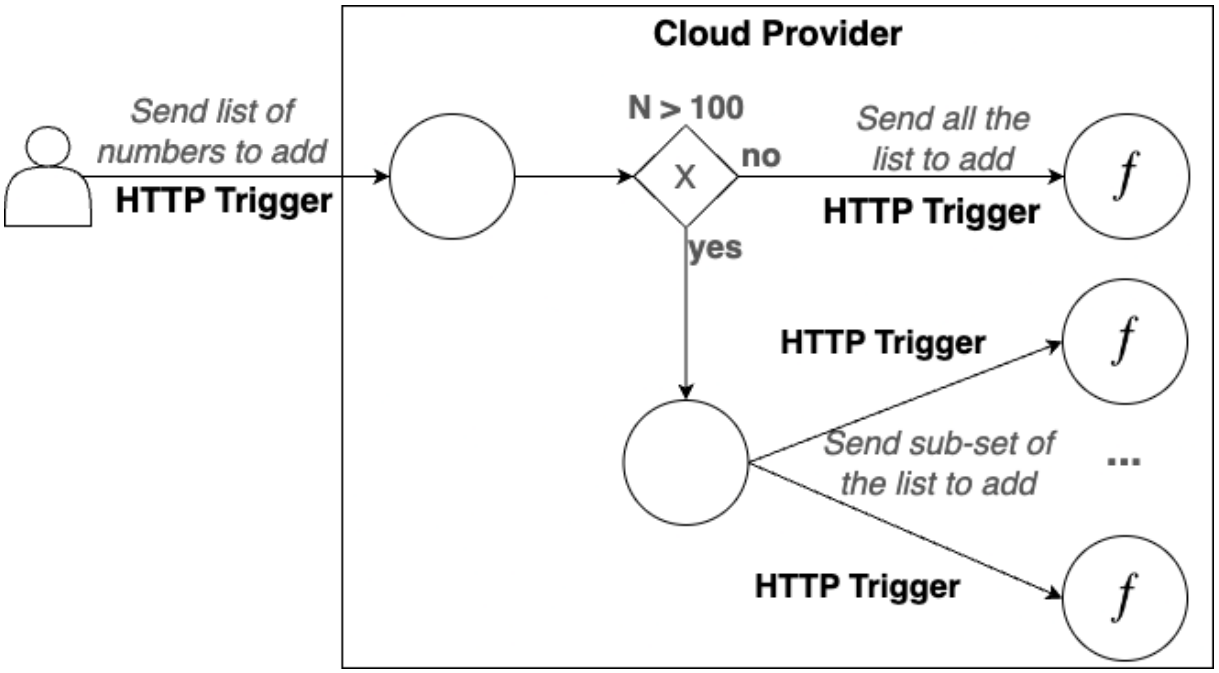


Figure 2.4: Conditional

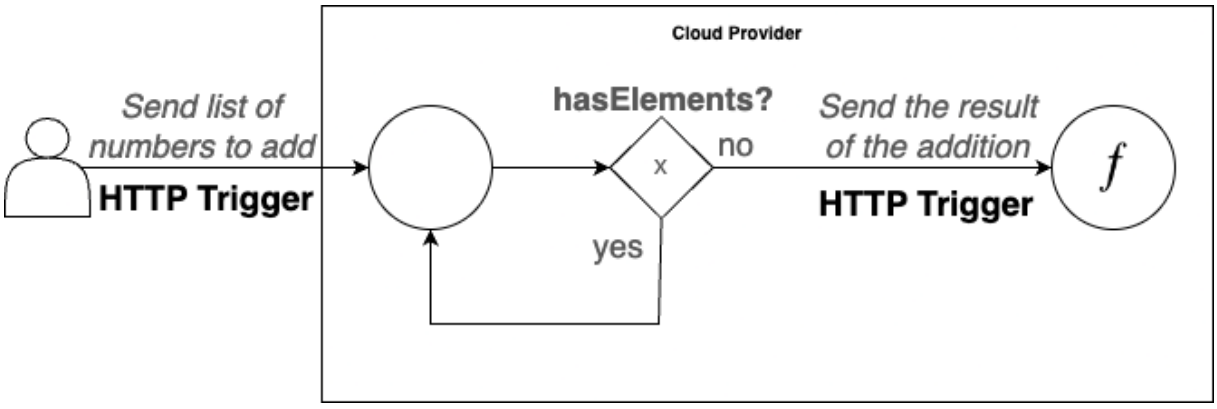


Figure 2.5: Iteration

2.3 Function-as-a-Service Providers

Cloud computing has been gaining popularity in the last few years because compared to the traditional computing model, it offers unprecedented advantages regarding reliability and costs. Cloud customers pass the responsibility of their computation and storage to public cloud providers, and pay for the service usage on demand, where usually the client needs to pay large upfront costs, for example, hardware provisioning, before using the actual services. The cloud charging model pay-as-you-go allows the user to pay for what was used and promises that when the resources are not enough the infrastructure is scaled automatically by demand. This way the users can avoid extra costs maintaining people to manage the servers and network infrastructure [23].

There are a growing number of cloud providers, such as GCP, Microsoft Azure, AWS, and more, that offer a variety of pricing options, performance, features, and requirements. Some providers offer multiple kinds of services via APIs, for instance, Platform-as-a-Service (PaaS), to build applications from scratch; Infrastructure-as-a-Service (IaaS), where a customer runs applications inside IaaS. However, in this context, we are taking into consideration only the FaaS perspective. Each cloud provider practices its benefits and ways of doing things [23].

Performed a simple exercise in the next three providers: Google Cloud Platform, Microsoft Azure, and Amazon Web Services, which consists in building a single cloud function to increment a number present in the query parameter. After that, a cloud orchestrator was created to increment the input number by 3, using the previously created function in the same provider. All the orchestrators follow the function chaining use case, where is passed an input number to the first function, the function processes, and the result (output) is passed as the query parameter (input) to the next function

and so on, occurring three times in total. The third and last call of the function returns the initial number incremented by 3. In the following subsections 2.3.1, 2.3.2, and 2.3.3 it is explained how the experience was, the pricing model, and the requirements for each provider. The objective of this demonstration was to go through some known cloud providers, to obtain some experience regarding this topic with a simple use case. So, it does not take into consideration the code exceptions, exception handling, and more complex solutions, not being ready for the enterprise production environment.

2.3.1 Google Cloud Platform

In the world of cloud platforms, GCP is one of the best known, as it has a wide range of solutions available, such as cloud computing, databases, security, Artificial Intelligence (AI) and more. In terms of serverless services, this platform provides solutions such as Cloud Functions [10] and Workflows [11], where it is possible to design and run code in the cloud. Google exposes its cloud services through APIs, where all the services integrate well with each other, making it possible to use several solutions in a single project. It also offers incentives for new users, to attract more consumers to the platform.

After following pretty straightforward documentation it was created a Google Cloud project for that purpose with the billing account, Cloud Functions, Cloud Build, Artifact Registry, Cloud Run, Logging, and Pub/Sub APIs enabled. As soon as the project setup is done, it is possible to create Cloud Functions, using the Google Cloud Console. Next, some configurations were selected such as the environment, second generation; function name; and authentication type, which in this case allows unauthenticated invocations. Finally is only missing the actual code to run in the specified runtime programming language, via an inline editor in the GCP present on the website. The selected runtime was Java 17, which is a common programming language used nowadays, and an online Integrated Development Environment (IDE) to create the function behaviour. The user experience was pretty clean so far and did not force the user to install any environment or requirement on his local machine. However, the FaaS client needs to have skills with any programming language available like *Java, Python, Node.js*, etc. The produced code can be seen in the Listing 2.1 and is ready to be deployed. The objective of this function, as indicated previously, is to return a number incremented by one, present in a query string called "number", if the parameter is not present the default value is 0.

```
1 public class Example implements HttpFunction {
2
```

```
3       @Override
4       public void service(HttpRequest request, HttpResponse response)
    throws Exception {
5           String toIncrementQuery = request.getFirstQueryParameter("
    number").orElse("0");
6           int nrIncremented = Integer.parseInt(toIncrementQuery) + 1;
7           var writer = new PrintWriter(response.getWriter());
8           writer.printf("%d", nrIncremented);
9       }
10  }
```

Listing 2.1: Gloogle Cloud Platform cloud function

The next step is to create, deploy and execute a workflow. In general, the workflow setup was pretty simple, using the Google Cloud Console, which started by activating services such as Workflows API for the project and the right roles for logging and accessing the project. As soon as the setup is done, it is possible to create an actual workflow, by defining the workflow name, the region, and the service account. After that, show a workflow editor to the user to write his workflow with the desired behaviour. The language used by Google is YAML Ain't Markup Language (YAML), to define the steps to be executed with specific keywords for the use cases, as you can see the Listing 2.2 below.

```
1  main:
2    params: [input]
3    steps:
4    - increment1:
5        call: http.get
6        args:
7          url: us-central1-func-test-36651.cloudfunctions.net/function1
8          query:
9            increment: ${input.number}
10       result: result1
11   - increment2:
12       call: http.get
13       args:
14         url: us-central1-func-test-36651.cloudfunctions.net/function1
15         query:
16           increment: ${result1.body}
17       result: result2
18   - increment3:
```

```
19        call: http.get
20        args:
21          url: us-central1-func-test-36651.cloudfunctions.net/function1
22          query:
23            increment: ${result2.body}
24        result: result3
25    - returnOutput:
26        return: ${result3.body}
```

Listing 2.2: Google Cloud Platform workflow

2.3.2 Microsoft Azure

The Microsoft Azure platform [38] offers serverless solutions such as Azure Functions and Durable Functions, which fulfil the same purpose as Google's Cloud Functions and Workflows, respectively. The serverless solutions provided by this platform have an event-driven structure, where the various types of functions can be run using triggers. Azure Functions are small pieces of code with logic that run when an event is triggered, for example, an HTTP request. On the other hand, Durable Functions orchestrate and compose multiple functions, and can be seen as functions that can aggregate several functions. With Durable Functions, it is possible to create more complex workflows by defining a trigger and the behaviour of the Durable Function. In this case, the Durable Function's behaviour is to call a simple function that increments a value three times. This platform also has several facilitates to attract new users, mainly with monetary aid [7].

To create the small project it was used as a basis for a simple quick start from Azure, which indicates the initial prerequisites to create a project and test it locally, and if everything works as expected it is ready to be deployed to the cloud. First of all, it was created an educational account with an Azure Subscription, after that, it was installed all the required tools to set up the local environment properly as, installing the editor Visual Studio Code with the Azure Functions extensions, following the installation of the Azure Functions Core Tools and the runtime environment to run the code, it was used JavaScript for this project with the durable functions package. So with all the requirements set it was created a parent project with the Azure Functions version 4, because the most basic Durable Functions application is composed of three main functions: orchestrator function, to describe a workflow that orchestrates other functions; activity function, called by the orchestrator function, performs work, and optionally

returns a value; and the client function, a regular Azure Function that starts an orchestrator function, where in this example uses an HTTP triggered function.

With this in mind, it was created a child project (module) for the orchestrator function, using a template, to manage and coordinate the order of the activity functions. As you can see below in Listing 2.3 each call to **context.df.callActivity**, invokes an activity function named Increment to increment the input number by one.

```
const df = require("durable-functions");

module.exports = df.orchestrator(function* (context) {
    var number = parseInt(context.df.getInput());
    number = yield context.df.callActivity("Increment", number);
    number = yield context.df.callActivity("Increment", number);
    number = yield context.df.callActivity("Increment", number);
    return number;
});
```

Listing 2.3: Azure orchestractor function

At this point, is required to create the activity function, where the action is performed, such as making a database call or performing a computation, however in this context is to increment the number and return it. It created a new sub-project inside the parent project, using the Visual Studio Code Extension using a simple template, see Listing 2.4. The behaviour is pretty simple, only gets from the input context the number, increment, and return it.

```
module.exports = async function (context) {
    var number = context.bindings.number
    return number + 1;
};
```

Listing 2.4: Azure activity function

Finally, is missing the client function, also called HTTP starter, which corresponds to the HTTP-triggered function that starts an orchestration, by calling **client.startNew**. Then it uses **client.createCheckStatusResponse** to return an HTTP response containing Uniform Resource Locators (URLs) that can be used to monitor and manage the new orchestration.

```
const df = require("durable-functions");

module.exports = async function (context, req) {
```

```
    const client = df.getClient(context);

    const defaultNumber = 0
    const instanceId = await client.startNew(req.params.functionName,
    undefined, req.query.number || defaultNumber);

    context.log(`Started orchestration with ID = '${instanceId}'.`);

    return client.createCheckStatusResponse(context.bindingData.req,
   instanceId);
};
```

Listing 2.5: Azure client function

Now the Durable Function application is ready to run locally and be deployed to Azure. When running locally the application prints out the URL endpoint that is listening to start the orchestration, and with another tool like *Postman* or *cURL*, is sent a HTTP POST request. The response is the initial result from the HTTP function shows the durable orchestration has started successfully. The response includes a few useful URLs, where one of them is the **statusQueryGetUri** that indicates the status of the orchestration instance, which shows the instance has completed, and shows the output of the durable function. Now it is time to publish the project to Azure, it is only required to sign in with the Azure account in the visual studio code and deploy it via the extension, and all the required resources will be created automatically.

2.3.3 Amazon Web Services

Like Azure or GCP, AWS is also popular for its wide range of serverless services, which abstract the infrastructure component from the user, and automatically scale resources when there is a heavy load and a price-on-demand model. The two solutions relevant to this context are AWS Lambda and Step Functions, which follow the same purpose as the solutions presented for GCP and Azure. AWS Lambda are functions that run code and are triggered by an event. Step Functions are responsible for orchestrating various tasks, enabling long-running workflows. A Step Function can be interpreted as a state machine with several states, where an example of a state is a call to a Lambda function. It's worth noting that serverless services integrate, highlighting the problem of vendor lock-in.

The First step to create a Lambda function from scratch is to select the desired name for the function, the runtime language to run the function, and the architecture (x86_64

or arm64), it is also possible to select other advanced configurations like the required execution role to run the function, enable code signing, tags, *vpc* and more, however for this case it will not be taken in consideration. The function can be created from scratch, using a blueprint template, or from a container image. After the first setup, the function resource is created and allocated, only missing the behaviour, and is when a page with an editor is open to writing the function execution steps. In the Listing 2.6 it is possible to notice the code used to execute to increment the input number present in the parameter. However, the function does not have any trigger configured, and we are required to add an API to the Lambda function to create an HTTP endpoint that invokes it. API Gateway supports two types of RESTful APIs: HTTP APIs and Representational State Transfer (REST) APIs. To expose the function was defined a new endpoint, using the AWS API Gateway service, without any security mechanism, which means, that the endpoint is open and public for everyone. At this moment, the function can be triggered via an HTTP request for a publicly generated URL. A public endpoint without any authentication is not ideal for production environments, however for this demonstration purpose is not mandatory.

```
1  export const handler = async(event) => {
2      var number = 0
3      if (event.queryStringParameters && event.queryStringParameters.
       increment) {
4          number = Number(event.queryStringParameters.increment)
5      }
6      number = number + 1
7      const response = {
8          statusCode: 200,
9          body: JSON.stringify(number),
10     };
11     return response;
12 };
```

Listing 2.6: Amazon Web Services lambda function

Finally, creating a workflow that calls the created function three times is required to create a Step Function. AWS supports built-in controls, that help examine the state of each step in the workflow to make sure that the application runs in order and as expected. Depending on the use case, it is possible to have Step Functions calling AWS Services, such as Lambda, to perform tasks. Regarding the creation of the State Machine, is shown a graphical console, to analyse the application's workflow as a series of event-driven steps, which will compose a JavaScript Object Notation (JSON) file

with all the required steps, or it is possible to build the JSON file. For this case it was used a bit of both worlds, to create the goal, and the result can be seen in Appendix A, Listing A.1.

2.4 Related Work

In this section, we present the related work on function composition in FaaS platforms, in order to, understand the context and how some architectures, tools, applications and frameworks have emerged in this field.

2.4.1 FaaSFlow

FaaSFlow is a work-side workflow schedule pattern for serverless workflow execution, that aims to enable efficient workflow execution compared to the traditional master-worker-based workflow architecture. The traditional master-side workflow schedule pattern establishes that the master node triggers functions and assigned them to worker nodes to be executed [24]. This approach can bring some heavy scheduling overhead because usually the function execution states are transferred from the master node to the worker nodes. Also, functions may rely on database storage services for data storage and delivery, resulting in a significant data movement overhead [17, 18, 25]. To decrease the scheduling overhead FaaSFlow brings in a new pattern called the "worker-side workflow schedule pattern" (workerSP), where the master node is only responsible for workflow graph partition. Each worker node contains a per-worker workflow engine that knows how to evaluate if a function should be triggered or not, depending on the execution state of the previous functions executed by other nodes. Furthermore, it is proposed a FaaStore, where the goal is to alleviate the overhead of the remote data movement, using a mechanism of adaptive storage, where functions in the same worker may communicate directly through the shared main memory, instead of the remote storage [24]. Were made some framework experimentation's, and the obtained results show that was a significant reduction in the time spent scheduling and transmitting, primarily with scientific workflows and real-world applications.

FaaSFlow and the present work are both tools within the realm of workflow management; however, there exist certain differences between them. FaaSFlow places emphasis on a novel design pattern that aims to distribute workflow jobs in a more optimal manner in order to streamline data flow. This pattern can be utilised by cloud providers for dispatching tasks to worker nodes during the execution of a workflow. Conversely, our framework focuses on the deployment of workflows across multiple cloud

providers, rather than on the runtime of the workflow itself. This pertains to a preliminary phase of the workflow management process and does not involve the runtime, but rather focuses on facilitating deployment.

2.4.2 Triggerflow

Bringing applications to the cloud is getting more and more popular, and there is where Triggerflow comes in, to enable the control of the life-cycle of those applications in a reactive and extensible way, by optimising the execution of tasks in reaction to events [26]. The purpose of this trigger-based orchestration of serverless workflows is to be the extensible, universal, and generic tool to support a high volume of event processing workloads with an auto-scale on-demand mechanism.

Triggerflow is composed of a Front-end RESTful API that a user can interact with, by defining workflows, event sources, event triggers, and to get the context of an executing workflow, using persistent storage. Is also composed of a trigger service that follows and event condition action architecture, based on the persisted data, triggers define which actions must be executed in response to events or event conditions. The framework clients can define the orchestration model, such as, Workflow, which represents a finite state machine with inputs, context variables, and states, like trigger, to launch the appropriate action corresponding to the next state. The mapping of workflows with triggers, represents the connection between a workflow and a set of triggers, containing all state transitions; substitution principle: a workflow must follow the specified action from start to finish determined by its initiation and termination event; dynamic trigger interception, a trigger can be intercepted dynamically and transparently to execute an action. This framework has two implementations, one using *KNative* with a push-based transfer system, and another one using *Kubernetes* for event-driven, following a pull-based mechanism. The major limitations of an event-based orchestration system are the required developer experience and the debugging capability.

In contrast to our library, which enables the deployment of workflows across multiple cloud providers, TriggerFlow addresses the creation and persistence of workflows, and the runtime workflow management. The framework emphasises the creation of a generic model for mapping workflows to corresponding triggers and actions, as well as managing workflows at runtime and supporting auto-scaling as necessary. This renders the framework similar to a cloud provider, although it operates in a more decoupled and generic manner.

2.4.3 Serverless Multicloud

The Serverless Multicloud is a redesign of the Serverless Framework, where the system design consists of three primary components: firstly, the deployment of the workload within the multi-cloud architecture; secondly, the multi-cloud library, which facilitates the utilisation of FaaS and Backend-as-a-Service (BaaS) offerings from diverse vendors in a streamlined manner; and lastly, the End Analysis System, which assesses the performance and cost of the FaaS providers.

The adoption of workload portability across multiple clouds faces a hurdle at the API level, where developers may encounter the need to familiarise themselves with the new Software Development Kits (SDKs) or APIs of different FaaS and/or BaaS offerings from various cloud providers. To overcome this obstacle, the framework offers a common interface that operates as an abstraction layer over the proprietary APIs or SDKs of cloud providers. Facilitating the deployment of multi-cloud workloads by allowing developers to access the cloud services supported by the library via a common interface. Furthermore, was created provider-like SDKs that are built on top of the proprietary SDKs and APIs for direct conversion. For example, a mocked AWS S3-like library module, which can access object storage services provided by other vendors while maintaining the syntax and semantics of the proprietary SDKs of AWS S3 [40].

In addition, the Evaluation-as-a-Service (EaaS) is available to perform a benchmarking analysis of the performance and cost of various FaaS providers for each workload. The EaaS framework comprises the following components: Authentication, Adapter (to select the appropriate provider), Cloud's Logging Query (for debugging purposes), Local Logging, Cost Model, and Analysis.

The Serverless multi-cloud project shares a similar objective to our own, as it too focuses on abstracting the deployment of multi-cloud serverless solutions. However, it employs custom SDKs (as opposed to supporting DSL) and is designed specifically for two solutions, FaaS and BaaS, deployment of cloud functions and Virtual Machines (VMs), respectively.

2.4.4 Serverless Workflow

Serverless Workflow aims to create a vendor-neutral, platform-independent, and declarative language for defining workflows in the serverless computing domain, by helping to close the gap between business needs and technology implementation. The absence of a standardised method for defining and modelling workflows results in the need to

constantly learn new techniques hinders the development of common libraries, tools, and infrastructure, and reduces portability and productivity in workflow orchestration. This framework addresses the need for a community-driven, vendor-neutral, and platform-independent language specification for workflows in the serverless computing domain. The use of a specification-based workflow language enables modelling workflows once and deploying them on various container/cloud platforms with consistent execution outcomes.

The Serverless Workflow specification has several components, starting from a workflow language definition using the Workflow JSON schema and YAML formats for modelling the workflows. SDKs are currently available for *Go, Java, .NET, TypeScript, and Python*. Set of workflow extensions which allow users to add additional non-execution-related information to improve workflow performance, for example, *kpis*, rate limiting, simulation, and tracing. Finally, a Technology Compatibility Kit (TCK) is to be used as a specification conformance tool for runtime implementations.

The TCKs are specific implementations of workflow management systems, to delegate and manage units of work. One of the provided runtimes is Synapse [30], which aims to start instances of workflows using configured Runtime Host to create an isolated computing unit, known as workers. For its processing, should it be a process (when running naively), a container (when running on Docker), or a pod (for Kubernetes). This way the model is created via the JSON, YAML, or SDKs and feeds the specification runtimes to execute and manage the workflows.

In this instance, Serverless Workflow aims to create an independent workflow management tool that offers the definition and execution of workflows using units of work. The platform abstracts the runtime environments to enable the use of various external workflow computing units. Serverless Workflow also employs its own workflow schemas, defined using DSL and SDKs. Conversely, our proposed framework seeks to abstract the DSL definition itself to support languages across all cloud providers, enabling deployment to their respective providers. Additionally, our framework does not focus on the runtime aspect, which is the responsibility of the respective provider.

2.4.5 Temporal Platform

The Temporal Platform is comprised of two key components: the Temporal Cluster and Worker Processes, which collectively provide a runtime environment for executing workflows. The Temporal Cluster is a server that is paired with a persistence mechanism, whereas the Worker Processor is responsible for polling the task queue, removing a task from a queue, executing code in response to a task, and providing results to

the server. The execution of a Temporal Application involves multiple Temporal Workflow Executions, each of which possesses a separate local state, operates concurrently with other executions, and communicates with the environment and other executions via message passing. This open-source platform facilitates microservices orchestration and offers SDKs for *Go, Java, PHP, and TypeScript*.

The Temporal Platform's Workflow and Activity functions are its two primary functions. Workflows in Temporal are cohesive functions that support retries, rollbacks, and human intervention steps in case of failure. Temporal encourages the Workflows-as-Code paradigm, which may be more intuitive for developers. Workflows in Temporal are not capable of directly calling external APIs; instead, they orchestrate the execution of Activities. An Activity is a standard function or object method that performs a single, well-defined action (either short or long-running), such as calling another service, transcoding a media file, or sending an email message. Workflow code coordinates the execution of Activities and persists in the outcomes. In the event that an Activity Function Execution fails, subsequent executions commence from an initial state. As a result, Activity functions may contain any code without constraints.

The focus of this tool is to create a generic workflow manager that can replace existing cloud providers, supporting multiple SDKs for different programming languages. It is capable of calling multiple types of services, error handling, and automatically scaling them using its own clusters. In comparison to our framework, the key difference lies in its ability to comprehend and support multiple popular cloud providers. Although the objective is similar, our framework places emphasis on facilitating the rapid migration of cloud providers. This is particularly relevant in the real world, where organisations often use cloud providers for various solutions and require the flexibility to migrate quickly and seamlessly.

2.4.6 QuickFaaS

QuickFaaS is a desktop tool for defining cloud-agnostic functions and deploying them on a cloud platform. QuickFaaS aim to mitigate the problem of vendor lock-in for cloud functions and to improve the developer's experience in serverless computing. This tool supports two cloud platforms, Azure and GCP, and aims to support a wide range of providers. In this way, this tool enables the creation and deployment of functions in the cloud, where the user develops the code in the language they want and exposes it in the cloud. All this can be done via the user interface, making it easy to choose the project, the name of the function, the dependencies and the function settings. Once the FaaS function has been deployed, the URL to trigger the function is

provided [9].

Comparing QuickFaaS with the aim of this work is quite similar as both try to eliminate the problem of vendor lock-in. However, this work aims to provide agnostic deployment of function composition, while QuickFaaS deploys independent agnostic functions, such as Google Cloud Functions or Azure Functions. Both, tool and library, aim to facilitate migration between cloud providers and the reuse of solutions in a multi-cloud environment.

3

OmniFlow Solution

The specificity of each platform enforcing its own unique workflows specifications and formats arises a vendor lock-in problem, as previously mentioned in Chapter 2. Consequently, the implementation of workflows can become heavily specific and reliant on the provider's requisite specifications. Each cloud provider has its serverless solutions that can naturally integrate easily with each other. For instance, AWS provides solutions such as S3 buckets, Lambda functions, and state machines, which integrate easily with each other. Typically, the technologies are specific to each provider and not supported by competing providers, since each provider supports and integrates with its frameworks. Thus, this complicates the use of different platforms. These challenges complicate the developer's ability to choose a cloud platform or migrate to a new one, introducing even more vendor lock-in risks. OmniFlow is a library focused on the development and deployment of function composition. The aim is to increase programmer productivity when developing and migrating workflows to multiple cloud providers. The library supports deployment on two cloud platforms, AWS and GCP, and is prepared to support more platforms.

In this chapter, we begin by presenting the main components of the solution, such as the entity model of a workflow. The DSL created to define cloud workflows is also introduced, and complemented with the development cycle and implementation details. Finally, we discuss the limitations encountered during the development of OmniFlow.

3.1 Components Overview

Omniflow involves two external interactions, one with developers, allowing them to generate and deploy workflows in the cloud, and the other with cloud providers, where the deployed function composition resides. In Figure 3.1, we present a C4 model diagram [6] that effectively models the context of the Omniflow software system, offering insight into its broader relationships with external entities. Within Omniflow, developers have the capability to perform two actions: defining function compositions, known as workflows, and subsequently deploying these defined workflows to FaaS platforms. Following the deployment to the chosen cloud provider, developers can manage, execute, and edit the workflow directly on the platforms.

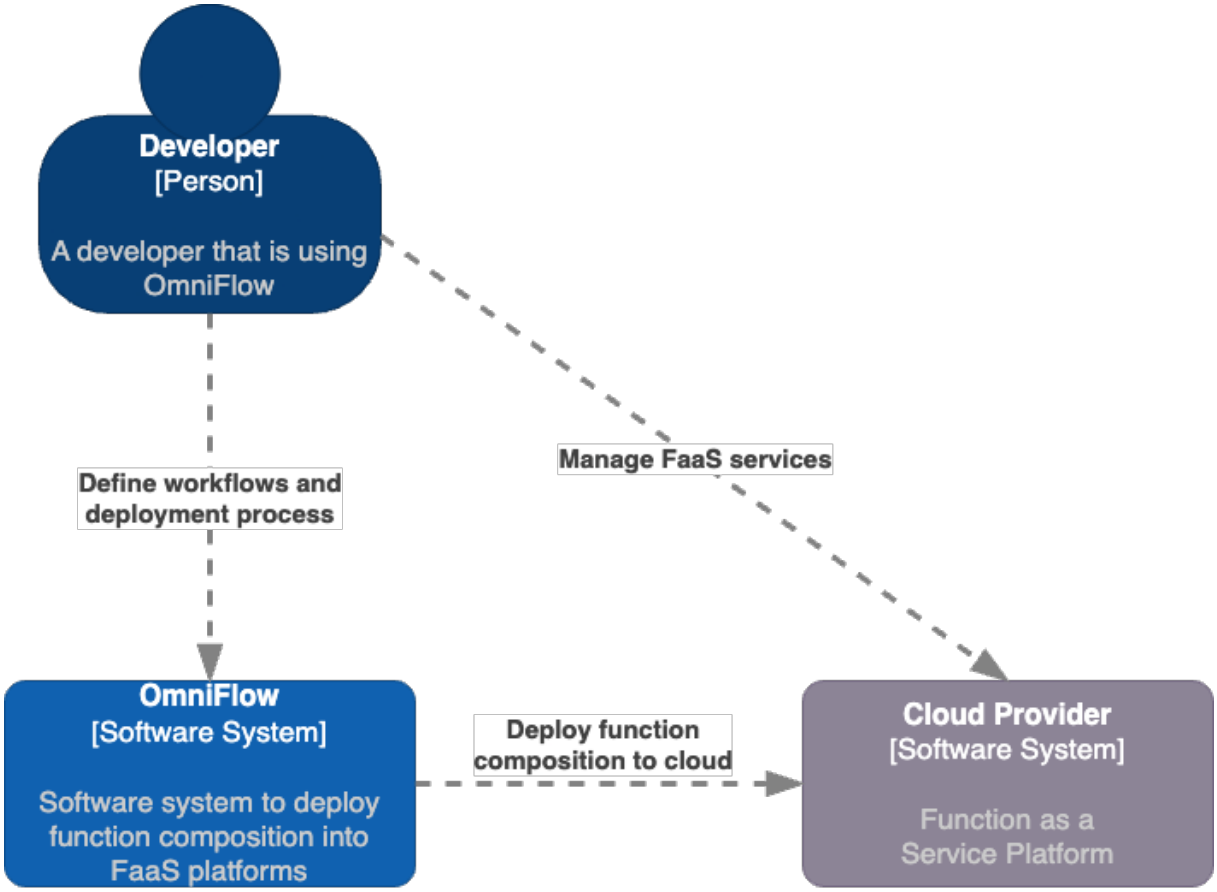


Figure 3.1: Omniflow system context diagram

Exploring the components of the Omniflow library at a deeper level, we can detect four core components: the **DSL**, the **model**, **renderers**, and **deployment** services. Each of these components has a distinct responsibility within the system: DSL, serves as an intuitive interface for constructing the model; Model, it is an abstract representation of the problem domain. Renderers are tasked with the important role of transforming the

model into the language specific to the chosen cloud provider. Finally, the deployment services step in to perform the deployment of a workflow definition to the cloud. The design is illustrated in the component diagram, Figure 3.2.

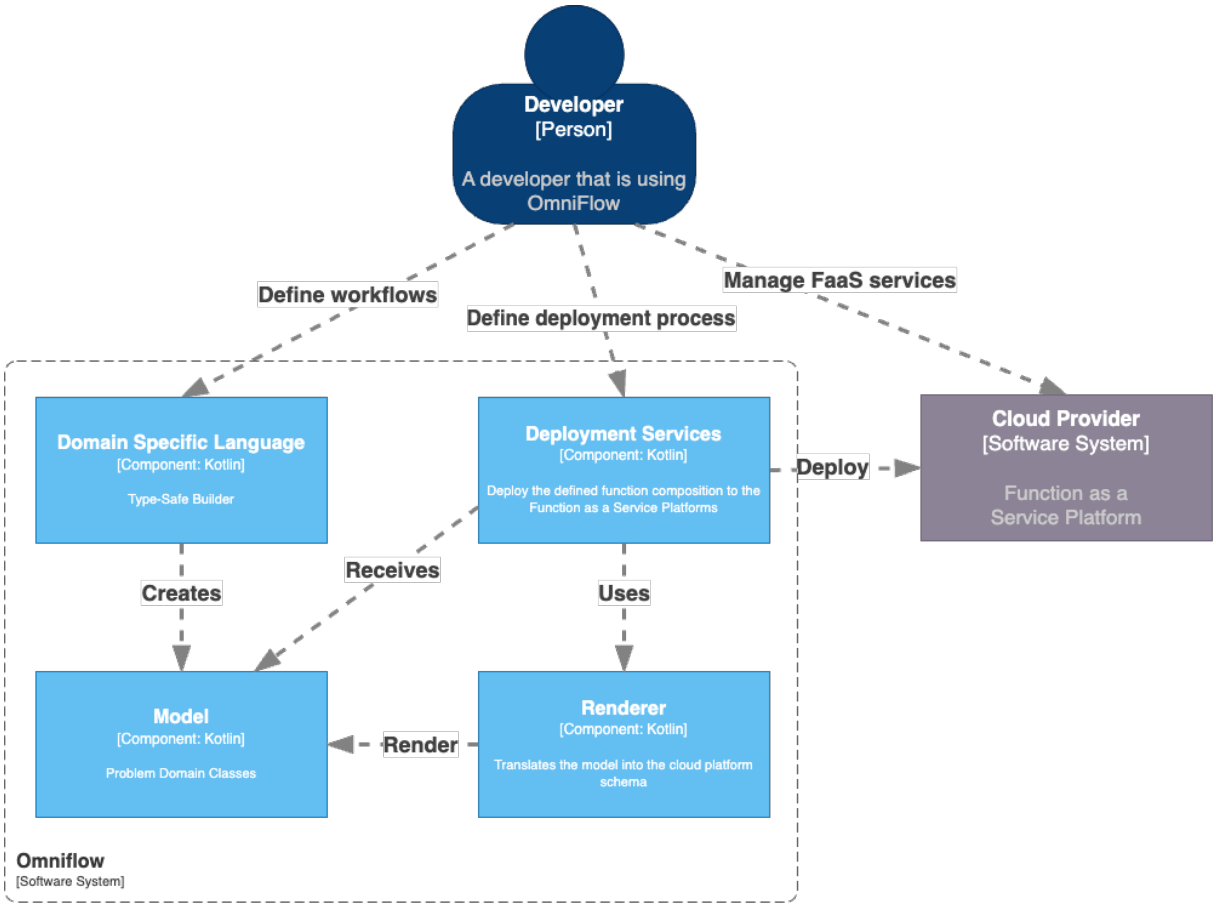


Figure 3.2: Omniflow component diagram

The ideal component structure necessitates scalability to accommodate new cloud providers, each with its unique implementations and solutions. Furthermore, the components structure should be agnostic enough to allow the same workflow to be deployed on any supported cloud platform within the library's ecosystem. The primary objective is to minimise vendor lock-in and enhance workflow migration across diverse cloud providers. To this end, the structure segregates the responsibilities between creating and deploying workflows in the cloud, ensuring that workflows remain generic and capable of deployment on multiple FaaS platforms. The proposed solution seeks to simplify the complexity of managing multiple cloud providers by providing a single library that leverages a flexible and user-friendly DSL. By employing this approach, users are empowered to deploy any workflow across a range of popular cloud providers with ease. Thus, users are only required to familiarise themselves with the provided DSL to effectuate workflow deployment across cloud providers and need not

acquire a new language or specific skills for each platform.

3.2 Workflow Entity Model

As previously mentioned, the primary objective of this work is to implement function composition across multiple FaaS platforms. To achieve this goal, it is essential to evaluate thoroughly the problem domain in this regard. Function composition involves executing a sequence of functions hosted in the cloud. A workflow represents a structured series of actions executed in a particular order. These actions can be seen as an aggregation of steps that can be compared to a state machine. In this context, the state machine executes individual steps with distinct functionalities, working together to achieve the overall workflow's purpose.

Given that each cloud provider interprets this problem domain uniquely, crafting a universal and generic definition for function deployments and their corresponding steps, which are both flexible and supported by all cloud providers, becomes not a straightforward task. The primary aim of function composition is to enable developers to address complex use cases effectively. This is achieved by executing multiple functions in a coordinated manner to achieve intricate outcomes. As demonstrated in the examples from Chapter 2, function composition involves orchestrating multiple functions within a specific flow. In essence, function composition, as the name suggests, is a composition of multiple functions. Each function incorporates a specific behaviour and functionality. By encapsulating these functions, where the initial step executes the first function (potentially with input parameters and direct outputs used by subsequent functions), we can alter the execution flow and initialise resources, among other tasks. Once we grasp the context of function composition, it is essential to establish a common ground for our domain model. In an abstract sense, a workflow is composed of multiple steps, with each step potentially having inputs and outputs based on its type. Consequently, a workflow may include optional input parameters and consist of one or more steps, each capable of various types and functionalities. Figure 3.3 depicts an Entity Relationship Diagram of the main entities involved in a workflow definition.

A workflow is composed by one or more steps, with each step representing an action, such as, iterating over a list or condition-based behaviour. Each step receives an input which is processed to generate or transform into an output. The input and output may consist of any type of data, including but not limited to numbers, strings, booleans, and objects. Furthermore, steps can be categorised into multiple types, including call types which represent HTTP requests (calls) to endpoints with specified requirements;

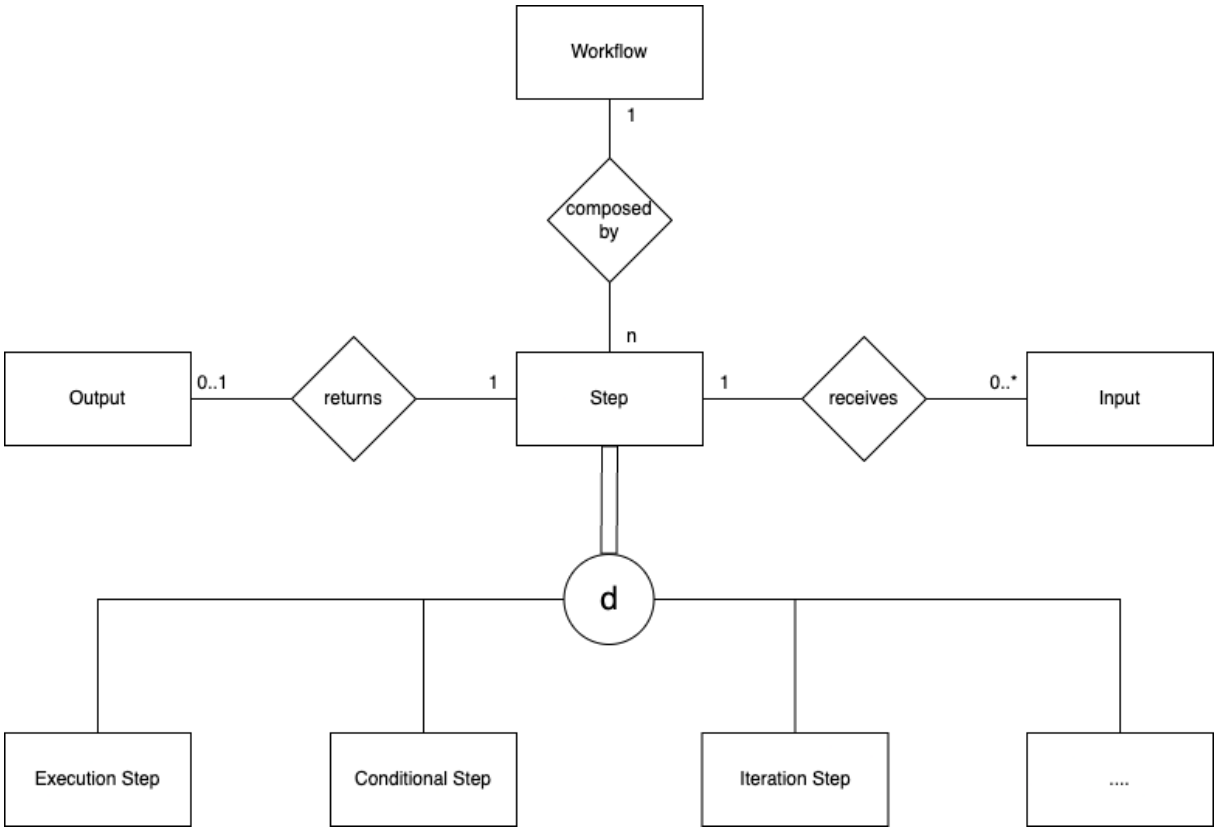


Figure 3.3: Abstract workflow definition

conditional types, which evaluate a condition based on provided data and determine the subsequent step; and iteration types, which process and iterate through a list of elements. Although additional step types exist, will not be taken into consideration. Introducing a variety of step types expands the spectrum of use cases available to programmers. Presently, we are emphasising three key step types:

- The Call Step type facilitates the invocation of external APIs, offering a wide range of possibilities for programmers. It empowers them to perform actions like creating, updating, or retrieving data from microservices, thereby expanding the scope of behaviour they can achieve.
- The Assign Step plays a role in creating and manipulating variables during the execution process. This capability is instrumental in managing the flow of data within workflows, enhancing flexibility and control.
- The Conditional Step provides control over the execution flow. Programmers can use it to define conditional logic, making decisions within workflows based on specific criteria.

These diverse step types not only enrich the toolkit available to programmers but also

unlock new ways for designing and implementing intricate workflows. This workflow definition is abstract and generic across all cloud providers. However, the actual implementation of the workflow is specific to each platform, and this is precisely where the challenge arises. The conversion of the abstract model into a provider-specific workflow, with the required specifications and details, is necessary.

3.3 Domain Specific Language

A proper understanding of the problem domain is crucial when creating ans abstracting the domain to the user. So there are multiple options to abstract the workflow definition, such as: 1. the definition of a DSL, to be more readable and closer to natural language compared to general-purpose programming languages; 2. by creating a new file format with a specific way to declare workflows, where a user does not require any programming language skills. Since the developers who create cloud workflows require software development skills, for creating variables, using conditional expression, and must be familiarised with the HTTP protocol, the software knowledge requisite is not problematic in this context. Besides that, the understating of the underlying application domain is mandatory, emphasising the design of a DSL, giving to the user the required expressiveness to generate the model easily [33].

A DSL is a high-level software implementation language, a specification language with a particular format and structure that abstracts from low-level implementation details, and possibly from particularities of the software, or platform used [37]. The potential advantages of DSLs are: the reduced time to market, reduced maintenance costs, reliability, optimizability, and testability [33]. On the other hand, the main disadvantage is the cost of their development, requiring both domain and language development expertise. [19] To create a well-designed DSL the detailed analysis and structuring of the application domains is a important prerequisite [33].

Therefore, there are still a few details to be refined in order to fully understand the problem, such as, what constitutes a workflow. A comprehensive workflow definition comprises several critical components: name, which acts as the unique identifier for the workflow; description, a human-readable field for explaining the purpose and function of the workflow; input, which designates the variable name that will capture all input parameters when the workflow is executed; steps, dedicated to specifying the steps that encapsulate the workflow's behaviour; result, is where the output of the workflow's execution is assigned. The basic unit in this domain is the Step, which, as the name suggests, represents a data processing unit with a small objective, such

as incrementing a number by one. A Step consists of metadata, such as name (identifier) and description, which are human-readable and informative fields. The context of a step corresponds to the step behaviour and purpose, for example, make HTTP requests, assign variables, and use conditions like, *if-else* and *when* expressions from *Kotlin*.

Going through the step context types, to initiate an HTTP request to an API or cloud function with specific requirements, the **Call Step** must be defined. A Call Steps is coupled to the HTTP protocol, so the step specification are tailored to it. These specifications encompass the method, host, path, authentication, body, headers, query parameters, timeout, and result. It is important to note that while method, host, path, and result are mandatory fields, the other fields are optional offering flexibility based on the use case. The "result" field serves as a variable where will be assigned the outcome of the request, whether it is a success or failure.

Assigning variables holds significant utility within workflows. To initialise or assign variables, the **Assign Step** is the essential tool. This step empowers the developer to create variables of various types. A variable is composed by a name and a correspondent value, where the value might be of any type, number, text, character or boolean.

Finally, in order to manage the execution flow through conditional expressions is accomplished using the **Conditional Step**. This step type allows the developer to define conditional expressions, specifying a name for the next step to jump to if the expression matches. There is also a default field, for the cases where all the previous conditions did not match, that is also a step name to jump to.

3.4 Workflow Development Cycle

The process for a user to deploy a workflow to the cloud starts by defining the workflow, followed by its deployment, regardless of the solutions provided by each cloud provider. The rendering process is abstracted away from the user, where the user does not require the knowledge of the cloud provider's schema and language. Workflow definition through DSL relies on Kotlin's Type-safe builders [21], while deployment is simplified using a straightforward builder, enhancing user experience.

The data structure that best applies to the model is trees, where each node in the tree corresponds to a class in the model and can contain multiple child nodes. The organisation of the tree is as follows: the root node is the workflow and its child nodes are the various steps; each step can contain multiple child nodes, depending on the type of step. To ensure that the order of the steps is respected, when traversing the tree

it is necessary to use a Depth-first search algorithm. Concerning the structure of the project, it has been organised by the supported cloud providers, and composed of a set of components that must be respected, such as renderers, deployers, services and strategies. This makes it easier to scale the solution when new cloud providers need to be added.

Each cloud provider is made up of multiple renderers for each model class; a deployer for a specific platform, which converts the workflow defined in the platform's schema and uses it to perform the deployment to the cloud; the service, which knows how to use the cloud platform's SDK and make requests to perform the deployment. Finally, the strategies that link the model classes with the renderers, so that when a node in the tree (model class) is obtained, the renderer of that class is known. However, there is a set of general behaviours, the DSL exposed to the user, the builders, and the way the tree is traversed, which are the same regardless of the cloud provider.

It is worth mentioning that most of the components created in this project are not customisable and accessible to the user, because there is only one way to deploy to a cloud platform, services usually only contain one specific implementation, and a model class only has one way of being rendered for each cloud provider. Thus, the only components accessible to the user are the DSL for creating workflows and the deployers for deploying the workflow created on a cloud platform. All the classes that are builders or deployers are public, while all the other classes are private to the module, also known as "internal" in the Kotlin environment, and cannot be accessed by the user.

Deployment components are responsible for rendering the model and deploying it to the cloud with specific requirements. OmniFlow performs all actions synchronously, including definition and deployment. However, it is worth noting that the software is designed to efficiently handle multiple deployments in parallel. The program relies on Kotlin's Data Classes, designed to efficiently manage data and enhance the cleanliness and readability of model classes. The following sequence diagram, illustrated in Figure 3.4, provides insight into OmniFlow's interactions with various cloud providers.

3.4.1 Workflow Definition

The design of the DSL is one of the most important phases, as it should be easy to use and generic enough to be used on all cloud platforms. As previously mentioned, we have implemented the Builder pattern to create model objects, abstracting the responsibility of their construction. Each Builder class implements a generic functional

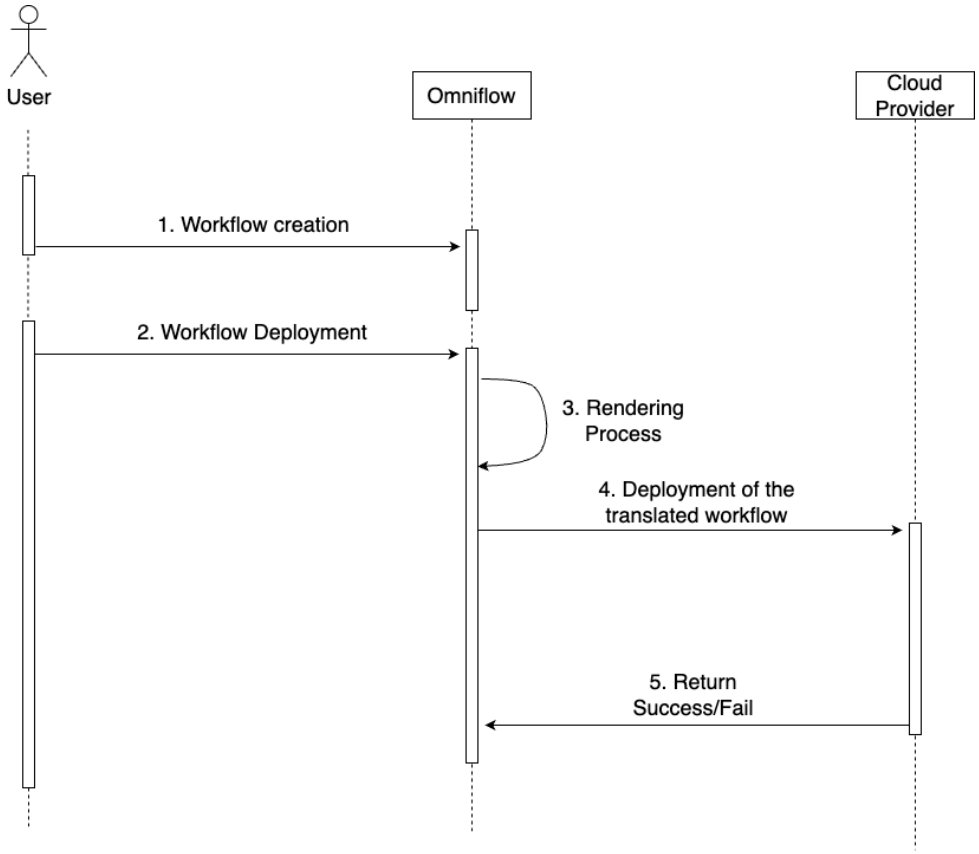


Figure 3.4: OmniFlow sequence diagram

interface, with a single *build* method. Corresponding to each model class, we have introduced dedicated builder classes formed by the necessary fields, to construct the respective model objects. The integration between the Kotlin's Type-sage Builders with the model builders, crafts the DSL. The primary motivation behind using Type-Safe builders lies in their capacity to facilitate the creation of Kotlin-based DSLs suitable for building complex hierarchical data structures. In Listing 3.1, it is possible to observe how a step is declared using the DSL, along with its associated fields. Notably, "name", and "description" are string fields without restrictions, while "context" corresponds to the step's behaviour, such as **call**, **assign**, or **switch**. All these fields are mandatory when creating a step.

```
1 step {
2     name("call-step")
3     description("Make request to example API")
4     context(...)
5 }
```

Listing 3.1: Step definition using Domain Specific Language

The **Call** step stands out as the most extensive, featuring a vast set of fields and meticulous details. Given its coupling with the HTTP protocol, this step type necessitates a broad set of specifications, refer to Listing 3.2.

1. The *method* parameter is determined by the HttpMethod enumerator, covering all the HTTP methods, including *GET*, *POST*, *DELETE*, *PUT*, and more.
2. *Host* and *path*, both are string-based fields, with no restrictions.
3. The *authentication* field contains a type, scope, scopes, and audience, each of which is expressed as a string.
4. *Body* declaration can take either a string or an object.
5. *Headers* and *queries*, are maps composed of multiple key-value pairs, where the key represents the header or query name, and the value is the header value.
6. *TimeoutInSeconds* is an integer field specifying the timeout duration of the request.
7. Finally, the *result* denotes the variable name wherein the response will be stored.

The *method*, *host* and *path*, and *result* parameters are mandatory, as the HTTP protocol needs them to provide the request, while the other parameters are optional.

```
step {
    name("call-step")
    description("Make HTTP request to API")
    context(
        call {
            method(HttpMethod.POST)
            host("example-api.com")
            path("/default/v1/")
            authentication(
                type("OAuth2")
            )
            body(
                "Hello World!"
            )
            header(
                "Content-Type" to "application/json"
            )
            query(
```

```
                "filter" to variable("name"),
                "page" to value(1)
            )
            timeoutInSeconds(5)
            result("sumResult")
        }
    )
}
```

Listing 3.2: Call Step example via Domain Specific Language

To facilitate variable assignments, the **Assign** context proves to be both intuitive and straightforward, as it exclusively deals with variable assignment and creation. Defining a variable is by calling the method *variable*, composed of a variable name, *equal* expression, and a value. The assigned value can take on all the Kotlin primitive data types, allowing you to leverage Kotlin's capabilities to generate values, for instance, you can employ constants like *Random.nextInt()*. The equals expression, an infix method in Kotlin, combines the left and right values – the variable name and the assigned value – into a Pair object, offering a highly declarative approach. See Listing 3.3 for a better understanding.

```
step {
    name("assign-step")
    description("Initialize variables")
    context(
        assign {
            variable("number" equal Random().nextInt())
            variable("text" equal "Hello World!")
            variable("isOk" equal true)
        }
    )
}
```

Listing 3.3: Assign Step example via Domain Specific Language

The **Switch** step shares similarities with the *when* statement in the Kotlin programming language, allowing you to create multiple conditional branches. This step comprises a set of conditions, and a *default* branch, which gets executed if none of the conditions are met. Each condition consists of a binary expression used for matching, when is matched, the step name specified in the *jump* field, will be executed next. A binary expression encloses a variable, a binary operator, and a constant value. Supported

binary operators include *equalTo*, *notEqualTo*, *greaterThan*, *greaterThanOrEqual*, *lessThan*, and *lessThanOrEqual*. All fields are mandatory except for default, see Listing 3.4.

```
step {
    name("condition-step")
    description("condition")
    context(
        switch {
            conditions(
                condition {
                    match(variable("c") equalTo value(0))
                    jump("Assign1ToC")
                },
                condition {
                    match(variable("c") greaterThan value(0))
                    jump("DivWithC")
                }
            )
            default("Assign1ToC")
        }
    )
}
```

Listing 3.4: Condition Step example via Domain Specific Language

All the steps listed above must be defined in the context of a **Workflow**. A workflow has a generic architecture, necessitating only a few essential components: a *name*, a *description*, an *input*, and a *result*. The *name* and *description* serve descriptive purposes, while the *input* designates the variable name where the execution arguments will be stored. The *steps* component encompasses the set of instructions that specify the workflow's behaviour, and the *result* denotes the variable where the workflow's output will be stored. Upon assigning a workflow to a variable, a Workflow model object is promptly created and allocated to the variable. It is worth noting that workflow creation is eager and occurs immediately. For a visual representation of this structure, refer to Listing 3.5 for a simplified example of a workflow's definition.

```
workflow {
    name("workflow-name")
    description("Workflow Description")
    input("args")
    steps(...)
```

```
6       result("workflow-result")
7   }
```

Listing 3.5: Workflow definition via Domain Specific Language

3.4.2 Workflow Deployment

Once the workflow is defined, the next phase is deploying it to the chosen cloud provider. As mentioned earlier, the supported cloud providers include Amazon's State Machines and Google's Cloud Workflows, each with its unique set of parameters and credential requirements. To successfully complete the deployment phase, is required to create a deployment context tailored to the cloud provider's specific needs. The deployment services necessitate two primary inputs: the deployment context, and the predefined workflow. It is important to note that while both cloud providers have their distinct deployment prerequisites, the workflow to be deployed remains the same - the Workflow model previously crafted using the DSL.

The DeployContext instance is created using the constructor of a *Data Class*, while the CloudDeployer service, responsible for the deployment itself, is constructed via the Builder pattern. Google's deployment prerequisites encompass various aspects of Google's project, and service account to which the workflow will be linked to. It is required to specify essential details such as the workflow's identifier, description, labels, and zone, all of which are outlined in Listing 3.6 for your reference. It is decisive to note that running this piece of code necessitates user credentials to authenticate the user's identity and verify access to the specified project and service account. To provide this proof, the environment variable *GOOGLE_APPLICATION_CREDENTIALS* must be set, which should point to the location of a credential JSON file. This JSON file can take one of two forms: a credential configuration file for workload identity federation or a service account key. It is also worth mentioning that Google's Application Default Credentials strategy [13] is employed by their authentication libraries to automatically locate the appropriate credentials based on the application environment, simplifying the authentication process.

```
1   val googleDeployContext = GoogleDeployContext(
2       workflowId = "workflow_example",
3       workflowDescription = "Workflow for testing",
4       workflowLabels = mapOf(
5           "environment" to "testing",
6           "app" to "omni-flow"
```

```
    ),
    projectId = "<your-project-id>",
    serviceAccount = "<your-service-account>",
    zone = "us-east1",
)
GoogleCloudDeployer.Builder()
    .build()
    .deploy(workflow, googleDeployContext)
```

Listing 3.6: Deployment of Google Workflow

Deploying to Amazon follows a similar process to Google's. It entails specifying the State Machine name, the ARN role, the region, and optionally, tags. Just like Google, Amazon requires authentication credentials to be defined via environment variables: *AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY, and AWS_SESSION_TOKEN*. During Amazon's deployment phase, it attempts to load the credentials provider [2] from these environment variables. These credentials are cryptographically signed and issued by AWS for secure authentication.

```
val amazonDeployContext = AmazonDeployContext(
    stateMachineName = "state_machine_example",
    tags = mapOf(
        "environment" to "testing",
        "app" to "omni-flow"
    ),
    roleArn = "<your-role-arn>",
    region = "us-east-1",
)
AmazonCloudDeployer.Builder()
    .build()
    .deploy(workflow, amazonDeployContext)
```

Listing 3.7: Deployment of Amazon State Machine

3.5 Implementation Details

In terms of technology we choose Kotlin [20] for several reasons: modernity, conciseness, safety, and interoperability with Java and other languages make it versatile. Additionally, Kotlin offers multiple avenues for code reuse across various platforms,

enhancing the project's adaptability. OmniFlow was meticulously crafted as a multi-module Maven project, designed for scalability and extensibility. This project architecture simplifies the integration of new components such as user interfaces, benchmarking tools, and more, as well as the inclusion of external dependencies. The OmniFlow was developed with Kotlin version 1.7.20 compatible with Java 11. The majority of the development dependencies used were the provided SDKs from the cloud providers for workflow deployment. Notably, a significant portion of the code base is module-internal, hidden from developers, with only the DSL and deployment components accessible, ensuring a simplified and secure development experience. For testing purposes, were used utility libraries *MockK* [28], for creating mock objects, and combined them with the assertions from *Strikt* [32], all within the trusted *JUnit* testing framework.

Omniflow has encountered various challenges that align with different design patterns, including creational, and behavioural patterns. Design patterns are solutions to recurring software design problems. In object-orientated programming languages, design patterns establish the relationships between code. Applying design patterns is a good practice for avoiding design problems and facilitating the reuse and flexibility of software [15]. A design pattern abstractly describes, identifies the main points and names a design structure for a problem that can be reused in multiple scenarios [14].

Begin by discussing the creational aspect, for managing object creation, we employed the Builder pattern. This pattern allows us to construct model objects, such as Workflows and Steps iteratively. By using the Builder pattern, we can create multiple representations for an object, abstracting away the intricate details of how each object is constructed. Consequently, creating builders for each model entity enhances modularity and simplifies the process of generating these model entities. Another essential pattern we employed is the Visitor pattern, a behavioural design pattern that emphasises the separation of concerns, by decoupling algorithms from the objects they operate on. This pattern proves especially useful when dealing with data structures, as it allows us to "traverse" these structures, visiting each object once and applying custom operations to them. In our implementation, we applied the Visitor pattern by segregating the rendering algorithm from each node in the tree, represented by model objects. By doing so, we abstracted the traversal process of the tree from the renderers. We opted for a Depth-first search algorithm to accomplish this, but our program is designed to accommodate other tree traversal algorithms as needed.

3.5.1 Limitations

OmniFlow aspires to facilitate workflow deployment across various common cloud providers, but this ambitious goal inherently comes with certain constraints. Each cloud provider adopts its unique approach to address the challenges posed by function composition, leading to diverse limitations in their solutions. Consequently, OmniFlow follows the most restrictive scenarios set by cloud providers. For instance, if Google enforces a maximum character limit of 15 for workflow names, while Amazon requires that workflow names can only contain alphanumeric characters, OmniFlow will adapt accordingly, imposing a maximum 15-character limit and alphanumeric constraints on workflow names. This approach ensures that OmniFlow remains compatible and generic for all supported cloud providers, but it also comes with the potential drawback of being overly restrictive for developers. Table 3.1 provides a comprehensive breakdown of the limitations enforced by OmniFlow and offers a comparison with the constraints imposed by supported cloud providers.

Feature	**OmniFlow**	**GCP**	**AWS**
Variables	Partially supported	Supported	Supported
Conditional Operators	Partially Supported	Supported	Supported
Step description	Supported	Not supported	Supported
Workflow identifier	Supported	Partially supported	Supported
Input/Output	Supported	Supported	Supported
Iteration Step	Not supported	Supported	Supported
Parallel Step	Not supported	Supported	Supported

Table 3.1: OmniFlow's feature limitations

Introducing the variable feature into OmniFlow presented a considerable challenge, entailing the resolution of numerous complexities. As a result, OmniFlow currently supports variable usage exclusively in conditions and the query parameters of the call step, with the variable assignment made possible through the assigning step. In contrast, both supported cloud providers, GCP and AWS, offer robust variable support, allowing variables to be used in every field across all step types within their workflow definitions. Furthermore, they allow variable assignments to other variables, and their conditional expressions enable comparisons between multiple variables. However, OmniFlow does not extend support to these advanced functionalities.

Turning to conditional expressions, GCP and AWS provide comprehensive support for

various conditional operators, encompassing logical (&&, ||, |), equality (==, !=), and comparison (<, <=, >, >=) operators. In contrast, OmniFlow limits conditional step comparisons to equality and comparison operators, restricting the range of possible comparisons in the conditional step.

Human-readable metadata plays a significant role in OmniFlow, assisting developers in comprehending the purpose and logic behind the defined steps. As such, OmniFlow binds a description for each step, a practice also followed by Amazon. In contrast, Google's defined steps lack descriptions, creating potential challenges for developers seeking clarity in workflow objectives. Workflow identifiers are another differentiating factor. While both OmniFlow and Amazon require mandatory workflow names, Google only requires workflow names during the deployment phase, avoiding the need for them in source code.

In the realm of function composition, the handling of step input and output varies across cloud providers. Google's procedure, while interesting, shares input and output across all steps, meaning that variables persist throughout the workflow execution unless explicitly deleted. In contrast, Amazon's approach is more modular, with each step having its own input and output variables. This setup enables steps to select their input and specify the output exposed to subsequent steps, promoting a more granular and optimised workflow structure.

Finally, OmniFlow currently lacks support for the Iteration Step, designed for traversing collections of elements sequentially without exposing their underlying structure. Similarly, the Parallel Step, which facilitates parallel processing of multiple steps concurrently, is also absent, although both Google and Amazon provide support for these advanced functionalities.

4

Evaluation

This chapter presents an evaluation of OmniFlow's performance, showing which metrics were used and how the benchmarks were developed. The metrics focus on common use cases, to understand how the rendering and deployment process behaves. In addition, the results obtained for the measurements in question are presented. After demonstrating the results, conclusions are drawn about the rendering and deployment process, which cases have the highest and lowest cost, and whether OmniFlow is beneficial to the programmer or not.

4.1 Overview

One of the main points of our library is the conversion and translation of the model into a language specific to the cloud provider required. Consequently, the efficiency of these transformations, carried out by the renderer component, becomes relevant. A detailed examination of the time taken by the software to carry out this transformation is therefore pertinent. Following this path, an investigation was carried out into the rendering process with various use cases, which use the various types of HTTP calls, variable assignments, and conditions. In this sense, multiple use cases were evaluated to cover a complete spectrum of characteristics, emerging five possible references, all different and remarkable. These references include workflows composed of, independent steps, steps that use and create variables, dynamic workflows with binary

conditions (also referred to as if-else conditions), and the fifth case is workflows with multiple decision points.

The two core topics in the implementation are the rendering phase, which is responsible for the model translation into a final serialised format; and the deployment phase, sending the result to the cloud and makes possible the workflow execution. Deployment components use out-of-the-box software components (SDKs) provided by each cloud platform to carry out the deployment of a workflow in the cloud. Therefore, the study of the time taken to carry out a deployment was omitted for our study case, as the performance of the external libraries is not our responsibility, but that of the cloud providers. However, it is interesting to evaluate how the quality of a workflow generated by the library affects deployment time, by comparing a workflow created using the DSL with a sample workflow. This comparison allows differences to be found between a workflow generated by the DSL and a workflow realised by a human and gives a greater insight into how these changes affect deployment time.

Once the selection of appropriate metrics is completed, the next step entails determining the optimal approach for their implementation. One of the initial challenges that surface involves defining a strategy for the implementation of rendering and deployment benchmarks. In the context of rendering benchmarks, it is clear that it is necessary to test numerous workflow use cases. This process starts with manipulating the number of steps in a workflow, followed by examining the relationships between steps, and diversifying the types of steps. The number of steps within a workflow varies from a small number of steps, workflows with one to ten steps for example, to a large number of steps, running into thousands, involving relationships between the steps (use of variables and conditions). The range of dimensions starts with 1 and then extends through a logarithmic progression, encompassing values such as 10, 100, 1.000, 10.000, and 100.000. The connections between steps are designed to accommodate various scenarios, structured as follows.

1. Independent Steps: These steps operate autonomously, without dependencies between them. They exclusively involve HTTP calls.

2. Variables: In this setup, HTTP calls incorporate predefined variables, enhancing their functionality.

3. Conditional Pathways: This arrangement introduces conditions to guide the workflow's progression based on expressions, featuring both single and multiple decision points.

These configurations enable a comprehensive study of step relationships within workflows. For a more detailed exploration of each configuration, refer to the upcoming Section 4.2.

Considering these factors, for a more tangible representation, each workflow is evaluated 100 times to determine the average execution time. Subsequently, we compare time spent across workflows of different sizes to understand whether it follows linear or exponential growth. This analysis helps us to identify potential scalability issues, specifically, when resource consumption increases disproportionately with a larger number of steps. If processing time displays an exponential pattern and metrics vary significantly, it may signal inefficient implementation. On the other hand, the deployment phase does not necessitate an extensive examination of workflow dimensions or types. Instead, it entails an exploration of the disparities between an example workflow, documented by the cloud provider, and a workflow generated by the DSL. This investigation aims to understand the influence of these differences on deployment time.

4.2 Rendering Process

The following sections explain in detail each use case for the various types of steps, exploring the advantages of each one.

4.2.1 Independent

A simple scenario for benchmarking begins with an elementary workflow composed of individual steps that operate independently of one another, without any dependencies. In this case, the rendering process from each step does not require any information from the previous steps, to successfully render the current one. Each step is autonomous, without depending on any contextual clues from previous steps. This implies that each step neither utilises prior step results for processing nor exhibits conditional behaviour based on them. Henceforth, this workflow is referred to as independent steps, which operate autonomously and are disconnected from one another.

4.2.2 Variables

In typical scenarios, programmers often find the need to utilise variables, to enable effective manipulation and storage of data, to achieve the desired behaviour. Recognising this significance, it was valuable to explore a workflow that incorporates the initialisation, or usage, of variables across all steps. This entails constructing a workflow wherein the steps rely on variables initialised through the **assign** step. Establishing this relationships between the steps, introduces a higher level of complexity to the rendering process, requiring careful consideration of the contextual information derived from previous steps.

4.2.3 Binary Conditions

Conditions allow programmers to control the program flow, depending on decisions, based on the outcome of logical statements or expressions. The control of the program flux is fundamental when implementing software. Recognising this importance, it became imperative to assess the impact of rendering conditions. In an initial approach, the focus was on comprehending how the renderers respond to binary conditions, such as **if-else** expression. Thus, it became necessary to create workflows composed of steps that are conditions, in order to, change the workflow behaviour when executed.

4.2.4 Multiple Decisions

When it comes to conditions, it is common to use conditional statements that involve multiple decisions. For instance, the *switch* statement in Java. With multiple decision expressions, expand the possibilities of program flow by allowing a branching based execution. So, it was considered relevant to explore the usage of conditions within workflows. In particular, the study involved creating workflows that incorporated **switch** steps, enabling the ability to skip undesired steps. This facilitated a more versatile and dynamic program flow, enhancing the overall flexibility and efficiency of the workflow execution.

4.2.5 How Workflow Variations Affect Deployment

The outcomes from the rendering process are engineered to emulate human-like definitions of workflows, at least as close as possible. However, achieving this balance

can be challenging. Therefore, a relevant aspect involves observing the differences between automated and human-crafted workflows. To explore these changes, the study employed exemplar workflows from both Amazon and Google. Correspondingly, a counterpart workflow was designed using the DSL, aligning with the same objective as the provided examples. Although these instances attempted to adhere only to the supported renderer scenarios, some modifications were made to ensure compatibility with the library and ease of replication. Once the disparities between the generated and example workflows were analysed, a noteworthy observation emerged: the impact of these differences on deployment time to the cloud. With this investigation, it is possible to identify the differences between the results of a workflow created by a human and one generated automatically.

4.3 Benchmark Development

Initially, the project structure was established, which involves creating a dedicated sub-module responsible for benchmarking the **OmniFlow** functionality, found in the deployment sub-module. The next step was selecting a suitable benchmark tool or framework that aligns with the software requirements and offers the desired level of customisation. For this purpose, where complexity is not high, and a deep control is not necessary, a trade-off was considered in terms of time, reliability, and expertise. As a result, the Java Microbenchmark Harness (JMH) emerged as the preferred choice due to its ability to strike a balance in this trade-off.

JMH provides a Java-based API for constructing and analysing benchmarks, offering measurement in various time units such as: nanoseconds, microseconds, milliseconds, and seconds. Being focused on the Java Virtual Machine (JVM), it generates optimised Java code and configures the benchmarking environment accordingly. With the JMH API, it becomes possible to warm up the JVM environment, ensuring that outlier results are minimised. Additionally, it enables measurement of average execution times, samples, throughput (operations per unit of time), and other metrics related to code processing. By opting for JMH, the project benefits from a robust tool that caters to the software's benchmarking requirements while providing the necessary functionality and flexibility.

In order, to measure the time spent for a set of executions, and understand how the rendering time scales, it is required to generate multiple workflows composed by a scaling range of steps, as was said previously. To generate those workflows was created a Workflow generator, which receives the number of steps to create, and returns a

possible workflow. In order to create independent steps, involves generating identical call steps multiple times, which do not interact between them, or require variables. The size of the workflow is determined precisely by the requested number of steps. In the next scenario, variables are introduced, and the logic is as follows: firstly, two variables named *number1*, and *number2* are assigned with random integer values. Subsequently, a call step is created that utilises these two variables by passing them as query strings in an HTTP call. To ensure the consistent pattern of an assigned step followed by a call step, the maximum number of steps possible is equal to the input step size plus one, in the case of an odd step size. Furthermore, similar logic is applied to other use cases involving binary conditions and multiple conditions. For binary conditions, the workflow begins with an independent step, followed by a binary decision branch. Depending on the outcome of the previous step, either the subsequent step is executed, or the step two places ahead is performed. Hence, the maximum number of steps generated for this workflow is determined by the input plus two. Lastly, the remaining use case follows to the same logic as the binary condition, but with a higher number of potential steps to jump to. In these case, the workflow starts with an independent step, and a switch is generated with three possible conditions based on the result of the preceding step. However, the maximum number of steps present in the workflow is the input plus four, considering the additional options in the switch. Overall, by employing these logical patterns, the workflow is designed to accommodate different scenarios, varying step sizes, and conditions, providing flexibility and scalability. Each scenario was meticulously executed across numerous workflows of varying dimensions, facilitated by the workflow generator. For each distinct workflow and use case, the environment underwent a sequence of operations. Specifically, the rendering process was executed five times in order to prepare the environment, ensuring its optimal state. Following this warm-up phase, the rendering process was conducted a hundred times to derive both, the average processing time, and the corresponding standard deviation.

When it comes to implementing the deployment benchmark, a distinct approach was taken. In this case, the utilisation of the workflow generator was not required, as this metric directly aligns with an illustrative workflow provided by the cloud providers. Specifically, two representative examples were chosen: "text translation" and "save and get a pet from a store," which originated from Google and Amazon, respectively. With these illustrative instances as a foundation, the setup of the benchmarks involved leveraging the DSL to define the workflows, thereby harmonising their purpose with that of the chosen examples and understanding the differences after the rendering phase. It is important to note that the benchmarks focused exclusively on

the deployment phase. This phase was executed twenty times for each cloud provider, a repetitive process that enabled the derivation of both, the average processing time and the corresponding standard deviation. The section 4.4 presents the findings from the benchmark executions and the conclusions drawn from them.

4.4 Results

In general, the obtained results indicate an anticipated behaviour and do not reveal any signs of outliers, particularly in the context of the more resource-intensive executions. It is worth noting that, the increase in the number of steps during the rendering process, demonstrates a consistent linear relationship between, time, and step count, across all scenarios. The metric that demands the most memory and processor time, involving a workflow composed of 100.000 steps. The experimentation was conducted on a MacBook Pro, with an Apple M1 chip and 16GB of memory.

Interestingly, the most expected behaviour would be that rendering the call steps was the fastest since it does not use variables or conditions. However, upon closer examination, the reason underlying this outcome becomes apparent. Considering the nature of the rendering process, serialising the model into a string format, the HTTP call step contains an extensive set of fields, mandatory, and optional, to be processed. Consequently, affects the final processing time when a workflow is composed exclusively of this kind of steps, presented in Figure 4.1. Analysing the results of GCP, the standard deviation range exhibits a minimal starting point of 0,021 milliseconds, with a maximum deviation of 1,352 milliseconds. Regarding Amazon, the standard deviation begins at the same value of 0,021 and ascends to 31,14 milliseconds at its peak. This discrepancy signifies that the results from Google demonstrate better consistency compared to those from Amazon. In general, the deviation of the time spent across 100 calls for both cloud providers remains minimal and evidences a consistent process.

By incorporating variables, the expected outcome, as previously discussed, is a higher rendering time, due to the fact that the steps create and use variables, introducing more complexity. However, it is important to note that only two variables (*number1* and *number2*) are constantly assigned in each assign step. This limited number of fields to render reduces significantly the processing time. An interesting exploration point would involve scenarios where the assign steps are composed of a higher volume of variable assignments — potentially in the hundreds or more. For all the rendering scenarios, the **using variables** evidenced better results. Figure 4.2 provides an insightful visual, indicating that Amazon tends to take slightly longer to render, the model, compared

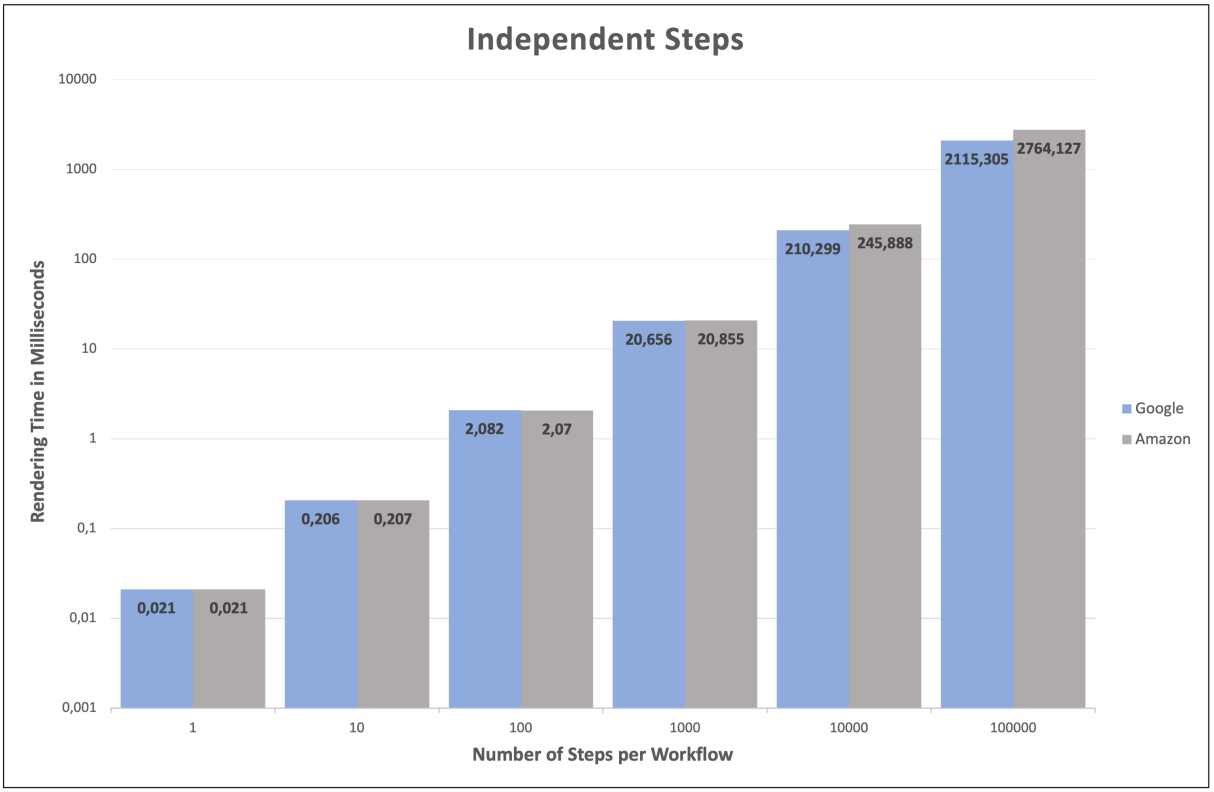


Figure 4.1: Rendering time for independent steps

to Google. Nevertheless, it is noteworthy that the total rendering time is considerably shorter than the independent steps scenario, with a difference of approximately one second. Digging further into the results, it is evident that for Amazon, the minimum standard deviation was 0,033 milliseconds, with a maximum of 1,66 milliseconds. On the other hand, Google showcases greater consistency in the lower limit, measured at 0,024 milliseconds, but displays less consistency for larger workflows, reaching 2,05 milliseconds. When comparing the standard deviation with the scenario involving independent steps, it becomes apparent that the utilisation of variables contributes to a heightened level of consistency during the processing phase.

Conditions introduce a degree of complexity to the rendering process, resulting in a relatively higher time consumption compared to the rendering of variables, but still less than, the independent steps. Notably, it is evident that Amazon exerts more effort in translation compared to Google. This distinction arises from the fact that defining a condition in Amazon involves a larger dimension of characters, contributing to the observed difference. Going deeper into the binary conditions, a notable distinction emerged: Google's results exhibited significantly faster performance compared to Amazon, showcasing a substantial difference of 804,119 milliseconds, in the worst-case scenario. This interesting disparity is particularly evident when dealing with a larger

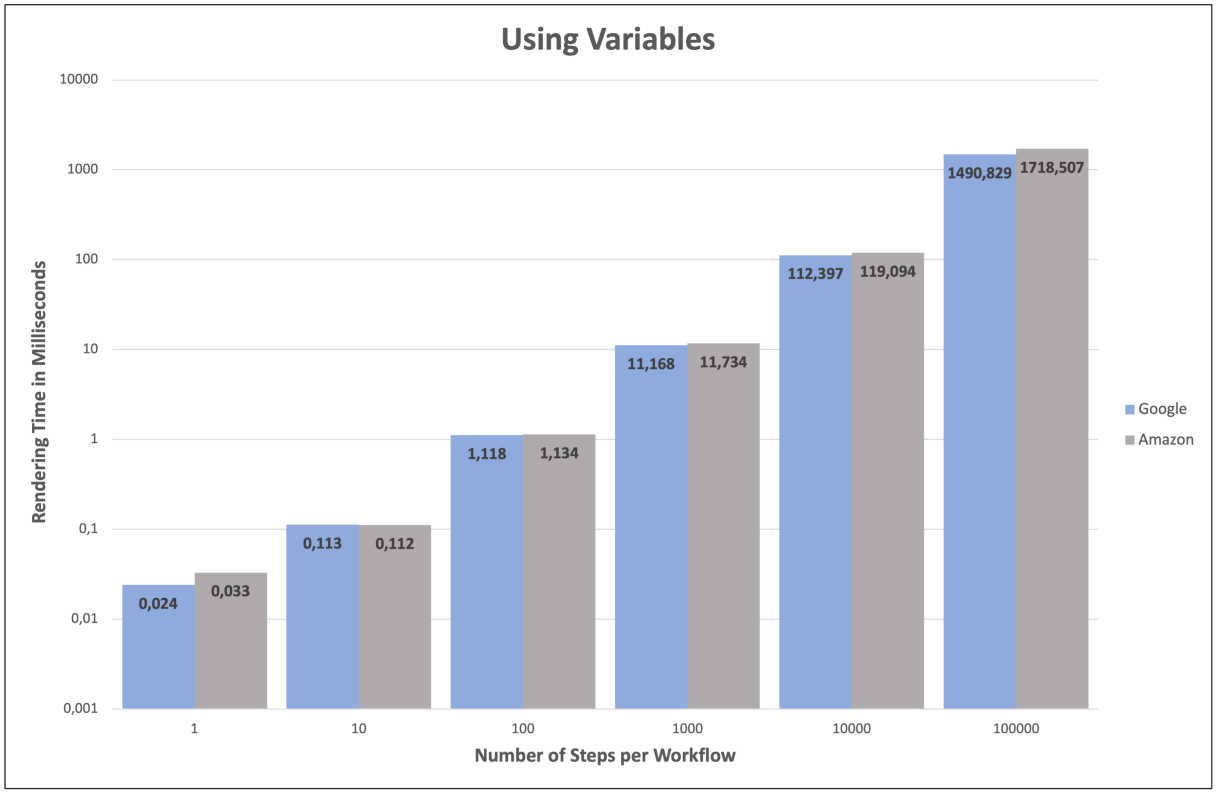


Figure 4.2: Rendering time using variables

number of steps, emphasising the impact of the number of steps composing a workflow. In terms of deviation patterns, the maximum values accounted for, 0,14%, and 0,05%, of the total time spent for Google and Amazon, respectively. This minimal variation reaffirms the consistent nature of the results for both cloud providers, as you can see in Figure 4.3.

A specific scenario involving the utilisation of a condition with multiple expressions, rather than the usual two, for example, if-else cases, introduces additional complexity due to the higher number of cases involved. This naturally leads to a slight increase in the time spent during rendering. See Figure 4.4. Although, the effort for the Google side is clear, having a more extensive set of expressions to render, results in a delay in time of 544,334 milliseconds, compared with binary conditions. Now, the difference between Google and Amazon is exactly 267,845 milliseconds, where Amazon maintains a similar result compared to the binary conditions. As a result, the contrast between Google and Amazon becomes even less evident, with a precise difference of 267,845 milliseconds. Remarkably, Amazon manages to maintain a comparable result when compared with the binary conditions scenario.

Last but certainly not least, the examination of how variations in workflows impact deployment time, has a significant importance. To comprehend the distinctions between

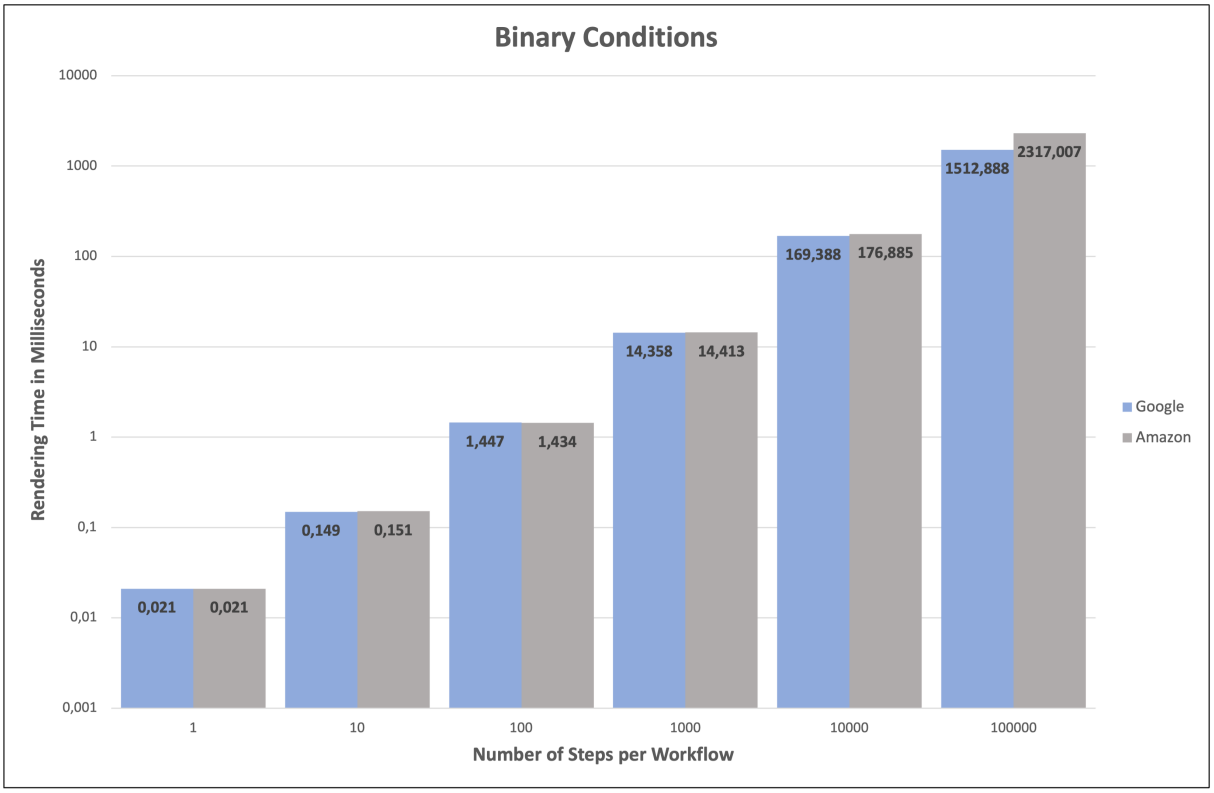


Figure 4.3: Rendering time using binary conditions

the serialised workflow, in JSON and YAML formats, and the example workflow, it is essential to visual analysis and the application of an algorithm. Looking at Amazon, significant disparities are evident. Generated steps consistently include **comments**, an explicit **InputPath** definition, variances in the order of the **Next** and **End** fields, and in the final step outcome, all of which are illustrated in Appendix B. In contrast, the quality of Google's results stands out. The discrepancies are limited to the example's use of a variable in both the URL and HTTP request body, whereas the generated models do not include these, due to unsupported features. Additionally, the name of the last step differs, but this does not impact the workflow's behaviour. In general, a visual inspection reveals noticeable differences in the field's order and content. For a more in-depth understanding of the variations between the two serialised formats, the *Levenshtein Distance* algorithm was applied. An online tool [22] facilitated the comparison between the generated and target workflows. The outcomes produced a *Levenshtein distance* of 616 and a similarity of 76.13% for Amazon. Meanwhile, Google achieved a *Levenshtein distance* of 80 and a similarity of 87.69%. The rendered model demonstrates a median similarity of 81.91% with the expected (example) workflows, signifying a relatively high degree of alignment in the final outcome, but it could be improved.

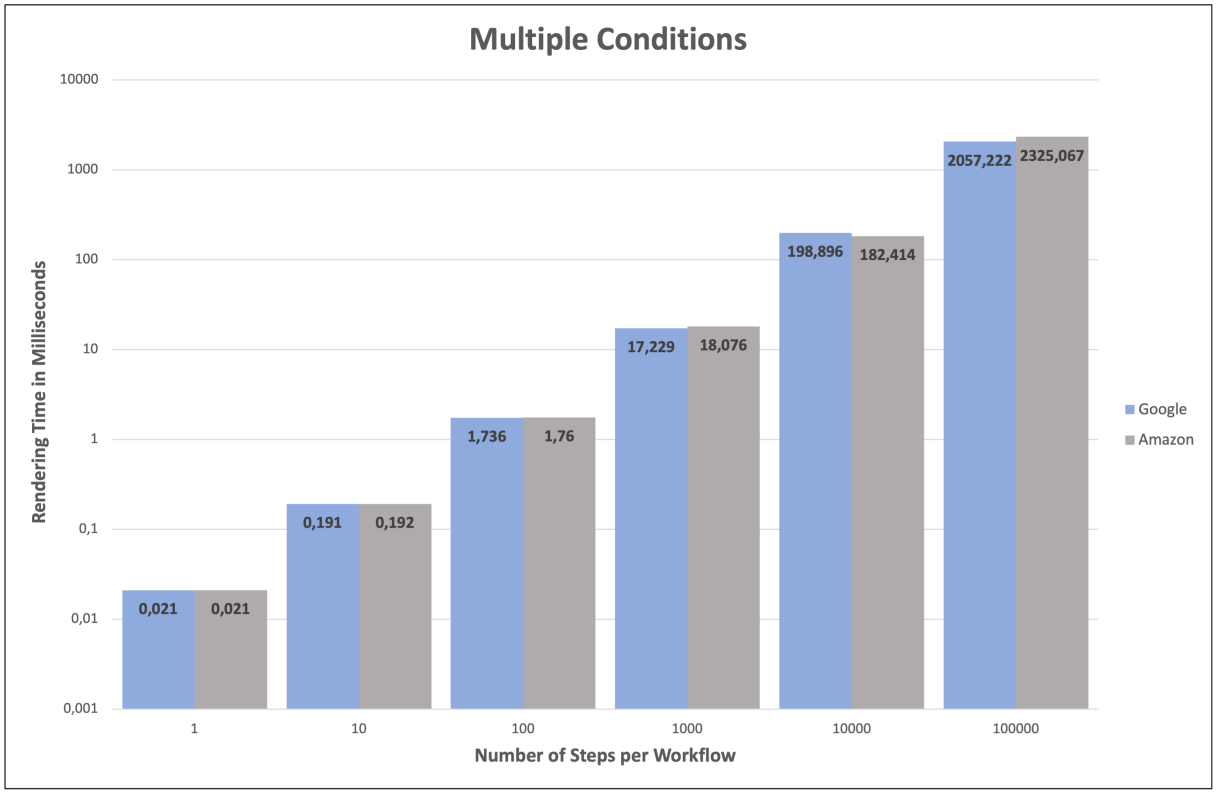


Figure 4.4: Rendering time using multiple conditions

Moving from the qualitative aspect to the quantitative aspect, in terms of the time required to deploy the workflow in the cloud, both Google and Amazon showed small differences in the deployment time of the sample and generated workflows. In the case of Google, there was a minimal difference of 26,47 milliseconds between the target and the created, while Amazon obtained a difference of 43,18 milliseconds. This leads to the conclusion that the difference between the workflows generated by OmniFlow and the example workflows obtained from the documentation does not have a significant impact on the final deployment times. As you might expect, the automatically generated workflows require more time to deploy in the cloud, due to their larger size. In addition, it can be concluded that the deployment process on Google's platform takes longer than on Amazon. All these results and conclusions are present in Figure 4.5.

What this study reveals is, the acquired results underscores a good performance, even under demanding circumstances. The maximum execution time reaches approximately 2,8 seconds under the worst-case conditions, while for a typical use case, such as workflow with ten steps, the execution time peaks at 0,207 milliseconds. In the realm of rendering, it is apparent that the call steps bear the highest processing cost, succeeded by conditions, and ultimately assign steps, which exhibit the most efficient processing times. As anticipated, the rendering process's quality is influenced by the volume of

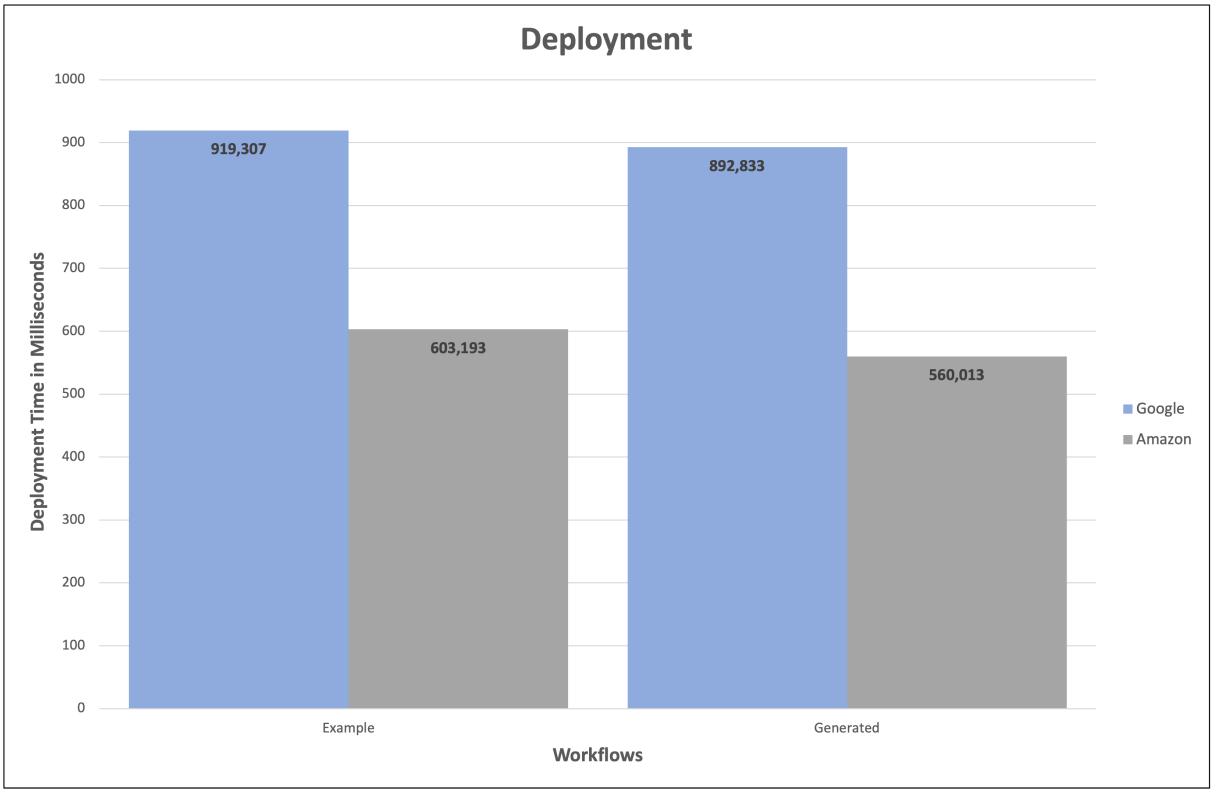


Figure 4.5: How the differences between two workflows affect deployment time

fields contained within each step. The larger the number of attributes, the more extended the execution time is. Importantly, the quality of the generated steps is also impacted by the fact that certain operations are not supported within **OmniFlow**. This factor contributes to variations in the quality of the outcomes.

5

Conclusions

Our goal was to propose a solution to support the developers in defining and deploying function-based workflow to different FaaS platforms. With this goal defined, the first step started with understanding the problem domain, in order to, design an architecture that fits the requirements of the problem. The general software components and their relationships were then established as a result of the diagrams in C4 model format. Their implementation was later realised in a Kotlin library, which we called OmniFlow, that provides a DSL for creating function composition. Omniflow makes it possible to create workflows and deploy them on any FaaS platform, without the need to make any changes. In this way, the same workflow can be deployed on any cloud provider supported by the library. To use Omniflow, all you have to do is add the maven dependency to the project, and all its features become available. Its interface, the DSL, has been designed to be intuitive and easy to use.

In addition, we also provided an evaluation of the solution's performance, using four workflow use cases, with multiple steps. The study showed good results in the times obtained regarding the rendering process, indicating that the size of a step has an impact on time. The step that generally takes the longest to render is Call, and the one that takes the least time is Assign for simple cases.

All the project's development, deployment, and benchmarks components, are publicly available in a GitHub repository [29]. Linked to the repository is a GitHub project where you can see all the decisions made organised chronologically, and new tasks including future work. Documentation on how to use the library is also included

in the form of Markdown. Being an open-source project allows other developers to contribute to improving any topic of the OmniFlow, documentation, implementation, tests, benchmarks, and more.

5.1 Aspects for Improvement

During the development of the project, several drawbacks were encountered that impacted various aspects of the library, in terms of, cross-cloud deployment and the creation of workflows. It is essential to highlight these drawbacks, as they influence the scope and perception of general workflow solutions. Omniflow presents some challenges that have to be overcome and limitations brought by solutions decoupled from the cloud. These can be found below:

Limited Cloud Provider Support At the moment, Omniflow only supports deployment on two of the best-known providers, AWS and GCP. Users are limited to these two platforms, as there is no support for other cloud platforms. The greater the support for multiple cloud providers, the less dependence on the provider.

Limited Advanced Features Some of the functionalities offered by cloud providers are so specific that OmniFlow does not support them, for example, Google's subworkflows. In this way, the features of Omniflow exposed by DSL are somewhat restricted to more general cloud provider solutions. Thus, in certain use cases, the user has to interact directly with the services provided by the cloud providers, bypassing the abstraction layer provided by Omniflow.

SDK dependency Since Omniflow deploys workflows in the cloud, is required a component responsible for deploying the workflow in a specific cloud provider. The software was therefore built to use the SDKs, made available by Google, Amazon, and future platforms. However, when the need to support a new platform that does not provide an SDK emerges, it is recommended the create a separate project with this responsibility and integrate it with OmniFlow.

Quality in Workflow creation Although the library is excellent for simplifying the implementation process, it does not guarantee the correctness and error-free execution of the workflows. This places significant responsibility on the user to design workflows that are free of logical errors, inaccuracies, and potential problems.

Even though OmniFlow simplifies the development and deployment of workflows across multiple clouds, these disadvantages must be carefully considered in the context

of the specific needs and constraints of cloud computing. Additionally, it is essential to note that these drawbacks must be taken into account for future work.

5.2 Future Work

As future work several new features could be realised, starting with the addition of new supported platforms, prioritising Azure. Supporting new types of steps, such as iteration and parallelism, would be a direction to take. Since this study is a continuation of QuickFaaS, it would be interesting to integrate it with OmniFlow and support the deployment of cloud functions in isolation. Other topics that would be interesting to add to OmniFlow would be: creating a validation layer during the creation of a workflow, having a low-code visualisation interface, and forecasting the cost of executing a workflow for each cloud provider. Not forgetting the creation of a pipeline for CI/CD, to facilitate the delivery process.

References

[1] Gojko Adzic and Robert Chatley, "Serverless computing: Economic and architectural impact", in *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, 2017, pages 884–889.

[2] *Credential providers*, https://docs.aws.amazon.com/sdk-for-kotlin/latest/developer-guide/credential-providers.html, Accessed: 2023-07-22.

[3] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, *et al.*, "Serverless computing: Current trends and open problems", *Research advances in cloud computing*, pages 1–20, 2017.

[4] Ioana Baldini, Perry Cheng, Stephen J Fink, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Philippe Suter, and Olivier Tardieu, "The serverless trilemma: Function composition for serverless computing", in *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2017, pages 89–103.

[5] Urmil Bharti, Deepali Bajaj, Anita Goel, and SC Gupta, "Sequential workflow in production serverless faas orchestration platform", in *Proceedings of International Conference on Intelligent Computing, Information and Control Systems: ICICCS 2020*, Springer, 2021, pages 681–693.

[6] *C4 model*, https://c4model.com/, Accessed: 2023-09-15.

[7] *What are durable functions?*, https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp, Accessed: 2023-01-22.

[8] Simon Eismann, Johannes Grohmann, Erwin Van Eyk, Nikolas Herbst, and Samuel Kounev, "Predicting the costs of serverless workflows", in *Proceedings of the ACM/SPEC international conference on performance engineering*, 2020, pages 265–276.

[9] Pedro Rodrigues, Filipe Freitas, and José Simão, "Quickfaas: Providing portability and interoperability between faas platforms", *Future Internet*, vol. 14, no. 12, 2022, ISSN: 1999-5903. DOI: 10.3390/fi14120360. [Online]. Available: https://www.mdpi.com/1999-5903/14/12/360.

[10] *Cloud functions*, https://cloud.google.com/functions, Accessed: 2022-12-22.

[11] *Workflows*, https://cloud.google.com/workflows, Accessed: 2022-12-22.

[12] Diimitrios Georgakopoulos, Mark Hornick, and Amit Sheth, "An overview of workflow management: From process modeling to workflow automation infrastructure", *Distributed and parallel Databases*, vol. 3, pages 119–153, 1995.

[13] *How application default credentials works*, https://cloud.google.com/docs/authentication/application-default-credentials, Accessed: 2023-07-22.

[14] KM Hasan and Mohammad Sabbir Hasan, "A parsing scheme for finding the design pattern and reducing the development cost of reusable object oriented software", *arXiv preprint arXiv:1006.1182*, 2010.

[15] Shuai Jiang and Huaxin Mu, "Design patterns in object oriented analysis and design", in *2011 IEEE 2nd International Conference on Software Engineering and Service Science*, IEEE, 2011, pages 326–329.

[16] Qingye Jiang, Young Choon Lee, and Albert Y Zomaya, "Serverless execution of scientific workflows", in *Service-Oriented Computing: 15th International Conference, ICSOC 2017, Malaga, Spain, November 13–16, 2017, Proceedings*, Springer, 2017, pages 706–721.

[17] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht, "Occupy the cloud: Distributed computing for the 99%", in *Proceedings of the 2017 symposium on cloud computing*, 2017, pages 445–451.

[18] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, *et al.*, "Cloud programming simplified: A berkeley view on serverless computing", *arXiv preprint arXiv:1902.03383*, 2019.

[19] Tomaž Kosar, Pablo E Martı, Pablo A Barrientos, Marjan Mernik, *et al.*, "A preliminary study on various implementation approaches of domain-specific language", *Information and software technology*, vol. 50, no. 5, pages 390–405, 2008.

[20] JetBrains. "Kotlin". (), [Online]. Available: https://kotlinlang.org/. (accessed: 24.02.2023).

[21] *Type-safe builders*, https://kotlinlang.org/docs/type-safe-builders.html, Accessed: 2023-06-05.

[22] *Levenshtein distance*, https://awsm-tools.com/levenshtein-distance, Accessed: 2023-07-22.

[23] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang, "Cloudcmp: Comparing public cloud providers", in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, 2010, pages 1–14.

[24] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo, "Faasflow: Enable efficient workflow execution for function-as-a-service", in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pages 782–796.

[25] Zijun Li, Linsong Guo, Jiagan Cheng, Quan Chen, BingSheng He, and Minyi Guo, "The serverless computing survey: A technical primer for design architecture", *ACM Computing Surveys (CSUR)*, vol. 54, no. 10s, pages 1–34, 2022.

[26] Pedro García López, Aitor Arjona, Josep Sampé, Aleksander Slominski, and Lionel Villard, "Triggerflow: Trigger-based orchestration of serverless workflows", in *Proceedings of the 14th ACM international conference on distributed and event-based systems*, 2020, page 1.

[27] Theo Lynn, Pierangelo Rosati, Arnaud Lejeune, and Vincent Emeakaroha, "A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms", in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, IEEE, 2017, pages 162–169.

[28] *Mockk*, https://mockk.io/, Accessed: 2023-07-28.

[29] *Omniflow*, https://github.com/costaber/omni-flow, Accessed: 2023-09-01.

[30] CNCF. "Synapse". (), [Online]. Available: https://github.com/serverlessworkflow/synapse. (accessed: 13.02.2023).

[31] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi, "Serverless computing: A survey of opportunities, challenges, and applications", *ACM Computing Surveys*, vol. 54, no. 11s, pages 1–32, 2022.

[32] *Strikt*, https://strikt.io/, Accessed: 2023-07-28.

[33] Arie Van Deursen and Paul Klint, "Domain-specific language design requires feature descriptions", *Journal of computing and information technology*, vol. 10, no. 1, pages 1–17, 2002.

[34] Erwin Van Eyk, Alexandru Iosup, Simon Seif, and Markus Thömmes, "The spec cloud group's research vision on faas and serverless architectures", in *Proceedings of the 2nd International Workshop on Serverless Computing*, 2017, pages 1–4.

[35] Erwin Van Eyk, Alexandru Iosup, Cristina L Abad, Johannes Grohmann, and Simon Eismann, "A spec rg cloud group's vision on the performance challenges of faas cloud architectures", in *Companion of the 2018 acm/spec international conference on performance engineering*, 2018, pages 21–24.

[36] Erwin Van Eyk, Johannes Grohmann, Simon Eismann, André Bauer, Laurens Versluis, Lucian Toader, Norbert Schmitt, Nikolas Herbst, Cristina L Abad, and Alexandru Iosup, "The spec-rg reference architecture for faas: From microservices and containers to serverless platforms", *IEEE Internet Computing*, vol. 23, no. 6, pages 7–18, 2019.

[37] Eelco Visser, "Webdsl: A case study in domain-specific language engineering", *Generative and Transformational Techniques in Software Engineering II: International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers*, pages 291–373, 2008.

[38] *What is azure*, https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-azure, Accessed: 2023-01-22.

[39] Vladimir Yussupov, Uwe Breitenbücher, Frank Leymann, and Christian Müller, "Facing the unplanned migration of serverless applications: A study on portability problems, solutions, and dead ends", in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, 2019, pages 273–283.

[40] Haidong Zhao, Zakaria Benomar, Tobias Pfandzelter, and Nikolaos Georgantas, "Supporting multi-cloud in serverless computing", *arXiv preprint arXiv:2209.09367*, 2022.

Workflow Examples

A.1 Amazon Web Services

The Listing A.1 illustrate a workflow example in JSON format.

```
{
  "Comment": "A description of my state machine",
  "StartAt": "Increment1",
  "States": {
    "Increment1": {
      "Type": "Task",
      "Resource": "arn:aws:states:::apigateway:invoke",
      "Parameters": {
        "ApiEndpoint": "ak7u4tmof2.execute-api.us-east-1.amazonaws.
    com",
        "Method": "GET",
        "Path": "/default/increment",
        "QueryParameters": {
          "increment.$": "States.Array(States.Format('{}', $.number))
    "
        },
        "AuthType": "IAM_ROLE"
      },
      "ResultSelector": {
```

```
18          "number.$": "$.ResponseBody"
19        },
20        "Next": "Increment2"
21      },
22      "Increment2": {
23        "Type": "Task",
24        "Resource": "arn:aws:states:::apigateway:invoke",
25        "Parameters": {
26          "ApiEndpoint": "ak7u4tmof2.execute-api.us-east-1.amazonaws.
    com",
27          "Method": "GET",
28          "Path": "/default/increment",
29          "QueryParameters": {
30            "increment.$": "States.Array(States.Format('{}', $))"
31          },
32          "AuthType": "IAM_ROLE"
33        },
34        "ResultSelector": {
35          "number.$": "$.ResponseBody"
36        },
37        "InputPath": "$.number",
38        "Next": "Increment3"
39      },
40      "Increment3": {
41        "Type": "Task",
42        "Resource": "arn:aws:states:::apigateway:invoke",
43        "Parameters": {
44          "ApiEndpoint": "ak7u4tmof2.execute-api.us-east-1.amazonaws.
    com",
45          "Method": "GET",
46          "Path": "/default/increment",
47          "QueryParameters": {
48            "increment.$": "States.Array(States.Format('{}', $))"
49          },
50          "AuthType": "IAM_ROLE"
51        },
52        "ResultSelector": {
53          "number.$": "$.ResponseBody"
54        },
55        "InputPath": "$.number",
```

```
56          "End": true
57        }
58      }
59  }
```

Listing A.1: Amazon Web Services State Machine

B

Workflow Differences

B.1 Google Cloud Platform

The following Listing B.1, and B.2 illustrate a generated Workflow, using Omniflow, and a sample from Google documentation, respectively.

```
main:
  params: [ args ]
  steps:
    - new_translation:
        call: http.post
        args:
          url: https://translation.googleapis.com/v3/projects/
    19823573:translateText
          auth:
            type: OAuth2
          body:
            contents: "Hello, my name is John!"
            sourceLanguageCode: "en-US"
            targetLanguageCode: "ru-RU"
        result: translate_response
    - assign_translation:
        assign:
```

```
17            - translation_result: ${translate_response.body.
    translations[0].translatedText}
18      - return_output:
19          return: ${translation_result}
```

Listing B.1: Generated Workflow using Omniflow based on the example

```
1  main:
2    params: [ args ]
3    steps:
4      - new_translation:
5          call: http.post
6          args:
7            url: ${"https://translation.googleapis.com/v3/projects/"+
     sys.get_env("GOOGLE_CLOUD_PROJECT_NUMBER")+":translateText"}
8            auth:
9              type: OAuth2
10           body:
11             contents: ${args.textToTranslate}
12             sourceLanguageCode: "en-US"
13             targetLanguageCode: "ru-RU"
14         result: translate_response
15     - assign_translation:
16         assign:
17           - translation_result: ${translate_response.body.
    translations[0].translatedText}
18     - return_result:
19         return: ${translation_result}
```

Listing B.2: Google example Workflow

B.2 Amazon Web Services

The following Listing B.3, and B.4 illustrate a generated Workflow, using Omniflow, and a sample from Amazon documentation, respectively.

```
1  {
2    "Comment": "Calling APIGW REST Endpoint",
3    "StartAt": "Add Pet to Store",
4    "States": {
```

```
    "Add Pet to Store": {
      "Comment": "Add Pet to store by calling APIGW Rest endpoint",
      "Type": "Task",
      "Resource": "arn:aws:states:::apigateway:invoke",
      "InputPath": "$",
      "Parameters": {
        "ApiEndpoint": "petstore.execute-api.us-east-1.amazonaws.com"
,
        "Method": "POST",
        "Path": "/pets",
        "RequestBody": {
          "Payload": "$.NewPet"
        },
        "AuthType": "IAM_ROLE"
      },
      "ResultSelector": {
        "ResponseBody.$": "$.ResponseBody"
      },
      "Next": "Pet was Added Successfully?"
    },
    "Pet was Added Successfully?": {
      "Comment": "Checks if the response was not successful",
      "Type": "Choice",
      "Choices": [
        {
          "Variable": "$.StatusCode",
          "NumericEquals": 200,
          "Next": "Retrieve Pet Store Data"
        }
      ],
      "Default": "Notify Result"
    },
    "Retrieve Pet Store Data": {
      "Comment": "Get all data about the pet store",
      "Type": "Task",
      "Resource": "arn:aws:states:::apigateway:invoke",
      "InputPath": "$",
      "Parameters": {
        "ApiEndpoint": "petstore.execute-api.us-east-1.amazonaws.com"
,
```

```
          "Method": "GET",
          "Path": "/pets",
          "AuthType": "IAM_ROLE"
        },
        "ResultSelector": {
          "Pets.$": "$.ResponseBody"
        },
        "Next": "Notify Result"
      },
      "Notify Result": {
        "Comment": "Call Notify Api to notify watchers with the result"
    ,
        "Type": "Task",
        "Resource": "arn:aws:states:::apigateway:invoke",
        "InputPath": "$",
        "Parameters": {
          "ApiEndpoint": "notifyApp.execute-api.us-east-1.amazonaws.com
    ",
          "Method": "POST",
          "Path": "/",
          "RequestBody": {
            "Payload": "Add pet to store status code - $.StatusCode"
          },
          "AuthType": "IAM_ROLE"
        },
        "ResultSelector": {
          "Notification.$": "$.ResponseBody"
        },
        "End": true
      }
    }
}
```

Listing B.3: Generated Workflow using Omniflow based on the example

```
{
  "Comment": "Calling APIGW REST Endpoint",
  "StartAt": "Add Pet to Store",
  "States": {
    "Add Pet to Store": {
      "Type": "Task",
```

```
7         "Resource": "arn:aws:states:::apigateway:invoke",
8         "Parameters": {
9           "ApiEndpoint": "petstore.execute-api.us-east-1.amazonaws.com"
    ,
10          "Method": "POST",
11          "Path": "/pets",
12          "RequestBody": {
13            "Payload": "$.NewPet"
14          },
15          "AuthType": "IAM_ROLE"
16        },
17        "Next": "Pet was Added Successfully?",
18        "ResultSelector": {
19          "ResponseBody.$": "$.ResponseBody"
20        }
21      },
22      "Pet was Added Successfully?": {
23        "Type": "Choice",
24        "Choices": [
25          {
26            "Variable": "$.StatusCode",
27            "NumericEquals": 200,
28            "Next": "Retrieve Pet Store Data"
29          }
30        ],
31        "Default": "Notify Result"
32      },
33      "Retrieve Pet Store Data": {
34        "Type": "Task",
35        "Resource": "arn:aws:states:::apigateway:invoke",
36        "Parameters": {
37          "ApiEndpoint": "petstore.execute-api.us-east-1.amazonaws.com"
    ,
38          "Method": "GET",
39          "Path": "/pets",
40          "AuthType": "IAM_ROLE"
41        },
42        "Next": "Notify Result",
43        "ResultSelector": {
44          "Pets.$": "$.ResponseBody"
```

```
45          }
46        },
47        "Notify Result": {
48          "Type": "Task",
49          "Resource": "arn:aws:states:::apigateway:invoke",
50          "Parameters": {
51            "ApiEndpoint": "notifyApp.execute-api.us-east-1.amazonaws.com
      ",
52            "Method": "POST",
53            "Path": "/",
54            "RequestBody": {
55              "Payload": "Add pet to store status code - $.StatusCode"
56            },
57            "AuthType": "IAM_ROLE"
58          },
59          "End": true,
60          "ResultSelector": {
61            "Notification.$": "$.NotificationStatus"
62          }
63        }
64      }
65  }
```

Listing B.4: Amazon example Workflow