Università degli Studi di Roma "Tor Vergata"

MACROAREA DI INGEGNERIA
Corso di Laurea Magistrale in Ingegneria Informatica

A QoS-Aware Broker for Multi-Provider Serverless Applications

Studente:
Andrea Graziani
Matricola 0273395

Docente:
Prof.ssa Valeria Cardellini

Correlatore:
Dott. Gabriele Russo Russo

Anno Accademico 2020–2021

?????????????????????? dedica da fare

Abstract

In recent years, due to their high scalability, ease of resource management, and pay-as-you-go pricing model, the *Function-as-a-Service* (FaaS) service model has significantly increased its popularity, allowing cloud users to decompose their applications into state-less and standalone computational units called *serverless functions*.

Exploiting several virtualization techniques, serverless functions executions take place on inside isolated environments called *function instances* that run on a FaaS platform provider, which take over all administration tasks regarding serverless function management, such as function deployment, resource management, scaling, and monitoring.

However, to run serverless functions, developers have to configure multiple parameters, such as memory, CPU, cloud provider, etc. However, to find a suitable configuration guaranteeing the meeting of *quality of service* (QoS) constraints can be very difficult.

In fact, since multiple logically equivalent serverless functions can exist, every of which exhibit different performances and cost, developers have to select the most appropriate one in order to respect aforementioned constraints. The status of the FaaS platform, where the function will be executed, need to be considered in order to manage the problem of *cold starts*, which cause a remarkable slowdown compared to native execution. Unfortunately, the situation can become quite more complicated when dealing with serverless applications; their more complex workflows, made up of multiple parallels, branches and loops, makes more difficult the performances estimation of a given application, making the selection of an appropriate configuration for all serverless functions, capable to respect QoS constraints, not trivial.

The aim of this work is focused on the develop of an analytical model able to estimate the average end-to-end response time and the charged cost of a given

serverless applications, providing a methodological way to determine all its serverless function configurations to meet user specified QoS constraints.

Existing solutions are unaware of the current status of FaaS platforms making them uncapable to deal with cold stars or with unavoidable delays which can occur when a FaaS platform is too overloaded. Moreover, they are incapable to deal with multiple FaaS provider simultaneously or to exploit the existence of multiple semantically and logically equivalent serverless functions that expose different performance behaviors.

Therefore, we developed both an analytical model and a software tool to overcome aforementioned limitations, finding the "*best*" configuration for all serverless functions belonging to a given application. This is done by solving an optimization problem, that is a linear programming problem (LP), derived from our model. Experiments conducted using our prototype reveal the effectiveness of our methodology in finding a configuration respecting QoS constraints.

Due to NP-hardness of the optimization problem, to guarantee a rapid system response, we propose a custom heuristic algorithm based on *Ant Colony Optimization* (ACO) algorithms family. Using our prototype, we conduct several experiments in order to show both the accuracy and limits of the heuristic approach.

Finally, to reduce the impact of vendor lock-in issues, we developed our prototype to interact with serverless functions hosted on multiple FaaS platforms exploiting an unique representation scheme for serverless applications. Moreover, exploiting a REST architectural style, we provide identical operations, based on standards `HTTP` method, through which users can easily manage their serverless applications although their functions are hosted on multiple providers, improving access transparency and reducing migrations cost.

Index terms— Serverless Paradigm; Quality of Service (QoS); Cost and Performance Modeling; Cost and Performance Optimization; Multi-Dimensional Multi-Choice Knapsack Problem; Ant Colony Optimization; State-Aware, Multi-Provider and Multiple Serverless Function Implementation Orchestrator; REST Architecture

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

In recent years, thanks to an increased popularity of several lightweight virtualization solutions, specifically containers and container-orchestration systems, many public cloud providers have launched a new set of cloud computing services belonging to new service type category called *Function-as-a-Service* (*FaaS*), where almost every aspect of the system administration tasks, needed to deploy a workload on the cloud, like provisioning, monitoring, resource management and scaling, are directly managed by the provider with a minimum involvement of the user.

With the development of FaaS platforms, the *serverless computing paradigm*, a new applications service model according to which cloud application are abstracted as a group of so-called *serverless functions*, hosted and orchestrated by FaaS platforms, has emerged.

A very important advantage of serverless computing is a new simplified programming model, according to which developers can focus on logic and business aspects of their applications only, without worrying about the server management, which can lead to a development cost reduction, decreasing go-to-market time of the software products.

Moreover, since many FaaS provider have adopted a very small-granularity billing pricing model, usually called "*pay-as-you-go*" model, by charging for serverless function execution time rather than for allocated resources, users can save costs. In simple terms, since serverless users pay only while their code executes and, included in the price, FaaS providers take care for several tasks that need to be provided sepa-

rately in a serverful context, like provisioning, redundancy, availability, monitoring, logging, and automated scaling, some studies claim that, in practice, customers see cost savings of 4×-10× when moving applications to serverless [38].

Serverless computing is an active area of development and, as its scope and popularity expands, especially due to the fast emerging IoT applications for which serverless computing has proved to be a very good fit [33], we believe that is very important to develop both a valid analytical model and methodology to determine serverless application performance; therefore, detailing them is the goal of our work.

1.1 Serverless Functions

A serverless function represents a stateless, event-driven, self-contained unit of computation implementing a business functionality.

Although a serverless function generally corresponds to a unit of executable code submitted to FaaS platforms by developers using one or a combination of the programming languages supported by FaaS providers, it can represent any cloud services eventually necessary to business logic, like cloud storage, message queue services, pub/sub messaging service etc.

A serverless function can be triggered through events or `HTTP` requests following which the FaaS provider takes care of its execution inside a containerized environment called *function instance*. Latter are isolated environments acting as tiny servers completely managed by the serverless computing platform, provided by virtualization solutions such as virtual machines, containers, unikernels or even processes. To invoke a function on any public FaaS platform, cloud users have to specify so-called *serverless function configuration*, which include several parameters, like timeout, memory size or CPU power [19].

We can identify three states for each function instance:

Initialization State which happens when the infrastructure is spinning up new function instance, which is needed to handle incoming requests.

A function instance will remain in the initializing state until it is able to handle incoming requests. According to FaaS policies, the time spent in this state is not billed.

Idle State After the fulfillment of all initialization tasks or when the processing of a previously received serverless function invocation request is over, the serverless platform keeps a function instances in idle state.

In that way, the FaaS provider keep aforementioned function instance able to handle future invocation request faster, since no initialization task is needed to be performed.

However, FaaS platform provider keeps a function instance in idle state for a limited amount of time; after that, all resources used to execute the function instance will be deallocated.

The user is not charged for an instance that is in the idle state.

Running State When an invocation request is submitted to an function instance, the latter goes into the running state, according to which aforementioned request is parsed and processed.

Clearly, the time spent in the running state is billed by the provider.

In order to be clear, we will use the expression *warm pool* when referring to the set of all function instances whose state is either idle or running state.

Despite the most basic scenario is represented by the invocation of a single function, when a more complex application is needed, serverless functions can be connected and appropriately orchestrated in a *workflow*, obtaining a so-called *serverless application*.

We define a serverless application as a stateless and event-driven software system made up of a serverless functions set hosted on one or more FaaS platforms and combined together by a so-called *coordinator* (or *orchestrator*). The latter is usually represented by a broker, needed to implement the business logic of any application; it chains together serverless function, handles events among functions and triggers them in the correct order according to the business logic defined by developers.

Nowadays, there are many public cloud platforms which provide serverless computing services to deploy and execute serverless functions, including AWS Lambda [1], IBM Cloud Function [9], Microsoft Azure Functions [3] and Google Cloud Functions [6]. Moreover, some providers have even introduced FaaS platforms acting as coordinator to build serverless application; AWS Step Functions is an example

which allows to combine multiple Lambda functions, including other serverless services offered by AWS, like the storage service Amazon S3, into an application.

Finally, in contrast to common cloud architectures that expect long-running applications, another very important feature of serverless functions is that they run typically for very short time. For example, in 2022, the maximum execution time for an AWS Lambda function is 15 minutes [2].

1.1.1 FaaS Auto-Scaling Technique

As already said previously, according to serverless computing paradigm, the FaaS platform provider has the responsibility to manage the amount of function instances required to allow users to perform their computation.

To be more precise, to match the rate of function invocations, the platform automatically scales the number of function instances available to execute developers functions in parallel; we call this activity *auto-scaling*.

Despite many auto-scaling techniques exist in literature [31], in this dissertation, we assume that all FaaS providers adopt an auto-scaling technique called *scale-per-request*. If such technique is adopted, when a request comes in, only one of the following events can occur:

Warm start if there is at least one function instance in idle state, the FaaS platform reuses it to serve the incoming request without launching a new one.

Cold start If the warm pool is empty or busy, that is there are no serverless function instances in idle state able to serve an newly incoming request, FaaS platform will trigger the launching of a new function instance, which will be added to the warm pool.

As said previously, from the FaaS provider point of view, this operation requires the start of a virtualized and isolated execution environment (i.e. virtual machine, container and so on) where user code will be run; in any case, regardless of the virtualization solution adopted, a cold start introduces a very important overhead to the response time experienced by users.

Therefore, due to initialization tasks performed to spin up a new function instance, cold starts could be orders of magnitude longer than warm starts for

some applications; therefore, too many cold starts could impact the application responsiveness and user experience.

As long as, for a given instance, requests are received within an interval time less than an *expiration threshold*, the function instance will be not deallocated.

At the same time, for each instance, at any moment in time, if a request has not been received in the last *expiration threshold* units of time, it will be expired and thus terminated by the platform, and the consumed resources will be released.

This technique is currently adopted by the vast majority of well-known public serverless computing platforms, like AWS Lambda, Google Cloud Functions, IBM Cloud Functions, Apache OpenWhisk and Azure Functions [31].

However, despite in most cases function instances are actually terminated when the timeout expires, in case of need, the provider could terminate some instances in advance to free up resources; therefore, there are no guarantees about function instance permanence in the warm pool within expiration threshold units of time.

1.1.2 FaaS Request Routing

In this dissertation, we assume that, in order to minimize the number of function instances that are kept warm and thus to free up system resources, the FaaS routes requests giving priority to instances which they have been idle for less time.

In other words the FaaS request routing gives low priority to function instances being in idle state for long time, increasing thus the chances of resource deallocation referring aforementioned instances.

1.1.3 Concurrency Limit

Any FaaS platform imposes a limitation on the number of serverless function instance runnable at the same time; this limit is generally known as *concurrency level* or *concurrency limits*.

Clearly, this kind of limitation is needed to assure enough resources for all users using the services provided by FaaS platform. However, despite all FaaS providers impose aforementioned limitation, these restriction are applied differently.

Informally speaking, there are two type of concurrency limit models:

Global (Per-Account) Concurrency Model according to which invocation threshold is shared by all serverless functions belonging to a given resource owner.

For example, in 2022, this approach is adopted both by AWS Lambda and IBM Cloud Functions, which do not allow more than 1000 serverless concrete function in running state at the same time.

Local (Per-Function) Concurrency Model where, opposed to global concurrency model, any invocation threshold is applied only on individual concrete functions.

This approach is adopted by Google Cloud Functions [1]

1.2 Limitations of Today's FaaS Platforms

Although serverless computing paradigm makes cloud application developing easier guaranteeing a cost-effective solution, there are several obstacles that prevent FaaS platforms to support more general workloads, especially those that must meet strict guarantees.

Scientific literature have already reported a huge amount of limitations preventing its worldwide adoption, some of which are:

- the lack of support for those applications having fine-grained state sharing needs, which is primarily due to stateless nature of serverless platforms [28].
- too simple scheduling policies which, being mainly based on first-come-first-served algorithms, limit how serverless function have to be managed in scenarios such as when incoming demands cannot be satisfied by currently available resources [41].
- very unpredictable performance due to multiple factors, including function placement, *cold starts*, namely the delay incurred to get a function instance up and running when it is invoked for the first time within a defined period, I/O and network conditions, type of VMs/containers, variability in the hardware and co-location with other functions [19][28].

[1] Despite there is no explicitly mentioned global concurrency limit, previous studies have observed a kind of global concurrency limit varying between 1000 and 2000.

- security concerns due to fine-grained resource sharing of serverless function which increases the exposure to *side-channel* attacks [38]

- Since serverless functions are short-lived, FaaS Providers must rely on the fine-grained interleaving of many short functions to achieve high throughput.

  According to Shahrad et al. [39], this situation causes temporal locality-exploiting structures, like branch predictors, to underperform, raising questions about present-day computer architectures' ability to execute FaaS efficiently. For instance, Shahrad et al. [39] observed a 20× increase in branch mispredictions per kilo-instruction (MPKI) when comparing shortest to longer functions executions.

However, one of the most important obstacles concerns the *quality of service* (*QoS*) levels, quantitatively measured considering charged cost and response time, that should be guaranteed when a serverless application is executed.

The main difficulty relating to QoS constraints fulfillment depend mostly on the lack of an adequate performance model, which is crucial to analyze and predict response times and billed costs for any generic workload serverless application.

To be precise, the aforementioned model is needed to select a configuration for each serverless function belonging to a given application. In fact, as we said previously, when cloud application developers want to run a serverless function, is necessary to specify several configuration parameters regarding its execution environment, like memory size, time limits or CPU power.

Unfortunately configuring such parameters correctly, meeting cost and response time constraints, is not trivial. In fact, several studies have already shown that aforementioned configuration parameters significantly affect the cost and response time of serverless functions; for instance, Akhtar et al. [19] have shown that function response time decreases when the memory size allocated to the function is increased; however, due to pricing models adopted by the vast majority of FaaS providers, according to which cost depends proportionally on the amount of memory allocated to a serverless function, a too large value for memory can result in higher costs. Conversely, a small value of memory size can lead to higher costs too, due to a longer execution time of the functions. Moreover, a decrease of the marginal improvement in response time as the memory increases is observed.

Despite solutions to that problem already exist in scientific literature, we believe that they suffer from a major limitation, as they limit their focus only on serverless function configurations parameters; in fact, none of existing performance models and frameworks, when they have to select a suitable configuration to fulfill QoS requirements, considers the existence of multiple implementations, or versions, of a same serverless function, that usually exhibit different performances and cost. Sometimes, depending on QoS requirements, can be useful to select a particular implementation with respect to another. In this dissertation, we will use the term *concrete serverless function*, or simply *concrete function*, to refer to semantically and logically equivalent function implementations, accepting the same set of input data and returning the same type of output data. However, they may expose different performance or cost behavior since they may be implemented with different programming languages or algorithms and they may be hosted on different FaaS platforms having different performance capabilities.

Similarly, no existing solution is able to manage serverless application whose functions are hosted on multiple FaaS platforms. Proposed solutions are generally unaware of the current status of FaaS platforms and they cannot adapt themselves in response to changes in their execution environments. Therefore, they are incapable to prevent cold starts or the unavoidable delays which usually occur when a FaaS platform is too overloaded or to select the provider having the best cost/performance ratio in order to better meet customer's expectations.

However, exploiting multiple FaaS platform in order to meet QoS levels is not trivial. Unfortunately, despite FaaS platforms continuously advance the support for serverless applications, existing solutions are dominated by a few large-scale public providers resulting in mostly non-portable FaaS solutions, using a variety of conventions for creating, executing, logging and monitoring serverless applications and its functions, exacerbating the so-called vendor lock-in issue, a very common problem in distributed systems. There is no even agreement on how a serverless application is to be represented among different FaaS providers, therefore shift to another vendor's platform can be difficult and costly, making customers more dependent on a single FaaS solution.

Therefore, the aim of our work is to overcome aforementioned limitations building a QoS-aware framework for the orchestration of serverless functions belonging

to generic serverless workflow, supporting multiple FaaS providers and functions implementations.

1.3 Contributions

Our contributions can be summarized as follows:

- We propose a formal definition of the serverless application workflow, abstracting sequences of serverless functions, including parallel, branch and conditional loop structures.
- We present a FaaS platform-aware performance model in order to evaluate and predict serverless application performance under a certain configuration in term of economic cost and response time.
- We propose a methodological way and a software tool to find the "*best*" serverless application configuration in order to satisfy user specified QoS requirements. This is achieved by solving an optimization problem, that is a linear programming problem (LP), derived from our performance model.
- We develop a software tool capable to run serverless functions hosted on multiple FaaS platforms. In contrast to existing approaches, our framework can exploit the differences between the various FaaS providers (performance, billing methodology etc.) to better meet customers expectations. Moreover, our tool is able to exploit multiple serverless concrete functions to achieve its goal.
- Owing to the complexity of the problem, since can be very difficult to find an optimal configuration for a serverless applications that guarantees the fulfillment of customer's constraints for very big applications, hence discouraging the adoption of serverless paradigm, we present a heuristic approach based on a probabilistic technique belonging to *Ant Colony Optimization* algorithm (ACO) family in order to provide a very rapid system response to our problem; that algorithm is called "*Pre-provisioned Ant Colony Optimization Algorithm with Lazy Pheromone Update*".

- In order to mitigate the impact of vendor lock-in issue, primarily related to the high migration costs paid by users when they wish to change FaaS provider to better meet their needs, an unique way to represent a serverless application, in such a way that very little changes to that representation code are needed to complete any switching process, is adopted.

  Our serverless application representation scheme is primarily based on an existing solution called *Abstract Function Choreography Language* (AFCL) [36], which we have partially adopted and extended in order to make it compatible with our multi FaaS provider system.

- To further reduce the vendor lock-in issue, we develop our own software system acting as orchestrator for serverless application management without relying on existing commercial solution.

- To improve access transparency to FaaS platforms services, hiding any difference regarding how serverless application on different providers can be accessed, we have developed our software system following *Representational State Transfer* (REST) architectural style, allowing our customers to create and access to serverless application, whose function are hosted on any supported provider, using identical operations based on standard `HTTP` methods.

- We partially address the security problem regarding key exchange between our customers and our system broker, necessary to allow our system to access to the FaaS services paid by our users, adopting asymmetric encryption during the creation of a serverless application on our system. Moreover, some of the customer's data is encrypted by default.

1.4 Organization

The remainder of this dissertation is organized as follows. In Chapter 2 we will discuss related works. In Chapter 3 we will present an overview of our framework, outlining its architecture and the main tasks of its components. In Chapter 4 we will describe all mathematical details about our QoS-aware analytical model, included

all equations and algorithms used by our framework to compute average end-to-end response time and cost of any serverless workflow. Based on aforementioned analytical model, in Chapter 5 we will present two different formulations of the optimization problem that must be solved to determine a suitable configuration in order to meet user specified constraints. Moreover we will present a heuristic algorithm to solve one of the aforementioned formulation. In Chapter 6 we will present the experimental evaluation of the aforementioned models and algorithms. Finally, Chapter 7 discusses and concludes our work, presenting directions for future works.

Chapter 2

Related Works

The serverless computing paradigm has attracted a great deal of interest from both academia and industry, therefore in this chapter we provide a short preview of the literature inherently related to our topic, namely performance modeling of serverless applications.

Obviously, several researchers have already proposed some performance and cost models which can be useful to address the aforementioned issue. For instance:

- Mahmoudi and Khazaei [31] proposed a very interesting analytical model to determine both the delay and charged cost for a generic serverless workflow, including a methodology to obtain a configuration for each serverless function belonging to the application in order to respect either delay or cost constraints imposed by a user.

  Their methodology is based on solving two different optimization problems, namely the best performance under the budget constraint and the best cost under the performance constraint. Moreover, researchers proposed a heuristic algorithm named *Probability Refined Critical Path* algorithm with four greedy strategies in order to rapidly solve aforementioned optimization problems.

- Akhtar et al. [19] proposed a statistical learning based configuration finder for serverless functions.

  More precisely, their framework, using bayesian optimization to statistically learn the relationship between cost/run-time, selects a serverless function con-

figuration in order to respect either delay or cost constraints imposed by a user, minimizing the cost.

- Lin and Khazaei [30] presented, according to our opinion, one of the best analytical performance model available today, which is suitable for analyzing and predict performance of current FaaS platforms.

  More precisely, they built an analytical model to compute response time, blocking probability and utilization of a FaaS platform exploiting a semi-Markov $M/G/m/m$ queueing system.

- Jindal et al. [27] provided a very refined solution called *Function Delivery Network* (FDN) which represents a network of distributed heterogeneous target platforms enabling automatic scheduling of functions to target platforms based on their computational and data requirements.

  This solution supports adaptive data management, enabling data migration between target platforms to exploit data affinity exploiting several data staging tactics.

  Their scheduling algorithm considers both performance capabilities and resource-usage of the target FaaS platform. Moreover, Jindal et al. [27]'s solution supports function invocations on multiple target platforms to guarantee QoS level constraints.

- Ristov et al. [35] developed a middleware which allow developers to build and run serverless applications on multiple FaaS systems, including a model which estimates functions response time by considering FaaS system limitations and submission delays too.

  Moreover, they proposed a region-ware scheduling algorithm, based on the earliest finish time approach, to select the region on which every serverless function will be scheduled in order to optimize applications executions.

Despite the high quality of aforementioned solutions, they still have some problems which can sometimes hinder the fulfillment of QoS requirements.

For instance, both performance models proposed by Akhtar et al. [19] and Mahmoudi and Khazaei [31] do not consider the current state of the FaaS platform when

a serverless function configuration has to be selected, totally ignoring the problem of cold starts and their impact on application performance. Conversely, Jindal et al. [27]'s solution takes into account the occurrence of cold starts, while their scheduler tries to reduce them by exploiting their serverless application model.

At the same time, the model proposed by Lin and Khazaei [30], despite being quite good for analyzing and predicting the performance of FaaS platforms, is not suitable to predict the performance for generic workload applications. Similarly, the Akhtar et al. [19] proposed solution is suitable only for sequences, or chains, of serverless functions, therefore useless for generic workloads.

There is no solution which considers the existence of multiple implementations, or versions, of the same serverless function, that usually exhibit different performances and cost. At the same time, only Jindal et al. [27] and Ristov et al. [35] solutions support serverless functions scheduling on multiple FaaS provider platforms, which can be exploited to guarantee the respect of user specified QoS constraints.

Chapter 3

System Architecture and Implementation

Our software is a standalone orchestrator, QoS-aware model-based optimizer and profiler for serverless applications, whose functions can be hosted on multiple FaaS platforms.

From an architectural point of view, according to the applications taxonomy proposed by Satyanarayanan et al. [37], our software system is a *cloud-native application* because the *Tier-1*, which represents "*the cloud*" in today's parlance, is the unique execution site where our application performs its tasks [1].

Our system was designed and implemented according to simple *client-server architecture*, where we recognize two types of processes:

- The *server process* implementing our service.
- The *client processes* which request a service from our server.

Since our system is made up of a centralized component, capable to handle all communications between users and all supported FaaS platform providers, our system acts as a *broker*.

[1] Despite some design and development efforts are still required, from a methodological point of view, there is no reason to constrain the execution of this software on the cloud to achieve the goals of this work. Any future development should consider to build a distributed version of this system, in order to exploit the advantages provided by an edge-accelerated or edge-native solution, like network proximity, ingress bandwidth demand reduction etc.

Clearly, for the purposes of our work, our application has to have all required knowledge about the interfaces adopted by all supported providers regarding serverless functions management; therefore, from an architectural point of view, our system requires a set of *adapters*[44], that is software components having aforementioned knowledge.

As trivial as it sounds, a very important advantage of this design choice is the *access transparency* provided to final users, because they can manage their applications using identical operations forwarded to our system without interacting directly with FaaS providers [44].

Despite being very easy to implement, the adopted centralized approach also has several shortcomings: it is not scalable, it potentially suffers from serious traffic congestion problems and, lastly, it represents a single point of failure [44].

We want to emphasize that, since they are out from our goal, most issues and pitfalls regarding the design of our system, like transparency, processes decoupling, message delivery reliability issues, scalability, traffic congestion, fault-tolerance or replication management, were out of the scope our this work.

3.1 System Components Overview

This section details all software components of our system, which consists of following main logical entities:

- A Logging subsystem.
- An Orchestrator.
- A profiler.
- A QoS-aware model-based optimizer.
- A Custom AFCL Parser

We will give a very brief overview of how these entities work while all details about the model on which they are based will be given in next chapter. The overview of our system is illustrated in Figure 3.1.

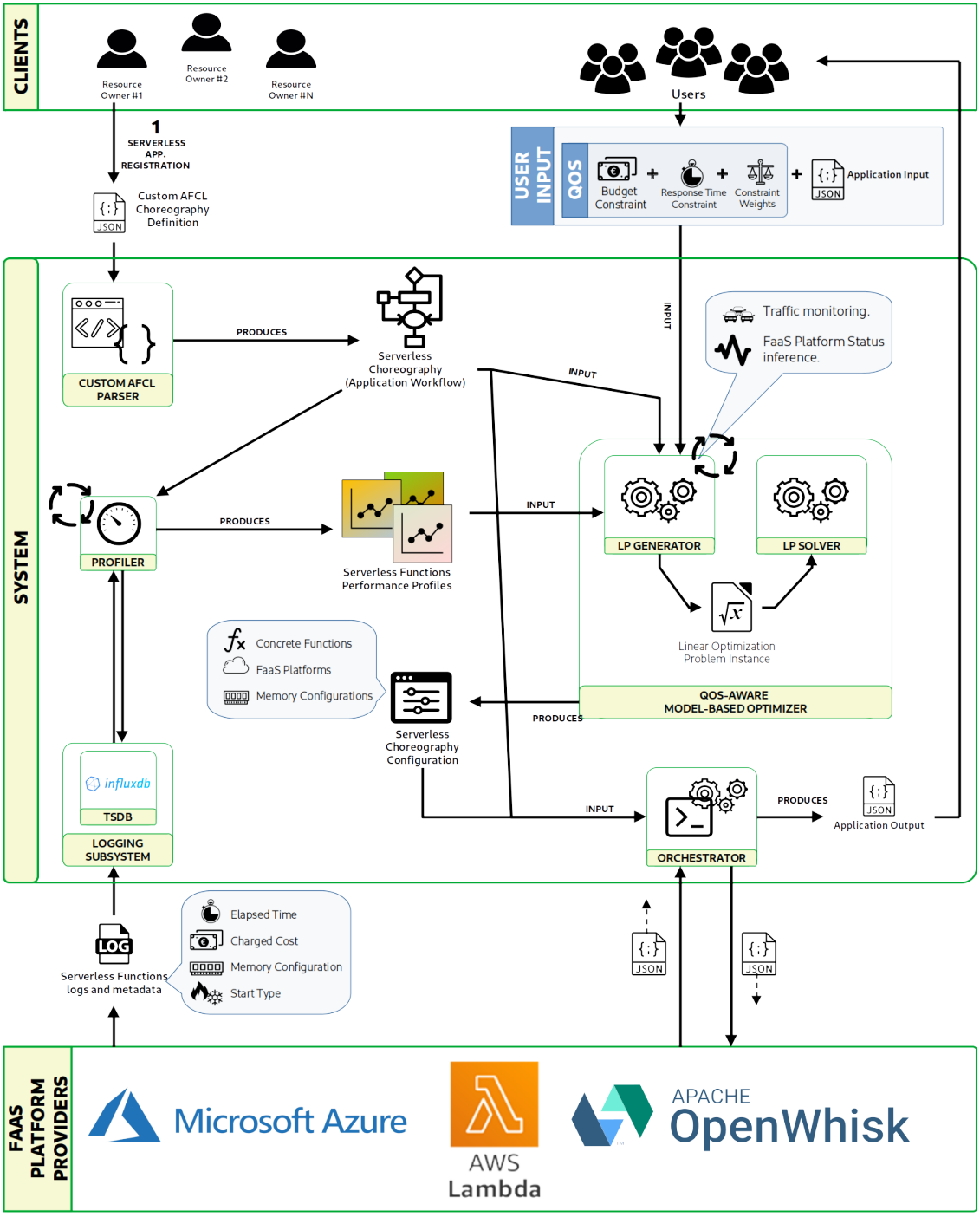

Figure 3.1: A simplified overview of our system, including its components and their interactions.

3.1.1 Logging Subsystem

In order to achieve our goal, a live overview of the state of both all FaaS platforms executions environment and system traffic is required.

More concretely, our framework builds and continuously updates a dataset containing all historical information regarding serverless function performance under any possible configuration.

After the invocation of any serverless function, the corresponding FaaS platform provider produces a log containing several kind of information as:

- The time spent to complete the serverless function execution.
- The charged cost.
- Memory size value.
- The occurrence, or not, of a cold start.

After every serverless function execution completion, FaaS produced logs are parsed and stored by the logging subsystem into an *time series database* (TSDB). Believing that our domain requires the capability to handle thousands of parallel inputs streams, to manage time-indexed data, horizontal scaling and a fast access to stored data, the use of a TSDB was been preferred respect to conventional *Relational DataBase Management System* (RDBMS) [34].

The data contained in FaaS platforms logs *can*[2] be considered as time series data, that is data points indexed in time order. According to our design, each point is uniquely identified by four components:

- A timestamp.
- Zero o more tags, key-value pairs that are used to store metadata associated with the point.
- One or more fields, that is scalars which represent the value recorded by that point.

[2]Strictly speaking they aren't. FaaS platforms use an activation ID to index their logs since two or more serverless functions can start or terminate at the same time. However, according to our design, it is meaningless.

- A *measurement*, which acts as a container used to group together all related points.

According to several TSDB design, the main difference between tags and fields is that the former are indexed, therefore searching by tag is quicker than searching by field; in fact, when a search by field is performed, a scan of entire value list - point after point - is required, making that operation slower on large datasets [34].

All data points, produced by a common FaaS platform provider, are grouped in a measurement, whose name is the same of the corresponding FaaS platform.

At the same time all points, related to all serverless functions owned by an resource owner, are stored inside directory whose name is shared with correlated user one.

Then following tags are used:

`FunctionName` to group together all point related to a unique serveless function.

`MemorySize` to perform a quick search regarding data points having a given memory size.

`StartType` to distinguish if data was collected after a warm or a cold start.

Actually, data points are stored into our TSDB instance using a simple first-come-first-served approach, therefore data point's index coincides with the actual time on our system framework.

With very few exceptions, all iterations involving our logging subsystem are done asynchronously, without blocking current tasks possibly performed by our system.

3.1.2 Orchestrator

The orchestrator, or coordinator, is probably the most important software component of our system, whose aim is to allow the composition of serverless functions in order to properly execute applications.

As already said, the orchestrator is needed to chain together all functions of the application, triggering them in the correct order according to both the business logic and, clearly, the user inputs.

More concretely, the orchestrator is invoked every time a user requests the execution of any serverless application. It mandatory requires a `JSON` input, possibly empty, representing the input data for the invoked application; when application execution ends, a `JSON` object, representing the application output, will be sent to the client.

Each function invocation of any serverless workflow is performed synchronously, which means that the orchestrator waits for the execution of the orchestrated functions to complete before attempting to invoke the next one. Unfortunately, at the current state of our design, if any error occurred during the execution of any orchestrated functions, the entire application execution will be aborted. When the execution of any orchestrated function ends, the orchestrator retrieves log information, mostly regarding performance and billed costs, from live metric streams from FaaS provider logs, which are forwarded, as already said asynchronously, to our logging subsystem.

Concerning workflow state management, state parameters are passed between orchestrated functions, always passing through the orchestrator. Currently, there is no limit on the size of state parameter, however it is expected that overheads related to state passing can significantly grow with increasing parameter sizes.

Currently, our orchestrator subsystem supports *function chaining* and *nesting* exploiting several control-flow structures in order to allow:

- *functions branching*, through the support for some conditional control-flow statements like `if` and `switch`.
- *functions looping*, providing the support for both `for` and `while` loop control-flow statements.

Moreover, the orchestrator supports parallel execution of two or more functions and the possibility to abort current application execution.

Today, all serverless functions orchestration services provided by main public FaaS platforms providers, like *IBM Composer*[10], *Amazon Step Functions*[14], and *Azure Durable Functions*[4], enforce developers to hardcode the specific function implementation to use inside a serverless workflow [36]; this fact prevents the possibility to select a specific function implementation for an optimal execution in order

to meet user specified constraints. Similarly, since FaaS platforms providers do not support the possibility to invoke a serverless function hosted on different FaaS platform, it can be difficult to satisfy user specified QoS levels under some conditions, like high traffic situations towards a specific FaaS platform or resources shortage for functions executions.

To overcome aforementioned limitation, our orchestrator is able to select a specific implementation of a functions inside any serverless workflow; for example, it can run a function that implements the Pollard's rho algorithm for integer factorization, written in Python, instead of a function implementing the trial division based algorithm, written in JAVA. Moreover, aforementioned implementation can be hosted on multiple FaaS platforms.

However, only our QoS-aware model-based optimizer has the responsibility to determine which implementation, for each function belonging to an application, must be invoked in order to meet QoS level constraints. To be more precise, during function orchestration, the orchestrator limits itself to invoke function implementations according to the so-called *choreography*[3] *configuration*, or simply *configuration*.

The choreography configuration, produced by our optimizer, is a sort of table which tells the orchestrator which implementation and configuration must be used for each function in order to meet user specified QoS level constraints; we will talk more in detail about this shortly.

3.1.3 Profiler

When a serverless application is created for the first time, the QoS-aware optimizer has no performance metric data about serverless functions to perform its task since they do not exist yet.

[3]Generally, the terms *orchestration* and *choreography* describe two different designs according to which distribute systems are built.

An orchestration represents a design where a single centralized executable process, i.e. the orchestrator, coordinates the interaction among different services. A choreography employs a decentralized approach for service composition where their interaction is done by exchange of messages according to specified rules and agreements between two or more endpoints [42].

However, in this dissertation, the term choreography is simply used as synonymous of serverless application; in other terms, a choreography has to be intended as composition of two or more serverless functions. At the same time, the term orchestrator has to be intended as a process which is responsible for invoking serverless functions belonging to a given choreography.

Therefore a performance profiling phase in order to acquire a series of function response time and cost under different memory sizes is needed.

According to our design, it is mandatory that the profiling phase for a given serverless application must be executed before its first execution; that phase must be explicitly started by the resource owner.

Synchronously interacting with the logging subsystem, it retrieves all needed performance metrics referring to each implementation and configuration regarding all serverless functions inside the invoked application. Aforementioned metrics include observed average response times and charged cost when both cold and warm starts occurred. Once data are obtained, they are processed according to our performance model, which we will explain in next chapter.

However, performing aforementioned tasks is generally expensive due to several reasons like the huge amount of data, expensive computation and network delay during data transmission between system components. In general, in a real situation, thousands data point must be collected and analyzed. It may happen that the TSDB instance, from which the data is obtained, can be connected to our framework through a lower-bandwidth and multi-hop connection.

Therefore, a *pre-fetching tactic* is used to anticipates data needs in order to minimize latency[29]. Our framework firstly collect and analyze all data points retrieving them from the TSDB, then it stores locally processed data. Data synchronization is performed periodically [4].

Since according to our performance model processed data must used for future invocations, they are stored inside a NoSQL distributed database instance for future use.

3.1.4 QoS-Aware Model-Based Optimizer

As already said, the optimizer has the responsibility to determine which configuration and implementation must be used during the execution of a given serverless application in order to satisfy user specified constraints.

The optimizer is invoked every time a user requests the invocation of a serverless applications; then, only one of the following event can occur:

[4]By default every 15 minutes

- if the user does not specify any QoS level constraint, the optimizer will produce the choreography configuration by selecting configurations and implementations randomly.
- if QoS level constraints are specified, the optimizer performs following activities:
  - It retrieves all the series of function response time and cost under different memory sizes for each serverless functions implementations involved in the current application. These data must have been already computed by our profiler subsystem.
  - The status of all FaaS platforms where serverless functions will be executed is checked, including average arrival rate regarding user requests. All these data are used to compute cold/warm start probabilities in order to build the best configuration to meet QoS levels.
  - An optimization problem, derived from our performance model, is resolved producing a choreography configuration. Then the orchestrator will be subsequently invoked passing to him the produced configuration.

All details about how the optimizer works will be discussed in next chapter.

3.1.5 Custom AFCL Parser

As already said in previous chapter, instead of requiring a separate implementation for each FaaS platform, our framework uses a unique serverless application representation scheme using which very little modifications are needed to switch form a provider to another, partially overcoming portability limitations and vendor lock-in issues.

Aforementioned representation scheme is based on an existing language called *Abstract Function Choreography Language* (AFCL), which is a feature-rich markup language developed to facilitate the creation of serverless workflows encoded in `YAML` or `JSON`. AFCL supports event-based, synchronous and asynchronous invocation of functions and it includes several control-flow constructs in order to construct iterations, branch, sequence or parallel executions inside a serverless workflow [36].

More precisely, in order to create a serverless workflow, all its functions, as well as all control- and data-flow connections among them, must be specified. According to AFCL terminology, a function can be either *base functions* or *compound* functions. The former refers to a single computational task, while the latter encloses some base functions or even nested compound functions.

For example, Listing 3.1 show the definition of a base function in AFCL using a meta-syntax which extends `YAML`, such that `YAML` elements can be contained in { } and appended with wildcards "`?`" (0 or 1), and "`+`"' (1 or more).

Listing 3.1: Definition of a base function in AFCL

```
function: {
        name: "name",
        type: "type",
        dataIns: [
                {
                        name: "name", type: "type",
                        source: "source"?, value: "value"?,
                        properties: [{name: "name", value: "value"}+]?
                        constraints: [{name: "name", value: "value"}+]?
                }+
        ]?,
        properties: [{name: "name", value: "value"}+]?
        constraints: [{name: "name", value: "value"}+]?
        dataOuts: [
                {
                        name: "name", type: "type",
                        source: "source"?, value: "value"?,
                        properties: [{name: "name", value: "value"}+]?
                        constraints: [{name: "name", value: "value"}+]?
                }+
        ]?,
}
```

As you can see from Listing 3.1, base function definition supports both the `dataIns` AFCL object, which is a `YAML`/`JSON` object used to specify input data, and the `dataOuts` AFCL object, used, obviously, to specify output data[36]. Optionally developers can specify properties and constraints for functions.

The number of the objects belonging to both `dataIns` and `dataOuts` depend on the chosen serverless function. These objects have to specify a name and a type while, optionally, they can include a constant value or the data source, which can be specified using the name of another base or compound function within the same

workflow.

To build realistic serverless workflow, AFCL support following functions: `sequence`, `if-then-else`, `switch`, `for`, `while`, `parallel`, and `parallelFor`. The specifications for the `name` attribute, `dataIns` and `dataOuts` ports are similar as for a base function. Anyway, a detailed description of AFCL is available in [36].

Despite quite refined, only a subset of AFCL features are really needed to achieve our goals and, at the same time, AFCL lacks some features which are indispensable to us. Therefore, more concretely, our serverless application representation scheme is based on a custom version of AFCL and to integrate its use in our framework, we have developed our own custom AFCL parser, invoked whenever a serverless application needs to be instantiated from its AFCL representation.

Listing 3.2: Definition of a serverless function according to our custom AFCL.

```
1       function: {
2               name: "name",
3               "implementations":[
4                       {
5                               providerName: "providerName",
6                               concreteFunctionName: "concreteFunctionName"
7                               host: "host"?
8                       }+
9               ],
10              "profilingPayloads": [
11                      {...}+
12              ]
13              dataIns: [
14                      {
15                              name: "name", type: "type",
16                              source: "source"
17                      }+
18              ],
19              dataOuts: [
20                      {
21                              name: "name", type: "type",
22                              source: "source"
23                      }+
24              ]?,
25      }
```

Listing 3.2 shows the definition of a serverless function according to our custom AFCL, which has follows characteristics and differences respect to the original one:

- Only synchronous invocation are supported.

- Neither `parallelFor` or `sequence` statements are supported and recognized by our parser since they are redundant because their effects on the workflow are achievable using existing control-flow statements.
- So-called AFCL event-based invocations, according to which a user can trigger the execution of one o more serverless workflow based on a specified event, is not supported [36].
- Properties and constraints are not supported
- According to our custom AFCL language, in order to allow our users to exploit different implementations of a serverless function, which can be eventually hosted on multiple hosts, any `function JSON` object literal must have the `implementations` field, which is used to specify concrete functions.

  Technically, the `implementations` field is an `JSON` array containing one or more `JSON` object having following fields:

  `providerName` which, clearly, represents the FaaS provider name.

  `concreteFunctionName` which represents the name of the serverless function hosted on the specified FaaS platform.

  `host` which contains the IP address of the FaaS platform provider. Currently, it is required only for providers using Apache OpenWhisk.

- According to our custom language, `function` object has to include another `JSON` key called `profilingPayloads` whose value is a a `JSON` array of `JSON` objects, every of which contains the `JSON` representation of a valid input for the function.

  That statement is used to provide valid input data for the profiling process, which is needed to collect performance metrics about a given serverless function's implementation for the first run.

- Unlike the original one, our custom AFLC language supports the use of the `exit` statement, whose effects are quite similar to `exit` system call inside a classical application. More precisely, the `exit` statement is used to terminate the execution of a serverless application returning a `JSON` output to the user.

Technically, the `exit` is a `JSON` key like any other AFCL key like `function`, `for`, `if`, etc. Its corresponding `JSON` value is a `JSON` object that, like any other AFCL object[36], has to contain the `dataIns` AFCL statement, which is a `JSON` key used to specify input data[36]. Unlike other AFLC objects, when the `exit` statement is used, the `dataOuts` field, used to specify output data[36], is not required since it is assumed equal to `dataIns`.

Despite the `exit` statement can be used anywhere in a serverless workflow, due to several limitations regarding both our implementation and performance model, its use is not allowed inside a parallel block.

3.2 Architectural style

Our system was been developed following the *Representational State Transfer* (REST) software architectural style, according to which the system can be viewed as a huge collection of *resources*, that is applications components that can be uniquely identified through a single naming scheme.

3.2.1 Resource Identification

According to the REST architectural style, the identification of any resources relies on identifiers called *Uniform Resource Identifiers* (URIs) which, in an `HTTP` context, are also known as *Uniform Resource Locator* (URLs). Like many other REST systems, the resources of our system are organized in a hierarchy and that is used as a basis to identify a resource.

Firstly, our system supports only two resources:

- *Resource owners*, which are at the top of the resources hierarchy.
- *Serverless applications*, which, instead, are at the bottom.

Resource owners represent the entities capable of *creating* and *modifying* their own serverless applications. Most importantly, a resource owner name is unique in all our system.

The URI, assigned to each serverless applications resources, is made up of the following:

- the name of the serverless applications owner, that is the resource owner name.
- the name of the serverless application itself which must be unique among all applications owned by the resource owner.

For instance, a URI such as the one reported in listing 3.3 denotes the serverless application called "*application*" owned by the resource owner called "*andrea*".

Listing 3.3: A serverless application URI example

```
1 http://<DOMAIN>:80/andrea/application
```

3.2.2 Operations on Resources

As known `HTTP` protocol provides several methods to interact with resources. Constraining operations on resources only to `HTTP` provided methods is one of the keys to enable loose-coupling between our software and clients, as they only need to support `HTTP` protocol to use our framework[25].

Moreover, as known, when an operation on a specified resource is invoked, `HTTP` protocol offers a way of expressing errors and exceptions, which are represented by standardized status codes sent back as part of the header in the `HTTP` response message. Naturally, we adopt them to report any issue occurring during any operation performed on any resources because, since `HTTP` status codes have well-known meanings for HTTP clients, we can achieve a lower coupling between our software and clients [25][8].

According to our design, following operations are supported:

`PUT` which is generally used to create a new resource, like a new serverless application or new user.

More concretely, as an example, a `PUT` on "`/genericOwner/app1`", with the `JSON` representation of the serverless application to create, returns an `HTTP` response message having one of the following status codes:

- If a new serverless application called `app1` and belonging to `genericOwner` has been successfully created, the response code will be `201` (Created).

- If our framework encountered an unexpected condition which prevented it from fulfilling the serverless application creation request, the response code will be `500` (Internal Server Error). Moreover, a `JSON` formatted data, describing occurred error, will be sent as part of the `HTTP` response message.
- If our framework detects that a serverless application called `app1` and owned by `genericOwner` already exists, the status code `409` (Conflict) will be sent.
- If the `JSON`'s representation of the serverless application to create contains syntax errors, the code `400` (Bad Request) will be sent. The same status code will be sent also when a malformed URI is submitted, that is when the user `genericOwner` does not exist.

Similarly to the previous example, a `PUT` on "`/genericOwner`", is used to register a resource owner whose name is "`genericOwner`". Returned status codes are similar to those used in the previous example.

All data about created objects are stored inside our NoSQL distributed database instance.

`GET` which is used both to execute serverless applications and to retrieve the state of both a resource owner and the system.

For example, a `GET` on "`/genericOwner/app1`" triggers the execution of the serverless application called `app1` and owned by `genericOwner`. The request message can include a `JSON` representation of the data to pass to the invoked application as input.

One of the following event can occurs:

- If no error occurred during serverless application execution, an HTTP response message, having status code `200` (OK) with a `JSON` representation of the data produced as output by the invoked application, will be sent to the client.
- If specified URI refers to an inexistent serverless application, naturally, the status code `404` (Not Found) will be sent.

- If any error occurred, the response code will be `500` and the response message will include a `JSON` formatted data describing occurred exception.

It is also possible to submit a `GET` on "`/genericOwner`" whose message response, when no error has occurred, contains a `JSON` representation of the resource owner `genericOwner`. When some errors has occurred, similar status code like in previous example are adopted.

Finally, submitting a `GET` on "`/`", will be sent a response message containing the RSA public key which must be used for message encrypting.

`DELETE` is used to delete a resource.

For example a `DELETE` on "`/genericOwner`", is used to delete the resource owner whose name is "`genericOwner`", including all its serverless applications. The returned status code can be one of the following:

- `200` when submitted operation succeeded without errors
- `410` (Gone), if specified URI refers to an inexistent or already deleted resource owner.
- `500` when any error occurred. In that case, the response message will include a `JSON` formatted data describing occurred exception.

Naturally, is possible to submit a `DELETE` to permanently remove a serverless application.

`POST` is used to change the state of a resource.

To be more precise, if a `POST` is submitted on a serverless application whose URI is "`/genericOwner/app1`", the profiling task, regarding the application called `app1`, will be started. We will talk about profiling task for serverless applications very soon. In this case, one of the following status codes will be sent:

- `201` when submitted operation succeeded without errors
- `404` if specified URI refers to an inexistent serverless application.

- `500` when any error occurred during profiling task. Like always, the response message will include a `JSON` formatted data describing occurred exception.

A `POST` operation can be submitted for FaaS platforms credentials uploading too; we will discuss about this very shortly.

3.3 Communication

Unfortunately, clients are obliged to communicate directly with our application running on the cloud using a classical *client-server style communication*, which use classic *point-to-point* and *synchronous* communications, according to which users wait until their requests has been fully processed by our system [44].

As known, the use of communication style leads to an high degree of coupling between our server and clients; to be more precise, they are:

- *temporally* coupled, which means that both our server and our clients processes that are communicating will both have to be up and running [44];
- *referentially* coupled, since explicit referencing in communication is adopted [44];

Since all currently supported operations are factually idempotent, that is they can be performed repeatedly with the same effect as if they had been performed exactly once[21], *at-least-once* communication semantic is safely adopted; therefore, any client receives either a result, in which case the invoker knows that the procedure was executed at least once, or an exception. As known, aforementioned communication semantic is achieved exploiting the retransmission of request messages, which masks so-called omission failures[5]. Unfortunately, our system cannot manage neither crash or arbitrary failures[6] [21].

[5] An omission failure occurs when a server fails to respond to a request.

[6] Arbitrary failures, also known as *Byzantine failures*, represent the most severe type of failure which happens when a server produces an output that it should never have produced, but which cannot be detected as being incorrect

Finally, we assume that all processes are connected by *reliable channels*, that is no messages are lost, duplicated or corrupted; in our prototype, that feature is obviously guaranteed by the use of the `HTTP` application layer protocol, which, as known, exploits the reliable transport layer protocol `TCP`.

3.4 Security Issues

Unfortunately, to invoke serverless function hosted on multiple FaaS providers, our framework requires security tokens of FaaS platforms, that is the set of authorization information required to actually use FaaS services paid by resource owners.

According to our design, token must be provided by resource owner itself and stored inside our system; this situation exposes users to serious security issues.

Despite security problems are out of the scope of our work, we have decided to guarantee at least data confidentiality through message encryption. This is done exploiting asymmetric encryption which works by creating a key pair with a public and private key. As known, the private key is kept secret from everyone but the creator of the key, while the public key is available to everyone.

The key pair is generated by our framework at start up. As already said previously, the public key can be obtained by our users submitting an `GET` method.

When an user called `genericOwner` wants to upload its FaaS tokens, it have to submit a `POST` operation on "`/genericOwner`", with a `JSON` object containing following fields:

`ActionName` That field is used to specifiy if the user want to insert or delete a FaaS credential. According to current implementation, only `InsertUserCredentials` and `DeleteUserCredentials` values are allowed.

`Credential` An array of `JSON` object containing FaaS credentials, every of which is made up of following fields:

- `ProviderName` a string value representing the FaaS provider name. Currently, only `openwhisk` and `aws` values are allowed.
- `ProviderIPAddress` which contains the IP address of the FaaS platform provider. It is valid only for providers using Apache OpenWhisk.

`Credential` which contains the FaaS token.

Clearly, the value of `Credential` field is raw bytes of the `CSV` file containing credentials of the corresponding FaaS provider. The structure of this `CSV` file depend on the requirement of the particular FaaS provider; however, when referring to an Apache OpenWhisk based provider, this `CSV` file must have one column and two rows where the first one contains the string `auth` while the second the token string.

The value of `Credential` fields inside all objects included into `Credential` array must be encrypted using the public key of our framework.

All encrypted credential data sent to our framework are then stored inside our NoSQL distributed database instance. When our orchestrator need to invoke a serverless function on a given FaaS, it decrypts credentials using its private key.

Unfortunately, since data signing is not currently supported, data integrity, message authentication, and non-repudiation are not provided.

3.5 Implementation Details

We developed our system from scratch using Go[5], a compiled programming language syntactically similar to C, but with memory safety, garbage collection and structural typing [16].

We choose to use Go rather than any objected oriented language, like Java or C#, in order to maximize framework performance and to develop applications efficiently and faster. In fact, according to Togashi and Klyuev [43]'s research, Go has quite better result than Java regarding concurrency performance too. Moreover, its compile time is about three times faster than Java and its model, which doesn't allow cyclic import, makes dependency analysis and management very easy [43]. Moreover, the use of goroutine and channel mechanisms hide the complexity of concurrent processes, simplifying development.

All external libraries used to build our system are reported in table 3.1; some of them require additional steps to be fully utilizable by our system.

[7] https://github.com/influxdata/influxdb-client-go
[8] https://github.com/gocql/gocql
[9] https://github.com/aws/aws-sdk-go-v2
[10] https://github.com/apache/openwhisk-client-go/

Table 3.1: Libraries used in our implementation

Name	Description
`influxdata/influxdb-client-go`[7]	Native Go InfluxDB client library.
`gocql/gocql`[8]	Go Cassandra client library.
`aws/aws-sdk-go-v2`[9]	AWS SDK for Go.
`apache/openwhisk-client-go`[10]	Openwhisk Go client library.
`rs/zerolog`[11]	A simple JSON Logger.
`draffensperger/golp`[12]	Golang wrapper for the LPSolve.

3.5.1 The Technical Debt Management

During development process our aim was to produce a software having high quality standards, despite limitations on time which naturally imposed to us the trad-off between delivery and quality.

Precisely, in order to reach our software quality aims, during all development process time, we try to reduce the number of *code smells* as much as possible.

Code smells are neither bug nor technically incorrect events and do not prevent the program from its expected behavior, however scientific literature shows that a correlation between code smells and the risk of bugs exists; in fact, the greater the number of code smells, the greater the risk of bugs within the code [22].

Moreover, code smells can be an indicator of factors that contribute to so-called *technical debt* [22]. We call technical debt the result of past decisions from software development point of view that negatively affect the project future. Steve McConnell defined technical debt as "*A design or construction approach that's expedient in the short-term but that creates a technical context in which the same work will cost more to do later than it would cost to do now*". In other words, technical debt regards any quality problem in the source code like duplication, excessive complexity and design problems [23][24].

When technical debt grows at an uncontrollable rate, you can rapidly reach a point making the software evolution unfeasible, requiring a complete software refactoring. To overcome such situation, it is important that the technical debt is properly managed and kept under control during software development[23].

[11] https://github.com/rs/zerolog
[12] https://github.com/draffensperger/golp

To manage technical debt of our system we used SonarQube[13], a tool able to report the technical debt as the amount of effort required to fix all code smells in a project; however, it does not just report the technical debt, it computes the number of duplicated code blocks, vulnerabilities, test coverage percentage and many other software metrics.

At the end of our software development process, which ends with a software made up of 8405 lines of code, SonarQube reports 0 code smells, 0 duplicated code blocks and 0 security vulnerabilities. This result should ensure a high software maintainability, simplifying the development of future releases.

3.5.2 FaaS Provider Support

Our system supports serverless functions invocation hosted on following FaaS platform providers:

- AWS Lambda [1];
- Apache OpenWhisk [12];

Therefore, only `openwhisk` and `aws` values are allowed and recognized by our software to specify where a concrete function is hosted inside a serverless workflow definition based on our custom AFCL.

Unfortunately, our system software has been not designed to manage serverless concrete functions hosted on FaaS platform providers. In other words, it is not actually possible to execute CRUD (*Create*, *Read*, *Update*, and *Delete*) operations regarding concrete functions on FaaS platform providers. Our framework limits itself to invoking serverless functions already deployed on FaaS platforms; conversely, CRUD operations regarding serverless applications are fully implemented.

3.5.3 Final Considerations

As TSDB, that is for the storage and the retrieval of time series data regarding serverless function performance, we adopted InfluxDB[17], an open-source time series database written in Go programming language.

To implement the directory where to store all points related to a specified resource owner, we exploit the InfluxDB's concept called *bucket*. A bucket is equivalent to a directory where time series data can be stored. Moreover, differently form a directory, a bucket has a retention period according to which InfluxDB drops all points with timestamps older than the bucket retention period; however, our prototype does not exploit this feature, therefore buckets retention periods are disabled by default.

Conversely, to store data about users, serverless applications and FaaS security token we used Cassandra[15], a free and open-source, distributed, wide-column store, NoSQL database written in Java.

Finally, to resolve the optimization problems, which according to our model must be solved to find a configuration allowing us to respect QoS constraints, we adopted `lp_solve`, a *Mixed Integer Linear Programming* (MILP) solver written in C [18].

Chapter 4

System Model

In this section, we will first formally describe the workflow of a serverless application, then we will develop the analytical model used by our framework to predict its the performance and cost.

4.1 SLA definition

As said previously, our system goal is the fulfillment of both *functional requirements*, concerning the orchestration of a serverless workflow, and *non-functional* requirements, concerning the *quality of service* (QoS) levels that should be guaranteed.

In our context, aforementioned non-functional requirements are the result of a commitment, signed by both our software system and a customer, which lead up to the definition of a *Service Level Agreement* (SLA) specifying guarantees about the *average values* of the following attributes:

Response time the interval of time elapsed from the serverless application invocation to its completion;

Cost the price charged for the execution of all serverless function belonging to an application;

Is very important to emphasize that our framework considers SLAs whose conditions should be hold at *local level*, that is focusing only on the fulfillment of all requirements regarding a *single* serverless application invocation's request, which

is forwarded by only one customer. Consequently, since the adaptation actions regards a single request, the framework acts irrespectively of whether aforementioned request belongs to some flow generated by one or more customers.

Formally, according to our model, for each serverless application invocation, a SLA is defined as follows:

$$SLA \stackrel{def}{=} \langle (RT, w_{RT}), (C, w_C) \rangle \tag{4.1}$$

where:

- $RT \in \mathbb{R}^+$ is the upper bound on the average response time of the serverless application.
- $C \in \mathbb{R}^+$ is the upper bound on average service cost per serverless application invocation.
- $w_{RT}, w_C \in \mathbb{R}^+ \cap [0, 1]$ represent, respectively, the SLA attributes weights regarding the response time and the cost; simply, greater a SLA attribute weight value, the greater is the importance of that attribute. Weights for the different SLAs attributes are used to improve our flexibility regarding meeting customer's expectations.

  Formally, following constraint must hold:

  $$w_{RT} + w_C = 1 \tag{4.2}$$

  We will describe later how weights for the different SLA attributes can be used.

4.2 Serverless Choreography

According to our model, a *serverless choreography*, or simply *choreography*, represents the *resource* used to model both the concepts of serverless function and serverless application (or compound serverless function).

Informally, that abstraction has been derived from that of a *control-flow graph* which, as known, describes, using graphs notation, all paths that might be traversed

through a serverless application during its execution. Similarly, a choreography describes calling relationships between functions belonging to an application in a serverless environment, combining them using several types of control-flow structures, like sequence, branch, loop or connectors for a parallel execution.

4.2.1 Preliminary Definitions

In order to provide a formal definition of a choreography, we have to define some very useful notations.

Let $n \in \mathbb{N}$ and $G = (\Phi, E)$ a directed graph where:

- Φ is a finite set of vertices, such that $|\Phi| = n$;
- $E \subseteq \Phi \times \Phi$ is a finite set of ordered pairs of vertices $e_{ij} = (\phi_i, \phi_j)$, where $\phi_i \in \Phi$ to $\phi_j \in \Phi$ for any $i, j \in \mathbb{N} \cap [1, n]$. Any ordered pair of vertices is also called *directed edge*;

Then, we will adopt following notations:

- A *path* of G is defined as a finite sequence of distinct vertices and edges. We will denote a path by π, which formally can be represented as follows:

$$\pi = \phi_1 e_1 \phi_2 e_2 \dots e_{n-2} \phi_{n-1} e_{n-1} \phi_n \tag{4.3}$$

  where:

  - $\phi_i \in \Phi$, for all $i \in \mathbb{N} \cap [1, n]$
  - $e_i = (\phi_i, \phi_{i+1}) \in E$, for all $i \in \mathbb{N} \cap [1, n-1]$

- Let $\phi_i, \phi_j \in \Phi$ for any $i, j \in \mathbb{N} \cap [1, n]$, the set denoted by $\Pi(\phi_i, \phi_j)$ identifies all possible paths starting from vertex ϕ_i and ending at vertex ϕ_j.
- For any $u \in \mathbb{N} \cap [1, n]$, the set $out(\phi_u)$ ($in(\phi_u)$) denotes all edges starting (ending) from (to) vertex ϕ_u, while the set $succ(\phi_u)$ ($pred(\phi_u)$) includes all direct successor (predecessors) vertices of ϕ_u. Formally:

$$out(\phi_u) \stackrel{def}{=} \{(\phi_u, \phi) \in E, \quad \forall \phi \in \Phi\} \tag{4.4}$$

$$in(\phi_u) \stackrel{def}{=} \{(\phi, \phi_u) \in E, \quad \forall \phi \in \Phi\} \tag{4.5}$$

$$succ(\phi_u) \stackrel{def}{=} \{\phi \in \Phi \mid (\phi_u, \phi) \in out(\phi_u)\} \tag{4.6}$$

$$pred(\phi_u) \stackrel{def}{=} \{\phi \in \Phi \mid (\phi, \phi_u) \in in(\phi_u)\} \tag{4.7}$$

4.2.2 Serverless Choreography Definition

According to serverless paradigm, the execution of an application always starts with a particular function, usually triggered through events or `HTTP` requests, acting as "*entry point*" of the serverless workflow; any other functions, belonging to the application, will be invoked subsequently according to the business logic specified by customer. Conversely, the execution of an application ends when the execution of its last function ends, which acts as the "*end point*" of the serverless workflow.

Assuming that any serverless application has only one function acting as entry point, let $n \in \mathbb{N} \setminus \{0\}$ and R a resource owner, then a choreography, owned by R, is a weakly connected[1] weighted directed graph denoted as follows:

$$\mathcal{C}_R \stackrel{def}{=} (\Phi, E) \tag{4.8}$$

where:

- Each vertex $\phi \in \Phi$, where $|\Phi| = n$, is called *abstract serverless function*, or simply *abstract function*, and it represents a generic computational unit:
  - $P_{exit} : \Phi \times \mathbb{R}^+ \to \mathbb{R}^+ \cap [0, 1]$ represents the so-called *exit probability function*, according to which $P_{exit}(\phi, t)$ denotes the probability of execution's termination of $\mathcal{C}_R$, when the execution of ϕ is terminated, at time t. We will describe very soon how to compute it.
- Let $i, j \in \mathbb{N} \cap [1, n]$ and $\phi_i, \phi_j \in \Phi$, any directed edge $e_{ij} = (\phi_i, \phi_j) \in E$ represents the calling relationship between two abstract functions, which depends

[1] A directed graph is called *weakly connected* if replacing all of its directed edges with undirected edges produces a connected undirected graph [45].

on the business logic defined by R.

In our context, any directed edge $(\phi_i, \phi_j) \in E$ states that the abstract function ϕ_j *can* be called by ϕ_i;

- Let $i, j \in \mathbb{N} \cap [1, n]$, the number $p_{ij} \in \mathbb{R}^+ \cap [0, 1]$ is the weight assigned to the edge $(\phi_i, \phi_j) \in E$, where:
  - The number p_{ij} represents the so-called *transition probability* from ϕ_i to ϕ_j, that is the probability of invoking ϕ_j when the execution of ϕ_i is terminated;
  - $P : \Phi \times \Phi \times \mathbb{R}^+ \to \mathbb{R}^+ \cap [0, 1]$, called *transition probability function*, is such that $P(\phi_i, \phi_j, t) = p_{ij}$, where t is the time. Obviously, when $(\phi_i, \phi_j) \notin E$, ϕ_i cannot invoke ϕ_j.
  - For any path $\pi = \phi_1 e_1 \dots e_{n-1} \phi_n$ of $\mathcal{C}_R$, we define *transition probability of the path* π the following quantity:

$$TPP(\pi, t) = \prod_{i=1}^{n-1} P(\phi_i, \phi_{i+1}, t) \tag{4.9}$$

    Particularly, an abstract function $\phi \in \Phi$ is said *unreachable* if, for any time t, following condition holds:

$$\sum_{\pi \in \Pi(\alpha(\mathcal{C}_R), \phi)} TPP(\pi, t) = 0 \tag{4.10}$$

    Conversely, a $\phi \in \Phi$ is said *reachable* when:

$$\sum_{\pi \in \Pi(\alpha(\mathcal{C}_R), \phi)} TPP(\pi, t) > 0 \tag{4.11}$$

- Φ must be such that following condition holds:

$$\exists! \phi \in \Phi \quad | \quad in(\phi) = \emptyset \tag{4.12}$$

$$\exists \phi \in \Phi \quad | \quad out(\phi) = \emptyset \tag{4.13}$$

that is, Φ has to contain only one abstract function acting as entry point and at least one acting as end point. In other terms, we can state that:

$$\begin{array}{lll} \phi \in \Phi \quad \text{is the } \textit{entry point} \text{ of } \mathcal{C}_{\mathcal{R}} & \Leftrightarrow & in(\phi) = \emptyset \\ \phi \in \Phi \quad \text{is an } \textit{end point} \text{ of } \mathcal{C}_{\mathcal{R}} & \Leftrightarrow & out(\phi) = \emptyset \end{array} \tag{4.14}$$

Finally, we will use the notation $\alpha(\mathcal{C}_R)$ to denote the vertex $\phi \in \Phi$ acting as entry point of the choreography $\mathcal{C}_R$; conversely, we will adopt the notation $\Omega(\mathcal{C}_R)$ to denote the set of end points of $\mathcal{C}_R$.

- Following condition must hold:

$$|\Pi(\alpha(\mathcal{C}_R), \phi)| \geq 1 \qquad \forall \phi \in \Phi \tag{4.15}$$

in other words, at least one path starting from the entry point of $\mathcal{C}_R$ to each abstract function $\phi \in \Phi$ must exist.

- Sometimes, we will use $P_{exit}(\mathcal{C}_R, t)$ notation to denote the so-called *exit probability* of $\mathcal{C}_R$ at time t, that is the probability to terminate its execution after its invocation; it can be computed using following formula:

$$P_{exit}(\mathcal{C}_R, t) \stackrel{def}{=} \sum_{\phi \in \Omega(\mathcal{C}_R)} \sum_{\pi \in \Pi(\alpha(\mathcal{C}_R), \phi)} TPP(\pi, t) \tag{4.16}$$

where:

  - for each $\phi \in \Omega(\mathcal{C}_R)$, any $\pi \in \Pi(\alpha(\mathcal{C}_R)$ represent the path starting from the entry point of $\mathcal{C}_R$ to one of its end node.

We always assume that $P_{exit}(\mathcal{C}_R, t) = 1$; that condition guarantees that any execution of $\mathcal{C}_R$ will terminate.

- We will use $E[I_{\mathcal{C}_R}]$ notation to denote the expected value of the number of executions of the choreography $\mathcal{C}_R$.

A choreography $\mathcal{C}_R$ can be uniquely identified by an ordered pair (a, b), where a is the name of the resource owner R, while b is the function choreography name.

We will say that a choreography models a serverless function when $|\Phi| = 1$ and $|E| = 0$; conversely, it models a serverless application when $|\Phi| > 1$ and $|E| > 0$.

From now, a choreography $\mathcal{C}_R$ will be briefly denoted by $\mathcal{C}$ when no confusion can arise about the resource owner R.

4.2.2.1 Abstract Serverless Function

Supposing that a choreography $\mathcal{C} = (\Phi, E)$ is given.

As said previously, any $\phi \in \Phi$ represents an abstract function, which is a *resource* modeling a computational unit required by business logic provided by developers.

According to our model, there are two types of abstract functions implementations:

- ϕ is called *serverless executable functions*, or simply *executable function*, when ϕ contains *executable code*; therefore, any executable function naturally models a serverless function.

  $\mathcal{F}_{\mathcal{E}}(\mathcal{C})$ is defined as the set containing all executable function of $\mathcal{C}$ which is formally defined as follows:

  $$\mathcal{F}_{\mathcal{E}}(\mathcal{C}) \stackrel{def}{=} \{\phi \in \Phi \mid \phi \text{ is a serverless executable function } \} \tag{4.17}$$

  However, multiple different implementations of a given executable function can be provided by developers which, although they must be semantically and logically equivalent, may eventually expose different performance or cost behavior. We call these different implementations as *concrete serverless function*, or simply, *concrete function* of ϕ.

  For any $\phi \in \mathcal{F}_{\mathcal{E}}(\mathcal{C})$, we will use $\mathbf{F}_\phi$ notation to represent the so-called *implementation-set* of ϕ, that is the set containing all concrete function implementing ϕ, which are denoted using f_ϕ notation.

  Later, we will explain how our framework is able to pick, for all $\phi \in \mathcal{F}_{\mathcal{E}}(\mathcal{C})$, exactly one $f_\phi \in \mathbf{F}_\phi$ whose properties allow us to meet user-specified QoS requirements.

Finally, according to our model point of view, following conditions must be hold:

$$|succ(\phi)| = n \qquad \forall \phi \in \mathcal{F}_{\mathcal{E}}(\mathcal{C}) \wedge n \in \{0, 1\} \tag{4.18}$$

$$P(\phi, \phi_x, t) = 1 \qquad \forall \phi, \phi_x \in \mathcal{F}_{\mathcal{E}}(\mathcal{C}) : \phi_x \in succ(\phi) \wedge |succ(\phi)| = 1, \forall t \tag{4.19}$$

$$P_{exit}(\phi, t) = 0 \qquad \forall \phi \in \mathcal{F}_{\mathcal{E}}(\mathcal{C}), \forall t \tag{4.20}$$

- ϕ is called *serverless orchestration functions*, or simply *orchestration functions*, when ϕ contains *orchestration code*.

  According to our model, orchestration code represents the logic required to chain together any components of an application, evaluate branch and loop conditions, handle events and trigger executable functions in the correct order according to the business logic. In other words, orchestration code is used to manage the control-flow of any application.

  $\mathcal{F}_{\mathcal{O}}(\mathcal{C})$ is defined as the set containing all orchestration functions of $\mathcal{C}$ and it is formally defined as follows:

$$\mathcal{F}_{\mathcal{O}}(\mathcal{C}) \stackrel{def}{=} \{\phi \in \Phi \mid \phi \text{ is a serverless orchestration function } \} \tag{4.21}$$

  Moreover, each orchestration function $\phi \in \mathcal{F}_{\mathcal{O}}(\mathcal{C})$ has a *type* which, as we will explain later, determines how choreography performances are computed. The orchestration function type can be determined based on certain characteristics of the graph of $\mathcal{C}$. According to our model, we distinguish four orchestration functions types: *Branch*, *Conditional Loop*, *Parallel* and *Exit*. We will give a better explanation about them very soon.

  Moreover, according to our model, orchestration functions travel always in pairs. Each $\phi \in \mathcal{F}_{\mathcal{O}}(\mathcal{C})$ can be either an *opening* or a *closing* orchestration function of a given type: the first indicates the beginning of particular section of $\mathcal{C}$, called *structure*, while the second denotes its end. Anyway, if τ denotes the orchestration function type, we will use τ_α and τ_ω notations to denote,

respectively, an opening and a closing orchestration function of type τ. Moreover, we will use the notation $\mathcal{T}(\phi)$ to represent the function returning the orchestration function type of ϕ. We will give more details about this very soon.

Finally, following condition must hold:

$$|succ(\phi)| = n \qquad \forall \phi \in \mathcal{F}_{\mathcal{O}}(\mathcal{C}) : \mathcal{T}(\phi) = \tau_\omega \wedge n \in \{0, 1\} \tag{4.22}$$

$$P(\phi, \phi_x, t) = 1 \qquad \forall \phi, \phi_x \in \mathcal{F}_{\mathcal{O}}(\mathcal{C}) : \mathcal{T}(\phi) = \tau_\omega \wedge \phi_x \in succ(\phi) \wedge |succ(\phi)| = 1, \forall t \tag{4.23}$$

Clearly, based on above definitions, we can say:

$$\mathcal{F}_{\mathcal{E}}(\mathcal{C}) \cap \mathcal{F}_{\mathcal{O}}(\mathcal{C}) = \emptyset \tag{4.24}$$

$$\mathcal{F}_{\mathcal{E}}(\mathcal{C}) \cup \mathcal{F}_{\mathcal{O}}(\mathcal{C}) = \Phi \tag{4.25}$$

$$|\mathcal{F}_{\mathcal{E}}(\mathcal{C})| + |\mathcal{F}_{\mathcal{O}}(\mathcal{C})| = |\Phi| \tag{4.26}$$

Any abstract function ϕ is uniquely identified by an ordered pair (a, b), where:

- a represents the identifier of the choreography $\mathcal{C}$;
- b is the name of the abstract serverless function ϕ;

4.2.2.2 Executability Condition

Let $\mathcal{C} = (\Phi, E)$ a choreography, we always assume that all concrete functions $f_\phi \in \mathbf{F}_\phi$, for all $\phi \in \mathcal{F}_{\mathcal{E}}(\mathcal{C})$, are already deployed on one or more FaaS platform by customers. Then, in order to effectively start the execution of $\mathcal{C}$, is required that, for each executable function $\phi \in \mathcal{F}_{\mathcal{E}}(\mathcal{C})$, *at least one* implementation exists.

Formally, we said that a choreography is *executable* when:

$$\mathcal{C} \text{ is executable} \quad \Leftrightarrow \quad |\mathbf{F}_\phi| \geq 1 \qquad \forall \phi \in \mathcal{F}_{\mathcal{E}}(\mathcal{C}) \tag{4.27}$$

We will only deal with executable choreographies.

4.2.2.3 Serverless Sub-Choreography

Let $\mathcal{C} = (\Phi, E)$ a choreography, the weakly connected weighted directed sub-graph $\mathcal{C}^*$ of $\mathcal{C}$, defined as follows:

$$\mathcal{C}^* \stackrel{def}{=} (\Phi^*, E^*) \qquad \text{where } \Phi^* \subseteq \Phi \wedge E^* \subseteq E \tag{4.28}$$

is called *serverless sub-choreography* of $\mathcal{C}$, or simply *sub-choreography* of $\mathcal{C}$, when the conditions 4.12, 4.13 and 4.15 are hold.

4.2.2.4 Serverless Pipeline Choreography

Suppose to have a choreography $\mathcal{C} = (\Phi, E)$ satisfying following conditions:

$$|\Omega(\mathcal{C})| = 1 \tag{4.29}$$

$$|in(\phi)| = 1 \qquad \forall \phi \in \Phi \setminus \{\alpha(\mathcal{C})\} \tag{4.30}$$

$$|out(\phi)| = 1 \qquad \forall \phi \in \Phi \setminus \Omega(\mathcal{C}) \tag{4.31}$$

$$P(\phi_x, \phi_y, t) = 1 \qquad \forall x, y \in \mathbb{N} \cap [1, |\Phi|], \forall t \tag{4.32}$$

Then $\mathcal{C}$ will be called *serverless pipeline choreography*, or simply a *pipeline choreography*.

4.3 Serverless Workflow Configuration

In order to reach its goal, our framework has to determine a so-called *serverless choreography configuration*, or simply *choreography configuration* or *configuration*, specifying which concrete function will be invoked when the corresponding executable function is executed; moreover, its configuration parameters, like memory size or CPU power, have to be determined.

Formally, let a choreography $\mathcal{C} = (\Phi, E)$, when an invocation request of $\mathcal{C}$ arrive on our system, the latter acts as follows:

- For each $\phi \in \mathcal{F}_{\mathcal{E}}(\mathcal{C})$, it selects only one concrete function $f_\phi \in \mathbf{F}_\phi$, which will be effectively invoked and executed on its corresponding FaaS platform.
- For each selected concrete function f_ϕ, it selects a value for memory size.

Clearly, as we will explain in detail later, to perform aforementioned selection, our framework has to acquire a series of function response time and charged costs when the $\mathcal{C}$ is executed using different concrete functions and memory sizes.

4.3.1 Executable Function Configuration

To build a configuration for a given choreography, it is clearly needed to select a configuration for its executable functions.

Then, let $\phi \in \mathcal{F}_{\mathcal{E}}(\mathcal{C})$ an executable function and $\mathbf{F}_\phi$ the corresponding implementation-set, formally an *executable function configuration* x_ϕ for the executable function ϕ is a two-dimensional vector defined as follows:

$$x_\phi = (f_\phi, m) \in f_\phi \times \mathbf{M}_{f_\phi} \subseteq \mathbf{F}_\phi \times \mathbb{N} \tag{4.33}$$

where:

- $f_\phi \in \mathbf{F}_\phi$ denotes a particular concrete function implementing the executable function ϕ.
- $m \in \mathbf{M}_{f_\phi}$ represents the allocated memory size during the execution of f_ϕ, where $\mathbf{M}_{f_\phi} \subseteq \mathbb{N}$ is the set holding all available memory size configurations allowed by provider where the concrete function f_ϕ is executed.

4.3.2 Serverless Choreography Configuration

Let $n, k \in \mathbb{N} \setminus \{0\}$ and a choreography $\mathcal{C} = (\Phi, E)$ such that $|\Phi| = n$ and $|\mathcal{F}_{\mathcal{E}}| = k$ where $k \leq n$.

Formally, a *serverless choreography configuration* $\mathbf{X}_\mathcal{C}$ for the choreography $\mathcal{C}$ is a vector such that:

$$
\begin{aligned}
\mathbf{x}_{\mathcal{C}} &\overset{def}{=} \{x_{\phi_1}, \ldots, x_{\phi_k}\} \\
&\in \left\{ \left\{ \bigcup_{j=1}^{|\mathbf{F}_{\phi_1}|} f_{\phi_{1_j}} \times \mathbf{M}_{f_{\phi_{1_j}}} \right\} \times \ldots \times \left\{ \bigcup_{j=1}^{|\mathbf{F}_{\phi_k}|} f_{\phi_{k_j}} \times \mathbf{M}_{f_{\phi_{k_j}}} \right\} \right\} \\
&= \bigtimes_{i=1}^{k} \left\{ \bigcup_{j=1}^{|\mathbf{F}_{\phi_i}|} f_{\phi_{i_j}} \times \mathbf{M}_{f_{\phi_{i_j}}} \right\} \\
&\subseteq \bigtimes_{i=1}^{k} \{\mathbf{F}_{\phi_i} \times \mathbb{N}\} = \mathbf{X}_{\mathcal{C}}
\end{aligned} \tag{4.34}
$$

where:

- x_{ϕ_i} represents the executable function configuration for the executable function ϕ_i, for some $i \in \mathbb{N} \cap [1, k]$.
- $f_{\phi_{i_j}}$ represent the j-th concrete function implementing the executable function ϕ_i, for some $i \in \mathbb{N} \cap [1, k]$ and $j \in \mathbb{N} \cap [1, |\mathbf{F}_{\phi_i}|]$.
- $\mathbf{M}_{f_{\phi_{i_j}}} \subseteq \mathbb{N}$ denotes the set containing all available memory size options, allowed by provider, during the execution of the concrete function $f_{\phi_{i_j}}$, for some $i \in \mathbb{N} \cap [1, k]$ and $j \in \mathbb{N} \cap [1, |\mathbf{F}_{\phi_i}|]$
- $\mathbf{X}_{\mathcal{C}}$ represents the so-called *solutions space*, which includes all possible choreography configurations.

From now, we will use $RT(\mathcal{C}, \mathbf{x}_{\mathcal{C}}, t)$ and $C(\mathcal{C}, \mathbf{x}_{\mathcal{C}}, t)$ notations to represent the average response time and billed cost when the choreography $\mathcal{C}$ is executed with the configuration $\mathbf{x}_{\mathcal{C}}$ at time t.

4.4 Serverless Choreography's Structure

Let $\mathcal{C} = (\Phi, E)$ a choreography.

Informally, a *structure* is defined as any sub-choreography $\mathcal{C}^* = (\Phi^*, E^*, \Theta_{\mathcal{C}^*})$ of $\mathcal{C}$ whose entry point $\alpha(\mathcal{C}^*)$ has multiple outgoing edges or, equivalently, has more than one successors.

Sometimes, structures have a special end point having multiple ingoing edges and the same type τ of $\alpha(\mathcal{C}^*)$; if it exists, we will use $\omega(\mathcal{C}^*)$ to denote it.

When $\omega(\mathcal{C}^*)$ exists, $\alpha(\mathcal{C}^*)$ and $\omega(\mathcal{C}^*)$ are, respectively, the opening and closing orchestration functions of the structure $\mathcal{C}^*$.

Formally:

$$\mathcal{C}^* \text{ is a structure } \Leftrightarrow |out(\alpha(\mathcal{C}^*))| > 1 \tag{4.35}$$

or, equivalently:

$$\mathcal{C}^* \text{ is a structure } \Leftrightarrow |succ(\alpha(\mathcal{C}^*))| > 1 \tag{4.36}$$

For the aim of our work, the most important aspect is that every structure can be viewed as a "*set*" of sub-choreographies. Formally, let $c \in \mathbb{N} \setminus \{0\}$ and $\mathcal{C}^{**} = (\Phi^{**}, E^{**})$ a sub-choreography of $\mathcal{C}^*$ such that:

$$\begin{array}{rcl} \Phi^{**} & \stackrel{def}{=} & \Phi^* \setminus \{\alpha(\mathcal{C}^*), \omega(\mathcal{C}^*)\} \\ E^{**} & \stackrel{def}{=} & E^* \setminus \Big[out\Big(\alpha(\mathcal{C}^*)\Big) \cup in\Big(\omega(\mathcal{C}^*)\Big)\Big] \end{array} \tag{4.37}$$

that is, $\mathcal{C}^{**}$ is obtained removing both the entry point and end point $\omega(\mathcal{C}^*)$ of $\mathcal{C}^*$, including any edges starting/ending from/to them. Then, $\Theta_{\mathcal{C}^*}$, such that $|\Theta_{\mathcal{C}^*}| = c$ denotes the set containing all connected components of $\mathcal{C}^{**}$ satisfying 4.12, 4.13 and 4.15 conditions; in other words, each connected component of $\mathcal{C}^{**}$ represents a sub-choreography of $\mathcal{C}$. Formally:

$$\begin{array}{rcl} \Theta_{\mathcal{C}^*} & \stackrel{def}{=} & \{\theta_1, \ldots, \theta_c\} \\ & = & \bigcup_{i=1}^{c} \left\{ \begin{array}{l} \theta_i = (\Phi_i^{**}, E_i^{**}) : \Phi_i^{**} \subset \Phi^{**} \wedge E_i^{**} \subset E^{**} \\ \Phi_i^{**} \cap \Phi_j^{**} = E_i^{**} \cap E_j^{**} = \emptyset \qquad j \in \mathbb{N} \cap [1, c] : j \neq i \\ \theta_i \text{ is a connected component of } \mathcal{C}^{**} \\ \theta_i \text{ satisfies 4.12, 4.13 and 4.15 conditions} \end{array} \right\} \end{array} \tag{4.38}$$

Sometimes, to reach our goal, it will be necessary to "*replace*" the structures, inside a given choreography, in order to convert it into a pipeline type, performing the so-called *workflow simplification process*.

Exploiting different methods for different structures which we will describe shortly, our model trims the graph associated with a choreography by removing or modifying vertices and edges in order to convert it into a pipeline choreography. Formally, this is done by replacing any structure with a single orchestration function ϕ_{fake}, obtaining a new choreography $\mathcal{C}' = (\Phi', E')$, such that:

$$\Phi' = \{\Phi \setminus \Phi^*\} \cup \{\phi_{fake}\} \tag{4.39}$$

$$E' = \{E \setminus E^*\} \cup \left\{\left(prev(\alpha(\mathcal{C}^*)), \phi_{fake}\right)\right\} \cup \left\{\left(\phi_{fake}, succ(\omega(\mathcal{C}^*))\right)\right\} \tag{4.40}$$

where $\phi_{fake} \in \mathcal{F}_{\emptyset}(\mathcal{C}^*)$ is such that:

$$RT_{\phi_{fake}}(\mathbf{x}_{\mathcal{C}}, t) = RT(\mathcal{C}^*, \mathbf{x}_{\mathcal{C}}, t) \tag{4.41}$$

$$C_{\phi_{fake}}(\mathbf{x}_{\mathcal{C}}, t) = C(\mathcal{C}^*, \mathbf{x}_{\mathcal{C}}, t) \tag{4.42}$$

$$P_{exit}(\phi_{fake}, t) = P_{exit}(\mathcal{C}^*, t) \tag{4.43}$$

4.4.1 Parallel

Any structure $\mathcal{P} = (\Phi', E', \Theta)$ such that:

$$\omega(\mathcal{P}) \in \Omega(\mathcal{P}) \tag{4.44}$$

$$\mathcal{T}(\alpha(\mathcal{P})) = \mathcal{T}(\phi_x) = \tau_\alpha \tag{4.45}$$

$$\mathcal{T}(\omega(\mathcal{P})) = \mathcal{T}(\phi_y) = \tau_\omega \tag{4.46}$$

$$TPP(\pi, t) = 1 \qquad \forall \pi \in \Pi(\phi_x, \phi_y), \forall t \tag{4.47}$$

$$|\Theta| = n \qquad n \in \mathbb{N} \setminus \{0\} \tag{4.48}$$

$$E[I_\theta] = 1 \qquad \forall \theta \in \Theta \tag{4.49}$$

$$P_{exit}(\theta, t) = 0 \qquad \forall \theta \in \Theta \tag{4.50}$$

$$|succ(\alpha(\mathcal{P}))| = |prev(\omega(\mathcal{P}))| > 1 \tag{4.51}$$

is called *parallel structure* and τ denotes a *parallel* type.

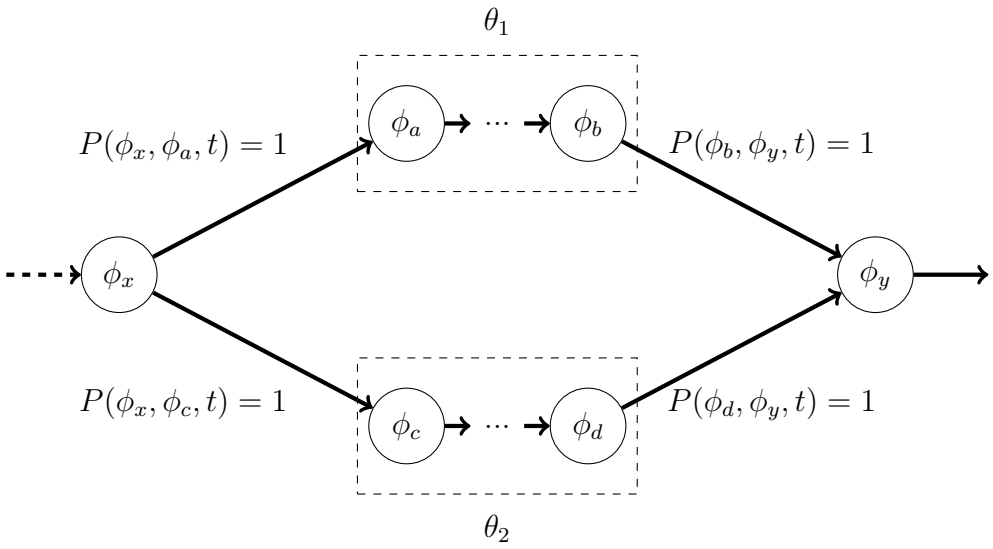

Figure 4.1: Parallel structure in a serverless workflow.

Clearly, the response time of a parallel structure is equal to the longest response time of all its sub-choreographies.

Formally, let $\theta_i \in \Theta$ the i-th sub-choreography of $\mathcal{P}$, where $i \in \mathbb{N} \cap [1, n]$, and $\mathbf{x}_\mathcal{C} \in \mathbf{X}_\mathcal{C}$ a choreography configuration.

At any time t, the average response time and billed cost of $\mathcal{P}$ can be computed as follows:

$$RT(\mathcal{P}, \mathbf{x}_\mathcal{C}, t) \stackrel{def}{=} max\left\{RT(\theta_i, \mathbf{x}_\mathcal{C}, t) \mid \theta \in \Theta\right\} \tag{4.52}$$

$$C(\mathcal{P}, \mathbf{x}_\mathcal{C}, t) \stackrel{def}{=} \sum_{i=1}^{n} C(\theta_i, \mathbf{x}_\mathcal{C}, t) \tag{4.53}$$

Since, by definition, we imposed that all sub-choreography of $\mathcal{P}$ have exit probability equal to zero, the exit probability of the a parallel structure is zero too. Therefore, formally:

$$P_{exit}(\mathcal{P}, t) = 0 \tag{4.54}$$

4.4.2 Conditional Loop

Any structure $\mathcal{L} = (\Phi', E', \Theta)$ such that:

$$\omega(\mathcal{L}) \in \Omega(\mathcal{L}) \tag{4.55}$$
$$\mathcal{T}(\alpha(\mathcal{L})) = \mathcal{T}(\phi_x) = \tau_\alpha \tag{4.56}$$
$$\mathcal{T}(\omega(\mathcal{L})) = \mathcal{T}(\phi_y) = \tau_\omega \tag{4.57}$$
$$|\Theta| = 1 \tag{4.58}$$
$$|succ(\alpha(\mathcal{P}))| = 2 \tag{4.59}$$
$$|prev(\alpha(\mathcal{P}))| \leq 2 \tag{4.60}$$
$$|prev(\omega(\mathcal{P}))| = 1 \tag{4.61}$$
$$E[I_\theta] \geq 0 \qquad \theta \in \Theta \tag{4.62}$$
$$P(\phi_x, \alpha(\theta), t) = x \tag{4.63}$$
$$P(\phi_x, \phi_y, t) = 1 - x \tag{4.64}$$
$$P(\omega(\theta), \phi_x, t) = 1 \qquad \theta \in \Theta \tag{4.65}$$
$$P(\phi_x, \alpha(\theta), t) + P(\phi_x, \phi_y, t) = 1 \qquad \theta \in \Theta \tag{4.66}$$

is called *conditional loop structure* and τ denotes a *conditional loop* type; it is used to model a `while` and `for` programming structures inside a serverless application.

Similarly to the branch structure, we have adopt the equation 4.82 to compute the transition probability $P(\phi_x, \alpha(\theta), t)$.

In order to compute performance data of a loop structure, $E[I_\theta(t)]$, that is the expected value of the number of iterations of $\mathcal{L}$ at the time t, is required.

To compute aforementioned information, we can model our problem using geometric distribution, which gives the probability according to which the first occurrence of success requires $k \in \mathbb{N}$ independent trials, each with success probability p and failure probability $q = 1 - p$. In our case, if the our success corresponds to event "*we will not execute the loop body*", we know that success probability p is given by:

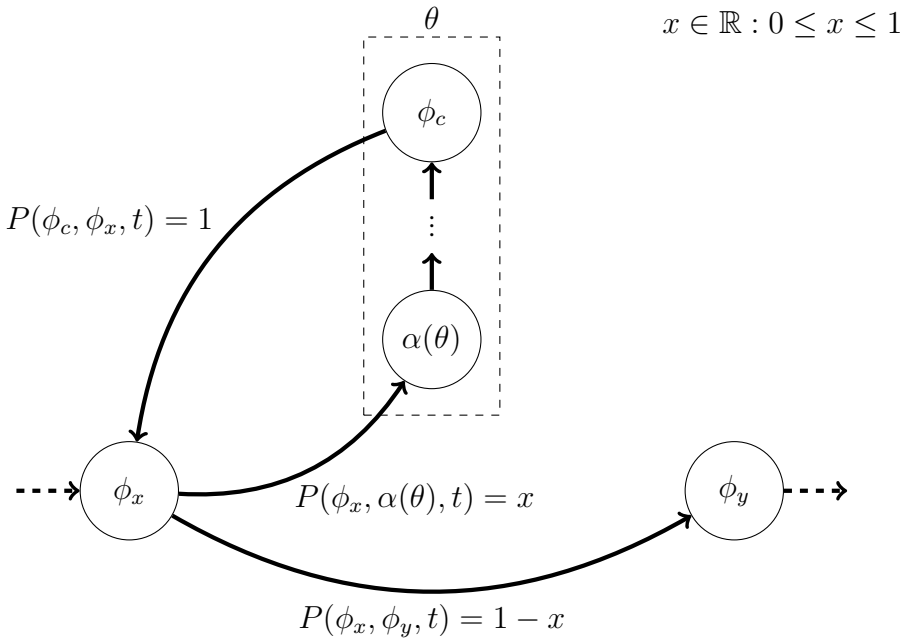

Figure 4.2: Conditional loop structure in a serverless workflow.

$$p = P(\phi_x, \phi_y, t) = 1 - P\Big(\phi_x, \alpha(\theta), t\Big) \tag{4.67}$$

Then:

$$\begin{aligned} P(I_\theta = k) &= p \cdot (1-p)^{k-1} \\ &= pq^{k-1} \\ &= \Big[1 - P\Big(\phi_x, \alpha(\theta), t\Big)\Big] \cdot \Big[P\Big(\phi_x, \alpha(\theta), t\Big)\Big]^{k-1} \end{aligned} \tag{4.68}$$

At any time t, the expected value regarding the number of iterations of $\mathcal{L}$, involving the execution of the sub-choreography θ, can be computed as follows:

$$
\begin{aligned}
E[I_\theta(t)] &= \sum_{k=1}^{\infty}(k-1)pq^{k-1} \\
&= p\sum_{k=0}^{\infty} kq^k \\
&= p \cdot \frac{q}{(1-q)^2} \\
&= \frac{q}{p} \\
&= \frac{P\Big(\phi_x, \alpha(\theta), t\Big)}{1 - P\Big(\phi_x, \alpha(\theta), t\Big)}
\end{aligned} \tag{4.69}
$$

At any time t, the response time and the cost of the conditional loop structure $\mathcal{L}$ can be computed as follows:

$$
RT(\mathcal{L}, \mathbf{x}_\mathcal{C}, t) \stackrel{def}{=} E[I_\theta(t)] \cdot RT(\theta_i, \mathbf{x}_\mathcal{C}, t) \tag{4.70}
$$

$$
C(\mathcal{L}, \mathbf{x}_\mathcal{C}, t) \stackrel{def}{=} E[I_\theta(t)] \cdot C(\theta_i, \mathbf{x}_\mathcal{C}, t) \tag{4.71}
$$

The exit probability of a conditional loop structure was been modeled, once again, exploiting the geometric distribution with success probability equal to $P_{exit}(\theta, t)$. To be precise, exit probability is modeled as the probability that the termination of the execution of the choreography to which $\mathcal{L}$ belongs requires $\lfloor E[I_\theta(t)] \rfloor$ executions of $\mathcal{L}$. Formally:

$$
P_{exit}(\mathcal{L}, t) \stackrel{def}{=} \Big(1 - P_{exit}(\theta, t)\Big)^{\lfloor E[I_\theta(t)] \rfloor - 1} \cdot P_{exit}(\theta, t) \tag{4.72}
$$

4.4.3 Branch

Any structure $\mathcal{B} = (\Phi', E', \Theta)$ such that:

$$\mathcal{T}(\alpha(\mathcal{B})) = \mathcal{T}(\phi_x) = \tau_\alpha \tag{4.73}$$

$$\mathcal{T}(\omega(\mathcal{B})) = \mathcal{T}(\phi_y) = \tau_\omega \tag{4.74}$$

$$|succ(\alpha(\mathcal{P}))| \geq 2 \tag{4.75}$$

$$|prev(\omega(\mathcal{P}))| \leq |succ(\alpha(\mathcal{P}))| \tag{4.76}$$

$$|\Theta| = n \qquad n \in \mathbb{N} \setminus \{0\} \tag{4.77}$$

$$P(\phi_x, \alpha(\theta_i), t) \leq 1 \qquad \forall i \in \mathbb{N} \cap [1, n] \tag{4.78}$$

$$\sum_{i=1}^{n} P(\phi_x, \alpha(\theta_i), t) = 1 \tag{4.79}$$

$$E[I_\theta] = 1 \qquad \forall \theta \in \Theta \tag{4.80}$$

$$\tag{4.81}$$

is called *branch structure* and τ denotes a *branch* type. More precisely, if $|\Theta| = 2$, $\mathcal{B}$ will be called *if-else-branch structure*; if $|\Theta| = 2$, $\mathcal{B}$ will be called *switch-branch structure*. When $|\Theta| = 1$ and $(\phi_x, \phi_y) \in E$, $\mathcal{B}$ is called *if-branch structure*. Clearly, it is used to model `if`, `if-else` and `switch` programming structures inside a serverless application. It is not necessary that $\omega(\mathcal{B})$ exists.

Since any branch have the same possibility of be traversed during different execution of $\mathcal{B}$, we have adopted naive probability to model transition probabilities. Formally:

$$P(\phi_x, \alpha(\theta), t) = \frac{|I_\theta(t)|}{|I_\mathcal{B}(t)|} \qquad \forall \theta \in \Theta \tag{4.82}$$

where $|I_\theta|$ ($|I_\mathcal{B}|$) denotes how many times the sub-choreography θ (structure $\mathcal{B}$) was been executed in the past at time t.

Let $\theta_i \in \Theta$ the i-th sub-choreography of $\mathcal{B}$, where $i \in \mathbb{N} \cap [1, n]$. At any time t, the response time and the cost of the branch structure $\mathcal{B}$ can be computed as follows:

$$RT(\mathcal{B}, \mathbf{x}_\mathcal{C}, t) \stackrel{def}{=} \sum_{i=1}^{n} P(\phi_x, \alpha(\theta_i), t) \cdot RT(\theta_i, \mathbf{x}_\mathcal{C}, t) \tag{4.83}$$

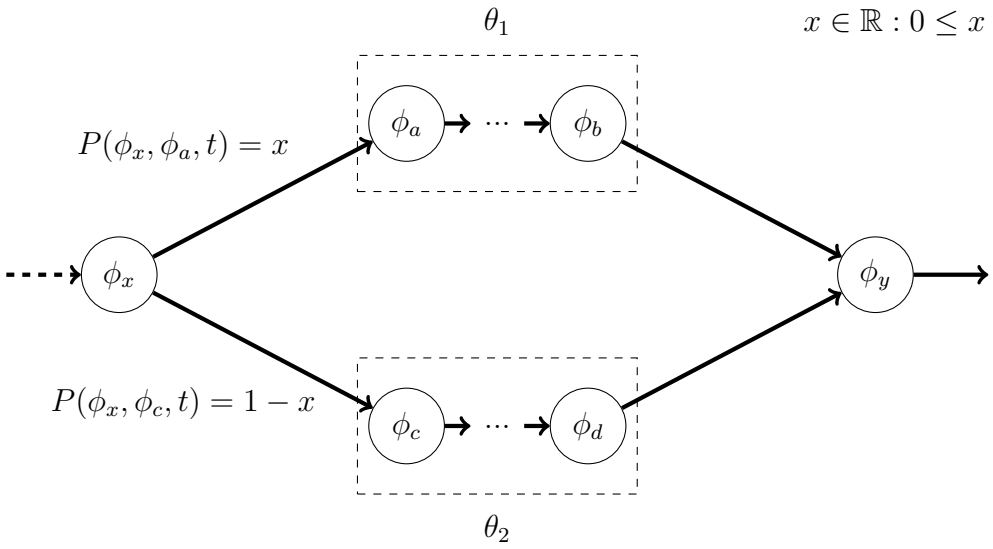

Figure 4.3: An if-else-branch structure in a serverless workflow

$$C(\mathcal{B}, \mathbf{x}_\mathcal{C}, t) \stackrel{def}{=} \sum_{i=1}^{n} P(\phi_x, \alpha(\theta_i), t) \cdot C(\theta_i, \mathbf{x}_\mathcal{C}, t) \tag{4.84}$$

The exit probability of a branch structure can be computed as follows:

$$P_{exit}(\mathcal{B}, t) \stackrel{def}{=} \sum_{i=1}^{n} P(\phi_x, \alpha(\theta_i), t) \cdot P_{exit}(\theta_i, t) \tag{4.85}$$

4.5 Performance Evaluation of Concrete Functions

Clearly, in order to select an appropriate serverless choreography configuration able to effectively guarantee SLA requirements specified by users, we need firstly to evaluate the performance of concrete functions, in terms of average values regarding response time and charged costs.

However, to develop both an analytical way and a software framework capable to evaluate concrete serverless function performance, we first need to understand how they are managed by FaaS platforms.

4.5.1 FaaS Platform Modeling

According to our model point of view, at any time t, any FaaS platform provider acts as a "*set*" of $M/G/K(t)_{\mathbf{C}_{max}}/K(t)_{\mathbf{C}_{max}}$ queueing systems, also called $K(t)$-*server loss systems*, where:

- $\mathbf{C}_{max} \in \mathbb{N} \setminus \{0\}$ is a scalar representing the queuing system's concurrency limit.

  In other words, $\mathbf{C}_{max}$ is the maximum size of the warm pool, therefore it represents the maximum number of function instances being in running state simultaneously at time t.

  Please note that $\mathbf{C}_{max}$ not necessary coincide with a global concurrency limit because it can represents a local concurrency limit too. We will give more details about this very soon.

- At any time $t \geq 0$, there are $K(t)$ function instances. Since each instance can process only one request, we can say that the system has capacity for $K(t)$ invocation requests in total. The number of function instance, and consequently system's capacity, can change over time.

  At any time t, following condition must be hold:

$$0 \leq K(t) \leq C_{\mathbf{max}} \tag{4.86}$$

- Since no queue is involved, if the maximum concurrency level is reached, that is $K(t) = C_{\mathbf{max}}$, any incoming request at time t, that sees all $K(t)$ function instances in running state, will be permanently dropped [2].

- No priority is considered among incoming request.

- Service times have a general distribution while a Poisson arrival process is assumed.

[2]Indeed, a queue exists; there are previous studies stating that all FaaS platform providers adopt a scheduling policies based on *first-come-first-served* (FIFO) algorithms. Anyways, since the built of a QoS-aware scheduler is out of the scope of this dissertation, in order to meet QoS objectives, we have decided to drop any request exceeding maximum concurrency level instead of relying on scheduling policies provided by FaaS platforms.

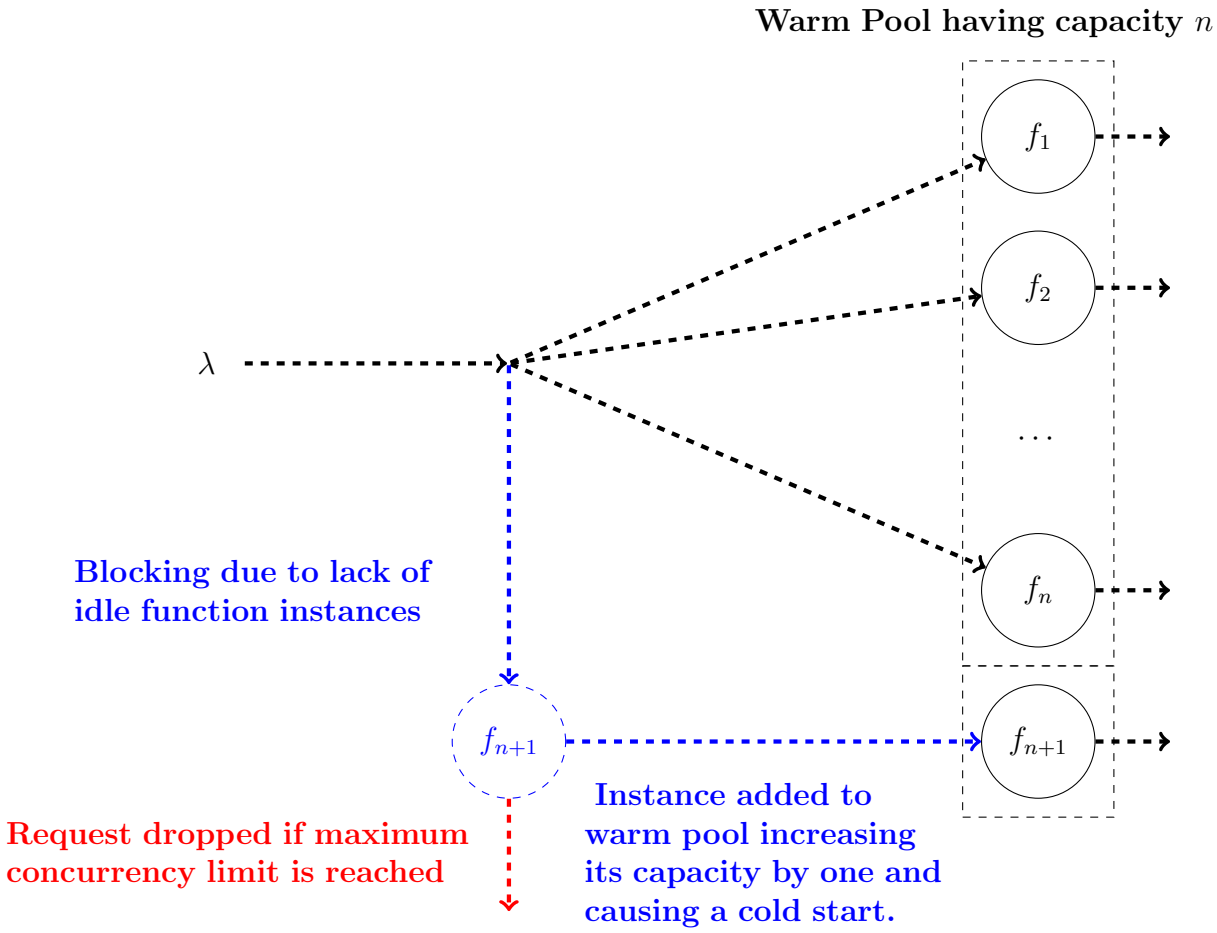

Figure 4.4: An overview of the proposed system model using $M/G/K(t)_{\mathbf{C}_{max}}/K(t)_{\mathbf{C}_{max}}$ queening systems.

4.5.2 Concrete Function Swarm

To complete the formalization of our model and understand how concrete function performances are evaluated, it is necessary to introduce another very important concept to make our model compatible with all concurrency limit models adoptable by FaaS providers.

Let $l \in \mathbb{N}$, R a resource owner, P a serverless computing platform provider and Ω_{P_R} the set of all concrete serverless functions hosted on P belonging to R.

A *concrete function swarm*, or simply *swarm*, is the set $\omega_{P_R}^{(l)} \subseteq \Omega_{P_R}$ containing all concrete functions sharing the same limit l in term of the max number of serverless function instance runnable at the same time by P; in that case, we call l as *swarm's concurrency limit*.

In simple terms, only at most l executions of any concrete functions belonging to $\omega_{P_R}^{(l)}$ can be performed simultaneously by P. The value of l depends on the concurrency model adopted by P.

- If P imposed a *per-function* limit, then $|\omega_{P_{R_l}}| = 1$, that is, $\omega_{P_R}^{(l)}$ contains only one function and l will represent the provider's per-function limit.

  In that case, a dedicated $M/G/K(t)_l/K(t)_l$ queuing system corresponds to each concrete function belonging to $\omega_{P_R}^{(l)}$, which will serve all invocation request regarding only its corresponding concrete function.

- If P imposed a *per-account* limit, then $\omega_{P_R}^{(l)} = \Omega_{P_R}$, that is the swarm includes all concrete function deployed on P by R, while l represents the provider's global concurrency limit.

  In that case, there is only one $M/G/K(t)_l/K(t)_l$ queuing system serving all invocation requests regarding any concrete function belonging to Ω_{P_R}.

From now, the sets $\omega_{P_R}^{(l)}$ and Ω_{P_R} will be briefly and respectively denoted by $\omega_P^{(l)}$ and Ω_P when no confusion can arise about the resource owner R. Moreover, we will adopt $\mathbf{Q}_{\omega_P^{(l)}}$ notation to refer to the $M/G/K(t)_l/K(t)_l$ queuing system serving all invocation requests of any concrete function belonging to ω_P.

4.5.2.1 Cold Start Probability

According to our model, if maximum concurrency level is not exceeded, the rejection of a request by the warm pool will trigger a cold start, adding a new function instance to the warm pool in order to handle aforementioned request.

Now we have to estimate the probability according to which a request is rejected by the warm pool; in other words, we must to compute the *cold start probability*.

Formally, let $l \in \mathbb{N}$, P a serverless computing platform provider, $\omega_P^{(l)}$ a swarm and $\mathbf{Q}_{\omega_P^{(l)}}$ its corresponding queuing system on the FaaS platform provider. Following functions will be used:

- $R(\mathbf{Q}_{\omega_P^{(l)}}, t)$ representing the function returning the number of running state function instances at time t on $\mathbf{Q}_{\omega_P^{(l)}}$.
- $K(\mathbf{Q}_{\omega_P^{(l)}}, t)$ returning instead the number of function instances at time t deployed on $\mathbf{Q}_{\omega_P^{(l)}}$.

In this section, to simplify our notations, since no confusion can arise about the queuing system, $R(\mathbf{Q}_{\omega_P^{(l)}}, t)$ will be briefly denoted as $R(t)$ and $K(\mathbf{Q}_{\omega_P^{(l)}}, t)$ as $K(t)$.

When a new invocation request of any function belonging to $\omega_P^{(l)}$ arrives on $\mathbf{Q}_{\omega_P^{(l)}}$, one of the following events can occur:

- If $K(t_n) < l$ and $R(t_n) < K(t_n)$, that is the number of function instances into the warm pool is less than the swarm maximum concurrency level and there are some function instance in idle state, a warm start will occur, resulting that $R(t_{n+1}) = R(t_n) + 1$.
- If $K(t_n) < l$ and $R(t_n) = K(t_n)$, that is the number of function instances into the warm pool is once again less than the swarm maximum concurrency level but there is no function instance in idle state, a cold start will occur and the size of the warm pool will be increased by one, that is $K(t_{n+1}) = K(t_n) + 1$.

  The accepted request will be managed by the newly spinned up function instance, causing that $R(t_{n+1}) = R(t_n) + 1$.
- If $K(t) = R(t) = l$, that is the warm pool reaches its maximum possible size according to FaaS policies, aforementioned request will be rejected.

If at time t_n a concrete function execution has been completed, that event will cause $R(t_{n+1}) = R(t_n) - 1$.

Conversely, if a function instance has not received any request during the last expiration threshold units of time, it will expire causing the decreasing of warm pool size, that is $K(t_{n+1}) = K(t_n) - 1$.

Finally, at any time t, the *cold start probability* $\mathbf{P}_{\omega_{P_R}^{(l)}}(t)$ referring to the invocations of any function belonging to the swarm $\omega_{P_R}^{(l)}$, can be formally defined as the probability that an arrival request of invocation finds all function instances of

the warm pool busy; using Erlang-B formula, that probability can be calculated as follows:

$$\mathbf{P}_{\omega_{P_R}^{(l)}}(t) = \frac{\rho(t)^{K(t)}}{K(t)!} \cdot \left(\sum_{j=0}^{K(t)} \frac{\rho(t)^j}{j!} \right)^{-1} \tag{4.87}$$

where:

- $\rho(t) = \dfrac{\lambda(t)}{\mu_w}$ represents system utilization or load of the warm pool at time t where:
  - $\lambda(t)$ represents the *average arrival rate* at time t, that is the rate at which invocation requests, regarding serverless concrete functions belonging to $\omega_{P_R}^{(l)}$, arrive to our system. It is expressed in $invocations \cdot s^{-1}$.

    Since arrival rate varies during the day, $\lambda(t)$ must be estimated at runtime, over multiple time intervals spent observing our system.

    We adopt the exponential moving average based approach to compute an estimation of $\lambda(t)$. Periodically, at any time t, our framework computes the number Y_t of all received invocation requests for any function belonging to $\omega_{P_R}^{(l)}$ within last second. Then $\lambda(t)$ is estimated as follows:

$$\lambda(t) \stackrel{def}{=} \begin{cases} Y_o & \text{if } t = 0 \\ \alpha \cdot Y_t + (1-\alpha) \cdot \lambda(t-1) & \text{if } t > 0 \end{cases}. \tag{4.88}$$

  - $\mu_w = E[S_w]$ represents the *warm start average service rate*, that is the rate at which executions requests are served when a warm start occurs, while $E[S_w]$ is the *warm start average time*, which is the average time required to complete aforementioned request.

Finally, in order to simplify the calculation of the expression 4.87, according to [46], it can be expressed recursively as follows:

$$
\begin{aligned}
B(\rho(t),0) &= 1 && (4.89)\\
B(\rho(t),i) &= \frac{\rho(t)\cdot B(\rho(t),i-1)}{\rho(t)\cdot B(\rho(t),i-1)+i} \qquad \forall j \in \mathbb{N}\cap[1,K(t)] && (4.90)\\
& && (4.91)
\end{aligned}
$$

where:

$$
B(\rho(t),i) \stackrel{def}{=} \frac{\rho(t)^i}{i!}\cdot\left(\sum_{j=0}^{i}\frac{\rho(t)^j}{j!}\right)^{-1} \tag{4.92}
$$

4.5.2.2 Choreography Swarm

Let $\mathcal{C}=(\Phi,E)$ a choreography, to build its configuration $\mathbf{x}_{\mathcal{C}}$, we first need to identify all swarms corresponding to each concrete function $f_\phi \in \mathbf{F}_\phi$, for all $\phi \in \mathcal{F}_\mathcal{E}$, because we have to check if we can execute all concrete functions specified in $\mathbf{x}_{\mathcal{C}}$, without exceeding maximum concurrency level on each FaaS platforms where aforementioned functions will be executed.

Informally, a *choreography swarm* $\widetilde{\mathbf{S}}_{\mathcal{C}}$ represents the set of all swarms corresponding to all concrete functions used by $\mathcal{C}$.

Formally, let $m \in \mathbb{N}$ the total number of FaaS platform providers where at least one concrete function is deployed by a given resource owner, a choreography swarm can be defined as follows:

$$
\widetilde{\mathbf{S}}_{\mathcal{C}} \stackrel{def}{=} \left\{ \omega_P^{(l)} \in \bigcup_{i=1}^{n}\Omega_{P_i} : \exists f_\phi \in \omega_P^{(l)} \text{ such that } f_\phi \in \mathbf{F}_\phi, \forall \phi \in \mathcal{F}_\mathcal{E}(\mathcal{C}) \right\} \tag{4.93}
$$

Please note that, let $\mathcal{C}_1=(\Phi_1,E_1)$ and $\mathcal{C}_2=(\Phi_2,E_2)$ two different choreographies belonging to the same resource owner, it is generally verified that:

$$
\widetilde{\mathbf{S}}_{\mathcal{C}_1} \cap \widetilde{\mathbf{S}}_{\mathcal{C}_2} \neq \emptyset \tag{4.94}
$$

that is a swarm can be shared by multiple choreographies, therefore any imple-

mentation of our model has to pay attention on possible race conditions.

Finally, we will use following function:

$$N(\mathcal{C}, \mathbf{x}_\mathcal{C}, \omega_P^{(l)}) = \sum_{x_\phi \in \mathbf{x}_\mathcal{C}} \sum_{\substack{f_\phi \in x_\phi \\ f_\phi \in \omega_P^{(l)}}} 1 \tag{4.95}$$

that is, $N(\mathcal{C}, \mathbf{x}_\mathcal{C}, \omega_P^{(l)}, t)$ denotes the number of concrete functions used in $\mathbf{x}_\mathcal{C}$

4.5.3 Concrete Function Performance Computation

Finally, we can introduce how concrete function performance can be computed according to our model.

Let $x_\phi = (f_\phi, m)$ an executable function configuration such that $f_\phi \in \omega_{P_R}^{(l)} \subset \mathbf{F}_\phi$ represents a concrete function implementing the executable function $\phi \in \mathcal{F}_\mathcal{E}(\mathcal{C})$ and belonging to the swarm $\omega_{P_R}^{(l)}$, where $l \in \mathbb{N}$ is its concurrency limit. Moreover, $m \in \mathbf{M}_{f_\phi} \subseteq \mathbb{N}$ denotes the value of the memory size selected by our framework for the execution of f_ϕ.

Let's define following functions:

- $C : \mathbf{F}_\phi \times \mathbf{M}_{f_\phi} \times \mathbb{R}^+ \to \mathbb{R}^+$ is the *cost function* for any serverless concrete functions belonging to the implementation-set $\mathbf{F}_\phi$.

  It returns the *average cost* paid by users when f_ϕ is executed using an allocated memory size equal to m at time t.

  It can be defined as follows:

$$\begin{aligned} C(x_\phi, t) &\stackrel{def}{=} C(f_\phi, m, t) \\ &= C_{avg}^{(c)}(f_\phi, m, t) \cdot \mathbf{P}_{\omega_{P_R}^{(l)}}(t) + \\ &\qquad C_{avg}^{(w)}(f_\phi, m, t) \cdot \left(1 - \mathbf{P}_{\omega_{P_R}^{(l)}}(t)\right) \end{aligned} \tag{4.96}$$

  where:

– At any time t. when f_ϕ is executed with an allocated memory size equal to m, $C^{(c)}_{avg}(f_\phi, m, t)$ represents the average cost paid when a cold start is occurred while, conversely, $C^{(w)}_{avg}(f_\phi, m, t)$ represents the average cost paid in case of warm start.

- $RT : \mathbf{F}_\phi \times \mathbf{M}_{f_\phi} \times \mathbb{R}^+ \to \mathbb{R}^+$ is the *delay function* for any serverless concrete functions belonging to the implementation-set $\mathbf{F}_\phi$.

  It returns the *average response time* paid by users when f_ϕ is executed using an allocated memory size equal to m at time t.

  It can be defined as follows:

$$\begin{aligned} RT(x_\phi, t) &\stackrel{def}{=} RT(f_\phi, m, t) \\ &= RT^{(c)}_{avg}(f_\phi, m, t) \cdot \mathbf{P}_{\omega^{(l)}_{P_R}}(t) + \\ &\qquad RT^{(w)}_{avg}(f_\phi, m, t) \cdot \left(1 - \mathbf{P}_{\omega^{(l)}_{P_R}}(t)\right) \end{aligned} \tag{4.97}$$

  where:

  – $RT^{(c)}_{avg}(f_\phi, m, t)$ represents the average response time occurred when a cold start is occurred and $RT^{(w)}_{avg}(f_\phi, m, t)$ is the one occurred in case of in case of warm start.

Obviously, our framework has to determine $RT^{(c)}_{avg}(f_\phi, m, t)$, $RT^{(w)}_{avg}(f_\phi, m, t)$, $C^{(c)}_{avg}(f_\phi, m, t)$ and $C^{(w)}_{avg}(f_\phi, m, t)$ using time series data, containing historical performance data of f_ϕ under different memory configurations, collected by our logging framework. To compute aforementioned estimations, we adopt the exponential moving average based approach.

For instance, referring to the computation of $RT^{(c)}_{avg}(f_\phi, m, t)$, let Y_t the response time of f_ϕ when it is executed with memory size m at a given time t, following formula is used:

$$RT^{(c)}_{avg}(f_\phi, m, t) \stackrel{def}{=} \begin{cases} Y_o & \text{if } t = 0 \\ \alpha \cdot Y_t + (1 - \alpha) \cdot \lambda(t-1) & \text{if } t > 0 \end{cases}. \tag{4.98}$$

4.6 Executable Function Performance Evaluation

In this section, we will describe how to evaluate the performance of any executable serverless function, in terms of average values regarding response time and charged costs, when it is executed.

Supposing to have a choreography $\mathcal{C} = (\Phi, E)$, let $x_{\phi_{i_j}} = (f_j, m_j)$ an executable function configuration for $\phi_i \in \mathcal{F}_\mathcal{E}(\mathcal{C})$, for some $i \in \mathbb{N} \cap [1, |\mathcal{F}_\mathcal{E}(\mathcal{C})|]$ and $j \in \mathbb{N} \cap [1, |\mathbf{F}_{\phi_i} \times \mathbb{N}|]$. In order to evaluate the performance of ϕ, we have to introduce some useful notations and functions as follows:

- For some $n \in \mathbb{N}$, $\widehat{\theta_k}^{(\phi_i)} \stackrel{def}{=} \left\{\theta_1^{(\phi_i)}, \ldots, \theta_n^{(\phi_i)}\right\}$ represents the set of all sub-choreographies of $\mathcal{C}$ such that:

$$\phi_i \in \mathcal{F}_\mathcal{E}(\theta_1^{(\phi_i)}) \tag{4.99}$$

$$\theta_n^{(\phi_i)} = \mathcal{C} \tag{4.100}$$

$$\theta_k^{(\phi_i)} = (\Phi_k, E_k) \text{ is a sub-choreography of } \theta_{k+1}^{(\phi_i)} \qquad \forall k \in [1; n-1] \tag{4.101}$$

- $E[I_{\phi_i}(t)]$ represents the expected value regarding the number of invocation of the executable function ϕ_i at time t, which can be defined as follows.

$$E[I_{\phi_i}(t)] \stackrel{def}{=} \begin{cases} 1 & if \quad n = 1 \\ \prod_{k=2}^{n} E[I_{\theta_{k-1}^{(\phi_i)}}(t)] & if \quad n \geq 2 \end{cases} \tag{4.102}$$

- $P_{exe}^C(\theta_k^{(\phi_i)}, t)$ is the probability to execute the entry point $\alpha(\theta_k^{(\phi_i)})$ of the choreography $\theta_k^{(\phi_i)}$ when the function $pred(\alpha(\theta_k^{(\phi_i)}))$ is executed at time t. Obviously, it is equal to the transition probability associated whit the edge starting from $pred(\alpha(\theta_k^{(\phi_i)}))$ and ending to $\alpha(\theta_k^{(\phi_i)}))$. Formally:

$$P_{exe}^{C}(\theta_k^{(\phi_i)}, t) \stackrel{def}{=} \begin{cases} 1 & if \quad n = 1 \\ P\Big(pred\Big(\alpha(\theta_k^{(\phi_i)})\Big), \alpha(\theta_k^{(\phi_i)})\Big) & if \quad n \geq 2 \end{cases} \tag{4.103}$$

- $P_{exe}^{F}(\theta_k^{(\phi_i)}, \phi, t)$ is the probability to execute $\phi \in \mathcal{F}_{\mathcal{E}}(\theta_k)$ when $\alpha(\theta_k)$ is executed at time t.

  In order to compute this probability, it is necessary to process any orchestration function $\phi_x \in \Pi(\alpha(\theta_k), \phi)$, in order to obtain a sort of pipeline choreography from $\alpha(\theta_k)$ to ϕ.

  Then, after performing the workflow simplification process, we will obtain a set $\Phi'_k = \{\phi_1, \ldots, \phi_n\} \subseteq \Phi_k$ such that:

$$succ(\phi_x) = \phi_{x+1} \qquad x \in \mathbb{N} \cap [1, n-1] \tag{4.104}$$

$$\phi_n = \phi \tag{4.105}$$

  Finally:

$$P_{exe}^{F}(\theta_k^{(\phi_i)}, \phi, t) \stackrel{def}{=} \prod_{i=1}^{n-1} (1 - P_{exit}(\phi_i, t)) \tag{4.106}$$

- We will use $\Gamma_{\theta_k^{(\phi_i)}}(t)$ to denote the probability to execute the entry point of the sub-choreography $\theta_k^{(\phi_i)}$ at time t.

$$\Gamma_{\theta_k^{(\phi_i)}}(t) \stackrel{def}{=} P_{exe}^{C}(\theta_k^{(\phi_i)}, t) \cdot \left(\prod_{y=k+1}^{n} P_{exe}^{C}(\theta_y^{(\phi_i)}, t) \cdot P_{exe}^{F}(\theta_y^{(\phi_i)}, pred(\alpha(\theta_{y-1}^{(\phi_i)})), t) \right) \tag{4.107}$$

Finally, we can compute response time $rt_{\phi_{i_j}}(t)$ and charged cost $c_{\phi_{i_j}}(t)$, regarding the executable function $\phi_i \in \mathcal{F}_{\mathcal{E}}(\mathcal{C})$ with configuration $x_{\phi_{i_j}}$, using following formulas:

$$c_{\phi_{i_j}}(t) \stackrel{def}{=} \begin{cases} C(x_{\phi_{i_j}}, t) \cdot P^F_{exe}(\theta_1^{(\phi_i)}, \phi_i, t) & \textbf{if} \quad m = 1 \\ C(x_{\phi_{i_j}}, t) \cdot P^F_{exe}(\theta_1^{(\phi_i)}, \phi_i, t) \cdot E[I_{\phi_i}(t)] \cdot \Gamma_{\theta_1^{(\phi_i)}}(t) & \textbf{if} \quad m \geq 2 \end{cases} \tag{4.108}$$

$$rt_{\phi_{i_j}}(t) \stackrel{def}{=} \begin{cases} RT(x_{\phi_{i_j}}, t) \cdot P^F_{exe}(\theta_1^{(\phi_i)}, \phi_i, t) & \textbf{if} \quad m = 1 \\ RT(x_{\phi_{i_j}}, t) \cdot P^F_{exe}(\theta_1^{(\phi_i)}, \phi_i, t) \cdot E[I_{\phi_i}(t)] \cdot \Gamma_{\theta_1^{(\phi_i)}}(t) & \textbf{if} \quad m \geq 2 \end{cases} \tag{4.109}$$

Let $\langle (RT, w_{RT}), (C, w_C) \rangle$ the SLAs constraints specified by a customer, for any executable function configuration $x_{\phi_{i_j}}$, its *score* $p_{\phi_{i_j}}(t)$ is determined as follows:

$$p_{\phi_{i_j}}(t) \stackrel{def}{=} w_{RT} \cdot p_{\phi_{i_j}}(t)^{(rt)} + w_C \cdot p_{\phi_{i_j}}(t)^{(c)} \tag{4.110}$$

where $p_{\phi_{i_j}}(t)^{(rt)}$ and $p_{\phi_{i_j}}(t)^{(c)}$ represent, respectively, the *response time score* and *cost score* which can be computed as follows:

$$p_{\phi_{i_j}}(t)^{(rt)} \stackrel{def}{=} \begin{cases} 1 & \textbf{if} \quad rt_{\phi_{i_{\mathbf{MAX}}}}(t) = rt_{\phi_{i_{\mathbf{MIN}}}}(t) \\ \dfrac{rt_{\phi_{i_{\mathbf{MAX}}}}(t) - rt_{\phi_{i_j}}(t)}{rt_{\phi_{i_{\mathbf{MAX}}}}(t) - rt_{\phi_{i_{\mathbf{MIN}}}}(t)} & otherwise \end{cases} \tag{4.111}$$

$$p_{\phi_{i_j}}(t)^{(c)} \stackrel{def}{=} \begin{cases} 1 & \textbf{if} \quad c_{\phi_{i_{\mathbf{MAX}}}}(t) = c_{\phi_{i_{\mathbf{MIN}}}}(t) \\ \dfrac{c_{\phi_{i_{\mathbf{MAX}}}}(t) - c_{\phi_{i_j}}(t)}{c_{\phi_{i_{\mathbf{MAX}}}}(t) - c_{\phi_{i_{\mathbf{MIN}}}}(t)} & otherwise \end{cases} \tag{4.112}$$

where:

- $rt_{\phi_{i_{\mathbf{MIN}}}}(t)$ and $rt_{\phi_{i_{\mathbf{MAX}}}}(t)$ represent, respectively, the minimum and maximum response time values regarding the execution of all concrete function implementing ϕ_i.

  In other words, they represent the minimum and maximum resource requirement, referring to the response time, of all items belonging to the i-th group

of our problem.

- $c_{\phi_{i_{\mathbf{MIN}}}}(t)$ and $c_{\phi_{i_{\mathbf{MAX}}}}(t)$ represent, respectively, the minimum and maximum cost values spent by all concrete function implementing ϕ_i.

4.7 Choreography Performance Evaluation

In this section, we will briefly explore the analytical methodology to compute the average end-to-end response time and charged cost of generic serverless applications, having branches, loops or even workflow portions executed in parallel, when a choreography configuration is given.

4.7.1 Pipeline Choreography Performance

Let $\mathcal{C} = (\Phi, E)$ a pipeline choreography as defined in section 4.2.2.4 and $\mathbf{x}_\mathcal{C} \in \mathbf{X}_\mathcal{C}$ a choreography configuration. Fortunately, exploiting pipeline choreography's properties, it is very simple to obtain a performance evaluation when $\mathbf{x}_\mathcal{C}$ is given.

Let $\Phi = \{\phi_1, \ldots, \phi_n\}$ such that:

$$succ(\phi_x) = \phi_{x+1} \qquad x \in \mathbb{N} \cap [1, n-1] \tag{4.113}$$

Then, at any time t, following equations can be used:

$$RT(\mathcal{C}, \mathbf{x}_\mathcal{C}, t) \stackrel{def}{=} RT(x_{\phi_1}, t) + \sum_{x=2}^{n} P^F_{exe}(\mathcal{C}, \phi_{x-1}, t) \cdot RT(x_{\phi_x}, t) \tag{4.114}$$

$$C(\mathcal{C}, \mathbf{x}_\mathcal{C}, t) \stackrel{def}{=} C(x_{\phi_1}, t) + \sum_{x=2}^{n} P^F_{exe}(\mathcal{C}, \phi_{x-1}, t) \cdot C(x_{\phi_x}, t) \tag{4.115}$$

4.7.2 Generic Choreography Performance

Unfortunately, equations 4.114 and 4.115 can be used only dealing with pipeline type choreographies, therefore they are not suitable for general choreographies having branch or loops.

However, exploiting what was said before regarding choreography structures, it is very simple to convert a generic choreography into a pipeline type one. A very naive iterative approach is reported in Algorithm 1.

Algorithm 1: Pseudo-code regarding a possible iterative approach to convert a generic choreography into a pipeline type one

```
1  C* ← C;
2  while C* is not a pipeline choreography do
3  |   branchList ← find_branches(C*);
4  |   for B in branchList do
5  |   |   C* ← process(B);
6  |   end
7  |   parallelList ← find_parallels(C*);
8  |   for P in parallelList do
9  |   |   C* ← process(P);
10 |   end
11 |   loopList ← find_loops(C*);
12 |   for L in loopList do
13 |   |   C* ← process(L);
14 |   end
15 end
16 return C*
```

Finally, in order to select the better one, to each choreography configuration is associated a score, like for executable function configurations.

Formally, let $\mathcal{C} = (\Phi, E)$ a serverless choreography, $\mathbf{x}_\mathcal{C} \in \mathbf{X}_\mathcal{C}$ a choreography configuration and $\langle (RT, w_{RT}), (C, w_C) \rangle$ the SLA. At any time t, the *score* $S(\mathbf{x}_\mathcal{C}, t)$, or *profit*, of the choreography configuration $\mathbf{x}_\mathcal{C}$ can be computed as follows:

$$S(\mathbf{x}_\mathcal{C}, t) \stackrel{def}{=} w_{RT} \cdot S(\mathbf{x}_\mathcal{C}, t)^{(RT)} + w_C \cdot S(\mathbf{x}_\mathcal{C}, t)^{(C)} \tag{4.116}$$

where $S(\mathbf{x}_\mathcal{C}, t)^{(RT)}$ and $S(\mathbf{x}_\mathcal{C}, t)^{(C)}$ represent, respectively, the *response time score* and *cost score* which can be computed as follows:

$$S(\mathbf{x}_{\mathcal{C}}, t)^{(RT)} \stackrel{def}{=} \begin{cases} 1 & RT_{\mathbf{max}}(t) = RT_{\mathbf{min}}(t) \\ \dfrac{RT_{\mathbf{max}}(t) - RT(\mathbf{x}_{\mathcal{C}}, t)}{RT_{\mathbf{max}}(t) - RT_{\mathbf{min}}(t)} & otherwise \end{cases} \quad (4.117)$$

$$S(\mathbf{x}_{\mathcal{C}}, t)^{(C)} \stackrel{def}{=} \begin{cases} 1 & C_{\mathbf{max}}(t) = C_{\mathbf{min}}(t) \\ \dfrac{C_{\mathbf{max}}(t) - C(\mathbf{x}_{\mathcal{C}}, t)}{C_{\mathbf{max}}(t) - C_{\mathbf{min}}(t)} & otherwise \end{cases} \quad (4.118)$$

where:

- $RT_{\mathbf{max}}(t)$ ($RT_{\mathbf{min}}(t)$) and $C_{\mathbf{max}}(t)$ ($C_{\mathbf{min}}(t)$) denote, respectively, the maximum (minimum) value of the response time and billed cost at time t, observed using all possible configuration in $\mathbf{X}_{\mathcal{C}}$.

Chapter 5

Optimization Problem Formulations

To achieve our goal consisting in finding the best choreography configuration capable to guarantee QoS constraints imposed by an user, we have to solve an optimization problem, which we have formulated in two different manners, where:

- the first one is based on the *Multi-Dimensional Knapsack Problem Formulation*.
- the second one is based on the *Multi-Dimensional Multi-Choice Knapsack Problem Formulation*.

In this section, we will describe and analyze both approaches, although only for the second one we have developed and implemented a heuristic approach capable to solve it in a faster way respect to exact algorithms.

5.1 Multidimensional Knapsack Problem Formulation

A very naive formulation to our problem can be expressed in term of the *Multidimensional Knapsack Problem* (MKP), a well-studied, strongly NP-hard combinatorial optimization problem occurring in many different applications, whose goal is to

choose an item, in our case a choreography configuration, having maximum total profit; selected item must, however, not exceed resource capacities, which are called *knapsack constraints*.

Let $\mathcal{C} = (\Phi, E)$ a choreography and the SLA $\langle (RT, w_{RT}), (C, w_C) \rangle$, the MKP formulation for our problem can be defined by the following ILP:

$$
\begin{aligned}
max \quad & \sum_{i=1}^{|\mathbf{x}_\mathcal{C}|} x_i \cdot S(\mathbf{x}_{\mathcal{C}_i}, t) && && (5.1)\\
\text{subject to} \quad & \sum_{\omega=1}^{|\mathbf{x}_\mathcal{C}|} x_i \cdot C(\mathcal{C}, \mathbf{x}_{\mathcal{C}_i}, t) \leq C && && (5.2)\\
& \sum_{i=1}^{|\mathbf{x}_\mathcal{C}|} x_i \cdot RT(\mathcal{C}, \mathbf{x}_\mathcal{C}, t) \leq RT && && (5.3)\\
& \sum_{i=1}^{|\mathbf{x}_\mathcal{C}|} x_i \cdot N(\mathcal{C}, \mathbf{x}_\mathcal{C}, \omega_P^{(l)}) \leq l - R(\mathbf{Q}_{\omega_P^{(l)}}, t) && \forall \omega_P^{(l)} \in \widetilde{\mathbf{S}}_\mathcal{C} \quad (5.4)\\
& \sum_{i=1}^{|\mathbf{x}_\mathcal{C}|} x_i = 1 && && (5.5)\\
& x_i \in \{0, 1\} && \forall i \in \mathbb{N} \cap [1, |\Omega|] \quad (5.6)
\end{aligned}
$$

where:

- x_i is a binary variable set to 1 when $\mathbf{x}_{\mathcal{C}_i}$ is used as configuration for $\mathcal{C}$; otherwise it is set to 0.

It is very important to observe, that each column of above formulation represents the profit and resource requirements referring to a choreography configuration.

The number of existing choreography configurations (or columns of LP) is very large because it grows exponentially. Therefore, it becomes impractical to generate and enumerate all possible choreography configurations.

For example, let's suppose to have a serverless choreography $\mathcal{C}$ made up of 6 executable functions. Supposing to have 46 possible memory choices, even if only one concrete function exists for each executable function, the solution space Ω will

contain 9.47 billion different configurations, making any exhaustive enumeration and search computationally unfeasible.

Even if we had a way of generating all configurations, that is all columns of our problem, due to its complexity, any attempt to find the optimal solution rapidly will be an illusion. Moreover, it will be very likely that any computer runs out his memory.

5.2 Multi-Dimensional Multi-Choice Knapsack Problem Formulation

To avoid the enumeration of all possible choreography configurations, we have decided to use another kind of formulation, which is based on the *Multidimensional Multiple-Choice Knapsack Problem* (MMKP), a variant of the previous one.

Let $n \in \mathbb{N} \setminus \{0\}$ and a choreography $\mathcal{C} = (\Phi, E)$ such that $|\mathcal{F}_{\mathcal{E}}(\mathcal{C})| = n$. For each $\phi_i \in \mathcal{F}_{\mathcal{E}}(\mathcal{C})$ corresponds a *group of items*, where each item corresponds to an executable configurations ϕ_i; therefore, the i-th group contains exactly $|\{\mathbf{F}_{\phi_i} \times \mathbb{N}\}|$ items or executable function configurations.

To resolve our MMKP, we have to pick exactly one item from each group, that is exactly one configuration x_{ϕ_1} for each executable function $\phi \in \mathcal{F}_{\mathcal{E}}(\mathcal{C})$, in such a way to maximize the profit value of the selected items, respecting several resource constraints of the knapsack, which, in our case, are represented both by SLAs attributes values imposed by users and by the actual available capacity of all queuing system associated to each swarm to which all concrete functions, specified in the solution, belong.

To properly describe our MMKP, since it is not trivial to formulate in a LP problem the relationship between executable functions belonging to a same parallel structure, according to which their response times depend on the slower one, we need to introduce the concept which we have called of *parallel path*.

From now, $\widetilde{\mathcal{P}} \stackrel{def}{=} \left\{\widetilde{\mathcal{P}}_1, \ldots, \widetilde{\mathcal{P}}_z\right\}$ represents the set containing all parallel structures of $\mathcal{C}$, while $\mathcal{F}_{\mathcal{E}}(\widetilde{\mathcal{P}})$ represents the set containing all executable functions such that:

$$\mathcal{F}_{\mathcal{E}}(\widetilde{\mathcal{P}}) \stackrel{def}{=} \bigcup_{i=1}^{z} \mathcal{F}_{\mathcal{E}}(\widetilde{\mathcal{P}}_i) \tag{5.7}$$

Let $z \in \mathbb{N} \setminus \{0\}$ and $\Pi(\alpha(\mathcal{C}), \omega(\mathcal{C}))$ the set containing all possible paths from the start point to the end point of $\mathcal{C}$ such that $|\Pi(\alpha(\mathcal{C}), \omega(\mathcal{C}))| = z$.

Formally, being π_i the i-th path belonging to $\Pi(\alpha(\mathcal{C}), \omega(\mathcal{C}))$, the i-th *parallel path* $\delta_{\mathcal{C}_i}$ is defined as follows:

$$\delta_{\mathcal{C}_i} \stackrel{def}{=} \left\{ \phi \in \mathcal{F}_{\mathcal{E}}(\widetilde{\mathcal{P}}) : \phi \in \pi_i \right\} \tag{5.8}$$

Clearly, the set containing all parallel paths of $\mathcal{C}$ is denoted by $\Delta_{\mathcal{C}}$ where:

$$\Delta_{\mathcal{C}} \stackrel{def}{=} \bigcup_{i=1}^{z} \delta_{\mathcal{C}_i} \tag{5.9}$$

Finally, wa are able to formulate our MMKP problem which, supposing to have a choreography $\mathcal{C} = (\Phi, E)$ and the SLA $\langle (RT, w_{RT}), (C, w_C) \rangle$, can be expressed as follows:

$$max \quad \sum_{i=1}^{|\mathcal{F}_{\mathcal{E}}(\mathcal{C})|} \sum_{j=1}^{|\mathbf{F}_{\phi_i} \times \mathbb{N}|} y_{\phi_{i_j}} \cdot p_{\phi_{i_j}}(t) \tag{5.10}$$

$$\text{subject to} \quad \sum_{i=1}^{|\mathcal{F}_{\mathcal{E}}(\mathcal{C})|} \sum_{j=1}^{|\mathbf{F}_{\phi_i} \times \mathbb{N}|} y_{\phi_{i_j}} \cdot c_{\phi_{i_j}}(t) \leq C \tag{5.11}$$

$$\sum_{\phi_i \in \mathcal{F}_{\mathcal{E}}(\mathcal{C}) \setminus \mathcal{F}_{\mathcal{E}}(\widetilde{\mathcal{P}})} \sum_{j=1}^{|\mathbf{F}_{\phi_i} \times \mathbb{N}|} y_{\phi_{i_j}} \cdot rt_{\phi_{i_j}}(t) + \\ + \sum_{\phi_h \in \delta_{\mathcal{C}}} \sum_{j=1}^{|\mathbf{F}_{\phi_h} \times \mathbb{N}|} y_{\phi_{h_j}} \cdot rt_{\phi_{h_j}}(t) \leq RT \qquad \forall \delta_{\mathcal{C}} \in \Delta_{\mathcal{C}} \tag{5.12}$$

$$\sum_{i=1}^{|\mathcal{F}_{\mathcal{E}}(\mathcal{C})|} \sum_{j=1}^{|\mathbf{F}_{\phi_i} \times \mathbb{N}|} y_{\phi_{i_j}} \cdot a_{(\phi_{i_j}, \omega_P^{(l)})} \leq l - R(\mathbf{Q}_{\omega_P^{(l)}}, t) \qquad \forall \omega_P^{(l)} \in \widetilde{\mathbf{S}}_{\mathcal{C}} \tag{5.13}$$

$$\sum_{j=1}^{|\mathbf{F}_{\phi_i} \times \mathbb{N}|} y_{\phi_{i_j}} = 1 \qquad \forall i \in \mathbb{N} \cap [1, |\mathcal{F}_{\mathcal{E}}|] \tag{5.14}$$

$$y_{\phi_{i_j}} \in \{0, 1\} \qquad \begin{matrix} \forall i \in \mathbb{N} \cap [1, |\mathcal{F}_{\mathcal{E}}|] \\ \forall j \in \mathbb{N} \cap [1, |\mathbf{F}_{\phi_i} \times \mathbb{N}|] \end{matrix} \tag{5.15}$$

where:

- All coefficients, or profits, associated to each executable function configuration, that appear inside the objective function 5.10, are computed using the formula 4.110.
- Constraint 5.11 represents the max billed cost requirement, whose coefficients are computed using the formula 4.108.
- The 5.12 represents the max response time requirements. According to our formulation, there are exactly $|\Delta_{\mathcal{C}}|$ constraints regarding the response time,

that is one of each parallel path. Since we don't know a priori the slowest executable function inside a parallel structure, the main idea is to assure us that any path executed in parallel respects response time constraint. Coefficients associated to each variable are computed using the formula 4.109.

- The 5.13 represents the capacity requirement, used to check if there are enough function instances on each system $\mathbf{Q}_{\omega_P^{(l)}}$ corresponding to each swarm of the choreography. The coefficient $a_{(\phi_{i_j},\omega_P^{(l)})}$ is computed as follows:

$$a_{(\phi_{i_j},\omega_P^{(l)})} \stackrel{def}{=} \begin{cases} 1 & if \quad f_{\phi_i} \in x_{\phi_{i_j}} \wedge f_{\phi_i} \in \omega_P^{(l)} \\ 0 & otherwise \end{cases} \tag{5.16}$$

  where $x_{\phi_{i_j}}$ is the j-th executable function configuration for ϕ_i. In simple terms, $a_{(\phi_{i_j},\omega_P^{(l)})}$ is set to 1 only if the corresponding configuration use a concrete function which belongs to the swarm $\omega_P^{(l)}$.

- The 5.14 assures us that only one concrete function configuration is selected for any executable function.

- The 5.15 states that y_i is a binary variable set to 1 when $x_{\phi_{i_j}}$ is used; otherwise it is set to 0.

To simplify our notations, from now, we will use the notation o_{i_j} to represent the i-th item belonging to the j-th group, which is simply denoted by g_j; in other words, the item o_{i_j} represents the j-th executable function configuration used to implement the executable function $\phi_i \in \mathcal{F}_{\mathcal{E}}(\mathcal{C})$, that is the $x_{\phi_{i_j}}$ executable configuration.

A solution for our MMKP is a set of selected objects $S \stackrel{def}{=} \{o_1, \ldots o_n\}$ representing, obliviously, a choreography configuration.

5.3 The Heuristic Approach Based on ACO

According to our experiments, the MMKP formulation performs notably better than MKP one, producing the exact solution quicker and using less memory; this is due both to the lack of an exhaustive choreography configurations enumeration, which

requires a lot of memory and time, and to a smaller number of columns, despite the MMKP formulation has more constraints compared to the MKP one.

However, since the MMKP is an NP-hard problem, we expect that is not always possible to find a feasible solution in a reasonable computing time especially for big instances, therefore we look for a heuristic approach for solving it quicker providing good quality solutions.

We have decided to built a *custom* algorithm based on *Ant Colony Optimization* (ACO), a class of stochastic meta-heuristics that have been applied to solve many combinatorial optimization problems such as traveling salesman problems, quadratic assignment problems, or vehicle routing problems.

Aforementioned class of meta-heuristics imitate the behavior shown by real ants when searching for food, where many simple interactions between single ants result in a very complex behavior of the whole ant colony Maniezzo et al. [32][40].

Any ACO algorithm is based on a set of computational agents, called *artificial ants*, which iteratively construct a so-called *partial solution* for the instance to solve-Maniezzo et al. [32][40]. At each iteration, each artificial ant moves from a partial solution to another, applying a series of stochastic local decisions whose policy is based on following parameters:

Attractiveness which is a value computed using a heuristic approach indicating the a *priori* desirability of a given partial solution.

Pheromone Trail which represents a value indicating how proficient it has been in the past to select a particular partial solution, representing, therefore, an a *posteriori* indication of its desirability.

In very simple terms, each ant incrementally constructs the partial solution to the problem and, after evaluating the found solution, it modifies the pheromone trail on the components used in its solution; that pheromone information will be use by future ants to find a better solutions to the problem instance. Indeed, by increasing or decreasing the level of pheromone trails, ants distinguish "*good*" from "*bad*" solutions. In other words, pheromone trails represent the way according to which ants communicate in order to find the best solution Maniezzo et al. [32][40].

5.3.1 Solution Components Graph

To solve MMKPs with ACO, we have to define how pheromone trails are laid by artificial ants, explaining how they follow them when constructing new partial solutions. In other words we need to decide which components of the constructed solutions should be rewarded, and how to exploit these rewards when constructing new solutions.

In order to correctly explain how we have decided to lay pheromone, we have to define the concept of *solution components graph*.

Formally, let the MMKP formulation reported in 5.10, any weighted directed graph $G = (O, E)$ such that:

$$|O| = \sum_{j=1}^{n} |g_j| \tag{5.17}$$

$$(o_{i_j}, o_{k_m}) \in E \Leftrightarrow i \neq k \qquad \forall i, k \in \mathbb{N} \cap \{1, n\}, \forall m, j \in \mathbb{N} \tag{5.18}$$

is called solution components graph, where:

- Each vertex o_{i_j} is adjacent only to all other vertices which belong to any group other than i; in other words, vertices belonging to the same group are not adjacent each others.

- The weight associated to each edge $(o_{i_j}, o_{k_m}) \in E$ is a positive real number representing the *pheromone trail* laid by our artificial ants and it can change during algorithm iterations.

  The value of pheromone trail associated to each edge (o_{i_j}, o_{k_m}) during the k-th iteration of the algorithm is denoted by $\tau_k(o_{i_j}, o_{k_m})$.

  Intuitively, the pheromone value associated to the edge $(o_{i_j}, o_{k_m}) \in E$ represents the desirability to select the object o_{k_m} when o_{i_j} was been previously selected as part of the solution.

  We have adopted the so-called $\mathcal{MAX} - \mathcal{MIN}$ Ant System [40], according to which following constraint must hold:

$$w_{\mathbf{min}} \leq \tau_k(o_{i_j}, o_{k_m}) \leq w_{\mathbf{max}} \tag{5.19}$$

$$\tau_0(o_{i_j}, o_{k_m}) = w_{\mathbf{max}} \qquad \forall i, k \in \mathbb{N} \cap \{1, n\}, \forall m, j \in \mathbb{N} \tag{5.20}$$

that is, we explicitly impose lower and upper bounds $w_{\mathbf{min}}$ and $w_{\mathbf{max}}$ on pheromone trails while pheromone trails are set to $w_{\mathbf{max}}$ at the beginning of the search.

5.3.2 The Pre-provisioned Colony Optimization Algorithm with Lazy Pheromone Update

ACO algorithms are conceptually very simple and they follow a specific scheme outlined in Algorithm 2. In general, after the initialization of the pheromone trails and some parameters, a main loop is repeated until a termination condition is met. The ants construct feasible solutions, which are subsequently improved by applying a local search algorithm. Finally the pheromone trails are updated.

Algorithm 2: Generic algorithmic skeleton for ACO algorithms

```
1 Initialization pheromone trails;
2 while Termination conditions not met do
3 |   ConstructSolutions;
4 |   ApplyLocalSearch;
5 |   UpdatePheromoneTrails;
6 end
```

However, we adopted a slightly different version of the ACO based algorithm described in Algorithm 2, calling it *Pre-provisioned Colony Optimization Algorithm with Lazy Pheromone Update*.

As our algorithm name suggests, we adopt a lazy approach for pheromone trails update. In fact, we believe that it is not very useful to update pheromone trails associated to the edges belonging to the solution component graph which, for some reasons, are never traversed by ants. Therefore, to update all edges can be not

efficient from a computational point of view, especially when aforementioned graph is very big.

Therefore we delay pheromone trails update until the first time it is needed by an ants, allowing our algorithm to perform faster.

Moreover our approach allows a parallel pheromone trails update too; as we will see shortly, since each ants start its journey inside the solution components graph in different randomly selected nodes, because every of which update different edges, it is possible to update pheromone trails with a very low risk to pay performance cost due to synchronization needs.

Finally, since can be quite expensive to allocate at run-time the solution components graph, increasing system response latency, we decided to exploit a pre-provisioning tactic[29] to anticipates data needs; despite this solution is not very efficient due to high memory consumption, it can assure lower latency.

A complete description of the algorithm performed by ants to build partial solutions is reported in Algorithm 3.

5.3.3 Transition Probabilities

Although firstly each ants selects randomly an object of his partial solution from a randomly chosen group, all subsequent objects are selected according to so-called "*transition probability*" [40][20][26], which depends on several parameters:

- the attractiveness of the objects.
- the path traveled so far by the ant.
- the pheromone trail laid on the solution components graph G.

Let n the number of the groups and $z \in \mathbb{N} \cap [1, n-1]$ the number of items selected by an ant.

Formally, during the k-th iteration of our algorithm, the path $\pi_{k_l}^{(z)}$, traversed by the l-th ant on the solution components graph $G(O, E)$ after the z-th object selection, can be defined as follows:

$$\pi_{k_l}^{(z)} = o_1 e_1 o_2 \dots o_{z-1} e_z o_{z+1} \tag{5.21}$$

Algorithm 3: Pseudo-code regarding partial solution generation performed by ants.

for $l \leftarrow 0$ **to** m **by** 1 **do**
  $z \leftarrow 0$;
  $\mathbf{S}_{k_l}^{(z)} \leftarrow \emptyset$;
  $\mathbf{G} \leftarrow \mathcal{G}$;
  $\mathbf{G}_i \leftarrow$ Randomly select a group from $\mathbf{G}$;
  $o_{i_j} \leftarrow$ Randomly select an object from $\mathbf{G}_i$ which does not viotale resource constraints;
  $z \leftarrow z + 1$;
  $\mathbf{S}_{k_l}^{(z)} \leftarrow \{o_{i_j}\}$;
  $\mathbf{G} \leftarrow \mathbf{G} \setminus \mathbf{G}_i$;
  while $\boldsymbol{G} \neq \emptyset$ **do**
    $\mathbf{G}_a \leftarrow$ Randomly select a group from $\mathbf{G}$;
    $\mathcal{O} \leftarrow$ Select a group of candidates objects belonging to $\mathbf{G}_a$ which do not viotale resource constraints ;
    $o_{a_b} \leftarrow$ Select an object from $\mathbf{G}_a$ having highest transition probability $P(o_{a_b}, \mathbf{S}_{k_l}^{(z)}, \pi_{k_l}^{(z)})$;
    $z \leftarrow z + 1$;
    $\mathbf{S}_{k_l}^{(z)} \leftarrow \mathbf{S}_{k_l}^{(z)} \cup o_{a_b}$;
    $\mathbf{G} \leftarrow \mathbf{G} \setminus \mathbf{G}_a$;
  end
end
$S_k \leftarrow$ Select $\mathbf{S}_{k_l}^{(z)}$ having maximum profit;
return S_k

where:

$$o_s \in O \qquad \forall s \in \mathbb{N} \cap [1, z] \tag{5.22}$$

$$e_s = (o_s, o_{s+1}) \in E \qquad \forall s \in \mathbb{N} \cap [1, z] \tag{5.23}$$

$$g(o_s) \neq g(o_r) \qquad \forall s, r \in \mathbb{N} \cap [1, z] : s \neq r \tag{5.24}$$

The last property states that each pair of selected objects belong to distinct groups. We recall that the first object of $\pi_{k_l}^{(z)}$ is always picked randomly.

Moreover, $\mathbf{S}_{k_l}^{(z)} = \{o_1, \ldots, o_{z+1}\}$ represents the partial solution built so far by the l-th ant after the z-th object selection; therefore, $|\mathbf{S}_{k_l}^{(z)}| = z + 1$.

Formally, after z object selection, the transition probability associated to an object $o_{i_j} \in O$, given $\mathbf{S}_{k_l}^{(z)}$ and $\pi_{k_l}^{(z)}$, can be computed as follows:

$$P(o_{i_j}, \mathbf{S}_{k_l}^{(z)}, \pi_{k_l}^{(z)}) \stackrel{def}{=} \frac{\left[\tau(o_{i_j}, \pi_{k_l}^{(z)})\right]^{\alpha} \cdot \left[\eta(o_{i_j}, \mathbf{S}_{k_l}^{(z)})\right]^{\beta}}{\sum_{o_{i_j} \in \mathcal{C}(\mathbf{G}_i, \mathbf{S}_{k_l}^{(z)})} \left[\tau(o_{i_j}, \pi_{k_l}^{(z)})\right]^{\alpha} \cdot \left[\eta(o_{i_j}, \mathbf{S}_{k_l}^{(z)})\right]^{\beta}} \tag{5.25}$$

where:

- $\tau(o_{i_j}, \pi_{k_l}^{(z)})$ is the pheromone factor.
- $\eta(o_{i_j}, \mathbf{S}_{k_l}^{(z)})$ is the heuristic factor.
- $\alpha, \beta \in \mathbb{R}^+$ are two parameter that determine, respectively, the relative importance of pheromone and heuristic factors.

The pheromone factor $\tau(o_{i_j}, \pi_{k_l}^{(z)})$ depends on the path $\pi_{k_l}^{(z)}$ and, in particular, on the quantity of pheromone laid on edges connecting the objects that already are in the partial solution $\mathbf{S}_{k_l}^{(z)}$.

Formally, be o_{z+1} the last vertex of the path $\pi_{k_l}^{(z)}$, the pheromone factor can be computed as follows:

$$\tau(o_{i_j}, \pi_{k_l}^{(z)}) \stackrel{def}{=} \tau(o_{z+1}, o_{i_j}) + \sum_{e \in \pi_{k_l}^{(z)}} \tau(e) \tag{5.26}$$

The heuristic factor $\eta(o_{i_j}, \mathbf{S}_{k_l}^{(z)})$ also depends on the whole set $\mathbf{S}_{k_l}^{(z)}$ of selected objects.

The following ratio:

$$h_{\mathbf{S}_{k_l}^{(z)}}(o_{i_j}) \stackrel{def}{=} \sum_{y=0}^{Y} \frac{c_{y_j} + 1}{\left(b_y - \sum\limits_{o_s \in \mathbf{S}_{k_l}^{(z)}} c_y(o_z)\right) + 1} \tag{5.27}$$

represents the tightness of the object o_{i_j} on the problem constraints relatively to the constructed solution $\mathbf{S}_{k_l}^{(z)}$; thus, the lower this ratio is, the more the object is profitable [20].

We can now define the heuristic factor formula as follows:

$$\eta(o_{i_j}, \mathbf{S}_{k_l}^{(z)}) \stackrel{def}{=} \frac{p_{i_j} + 1}{h_{\mathbf{S}_{k_l}^{(z)}}(o_{i_j}) + 1} \tag{5.28}$$

5.3.4 Local Search

According to Maniezzo et al. [32], to make ACO algorithm competitive with state-of-the-art algorithm for combinatorial optimization problems, several local search procedures must be used. These algorithms are invoked when solution construction phase is complete and $\mathbf{S}_k$, that is the best solution built by ants, is obtained. The aim of that algorithms is to improve current best solution.

To be precise, we have adopted a *random local search* algorithm, which represents an exhaustive search within a group to improve the solution. It replaces current selected object of a group with every other object that does not violate resource constraints and checks if it is a better solution. The total procedure is repeated a number of times, each time for a random group. The pseudo-code of that procedure is reported in algorithm 4.

5.3.5 Pheromone Trail Update

Let $m \in \mathbb{N}\backslash\{0\}$ the amount of ants, $l \in \mathbb{N} \cap [1, m]$, S_{best} the best solution, having maximal profit, constructed from the beginning of the algorithm and $\mathbf{S}_k = \{S_{k_1}, \ldots, S_{k_m}\}$ the set containing all partial solutions constructed by ants during k-th iteration,

Algorithm 4: Pseudo-code of the random local search algorithm.

```
for a specified number of times do
    G_i ← Randomly select a group from 𝒢;
    for each object o_{i_j} ∈ G_i other than the one in S_k do
        o_{i_s} ← The object belonging to S_k such that g(o_{i_s}) = g(o_{i_j})
        S_k^(temp) ← {S_k \ {o_{i_s}}} ∪ {o_{i_j}}
        if S_k^(temp) do not violate MMKP constraints then
            if profit(S_k^(temp)) > profit(S_k) then
                S_k ← S_k^(temp)
            end
        end
    end
end
return S_k
```

where S_{k_l} is the partial solution found by l-th ant. Finally, if the l-th ant fails to discover a partial solution, we will assume that $S_{k_l} = \emptyset$.

For any $a, b \in \mathbb{N}$ and $i, j \in \mathbb{N} \cap [1, n]$ such that $i \neq j$, during k-th iteration of the algorithm, when solutions construction phase is complete, all pheromone trails associated to any edge (o_{i_j}, o_{a_b}), belonging to the solution components graph $G_{\mathbf{S}}$, are updated using following formula:

$$\tau(o_{i_a}, o_{j_b})_k = \tau(o_{i_a}, o_{j_b})_{k-1} \cdot \rho + \Delta\tau(o_{i_a}, o_{j_b})_k \tag{5.29}$$

where:

- $\rho \in \mathbb{R} \cap [0, 1]$ represents the so called *evaporation coefficient*.

  That coefficient is involved into a mechanism called *evaporation* according to which, lowering the pheromone trails by a constant factor, it is possible to avoid unlimited accumulation of pheromone trail over components, decreasing their desirability during future algorithm iterations.

- $\Delta\tau(o_{i_a}, o_{j_b})_k$ is the amount of pheromones laid on edge (o_{i_a}, o_{j_b}) by all ants who

have traversed that edge for their solution construction. It can be computed as follows:

$$\Delta\tau(o_{i_a}, o_{j_b})_k \stackrel{def}{=} \sum_{l=1}^{m} \tau(o_{i_a}, o_{j_b})_k^{(l)} \tag{5.30}$$

where:

– $\tau(o_{i_a}, o_{j_b})_k^{(l)}$ is the amount of pheromones laid on edge (o_{i_a}, o_{j_b}) by l-th ant, which, letting π_{k_l} the path traversed by that ant, can be computed as follows:

$$\tau(o_{i_j}, o_{a_b})_{k-1}^{l} \stackrel{def}{=} \begin{cases} 0 & \textbf{if} \quad S_{k_l} = \emptyset \\ \dfrac{1}{1 + profit(S_{best}) - profit(S_{k_l})} & \textbf{if} \quad \substack{S_{k_l} \neq \emptyset \\ (o_{i_j}, o_{a_b}) \in \pi_{k_l}} \end{cases} \tag{5.31}$$

5.3.6 Termination Conditions

According to our model, let $\mathcal{C} = (\Phi, E)$ a choreography, an upper bound on the value of the optimal solution exists; since is true that $0 \leq p_{i_j}(t) \leq 1$, where $p_{i_j}(t)$ is the score associated to the executable function configuration $x_{\phi_{i_j}}$, the upper bound of the optimal solution is simply equal to $|\mathcal{F}_{\mathcal{E}}(\mathcal{C})|$.

Then, our algorithm stops when:

- an ant has found an optimal solution.
- a maximum number of iterations have been performed.
- the improvement in the objective function in an iteration is less than a threshold; by default, if the improvement of the solution is less than the 10% respect to the previous one, our algorithm will terminate.
- No solution is found during last 2 iterations.

Chapter 6

Experimental evaluations

In this chapter, we assess both our performance model and prototype implementation conducting two sets of experiments:

- During the first one we validate the respect of user specified QoS constraints by conducting some experiments on a serverless application, whose concrete functions are deployed on two different provider, executeing it under different conditions.
- In the last one we assess accuracy, performance and the overhead of our ACO based heuristic algorithm presented in Chapter 5.3 comparing it with those of exact algorithm.

During all subsequent experiments, the parameters of our ACO based heuristic algorithm are set as follows:

- α, that is the parameter that determine the relative importance of pheromone factor, is set to 2.
- β, which, conversely, represents the relative importance of heuristic factor, is set to 2.
- ρ, that is the evaporation coefficient, is set to 0.02.
- $w_{\mathbf{min}}$ and $w_{\mathbf{max}}$, that is the lower and the upper bounds on pheromone trails, are set to 0.01 and 6.00 respectively.

According to Alaya et al. [20] research, selected parameters allow our algorithm to quickly find good solutions even though it may fail to find the optimal (or best) solution. The number of ants is set to 4, which is the double of the number of physical cores in our CPU. Finally, we limited the number of iteration of our algorithm to 1000.

Before the execution of any experiment, we performed the profiling of our concrete functions in order to obtain their average end-to-end response time and cost by invoking each of them 50 times under all available memory configuration.

All experiments were done on a PC with an Intel® Core™ `i3-1115G4` with 3.00 Ghz and 15.4 GiB of RAM running Kubuntu `22.04`.

All concrete functions of the application involved in our experiments are hosted and executed either by AWS or by a local OpenWhisk deployment using the so-called "Standalone" OpenWhisk stack, that is a full-featured OpenWhisk stack running as a Java process, where serverless functions run within Docker containers. Finally, both InfluxDB and Cassandra servers run locally within Docker containers.

Unless otherwise stated, all concrete function are deployed with an allocated memory varying from a maximum and a minimum in 32 MB increments. To be more precise, the allocated memory of each AWS Lambda concrete function varies between 256 MB and 10240 MB in 32 MB increments, resulting in 313 possible choices. Conversely, the allocated memory of OpenWhisk functions varies between 256 MB and 512 MB with 32 MB increments, resulting in 9 possible choices.

Finally, we configured our local OpenWhisk instance to support a (global) concurrency limit equal to 30, therefore, only up to 30 serverless functions can be executed at the same time.

6.1 Experiments about QoS constraints respect

To validate the respect of the QoS constraints by our prototype, we design a simple image-processing serverless application whose control-flow graph is reported in Figure 6.1. This serverless application is made up of 11 executable functions every of which has 2 different concrete function implementing them and hosted on two different FaaS provider, which, in this experiment, are both based on OpenWhisk deployed locally. According to user input, these functions perform several activi-

ties like face recognition, thumbnail generation, reverse colored or grayscale image version generation and apply a sepia tone on input image.

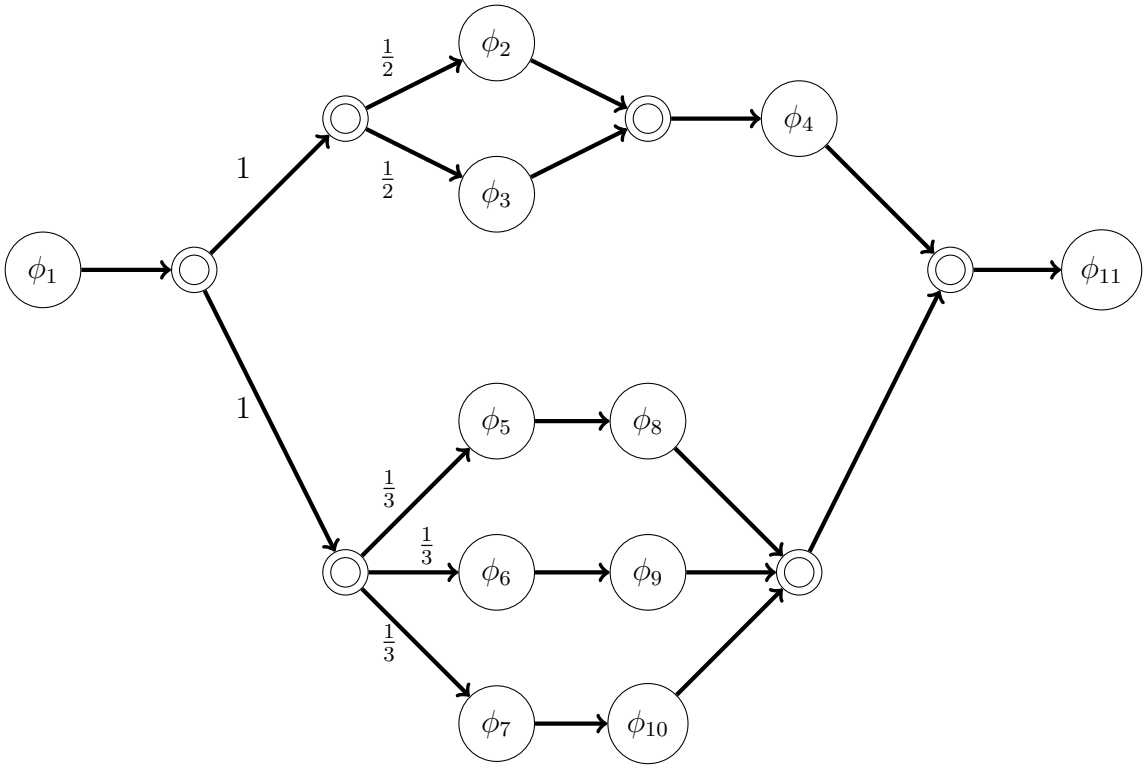

Figure 6.1: An image-processing serverless choreography used for our experiments.

Firstly, we conduct some experiments to test the respect of user specified QoS constraints. In Table 6.1 we report the result of several application invocations under different QoS constraints. Each invocation is done sequentially, one after another. In this table, paid cost and experienced response time due to the invocation of the application was been determined using the choreography configuration produced by the heuristic algorithm.

Afterwards, we conduct the same experiment as before but with several serverless application executed in parallel in a such way to run out any available function instances on one FaaS provider to force our prototype to schedule concrete function on the other.

To do that, we design and deploy another dummy serverless application consisting of a single executable function with only one concrete function hosted exclusively

$\langle (RT, w_{RT}), (C, w_C) \rangle$	Experienced Response Time	Paid Cost
$\langle (12000, 0.5), (10000, 0.5) \rangle$	3171.88	250
$\langle (12000, 1), (10000, 0) \rangle$	3061.88	667.80
$\langle (12000, 0), (10000, 1) \rangle$	5587.39	0
$\langle (8500, 0.8), (8000, 0.2) \rangle$	3061.88	250
$\langle (8500, 0.2), (8000, 0.8) \rangle$	3061.88	0
$\langle (7000, 0), (0, 1) \rangle$	3835.20	0
$\langle (7000, 1), (0, 0) \rangle$	3061.88	0
$\langle (1000, 0.5), (0, 0.5) \rangle$	*No Solution*	*No Solution*

Table 6.1: An image-processing serverless choreography used for our experiments.

on a FaaS provider, which we will call "*OpenWhisk 1*". The concrete function of the dummy application does nothing other than wait 1 second.

To saturate the FaaS provider "*OpenWhisk 1*" forcing our prototype to schedule the concrete functions of our image-processing application on the other, which we will call "*OpenWhisk 2*", we executed, in parallel, aforementioned dummy serverless applications 50 times while, at the same time, we request the invocation of our image-processing application 10 times.

After very few moments after the start of our experiment, our prototype produces a choreography configuration for the image processing application which still use some concrete functions on the "*OpenWhisk 1*" provider, since there are some function instance still available. That configuration is showed in Figure 6.2.

However, at some point, "*OpenWhisk 1*" provider runs out all available function instances because several dummy application executions are still processing and they do not free allocated function instances. Therefore, as showed in Figure 6.3, our prototype produces a choreography configuration whose concrete functions are executed only on "*OpenWhisk 2*" provider, guaranteeing the respect of QoS constraints.

Unfortunately, according to our experiment results, some invocations of the image processing application fail because our prototype cannot produce a feasible choreography configuration capable to respect QoS constraints since there are not enough function instances on any provider. For the same reasons, several dummy application invocations fail too.

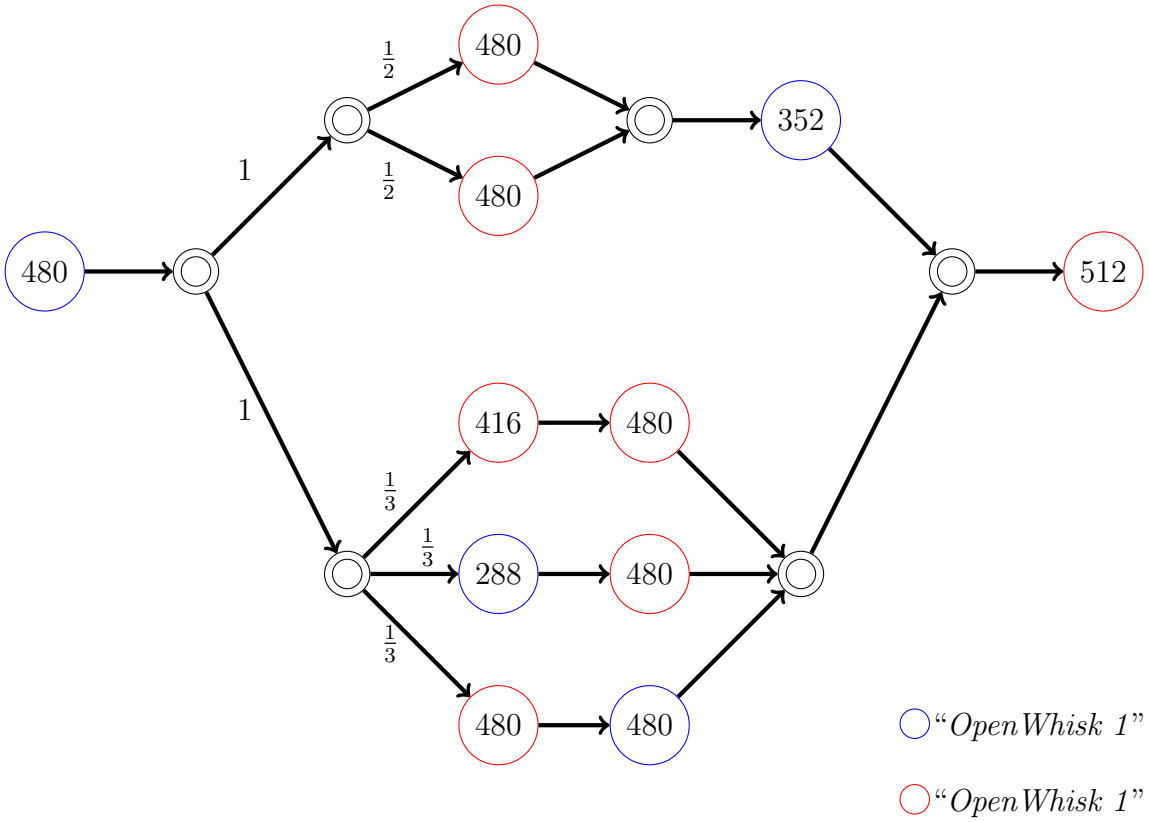

Figure 6.2: Choreography configuration produced when "*OpenWhisk 1*" provider has still some function instances to host the execution of some concrete function of our image processing application. The number inside the node represent the amount of memory selected by the our prototype.

6.2 Heuristic Algorithm Evaluation

In this section, we compare the performance and accuracy of the our custom ACO based algorithm to optimal algorithm.

To perform our evaluation, we design three set of experiments in order to analyze our heuristic algorithm performance under several point of view.

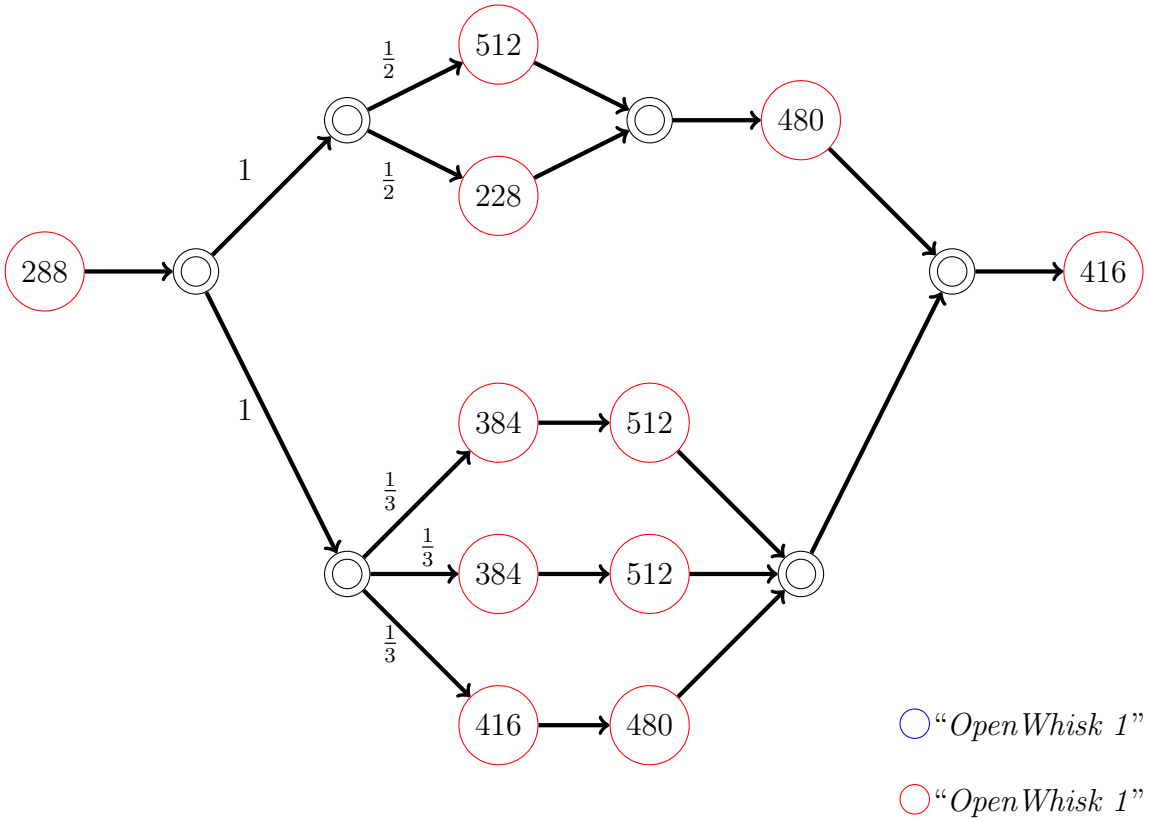

Figure 6.3: Choreography configuration produced when "*OpenWhisk 1*" provider has not enough function instances to host the execution of any concrete function of our image processing application. The number inside the node represent the amount of memory selected by the our prototype.

6.2.1 First Set of Experiments

In this first set of evaluations, we design several serverless applications having a pipeline workflow composed of various number of executable functions whose concrete functions are deployed on both AWS Lambda and Apache OpenWhisk.

For each executable function there are exactly 2 concrete functions which implement it. By design, since, as said previously, we have adopted memory increments of 32 MB, there are 322 different configuration per executable function; therefore, the total amount of serverless choreography configurations is equal to 322^x, where

x is the number of executable functions inside a serverless application.

Finally, the SLA $\langle (RT, w_{RT}), (C, w_C) \rangle$ is set to $\langle (10000, 0.5), (10000, 0.5) \rangle$ for all run. We executed each serverless applications 20 times to collect data.

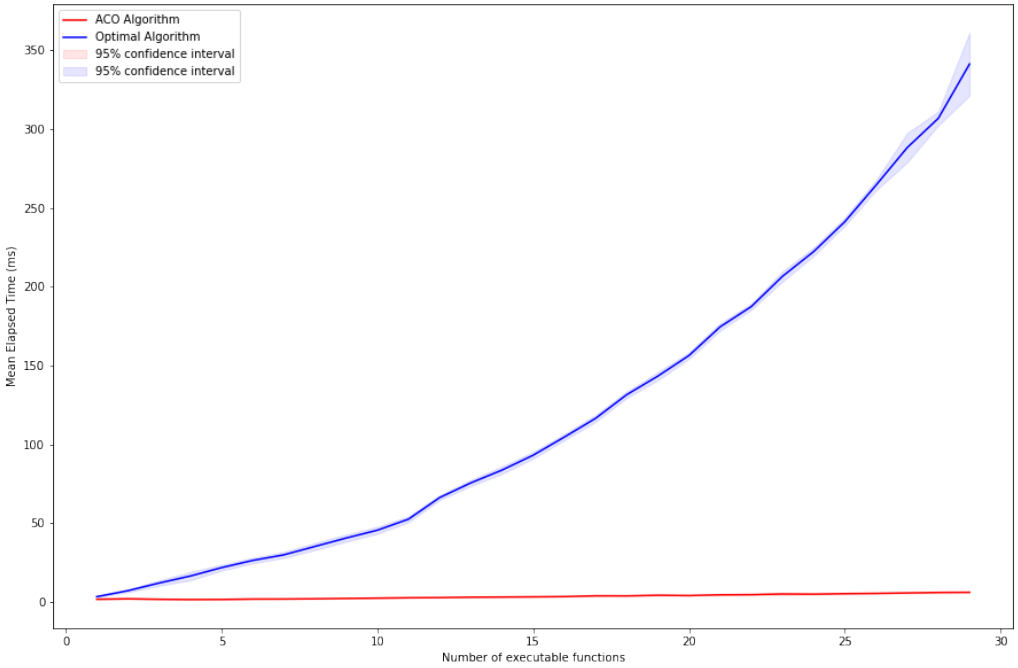

Figure 6.4: Mean elapsed time of both heuristic and optimal algorithm (1).

The Figure 6.5 shows the mean amount of time required to produce a feasible choreography configuration, that is capable to meet user specified QoS constraints, by both our custom ACO based algorithm and the optimal approach when the number of executable functions of an application increases.

Apparently, results show that the response time of the ACO based algorithm grows very slowly when the number of executable functions increases, performing quite better than the optimal approach whose performances progressively worse.

The Figure 6.5 shows the average accuracy of the solution produced by our ACO algorithm when the number of executable functions of an application increases.

In order to measure the accuracy of a given solution we used following formula:

$$accuracy \stackrel{def}{=} \frac{s_{\mathbf{opt}} - s_{\mathbf{aco}}}{s_{\mathbf{opt}}} \cdot 100 \tag{6.1}$$

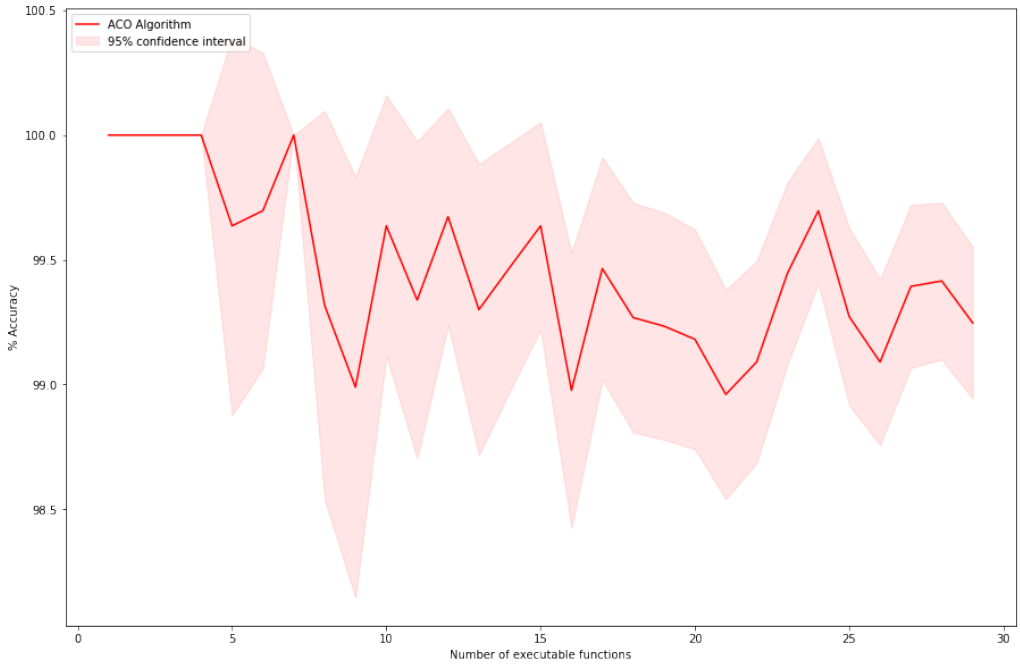

Figure 6.5: Average accuracy of the heuristic algorithm (1).

where $s_{\mathbf{opt}}$ and $s_{\mathbf{aco}}$ represent, respectively, the score of the solutions produced by the optimal and the ACO algorithm.

In this case, results show that average accuracy is above the 98.5% despite it progressively decreases when the number of executable function increases.

Figure 6.6 shows the average amount of memory required by our ACO algorithm to resolve the optimization problem described in section 5.2. Unfortunately, our solution performs very badly since greater the number of executable function, greater the amount of memory required by our algorithm.

That result suggests that this solution cannot be deployed on resource-constrained devices and limits the possibility to adapt it inside a edge based solution.

Finally, Figure 6.7 represent a summary of all above experiments, reporting the relationship between accuracy, memory consumption and response time when the number of executable functions are progressively increased.

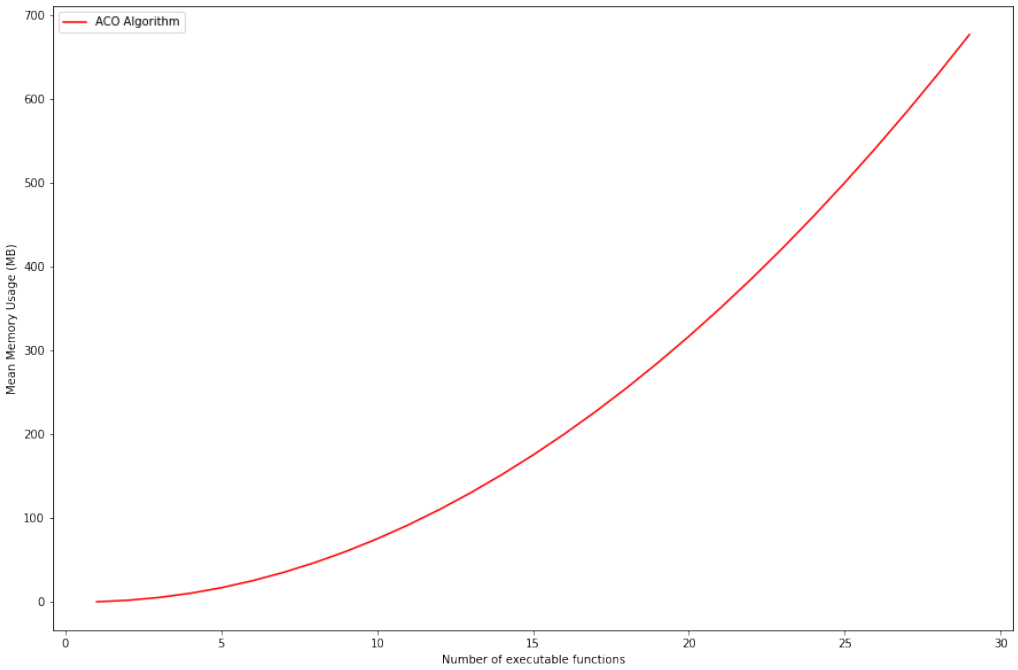

Figure 6.6: Average memory consumption of the heuristic algorithm (1).

6.2.2 Second Set of Experiments

The second set of experiment involves the use of several serverless applications having a pipeline workflow with exactly 5 executable functions whose number of concrete functions implementing them is progressively increased.

Like previously, concrete function are deployed on both AWS Lambda and Apache OpenWhisk. Memory allocation values for each executable configuration, SLA attributes values and profiling approach remain unchanged respect to the previous set of experiment.

The Figure 6.8 shows the mean amount of time required to produce a feasible choreography configuration by both the heuristic and the optimal algorithm when the number of concrete functions for each executable function increases.

Results show that the optimal algorithm performs quite worse than in the previous set of experiment, since its average response time increase faster when the number of concrete function increases than when executable functions are increased. In some run, the average response time of the optimal algorithm require more than

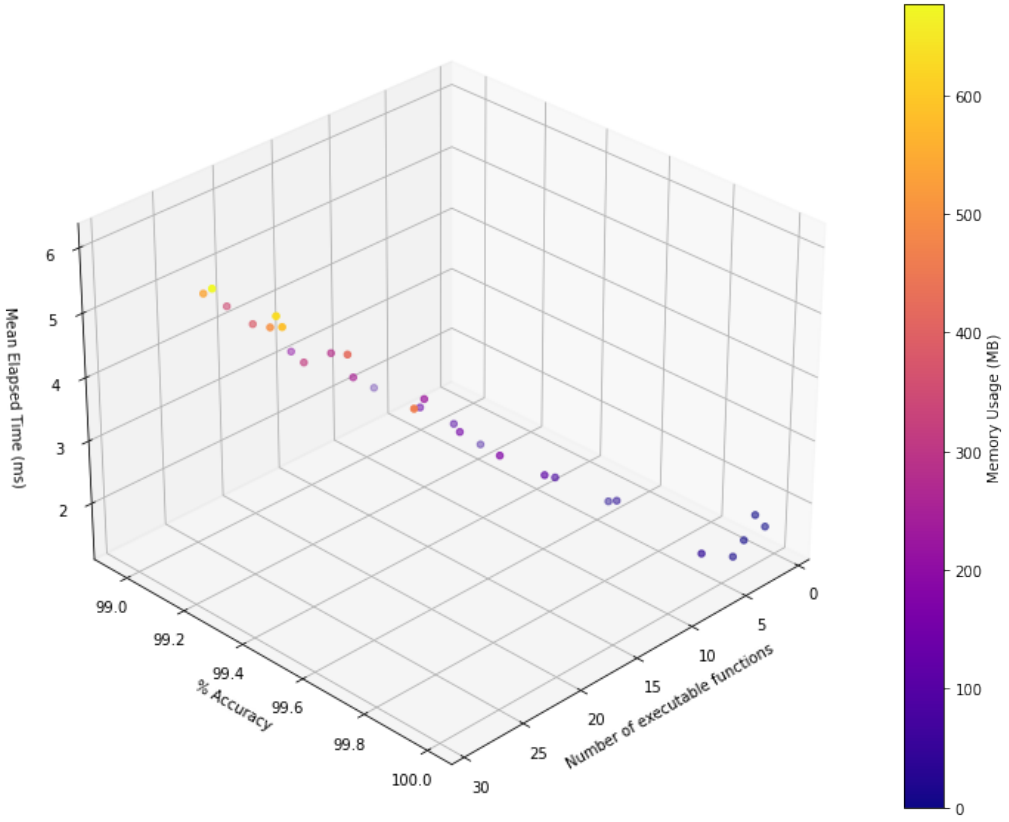

Figure 6.7: Heuristic algorithm performance.

1 second.

Conversely, the response time of the ACO based algorithm grows very slowly, performing quite better than the optimal approach.

The Figure 6.9 shows the average accuracy of the solution produced by our heuristic algorithm when the number of concrete functions for each executable function is increased. Results show that average accuracy remains above the 99.8%.

Figure 6.10 shows the average amount of memory required by our heuristic algorithm to resolve one optimization problem when the number of concrete functions for executable function is progressively increased. Unfortunately, the amount of memory required by our algorithm grows very rapidly. To be more precise, results show that the average amount of memory increase faster when the number of concrete function increases than when executable functions are increased.

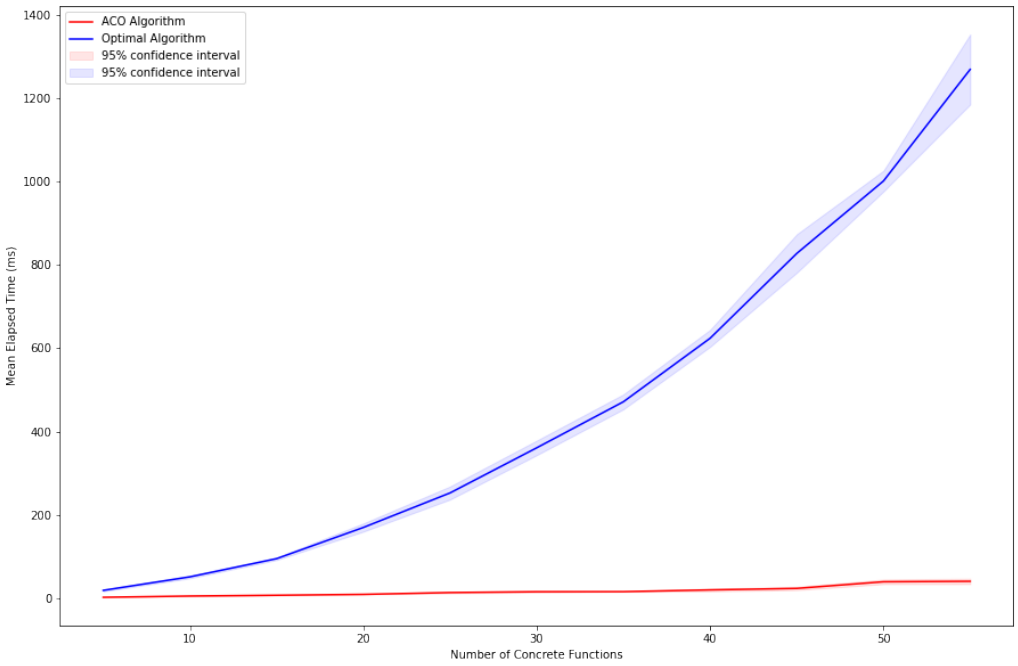

Figure 6.8: Mean elapsed time of both heuristic and optimal algorithm (2).

6.2.3 Third Set of Experiments

During the last set of experiments, we exploit the serverless application for image manipulation described previously.

We evaluate heuristic performance changing the granularity of the amount of memory allocable to each concrete function; in simple terms, we changed the number of available configurations for each executable function. For instance, since the amount of memory of an OpenWhisk functions varies between 256 MB and 512, with an increment equal to 256 MB we have only 2 possible configuration per concrete function while, with an increment of 8 MB, we have 33 different memory configuration.

The Figure 6.11 shows the average amount of time required by both the heuristic and the optimal algorithm for different memory increment values. As expected, required average time progressively decrease for both algorithms despite it seems that the heuristic approach performs worser than the optimal for smaller problem instances.

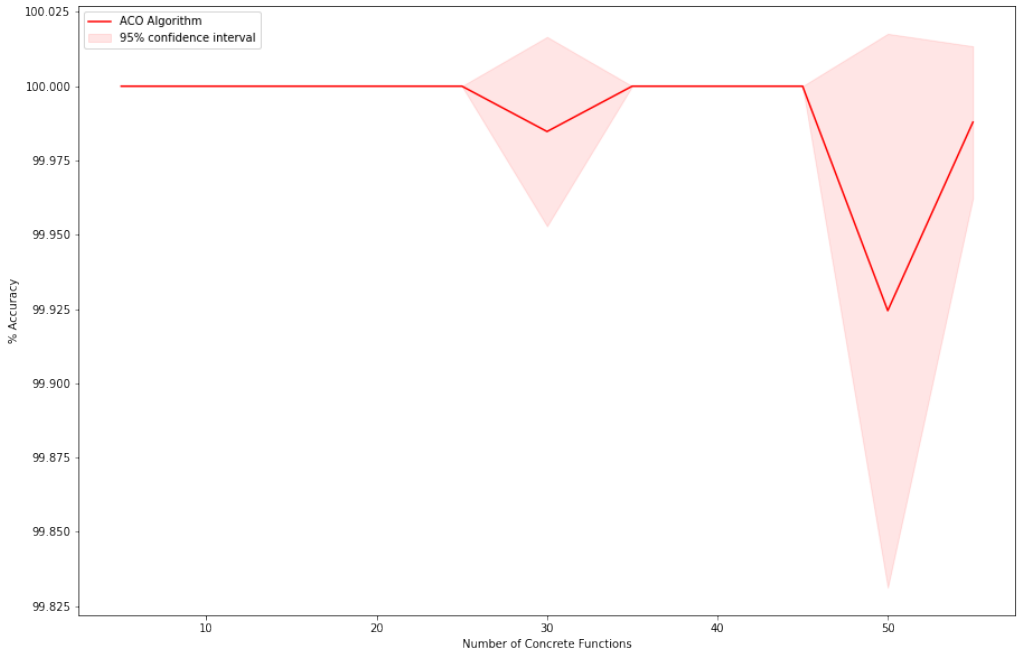

Figure 6.9: Average accuracy of the heuristic algorithm (2).

As expected, the Figure 6.12 shows that the amount of memory required by our heuristic algorithm decreases progressively when the size of the optimization problem decrease as well.

Finally, the Figure 6.13 shows the average accuracy of the solution produced by our heuristic algorithm for different memory increment values.

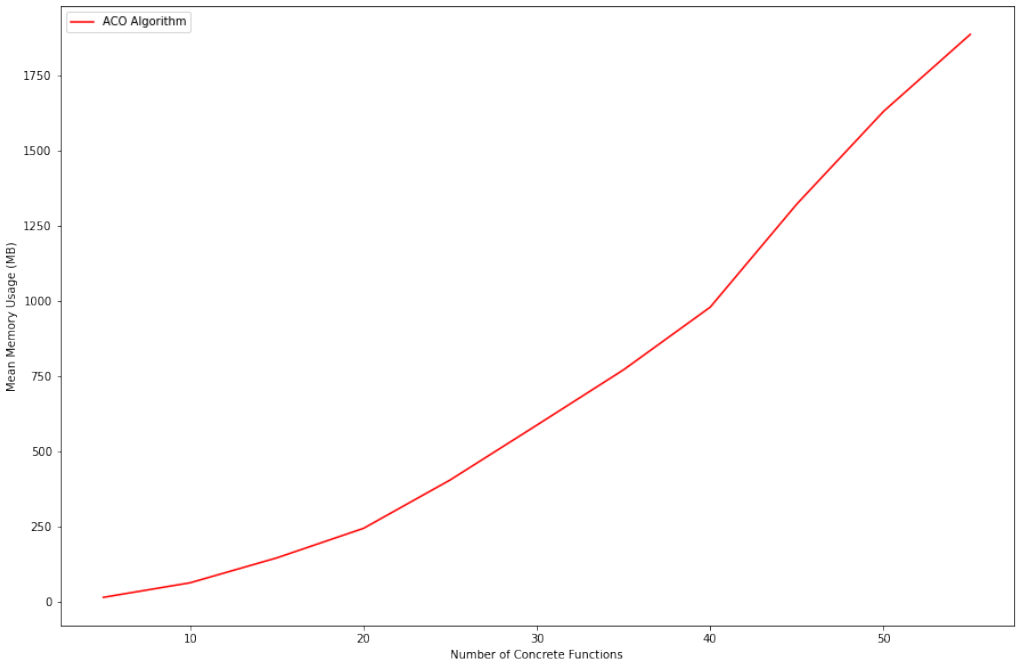

Figure 6.10: Average memory consumption of the heuristic algorithm (2).

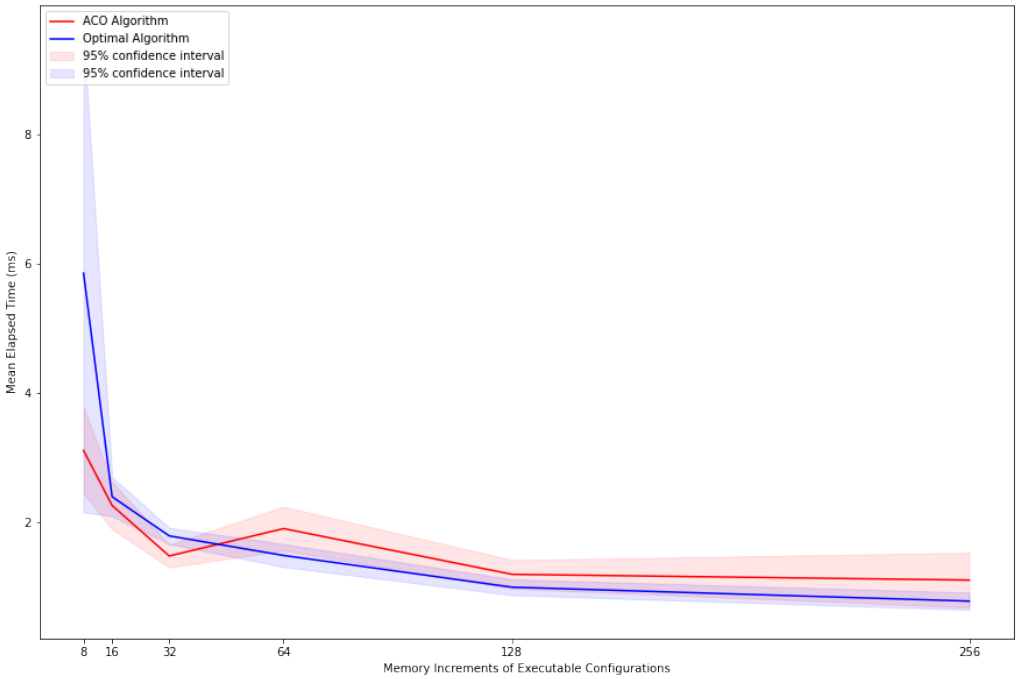

Figure 6.11: Mean elapsed time of both heuristic and optimal algorithm (3).

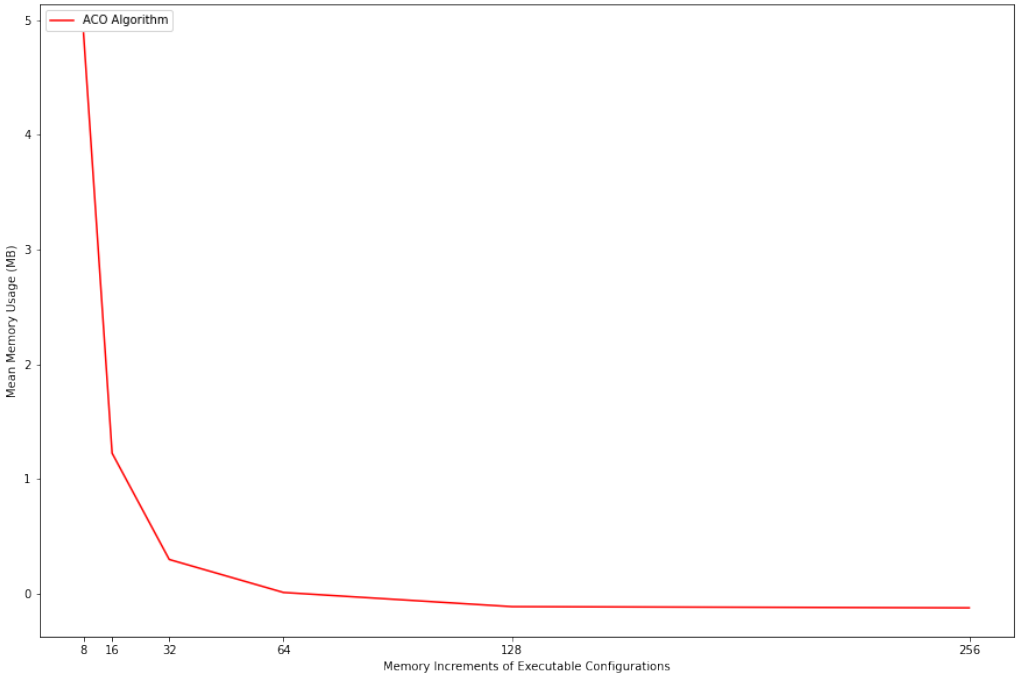

Figure 6.12: Average memory consumption of the heuristic algorithm (3).

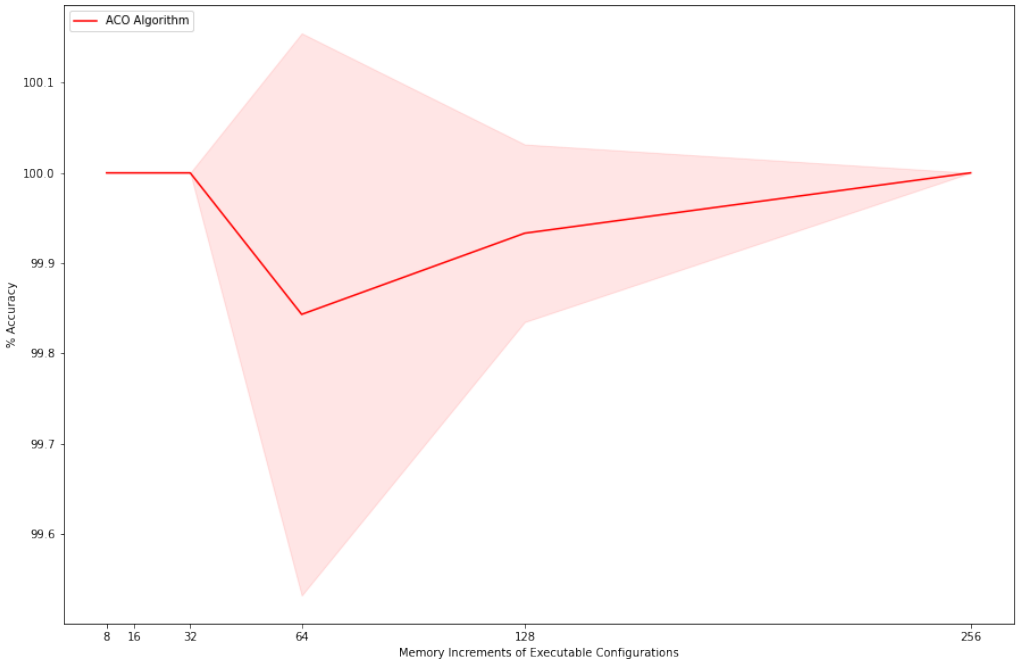

Figure 6.13: Average accuracy of the heuristic algorithm (3).

Chapter 7

Conclusion

In this work we tried to address the problem to provide QoS guarantees during serverless application execution which represents a challenge towards widespread adoption of serverless computing paradigm.

To achieve that goal, we presented an analytical model to compute average cost and execution time for generic serverless workflows having multiple concrete functions hosted on multiple FaaS providers. Then, we defined two different optimization problems formulation to address the problem of finding a suitable serverless application configuration in order to meet user defined QoS constraints; moreover, we developed a heuristic algorithm named "Pre-provisioned Colony Optimization Algorithm with Lazy Pheromone Update" to rapidly solve one of them.

Again, we verified the validity of both the proposed algorithms and the analytical model through experimental evaluations developing a prototype supporting AWS and OpenWhisk FaaS platforms.

Some hints, regarding what a possibly future work should focus on, are summarized as follows:

- Our model does not consider data transfer delay between our broker and the FaaS platform where the serverless function will be executed. Naturally, the greater the size of the input data passed to the invoked serverless function, the larger the delay; that situation can lead to a violation of QoS expectations. The optimization problems used by our model should take into account data transfer delays.

- Unfortunately, our model do not take into account the data access behavior of scheduled serverless functions too; this can lead to longer execution times and a violation of the QoS constraints.

  For example, migrating data "closer" to the FaaS platform can reduce the access latency for I/O intensive functions.

  Moreover, to further reduce network delay and latency, the model should physically co-locate function and data to assure faster data access and smaller network traffic.

  More in general, the development of a location-aware model should be considered.

- Failures may be caused by various reasons, which may affect the entire serverless application execution.

  For instance, some functions may not even start if the input JSON file exceeds the size limit imposed by the corresponding FaaS provider[1].

  Failures may be caused by network problems or authentication errors too; meanwhile arbitrary failures can lead to wrong results produced by the entire application.

  Therefore, techniques to achieve fault tolerance should be investigated and developed.

  Instead of aborting the entire application execution, the framework should invoke alternative function implementations even on other providers when any concrete function reports some error; obviously, aforementioned alternative function hasn't to violate QoS constraints.

- As said in Chapter 3, the centralized approach adopted by our framework has several problems from various points of view.

  Transparency, scalability, replication management, process decoupling and traffic congestion issues must be addressed abandoning the centralized design, solving the single point of failure problem too.

[1] For example, Amazon Lambda imposes a maximum payload size equal to 6 MB for synchronous invocation and 256 KB for asynchronous invocation [11]. At the same time, for Google Cloud Functions aforementioned limits are set to 32 MB and 512 KB respectively.[7]

- Based on the historical workload traces stored inside our TSDB, in some use cases it should be convenient to develop a protocol which, proactively, pre-warms selected function instances to further reduce latency and response time.

  In addition, the adoption of machine learning techniques, like those based on deep learning, should be considered to analyze collected time series data to make better decisions about function instance pre-warming or proactive data migration.

- Unfortunately, our model provides no information about the reliability about estimations of the average value for cost and response time exhibited by serverless functions. In other words, we said nothing about how close the estimated values (regarding average cost and response time) might be to the true values.

  A properly evaluation of adopted estimator should be done. Moreover, other estimation techniques must be investigated, evaluated and compared preferring, for example, the one which has lower mean squared error.

- To avoid delays due to scheduling policies adopted by FaaS platforms under certain conditions incurring into QoS violation, we adopt capacity requirement to check if enough function instance are available on target FaaS before to start an application execution. This choice assures us that no serverless function invocations will be enqueued on FaaS platforms incurring in unavoidable delays capable to violate user specified constraints.

  However, our choice is sub-optimal since we cannot maximize the utilization of function instances available on the FaaS platform.

  The development of a better scheduler should be considered.

Bibliography

[1] Serverless computing - aws lambda - amazon web services. https://aws.amazon.com/lambda/, 2022. [Online; accessed 17-March-2022].

[2] Aws lambda timeout. https://aws.amazon.com/about-aws/whats-new/2018/10/aws-lambda-supports-functions-that-can-run-up-to-15-minutes/, 2022. [Online; accessed 09-April-2022].

[3] Azure functions - serverless apps and computing — microsoft azure. https://azure.microsoft.com/en-us/services/functions/, 2022. [Online; accessed 17-March-2022].

[4] Azure durable functions overview. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview, 2022. [Online; accessed 24-March-2022].

[5] The go programming language. https://go.dev/, 2022. [Online; accessed 08-April-2022].

[6] Cloud functions — google cloud. https://cloud.google.com/functions/, 2022. [Online; accessed 17-March-2022].

[7] Quota limits for google cloud functions. https://cloud.google.com/functions/quotas, 2022. [Online; accessed 13-April-2022].

[8] Hypertext transfer protocol – http/1.1. https://www.ietf.org/rfc/rfc2616.txt, 2022. [Online; accessed 22-March-2022].

[9] Ibm cloud functions. https://cloud.ibm.com/functions/, 2022. [Online; accessed 17-March-2022].

[10] Introducing serverless composition for ibm cloud functions. https://www.ibm.com/cloud/blog/serverless-composition-ibm-cloud-functions, 2022. [Online; accessed 24-March-2022].

[11] Lambda quotas. https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html, 2022. [Online; accessed 13-April-2022].

[12] Apache openwhisk. https://openwhisk.apache.org/, 2022. [Online; accessed 21-March-2022].

[13] Code quality and code security — sonarqube. https://www.sonarqube.org/, 2022. [Online; accessed 08-April-2022].

[14] Aws step functions - visual workflows for modern applications. https://aws.amazon.com/step-functions/, 2022. [Online; accessed 24-March-2022].

[15] Cassandrea - open source nosql database. https://cassandra.apache.org/_/index.html, 2022. [Online; accessed 09-April-2022].

[16] Frequently asked questions about go programming language. https://go.dev/doc/faq, 2022. [Online; accessed 09-April-2022].

[17] Influxdb: Open source time series database — influxdata. https://www.influxdata.com/, 2022. [Online; accessed 29-March-2022].

[18] `lp_solve` - a mixed integer linear programming solver. https://web.mit.edu/lpsolve/doc/, 2022. [Online; accessed 09-April-2022].

[19] Nabeel Akhtar, Ali Raza, Vatche Ishakian, and Ibrahim Matta. Cose: Configuring serverless functions using statistical learning. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*, pages 129–138, July 2020. doi: 10.1109/INFOCOM41043.2020.9155363.

[20] Ines Alaya, Christine Solnon, and Khaled Ghedira. Ant algorithm for the multidimensional knapsack problem. In *International conference on Bioinspired Methods and their Applications (BIOMA 2004)*, pages 63–72, Ljubljana, Slovenia, May 2004. URL https://hal.archives-ouvertes.fr/hal-01541529.

[21] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design*. Addison-Wesley Publishing Company, USA, 5th edition, 2011. ISBN 0132143011.

[22] Ananta Kumar Das, Shikhar Yadav, and Subhasish Dhal. Detecting code smells using deep learning. In *TENCON 2019 - 2019 IEEE Region 10 Conference (TENCON)*, pages 2081–2086, 2019. doi: 10.1109/TENCON.2019.8929628.

[23] Jandisson S. de Jesus and Ana C.V. de Melo. Technical debt and the software project characteristics. a repository-based exploratory analysis. In *2017 IEEE 19th Conference on Business Informatics (CBI)*, volume 01, pages 444–453, 2017. doi: 10.1109/CBI.2017.62.

[24] Davide Falessi and Andreas Reichel. Towards an open-source tool for measuring and visualizing the interest of technical debt. In *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, pages 1–8, 2015. doi: 10.1109/MTD.2015.7332618.

[25] Dominique Guinard. A web of things application architecture integrating the real-world into the web. 2011.

[26] Shahrear Iqbal, Md. Faizul Bari, and Mohammad Rahman. A novel aco technique for fast and near optimal solutions for the multidimensional multi-choice knapsack problem. 12 2010. doi: 10.1109/ICCITECHN.2010.5723825.

[27] Anshul Jindal, Michael Gerndt, Mohak Chadha, Vladimir Podolskiy, and Pengfei Chen. Function delivery network: Extending serverless computing for heterogeneous platforms, 02 2021.

[28] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Popa, Ion Stoica, and David Patterson. Cloud programming simplified: A berkeley view on serverless computing, 02 2019.

[29] Grace Lewis and Patricia Lago. Architectural tactics for cyber-foraging: Results of a systematic literature review. *Journal of Systems and Software*, 107, 06 2015. doi: 10.1016/j.jss.2015.06.005.

[30] Changyuan Lin and Hamzeh Khazaei. Modeling and optimization of performance and cost of serverless applications. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):615–632, 2021. doi: 10.1109/TPDS.2020.3028841.

[31] Nima Mahmoudi and Hamzeh Khazaei. Performance modeling of serverless computing platforms. *IEEE Transactions on Cloud Computing*, pages 1–1, 2020. ISSN 2168-7161. doi: 10.1109/TCC.2020.3033373.

[32] Vittorio Maniezzo, Luca Maria Gambardella, and Fabio Luigi. Ant colony optimization. 05 2004. doi: 10.1007/978-3-540-39930-8_5.

[33] Garrett McGrath and Paul R. Brenner. Serverless computing: Design, implementation, and performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 405–410, 2017. doi: 10.1109/ICDCSW.2017.36.

[34] Stefano Rinaldi, Federico Bonafini, Paolo Ferrari, Alessandra Flammini, Emiliano Sisinni, and Devis Bianchini. Impact of data model on performance of time series database for internet of things applications. In *2019 IEEE International Instrumentation and Measurement Technology Conference (I2MTC)*, pages 1–6, 2019. doi: 10.1109/I2MTC.2019.8827164.

[35] S. Ristov, S. Pedratscher, and T. Fahringer. xafcl: Run scalable function choreographies across multiple faas systems. Number 01, pages 1–1, Los Alamitos, CA, USA, nov 5555. IEEE Computer Society. doi: 10.1109/TSC.2021.3128137.

[36] Sasko Ristov, Stefan Pedratscher, and Thomas Fahringer. Afcl: An abstract function choreography language for serverless workflow specification. *Future Generation Computer Systems*, 114, 08 2020. doi: 10.1016/j.future.2020.08.012.

[37] Mahadev Satyanarayanan, Guenter Klas, Marco Silva, and Simone Mangiante. The seminal role of edge-native applications. In *2019 IEEE Inter-*

national Conference on Edge Computing (EDGE), pages 33–40, 2019. doi: 10.1109/EDGE.2019.00022.

[38] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J. Yadwadkar, Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica, and David A. Patterson. What serverless computing is and should become: The next phase of cloud computing. *Commun. ACM*, 64(5):76–84, apr 2021. ISSN 0001-0782. doi: 10.1145/3406011. URL https://doi.org/10.1145/3406011.

[39] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. Architectural implications of function-as-a-service computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 1063–1075, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450369381. doi: 10.1145/3352460.3358296. URL https://doi.org/10.1145/3352460.3358296.

[40] Thomas Stützle and Holger Hoos. Max-min ant system. 16, 11 1999.

[41] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. Sequoia: enabling quality-of-service in serverless computing. pages 311–327, 10 2020. doi: 10.1145/3419111.3421306.

[42] Ján Terpák, Pavel Horovčák, and Matej Lukáč. Mathematical models creation using orchestration and choreography of web services. In *2016 17th International Carpathian Control Conference (ICCC)*, pages 739–742, 2016. doi: 10.1109/CarpathianCC.2016.7501193.

[43] Naohiro Togashi and Vitaly Klyuev. Concurrency in go and java: Performance analysis. In *2014 4th IEEE International Conference on Information Science and Technology*, pages 213–216, 2014. doi: 10.1109/ICIST.2014.6920368.

[44] M. van Steen and A.S. Tanenbaum. *Distributed Systems*. distributed-systems.ne, 3 edition, 2017.

[45] Maarten van Steen. *Graph Theory and Complex Networks*. distributed-systems.ne, 1 edition, 2010.

[46] Guoping Zeng. Two common properties of the erlang-b function, erlang-c function, and engset blocking function. *Mathematical and Computer Modelling*, 37(12):1287–1296, 2003. ISSN 0895-7177. doi: https://doi.org/10.1016/S0895-7177(03)90040-9. URL https://www.sciencedirect.com/science/article/pii/S0895717703900409.