Contents

$$
\begin{aligned}
\min \quad & \sum_{n=1}^{N}\sum_{k=1}^{K}\sum_{x\in C} g(x)^{f_k^i} Y_x^{f_k^i} \\
& \sum_{n=1}^{N}\sum_{k=1}^{K}\sum_{x\in C} T(x)^{f_k^i} Y_x^{f_k^i} \quad \leq \quad D_{\mathbf{F}} \\
& \sum_{n=1}^{N}\sum_{k=1}^{K}\sum_{x\in C} Y_x^{f_k^i} \qquad\qquad = \qquad 1 \\
& Y_x^{f_k^i} \in \{0,1\}
\end{aligned}
\tag{1}
$$

$$
\begin{aligned}
\min \quad & \sum_{\pi\in\Pi}\sum_{n=1}^{N}\sum_{k=1}^{K}\sum_{x\in C} g(x)^{f_k^i} Y_{(\pi,x)}^{f_k^i} \\
& \sum_{n=1}^{N}\sum_{k=1}^{K}\sum_{x\in C} T(x)^{f_k^i} Y_x^{f_k^i} \qquad\qquad \leq \quad D_{\mathbf{F}} \\
& \sum_{\pi\in\Pi}\sum_{n=1}^{N}\sum_{k=1}^{K}\sum_{x\in C} Y_{(\pi,x)}^{f_k^i} \qquad\qquad = \qquad 1 \\
& Y_{(\pi,x)}^{f_k^i} \in \{0,1\}
\end{aligned}
\tag{2}
$$

0.1 Serverless Computing Paradigm

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

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

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

0.1.1 Serverless Function

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

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

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

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

0.1.2 Serverless Application

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

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

Generally, the coordinator is a broker needed to implement the business logic of any application: it chains together serverless function, handles events among functions and triggers them in the correct order according to the business logic defined by developers.

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

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

and provider lock-in

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

FC languages and runtime systems are still in their infancy with numerous drawbacks. Current commercial and open source offerings of FC systems are dominated by a few large- scale providers who offer their own platforms,

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

0.1.3 fdsfsd

•

0.1.4 title

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

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

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

a model for each executable belonging to a given choreography.

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

A cold start

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

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

0.2 Orchestrator Model

0.2.1 Resource owner

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

Chapter 1

Specification Mode

1.1 Serverless Choreography

In this section we will introduce the notion of *serverless choreography*, a very important *resource* adopted to model and implement both serverless functions and serverless applications.

Informally, the notion of serverless choreography has been derived from that of a control-flow graph; as known, the latter describes, using graph notation, all paths that might be traversed through a serverless application during its execution. Similarly, a serverless choreography describes calling relationships between serverless functions which can be combined using several types of control-flow connectors.

1.1.1 Preliminary Definitions

Before to define what we mean for serverless choreography, is necessary to introduce some very useful notations.

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

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

Then, we will adopt following notations:

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

$$\pi = \phi_1 e_1 \phi_2 e_2 \dots e_{n-2} \phi_{n-1} e_{n-1} \phi_n \tag{1.1}$$

  where:

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

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

$$out(\phi_u) \stackrel{def}{=} \{(\phi_u, \phi) \in E, \quad \forall \phi \in \Phi\} \tag{1.2}$$

$$in(\phi_u) \stackrel{def}{=} \{(\phi, \phi_u) \in E, \quad \forall \phi \in \Phi\} \tag{1.3}$$

$$succ(\phi_u) \stackrel{def}{=} \{\phi \in \Phi \mid (\phi_u, \phi) \in out(\phi_u)\} \tag{1.4}$$

$$pred(\phi_u) \stackrel{def}{=} \{\phi \in \Phi \mid (\phi, \phi_u) \in in(\phi_u)\} \tag{1.5}$$

1.1.2 Definition

According to the serverless paradigm, the execution of a serverless application starts with a particular function, which we will call *entry point*, while any other serverless functions, belonging to the application, will be invoked subsequently according to specified business logic defined by developers; as we will see shortly, the latter can be naturally modeled by a weighted directed graph.

Clearly, the execution of a serverless application ends when the execution of the last function of the application ends; that ending function will be call as *end point*. We assume that the entry point is unique.

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

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

$$\mathcal{C}_R \stackrel{def}{=} (\Phi, E) \tag{1.6}$$

where:

- $|\Phi| = n$;
- Each vertex $\phi \in \Phi$ is called *abstract serverless function* and represents a computational unit; we will describe more in detail what we mean by abstract serverless function shortly;
- Let $i, j \in \mathbb{N} \cap [1, n]$ and $\phi_i, \phi_j \in \Phi$, any directed edge $e_{ij} = (\phi_i, \phi_j) \in E$ represents the calling relationship between two abstract serverless function which depends on the business logic defined by R.

  In our context, any directed edge $(\phi_i, \phi_j) \in E$ states that the abstract serverless function ϕ_j *can* be called by ϕ_i;
- Let $i, j \in \mathbb{N} \cap [1, n]$, the number $p_{ij} \in \mathbb{R} \cap [0, 1]$ is the weight assigned to the edge $(\phi_i, \phi_j) \in E$, where:
  - The number p_{ij} represents the so-called *transition probability* from ϕ_i to ϕ_j, that is the probability of invoking ϕ_j after finishing the execution of ϕ_i;

– We will use a function $P : \Phi \times \Phi \to [0, 1]$, called *transition probability function*, such that $P(\phi_i, \phi_j) = p_{ij}$. When $P(\phi_i, \phi_j) = 0$, it implies that the directed edge $(\phi_i, \phi_j) \notin E$, therefore ϕ_i cannot invoke ϕ_j;

– For any path $\pi = \phi_1 e_1 \dots e_{n-1} \phi_n$ of $\mathcal{C}_R$, we define *transition probability of the path* π the following quantity:

$$TPP(\pi) = \prod_{i=1}^{n-1} P(\phi_i, \phi_{i+1}) \tag{1.7}$$

- Φ contains only one serverless abstract function, denoted by ϕ_{entry}, acting as the entry point and at least one acting as the end point, which is denoted, instead, by ϕ_{end}. Formally, we define aforementioned vertices as follows:

$$\begin{array}{lll} \phi \in \Phi \quad \text{is the entry point of } \mathcal{C}_\mathcal{R} & \Leftrightarrow & in(\phi) = \emptyset \\ \phi \in \Phi \quad \text{is the end point of } \mathcal{C}_\mathcal{R} & \Leftrightarrow & out(\phi) = \emptyset \end{array} \tag{1.8}$$

  Then, following conditions must be hold:

$$\exists! \phi \in \Phi \quad | \quad in(\phi) = \emptyset \tag{1.9}$$

$$\exists \phi \in \Phi \quad | \quad out(\phi) = \emptyset \tag{1.10}$$

  Finally, we will also use the notation $\alpha(\mathcal{C}_R)$ to denote the entry point ϕ_{entry} of a choreography $\mathcal{C}_R$. Conversely, we will adopt the notation $\omega(\mathcal{C}_R)$ to denote the set of all end points of $\mathcal{C}_R$

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

$$\sum_{\phi_{end} \in \omega(\mathcal{C}_R)} \Big(\sum_{\pi \in \Pi(\phi_{entry}, \phi_{end})} TPP(\pi) \Big) = 1 \tag{1.11}$$

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

- A function, denoted by $D : \Phi \times \Phi \to [0, \infty)$, represents a delay function according to which $D(\phi_i, \phi_j)$ identifies the delay from ϕ_i to ϕ_j due to network delay or orchestration task;

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

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

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

1.1.2.1 Abstract Serverless Function

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

As said previously, any $\phi \in \Phi$ is called *abstract serverless function*, or simply *abstract function*, that is a *resource* representing a computational unit required by business logic provided by developers.

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

- ϕ is called *serverless executable functions*, or simply *executable function*, when ϕ contains executable code; therefore, in that case, we said that an executable function models a serverless function.

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

  $$\mathcal{F}_{\mathcal{E}} \stackrel{def}{=} \{\phi \in \Phi \mid \phi \text{ is a serverless executable function } \} \tag{1.12}$$

  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.

  For any $\phi \in \mathcal{F}_{\mathcal{E}}$, we will use $\mathbf{F}_{\phi}$ notation to represent the so-called *implementation-set* of ϕ, that is the set containing any concrete implementation, denoted as f_{ϕ}, of ϕ.

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

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

  According to our model, orchestration code represents the logic required to chain together any components of an application, 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; later we will describe how it is possible.

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

  $$\mathcal{F}_{\mathcal{O}} \stackrel{def}{=} \{\phi \in \Phi \mid \phi \text{ is a serverless orchestration function } \} \tag{1.13}$$

Clearly, based on above definitions, we can say:

$$\mathcal{F}_{\mathcal{E}} \cap \mathcal{F}_{\mathcal{O}} = \emptyset \tag{1.14}$$

$$\mathcal{F}_{\mathcal{E}} \cup \mathcal{F}_{\mathcal{O}} = \Phi \tag{1.15}$$

$$|\mathcal{F}_{\mathcal{E}}| + |\mathcal{F}_{\mathcal{O}}| = |\Phi| \tag{1.16}$$

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;

1.1.2.2 Executability condition

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

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

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

$$\mathcal{C} \text{ is executable} \quad \Leftrightarrow \quad |\mathbf{F}_\phi| \geq 1 \qquad \forall \phi \in \mathcal{F}_{\mathcal{E}} \tag{1.17}$$

We will only deal with executable choreographies. Moreover, for all $\phi \in \mathcal{F}_{\mathcal{E}}$, we always assume that all $f_\phi \in \mathbf{F}_\phi$ are already deployed on one or more FaaS platform by developers.

1.1.2.3 Serverless Sub-choreography

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

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

$$C^* \stackrel{def}{=} (\Phi^*, E^*) \qquad \text{where } \Phi^* \subseteq \Phi \wedge E^* \subseteq E \tag{1.18}$$

is called *serverless sub-choreography* of $\mathcal{C}$, or simply *sub-choreography* of $\mathcal{C}$, when the conditions 1.9, 1.10 and 1.11 are hold.

1.1.2.4 Basic Serverless Choreography

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

$$|\mathcal{F}_{\mathrm{O}}| = 0 \tag{1.19}$$
$$|\omega(\mathcal{C})| = 1 \tag{1.20}$$

We said that any choreography $\mathcal{C}$, satisfying the conditions 1.19 and 1.20, represents a *basic serverless choreography*, or simply a *basic choreography*.

1.2 Serverless Choreography Configuration

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

Informally, a configuration specifies which concrete function will be use for all executable function inside a choreography, specifying several parameters, like memory size or CPU power.

1.2.1 Executable Function Configuration

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

Let $\phi \in \mathcal{F}_{\mathcal{E}}$ 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:

$$x_{\phi} = (f_{\phi}, m) \in f_{\phi} \times \mathbf{M}_{f_{\phi}} \subseteq \mathbf{F}_{\phi} \times \mathbb{N} \tag{1.21}$$

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.

1.2.2 Serverless Choreography Configuration

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

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

$$\begin{aligned}
\mathbf{X}_{\mathcal{C}} &\overset{def}{=} \{x_{\phi_1}, \ldots, x_{\phi_k}\} \\
&\in \left\{ \left\{ \bigcup_{j=1}^{|\mathbf{F}_{\phi_1}|} f_{\phi_{1_j}} \times \mathbf{M}_{f_{\phi_{1_j}}} \right\} \times \ldots \times \left\{ \bigcup_{j=1}^{|\mathbf{F}_{\phi_k}|} f_{\phi_{k_j}} \times \mathbf{M}_{f_{\phi_{k_j}}} \right\} \right\} \\
&= \bigtimes_{i=1}^{k} \left\{ \bigcup_{j=1}^{|\mathbf{F}_{\phi_i}|} f_{\phi_{i_j}} \times \mathbf{M}_{f_{\phi_{i_j}}} \right\} \\
&\subseteq \bigtimes_{i=1}^{k} \{\mathbf{F}_{\phi_i} \times \mathbb{N}\}
\end{aligned} \tag{1.22}$$

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

1.3 Executable Function Performance

In order to select an appropriate serverless choreography configuration able to effectively deliver and guarantee the QoS level required by our users, we need to evaluate the performance of any executable function inside a choreography when a particular concrete function is chosen to implement it.

1.3.1 Concurrency level

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

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

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

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

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

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

This approach is adopted by Google Cloud Functions although there are reports stating that, despite there is no explicitly mentioned global concurrency limit, a kind of global concurrency limit, varying between 1000 and 2000, is observed.

1.3.2 Platform provider modelling

In order to formalize the notion of concurrency limit, explaining its role inside the way according to which serverless executable function performance are computed, is necessary to explain how FaaS platform providers had been modeled.

From our model point of view, any FaaS platform provider can acting as a "*set*" of $M/G/K(t)_{\mathbf{C}_{max}}/K(t)_{\mathbf{C}_{max}}$ queueing systems, where:

- $\mathbf{C}_{max} \in \mathbb{N}$ represents the *maximum value of queue concurrency limit*, that is the maximum number of function instances being in running state simultaneously at time t inside that queue.

  Please note that $\mathbf{C}_{max}$ not necessary coincide with global concurrency limit because it can represents a local concurrency limit too.

- At time $t \geq 0$, there are $K(t)$ function instances that can each process one request and the system has capacity for $K(t)$ invocation requests in total.

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

At any time t, following condition must be hold:

$$0 \leq K(t) \leq C_{\mathbf{max}} \tag{1.23}$$

- No queue is involved, therefore any incoming request at time t, which sees all $K(t)$ function instances busy and $K(t) = \mathbf{C_{max}}$, is considered dropped.
- No priority is considered among incoming request.
- Service times have a general distribution while a Poisson arrival process is assumed.

Please note that, according to our model, any FaaS platform can host several $M/G/K(t)_{\mathbf{C}_{max}}/K(t)_{\mathbf{C}_{max}}$ queueing systems, many of which can belong to an unique resource owner.

1.3.3 Serverless concrete function swarms

To complete the formalization of our model, is necessary to introduce another very important concept.

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 and belonging to R.

A *serverless concrete function swarms* is any set $\omega_{P_{R_l}} \subseteq \Omega_{P_R}$ containing all concrete functions belonging to R that share the same limit l in term of the max number of serverless function instance runnable at the same time by the corresponding platform provider P.

That limit can be called *server-less function swarm's concurrency limit*, or simply, *concurrency limit*.

Is very important to make clear that only at most l concrete functions belonging to $\omega_{P_{R_l}}$ can be executed simultaneously by P. Clearly, the value of l depends on the concurrency model by P and this kind of formalization make our model compatible with both approaches described previously. Formally:

- 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, there is a $M/G/K(t)_l/K(t)_l$ queuing system for each concrete function in Ω_{P_R}, every of which will serve all invocation request regarding only one concrete function in $\omega_{P_{R_l}}$.

  In other words, there are $|\omega_{P_{R_l}}|$ different queues.

- 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 and l represents the provider's global concurrency limit.

  In that case, there is only one $M/G/K(t)_l/K(t)_l$ queuing system serving all invocation request regarding any concrete function in $\omega_{P_{R_l}}$.

1.3.4 Executable function state

Similarly to previous studies, a function instance can be in one of the following states:

Initialization state which happens when the infrastructure is spinning up new function instances which are needed to handle incoming requests.

Generally, depending on the approach adopted by the FaaS provider, during this state is needed to set up new virtual machines, unikernels, containers, or even, processes, in order to be able to process new requests.

A function instance will remain in the initializing state until it is able to handle incoming requests, entering into idle state. The time spent in this state is not billed.

Idle State After the fulfillment of all initialization task or when the processing of a previously received invocation request is over, the serverless platform keeps the instances in idle state, keeping the function instance able to handle future invocation request faster, since no initialization task is needed to be performed.

FaaS platform provider keeps a function instance in idle state for a limited amount of time. 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.

1.3.5 Start/Cold start

When a request arrives on a FaaS platform,

Cold start If the warm pool is busy, that is there are no serverless function instances in idle state able to serve an newly incoming request, latter will trigger the launching of a new function instance, which will be added to the warm pool. This event is called *cold start*.

From the FaaS provider point of view, this operation requires to start either a new virtual machine or a new container; in any case, regardless of the method adopted, a cold start introduces a very important overhead to the response time experienced by users.

Warm start When there is at least one serverless function instance in idle state, the FaaS platform reuses it to serve the incoming request without launching a new one. This is called *warm start*.

A very important aspect is that 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.

To be more precise, let $k \in \mathbb{N}$ and $C_{\max,P}$ the maximum concurrency level imposed by FaaS provider P, when a new request arrives on the platform, one of the following events can occur:

- If $k < C_{\max,P}$, that is the number of function instances into the warm pool is less than the maximum concurrency level, the request will be blocked by the system causing a cold start.
- If $k = C_{\max,P}$, that is the warm pool size reaches the maximum concurrency level, any further request will be rejected by the platform

1.3.5.1 Cold start probability

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

In order to fully describe how our framework works, we have to know the probability according to which a request is rejected by the warm pool; in other words, we must to compute the *cold start probability*.

Let $k \in N$ the size of the warm pool, that is the number of existing function instances currently active on the provider.

Formally, the cold start probability $\mathbf{P}_{C_R}$ is the probability that an arrival finds all k function instances of the warm pool busy. Using Erlang-B formula, aforementioned probability can be calculated as follows:

$$\mathbf{P}_C = \frac{\frac{\rho^k}{k!}}{\sum_{j=0}^{k} \frac{\rho^j}{j!}} \tag{1.24}$$

where:

- $\rho = \dfrac{\lambda}{\mu_w}$ represents system utilization or load of the pool where:
  - λ represents the *average arrival rate*, that is the rate at which serverless compound functions execution requests arrive to our system.
  - $\mu_w = E[S_w]$ 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.

At this point, we can define some useful functions:

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

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

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

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

1.4 The problem of cold starts

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

1.4.1 Metrics

Let $\mathcal{C} = (\Phi, E)$ a basic choreography as defined in section 1.1.2.4.

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

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

$$RT_P(\pi, \mathbf{X}) = \sum_{i=1}^{n} (RT_{\phi_i}(X(i)) + \sum_{i=1}^{n-1} D(\phi_i, \phi_{i+1}) \tag{1.25}$$

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

$$C_P(\pi, \mathbf{X}) = \sum_{\substack{1 \leq i \leq n \\ \phi_i \in \bar{\mathcal{F}}_{\mathcal{E}}}} N(\phi_i) \cdot C_{\mathbf{F}_{\phi_i}}(X(i)) \tag{1.26}$$

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

$$RT_C(\mathcal{S}, \mathbf{X}) = \sum_{\pi \in \Pi(\phi_{entry}, \phi_{end})} TPP(\pi) \cdot RT_P(\pi, \mathbf{X}) \tag{1.27}$$

$$C_C(\mathcal{S}, \mathbf{X}) = \sum_{\pi \in \Pi(\phi_{entry}, \phi_{end})} TPP(\pi) \cdot C_P(\pi, \mathbf{X}) \tag{1.28}$$

1.5 Serverless Choreography Structures

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

$$S^* \text{ is a structure } \Leftrightarrow \begin{cases} \phi^*_{entry} & \in \mathcal{F}^*_{\mathcal{O}} \quad \wedge \quad \mathcal{T}(\phi^*_{entry}) = \tau_\alpha \\ \phi^*_{end} & \in \mathcal{F}^*_{\mathcal{O}} \quad \wedge \quad \mathcal{T}(\phi^*_{end}) = \tau_\omega \end{cases} \tag{1.29}$$

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

$$\begin{aligned} A &\stackrel{def}{=} \Phi^* \setminus \{\phi^*_{entry}, \phi^*_{end}\} \\ B &\stackrel{def}{=} E^* \setminus \left(out\left(\phi^*_{entry}\right) \cup in\left(\phi^*_{end}\right)\right) \end{aligned} \tag{1.30}$$

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

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

$$\Theta_{\mathcal{S}^*} \stackrel{def}{=} \{\theta_i, \ldots, \theta_c\}$$

where

$$\theta_i \stackrel{def}{=} (\Phi_i^{**}, E_i^{**}) \text{ is a sub-choreography of } \mathcal{S}^* \quad \Phi_i^{**} \subset \Phi^* \wedge E_i^{**} \subset E^* \qquad \forall i \in [1, c] \tag{1.31}$$

1.5.0.1 Parallel

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

$$TPP(\pi) = 1 \qquad \forall \pi \in \Pi(\phi_{start}, \phi_{end}) \tag{1.32}$$

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

$$RT_{parallel}(\mathcal{P}, \mathbf{X}) = max\left\{RT_C(\theta, \mathbf{X}) \mid \theta \in \Theta\right\} \tag{1.33}$$

$$C_{parallel}(\mathcal{P}, \mathbf{X}) = \sum_{\theta \in \Theta} C_C(\theta, \mathbf{X}) \tag{1.34}$$

1.5.0.2 Branch/Switch

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

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

$$TPP(\pi) \neq 1 \qquad \forall \pi \in \Pi(\phi_{entry}, \phi_{end}) \tag{1.35}$$

$$|\Theta| = n \qquad n \in \{1, 2\} \tag{1.36}$$

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

$$RT_{branch}(\mathcal{B}, \mathbf{X}) = \sum_{i=1}^{n} P(\phi_{entry}, \alpha(\theta_i)) RT_C(\theta_i, \mathbf{X}) \tag{1.37}$$

$$C_{branch}(\mathcal{B}, \mathbf{X}) = \sum_{i=1}^{n} C_C(\phi_{entry}, \alpha(\theta_i)) C_C(\theta_i, \mathbf{X}) \tag{1.38}$$

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

1.5.0.3 Loop

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

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

$$P(\phi_{start}, \alpha(\theta)) = 1 \tag{1.39}$$

$$P(\phi_{end}, \phi_{entry}) \neq 0 \tag{1.40}$$

$$P(\phi_{end}, \phi_{entry}) + \sum_{\phi \in succ(\phi_{end})} P(\phi_{end}, \phi) = 1 \tag{1.41}$$

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

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

$$p = 1 - P(\phi_{end}, \phi_{entry}) \tag{1.42}$$

Then:

$$P(EI(\mathcal{L}) = k) = \Big[1 - P(\phi_{end}, \phi_{entry})\Big] \cdot \Big[P(\phi_{end}, \phi_{entry})\Big]^{k-1} \tag{1.43}$$

$$= pq^{k-1} \tag{1.44}$$

$$\begin{aligned} E[X] &= \sum_{k=1}^{\infty} (k-1) p q^{k-1} \\ &= p \sum_{k=0}^{\infty} k q^k \\ &= p \cdot \frac{q}{(1-q)^2} \\ &= \frac{q}{p} \\ &= \frac{P(\phi_{end}, \phi_{entry})}{1 - P(\phi_{end}, \phi_{entry})} \end{aligned} \tag{1.45}$$

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

$$RT_{loop}(\mathcal{L}, \mathbf{X}) = E[I_{\mathcal{L}}] \cdot RT_C(\theta, \mathbf{X}) \tag{1.46}$$

$$C_{loop}(\mathcal{L}, \mathbf{X}) = E[I_{\mathcal{L}}] \cdot C_C(\theta, \mathbf{X}) \tag{1.47}$$

1.6 Optimization

1.6.1 Performance Modeling

We define the utility function as follows:

$$F(\mathbf{x}) \stackrel{def}{=} w_{RT} \cdot \frac{RT_{max} - RT(\mathbf{x})}{RT_{max} - RT_{min}} + w_C \cdot \frac{C_{max} - C(\mathbf{x})}{C_{max} - C_{min}} \tag{1.48}$$

where $w_{RT}, w_C \geq 0$, $w_{RT} + w_C = 1$, are weights for the different QoS attributes, while RT_{max} (RT_{min}) and C_{max} (C_{min}) denote, respectively, the maximum (minimum) value for the overall expected response time and cost.

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

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

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

$$\begin{aligned}
max \quad & \sum_{\omega=1}^{|\Omega|} x_\omega F(\mathbf{X}_\omega) \\
\text{subject to} \quad & \sum_{\omega=1}^{|\Omega|} x_\omega C(\mathbf{X}_\omega) \leq C_{user} \\
& \sum_{\omega=1}^{|\Omega|} x_\omega RT(\mathbf{X}_\omega) \leq RT_{user} \\
& \sum_{\omega=1}^{|\Omega|} x_\omega = 1 \\
& x_\omega \in \{0,1\} \qquad\qquad \forall \omega \in \mathbb{N} \cap [1, |\Omega|]
\end{aligned} \tag{1.49}$$

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

1.6.2 Multi-dimensional Multi-choice Knapsack Problem

$$
\begin{aligned}
max \quad & \sum_{i=1}^{|\mathcal{F}_{\mathcal{E}}|} \sum_{j=1}^{|\mathbf{F}_{\phi_i} \times \mathbb{N}|} x_{\phi_{i_j}} p_{\phi_{i_j}} \\
\text{subject to} \quad & \sum_{i=1}^{|\mathcal{F}_{\mathcal{E}}|} \sum_{j=1}^{|\mathbf{F}_{\phi_i} \times \mathbb{N}|} x_{\phi_{i_j}} c_{\phi_{i_j}} \leq C_{user} \\
& \sum_{i=1}^{|\mathcal{F}_{\mathcal{E}}|} \sum_{j=1}^{|\mathbf{F}_{\phi_i} \times \mathbb{N}|} x_{\phi_{i_j}} t_{\phi_{i_j}} \leq RT_{user} \\
& \sum_{j=1}^{|\mathbf{F}_{\phi_i} \times \mathbb{N}|} x_{\phi_{i_j}} = 1 \qquad \forall i \in \mathbb{N} \cap [1, |\mathcal{F}_{\mathcal{E}}|] \\
& x_{\phi_{i_j}} \in \{0, 1\} \qquad \forall i \in \mathbb{N} \cap [1, |\mathcal{F}_{\mathcal{E}}|] \wedge \forall j \in \mathbb{N} \cap [1, |\mathbf{F}_{\phi_i} \times \mathbb{N}|]
\end{aligned}
\tag{1.50}
$$

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

Let $x_{\phi_{i_j}} = (f_j, m_j)$ an executable function configuration for the executable function ϕ_i, for some $i \in \mathbb{N} \cap [1, |\mathcal{F}_{\mathcal{E}}|]$ and $j \in \mathbb{N} \cap [1, |\mathbf{F}_{\phi_i} \times \mathbb{N}|]$.

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

$$
c_{\phi_{i_j}} \stackrel{def}{=} C(x_{\phi_{i_j}}) \cdot \prod_{\theta \in \Theta_{\phi_i}} P\left(pred(\alpha(\theta)), \alpha(\theta)\right) \cdot E[I_\theta] \tag{1.51}
$$

$$
c_{\phi_{i_j}} \stackrel{def}{=} C(x_{\phi_{i_j}}) \cdot \prod_{\theta \in \Theta_{\phi_i}} P\left(pred(\alpha(\theta)), \alpha(\theta)\right) \cdot E[I_\theta] \tag{1.52}
$$

where:

- $\Pi(\phi_{entry}, \phi_i)$
- $\Pi(\phi_{entry}, \phi_i)$ denotes the set of all possible paths starting from the entry point of choreography $\mathcal{C}_{\mathcal{R}}$ to the executable function ϕ_i.

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

$$
p_{\phi_{i_j}} \stackrel{def}{=} w_{RT} \cdot \frac{t_{\phi_{i_{\mathbf{MAX}}}} - t_{\phi_{i_j}}}{t_{\phi_{i_{\mathbf{MAX}}}} - t_{\phi_{i_{\mathbf{MIN}}}}} + w_C \cdot \frac{c_{\phi_{i_{\mathbf{MAX}}}} - c_{\phi_{i_j}}}{c_{\phi_{i_{\mathbf{MAX}}}} - c_{\phi_{i_{\mathbf{MIN}}}}} \tag{1.53}
$$

where:

- $t_{\phi_{i_{\mathbf{MIN}}}}$ and $t_{\phi_{i_{\mathbf{MAX}}}}$ represent, respectively, the minimum and maximum response time values registered during the execution of all concrete function implementing ϕ_i.

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

$$w_C \geq 0 \tag{1.54}$$
$$w_{RT} \geq 0 \tag{1.55}$$
$$w_{RT} + w_C = 1 \tag{1.56}$$

Algorithm 1: Algorithmic skeleton for ACO algorithms

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

```
Data: this text
Result: Some solution
for k ← 0 to |A| by 1 do
    S_k ← ∅ ;
    G_candidates ← G;
    G_i ← Randomly select a group from G_candidates for some i ∈ [1; |G|];
    o_{i_k} ← Randomly select an object from G_i for some k ∈ [1; |G_i|];
    S_k ← {o_{i_k}} ;
    G_candidates ← G_candidates \ G_i;
    while G_candidates ≠ ∅ do
        G_i ← Randomly select a group from G_candidates;
        O ← without violating resource constraints ;
    end
end
```

Chapter 2

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.

2.1 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.

2.2 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.

2.2.1 InfluxDB

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

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

Using InfluxDB terminology, each point is uniquely identified by four components:

- A timestamp.
- Zero o more tags, key-value pairs that are used to store metadata associated with the point.
- One or more fields, that is scalars which represent the value recorded by that point.
- A measurement, which acts as a container used to group together all related points.

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

Bibliography

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

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

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