Università degli Studi di Roma "Tor Vergata"

FACOLTÀ 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

Contents

Chapter 1

Introduction

1.1 The Serverless Computing Paradigm

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's 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 application service model according to which cloud applications are abstracted as a group of so-called *serverless functions*, hosted and orchestrated by FaaS platforms, has emerged.

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

Although a serverless function generally corresponds to an 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 fol-

lowing which the FaaS provider takes care of its execution inside a containerized environment called *function instance*, like container, virtual machine or even processes, using a user specified *serverless function configuration*, which include several parameters, like timeout, memory size or CPU power [1]. Function instances acts as tiny servers completely managed by the serverless computing platform.

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*.

Generally, we can 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 is 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 providing serverless computing services to deploy and execute serverless functions, including AWS Lambda **??**, IBM Cloud Function **??**, Microsoft Azure Functions **??** and Google Cloud Functions **??**. 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 applications.

Clearly, 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 worry about the server management, which can lead to a development's 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.

1.2 Motivations and Study's Objectives

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 a worldwide adoption, among which we recall:

- the lack of support for those applications having fine-grained state sharing needs, which is primarily due to stateless nature of serverless platforms.
- the absence of fine-grained coordination capabilities between serverless functions.
- 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.
- very unpredictable performance due to multiple factors, including function placement, *cold starts*, namely the delay incurred due to set-up time required 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.
- security concerns due to fine-grained resource sharing of serverless function which increases the exposure to *side-channel* attacks

However, according to our point of view, one of the most important obstacles concerns the *quality of service* (QoS) levels that should be guaranteed when a server-

less application is executed which, as described later, in our study, are quantitatively measured considering cost and response time.

Main difficulties relating to guaranteeing QoS levels are related to several factors, the most important of which is the lack of an adequate performance and cost model, which is crucial to assure support for several constrained applications. Several researchers have already proposed some useful performance and cost models like in **??**, **??** or **??**, but none of them is actually capable to exploit the existing of multiple implementations, that usually exhibit different performances and cost, regarding a same serverless function inside an application. Moreover, no existing solution is able to manage serverless application whose functions are hosted on multiple FaaS platform. Managing multiple function implementation on multiple FaaS platform can be quite useful to guarantee (QoS) levels.

Another very important obstacle, extensively studied by various researchers, is the optimization problem regarding serverless function configuration parameters. In fact, as we said previously, when cloud application developers want to run a serverless function, is necessary to specify several configuration's parameters regarding its execution environment, like RAM memory size, time limits or CPU power; unfortunately configuring such parameters correctly, meeting cost and response time constraints, is not trivial.

Several studies have already shown that aforementioned configuration parameters significantly affect the cost and response time of any serverless functions. **??** have shown that serverless function's response time decreases when the RAM 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 RAM memory size allocated to a serverless function, a too large value for memory can result in higher costs.

Serverless function's configuration parameters problem is quite more complicated than that because, due to the tight coupling between the pricing model and the amount of allocated resources, even setting a small value of RAM memory size can lead to higher costs, due to a longer execution time of the functions. Moreover, the marginal improvement in response time decreases as the memory increases. Despite several solutions, based on statistical learning, like in **??**, or greedy algorithms, as in **??**, already exist, none of them seem to consider the changing of execution

environment over the time. In fact, besides parameters configuration, serverless functions performance is affected by several other factors like traffic, *cold starts* and network conditions. Sometimes, in order to meet QoS levels, due to changing regarding execution environments on FaaS platform, can be useful to change, at runtime, serverless functions configuration. There is no existing solution capable to dynamically adapt itself when execution environments on FaaS platform changes.

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 which exacerbates the provider lock-in problem. There is no 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. We believe that a prospective user must be able to choose the FaaS platform that best suit his/her QoS requirements.

The main goal of our work is to mitigate

1.3 Contributions

Our work's contributions can be summarized as follows:

- Firstly, we developed a formal definition of a serverless application workflow, abstracting sequences of serverless functions, including parallel, branch and loop structures.

- We formalized a performance model in order to predict a serverless application's performance, in term of economic cost and response time, when the latter is executed with a given "*configuration*".

  Aforementioned model was been necessary to develop a methodological way to guarantee the service level agreement (SLA) of serverless applications in an economic manner.

- In order to significantly promote the use of serverless paradigm by applications that must meet strict guarantees, we present a methodology and a software framework capable to find the "*best configuration*" for a serverless application.

This can be done, providing the suitable configuration of each serverless function within a serverless application by solving an optimization problem, that is a LP problem, derived from our performance model.

- Generally, owing to he complexity of the problem, its very difficult to find an optimal configuration for a serverless applications that guarantees the respect of users performance constraints.

  This situation can discourage the adoption of serverless paradigm, mostly by users that use real-time applications.

  To provide a predictable and a very rapid system response, we developed a heuristic algorithm capable to find a near optimal solution exploiting a probabilistic technique belonging to colony optimization algorithm (ACO) family.

- In order to mitigate the impact of vendor lock-in issue, primarily related to the high migration costs paid by users when they wish change FaaS provider to better meet their needs, our software system exploits 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.

  To further reduce vendor lock-in issue, the managing of serverless application exploiting diff serverless function hosted on any supported provider using identical operations.

- Since different FaaS platform providers can exhibit different performance imposing different billing methodology, in order to find the best possible configuration of a given serverless application capable to meet users expectations, we developed both our performance model and our software system compatible with a hybrid FaaS platform execution.

  In other words, to find the best possible serverless application's configuration, our solution is able to invoke the execution of serverless function hosted on different FaaS providers, exploiting their different characteristics in term of performance or billing system.

- Since serverless application decouples its business logic into a group of serverless functions hosted on FaaS platforms

We have developed a software system acting as coordinator

To complete the business logic of the application, in- teractions among decoupled functions are indispensable. In most cases, a coordinator is required to chain together components of the application, handle events among functions, and trigger functions in the correct order defined by the business logic.

of provider migrati, owing to high from one provider to another,

In fact, hiding any difference regarding serverless compound functions representation and the way according to which the latter can be accessed, reaching an agreement on how a compound function is to be represented among different FaaS providers, we allow our users to access to any serverless function hosted on any supported provider using identical operations.

- We can significantly reduce the impact of vendor lock-in problem.

  Providing an unique way to represent a serverless compound functions, if our final users wish, they will be able to use serverless functions hosted on another FaaS provider without rewriting their serverless compound functions entirely, because very little changes to FC representation code are needed to complete the switching process.

- To improve access transparency to FaaS platform services, we have developed a middle-ware with following characteristics:

The goal of our work is to propose a

Self-Adaptation (SA) has been widely rec- ognized [17], [18], [26] as an effective approach to deal with the increasing complexity, uncertainty and dynamicity of these systems. A well recognized engineering approach to realize self-adaptation is by means of a feedback control loop called MAPE-K [26], [13] and conceived as a sequence of four computations Monitor-Analyze-Plan-Execute over a Knowledge base.

The goal of self-adaptation is to alleviate the manage- ment problem of complex software systems that operate in highly changing and evolving environments. Such systems should be able to dynamically adapt themselves to their environment with little or no human inter- vention, in order to meet both functional requirements

concerning the overall logic to be implemented and non- functional requirements concerning the quality of service (QoS) levels that should be guaranteed.

Chapter 2

Conceptual Model

From an architectural point of view, our system software framework acts as a simple client-server model, according to which processes are divided into two groups:

- fdsfdsfsd
- dfsfffsdfdsfdsfsdfdfdsfsdfdfsdfdfsdfsd

In the basic client-server model, processes in a distributed system are divided into two (possibly overlapping) groups. A server is a process implementing a specific service, for example, a file system service or a database service. A client is a process that requests a service from a server by sending it a request and subsequently waiting for the server's reply. This client-server interaction, also known as request-reply behavior A framework

and has the ability to adapt to changes in the execution time of serverless functions.

The basic goal of adaptation is to make the system able to fulfill its functional and/or non functional require- ments, despite variations in its operating environment, which are very likely to occur in the SOA domain. As pointed out in the introduction, our focus in this paper is on non functional requirements concerning the delivered QoS and cost. In the SOA domain, these requirements are usually the result of a negotiation process engaged between the service provider and user, which culmi- nates in the definition of a Service Level Agreement (SLA) concerning their respective obligations and expectations [39]. In a stochastic setting, a SLA specifies

guarantees about the average value of quality attributes, or more tough guarantees about the higher moments or percentiles of these attributes.

From an architectural point of view, our software system uses a top-down approach based on MAPE-K feedback control loop. Re- searchers wrote that:

This section details the architecture of Sequoia. Sequoia is a standalone scheduling framework that can be deployed as a proxy to existing cloud services or directly integrated into platforms such as OpenWhisk. The framework consists of three main logical entities (Figure 9): a QoS Scheduler, a Log- ging Framework, and a Policy Framework. The QoS Scheduler decides where, when, and how to run specic functions or function chains. The QoS Scheduler integrates tightly with the Logging Framework, whose responsibility is to store real-time metrics that describe the current and historical state of the serverless environment. Both the QoS Scheduler and Logging Framework interface with the Policy Frame- work to make scheduling decisions. All three components are highlighted below.

Modern software systems typically operate in dynamic environments and deal with highly changing operational con- ditions:

well recognized engineering approach to realize self-adaptation is by means of a feedback control loop called MAPE-K [26], [13] and conceived as a sequence of four computations Monitor-Analyze-Plan-Execute over a Knowledge base.

Why. The basic goal of adaptation is to make the system able to fulfill its functional and/or non functional require- ments, despite variations in its operating environment, which are very likely to occur in the SOA domain. As pointed out in the introduction, our focus in this paper is on non functional requirements concerning the delivered QoS and cost. In the SOA domain, these requirements are usually the result of a negotiation process engaged between the service provider and user, which culmi- nates in the definition of a Service Level Agreement (SLA) concerning their respective obligations and expectations [39]. In a stochastic setting, a SLA specifies guarantees about the average value of quality attributes, or more tough guarantees about the higher moments or percentiles of these attributes.

2.0.1 Communication Semantics

Since message delivery reliability issues were out of the scope our this work, we have adopt the so-called maybe comunication semantics

Maybe semantics: With maybe semantics, the remote procedure call may be executed once or not at all. Maybe semantics arises when no fault-tolerance measures are applied and can suffer from the following types of failure: • omission failures if the request or result message is lost; • crash failures when the server containing the remote operation fails. If the result message has not been received after a timeout and there are no retries, it is uncertain whether the procedure has been executed. If the request message was lost, then the procedure will not have been executed. On the other hand, the procedure may have been executed and the result message lost. A crash failure may occur either before or after the procedure is executed. Moreover, in an asynchronous system, the result of executing the procedure may arrive after the timeout. Maybe semantics is useful only for applications in which occasional failed calls are acceptable.

2.0.2 title

When building a distributed system out of existing components, we immedi- ately bump into a fundamental problem: the interfaces offered by the legacy component are most likely not suitable for all applications. In Section 1.3 we discussed how enterprise application integration could be established through middleware as a communication facilitator, but there we still implicitly as- sumed that, in the end, components could be accessed through their native interfaces.

2.1 Software system architecture

From an architectural point of view, our software system acts as a standalone framework QoS-driven runtime adaptation for serverless applications.

that implements it, for QoS- driven runtime adaptation

This section details the architecture of Sequoia. Sequoia is a standalone scheduling framework that can be deployed as a proxy to existing cloud services or directly

integrated into platforms such as OpenWhisk. The framework consists of three main logical entities (Figure 9): a QoS Scheduler, a Log- ging Framework, and a Policy Framework. The QoS Scheduler decides where, when, and how to run specic functions or function chains. The QoS Scheduler integrates tightly with the Logging Framework, whose responsibility is to store real-time metrics that describe the current and historical state of the serverless environment. Both the QoS Scheduler and Logging Framework interface with the Policy Frame- work to make scheduling decisions. All three components are highlighted below.

2.1.1 Resource owner

A *resource owner*, henceforward denoted with R, represents an entity capable of *creating*, *modifying* and *authorizing* access to several resources of our system.

Chapter 3

Model's formalization

3.1 SLA's definition

As said previously, our system's 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:

$$SLA \stackrel{def}{=} \langle (RT, w_{RT}), (C, w_C) \rangle \tag{3.1}$$

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 be hold:

  $$w_{RT} + w_C = 1 \tag{3.2}$$

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

3.2 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.

3.2.1 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:

$$\pi = \phi_1 e_1 \phi_2 e_2 \dots e_{n-2} \phi_{n-1} e_{n-1} \phi_n \tag{3.3}$$

  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:

$$out(\phi_u) \stackrel{def}{=} \{(\phi_u, \phi) \in E, \quad \forall \phi \in \Phi\} \tag{3.4}$$

$$in(\phi_u) \stackrel{def}{=} \{(\phi, \phi_u) \in E, \quad \forall \phi \in \Phi\} \tag{3.5}$$

$$succ(\phi_u) \stackrel{def}{=} \{\phi \in \Phi \mid (\phi_u, \phi) \in out(\phi_u)\} \tag{3.6}$$

$$pred(\phi_u) \stackrel{def}{=} \{\phi \in \Phi \mid (\phi, \phi_u) \in in(\phi_u)\} \tag{3.7}$$

3.2.2 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:

$$\mathcal{C}_R \stackrel{def}{=} (\Phi, E) \tag{3.8}$$

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.
- 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 on the business logic defined by R.

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

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, $P(\phi_i, \phi_j, t) = 0$ implies that ϕ_i cannot invoke ϕ_j but it does not imply that the directed edge $(\phi_i, \phi_j) \notin E$; directed edges having null weight can belong to E.
  - 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:

$$TPP(\pi, t) = \prod_{i=1}^{n-1} P(\phi_i, \phi_{i+1}, t) \tag{3.9}$$

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

$$\sum_{\pi \in \Pi(\alpha(\mathcal{C}_R), \phi)} TPP(\pi, t) = 0 \tag{3.10}$$

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

$$\sum_{\pi \in \Pi(\alpha(\mathcal{C}_R), \phi)} TPP(\pi, t) > 0 \tag{3.11}$$

- Φ must be such that following condition are hold:

$$\exists! \phi \in \Phi \quad | \quad in(\phi) = \emptyset \tag{3.12}$$

$$\exists! \phi \in \Phi \quad | \quad out(\phi) = \emptyset \tag{3.13}$$

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

$$\begin{aligned} \phi \in \Phi \quad &\text{is the } \textit{entry point} \text{ of } \mathcal{C}_{\mathcal{R}} \quad \Leftrightarrow \quad in(\phi) = \emptyset \\ \phi \in \Phi \quad &\text{is an } \textit{end point} \text{ of } \mathcal{C}_{\mathcal{R}} \quad \Leftrightarrow \quad out(\phi) = \emptyset \end{aligned} \tag{3.14}$$

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 end point.

- Following condition must be hold:

$$|\Pi(\alpha(\mathcal{C}_R), \phi)| \geq 1 \qquad \forall \phi \in \Phi \tag{3.15}$$

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

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.

3.2.2.1 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:

$$\mathcal{F}_{\mathcal{E}}(\mathcal{C}) \stackrel{def}{=} \{\phi \in \Phi \mid \phi \text{ is a serverless executable function } \} \tag{3.16}$$

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 denoting 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:

$$|succ(\phi)| = n \qquad \forall \phi \in \mathcal{F}_{\mathcal{E}}(\mathcal{C}) \wedge n \in \{0, 1\} \tag{3.17}$$

$$P(\phi, \phi_x, t) = 1 \qquad \forall \phi, \phi_x \in \mathcal{F}_{\mathcal{E}}(\mathcal{C}) : \phi_x \in succ(\phi) \wedge |succ(\phi)| = 1, \forall t \tag{3.18}$$

$$P_{exit}(\phi, t) = 0 \qquad \forall \phi \in \mathcal{F}_{\mathcal{E}}(\mathcal{C}), \forall t \tag{3.19}$$

- 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, handling events and triggering 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:

$$\mathcal{F}_{\mathcal{O}}(\mathcal{C}) \stackrel{def}{=} \{\phi \in \Phi \mid \phi \text{ is a serverless orchestration function } \} \tag{3.20}$$

Moreover, each orchestration function $\phi \in \mathcal{F}_{\mathcal{O}}(\mathcal{C})$ has a *type* which, as we will explain later, determines how choreography's performances are computed. The orchestration function's type can be determined based on certain characteristics of the graph of $\mathcal{C}$. According to our model, we distinguish four orchestration functions types exist: *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's 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 be hold:

$$|succ(\phi)| = n \qquad \forall \phi \in \mathcal{F}_{\mathcal{O}}(\mathcal{C}) : \mathcal{T}(\phi) = \tau_\omega \wedge n \in \{0, 1\} \tag{3.21}$$

$$P(\phi, \phi_x, t) = 1 \qquad \forall \phi, \phi_x \in \mathcal{F}_{\mathcal{O}}(\mathcal{C}) : \tag{3.22}$$

$$\mathcal{T}(\phi) = \tau_\omega \wedge \phi_x \in succ(\phi) \wedge |succ(\phi)| = 1, \forall t \tag{3.23}$$

$$\tag{3.24}$$

Clearly, based on above definitions, we can say:

$$\mathcal{F}_{\mathcal{E}}(\mathcal{C}) \cap \mathcal{F}_{\mathcal{O}}(\mathcal{C}) = \emptyset \tag{3.25}$$

$$\mathcal{F}_{\mathcal{E}}(\mathcal{C}) \cup \mathcal{F}_{\mathcal{O}}(\mathcal{C}) = \Phi \tag{3.26}$$

$$|\mathcal{F}_{\mathcal{E}}(\mathcal{C})| + |\mathcal{F}_{\mathcal{O}}(\mathcal{C})| = |\Phi| \tag{3.27}$$

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;

3.2.2.2 Executability condition

Unfortunately, our system software has been not designed to manage serverless concrete functions hosted on FaaS platform providers. In other words, is not actually possible to execute CRUD (*Create*, *Read*, *Update*, and *Delete*) operations regarding concrete functions on FaaS platform providers.

Therefore, let $\mathcal{C} = (\Phi, E)$ a choreography, we assume always 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:

$$\mathcal{C} \text{ is executable} \quad \Leftrightarrow \quad |\mathbf{F}_\phi| \geq 1 \qquad \forall \phi \in \mathcal{F}_{\mathcal{E}}(\mathcal{C}) \tag{3.28}$$

We will only deal with executable choreographies.

3.2.2.3 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:

$$\mathcal{C}^* \stackrel{def}{=} (\Phi^*, E^*) \qquad \text{where } \Phi^* \subseteq \Phi \wedge E^* \subseteq E \tag{3.29}$$

is called *serverless sub-choreography* of $\mathcal{C}$, or simply *sub-choreography* of $\mathcal{C}$, when the conditions 3.12, 3.13 and 3.15 are hold.

3.2.2.4 Serverless pipeline choreography

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

$$|in(\phi)| = 1 \qquad \forall \phi \in \Phi \setminus \{\alpha(\mathcal{C})\} \tag{3.30}$$

$$|out(\phi)| = 1 \qquad \forall \phi \in \Phi \setminus \{\omega(\mathcal{C})\} \tag{3.31}$$

$$P(\phi_x, \phi_y, t) = 1 \qquad \forall x, y \in \mathbb{N} \cap [1, |\Phi|], \forall t \tag{3.32}$$

Any choreography $\mathcal{C}$ satisfying 3.30, 3.31 and 3.32 will be called *serverless pipeline choreography*, or simply a *pipeline choreography*.

3.3 Serverless choreography configuration

In order to reach his 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's parameters, like RAM memory size or CPU power, has to be determined.

Formally, let a choreography $\mathcal{C} = (\Phi, E)$, when an invocation's 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 RAM 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.

3.3.1 Executable function configuration

To build a configuration for a given choreography, 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:

$$x_\phi = (f_\phi, m) \in f_\phi \times \mathbf{M}_{f_\phi} \subseteq \mathbf{F}_\phi \times \mathbb{N} \tag{3.33}$$

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.

3.3.2 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:

$$\begin{aligned}
\mathbf{x}_\mathcal{C} &\overset{def}{=} \{x_{\phi_1}, \ldots, x_{\phi_k}\} \\
&\in \left\{ \left\{ \bigcup_{j=1}^{|\mathbf{F}_{\phi_1}|} f_{\phi_{1_j}} \times \mathbf{M}_{f_{\phi_{1_j}}} \right\} \times \ldots \times \left\{ \bigcup_{j=1}^{|\mathbf{F}_{\phi_k}|} f_{\phi_{k_j}} \times \mathbf{M}_{f_{\phi_{k_j}}} \right\} \right\} \\
&= \bigtimes_{i=1}^{k} \left\{ \bigcup_{j=1}^{|\mathbf{F}_{\phi_i}|} f_{\phi_{i_j}} \times \mathbf{M}_{f_{\phi_{i_j}}} \right\} \\
&\subseteq \bigtimes_{i=1}^{k} \{\mathbf{F}_{\phi_i} \times \mathbb{N}\} = \mathbf{X}_\mathcal{C}
\end{aligned} \tag{3.34}$$

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.

3.4 Concrete function's performance evaluation

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 serverless, 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.

3.4.1 Function instances

According to serverless computing paradigm, computation is done inside isolated environments, provided by virtualization solutions such as virtual machines, containers, unikernels or even processes, called *function instances*.

As known, these instances can be considered as lightweight servers which management, provisioning and the fulfillment of any other infrastructure issues are the responsibilities of serverless computing platform provider.

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.

3.4.2 FaaS auto-scaling technique

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.

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's 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.

3.4.3 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 routeing give low priority to function instances being in idle state for long time, increasing thus the chances of resource deallocation referring aforementioned instances.

3.4.4 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 [2]

3.4.5 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.

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

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:

$$0 \leq K(t) \leq C_{\mathbf{max}} \tag{3.35}$$

- 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 [3].

- No priority is considered among incoming request.

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

3.4.6 Concrete function's swarm

To complete the formalization of our model and understand how concrete function performances are evaluated, is necessary to introduce another very important concept needed 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.

[3]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.

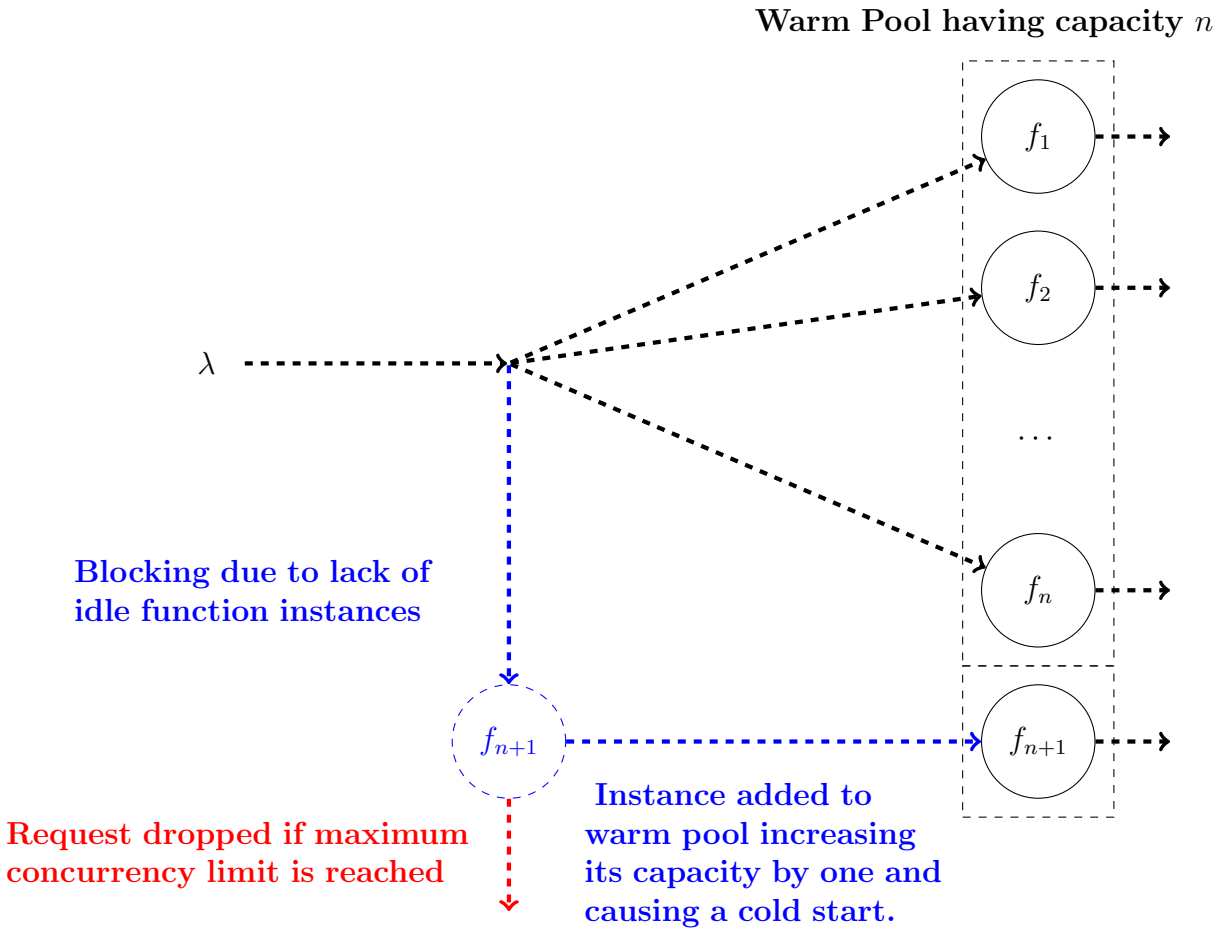

Figure 3.1: Parallel structure in the serverless workflow.

A *concrete function's 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, to each concrete function belonging to $\omega_{P_R}^{(l)}$ corresponds a dedicated $M/G/K(t)_l/K(t)_l$ queuing system, 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 serverless 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.

3.4.6.1 Cold start probability

According to our model, if maximum concurrency level is not exceed, 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 know 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's 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's 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's 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 newly 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 is maximum possible size according to FaaS policies, aforementioned request will be rejected.

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

Conversely, if a function instance has not been received any request during the last expiration threshold units of time, it will be expired 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:

$$\mathbf{P}_{\omega_{P_R}^{(l)}}(t) = \frac{\rho(t)^{K(t)}}{K(t)!} \cdot \left(\sum_{j=0}^{K(t)} \frac{\rho(t)^j}{j!} \right)^{-1} \tag{3.36}$$

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 estimate at runtime, over multiple time intervals spent observing our system.

  We have adopt the exponential moving average based approach to compute an estimation of $\lambda(t)$. Periodically, at any time t, our framework compute 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:

$$\lambda(t) \stackrel{def}{=} \begin{cases} Y_o & \text{if } t = 0 \\ \alpha \cdot Y_t + (1 - \alpha) \cdot \lambda(t-1) & \text{if } t > 0 \end{cases}. \tag{3.37}$$

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

3.4.6.2 Choreography's swarm

Let $\mathcal{C} = (\Phi, E)$ a choreography, to build its configuration $\mathbf{x}_\mathcal{C}$, we firstly 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's 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's swarm can be defined as follows:

$$\widetilde{\mathbf{S}}_\mathcal{C} \stackrel{def}{=} \left\{ \omega_P^{(l)} \in \bigcup_{i=1}^{n} \Omega_{P_i} : \exists f_\phi \in \omega_P^{(l)} \text{ such that } f_\phi \in \mathbf{F}_\phi, \forall \phi \in \mathcal{F}_\mathcal{E}(\mathcal{C}) \right\} \tag{3.38}$$

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

$$\widetilde{\mathbf{S}}_{\mathcal{C}_1} \cap \widetilde{\mathbf{S}}_{\mathcal{C}_2} \neq \emptyset \tag{3.39}$$

that is a swarm can be shared by multiple choreographies.

3.4.7 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:

$$\begin{aligned} C(x_\phi, t) &\stackrel{def}{=} C(f_\phi, m, t) \\ &= C_{avg}^{(c)}(f_\phi, m, t) \cdot \mathbf{P}_{\omega_{P_R}^{(l)}}(t) + \\ &\qquad C_{avg}^{(w)}(f_\phi, m, t) \cdot \left(1 - \mathbf{P}_{\omega_{P_R}^{(l)}}(t)\right) \end{aligned} \tag{3.40}$$

  where:

  - At any time t. when f_ϕ is executed with an allocated memory size equal to m, $C_{avg}^{(c)}(f_\phi, m, t)$ 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:

$$\begin{aligned} RT(x_\phi, t) &\stackrel{def}{=} RT(f_\phi, m, t) \\ &= RT^{(c)}_{avg}(f_\phi, m, t) \cdot \mathbf{P}_{\omega^{(l)}_{P_R}}(t) + \\ &\qquad RT^{(w)}_{avg}(f_\phi, m, t) \cdot \left(1 - \mathbf{P}_{\omega^{(l)}_{P_R}}(t)\right) \end{aligned} \tag{3.41}$$

  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.

Obliviously, our framework's duty is 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 have 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 was been used:

$$RT^{(c)}_{avg}(f_\phi, m, t) \stackrel{def}{=} \begin{cases} Y_o & \text{if } t = 0 \\ \alpha \cdot Y_t + (1 - \alpha) \cdot \lambda(t-1) & \text{if } t > 0 \end{cases}. \tag{3.42}$$

3.5 Executable function's performance evaluation

In this section, we will describe how to evaluate the performance of any executable serverless, 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 $n \in \mathbb{N}$, $\widehat{\theta_k}^{(\phi_i)} \stackrel{def}{=} \theta_1^{(\phi_i)}, \ldots, \theta_n^{(\phi_i)}$ represents the sequence of all sub-choreographies of $\mathcal{C}$ such that:

$$\phi_i \in \mathcal{F}_\mathcal{E}(\theta_1^{(\phi_i)}) \tag{3.43}$$

$$\theta_n^{(\phi_i)} = \mathcal{C} \tag{3.44}$$

$$\theta_k^{(\phi_i)} \text{ is a sub-choreography of } \theta_{k+1}^{(\phi_i)} \qquad \forall k \in [1; n-1] \tag{3.45}$$

- $E[I_{\phi_i}(t)]$ represents the expected value regarding the number's invocation of the executable function ϕ_i at time t, which can be defined as follows.

$$E[I_{\phi_i}(t)] \stackrel{def}{=} \begin{cases} 1 & if \quad n = 1 \\ \prod_{k=2}^{n} E[I_{\theta_{k-1}^{(\phi_i)}}(t)] & if \quad n \geq 2 \end{cases} \tag{3.46}$$

- $P_{exe}^{C}(\theta_k^{(\phi_i)}, t)$ 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. Obliviously, it is equal to the transition probability associated to the edge starting from $pred(\alpha(\theta_k^{(\phi_i)}))$ and ending to $\alpha(\theta_k^{(\phi_i)}))$. Formally:

$$P^C_{exe}(\theta^{(\phi_i)}_k, t) \stackrel{def}{=} \begin{cases} 1 & if \quad n = 1 \\ P\Big(pred\Big(\alpha(\theta^{(\phi_i)}_k)\Big), \alpha(\theta^{(\phi_i)}_k)\Big) & if \quad n \geq 2 \end{cases} \tag{3.47}$$

- $P^F_{exe}(\theta^{(\phi_i)}_k, \phi, t)$ is the probability to execute $\phi \in \mathcal{F}_{\mathcal{E}}(\theta_k)$ when $\alpha(\theta_k)$ is executed at time t. To compute it, firstly is necessary to convert $\theta^{(\phi_i)}_k$ into a pipeline choreography; then, let $\pi_\phi \in \Pi(\alpha(\theta_k), \phi)$ the unique path from the entry point of $\theta^{(\phi_i)}_k$ to ϕ, it can be computed using following formula:

$$P^F_{exe}(\theta^{(\phi_i)}_k, \phi, t) \stackrel{def}{=} \prod_{\phi_i \in \pi_\phi} (1 - P_{exit}(\phi_i, t)) \tag{3.48}$$

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

$$\Gamma_{\theta^{(\phi_i)}_k}(t) \stackrel{def}{=} P^C_{exe}(\theta^{(\phi_i)}_k, t) \cdot \left(\prod_{y=k+1}^{n} P^C_{exe}(\theta^{(\phi_i)}_y, t) \cdot P^F_{exe}(\theta^{(\phi_i)}_y, pred(\alpha(\theta^{(\phi_i)}_{y-1})), t) \right) \tag{3.49}$$

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:

$$c_{\phi_{i_j}}(t) \stackrel{def}{=} \begin{cases} C(x_{\phi_{i_j}}, t) \cdot P^F_{exe}(\theta^{(\phi_i)}_1, \phi_i, t) & \textbf{if} \quad m = 1 \\ C(x_{\phi_{i_j}}, t) \cdot P^F_{exe}(\theta^{(\phi_i)}_1, \phi_i, t) \cdot E[I_{\phi_i}(t)] \cdot \Gamma_{\theta^{(\phi_i)}_1}(t) & \textbf{if} \quad m \geq 2 \end{cases} \tag{3.50}$$

$$rt_{\phi_{i_j}}(t) \stackrel{def}{=} \begin{cases} RT(x_{\phi_{i_j}}, t) \cdot P^F_{exe}(\theta^{(\phi_i)}_1, \phi_i, t) & \textbf{if} \quad m = 1 \\ RT(x_{\phi_{i_j}}, t) \cdot P^F_{exe}(\theta^{(\phi_i)}_1, \phi_i, t) \cdot E[I_{\phi_i}(t)] \cdot \Gamma_{\theta^{(\phi_i)}_1}(t) & \textbf{if} \quad m \geq 2 \end{cases} \tag{3.51}$$

Let $\langle (RT, w_{RT}), (C, w_C) \rangle$ 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;

$$p_{\phi_{i_j}}(t) \stackrel{def}{=} w_{RT} \cdot \frac{t_{\phi_{i_{\mathbf{MAX}}}}(t) - rt_{\phi_{i_j}}(t)}{t_{\phi_{i_{\mathbf{MAX}}}}(t) - rt_{\phi_{i_{\mathbf{MIN}}}}(t)} + w_C \cdot \frac{c_{\phi_{i_{\mathbf{MAX}}}}(t) - c_{\phi_{i_j}}(t)}{c_{\phi_{i_{\mathbf{MAX}}}}(t) - c_{\phi_{i_{\mathbf{MIN}}}}(t)} \tag{3.52}$$

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.

3.6 Choreography's performance evaluation

In this section, we will explore some analytical methodologies to accurately compute the average end-to-end response time and charged cost of generic serverless applications, having branches, loops or even workflow's portions executed in parallel.

3.6.1 Pipeline choreography performance

Let $\mathcal{C} = (\Phi, E)$ a pipeline choreography as defined in section 3.2.2.4 and $\mathbf{x}_\mathcal{C} \in \mathbf{X}_\mathcal{C}$ a choreography configuration.

Exploiting pipeline choreography's properties, is very simple to obtain a performance evaluation when a choreography configuration is given. At any time t, following equations can be used:

$$RT(\mathcal{C}, \mathbf{x}_\mathcal{C}, t) = \sum_{\phi \in \mathcal{F}_\mathcal{E}(\mathcal{C})} RT(x_\phi, t) + \sum_{\phi \in \mathcal{F}_\mathcal{O}(\mathcal{C})} RT_{\mathcal{T}(\phi)}(\mathbf{x}_\mathcal{C}, t) \tag{3.53}$$

$$C(\mathcal{C}, \mathbf{x}_\mathcal{C}, t) = \sum_{\phi \in \mathcal{F}_\mathcal{E}(\mathcal{C})} C(x_\phi, t) + \sum_{\phi \in \mathcal{F}_\mathcal{O}(\mathcal{C})} C_{\mathcal{T}(\phi)}(\mathbf{x}_\mathcal{C}, t) \tag{3.54}$$

where:

- $RT_{\mathcal{T}(\phi)}(\mathbf{x}_{\mathcal{C}}, t)$ and $C_{\mathcal{T}(\phi)}(\mathbf{x}_{\mathcal{C}}, t)$ denote, respectively, the average response time and charged cost of the orchestration function $\phi \in \mathcal{F}_{\mathcal{O}}(\mathcal{C})$, having type $\mathcal{T}(\phi)$, when it is executed using the choreography configuration $\mathbf{x}_{\mathcal{C}}$ at time t. We will explain how to compute them very soon.

3.6.2 Serverless choreography's structures

Unfortunately, equations 3.55 and 3.56 can be used only dealing with pipeline type choreographies, therefore they are not suitable for general choreographies having branch or loops.

A very simple approach to obtain a performance evaluation of any choreography's kind, is to simplify the serverless workflow in a such way to obtain a pipeline type choreography, allow us to apply aforementioned equations. To do that, exploiting different methods for different "*structures*", the performance model trims the graph associated with a choreography by removing or modifying vertices and edges in order to convert it into a pipeline choreography. Firstly, we must explain what we mean for 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 $\alpha(\mathcal{C}^*)$ and the end point $\omega(\mathcal{C}^*)$ are, respectively, opening and closing orchestration functions having same type τ possibly coinciding. Formally:

$$\mathcal{C}^* \text{ is a structure } \Leftrightarrow \begin{cases} \alpha(\mathcal{C}^*), \omega(\mathcal{C}^*) \in \mathcal{F}_{\mathcal{O}}(\mathcal{C}^*) \\ \mathcal{T}(\alpha(\mathcal{C}^*)) = \tau_{\alpha} \\ \mathcal{T}(\omega(\mathcal{C}^*)) = \tau_{\omega} \end{cases} \tag{3.55}$$

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:

$$\begin{array}{rcl} \Phi^{**} & \stackrel{def}{=} & \Phi^* \setminus \{\alpha(\mathcal{C}^*), \omega(\mathcal{C}^*)\} \\ E^{**} & \stackrel{def}{=} & E^* \setminus \Big[out\Big(\alpha(\mathcal{C}^*)\Big) \cup in\Big(\omega(\mathcal{C}^*)\Big) \Big] \end{array} \tag{3.56}$$

that is, $\mathcal{C}^{**}$ is obtained removing both the entry point and end point 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 3.12, 3.13 and 3.15 conditions; in other words, each connected component of $\mathcal{C}^{**}$ represents a sub-choreography of $\mathcal{C}$. Formally:

$$\begin{aligned} \Theta_{\mathcal{C}^*} &\stackrel{def}{=} \{\theta_1, \ldots, \theta_c\} \\ &= \bigcup_{i=1}^{c} \left\{ \begin{array}{l} \theta_i = (\Phi_i^{**}, E_i^{**}) : \Phi_i^{**} \subset \Phi^{**} \wedge E_i^{**} \subset E^{**} \\ \Phi_i^{**} \cap \Phi_j^{**} = E_i^{**} \cap E_j^{**} = \emptyset \qquad j \in \mathbb{N} \cap [1, c] : j \neq i \\ \theta_i \text{ is a connected component of } \mathcal{C}^{**} \\ \theta_i \text{ satisfies 3.12, 3.13 and 3.15 conditions} \end{array} \right\} \end{aligned} \tag{3.57}$$

When we need to convert a generic choreography into a pipeline type during the workflow simplification process, the entire structure can be replaced by a single orchestration function ϕ_{fake}, obtaining a new choreography $\mathcal{C}' = (\Phi', E')$ such that:

$$\Phi' = \{\Phi \setminus \Phi^*\} \cup \{\phi_{fake}\} \tag{3.58}$$

$$E' = \{E \setminus E^*\} \cup \left\{ \left(prev(\alpha(\mathcal{C}^*)), \phi_{fake} \right) \right\} \cup \left\{ \left(\phi_{fake}, succ(\omega(\mathcal{C}^*)) \right) \right\} \tag{3.59}$$

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

$$RT_{\phi_{fake}}(\mathbf{x}_{\mathcal{C}}, t) = RT(\mathcal{C}^*, \mathbf{x}_{\mathcal{C}}, t) \tag{3.60}$$

$$C_{\phi_{fake}}(\mathbf{x}_{\mathcal{C}}, t) = C(\mathcal{C}^*, \mathbf{x}_{\mathcal{C}}, t) \tag{3.61}$$

$$P_{exit}(\phi_{fake}, t) = P_{exit}(\mathcal{C}^*, t) \tag{3.62}$$

3.6.2.1 Parallel

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

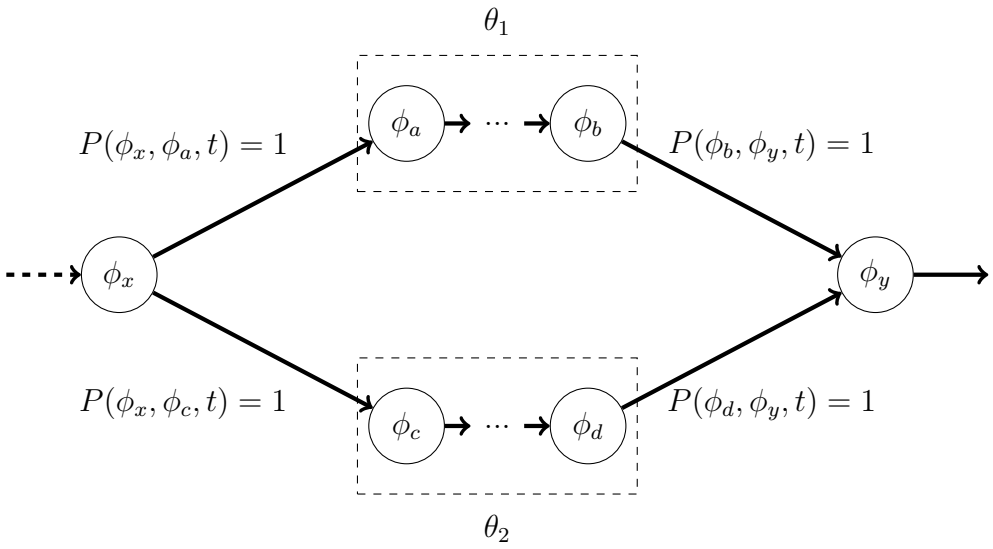

Figure 3.2: Parallel structure in a serverless workflow.

$$\mathcal{T}(\alpha(\mathcal{P})) = \mathcal{T}(\phi_x) = \tau_\alpha \tag{3.63}$$

$$\mathcal{T}(\omega(\mathcal{P})) = \mathcal{T}(\phi_y) = \tau_\omega \tag{3.64}$$

$$TPP(\pi, t) = 1 \qquad \forall \pi \in \Pi(\phi_x, \phi_y), \forall t \tag{3.65}$$

$$|\Theta| = n \qquad n \in \mathbb{N} \setminus \{0\} \tag{3.66}$$

$$E[I_\theta] = 1 \qquad \forall \theta \in \Theta \tag{3.67}$$

$$P_{exit}(\theta, t) = 0 \qquad \forall \theta \in \Theta \tag{3.68}$$

$$|succ(\alpha(\mathcal{P}))| = |prev(\omega(\mathcal{P}))| \tag{3.69}$$

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

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:

$$RT(\mathcal{P}, \mathbf{x}_\mathcal{C}, t) \stackrel{def}{=} max\left\{RT(\theta_i, \mathbf{x}_\mathcal{C}, t) \mid \theta \in \Theta\right\} \tag{3.70}$$

$$C(\mathcal{P}, \mathbf{x}_\mathcal{C}, t) \stackrel{def}{=} \sum_{i=1}^{n} C(\theta_i, \mathbf{x}_\mathcal{C}, t) \tag{3.71}$$

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:

$$P_{exit}(\mathcal{P}, t) = 0 \tag{3.72}$$

3.6.2.2 Branch

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

$$\mathcal{T}(\alpha(\mathcal{B})) = \mathcal{T}(\phi_x) = \tau_\alpha \tag{3.73}$$

$$\mathcal{T}(\omega(\mathcal{B})) = \mathcal{T}(\phi_y) = \tau_\omega \tag{3.74}$$

$$|\Theta| = n \qquad n \in \mathbb{N} \setminus \{0\} \tag{3.75}$$

$$P(\phi_x, \alpha(\theta_i), t) \leq 1 \qquad \forall i \in \mathbb{N} \cap [1, n] \tag{3.76}$$

$$\sum_{i=1}^{n} P(\phi_x, \alpha(\theta_i), t) = 1 \tag{3.77}$$

$$E[I_\theta] = 1 \qquad \forall \theta \in \Theta \tag{3.78}$$

$$|succ(\alpha(\mathcal{B}))| = |prev(\omega(\mathcal{B}))| \tag{3.79}$$

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.

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:

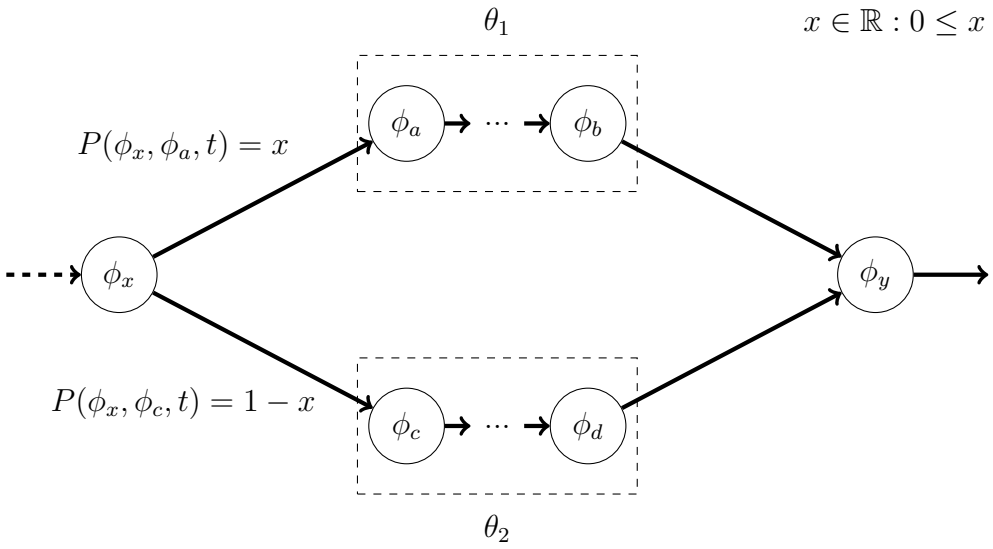

Figure 3.3: An if-else-branch structure in a serverless workflow

$$P(\phi_x, \alpha(\theta), t) = \frac{|I_\theta(t)|}{|I_\mathcal{B}(t)|} \qquad \forall \theta \in \Theta \tag{3.80}$$

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:

$$RT(\mathcal{B}, \mathbf{x}_\mathcal{C}, t) \stackrel{def}{=} \sum_{i=1}^{n} P(\phi_x, \alpha(\theta_i), t) \cdot RT(\theta_i, \mathbf{x}_\mathcal{C}, t) \tag{3.81}$$

$$C(\mathcal{B}, \mathbf{x}_\mathcal{C}, t) \stackrel{def}{=} \sum_{i=1}^{n} P(\phi_x, \alpha(\theta_i), t) \cdot C(\theta_i, \mathbf{x}_\mathcal{C}, t) \tag{3.82}$$

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

$$P_{exit}(\mathcal{B}, t) \stackrel{def}{=} \sum_{i=1}^{n} P(\phi_x, \alpha(\theta_i), t) \cdot P_{exit}(\theta_i, t) \tag{3.83}$$

3.6.2.3 Conditional loop

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

$$\mathcal{T}(\alpha(\mathcal{L})) = \mathcal{T}(\phi_x) = \tau_\alpha \tag{3.84}$$

$$\mathcal{T}(\omega(\mathcal{L})) = \mathcal{T}(\phi_y) = \tau_\omega \tag{3.85}$$

$$|\Theta| = 1 \tag{3.86}$$

$$E[I_\theta] \geq 0 \qquad \theta \in \Theta \tag{3.87}$$

$$P(\phi_x, \alpha(\theta), t) = x \tag{3.88}$$

$$P(\phi_x, \phi_y, t) = 1 - x \tag{3.89}$$

$$P(\omega(\theta), \phi_x, t) = 1 \qquad \theta \in \Theta \tag{3.90}$$

$$P(\phi_x, \alpha(\theta), t) + P(\phi_x, \phi_y, t) = 1 \qquad \theta \in \Theta \tag{3.91}$$

$$\tag{3.92}$$

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 3.82 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:

$$p = P(\phi_x, \phi_y, t) = 1 - P\Big(\phi_x, \alpha(\theta), t\Big) \tag{3.93}$$

$$\tag{3.94}$$

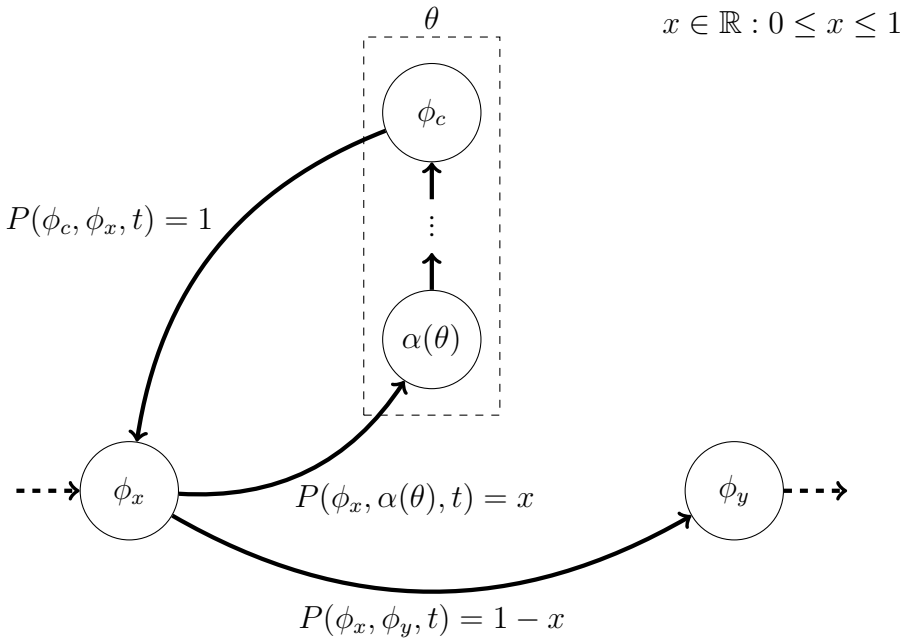

Figure 3.4: Conditional loop structure in a serverless workflow.

Then:

$$
\begin{aligned}
P(I_\theta = k) &= p \cdot (1-p)^{k-1} && (3.95)\\
&= pq^{k-1} && (3.96)\\
&= \Big[1 - P\Big(\phi_x, \alpha(\theta), t\Big)\Big] \cdot \Big[P\Big(\phi_x, \alpha(\theta), t\Big)\Big]^{k-1} && (3.97)\\
& && (3.98)
\end{aligned}
$$

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:

$$
\begin{aligned}
E[I_\theta(t)] &= \sum_{k=1}^{\infty}(k-1)pq^{k-1} \\
&= p\sum_{k=0}^{\infty}kq^k \\
&= p\cdot\frac{q}{(1-q)^2} \\
&= \frac{q}{p} \\
&= \frac{P\Big(\phi_x,\alpha(\theta),t\Big)}{1-P\Big(\phi_x,\alpha(\theta),t\Big)}
\end{aligned}
\tag{3.99}
$$

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

$$
RT(\mathcal{L},\mathbf{x}_\mathcal{C},t) \stackrel{def}{=} E[I_\theta(t)]\cdot RT(\theta_i,\mathbf{x}_\mathcal{C},t) \tag{3.100}
$$

$$
C(\mathcal{L},\mathbf{x}_\mathcal{C},t) \stackrel{def}{=} E[I_\theta(t)]\cdot C(\theta_i,\mathbf{x}_\mathcal{C},t) \tag{3.101}
$$

The exit probability of a conditional loop structure was been modeled, once again, exploiting 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:

$$
P_{exit}(\mathcal{L},t) \stackrel{def}{=} \Big(1-P_{exit}(\theta,t)\Big)^{\lfloor E[I_\theta(t)]\rfloor-1}\cdot P_{exit}(\theta,t) \tag{3.102}
$$

$$
\tag{3.103}
$$

3.6.2.4 Exit

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

$$\mathcal{T}(\alpha(\mathcal{L})) = \mathcal{T}(\phi_x) = \tau_\alpha \tag{3.104}$$

$$\mathcal{T}(\omega(\mathcal{L})) = \mathcal{T}(\phi_y) = \tau_\omega \tag{3.105}$$

$$|\Theta| = 0 \tag{3.106}$$

$$P(\phi_x, \phi_y, t) = 0 \qquad \forall t \tag{3.107}$$

$$\tag{3.108}$$

is called *exit structure* and τ denotes a *exit* type. That structure is used to model a `return` or `exit` statement inside a serverless application.

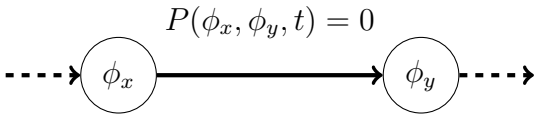

Figure 3.5: An exit structure in a serverless workflow.

At any time t, following is true that:

$$RT(\mathcal{E}, \mathbf{x}_\mathcal{C}, t) = 0 \tag{3.109}$$

$$C(\mathcal{E}, \mathbf{x}_\mathcal{C}, t) = 0 \tag{3.110}$$

$$P_{exit}(\mathcal{E}, t) = 1 \tag{3.111}$$

3.6.3 Generic choreography performance

3.7 Optimization Problem Formulations

To achieve our goal consisting in to find 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 very different manners, where:

- the first one is based on the Multidimensional Knapsack Problem formulation.

```
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 |  exitList ← find_exits(C*);
16 |  for E in exitList do
17 |  |  C* ← process(E);
18 |  end
19 end
20 return C*
```

- the second one is based on the multi-dimensional multi-choice knapsack problem formulation.

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

3.7.1 Multidimensional Knapsack Problem Formulation

Let $\mathcal{C} = (\Phi, E)$ a serverless choreography, $\mathbf{X} \in \Omega$ a choreography configuration and $\langle (RT, w_{RT}), (C, w_C) \rangle$ the SLA imposed by an user regarding the execution of $\mathcal{C}$

At any time t, the *score* $S(\mathbf{X}, t)$, or *profit*, of the choreography configuration $\mathbf{X}$ can be computed as follows:

$$S(\mathbf{X}, t) \stackrel{def}{=} w_{RT} \cdot \frac{RT_{C_{\mathbf{max}}}(t) - RT_C(\mathbf{X}, t)}{RT_{C_{\mathbf{max}}}(t) - RT_{C_{\mathbf{min}}}(t)} + w_C \cdot \frac{C_{C_{\mathbf{max}}}(t) - C_C(\mathbf{X}, t)}{C_{C_{\mathbf{max}}}(t) - C_{C_{\mathbf{min}}}(t)} \tag{3.112}$$

where:

- $RT_{C_{\mathbf{max}}}(t)$ ($RT_{C_{\mathbf{min}}}(t)$) and $C_{C_{\mathbf{max}}}(t)$ ($C_{C_{\mathbf{min}}}(t)$) denote, respectively, the maximum (minimum) value for the overall expected response time and cost observed at time t.

A very naive formulation to our problem can be expressed in term of the the Multidimensional Knapsack Problem (MKP), a well-studied, strongly NP-hard combinatorial optimization problem occurring in many different applications, whose goal is to choose a subset of items with maximum total profit. Selected items must, however, not exceed resource capacities, which are called as knapsack constraints.

The MKP formulation for our problem can be defined by the following ILP:

$$max \quad \sum_{\omega=1}^{|\Omega|} x_\omega F(\mathbf{X}_\omega) \tag{3.113}$$

$$\text{subject to} \quad \sum_{\omega=1}^{|\Omega|} x_\omega C(\mathbf{X}_\omega) \leq C_{user} \tag{3.114}$$

$$\sum_{\omega=1}^{|\Omega|} x_\omega RT(\mathbf{X}_\omega) \leq RT_{user} \tag{3.115}$$

$$\sum_{\omega=1}^{|\Omega|} x_\omega = 1 \tag{3.116}$$

$$x_\omega \in \{0,1\} \qquad \forall \omega \in \mathbb{N} \cap [1, |\Omega|] \tag{3.117}$$

where:

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 configuration (or columns above LP) is very large because it grows exponentially. Therefore, become impractical to generate and enumerate all possible choreography configuration.

For example, 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 exist 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.

3.7.2 Multi-dimensional Multi-choice Knapsack Problem Formulation

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

Let $\mathcal{C} = (\Phi, E)$ a choreography and $\mathcal{F}_{\mathcal{E}}(\mathcal{C})$ the set containing all executable functions of $\mathcal{C}$. Moreover, suppose that $|\mathcal{F}_{\mathcal{E}}(\mathcal{C})| = n \in \mathbb{N} \setminus \{0\}$.

In this context, we have n *groups* of *items* where, for $i \in \mathbb{N} \cap [1, n]$, the i-th group represents the set of all possible executable configurations for the executable function $\phi_i \in \mathcal{F}_{\mathcal{E}}(\mathcal{C})$, which belong to the set $\{\mathbf{F}_{\phi_i} \times \mathbb{N}\}$, where $\mathbf{F}_{\phi_i}$, we recall, is the implementation-set for ϕ_i. Therefore, each i-th group contains $|\{\mathbf{F}_{\phi_i} \times \mathbb{N}\}|$ items, that is executable function configurations.

According to MMKP, our objective is to pick exactly one item from each group, that is exactly one configuration 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 by SLAs attributes values imposed by users.

In this context, any $\xi \in \Xi$ represents a set of executable functions belonging to an unique combinations of parallel paths.

From now, we will use $\widetilde{\mathcal{P}}$ to denote the set containing all parallel structures of $\mathcal{C}$, while $\widetilde{\mathcal{P}}_i$ represents the i-th parallel structure belonging to $\widetilde{\mathcal{P}}$ for $i \in \mathbb{N} \cap [1; p]$, where $p = |\widetilde{\mathcal{P}}|$. Clearly, the notation $\mathcal{F}_{\mathcal{E}}(\widetilde{\mathcal{P}})$ represents the set of all executable functions contained inside any parallel structure of a given choreography.

We will use $\Delta_{\widetilde{\mathcal{P}}_{i_j}}$ notation to represent the set containing all executable functions $\phi \in \mathcal{F}_{\mathcal{E}}(\theta_{i_j})$, where θ_{i_j} is the j-th sub-choreography of the parallel structure $\widetilde{\mathcal{P}}_i$, which can be formally defined as follows:

$$\Delta_{\widetilde{\mathcal{P}}_{i_j}} \stackrel{def}{=} \left\{ \phi \in \mathcal{F}_{\mathcal{E}}(\mathcal{C}) \cap \mathcal{F}_{\mathcal{E}}(\theta_{i_j}) : \ \theta_{i_j} \text{ is the } j\text{-th sub-choreography of } \widetilde{\mathcal{P}}_i \right\} \quad (3.118)$$

In this context, $\Delta_{\widetilde{\mathcal{P}}_{i_j}}$ can be viewed as one of the parallel paths of $\widetilde{\mathcal{P}}_i$. Since the actual response time of any sub-choreography of $\widetilde{\mathcal{P}}_i$ depend on to the slower

sub-choreography of $\widetilde{\mathcal{P}}_i$, can be not trivial to compute the profit and resource requirements for every function inside a parallel structure.

Therefore, the main idea, regarding profit and resource requirements computation, is to verify if, for every parallel path for every parallel structure of the choreography, resource requirements are respected.

We will use Ξ to denote the set of all possible parallel paths combination inside a choreography. Formally it can be defined as follows:

$$\begin{aligned} \Xi &\overset{def}{=} \left\{ \Delta_{\widetilde{\mathcal{P}}_1}, \ldots, \Delta_{\widetilde{\mathcal{P}}_p} \right\} \\ &\in \left\{ \left\{ \bigcup_{j=1}^{|\widetilde{\mathcal{P}}_i|} \Delta_{\widetilde{\mathcal{P}}_{1_j}} \right\} \times \ldots \times \left\{ \bigcup_{j=1}^{|\widetilde{\mathcal{P}}_p|} \Delta_{\widetilde{\mathcal{P}}_{p_j}} \right\} \right\} \\ &= \bigtimes_{i=1}^{p} \left\{ \bigcup_{j=1}^{|\widetilde{\mathcal{P}}_i|} \Delta_{\widetilde{\mathcal{P}}_{i_j}} \right\} \end{aligned} \tag{3.119}$$

3.7.2.1 Formulation

Before to formulate our MMKP instance, we have to explain how we can compute profit and resource requirements, expressed in term of response time and cost, for each executable function configuration belonging to each group, which is somewhat complicated.

Finally, wa are able to formulate our MMKP instance problem, which can be expressed as follows:

$$max \quad \sum_{i=1}^{|\mathcal{F}_{\mathcal{E}}|} \sum_{j=1}^{|\mathbf{F}_{\phi_i} \times \mathbb{N}|} y_{\phi_{i_j}} p_{\phi_{i_j}} \tag{3.120}$$

$$\text{subject to} \quad \sum_{i=1}^{|\mathcal{F}_{\mathcal{E}}|} \sum_{j=1}^{|\mathbf{F}_{\phi_i} \times \mathbb{N}|} y_{\phi_{i_j}} c_{\phi_{i_j}} \leq C_{user} \tag{3.121}$$

$$\sum_{\phi_i \in \mathcal{F}_{\mathcal{E}} \cap \mathcal{F}_{\mathcal{E}}(\widetilde{\mathcal{P}})} \sum_{j=1}^{|\mathbf{F}_{\phi_i} \times \mathbb{N}|} y_{\phi_{i_j}} t_{\phi_{i_j}} + + \sum_{\phi_h \in \xi} \sum_{j=1}^{|\mathbf{F}_{\phi_h} \times \mathbb{N}|} y_{\phi_{h_j}} t_{\phi_{h_j}} \leq RT_{user} \qquad \forall \xi \in \Xi \tag{3.122}$$

$$\sum_{\phi_i \in \mathcal{F}_{\mathcal{E}} \cap \omega_{P_s}^{(l)}} \sum_{j=1}^{|\mathbf{F}_{\phi_i} \times \mathbb{N}|} y_{\phi_{i_j}} \leq l - r_s(t) \qquad \forall s \in \mathbb{N} \cap [1, S] \tag{3.123}$$

$$\sum_{j=1}^{|\mathbf{F}_{\phi_i} \times \mathbb{N}|} y_{\phi_{i_j}} = 1 \qquad \forall i \in \mathbb{N} \cap [1, |\mathcal{F}_{\mathcal{E}}|] \tag{3.124}$$

$$y_{\phi_{i_j}} \in \{0, 1\} \qquad \begin{matrix} \forall i \in \mathbb{N} \cap [1, |\mathcal{F}_{\mathcal{E}}|] \\ \forall j \in \mathbb{N} \cap [1, |\mathbf{F}_{\phi_i} \times \mathbb{N}|] \end{matrix} \tag{3.125}$$

To simplify our notations, from now, we will use o_{i_j} notation to denote the i-th item belonging to the j-th group, the latter denoted with g_i; clearly, in our context, the item o_{i_j} coincides with the j-th executable function configuration $x_{\phi_{i_j}}$ for the executable function $\phi_i \in \mathcal{F}_{\mathcal{E}}(\mathcal{C})$, for some $i \in \mathbb{N} \cap [1, |\mathcal{F}_{\mathcal{E}}(\mathcal{C})|]$ and $j \in \mathbb{N} \cap [1, |\mathbf{F}_{\phi_i} \times \mathbb{N}|]$.

A solution of a MMKP is a set of selected objects $S \stackrel{def}{=} \{o_1, \ldots o_n\}$ where, clearly, an object o_{i_j} is selected if the corresponding decision variable $y_{\phi_{i_j}}$ has been set to 1.

3.7.2.2 The heuristic approach based on ACO

Being MMKP an NP-hard problem, it is not always possible to find a feasible solution in a reasonable computing time especially for big instances. Although the 3.120

therefore an heuristic approach is needed for solving it.

We have decided to built an 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.

Any ACO algorithm is based on a set of computational agent, called *artificial ants*, which iteratively constructs a so-called *partial solution* for the instance to solve. 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 an 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 find 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. In fact, 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 each ants communicate in order to find the best solution.

3.7.2.3 Definition

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.

We define the *solution's components graph* $\mathcal{G} = (\mathbf{O}, \mathbf{E})$, on which ants lay pheromone trails, as a directed weighted graph such that:

$$\mathbf{O} \stackrel{def}{=} \bigcup_{j=1}^{n} \bigcup_{j=1}^{|g_i|} \{o_{i_j}\} \tag{3.126}$$

$$(o_{i_j}, o_{k_m}) \in E \Leftrightarrow i \neq k \qquad \forall i, k \in \mathbb{N} \cap \{1, n\}, \forall m, j \in \mathbb{N} \tag{3.127}$$

where:

- The set of vertices $\mathbf{O}$ is made up all items belonging to each group.

  Moreover, any object o_{i_j} is adjacent to all other objects belonging to any group except those which belong to its same group.

- The weight $w \in \mathbb{R}^+$ associated to each edge $(o_{i_j}, o_{k_m}) \in \mathbf{E}$ is the value of the *pheromone trail* laid by our artificial ants and it can change during algorithm execution.

  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 as follows:

  $$\tau_k(o_{i_j}, o_{k_m}) = w_k \tag{3.128}$$

  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, according to which following constrain must be hold:

$$w_{\mathbf{min}} \leq w_k \leq w_{\mathbf{max}} \tag{3.129}$$

$$\tau_0(o_{i_j}, o_{k_m}) = w_{\mathbf{max}} \qquad \forall i, k \in \mathbb{N} \cap \{1, n\}, \forall m, j \in \mathbb{N} \tag{3.130}$$

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

the beginning of the search. At each cycle of this algorithm, every ant constructs a solution.

It first randomly chooses an initial object, and then iteratively adds objects that are chosen within a set Candidates that contains all the objects that can be selected without violating resource constraints. Once each ant has constructed a solution, pheromone trails are updated.

Algorithm 1: Algorithmic skeleton for ACO algorithms

```
Data: this text
Result: Some solution
Initialization pheromone trails;
while Termination conditions not met do
    ConstructSolutions;
    ApplyLocalSearch;
    UpdatePheromoneTrails;
end
```

Algorithm 2: 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 $\mathbf{G}_i$;
  $z \leftarrow z + 1$;
  $\mathbf{S}_{k_l}^{(z)} \leftarrow \{o_{i_j}\}$;
  $\mathbf{G} \leftarrow \mathbf{G} \setminus \mathbf{G}_i$;
  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

3.7.2.4 Transition Probabilities

Although firstly each ants selects randomly a object of his partial solution from a randomly chosen group, all subsequent objects are selected according to so-called "*transition probability*", which depend on several parameters, like the attractiveness of the objects, the path traveled by the ant and the pheromone trail laid on the solution components graph G_S.

Let $z \in \mathbb{N} \cap [1, n-1]$ the number of items selected by an ant where n, we recall, represents the number of groups. 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 $G_{\mathbf{S}}(\mathbf{O}, E)$ after the z-th object selection, can be defined as follows:

$$\pi_{k_l}^{(z)} = o_1 e_1 o_2 \dots o_{z-1} e_z o_{z+1} \tag{3.131}$$

where:

$$o_s \in \mathbf{O} \qquad \forall s \in \mathbb{N} \cap [1, z] \tag{3.132}$$

$$e_s = (o_s, o_{s+1}) \in E \qquad \forall s \in \mathbb{N} \cap [1, z] \tag{3.133}$$

$$g(o_s) \neq g(o_r) \qquad \forall s, r \in \mathbb{N} \cap [1, z] : s \neq r \tag{3.134}$$

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

Moreover, $\mathbf{S}_{k_l}^{(z)} = \{o_1, \dots, o_{z+1}\}$ 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 $o_{i_j} \in \mathbf{O}$, given $\mathbf{S}_{k_l}^{(z)}$ and $\pi_{k_l}^{(z)}$, can be computed as follows:

$$P(o_{i_j}, \mathbf{S}_{k_l}^{(z)}, \pi_{k_l}^{(z)}) \stackrel{def}{=} \frac{\left[\tau(o_{i_j}, \pi_{k_l}^{(z)})\right]^{\alpha} \cdot \left[\eta(o_{i_j}, \mathbf{S}_{k_l}^{(z)})\right]^{\beta}}{\sum_{o_{i_j} \in \mathcal{C}(\mathbf{G}_i, \mathbf{S}_{k_l}^{(z)})} \left[\tau(o_{i_j}, \pi_{k_l}^{(z)})\right]^{\alpha} \cdot \left[\eta(o_{i_j}, \mathbf{S}_{k_l}^{(z)})\right]^{\beta}} \tag{3.135}$$

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:

$$\tau(o_{i_j}, \pi_{k_l}^{(z)}) \stackrel{def}{=} \tau(o_{z+1}, o_{i_j}) + \sum_{e \in \pi_{k_l}^{(z)}} \tau(e) \tag{3.136}$$

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:

$$h_{\mathbf{S}_{k_l}^{(z)}}(o_{i_j}) \stackrel{def}{=} \sum_{y=0}^{Y} \frac{c_{y_j}}{b_y - \sum_{o_s \in \mathbf{S}_{k_l}^{(z)}} c_y(o_z)} \tag{3.137}$$

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

We can now define the heuristic factor formula as follows:

$$\eta(o_{i_j}, \mathbf{S}_{k_l}^{(z)}) \stackrel{def}{=} \frac{p_{i_j}}{h_{\mathbf{S}_{k_l}^{(z)}}(o_{i_j})} \tag{3.138}$$

3.7.2.5 Local Search

According to **??**, 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 iteration best solution, performing a local search,

We have adopted two different local:

Random Local Search an exhaustive search within a group to improve the solution.

It replaces current selected object of a group with every other object that do 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.

Random Item Swap is an extended version of the random local search.

In this case, a specified number (> 1) of objects are swapped with other random objects from the same group without checking the resource constraints, then it checks if it is a valid solution and if it improves the solution.

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

3.7.2.6 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, 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:

$$\tau(o_{i_a}, o_{j_b})_k = \tau(o_{i_a}, o_{j_b})_{k-1} \cdot \rho + \Delta\tau(o_{i_a}, o_{j_b})_k \tag{3.139}$$

where:

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

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

$$\Delta\tau(o_{i_a}, o_{j_b})_k \overset{def}{=} \sum_{l=1}^{m} \tau(o_{i_a}, o_{j_b})_k^{(l)} \tag{3.140}$$

  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:

$$\tau(o_{i_j}, o_{a_b})_{k-1}^{l} \stackrel{def}{=} \begin{cases} 0 & \textbf{if } S_{k_l} = \emptyset \\ \dfrac{1}{1 + profit(S_{best}) - profit(S_{k_l})} & \textbf{if } S_{k_l} \neq \emptyset \wedge (o_{i_j}, o_{a_b}) \in \pi_{k_l} \end{cases} \tag{3.141}$$

3.7.2.7 Termination Conditions

Since for each item o_{i_j} is true that $0 \leq p_{i_j} \leq 1$,

The algorithm stops either when an ant has found an optimal solution (when the optimal bound is known), or when a maximum number of cycles has been performed.

Chapter 4

Computational Model

We have adopted for follwoing reasons:

1. We are able to guarantee access transparency.

   In fact, hiding any difference regarding serverless compound functions representation and the way according to which the latter can be accessed, reaching an agreement on how a compound function is to be represented among different FaaS providers, we allow our users to access to any serverless function hosted on any supported provider using identical operations.

2. We can significantly reduce the impact of vendor lock-in problem.

   Providing an unique way to represent a serverless compound functions, if our final users wish, they will be able to use serverless functions hosted on another FaaS provider without rewriting their serverless compound functions entirely, because very little changes to FC representation code are needed to complete the switching process.

   In that way, guaranteeing reduced switching costs, using our system is possible to mitigate provider lock-in.

4.1 Abstract Function Choreography Language

To overcome these weaknesses, we introduce a new Abstract Function Choreography Language (AFCL), which is a novel approach to specify FCs at a high-level of

abstraction.

From implementation point of view, our AFCL parser implementation is completely indepented from garanteeing low level coupling betweeng AFLC parser and choreography implementation.

4.2 Pr

Data collection is a major bottleneck in machine learning and an active res

f there are no existing datasets that can be used for training, then another option is to generate the datasets either manu- ally or automatically. For manual construction, crowdsourc- ing is the standard method where human workers are given tasks to gather the necessary bits of data that collectively become the generated dataset. Alternatively, automatic tech- niques can be used to generate synthetic datasets. Note that data generation can also be viewed as data augmentation if there is existing data where some missing parts needs to be filled in.

4.2.1 InfluxDB

A very important characteristic of our data-set it that it contains time series data, where the time of each instance, containing the attribute value regarding power consumption, is given by a timestamp attribute; thus our data-set represents a sequence of discrete- time data [8]. All data are listed in time order.

InfluxDB is a TSDB that stores *points*, that is single values indexed by time.

Using InfluxDB terminology, 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.

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

Is very important to note that
In our implementation, data points
A bucket is a named location where time series data is stored.

Bibliography

[1] 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.