Contents

1 Introduction **3**

1.1 Serverless Computing Paradigm 3

1.1.1 Serverless Function 3

1.1.2 Serverless Application 4

1.1.3 fdsfsd 5

1.1.4 title 5

1.2 Motivation 6

1.3 Contribution 7

2 Conceptual Model **11**

2.1 Software system architecture 12

2.1.1 Resource owner 12

3 Performance Model Formalization **13**

3.1 SLAs model formalization 13

3.2 Serverless choreography 14

3.2.1 Preliminary definitions 15

3.2.2 Serverless choreography definition 16

3.3 Serverless Choreography Configuration 21

3.3.1 Executable Function Configuration 21

3.3.2 Serverless Choreography Configuration 22

3.4 Concrete Function Performance 23

3.4.1 Function Instances 23

3.4.2 FaaS auto-scaling technique 24

3.4.3 FaaS Request Routing 25

3.4.4 Concurrency limit . . . 25
3.4.5 FaaS platform modeling . . . 26
3.4.6 Serverless Concrete Function Swarms . . . 28
3.4.7 Cold Start Probability . . . 28
3.4.8 Concrete function performance . . . 30
3.5 The problem of cold starts . . . 31
3.5.1 Metrics . . . 31
3.6 Serverless Choreography Structures . . . 32
3.7 Optimization . . . 36
3.7.1 Performance Modeling . . . 36
3.7.2 Multi-dimensional Multi-choice Knapsack Problem . . . 38

4 Computational Model **43**
4.1 Abstract Function Choreography Language . . . 43
4.2 Pr . . . 43
4.2.1 InfluxDB . . . 44

Chapter 1

Introduction

1.1 Serverless Computing Paradigm

In serverless computing platforms, computation is done by so-called *function instances* which are completely managed by the serverless computing platform provider and act as tiny servers that can be invoked based on events forwarded by end-users [2].

Serverless computing platforms handle almost every aspect of the system administration tasks needed to deploy a workload on the cloud, providing a new simplified programming model according to which developers can focus on the business aspects of their applications only [1].

Moreover, the paradigm lowers the cost of deploying applications too, adopting a so-called "*pay as you go*" pricing model, by charging for execution time rather than for allocated resources [1].

1.1.1 Serverless Function

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

Although a serverless function generally represents a unit of executable code, submitted to FaaS platforms by developers using one or a combination of the programming languages supported by FaaS providers, a serverless function can be any cloud services eventually necessary to business logic, like cloud storage, message

queue services, pub/sub messaging service etc.

When it represents executable code, developers can specify several configuration parameters, like timeout, memory size or CPU power [1].

A serverless function can be triggered through events or HTTP requests following which the FaaS provider executes it in a containerized environment, like container, virtual machine or even processes, using the specified configuration.

1.1.2 Serverless Application

In serverless paradigm, the most basic scenario is to invoke a single function through a HTTP request; however, if you want to built more complex applications, constructing so-called *serverless application*, serverless functions must be connected and appropriately coordinated.

Formally, we define a serverless application as a stateless and event-driven software system made up of a serverless functions set hosted on one or more FaaS platforms and combined together by a so-called *coordinator* (or *orchestrator*).

Generally, the coordinator 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.

Most public cloud providers for serverless computing have introduced platforms to define and coordinate serverless function in order to built serverless application, like AWS Step Functions which combines multiple Lambda functions and other serverless services offered by AWS into responsive serverless applications

Although FaaS platforms continuously advance the support for serverless applications, existing solutions are dominated by a few large-scale providers resulting in mostly non-portable FaaS solutions

and provider lock-in

occurs when transitioning data, products, or services to another vendor's platform is difficult and costly, making customers more dependent (locked-in) on a single cloud storage solution.

FC languages and runtime systems are still in their infancy with numerous drawbacks. Current commercial and open source offerings of FC systems are dominated

by a few large- scale providers who offer their own platforms,

As FaaS platforms take over operational responsibilities, besides uploading the source code of functions, users have limited control over resources on FaaS platforms. Taking AWS Lambda as an example, the amount of allocated mem- ory during execution and the concurrency level are only op- tions for tuning the performance of functions. The amount of allocated memory is between 128 MB and 3,008 MB in 64MB increments [13]. Previous researches have proven that computational power and network throughput are in proportion to the amount of allocated memory, and disk performance also increases with larger memory size due to less contention [14], [15]. By reserving and provisioning more instances to host functions, high concurrency level can decrease fluctuations in the function performance incurred by cold starts (container initialization provisioning delay if no warm instance is available) and reduce the number of throttles under very heavy request loads

1.1.3 fdsfsd

-

1.1.4 title

the run-time of these serverless functions decreases with the increase of memory size allocated to the function. However, the marginal improvement in the run-time decreases as the memory increases.

This behavior is because the pricing model as exposed by the cloud providers is tightly coupled with the amount of resources specified to execute the serverless function (c.f. Figure 1a), and the dependency between memory and CPU resource allocation – AWS Lambda allocates CPU power

a model for each executable belonging to a given choreography.

This helps developers decide if a developed workload would comply with their QoS agreements, and if not, how much performance improvement they would need to do so. The performance improvement decided could be achieved either by improving the design, quality of code, or by simply resizing the resource allocated to each instance, which is usually set by changing the allocated memory.

A cold start

Cold/Warm start: as defined in previous work [3], [8], [10], we refer to cold start request when the request goes through the process of launching a new function instance. For the platform, this could include launching a new virtual machine, deploying a new function, or creating a new instance on an existing virtual machine, which introduces an overhead to the response time experienced by users. In case the platform has an instance in the idle state when a new request arrives, it will reuse the existing function instance instead of spinning up a new one. This is commonly known as a warm start request. Cold starts could be orders of magnitude longer than warm starts for some applications. Thus, too many cold starts could impact the application's responsiveness and user experience [3].

his equation gives the probability of a request being rejected (blocked) by the warm pool, assuming there are m warm servers. If m is less than the maximum concurrency level, the request blocked by the warm pool causes a cold start. If the warm pool has reached the maximum concur- rency level, any request rejected by the warm pool will be rejected by the platform.

1.2 Motivation

Although serverless paradigm guarantees a cost-effective solution, giving to, FaaS platform generally exhibit variable performance which precludes their use in applications that must meet strict guarantees.

1. scheduling policies today typically imple- ment basic rst-come-rst-served algorithms

FaaS and BaaS can exhibit variable performance which precludes their use in applications that must meet strict guarantees.

Despite the fact that serverless solution is cost-effective and eases the resource management, there are pain points hindering its wide adoption by potential users, including the lack of performance and cost model [6], [7], [8], and the trade-off analysis between the performance and cost of serverless applications [8], [9]. Performance and cost modeling and optimization are non-trivial and necessary steps towards guaranteeing the service level agreement (SLA) of serverless applications

in an economic manner, which is basically finding an acceptable trade-off between performance and cost. The difficulties of such modeling and optimization problems lie in the following aspects:

Serverless computing has given a much-needed agility to developers, abstracted away the management and maintenance of physical resources, and provided them with a relatively small set of configuration parameters: memory and CPU. While relatively simpler, configuring the "best" values for these parameters while minimizing cost and meeting perfor- mance and delay constraints poses a new set of challenges. This is due to several factors that can significantly affect the running time of serverless functions.

While many FaaS oerings exist [3, 4, 10, 12, 15, 17, 18, 20], relatively basic techniques to manage serverless function invocations are provided today. As function requests are re- ceived, the cloud provider schedules functions, mostly in a rst-in-rst-out manner.

Opportunities to invoke a more informed scheduling policy are missed, however, in chal- lenging scenarios such as when incoming demands cannot be satised by currently available resources. Such scenarios occur when providers cannot accommodate a rise of invo- cations due to cold starts or inecient resource allocation or alternatively when function invocation limits are imposed. Function invocation limits bound the number of functions running either instantaneously or over a time period. Be- cause serverless technologies automatically scale to meet demands, function invocation limits ensure a bug or miscon- guration in tenant workloads does not inappropriately scale. In addition, limits help developers manage costs and better

Thus, a prospective user could choose the services that best suit his/her QoS requirements.

1.3 Contribution

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, parallelism, 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 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.

2.1 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

Performance Model Formalization

3.1 SLAs model formalization

Our framework considers SLAs whose conditions should be hold at *local level*, focusing only on the fulfillment of the requirements regard a single serverless application invocation request forwarded by one user.

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 users.

As known, a SLA may include a large set of parameters, referring to different kinds of *functional* and *non-functional* attributes, including several ways of measuring them.

Then, our framework's goal is to exploit its self-adaptation capability to fulfill *non-functional* SLA requirements, considering 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;

According to our model, for each serverless application invocation, SLAs requirements can be expressed by end users 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) is a scalar representing the weight of response time's (cost's) requirement; 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 end user expectations.

  Formally, regarding these weights, following constraints must be hold:

$$w_{RT}, w_C \in \mathbb{R}^+ \cap [0, 1] \tag{3.2}$$

$$w_{RT} + w_C = 1 \tag{3.3}$$

To be precise, the goal of our framework is to guarantee that thresholds RT and C will hold on average.

We will describe later how weights for the different SLAs attributes have been used to identify the best suitable configuration for a given serverless application invocation.

3.2 Serverless choreography

According to our model, a *serverless choreography* represent the *resource* used to abstract both the concepts of serverless functions and serverless applications.

Informally, that abstraction has been derived from that of a *control-flow graph* which, as known, describes, using graph notation, all paths that might be traversed through a serverless application during its execution [TODO].

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

3.2.1 Preliminary definitions

In order to provide a formal definition of serverless 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:

$$\pi = \phi_1 e_1 \phi_2 e_2 \dots e_{n-2} \phi_{n-1} e_{n-1} \phi_n \tag{3.4}$$

  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:

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

$$in(\phi_u) \stackrel{def}{=} \{(\phi, \phi_u) \in E, \quad \forall \phi \in \Phi\} \tag{3.6}$$

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

$$pred(\phi_u) \stackrel{def}{=} \{\phi \in \Phi \mid (\phi, \phi_u) \in in(\phi_u)\} \tag{3.8}$$

3.2.2 Serverless choreography definition

According to serverless paradigm, the execution of a serverless application always starts with a particular serverless 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 specified business logic. Finally, the execution of a serverless application ends when the execution of its last function ends, which acts as the "*end point*" of the serverless workflow. We believe that aforementioned serverless application's behavior can be naturally modeled by a weighted directed graph.

Assuming that any serverless application has only one function acting as entry point, we can provide a formal definition of serverless choreography as follows.

Let $n \in \mathbb{N} \setminus \{0\}$ and R a resource owner.

A *serverless choreography* owned by R, or simply *choreography* of R, is a resource represented by a weighted directed graph which is denoted as follows:

$$\mathcal{C}_R \stackrel{def}{=} (\Phi, E) \tag{3.9}$$

where:

- Each vertex $\phi \in \Phi$, where $|\Phi| = n$, is called *abstract serverless function* and represents a computational unit; we will describe more in detail what we mean by abstract serverless function shortly;

- 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 serverless function which depends on the business logic defined by R.

  In our context, any directed edge $(\phi_i, \phi_j) \in E$ states that the abstract serverless 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 after finishing the execution of ϕ_i;
  - We will use a function $P : \Phi \times \Phi \to [0, 1]$, called *transition probability function*, such that $P(\phi_i, \phi_j) = p_{ij}$. When $P(\phi_i, \phi_j) = 0$, it implies that the directed edge $(\phi_i, \phi_j) \notin E$, therefore ϕ_i cannot invoke ϕ_j;
  - For any path $\pi = \phi_1 e_1 \dots e_{n-1} \phi_n$ of $\mathcal{C}_R$, we define *transition probability of the path* π the following quantity:

$$TPP(\pi) = \prod_{i=1}^{n-1} P(\phi_i, \phi_{i+1}) \tag{3.10}$$

- Φ must be such that following condition are hold:

$$\exists! \phi \in \Phi \quad | \quad in(\phi) = \emptyset \tag{3.11}$$

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

  that is, Φ has to contain only one serverless abstract function acting as entry point of the serverless application represented by $\mathcal{C}_R$ and, moreover, it must contain at least one acting as end point.

  Formally, we state that:

$$\begin{aligned} \phi \in \Phi \quad &\text{is the } \textit{entry point} \text{ of } \mathcal{C}_{\mathcal{R}} \quad \Leftrightarrow \quad in(\phi) = \emptyset \\ \phi \in \Phi \quad &\text{is an } \textit{end point} \text{ of } \mathcal{C}_{\mathcal{R}} \quad \Leftrightarrow \quad out(\phi) = \emptyset \end{aligned} \tag{3.13}$$

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

- Must be hold the condition according to which the transition probabilities of all paths between the entry point and the end point of $\mathcal{C}_R$ sum up to 1. Formally:

$$\sum_{\phi_{end} \in \omega(\mathcal{C}_R)} \Big(\sum_{\pi \in \Pi(\phi_{entry}, \phi_{end})} TPP(\pi) \Big) = 1 \tag{3.14}$$

  where ϕ_{entry} is the entry point of $\mathcal{C}_R$.

  In other words, above conditions guarantees that any execution starting from ϕ_{entry} will terminate.

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.

Clearly, we say that choreography models a serverless function when $|\Phi| = 1$ and $|E| = 0$; conversely, it models a serverless function 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 $\phi \in \Phi$ is called *abstract serverless function*, or simply *abstract function*, that is a *resource* representing 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 directly 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:

$$\mathcal{F}_{\mathcal{E}}(\mathcal{C}) \stackrel{def}{=} \{\phi \in \Phi \mid \phi \text{ is a serverless executable function } \} \tag{3.15}$$

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 objective.

- 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:

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

Clearly, based on above definitions, we can say:

$$\mathcal{F}_{\mathcal{E}}(\mathcal{C}) \cap \mathcal{F}_{\mathcal{O}}(\mathcal{C}) = \emptyset \tag{3.17}$$

$$\mathcal{F}_{\mathcal{E}}(\mathcal{C}) \cup \mathcal{F}_{\mathcal{O}}(\mathcal{C}) = \Phi \tag{3.18}$$

$$|\mathcal{F}_{\mathcal{E}}(\mathcal{C})| + |\mathcal{F}_{\mathcal{O}}(\mathcal{C})| = |\Phi| \tag{3.19}$$

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

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

Is very important to note that our system software has been not designed to manage serverless concrete function hosted on a FaaS provider. In other words, at the current state of our framework, is not possible to execute CRUD (*Create*, *Read*, *Update*, and *Delete*) operations regarding concrete functions on FaaS platform using our tool.

Therefore, we always assume that all $f_\phi \in \mathbf{F}_\phi$, for all $\phi \in \mathcal{F}_{\mathcal{E}}(\mathcal{C})$, are already deployed on one or more FaaS platform by developers.

Then, in order to effectively start the execution of the choreography $\mathcal{C}$, is required that, for each executable function $\phi \in \mathcal{F}_{\mathcal{E}}(\mathcal{C})$, *at least one* concrete implementation f_ϕ exists.

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

$$\mathcal{C} \text{ is executable} \quad \Leftrightarrow \quad |\mathbf{F}_\phi| \geq 1 \qquad \forall \phi \in \mathcal{F}_{\mathcal{E}}(\mathcal{C}) \tag{3.20}$$

We will only deal with executable choreographies.

3.2.2.3 Serverless Sub-choreography

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

The weighted directed sub-graph of $\mathcal{C}$ defined as follows:

$$\mathcal{C}^* \stackrel{def}{=} (\Phi^*, E^*) \qquad \text{where } \Phi^* \subseteq \Phi \wedge E^* \subseteq E \tag{3.21}$$

is called *serverless sub-choreography* of $\mathcal{C}$, or simply *sub-choreography* of $\mathcal{C}$, when the conditions 3.11, 3.12 and 3.14 are hold.

3.3 Serverless Choreography Configuration

Given a choreography $\mathcal{C} = (\Phi, E)$, the basic goal of our framework is to determine a so-called *serverless choreography configuration*, called also *choreography configuration* or, simply, *configuration*, allowing us to meet user-specified QoS objectives.

Technically, a choreography configuration is the result produced by our framework after the execution of the "*Plan*" phase of MAPE-K loop; it contains all information such that, during the "*Execution*" phase of MAPE-K loop:

- For each $\phi \in \mathcal{F}_{\mathcal{E}}(\mathcal{C})$, only one concrete function $f_\phi \in \mathbf{F}_\phi$ will be effectively invoked and executed on FaaS platform
- For each selected concrete function f_ϕ, a memory configuration must be selected.

In simple terms, a choreography configuration specifies which concrete function will be used for all executable function inside a choreography, including theirs memory size configuration parameter.

3.3.1 Executable Function Configuration

In this section, we will explain the concept of *executable function configuration*.

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:

$$x_\phi = (f_\phi, m) \in f_\phi \times \mathbf{M}_{f_\phi} \subseteq \mathbf{F}_\phi \times \mathbb{N} \tag{3.22}$$

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:

$$
\begin{aligned}
\mathbf{X}_\mathcal{C} &\overset{def}{=} \{x_{\phi_1}, \ldots, x_{\phi_k}\} \\
&\in \left\{ \left\{ \bigcup_{j=1}^{|\mathbf{F}_{\phi_1}|} f_{\phi_{1_j}} \times \mathbf{M}_{f_{\phi_{1_j}}} \right\} \times \ldots \times \left\{ \bigcup_{j=1}^{|\mathbf{F}_{\phi_k}|} f_{\phi_{k_j}} \times \mathbf{M}_{f_{\phi_{k_j}}} \right\} \right\} \\
&= \bigtimes_{i=1}^{k} \left\{ \bigcup_{j=1}^{|\mathbf{F}_{\phi_i}|} f_{\phi_{i_j}} \times \mathbf{M}_{f_{\phi_{i_j}}} \right\} \\
&\subseteq \bigtimes_{i=1}^{k} \{\mathbf{F}_{\phi_i} \times \mathbb{N}\}
\end{aligned} \tag{3.23}
$$

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 $j \in \mathbb{N} \cap [1, |\mathbf{F}_{\phi_i}|]$

3.4 Concrete Function Performance

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 invocation cost.

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 although there are reports stating that, despite there is no explicitly mentioned global concurrency limit, a kind of global concurrency limit, varying between 1000 and 2000, is observed.

3.4.5 FaaS platform modeling

At this point, we can provide an analytical model to better understand how FaaS platforms work.

According to our model point of view, any FaaS platform provider act as a "*set*" of $M/G/K(t)_{\mathbf{C}_{max}}/K(t)_{\mathbf{C}_{max}}$ queueing systems, that is a $K(t)$-*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, representing therefore the maximum number of function instances being in running state simultaneously at time t.

  Please note that $\mathbf{C}_{max}$ not necessary coincide with 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.

  In other words, the number of function instance, and consequently the capacity of system, can change over time.

At any time t, following condition must be hold:

$$0 \leq K(t) \leq C_{\mathbf{max}} \tag{3.24}$$

- 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 which sees all $K(t)$ function instances in running state, will be permanently dropped.
- No priority is considered among incoming request.
- Service times have a general distribution while a Poisson arrival process is assumed.

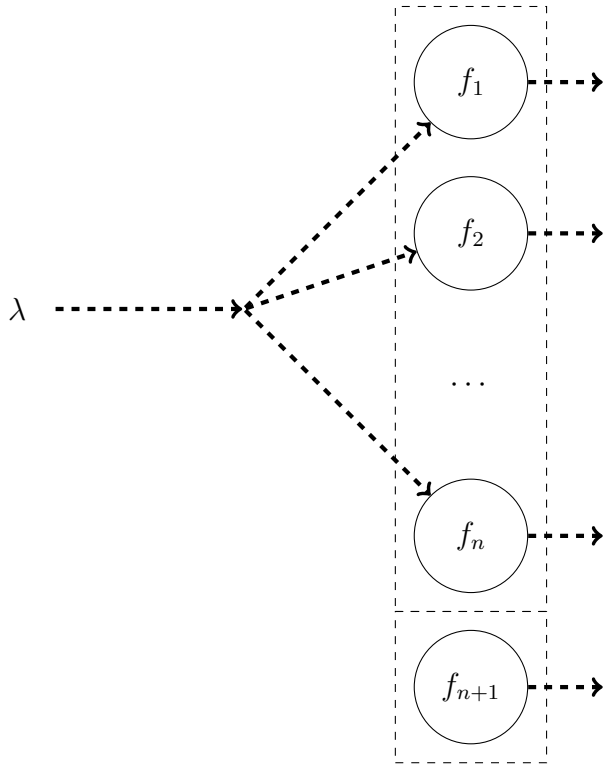

Figure 3.1: Parallel structure in the serverless workflow.

3.4.6 Serverless Concrete Function Swarms

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

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

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

In simple terms, only at most l executions of all 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 request regarding any concrete function belonging to Ω_{P_R}.

3.4.7 Cold Start Probability

As known, the rejection of a request by the warm pool triggers a cold start, which, subsequently, adds a new function instance to the warm pool in order to handle received request.

In order to fully describe how our framework works, 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*.

Precisely, let $l \in \mathbb{N}$, R a resource owner, P a serverless computing platform provider, $\omega_{P_R}^{(l)}$ a swarm and $M/G/K(t)_l/K(t)_l$ the queuing system associated to that swarm. Moreover, let $r(t)$ the number of function instances being in running state at time t.

When a new request of invocation of a function belonging $\omega_{P_R}^{(l)}$, arrives on the FaaS platform P, one of the following events can occur:

- If at time t_n is verified that $K(t_n) < l$ 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 at time t_n is verified that $K(t_n) < l$ 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 at time t_n is true that $K(t) = l$ and $r(t_n) = K(t_n)$, that is the warm pool reaches is maximum possible size according to FaaS policies, aforementioned request will be rejected.

Clearly, when at time tn a execution's request has been completed by a function instance, that event cause that $r(t_{n+1}) = r(t_n) - 1$.

In the same way, if a function instance has not been received in 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* $\mathbf{P}(t)$ 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 $K(t)$ function instances of the warm pool busy. Using Erlang-B formula, aforementioned probability can be calculated as follows:

$$\mathbf{P}(t) = \frac{\frac{\rho(t)^{K(t)}}{K(t)!}}{\sum_{j=0}^{K(t)} \frac{\rho(t)^j}{j!}} \tag{3.25}$$

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 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:

    $$\lambda(t) \stackrel{def}{=} \begin{cases} Y_o & \text{if } t = 0 \\ \alpha \cdot Y_t + (1-\alpha) \cdot \lambda(t-1) & \text{if } t > 0 \end{cases}. \tag{3.26}$$

  - $\mu_w = E[S_w]$ 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.

3.4.8 Concrete function performance

At this point, we can define some useful functions:

- $C_{\mathbf{F}_\phi} : \mathbf{F}_\phi \times \mathbb{N} \to [0, \infty)$ is the *cost function* for any serverless functions belonging to the implementation-set $\mathbf{F}_\phi$.

  Precisely, for all $f_\phi \in \mathbf{F}_\phi$ and $m \in \mathbb{N}$, $C_{\mathbf{F}_\phi}(f_\phi, m)$ returns the *average cost* paid by developers to execute f_ϕ using an allocated memory size equal to m.

- $RT_{\mathbf{F}_\phi} : \mathbf{F}_\phi \times \mathbb{N} \to [0, \infty)$ is a *delay function* for any serverless functions belonging to the implementation-set $\mathbf{F}_\phi$.

  For all $f_\phi \in \mathbf{F}_\phi$ and $m \in \mathbb{N}$, $RT_{\mathbf{F}_\phi}(f_\phi, m)$ returns the *average response time* when f_ϕ is invoked with memory size equal to m;

3.5 The problem of cold starts

In order to find a suitable serverless choreography configuration capable to satisfy QoS imposed by end users, we must find a way to decide

3.5.1 Metrics

Let $\mathcal{C} = (\Phi, E)$ a basic choreography as defined in section **??**.

Suppose to have a path $\pi = \phi_1 e_1 \phi_2 e_2 \dots e_{n-2} \phi_{n-1} e_{n-1} \phi_n$ and a choreography configuration $\mathbf{X}$ for $\mathcal{S}$.

We can define the *response time* of π given $\mathbf{X}$ as follows:

$$RT_P(\pi, \mathbf{X}) = \sum_{i=1}^{n} (RT_{\phi_i}(X(i)) + \sum_{i=1}^{n-1} D(\phi_i, \phi_{i+1}) \tag{3.27}$$

Similarly, we define the cost of π given $\mathbf{X}$ as follows:

$$C_P(\pi, \mathbf{X}) = \sum_{\substack{1 \le i \le n \\ \phi_i \in \mathcal{F}_\mathcal{E}}} N(\phi_i) \cdot C_{\mathbf{F}_{\phi_i}}(X(i)) \tag{3.28}$$

Then the response time and the cost of the choreography $\mathcal{S}$ a follows:

$$RT_C(\mathcal{S}, \mathbf{X}) = \sum_{\pi \in \Pi(\phi_{entry}, \phi_{end})} TPP(\pi) \cdot RT_P(\pi, \mathbf{X}) \tag{3.29}$$

$$C_C(\mathcal{S}, \mathbf{X}) = \sum_{\pi \in \Pi(\phi_{entry}, \phi_{end})} TPP(\pi) \cdot C_P(\pi, \mathbf{X}) \tag{3.30}$$

3.6 Serverless Choreography Structures

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

Formally

Supposing to have a serverless choreography $\mathcal{S} = (\Phi, E)$, we call *structure* any sub-choreography $\mathcal{S}^* = (\Phi^*, E^*)$ whose entry point and the end point are, respectively, an opening and a closing orchestration functions of the same type. Formally:

$$S^* \text{ is a structure } \Leftrightarrow \begin{cases} \phi^*_{entry} & \in \mathcal{F}^*_{\mho} \quad \wedge \quad \mathcal{T}(\phi^*_{entry}) = \tau_\alpha \\ \phi^*_{end} & \in \mathcal{F}^*_{\mho} \quad \wedge \quad \mathcal{T}(\phi^*_{end}) = \tau_\omega \end{cases} \tag{3.31}$$

The most important aspect is that every structure can be viewed as a set of sub-choreographies of $\mathcal{S}^*$. Formally, suppose that $\Lambda = (A, B)$ is a graph such that:

$$\begin{aligned} A &\stackrel{def}{=} \Phi^* \setminus \left\{\phi^*_{entry}, \phi^*_{end}\right\} \\ B &\stackrel{def}{=} E^* \setminus \left(out\left(\phi^*_{entry}\right) \cup in\left(\phi^*_{end}\right)\right) \end{aligned} \tag{3.32}$$

Let $c \in \mathbb{N}$ and suppose that c is the number of connected components of the graph Λ.

We denote by $\Theta_{\mathcal{S}^*}$ as the set containing all connected components of Λ every of which is considered as a sub-choreography of $\mathcal{S}^*$. Formally:

$$\begin{gathered} \Theta_{\mathcal{S}^*} \stackrel{def}{=} \{\theta_i, \ldots, \theta_c\} \\ \text{where} \\ \theta_i \stackrel{def}{=} (\Phi^{**}_i, E^{**}_i) \text{ is a sub-choreography of } \mathcal{S}^* \\ \Phi^{**}_i \subset \Phi^* \wedge E^{**}_i \subset E^* \qquad \forall i \in [1, c] \end{gathered} \tag{3.33}$$

3.6.0.1 Parallel

A structure $\mathcal{P} = (\Phi, E, \Theta)$ is called *parallel structure* when:

$$TPP(\pi) = 1 \qquad \forall \pi \in \Pi(\phi_{start}, \phi_{end}) \tag{3.34}$$

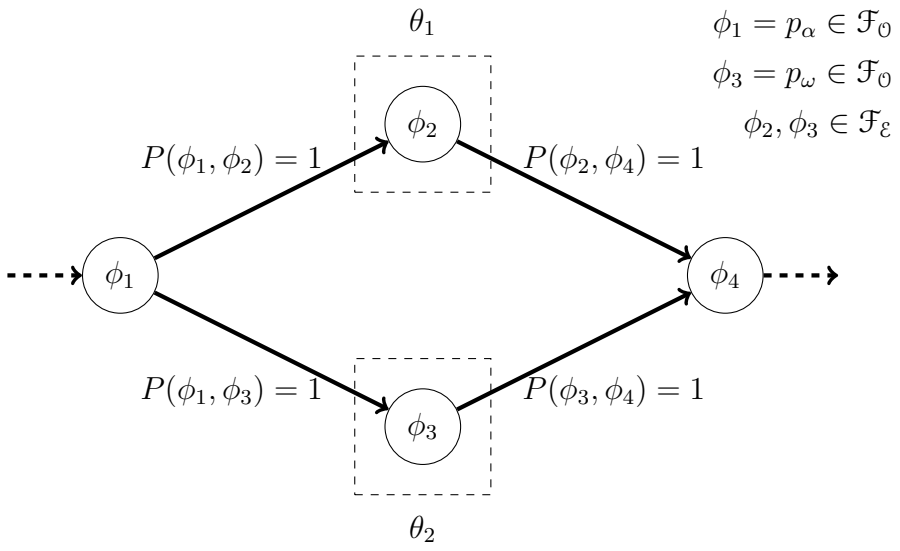

Figure 3.2: Parallel structure in the serverless workflow.

Let $n \in \mathbb{N}$, supposing that $|\Theta| = n$, the response time of a parallel structure is the longest response time of all sub-choreographies $\theta_i \in \Theta$ for all $i \in [1, n]$, while its cost is equal to the sum their costs of execution. Formally, being $\mathbf{X}$ a configuration of $\mathcal{P}$:

$$RT_{parallel}(\mathcal{P}, \mathbf{X}) = max\left\{RT_C(\theta, \mathbf{X}) \mid \theta \in \Theta\right\} \tag{3.35}$$

$$C_{parallel}(\mathcal{P}, \mathbf{X}) = \sum_{\theta \in \Theta} C_C(\theta, \mathbf{X}) \tag{3.36}$$

3.6.0.2 Branch/Switch

Let $\mathcal{B} = (\Phi, E, \Theta)$ a structure and suppose that $\alpha(\mathcal{B}) = \phi_{entry}$ and $\omega(\mathcal{B}) = \phi_{end}$.

$\mathcal{B}$ is called *branch structure* when following conditions are hold:

$$TPP(\pi) \neq 1 \qquad \forall \pi \in \Pi(\phi_{entry}, \phi_{end}) \tag{3.37}$$

$$|\Theta| = n \qquad n \in \{1, 2\} \tag{3.38}$$

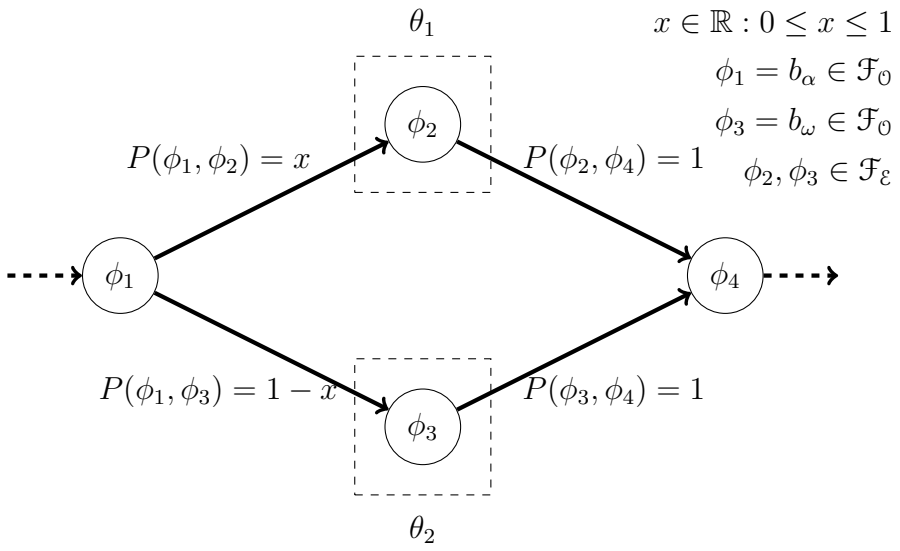

Figure 3.3: Branch/Switch structure in the serverless workflow.

Let all $\theta_i \in \Theta$ sub-choreographies of $\mathcal{B}$, for all $i \in \{1, 2\}$. Then the response time and the cost of a branch structure can be defined as follows:

$$RT_{branch}(\mathcal{B}, \mathbf{X}) = \sum_{i=1}^{n} P(\phi_{entry}, \alpha(\theta_i)) RT_C(\theta_i, \mathbf{X}) \tag{3.39}$$

$$C_{branch}(\mathcal{B}, \mathbf{X}) = \sum_{i=1}^{n} C_C(\phi_{entry}, \alpha(\theta_i)) C_C(\theta_i, \mathbf{X}) \tag{3.40}$$

Finally, according to our model, $\mathcal{B}$ is called *switch structure* when $|\Theta| > 2$.

3.6.0.3 Loop

Let $\mathcal{L} = (\Phi, E, \Theta)$ a structure and suppose that $\alpha(\mathcal{L}) = \phi_{entry}$ and $\omega(\mathcal{L}) = \phi_{end}$.

If $|\Theta| = 1$ and $(\phi_{end}, \phi_{entry}) \in E$, denoting by θ the unique sub-choreography of $\mathcal{L}$, $\mathcal{L}$ will be called *loop structure* when following conditions are hold:

$$P(\phi_{start}, \alpha(\theta)) = 1 \tag{3.41}$$

$$P(\phi_{end}, \phi_{entry}) \neq 0 \tag{3.42}$$

$$P(\phi_{end}, \phi_{entry}) + \sum_{\phi \in succ(\phi_{end})} P(\phi_{end}, \phi) = 1 \tag{3.43}$$

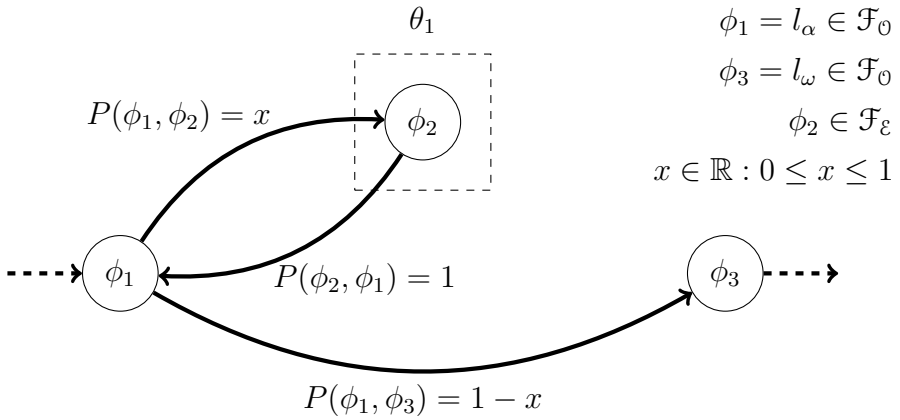

Figure 3.4: Parallel structure in the serverless workflow.

In order to compute the mean response time and the mean cost of a loop structure, we need to know $EI(\mathcal{L})$, that is the expected value of the number of iterations of $\mathcal{L}$.

We know that geometric distribution gives the probability that 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:

$$p = 1 - P(\phi_{end}, \phi_{entry}) \tag{3.44}$$

Then:

$$P(EI(\mathcal{L}) = k) = \Big[1 - P(\phi_{end}, \phi_{entry})\Big] \cdot \Big[P(\phi_{end}, \phi_{entry})\Big]^{k-1} \quad (3.45)$$

$$= pq^{k-1} \quad (3.46)$$

$$\begin{aligned} E[X] &= \sum_{k=1}^{\infty}(k-1)pq^{k-1} \\ &= p\sum_{k=0}^{\infty} kq^k \\ &= p \cdot \frac{q}{(1-q)^2} \\ &= \frac{q}{p} \\ &= \frac{P(\phi_{end}, \phi_{entry})}{1 - P(\phi_{end}, \phi_{entry})} \end{aligned} \quad (3.47)$$

Then the response time and the cost of a loop structure can be defined as follows:

$$RT_{loop}(\mathcal{L}, \mathbf{X}) = E[I_{\mathcal{L}}] \cdot RT_C(\theta, \mathbf{X}) \quad (3.48)$$

$$C_{loop}(\mathcal{L}, \mathbf{X}) = E[I_{\mathcal{L}}] \cdot C_C(\theta, \mathbf{X}) \quad (3.49)$$

3.7 Optimization

3.7.1 Performance Modeling

We define the utility function as follows:

$$F(\mathbf{x}) \stackrel{def}{=} w_{RT} \cdot \frac{RT_{max} - RT(\mathbf{x})}{RT_{max} - RT_{min}} + w_C \cdot \frac{C_{max} - C(\mathbf{x})}{C_{max} - C_{min}} \quad (3.50)$$

where $w_{RT}, w_C \geq 0$, $w_{RT} + w_C = 1$, are weights for the different QoS attributes, while RT_{max} (RT_{min}) and C_{max} (C_{min}) denote, respectively, the maximum (mini-

mum) value for the overall expected response time and cost.

Following formulation is based on the enumeration of all possible choreography configurations, that is of all combinations of executable function configuration of a given choreography.

For the sake of conciseness, in the following we use following notation:

p to define both a pattern and its index, and Ω to define both the set of patterns and the set of patterns indices.

$$\begin{aligned} max \quad & \sum_{\omega=1}^{|\Omega|} x_\omega F(\mathbf{X}_\omega) \\ \text{subject to} \quad & \sum_{\omega=1}^{|\Omega|} x_\omega C(\mathbf{X}_\omega) \leq C_{user} \\ & \sum_{\omega=1}^{|\Omega|} x_\omega RT(\mathbf{X}_\omega) \leq RT_{user} \\ & \sum_{\omega=1}^{|\Omega|} x_\omega = 1 \\ & x_\omega \in \{0,1\} \qquad\qquad \forall \omega \in \mathbb{N} \cap [1, |\Omega|] \end{aligned} \tag{3.51}$$

Above formulation is based on the enumeration of all possible choreography configuration.

3.7.2 Multi-dimensional Multi-choice Knapsack Problem

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

For the sake of dasdasdsadsadasdsadasdas, we will use $\widetilde{\mathcal{P}}$ to denote the set containing all parallel structures of $\mathcal{C}$, while $\widetilde{\mathcal{P}}_i$ is used to refer to i-th parallel structure of $\widetilde{\mathcal{P}}$, where $i \in \mathbb{N} \cap \left[1; |\widetilde{\mathcal{P}}|\right]$.

Let's define $\Delta_{\widetilde{\mathcal{P}}_{i_j}}$ the set containing all executable functions $\phi \in \mathcal{F}_{\mathcal{E}}$ belonging to the j-th sub-choreography of the parallel structure $\widetilde{\mathcal{P}}_i$. Formally:

$$\Delta_{\widetilde{\mathcal{P}}_{i_j}} \stackrel{def}{=} \{\phi \in \mathcal{F}_{\mathcal{E}} \cap \mathcal{F}_{\mathcal{E}}(\theta_j) \text{ where } \theta_j \in\} \tag{3.52}$$

We

$$\begin{aligned} \chi &\stackrel{def}{=} \left\{\Delta_{\widetilde{\mathcal{P}}_1}, \ldots, \Delta_{\widetilde{\mathcal{P}}_n}\right\} \\ &\in \left\{\left\{\bigcup_{j=1}^{|\widetilde{\mathcal{P}}_i|} \Delta_{\widetilde{\mathcal{P}}_{1_j}}\right\} \times \ldots \times \left\{\bigcup_{j=1}^{|\widetilde{\mathcal{P}}_n|} \Delta_{\widetilde{\mathcal{P}}_{n_j}}\right\}\right\} \\ &= \bigtimes_{i=1}^{n} \left\{\bigcup_{j=1}^{|\widetilde{\mathcal{P}}_i|} \Delta_{\widetilde{\mathcal{P}}_{1_j}}\right\} \end{aligned} \tag{3.53}$$

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

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

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

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

$$x_{\phi_{i_j}} \in \{0, 1\} \qquad \forall i \in \mathbb{N} \cap [1, |\mathcal{F}_\mathcal{E}|] \wedge \forall j \in$$

(3.54)

The MMKP is an NP-hard in the strong sense [2]. It is not always possible to find a feasible solution in a reasonable computing time especially for big instances. Since exact methods are reserved for limited scale problem, heuristic approaches are needed for solving it.

Let $\mathcal{C} = (\Phi, E)$ a serverless choreography, $x_{\phi_{i_j}} = (f_j, m_j)$ an executable function configuration 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 $j \in \mathbb{N} \cap [1, |\mathbf{F}_{\phi_i} \times \mathbb{N}|]$.

Let's define $\widehat{\theta_k}^{(\phi_i)} = \theta_1^{(\phi_i)}, \ldots, \theta_n^{(\phi_i)}$ as the sequence of all sub-choreographies of $\mathcal{C}$ such that:

$$\phi_i \in \mathcal{F}_\mathcal{E}(\theta_1^{(\phi_i)}) \tag{3.55}$$

$$\theta_n^{(\phi_i)} = \mathcal{C} \tag{3.56}$$

$$\theta_k^{(\phi_i)} \text{ is a sub-choreography of } \theta_{k+1}^{(\phi_i)} \qquad \forall k \in [1; n-1] \tag{3.57}$$

At this point, we can define $E[I_{\phi_i}]$, that is the expected value of the number of invocations of the executable function ϕ_i, as follows.

$$E[I_{\phi_i}] \stackrel{def}{=} \begin{cases} 1 & if \quad n = 1 \\ \prod_{k=2}^{n} E[I_{\theta_{k-1}^{(\phi_i)}}] & if \quad n \geq 2 \end{cases} \tag{3.58}$$

For some $k \in \mathbb{N} \cap [1; n-1]$, we will use $P_{exe}^{C}(\theta_k^{(\phi_i)})$ notation to refer to the probability to execute the abstract function entry point $\alpha(\theta_k^{(\phi_i)})$ of the choreography $\theta_k^{(\phi_i)}$, when the abstract function $pred(\alpha(\theta_k^{(\phi_i)}))$ is invoked. That probability can be computed as follows:

$$P_{exe}^{C}(\theta_k^{(\phi_i)}) \stackrel{def}{=} \begin{cases} 1 & if \quad n = 1 \\ P\Big(pred\Big(\alpha(\theta_k^{(\phi_i)})\Big), \alpha(\theta_k^{(\phi_i)})\Big) & if \quad n \geq 2 \end{cases} \tag{3.59}$$

Finally, we define $P_{exe}^{F}(\theta, \phi)$ as the probability to execute the executable function $\phi \in \mathcal{F}_{\mathcal{E}}(\theta)$ when the abstract function $\alpha(\theta)$ is invoked. We can compute this probability using following formula:

$$P_{exe}^{F}(\theta, \phi) \stackrel{def}{=} \prod_{\phi_k \in \mathcal{F}_{\mathcal{O}}(\theta) \cap preds(\phi)} (1 - P_{exit}(\phi_k)) \tag{3.60}$$

Defining:

$$\Gamma_{\phi_{i_j}} \stackrel{def}{=} E[I_{\phi_i}] \prod_{k=2}^{n} P_{exe}^{C}(\theta_{k-1}^{(\phi_i)}) \cdot P_{exe}^{F}(\theta_k^{(\phi_i)}, pred(\alpha(\theta_{k-1}^{(\phi_i)}))) \tag{3.61}$$

we can compute resource requirements $c_{\phi_{i_j}}$ and $t_{\phi_{i_j}}$ in the following ways:

$$c_{\phi_{i_j}} \stackrel{def}{=} \begin{cases} C_{avg}(x_{\phi_{i_j}}) \cdot P_{exe}^{F}(\theta_1^{(\phi_i)}, \phi_i) & \textbf{if} \quad n = 1 \\ C_{avg}(x_{\phi_{i_j}}) \cdot P_{exe}^{F}(\theta_1^{(\phi_i)}, \phi_i) \cdot \Gamma_{\phi_{i_j}} & \textbf{if} \quad n \geq 2 \end{cases} \tag{3.62}$$

$$t_{\phi_{i_j}} \stackrel{def}{=} \begin{cases} T_{avg}(x_{\phi_{i_j}}) \cdot P_{exe}^{F}(\theta_1^{(\phi_i)}, \phi_i) & \textbf{if} \quad n = 1 \\ T_{avg}(x_{\phi_{i_j}}) \cdot P_{exe}^{F}(\theta_1^{(\phi_i)}, \phi_i) \cdot \Gamma_{\phi_{i_j}} & \textbf{if} \quad n \geq 2 \end{cases} \tag{3.63}$$

Finally, for any executable function configuration $x_{\phi_{i_j}}$, its profit $p_{\phi_{i_j}}$ is determined as follows;

$$p_{\phi_{i_j}} \stackrel{def}{=} w_{RT} \cdot \frac{t_{\phi_{i_{\mathbf{MAX}}}} - t_{\phi_{i_j}}}{t_{\phi_{i_{\mathbf{MAX}}}} - t_{\phi_{i_{\mathbf{MIN}}}}} + w_C \cdot \frac{c_{\phi_{i_{\mathbf{MAX}}}} - c_{\phi_{i_j}}}{c_{\phi_{i_{\mathbf{MAX}}}} - c_{\phi_{i_{\mathbf{MIN}}}}} \tag{3.64}$$

where:

- $t_{\phi_{i_{\mathbf{MIN}}}}$ and $t_{\phi_{i_{\mathbf{MAX}}}}$ represent, respectively, the minimum and maximum response time values registered during the execution of all concrete function implementing ϕ_i.
- $c_{\phi_{i_{\mathbf{MIN}}}}$ and $c_{\phi_{i_{\mathbf{MAX}}}}$ represent, respectively, the minimum and maximum cost values spent by all concrete function implementing ϕ_i.
- w_{RT} and w_C are weights for the different QoS attributes such that:

$$w_C \geq 0 \tag{3.65}$$

$$w_{RT} \geq 0 \tag{3.66}$$

$$w_{RT} + w_C = 1 \tag{3.67}$$

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
```

Data: this text
Result: Some solution

for $k \leftarrow 0$ **to** $|\mathcal{A}|$ **by** 1 **do**
  $S_k \leftarrow \emptyset$;
  $G_{candidates} \leftarrow \mathcal{G}$;
  $\mathbf{G}_i \leftarrow$ Randomly select a group from $G_{candidates}$ for some $i \in [1; |\mathcal{G}|]$;
  $o_{i_k} \leftarrow$ Randomly select an object from $\mathbf{G}_i$ for some $k \in [1; |\mathbf{G}_i|]$;
  $S_k \leftarrow \{o_{i_k}\}$;
  $G_{candidates} \leftarrow G_{candidates} \setminus \mathbf{G}_i$;
  while $G_{candidates} \neq \emptyset$ **do**
    $\mathbf{G}_i \leftarrow$ Randomly select a group from $G_{candidates}$;
    $\mathcal{O} \leftarrow$ without violating resource constraints ;
  end
end

Chapter 4

Computational Model

We have adopted for follwoing reasons:

1. We are able to guarantee access transparency.

   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.

[2] Nima Mahmoudi and Hamzeh Khazaei. Performance modeling of serverless computing platforms. *IEEE Transactions on Cloud Computing*, pages 1–1, 2020.

[3] Xiongfei Weng, Hongliang Yu, Guangyu Shi, Jian Chen, Xu Wang, Jing Sun, and Weimin Zheng. Understanding locality-awareness in peer-to-peer systems. In *2008 International Conference on Parallel Processing - Workshops*, pages 59–66, 2008.