Università degli Studi di Roma "Tor Vergata"

A QoS-Aware Orchestrator Broker
for Multi-Provider Serverless Applications

Studente:

Andrea Graziani

Matricola 0273395

Docente:

Prof.ssa Valeria Cardellini

Correlatore:

Dott. Gabriele Russo Russo

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 applications 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 following

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 [10]. Function instances acts as tiny servers completely managed by the serverless computing platform.

As known, 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 separately in a serverful context, like provisioning, redundancy, availability, monitoring, logging, and automated scaling, some studies claim that, in practice, customers see cost savings of $4\times$-$10\times$ when moving applications to serverless [17].

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

We define a serverless application as a stateless and event-driven software system made up of a serverless functions set hosted on one or more FaaS platforms and combined together by a so-called *coordinator* (or *orchestrator*). The latter is usually represented by 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 which provide serverless computing services in order to deploy and execute serverless functions, including AWS Lambda [1], IBM Cloud Function [6], Microsoft Azure Functions [2] and Google Cloud Functions [4]. 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.

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

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

- the lack of support for those applications having fine-grained state sharing needs, which is primarily due to stateless nature of serverless platforms [13].

- 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 [18].

- 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 [10][13].

- security concerns due to fine-grained resource sharing of serverless function which increases the exposure to *side-channel* attacks [17]

1.2 Motivations of the study

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

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

To be precise, aforementioned model is needed to select a configuration for each serverless function belonging to a given application. In fact, as we said previously, when cloud application developers want to run a serverless function, is necessary to specify several configuration'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. In fact, several studies have already shown that aforementioned configuration parameters significantly affect the cost and response time of any serverless functions; for instance, Akhtar et al. [10] have shown that 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. Conversely, a small value of RAM memory size can lead to higher costs too, due to a longer execution time of the functions. Moreover, a decrease of the marginal improvement in response time as the memory increases is observed.

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

- Lin and Khazaei [14] which proposed a very interesting analytical model to determine both the delay and charged cost for a generic serverless workflow, including a methodology to obtain a configuration for each serverless function belonging to the application, computed exploiting an heuristic approach based on critical path method, in order to respect either a delay or cost constraints imposed by an user.

- Akhtar et al. [10] proposed a framework that, using bayesian optimization to statistically learn the relationship between cost/run-time, selects a serverless function's configuration in order to respect either a delay or cost constraints imposed by an user, minimizing the cost.

- Lin and Khazaei [14] presented, according to our opinion, one of the best analytical performance model available today, which is suitable for analyzing and predict the performance, utilization and cost of current FaaS platforms.

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

For instance, both performance models proposed by Akhtar et al. [10] and Lin and Khazaei [14] do not consider the current state of the FaaS platform when a serverless function configuration has to be selected, totally ignoring the problem of cold starts and their impact on application's performances. Conversely, the model proposed by Lin and Khazaei [14], despite is quite good for analyzing and predict the performance of a FaaS platforms, is not suitable to predict the performance for generic workload applications. Similarly, the Akhtar et al. [10] proposed solution is suitable only for sequence, or chain, of serverless functions, therefore useless for generic workload.

We believe that existing solutions can suffer of the fact that they limit their focus only on serverless function's configurations parameters; in fact, none of existing performance models and frameworks, when it have to select a suitable configuration to fulfill QoS requirements, considers the existing of multiple implementations, or versions, of a same serverless function, that usually exhibit different performances and cost. Sometimes, depending on QoS requirements, can be useful to select a particular implementation respect to another.

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

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

platform can be difficult and costly, making customers more dependent on a single
FaaS solution.

1.3   Contributions

The aim of our work is to overcome aforementioned limitations regarding existing
performance models, building a QoS-aware framework for the orchestration of server-
less functions, belonging to generic workflow serverless application, which supports
multiple FaaS providers and functions implementations.

To be more precise, our work's contributions can be summarized as follows:

- We developed a formal definition of a serverless application, abstracting se-
  quences of serverless functions, including parallel, branch and conditional loop
  structures.

- We formalized a FaaS platform aware performance model in order to evaluate
  and predict the serverless applications performances, in term of economic cost
  and response time.

- We propose a methodological way to guarantee customer's QoS requirements
  developing a software framework capable to find the *"best configuration"* for
  a given serverless application's invocation forwarded by a customer.

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

- We developed a software system compatible with a hybrid FaaS platform ex-
  ecution, in order to exploit different performance and billing methodology to
  better meet customers expectations.

- Owing to the complexity of the problem, since can be very difficult to find an
  optimal configuration for a serverless applications that guarantees the fulfill-
  ment of customer's constraints for very big applications, hence discouraging the
  adoption of serverless paradigm, we have built an heuristic approach based on

a probabilistic technique belonging to *Colony Optimization Algorithm* (ACO) family in order to provide a very rapid system response to our problem.

Moreover, we have developed a custom ACO based algorithm, which we have called *Pre-provisioned Colony Optimization Algorithm with Lazy Pheromone Update*", in order to achieve a faster system response than existing ACO based solutions.

- 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, we have developed our software providing 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.

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

- To further reduce vendor lock-in issue, we have built our own software system acting as orchestrator for serverless application management.

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

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

Chapter 2

Architecture

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

From an architectural point of view, according to the applications taxonomy proposed by Satyanarayanan et al. [16], our software system is a *cloud-native application* because the *Tier-1*, which represents *"the cloud"* in today's parlance, is the unique execution site where our application performs its tasks [1].

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

- The *server process* implementing our service.

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

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

Clearly, for the purposes of our work, our application has to have all required knowledge about the interfaces adopted by all supported providers regarding serverless functions managing; therefore, from an architectural point of view, our system

---

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

requires a set of *adapters*[19], that is software components having aforementioned knowledge. At the current state of this work, only the adapters for following public serverless cloud platforms are provided:

- AWS Lambda [1];

- Apache OpenWhisk [8];

As trivial as it sounds, a very important advantage of this design choice is the *access transparency* provided to final users, because they can manage their applications using identical operations forwarded to our system without interact directly with FaaS providers [19].

Despite it is very easy to implement, the adopted centralized approach also has, obviously, several shortcomings since it is both not scalable and and network traffic inefficient, exacerbating traffic congestion problems, beside from representing a single point of failure [19].

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

2.1   System's components overview

Our framework consists of following main logical entities:

Orchestrator This is the most important software component of our system whose aim to allow the composition of serverless functions in order to properly execute applications, defining a stateful serverless workflows.

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

More concretely, the orchestrator is invoked when the execution of any serverless application is required by an user, passing a JSON input which represent,

obviously, the input data for the invoked application; then, when the execution is ended, it will be generated a JSON output which will be sent to the client.

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

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

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

- *functions branching*, through the support for `if` and `switch` statements.
- *functions looping*, according to which the use of both `for` and `while` statements are supported.

Moreover, the orchestrator is capable to manage *parallel executions*. It is also possible to *abort* current application's execution exploiting the `exit` statement.

Today, all serverless functions orchestration services provided by main public FaaS platforms providers, like *IBM Composer*[7], *Amazon Step Functions*[9], and *Azure Durable Functions*[3], enforce developers to hardcode the specific function implementation to use inside a serverless workflow [15]; this fact prevents the possibility to select a specific function implementation for an optimal execution in order to meet user specified constraints. Similarly, since each FaaS platforms providers does not support the possibility to invoke a serverless function hosted on different FaaS platform, can be difficult to satisfy user specified

QoS levels under some conditions, like high traffic situations towards a specific FaaS platform or resources shortage for functions executions.

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

However, only our QoS-aware model-bases optimizer has the responsibility to determine which implementation, for each function belonging to an application, must be invoked in order to meet QoS level constraints. To be more precise, during function orchestration, the orchestrator limits itself to invoke function implementations according to the so-called *choreography configuration*, or simply *configuration*, produced by the optimizer. The choreography configuration tells to the orchestrator which implementation and configuration must be used for each function in order to meet user specified QoS level constraints; we will talk more in detail about this shortly.

QoS-aware model-bases optimizer As already said, the optimizer has the responsibility to identify, for each serverless function belonging to an application, which configuration and implementation must be used in order to satisfy user specified constraints.

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

- if the user does not specify any QoS level constraints, the optimizer will produce the choreography configuration by select configurations and implementations randomly.

- if QoS level constraints are specified, the optimizer firstly retrieving, interacting mostly with our logging subsytem, all needed metrics to perform its task, such as observed average response times and charged cost for each implementation and configuration for every function inside the invoked applications; moreover, the optimizer will also checks both the status for

14

each FaaS platforms and the actual users requests traffics on the system. We will precise what metrics are effectively used by the optimizer in next chapter.

When all needed data are collected, the optimizer provides a suitable configuration of each serverless function within an application, by solving an optimization problem, derived from our performance model.

When a configuration is finally produced, the orchestrator will be subsequently invoked with it to actually perform the execution of the requested application.

2.1.0.1 Logging subsytem

In order that our optimizer performs its tasks, a live overview of the state of both all FaaS platforms executions environment and the system is, obviously, required. Moreover, historical information regarding serverless function performances under any possible configuration is strictly needed to build a suitable configuration to meet any user-specified constraints.

Fr

As the Logging Framework receives logs from each of its sources, it computes running metrics such as total con- currency of specic functions and chains, number of drops observed in the serverless framework, and inter-arrival times of function chains tracked by the Producer. These metrics are available to be read at any time by the Policy Framework and can be exported to long-term storage to be available for post-hoc analysis.

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.

In fact, besides parameters configuration, serverless functions performance is affected by several other factors like traffic, *cold starts* and network conditions.

2.2 Architectural style

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

2.2.1 Resource identification

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

Firstly, our system supports only two resources:

- *Resource owners*, which are at the top of the resources hierarchy.

- *Serverless applications*, which, instead, are at the bottom.

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

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

- the name of the serverless applications owner, that is the resource owner name.

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

For instance, a URI such as the one reported in listing 2.1 denotes the serverless application called "*application*" owned by the resource owner called "*andrea*".

Listing 2.1: A serverless application URI example

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

2.2.2 Operations on resources

As known `HTTP` protocol provides several methods to interact with resources. Constraining operations on resources only to `HTTP` provided methods is one of the keys to enable loose-coupling between our software and clients, as they only need to support `HTTP` protocol to use our framework[12].

Moreover, as known, when an operation on a specified resource is invoked, `HTTP` protocol offers a way of expressing errors and exceptions, which are represented by standardized status codes sent back as part of the header in the `HTTP` response message. Naturally, we have adopt them to report any issue occurring during any operation performed on any resources because, since `HTTP` status codes have well-known meanings for HTTP clients, we can achieve a lower coupling between our software and clients [12][5].

According to our design, following operations are supported:

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

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

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

- If our framework encountered an unexpected condition which prevented it from fulfilling the serverless application creation request, the response code will be `500` (Internal Server Error). Moreover, a JSON formatted data, describing occurred error, will be sent as part of the `HTTP` response message.

- If our framework detects that a serverless application called `app1` and owned by `genericOwner` already exists, the status code `409` (Conflict) will be sent.

- If the JSON's representation of the serverless application to create contains syntax errors, the code `400` (Bad Request) will be sent. That status code, will be sent also when a malformed URI is submitted, that is when the user `genericOwner` does not exist.

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

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

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

One of the following event can occurs:

- If no error occurred during serverless application's execution, an HTTP response message, having status code `200` (OK) with a JSON represen-

tation of the data produced as output by the invoked application, will be
sent to the client.

- If specified URI refers to an inexistent serverless application, naturally,
  the status code `404` (Not Found) will be sent.

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

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

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

`DELETE` is used to delete a resource.

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

- `200` when submitted operation succeeded without errors

- `410` (Gone), if specified URI refers to an inexistent or already deleted
  resource owner.

- `500` when any error occurred. In that case, the response message will
  include a JSON formatted data describing occurred exception.

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

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

To be more precise, if a `POST` is submitted on a serverless application whose
URI is "`/genericOwner/app1`", the profiling task, regarding the application
called `app1`, will be started. We will talk about profiling task for serverless

19

applications very soon. In this case, one of the following status codes will be sent:

- `201` when submitted operation succeeded without errors

- `404` if specified URI refers to an inexistent serverless application.

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

Conversely, when a `POST` operation is submitted on "`/genericOwner`", with a JSON data including FaaS platforms credentials, the latter will be stored on our system letting it to invoke any serverless functions on FaaS platform.

2.3 Communication's features

Unfortunately, clients are obliged to communicate directly with our application running on the cloud using a classical *client-server style communication*, which use classic *point-to-point* and *synchronous* communications, according to which users wait until their requests has been fully processed by our system [19].

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

- *temporally* coupled, which means that both our server and our clients processes that are communicating will both have to be up and running [19];

- *referentially* coupled, since explicit referencing in communication is adopted [19];

Since all currently supported operations are factually idempotent, that is they can be performed repeatedly with the same effect as if they had been performed exactly once[11], an *at-least-once* communication's semantics is safely adopted; therefore, any client receives either a result, in which case the invoker knows that the procedure was executed at least once, or an exception. As known, aforementioned

communication semantics is achieved exploiting the retransmission of request messages, which masks so-called omission failures[2]. Unfortunately, our system cannot manage neither crash or arbitrary failures[3] [11].

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

---

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

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

Chapter 3

Model's formalization

3.1 SLA's definition

As said previously, our system's goal is the fulfillment of both *functional require-ments*, 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 invo-cation 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 con-ditions 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 re-gards 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:

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

  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:

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

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

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

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

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:

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

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. We will describe very soon how to compute it.

- Let $i, j \in \mathbb{N} \cap [1, n]$ and $\phi_i, \phi_j \in \Phi$, any directed edge $e_{ij} = (\phi_i, \phi_j) \in E$ represents the calling relationship between two abstract functions, which depends

---

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

on the business logic defined by R.

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

- Let $i, j \in \mathbb{N} \cap [1, n]$, the number $p_{ij} \in \mathbb{R}^+ \cap [0, 1]$ is the weight assigned to the edge $(\phi_i, \phi_j) \in E$, where:

  - The number p_{ij} represents the so-called *transition probability* from ϕ_i to ϕ_j, that is the probability of invoking ϕ_j when the execution of ϕ_i is terminated;

  - $P : \Phi \times \Phi \times \mathbb{R}^+ \to \mathbb{R}^+ \cap [0, 1]$, called *transition probability function*, is such that $P(\phi_i, \phi_j, t) = p_{ij}$, where t is the time. Obviously, $P(\phi_i, \phi_j, t) = 0$ 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 $\pi = \phi_1 e_1 \ldots e_{n-1} \phi_n$ 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:

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

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:

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

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:

$$
\begin{array}{llll}
|succ(\phi)| & = & n & \forall \phi \in \mathcal{F}_{\mathcal{E}}(\mathcal{C}) \wedge n \in \{0,1\} & (3.17) \\
P(\phi, \phi_x, t) & = & 1 & \forall \phi, \phi_x \in \mathcal{F}_{\mathcal{E}}(\mathcal{C}) : \phi_x \in succ(\phi) \wedge |succ(\phi)| = 1, \forall t & (3.18) \\
P_{exit}(\phi, t) & = & 0 & \forall \phi \in \mathcal{F}_{\mathcal{E}}(\mathcal{C}), \forall t & (3.19)
\end{array}
$$

- 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.

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

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

Moreover, each orchestration function $\phi \in \mathcal{F}_\mathbb{O}(\mathcal{C})$ 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 $\phi \in \mathcal{F}_\mathbb{O}(\mathcal{C})$ 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:

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

$$P(\phi, \phi_x, t) = 1 \qquad \forall \phi, \phi_x \in \mathcal{F}_\mathbb{O}(\mathcal{C}) : \qquad (3.22)$$
$$\mathcal{T}(\phi) = \tau_\omega \wedge \phi_x \in succ(\phi) \wedge |succ(\phi)| = 1, \forall t$$

Clearly, based on above definitions, we can say:

$$\mathcal{F}_{\mathcal{E}}(\mathcal{C}) \cap \mathcal{F}_{\mathcal{O}}(\mathcal{C}) \;\; = \;\; \emptyset \tag{3.23}$$

$$\mathcal{F}_{\mathcal{E}}(\mathcal{C}) \cup \mathcal{F}_{\mathcal{O}}(\mathcal{C}) \;\; = \;\; \Phi \tag{3.24}$$

$$|\mathcal{F}_{\mathcal{E}}(\mathcal{C})| + |\mathcal{F}_{\mathcal{O}}(\mathcal{C})| \;\; = \;\; |\Phi| \tag{3.25}$$

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:

$$\mathcal{C} \text{ is executable} \;\; \Leftrightarrow \;\; |\mathbf{F}_\phi| \geq 1 \qquad \forall \phi \in \mathcal{F}_{\mathcal{E}}(\mathcal{C}) \tag{3.26}$$

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:

$$\mathcal{C}^* \stackrel{def}{=} (\Phi^*, E^*) \qquad \text{where } \Phi^* \subseteq \Phi \wedge E^* \subseteq E \tag{3.27}$$

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:

$$
\begin{align}
|in(\phi)| &= 1 & \forall \phi \in \Phi \setminus \{\alpha(\mathcal{C})\} \tag{3.28} \\
|out(\phi)| &= 1 & \forall \phi \in \Phi \setminus \{\omega(\mathcal{C})\} \tag{3.29} \\
P(\phi_x, \phi_y, t) &= 1 & \forall x, y \in \mathbb{N} \cap [1, |\Phi|], \forall t \tag{3.30}
\end{align}
$$

Any choreography $\mathcal{C}$ satisfying 3.28, 3.29 and 3.30 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:

$$x_{\phi} = (f_{\phi}, m) \in f_{\phi} \times \mathbf{M}_{f_{\phi}} \subseteq \mathbf{F}_{\phi} \times \mathbb{N} \tag{3.31}$$

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:

$$
\begin{aligned}
\mathbf{X}_{\mathcal{C}} &\stackrel{def}{=} \{x_{\phi_1}, \ldots, x_{\phi_k}\} \\
&\in \left\{ \left\{ \bigcup_{j=1}^{|\mathbf{F}_{\phi_1}|} f_{\phi_{1_j}} \times \mathbf{M}_{f_{\phi_{1_j}}} \right\} \times \ldots \times \left\{ \bigcup_{j=1}^{|\mathbf{F}_{\phi_k}|} f_{\phi_{k_j}} \times \mathbf{M}_{f_{\phi_{k_j}}} \right\} \right\} \\
&= \bigtimes_{i=1}^{k} \left\{ \bigcup_{j=1}^{|\mathbf{F}_{\phi_i}|} f_{\phi_{i_j}} \times \mathbf{M}_{f_{\phi_{i_j}}} \right\} \\
&\subseteq \bigtimes_{i=1}^{k} \{\mathbf{F}_{\phi_i} \times \mathbb{N}\} = \mathbf{X}_{\mathcal{C}}
\end{aligned}
\tag{3.32}
$$

where:

- x_{ϕ_i} represents the executable function configuration for the executable function ϕ_i, for some $i \in \mathbb{N} \cap [1, k]$.

- $f_{\phi_{i_j}}$ represent the j-th concrete function implementing the executable function ϕ_i, for some $i \in \mathbb{N} \cap [1, k]$ and $j \in \mathbb{N} \cap [1, |\mathbf{F}_{\phi_i}|]$.

- $\mathbf{M}_{f_{\phi_{i_j}}} \subseteq \mathbb{N}$ denotes the set containing all available memory size options, allowed by provider, during the execution of the concrete function $f_{\phi_{i_j}}$, for some $i \in \mathbb{N} \cap [1, k]$ and $j \in \mathbb{N} \cap [1, |\mathbf{F}_{\phi_i}|]$

- $\mathbf{X}_{\mathcal{C}}$ represents the so-called *solutions space*, which includes all possible choreography configurations.

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

3.4   Serverless choreography's structure

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

Informally, a *structure* is defined as any sub-choreography $\mathcal{C}^* = (\Phi^*, E^*, \Theta_{\mathcal{C}^*})$ of $\mathcal{C}$ whose entry point $\alpha(\mathcal{C}^*)$ and the end point $\omega(\mathcal{C}^*)$ are, respectively, opening and closing orchestration functions having same type τ possibly coinciding. Formally:

$$\mathcal{C}^* \text{ is a structure } \Leftrightarrow \begin{cases} \alpha(\mathcal{C}^*), \omega(\mathcal{C}^*) \in \mathcal{F}_0(\mathcal{C}^*) \\ \mathcal{T}(\alpha(\mathcal{C}^*)) = \tau_\alpha \\ \mathcal{T}(\omega(\mathcal{C}^*)) = \tau_\omega \end{cases} \tag{3.33}$$

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:

$$\begin{aligned} \Phi^{**} &\stackrel{def}{=} \Phi^* \setminus \{\alpha(\mathcal{C}^*), \omega(\mathcal{C}^*)\} \\ E^{**} &\stackrel{def}{=} E^* \setminus \left[out\left(\alpha(\mathcal{C}^*)\right) \cup in\left(\omega(\mathcal{C}^*)\right) \right] \end{aligned} \tag{3.34}$$

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:

$$
\Theta_{\mathcal{C}^*} \stackrel{def}{=} \{\theta_1, \ldots, \theta_c\} \tag{3.35}
$$

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

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

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

$$
\Phi' = \{\Phi \setminus \Phi^*\} \cup \{\phi_{fake}\} \tag{3.36}
$$

$$
E' = \{E \setminus E^*\} \cup \left\{ \left(prev(\alpha(\mathcal{C}^*)), \phi_{fake} \right) \right\} \cup \left\{ \left(\phi_{fake}, succ(\omega(\mathcal{C}^*)) \right) \right\} \tag{3.37}
$$

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

$$
RT_{\phi_{fake}}(\mathbf{x}_{\mathcal{C}}, t) = RT(\mathcal{C}^*, \mathbf{x}_{\mathcal{C}}, t) \tag{3.38}
$$

$$
C_{\phi_{fake}}(\mathbf{x}_{\mathcal{C}}, t) = C(\mathcal{C}^*, \mathbf{x}_{\mathcal{C}}, t) \tag{3.39}
$$

$$
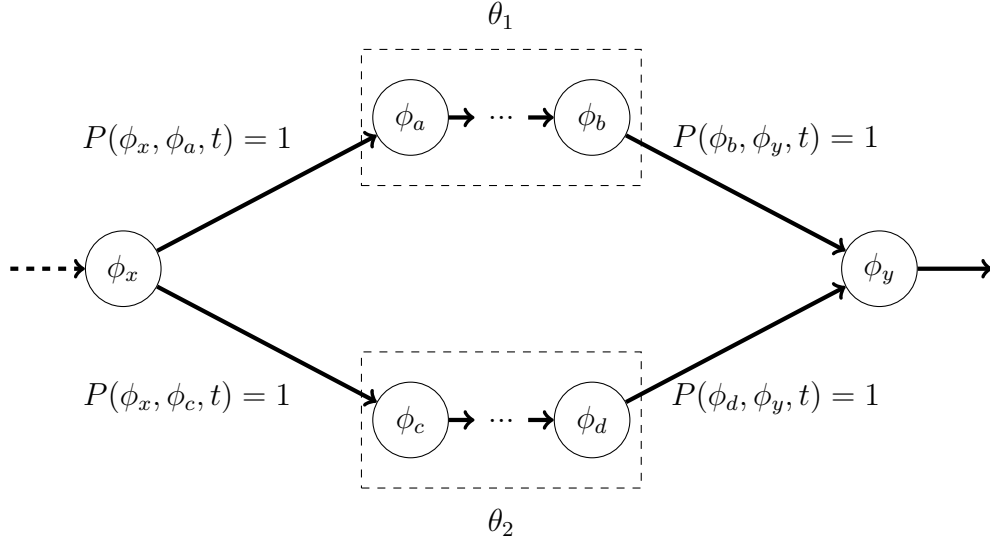P_{exit}(\phi_{fake}, t) = P_{exit}(\mathcal{C}^*, t) \tag{3.40}
$$

Figure 3.1: Parallel structure in a serverless workflow.

3.4.1 Parallel

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

$$\mathcal{T}(\alpha(\mathcal{P})) = \mathcal{T}(\phi_x) = \tau_\alpha \tag{3.41}$$

$$\mathcal{T}(\omega(\mathcal{P})) = \mathcal{T}(\phi_y) = \tau_\omega \tag{3.42}$$

$$TPP(\pi, t) = 1 \qquad \forall \pi \in \Pi(\phi_x, \phi_y), \forall t \tag{3.43}$$

$$|\Theta| = n \qquad n \in \mathbb{N} \setminus \{0\} \tag{3.44}$$

$$E[I_\theta] = 1 \qquad \forall \theta \in \Theta \tag{3.45}$$

$$P_{exit}(\theta, t) = 0 \qquad \forall \theta \in \Theta \tag{3.46}$$

$$|succ(\alpha(\mathcal{P}))| = |prev(\omega(\mathcal{P}))| \tag{3.47}$$

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:

$$RT(\mathcal{P}, \mathbf{x}_{\mathcal{C}}, t) \overset{def}{=} max \left\{ RT(\theta_i, \mathbf{x}_{\mathcal{C}}, t) \mid \theta \in \Theta \right\} \tag{3.48}$$

$$C(\mathcal{P}, \mathbf{x}_{\mathcal{C}}, t) \overset{def}{=} \sum_{i=1}^{n} C(\theta_i, \mathbf{x}_{\mathcal{C}}, t) \tag{3.49}$$

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:

$$P_{exit}(\mathcal{P}, t) = 0 \tag{3.50}$$

3.4.2 Branch

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

$$\mathcal{T}(\alpha(\mathcal{B})) = \mathcal{T}(\phi_x) = \tau_\alpha \tag{3.51}$$

$$\mathcal{T}(\omega(\mathcal{B})) = \mathcal{T}(\phi_y) = \tau_\omega \tag{3.52}$$

$$|\Theta| = n \qquad n \in \mathbb{N} \setminus \{0\} \tag{3.53}$$

$$P(\phi_x, \alpha(\theta_i), t) \leq 1 \qquad \forall i \in \mathbb{N} \cap [1, n] \tag{3.54}$$

$$\sum_{i=1}^{n} P(\phi_x, \alpha(\theta_i), t) = 1 \tag{3.55}$$

$$E[I_\theta] = 1 \qquad \forall \theta \in \Theta \tag{3.56}$$

$$|succ(\alpha(\mathcal{B}))| = |prev(\omega(\mathcal{B}))| \tag{3.57}$$

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.
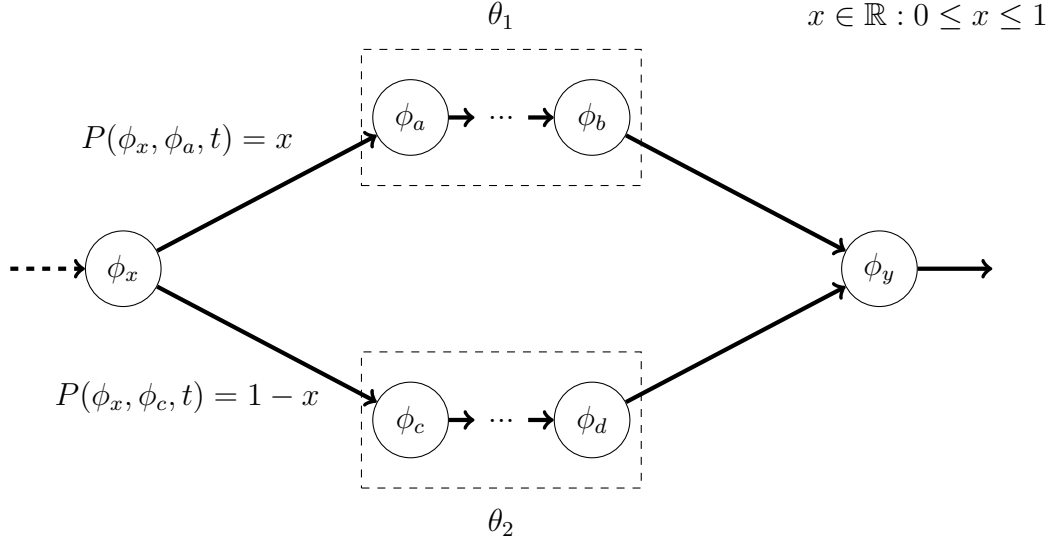
Figure 3.2: An if-else-branch structure in a serverless workflow

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

$$P(\phi_x, \alpha(\theta), t) = \frac{|I_\theta(t)|}{|I_\mathcal{B}(t)|} \qquad \forall \theta \in \Theta \tag{3.58}$$

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:

$$RT(\mathcal{B}, \mathbf{x}_\mathcal{C}, t) \overset{def}{=} \sum_{i=1}^{n} P(\phi_x, \alpha(\theta_i), t) \cdot RT(\theta_i, \mathbf{x}_\mathcal{C}, t) \tag{3.59}$$

$$C(\mathcal{B}, \mathbf{x}_\mathcal{C}, t) \overset{def}{=} \sum_{i=1}^{n} P(\phi_x, \alpha(\theta_i), t) \cdot C(\theta_i, \mathbf{x}_\mathcal{C}, t) \tag{3.60}$$

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

$$P_{exit}(\mathcal{B}, t) \stackrel{def}{=} \sum_{i=1}^{n} P(\phi_x, \alpha(\theta_i), t) \cdot P_{exit}(\theta_i, t) \tag{3.61}$$

3.4.3 Conditional loop

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

$$\mathcal{T}(\alpha(\mathcal{L})) = \mathcal{T}(\phi_x) = \tau_\alpha \tag{3.62}$$

$$\mathcal{T}(\omega(\mathcal{L})) = \mathcal{T}(\phi_y) = \tau_\omega \tag{3.63}$$

$$|\Theta| = 1 \tag{3.64}$$

$$E[I_\theta] \geq 0 \qquad \theta \in \Theta \tag{3.65}$$

$$P(\phi_x, \alpha(\theta), t) = x \tag{3.66}$$

$$P(\phi_x, \phi_y, t) = 1 - x \tag{3.67}$$

$$P(\omega(\theta), \phi_x, t) = 1 \qquad \theta \in \Theta \tag{3.68}$$

$$P(\phi_x, \alpha(\theta), t) + P(\phi_x, \phi_y, t) = 1 \qquad \theta \in \Theta \tag{3.69}$$

$$\tag{3.70}$$

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 3.58 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:

$$\theta \qquad\qquad x \in \mathbb{R} : 0 \le x \le 1$$
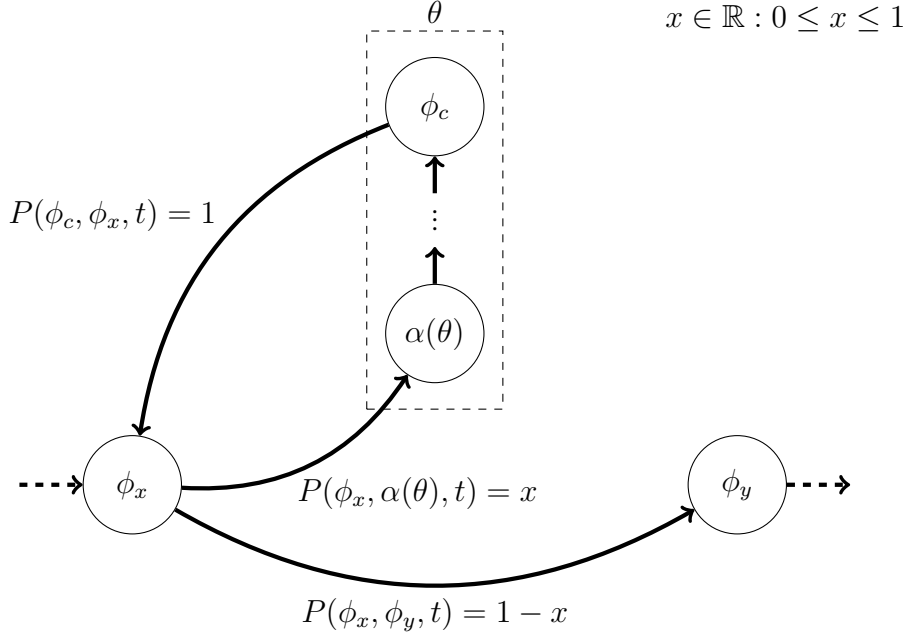
Figure 3.3: Conditional loop structure in a serverless workflow.

$$p = P(\phi_x, \phi_y, t) = 1 - P\Big(\phi_x, \alpha(\theta), t\Big) \tag{3.71}$$

$$\tag{3.72}$$

Then:

$$
\begin{aligned}
P(I_\theta = k) &= p \cdot (1 - p)^{k-1} & \text{(3.73)} \\
&= pq^{k-1} & \text{(3.74)} \\
&= \Big[1 - P\Big(\phi_x, \alpha(\theta), t\Big)\Big] \cdot \Big[P\Big(\phi_x, \alpha(\theta), t\Big)\Big]^{k-1} & \text{(3.75)} \\
& & \text{(3.76)}
\end{aligned}
$$

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:

$$
\begin{aligned}
E[I_\theta(t)] &= \sum_{k=1}^{\infty}(k-1)pq^{k-1} \\
&= p\sum_{k=0}^{\infty}kq^k \\
&= p\cdot\frac{q}{(1-q)^2} \\
&= \frac{q}{p} \\
&= \frac{P\Big(\phi_x,\alpha(\theta),t\Big)}{1-P\Big(\phi_x,\alpha(\theta),t\Big)}
\end{aligned}
\tag{3.77}
$$

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

$$
RT(\mathcal{L},\mathbf{x}_\mathcal{C},t) \stackrel{def}{=} E[I_\theta(t)]\cdot RT(\theta_i,\mathbf{x}_\mathcal{C},t)
\tag{3.78}
$$

$$
C(\mathcal{L},\mathbf{x}_\mathcal{C},t) \stackrel{def}{=} E[I_\theta(t)]\cdot C(\theta_i,\mathbf{x}_\mathcal{C},t)
\tag{3.79}
$$

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:

$$
P_{exit}(\mathcal{L},t) \stackrel{def}{=} \Big(1-P_{exit}(\theta,t)\Big)^{\lfloor E[I_\theta(t)]\rfloor-1}\cdot P_{exit}(\theta,t)
\tag{3.80}
$$

$$
\tag{3.81}
$$

3.4.4 Exit

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

$$\mathfrak{T}(\alpha(\mathcal{L})) = \mathfrak{T}(\phi_x) = \tau_\alpha \tag{3.82}$$

$$\mathfrak{T}(\omega(\mathcal{L})) = \mathfrak{T}(\phi_y) = \tau_\omega \tag{3.83}$$

$$|\Theta| = 0 \tag{3.84}$$

$$P(\phi_x, \phi_y, t) = 0 \qquad \forall t \tag{3.85}$$

$$\tag{3.86}$$

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.
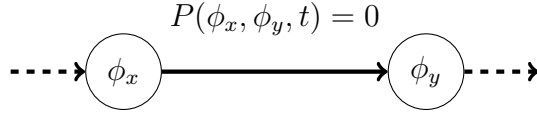
Figure 3.4: An exit structure in a serverless workflow.

At any time t, following is true that:

$$RT(\mathcal{E}, \mathbf{x}_\mathcal{C}, t) = 0 \tag{3.87}$$

$$C(\mathcal{E}, \mathbf{x}_\mathcal{C}, t) = 0 \tag{3.88}$$

$$P_{exit}(\mathcal{E}, t) = 1 \tag{3.89}$$

3.5 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.

3.5.1   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.

3.5.2 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.

3.5.3 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.

3.5.4 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]

---

[2]Despite there is no explicitly mentioned global concurrency limit, previous studies have ob-

3.5.5 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.

  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:

$$0 \leq K(t) \leq C_{\mathbf{max}} \tag{3.90}$$

- 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.

---

served a kind of global concurrency limit varying between 1000 and 2000.

[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.

Warm Pool having capacity n

λ

Blocking due to lack of
idle function instances

Request dropped if maximum
concurrency limit is reached

Instance added to
warm pool increasing
its capacity by one and
causing a cold start.

f_1

f_2

$\ldots$

f_n

f_{n+1}
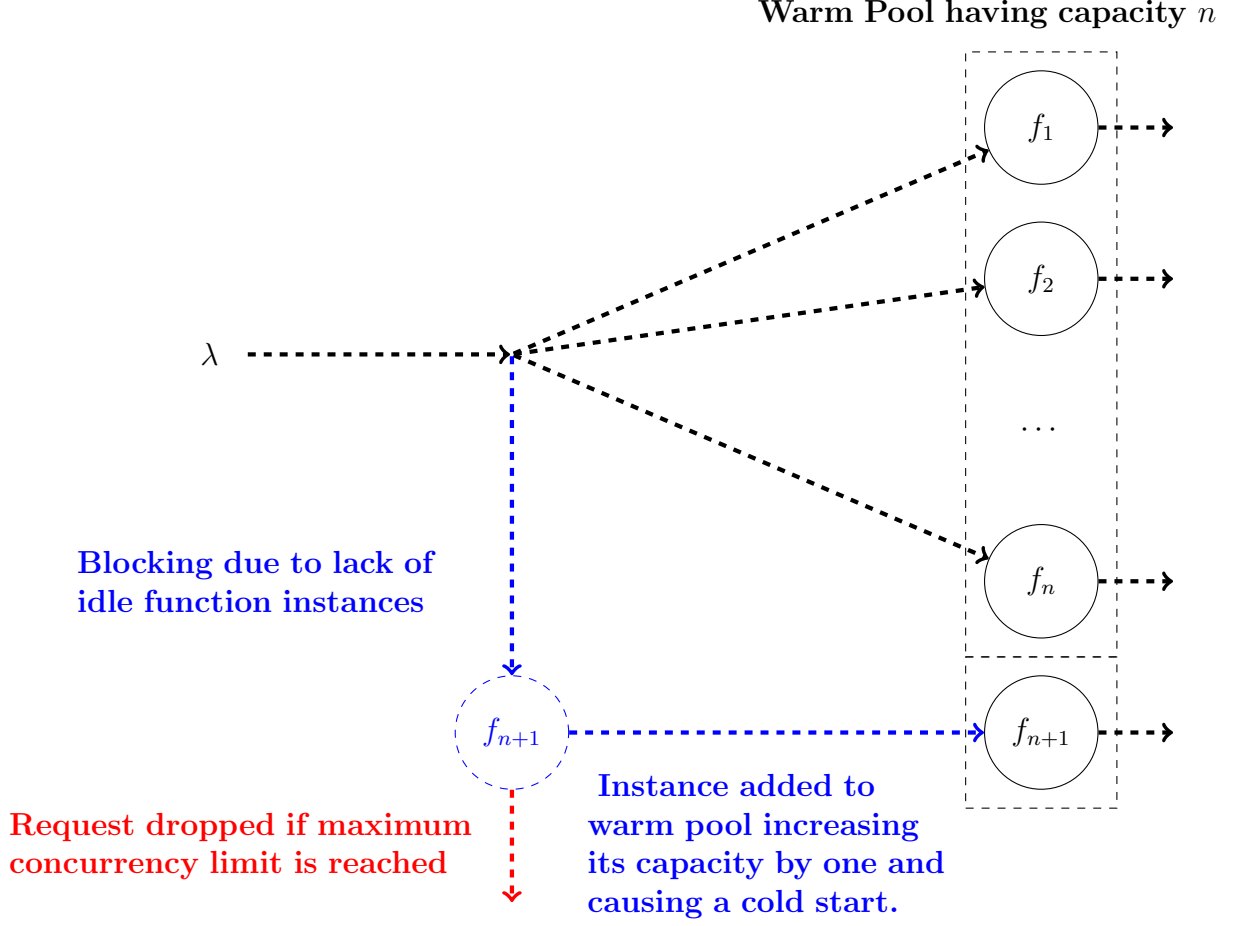
Figure 3.5: Parallel structure in the serverless workflow.

3.5.6 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.

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.

3.5.6.1 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:

$$\mathbf{P}_{\omega_{P_R}^{(l)}}(t) = \frac{\rho(t)^{K(t)}}{K(t)!} \cdot \left(\sum_{j=0}^{K(t)} \frac{\rho(t)^j}{j!} \right)^{-1} \quad (3.91)$$

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:

$$\lambda(t) \overset{def}{=} \begin{cases} Y_o & \text{if } t = 0 \\ \alpha \cdot Y_t + (1 - \alpha) \cdot \lambda(t-1) & \text{if } t > 0 \end{cases}. \quad (3.92)$$

  - $\mu_w = E[S_w]$ 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.

3.5.6.2 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:

$$\widetilde{\mathbf{S}}_{\mathcal{C}} \stackrel{def}{=} \left\{ \omega_P^{(l)} \in \bigcup_{i=1}^{n} \Omega_{P_i} : \exists f_\phi \in \omega_P^{(l)} \text{ such that } f_\phi \in \mathbf{F}_\phi, \forall \phi \in \mathcal{F}_{\mathcal{E}}(\mathcal{C}) \right\} \quad (3.93)$$

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

$$\widetilde{\mathbf{S}}_{\mathcal{C}_1} \cap \widetilde{\mathbf{S}}_{\mathcal{C}_2} \neq \emptyset \quad (3.94)$$

that is a swarm can be shared by multiple choreographies, therefore any implementation of our model has to pay attention on eventually race conditions.

Finally, we will use following function:

$$N(\mathcal{C}, \mathbf{x}_{\mathcal{C}}, \omega_P^{(l)}) = \sum_{x_\phi \in \mathbf{x}_{\mathcal{C}}} \sum_{\substack{f_\phi \in x_\phi \\ f_\phi \in \omega_P^{(l)}}} 1 \quad (3.95)$$

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

3.5.7 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:

$$
\begin{aligned}
C(x_\phi, t) \;\overset{def}{=}\; & C(f_\phi, m, t) \\
= \; & C_{avg}^{(c)}(f_\phi, m, t) \cdot \mathbf{P}_{\omega_{P_R}^{(l)}}(t) + \\
& C_{avg}^{(w)}(f_\phi, m, t) \cdot \left(1 - \mathbf{P}_{\omega_{P_R}^{(l)}}(t)\right)
\end{aligned}
\tag{3.96}
$$

  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, $C_{avg}^{(w)}(f_\phi, m, t)$ 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:

$$
\begin{aligned}
RT(x_\phi, t) \;\overset{def}{=}\; & RT(f_\phi, m, t) \\
= \; & RT_{avg}^{(c)}(f_\phi, m, t) \cdot \mathbf{P}_{\omega_{P_R}^{(l)}}(t) + \\
& RT_{avg}^{(w)}(f_\phi, m, t) \cdot \left(1 - \mathbf{P}_{\omega_{P_R}^{(l)}}(t)\right)
\end{aligned}
\tag{3.97}
$$

  where:

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

Obliviously, our framework's duty is to determine $RT_{avg}^{(c)}(f_\phi, m, t)$, $RT_{avg}^{(w)}(f_\phi, m, t)$, $C_{avg}^{(c)}(f_\phi, m, t)$ and $C_{avg}^{(w)}(f_\phi, m, t)$ 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 $RT_{avg}^{(c)}(f_\phi, m, t)$, 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:

$$RT_{avg}^{(c)}(f_\phi, m, t) \stackrel{def}{=} \begin{cases} Y_o & \text{if } t = 0 \\ \alpha \cdot Y_t + (1 - \alpha) \cdot \lambda(t - 1) & \text{if } t > 0 \end{cases}. \tag{3.98}$$

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

$$\phi_i \in \mathcal{F}_\mathcal{E}(\theta_1^{(\phi_i)}) \tag{3.99}$$

$$\theta_n^{(\phi_i)} = \mathcal{C} \tag{3.100}$$

$$\theta_k^{(\phi_i)} = (\Phi_k, E_k) \text{ is a sub-choreography of } \theta_{k+1}^{(\phi_i)} \qquad \forall k \in [1; n-1] \quad (3.101)$$

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

$$E[I_{\phi_i}(t)] \stackrel{def}{=} \begin{cases} 1 & if \quad n = 1 \\ \prod_{k=2}^{n} E[I_{\theta_{k-1}^{(\phi_i)}}(t)] & if \quad n \geq 2 \end{cases} \quad (3.102)$$

- $P_{exe}^C(\theta_k^{(\phi_i)}, t)$ is the probability to execute the entry point $\alpha(\theta_k^{(\phi_i)})$ of the choreography $\theta_k^{(\phi_i)}$ when the function $pred(\alpha(\theta_k^{(\phi_i)}))$ is executed at time t. 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:

$$P_{exe}^C(\theta_k^{(\phi_i)}, t) \stackrel{def}{=} \begin{cases} 1 & if \quad n = 1 \\ P\left(pred\left(\alpha(\theta_k^{(\phi_i)})\right), \alpha(\theta_k^{(\phi_i)})\right) & if \quad n \geq 2 \end{cases} \quad (3.103)$$

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

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

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

$$succ(\phi_x) = \phi_{x+1} \qquad x \in \mathbb{N} \cap [1, n-1] \quad (3.104)$$
$$\phi_n = \phi \quad (3.105)$$

  Finally:

53

$$P_{exe}^{F}(\theta_k^{(\phi_i)}, \phi, t) \stackrel{def}{=} \prod_{i=1}^{n-1} (1 - P_{exit}(\phi_i, t)) \tag{3.106}$$

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

$$\Gamma_{\theta_k^{(\phi_i)}}(t) \stackrel{def}{=} P_{exe}^{C}(\theta_k^{(\phi_i)}, t) \cdot \left(\prod_{y=k+1}^{n} P_{exe}^{C}(\theta_y^{(\phi_i)}, t) \cdot P_{exe}^{F}(\theta_y^{(\phi_i)}, pred(\alpha(\theta_{y-1}^{(\phi_i)})), t) \right) \tag{3.107}$$

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

$$c_{\phi_{i_j}}(t) \stackrel{def}{=} \begin{cases} C(x_{\phi_{i_j}}, t) \cdot P_{exe}^{F}(\theta_1^{(\phi_i)}, \phi_i, t) & \text{if } m = 1 \\ C(x_{\phi_{i_j}}, t) \cdot P_{exe}^{F}(\theta_1^{(\phi_i)}, \phi_i, t) \cdot E[I_{\phi_i}(t)] \cdot \Gamma_{\theta_1^{(\phi_i)}}(t) & \text{if } m \geq 2 \end{cases} \tag{3.108}$$

$$rt_{\phi_{i_j}}(t) \stackrel{def}{=} \begin{cases} RT(x_{\phi_{i_j}}, t) \cdot P_{exe}^{F}(\theta_1^{(\phi_i)}, \phi_i, t) & \text{if } m = 1 \\ RT(x_{\phi_{i_j}}, t) \cdot P_{exe}^{F}(\theta_1^{(\phi_i)}, \phi_i, t) \cdot E[I_{\phi_i}(t)] \cdot \Gamma_{\theta_1^{(\phi_i)}}(t) & \text{if } m \geq 2 \end{cases} \tag{3.109}$$

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;

$$p_{\phi_{i_j}}(t) \stackrel{def}{=} w_{RT} \cdot \frac{t_{\phi_{i\textbf{MAX}}}(t) - rt_{\phi_{i_j}}(t)}{t_{\phi_{i\textbf{MAX}}}(t) - rt_{\phi_{i\textbf{MIN}}}(t)} + w_C \cdot \frac{c_{\phi_{i\textbf{MAX}}}(t) - c_{\phi_{i_j}}(t)}{c_{\phi_{i\textbf{MAX}}}(t) - c_{\phi_{i\textbf{MIN}}}(t)} \tag{3.110}$$

where:

- $rt_{\phi_{i\textbf{MIN}}}(t)$ and $rt_{\phi_{i\textbf{MAX}}}(t)$ 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.

3.7   Choreography's performance evaluation

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

3.7.1   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. Fortunately, exploiting pipeline choreography's properties, is very simple to obtain a performance evaluation when $\mathbf{x}_{\mathcal{C}}$ is given.

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

$$succ(\phi_x) = \phi_{x+1} \qquad x \in \mathbb{N} \cap [1, n-1] \tag{3.111}$$

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

$$RT(\mathcal{C}, \mathbf{x}_{\mathcal{C}}, t) \stackrel{def}{=} RT(x_{\phi_1}, t) + \sum_{x=2}^{n} P_{exe}^{F}(\mathcal{C}, \phi_{x-1}, t) \cdot RT(x_{\phi_x}, t) \tag{3.112}$$

$$C(\mathcal{C}, \mathbf{x}_{\mathcal{C}}, t) \stackrel{def}{=} C(x_{\phi_1}, t) + \sum_{x=2}^{n} P_{exe}^{F}(\mathcal{C}, \phi_{x-1}, t) \cdot C(x_{\phi_x}, t) \tag{3.113}$$

3.7.2 Generic choreography's performance

Unfortunately, equations 3.112 and 3.113 can be used only dealing with pipeline type choreographies, therefore they are not suitable for general choreographies having branch or loops.

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

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

---

1   $\mathcal{C}^* \leftarrow \mathcal{C}$;
2   **while** $\mathcal{C}^*$ *is not a pipeline choreography* **do**
3   branchList $\leftarrow$ find_branches($\mathcal{C}^*$);
4   **for** $\mathcal{B}$ *in branchList* **do**
5         $\mathcal{C}^* \leftarrow$ process($\mathcal{B}$);
6   **end**
7   parallelList $\leftarrow$ find_parallels($\mathcal{C}^*$);
8   **for** $\mathcal{P}$ *in parallelList* **do**
9         $\mathcal{C}^* \leftarrow$ process($\mathcal{P}$);
10   **end**
11   loopList $\leftarrow$ find_loops($\mathcal{C}^*$);
12   **for** $\mathcal{L}$ *in loopList* **do**
13         $\mathcal{C}^* \leftarrow$ process($\mathcal{L}$);
14   **end**
15   exitList $\leftarrow$ find_exits($\mathcal{C}^*$);
16   **for** $\mathcal{E}$ *in exitList* **do**
17         $\mathcal{C}^* \leftarrow$ process($\mathcal{E}$);
18   **end**
19 **end**
20 **return** $\mathcal{C}^*$

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

Formally, let $\mathcal{C} = (\Phi, E)$ a serverless choreography, $\mathbf{x}_{\mathcal{C}} \in \mathbf{X}_{\mathcal{C}}$ a choreography configuration and $\langle (RT, w_{RT}), (C, w_C) \rangle$ the SLA.

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

$$S(\mathbf{x}_\mathcal{C}, t) \stackrel{def}{=} w_{RT} \cdot \frac{RT_{\mathbf{max}}(t) - RT(\mathbf{X}, t)}{RT_{\mathbf{max}}(t) - RT_{\mathbf{min}}(t)} + w_C \cdot \frac{C_{\mathbf{max}}(t) - C(\mathbf{X}, t)}{C_{\mathbf{max}}(t) - C_{\mathbf{min}}(t)} \qquad (3.114)$$

where:

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

3.8 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.

- 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.

3.8.1 Multidimensional Knapsack Problem Formulation

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

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

$$max \quad \sum_{i=1}^{|\mathbf{X}_{\mathcal{C}}|} x_i \cdot S(\mathbf{x}_{\mathcal{C}_i}, t) \tag{3.115}$$

$$\text{subject to} \quad \sum_{\omega=1}^{|\mathbf{X}_{\mathcal{C}}|} x_i \cdot C(\mathcal{C}, \mathbf{x}_{\mathcal{C}_i}, t) \leq C \tag{3.116}$$

$$\sum_{i=1}^{|\mathbf{X}_{\mathcal{C}}|} x_i \cdot RT(\mathcal{C}, \mathbf{x}_{\mathcal{C}}, t) \leq RT \tag{3.117}$$

$$\sum_{i=1}^{|\mathbf{X}_{\mathcal{C}}|} x_i \cdot N(\mathcal{C}, \mathbf{x}_{\mathcal{C}}, \omega_P^{(l)}) \leq l - R(\mathbf{Q}_{\omega_P^{(l)}}, t) \qquad \forall \omega_P^{(l)} \in \widetilde{\mathbf{S}}_{\mathcal{C}} \tag{3.118}$$

$$\sum_{i=1}^{|\mathbf{X}_{\mathcal{C}}|} x_i = 1 \tag{3.119}$$

$$x_i \in \{0, 1\} \qquad \forall i \in \mathbb{N} \cap [1, |\Omega|] \tag{3.120}$$

where:

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

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.

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

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

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

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

$$\mathcal{F}_{\mathcal{E}}(\widetilde{\mathcal{P}}) \stackrel{def}{=} \bigcup_{i=1}^{z} \mathcal{F}_{\mathcal{E}}(\widetilde{\mathcal{P}}_i) \tag{3.121}$$

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

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

$$\delta_{\mathcal{C}_i} \overset{def}{=} \left\{ \phi \in \mathcal{F}_{\mathcal{E}}(\widetilde{\mathcal{P}}) : \phi \in \pi_i \right\} \tag{3.122}$$

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

$$\Delta_{\mathcal{C}} \overset{def}{=} \bigcup_{i=1}^{z} \delta_{\mathcal{C}_i} \tag{3.123}$$

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

$$max \quad \sum_{i=1}^{|\mathcal{F}_{\mathcal{E}}(\mathcal{C})|} \sum_{j=1}^{|\mathbf{F}_{\phi_i} \times \mathbb{N}|} y_{\phi_{i_j}} \cdot p_{\phi_{i_j}}(t) \tag{3.124}$$

$$\text{subject to} \quad \sum_{i=1}^{|\mathcal{F}_{\mathcal{E}}(\mathcal{C})|} \sum_{j=1}^{|\mathbf{F}_{\phi_i} \times \mathbb{N}|} y_{\phi_{i_j}} \cdot c_{\phi_{i_j}}(t) \leq C \tag{3.125}$$

$$\sum_{\phi_i \in \mathcal{F}_{\mathcal{E}}(\mathcal{C}) \backslash \mathcal{F}_{\mathcal{E}}(\widetilde{\mathcal{P}})} \sum_{j=1}^{|\mathbf{F}_{\phi_i} \times \mathbb{N}|} y_{\phi_{i_j}} \cdot rt_{\phi_{i_j}}(t) +$$

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

$$\tag{3.126}$$

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

$$\tag{3.127}$$

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

$$\tag{3.128}$$

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

$$\tag{3.129}$$

where:

- All coefficients, or profits, associated to each executable function configuration, that appear inside the objective function 3.124, are computed using the formula 3.110.

- Expression 3.125 represents the max billed cost requirement, whose coefficients are computed using the formula 3.108.

- The 3.126 represents the max response time requirements. According to our formulation, there are exactly $|\Delta_{\mathcal{C}}|$ constraints regarding the response time,

that is one of each parallel path. Since we don't know a priori the slowest executable function inside a parallel structure, the main idea is to assure us that any path executed in parallel respect response time's constraint. Coefficients associated to each variable are computed using the formula 3.109.

- The 3.127 represents the capacity requirement, used to check if there are enough function instances on each system $\mathbf{Q}_{\omega_P^{(l)}}$ corresponding to each swarm of the choreography. The coefficient $a_{(\phi_{i_j}, \omega_P^{(l)})}$ is computed as follows:

$$
a_{(\phi_{i_j}, \omega_P^{(l)})} \stackrel{def}{=} \begin{cases} 1 & if \quad f_{\phi_i} \in x_{\phi_{i_j}} \wedge f_{\phi_i} \in \omega_P^{(l)} \\ 0 & otherwise \end{cases} \tag{3.130}
$$

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

- The 3.128 assures us that only one concrete function configuration is selected for any executable function.

- The 3.129 states that y_i is a binary variable set to 1 when $x_{\phi_{i_j}}$ is used; otherwise it is set to 0.

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

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

3.8.2.1 The heuristic approach based on ACO

According to our experiments, the MMKP formulation performs notably better than MKP one, producing the exact solution quicker and using less memory; this is due both to the lack of need of choreography's configurations enumeration, which require a lot of memory and time, and to a minor number of columns, despite the MMKP's formulation has more constraints respect to the MKP one.

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

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

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

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.

3.8.2.2 Solution components graph

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

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

Formally, let the MMKP formulation reported in 3.124, any weighted directed graph $G = (O, E)$ such that:

$$|O| = \sum_{j=1}^{n} |g_j| \qquad (3.131)$$

$$(o_{i_j}, o_{k_m}) \in E \Leftrightarrow i \neq k \qquad \forall i, k \in \mathbb{N} \cap \{1, n\}, \forall m, j \in \mathbb{N} \qquad (3.132)$$

is called solution components graph, where:

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

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

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

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

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

$$w_{\mathbf{min}} \leq \tau_k(o_{i_j}, o_{k_m}) \leq w_{\mathbf{max}} \qquad (3.133)$$

$$\tau_0(o_{i_j}, o_{k_m}) = w_{\mathbf{max}} \qquad \forall i, k \in \mathbb{N} \cap \{1, n\}, \forall m, j \in \mathbb{N} \qquad (3.134)$$

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

3.8.2.3 Algorithm description

ACO algorithms are conceptually very simple and they follow a specific scheme outlined in algorithm 2.

In general, after the initialization of the pheromone trails and some parameters, a main loop is repeated until a termination condition is met. The ants construct feasible solutions, which are subsequently improved by applying a local search algorithm. Finally the pheromone trails are updated.

Algorithm 2: Algorithmic skeleton for ACO algorithms

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

3.8.2.4 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:

- the attractiveness of the objects.

- the path traveled so far by the ant.

- the pheromone trail laid on the solution components graph G.

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

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

$$\pi_{k_l}^{(z)} = o_1 e_1 o_2 \ldots o_{z-1} e_z o_{z+1} \tag{3.135}$$

where:

$$o_s \in O \qquad \forall s \in \mathbb{N} \cap [1, z] \tag{3.136}$$

$$e_s = (o_s, o_{s+1}) \in E \qquad \forall s \in \mathbb{N} \cap [1, z] \tag{3.137}$$

$$g(o_s) \neq g(o_r) \qquad \forall s, r \in \mathbb{N} \cap [1, z] : s \neq r \tag{3.138}$$

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

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

3.8.2.5 Ant's solution construction algorithm

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

$$P(o_{i_j}, \mathbf{S}_{k_l}^{(z)}, \pi_{k_l}^{(z)}) \stackrel{def}{=} \frac{\left[\tau(o_{i_j}, \pi_{k_l}^{(z)})\right]^{\alpha} \cdot \left[\eta(o_{i_j}, \mathbf{S}_{k_l}^{(z)})\right]^{\beta}}{\displaystyle\sum_{o_{i_j} \in \mathcal{C}(\mathbf{G}_i, \mathbf{S}_{k_l}^{(z)})} \left[\tau(o_{i_j}, \pi_{k_l}^{(z)})\right]^{\alpha} \cdot \left[\eta(o_{i_j}, \mathbf{S}_{k_l}^{(z)})\right]^{\beta}} \tag{3.139}$$

where:

- $\tau(o_{i_j}, \pi_{k_l}^{(z)})$ is the pheromone factor.

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

1 **for** $l \leftarrow 0$ **to** m **by** 1 **do**

2 $z \leftarrow 0$;

3 $\mathbf{S}_{k_l}^{(z)} \leftarrow \emptyset$;

4 $\mathbf{G} \leftarrow \mathcal{G}$;

5 $\mathbf{G}_i \leftarrow$ Randomly select a group from $\mathbf{G}$;

6 $o_{i_j} \leftarrow$ Randomly select an object from $\mathbf{G}_i$;

7 $z \leftarrow z + 1$;

8 $\mathbf{S}_{k_l}^{(z)} \leftarrow \{o_{i_j}\}$;

9 $\mathbf{G} \leftarrow \mathbf{G} \setminus \mathbf{G}_i$;

10   **while** $G \neq \emptyset$ **do**

11 $\mathbf{G}_a \leftarrow$ Randomly select a group from $\mathbf{G}$;

12 $\mathcal{O} \leftarrow$ Select a group of candidates objects belonging to $\mathbf{G}_a$ which do not viotale resource constraints ;

13 $o_{a_b} \leftarrow$ Select an object from $\mathbf{G}_a$ having highest transition probability $P(o_{a_b}, \mathbf{S}_{k_l}^{(z)}, \pi_{k_l}^{(z)})$;

14 $z \leftarrow z + 1$;

15 $\mathbf{S}_{k_l}^{(z)} \leftarrow \mathbf{S}_{k_l}^{(z)} \cup o_{a_b}$;

16 $\mathbf{G} \leftarrow \mathbf{G} \setminus \mathbf{G}_a$;

17   **end**

18 **end**

19 $S_k \leftarrow$ Select $\mathbf{S}_{k_l}^{(z)}$ having maximum profit;

20 **return** S_k

- $\eta(o_{i_j}, \mathbf{S}_{k_l}^{(z)})$ 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:

$$\tau(o_{i_j}, \pi_{k_l}^{(z)}) \overset{def}{=} \tau(o_{z+1}, o_{i_j}) + \sum_{e \in \pi_{k_l}^{(z)}} \tau(e) \tag{3.140}$$

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:

$$h_{\mathbf{S}_{k_l}^{(z)}}(o_{i_j}) \overset{def}{=} \sum_{y=0}^{Y} \frac{c_{y_j} + 1}{\left(b_y - \sum_{o_s \in \mathbf{S}_{k_l}^{(z)}} c_y(o_z)\right) + 1} \tag{3.141}$$

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:

$$\eta(o_{i_j}, \mathbf{S}_{k_l}^{(z)}) \overset{def}{=} \frac{p_{i_j}}{h_{\mathbf{S}_{k_l}^{(z)}}(o_{i_j}) + 1} \tag{3.142}$$

3.8.2.6 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 best solution.

To be precise, we have adopted a *random local search* algorithm, which represents an exhaustive search within a group to improve the solution. It replaces current selected object of a group with every other object that 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. The pseudo-code of that procedure is reported in algorithm 4.

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

1   **for** *a specified number of times* **do**

2     $\mathbf{G}_i \leftarrow$ Randomly select a group from $\mathcal{G}$;

3   **for** *each object $o_{i_j} \in \mathbf{G}_i$ other than the one in $\mathbf{S}_k$* **do**

4       $o_{i_s} \leftarrow$ The object belonging to $\mathbf{S}_k$ such that $g(o_{i_s}) = g(o_{i_j})$

5       $\mathbf{S}_k^{(temp)} \leftarrow \{\mathbf{S}_k \setminus \{o_{i_s}\}\} \cup \{o_{i_j}\}$

6   **if** $\boldsymbol{S}_k^{(temp)}$ *not violate MMKP constraints* **then**

7   **if** $profit(\boldsymbol{S}_k^{(temp)}) > profit(\boldsymbol{S}_k)$ **then**

8          $\mathbf{S}_k \leftarrow \mathbf{S}_k^{(temp)}$

9   **end**

10   **end**

11   **end**

12 **end**

13 **return** $\mathbf{S}_k$

3.8.2.7   Pheromone Trail Update

Let $m \in \mathbb{N} \setminus \{0\}$ 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:

$$\tau(o_{i_a}, o_{j_b})_k = \tau(o_{i_a}, o_{j_b})_{k-1} \cdot \rho + \Delta\tau(o_{i_a}, o_{j_b})_k \qquad (3.143)$$

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:

$$\Delta\tau(o_{i_a}, o_{j_b})_k \stackrel{def}{=} \sum_{l=1}^{m} \tau(o_{i_a}, o_{j_b})_k^{(l)} \qquad (3.144)$$

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:

$$\tau(o_{i_j}, o_{a_b})_{k-1}^{l} \stackrel{def}{=} \begin{cases} 0 & \text{if} \quad S_{k_l} = \emptyset \\ \dfrac{1}{1 + profit(S_{best}) - profit(S_{k_l})} & \text{if} \quad \substack{S_{k_l} \neq \emptyset \\ (o_{i_j}, o_{a_b}) \in \pi_{k_l}} \end{cases}$$
$$(3.145)$$

3.8.2.8 Termination Conditions

According to our model, let $\mathcal{C} = (\Phi, E)$ a choreography, an upper bound on the value of the optimal solution exists; since is true that $0 \leq p_{i_j}(t) \leq 1$, where $p_{i_j}(t)$ is

the score associated to the executable function configuration $x_{\phi_{i_j}}$, the upper bound of the optimal solution is simply equal to $|\mathcal{F}_{\mathcal{E}}(\mathcal{C})|$.

Then, our algorithm stops when:

- an ant has found an optimal solution.

- a maximum number of cycles has been performed.

- the improvement in the objective function in an iteration is less than a threshold; by default, if the improvement of the solution is less than the 10% respect to the previous one, our algorithm will terminate.

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

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[16] Mahadev Satyanarayanan, Guenter Klas, Marco Silva, and Simone Mangiante. The seminal role of edge-native applications. In *2019 IEEE International Conference on Edge Computing (EDGE)*, pages 33–40, 2019. doi: 10.1109/EDGE.2019.00022.

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

75

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

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