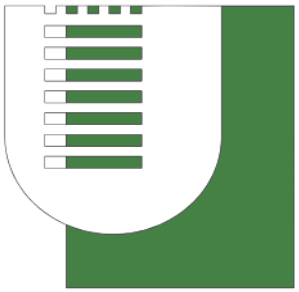


TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Macroarea di Ingegneria

A QoS-Aware Broker for Multi-Provider Serverless Applications

Andrea Graziani
m. 0273395

Supervisor:	Valeria Cardellini
Tutor:	Gabriele Russo Russo

May 23, 2022

Goals

My thesis is focused on "**Serverless Computing**".

Two main goals:

1. The **orchestration** of a serverless workflow supporting:
   - **Multiple FaaS Providers**
   - **Multiple Functions Implementations.**

2. The fulfillment of **non-functional requirements** concerning the **quality of service** (QoS) levels that should be guaranteed.

Serverless Computing: Overview

What's meant by serverless computing?

An applications service model where:

- Administration tasks (provisioning, monitoring, scaling, etc.) are directly managed by the provider.
- Small-granularity billing pricing model: **pay-as-you-go**.
- Cloud application are abstracted as a group of so-called "**Serverless Functions**".

What is a serverless function?

A computation unit implementing a business functionality.

- **Stateless**
- **Event-Driven**
- **Short-Lived**

Serverless Application

A software system made up of a serverless functions set.

Serverless Computing: Overview

A serverless function is executed inside a containerized environment: the so-called "**Function Instance**".

The FaaS platform **automatically** scales the number of function instances.

When a request comes in, only one of the following events can occur:

- **Warm start**.
- **Cold start.**

Concurrency Level

The limitation, imposed by FaaS platforms, on the **number** of function instance runnable at the **same time**.

Providers apply that restriction differently:

- **Global (Per-Account) Concurrency Model**.
  - AWS Lambda.
  - IBM Cloud Functions.
- **Local (Per-Function) Concurrency Model**.
  - Google Cloud Functions.

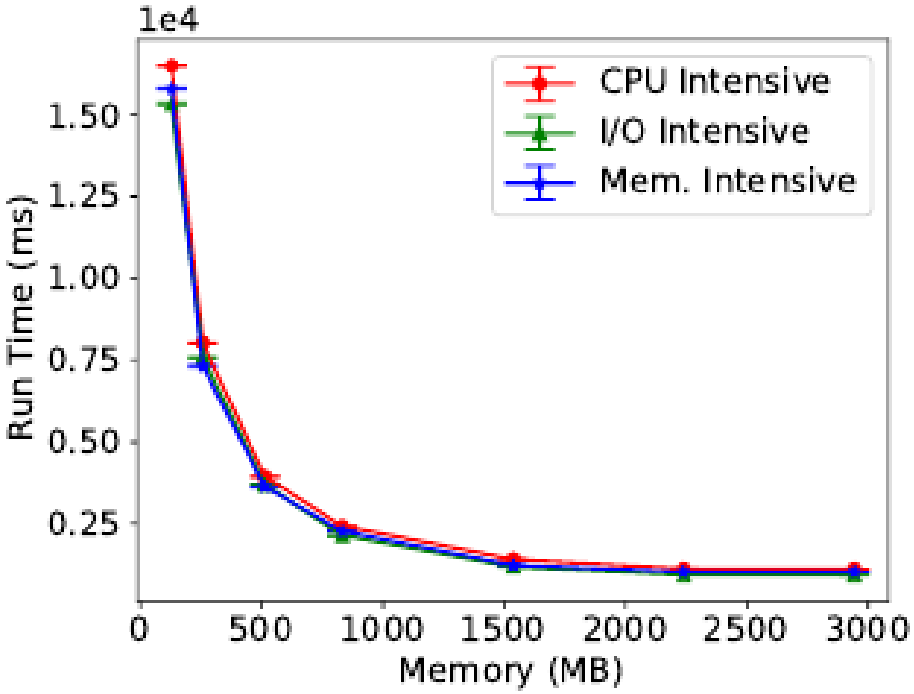
Serverless Computing: Overview

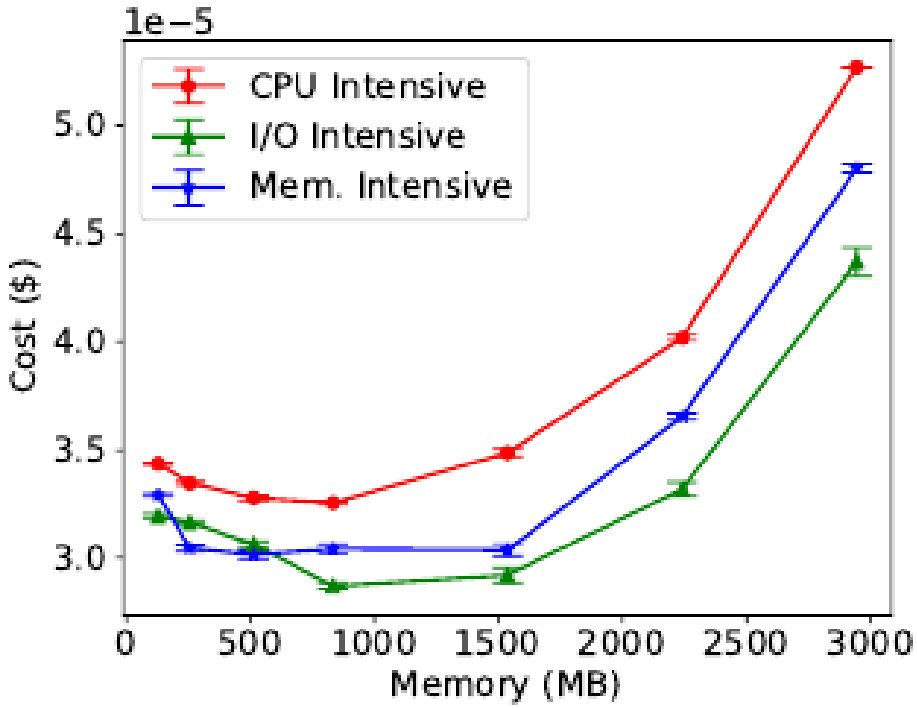
To invoke a **serverless function**, users have to specify a so-called **serverless function configuration**.

To invoke a **serverless application**, users have to specify a configuration for **all** its functions.

Serverless Computing: Overview

Configuration parameters **significantly** affect the **cost** and **response time** of serverless functions.





Serverless Computing: Problems

What current problems or limitations of the serverless computing paradigm I tried to solve?

- No support for application whose functions are hosted on multiple providers.
- No support for application whose functions have more than one implementation (**Concrete Functions**).
- Vendor lock-in issues.
- The fulfillment of quality of service (QoS) levels.

Serverless Computing: Problems

Why is so difficult to guarantee QoS levels?

- Lack of an analytical model to predict application performance.
- Lack of a methodological way to find a suitable configuration for a serverless application.

Related Works

Solutions **already** exist, but:

- They are unaware of the current status of FaaS platforms.
- No support for Multi-Provider, Multiple-Implementations serverless applications.
- Many do not support generic workload applications.

My Contributions

My contributions respect to the current state of art:

1. An **Analytical Model** to evaluate applications performance.
   - Supports generic workflow (branch, loop, parallel)
   - Multi-Provider support.
   - Support for Multi-Implementation for functions.
   - FaaS-Status-Aware.

2. A **Methodological Way** to find the "best" configuration to satisfy QoS constraints.
   - By solving an **Optimization Problem** (LP).

My Contributions

3. A custom **Heuristic Algorithm** to resolve the aforementioned optimization problem.
   - Based on **Ant Colony Optimization** algorithm (ACO) family.
4. A **QoS-Aware Software Framework** for the orchestration for multi-provider, multiple-implementations serverless applications.
5. An extension to an already existent **Representation Scheme** to define a serverless application workflow.
   - Based on an existing language called **Abstract Function Choreography Language** (AFCL).

The Prototype

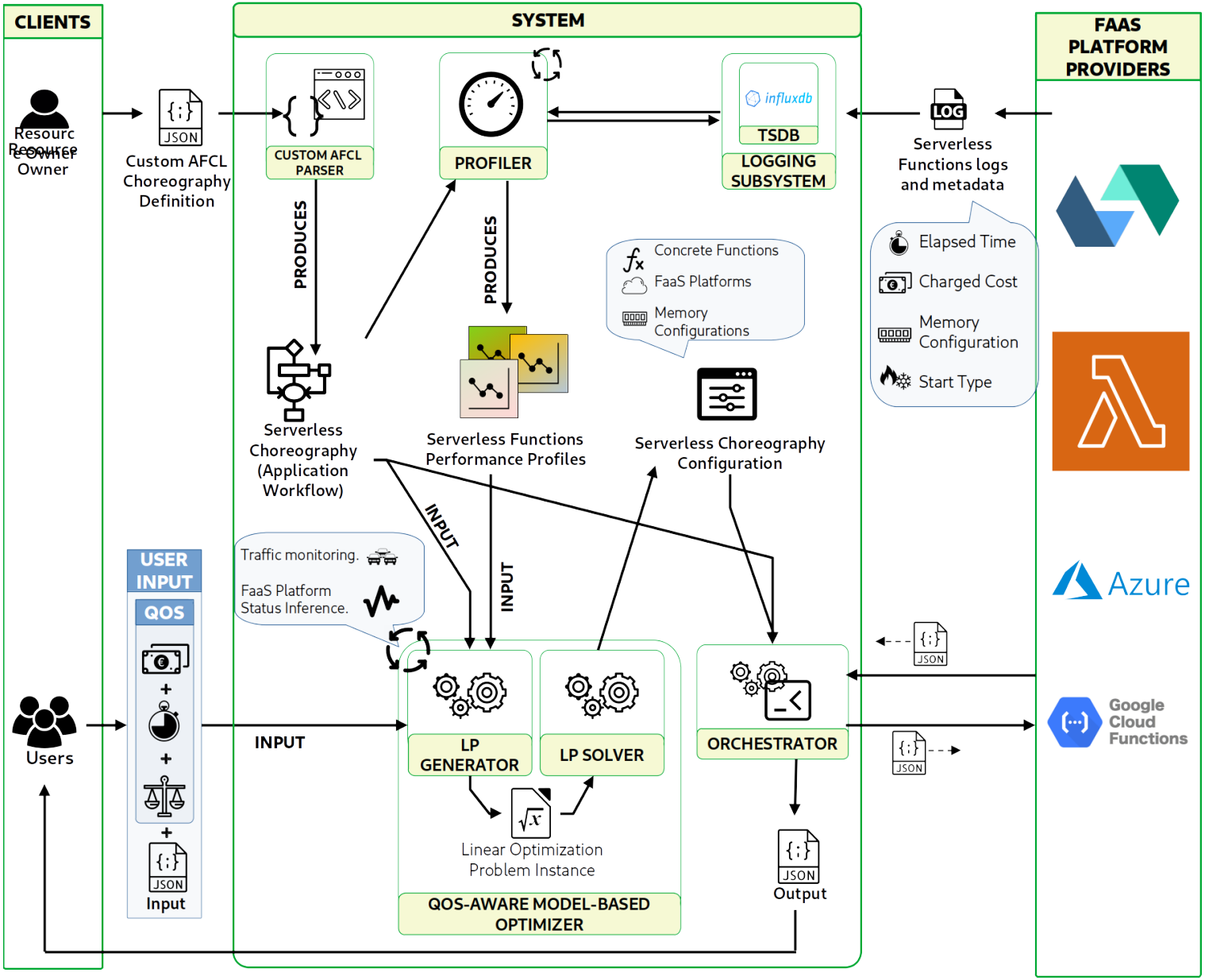
Main features of our software framework:

- **Client-Server architecture**.
- **Cloud-native** application.
- Includes a set of **adapters** to interact with following FaaS providers:
  - AWS Lambda.
  - Apache OpenWhisk.
- **REST** architectural style.

The Prototype

Main logical entities of our system:

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


CLIENTS
SYSTEM
FAAS PLATFORM PROVIDERS
Resourc
Resource
Owner
Custom AFCL Choreography Definition
JSON
CUSTOM AFCL PARSER
PROFILER
influxdb
TSDB
LOGGING SUBSYSTEM
LOG
Serverless Functions logs and metadata
PRODUCES
PRODUCES
Concrete Functions
FaaS Platforms
Memory Configurations
Elapsed Time
Charged Cost
Memory Configuration
Start Type
Serverless Choreography (Application Workflow)
Serverless Functions Performance Profiles
Serverless Choreography Configuration
INPUT
INPUT
Traffic monitoring.
FaaS Platform Status Inference.
USER INPUT
QOS
Azure
Users
INPUT
LP GENERATOR
LP SOLVER
ORCHESTRATOR
Google Cloud Functions
Linear Optimization Problem Instance
QOS-AWARE MODEL-BASED OPTIMIZER
Output
Input

The Prototype

The use of an unique representation scheme for serverless workflows **mitigates portability limitations and vendor lock-in issues**.

Serverless application are managed using identical operations based on standard `HTTP` methods providing **Access Transparency**.

The Analytical Model

Serverless applications are abstracted to a **Weakly Connected Weighted Directed Graph**.

- Each **vertex** models an **abstract serverless function**, that is a computational unit required by business logic.
  - To each abstract function corresponds a set containing all concrete function implementing it.
- Each **edge** represents the calling relationship between two abstract functions.
  - Edge weights represent the so-called **transition probability**.

The Analytical Model

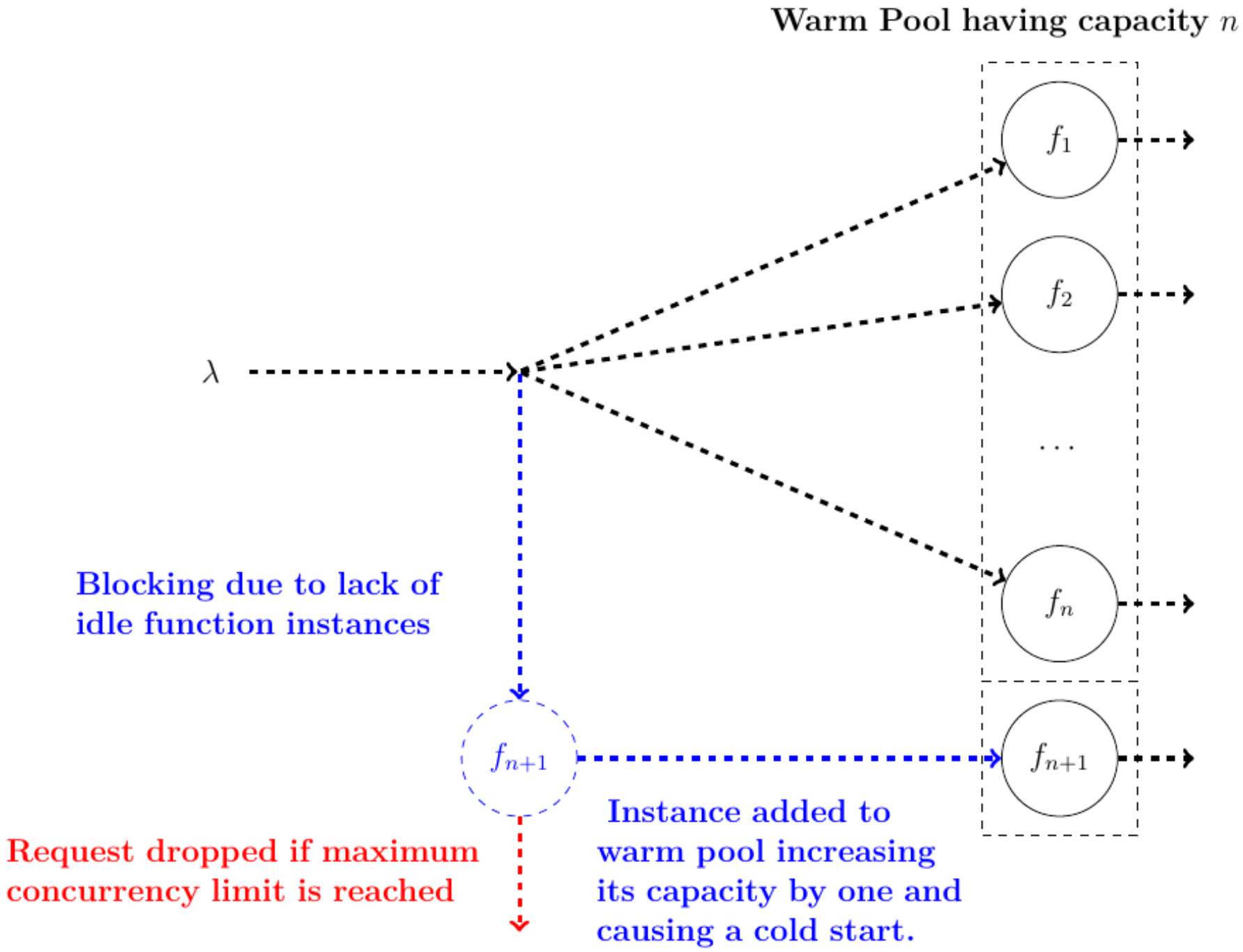
To evaluate the performance of a **Concrete Function**:

- Estimations about the **Average Response Time** (**Cost**) in case of cold (warm) start are required.
  - To compute aforementioned estimations, **Exponential Moving Average** based approach is adopted.
- An estimation of the probability according to which a request follows a cold start (**Cold Start Probability**) is required.

The Analytical Model

To compute **Cold Start Probability**:

- Any FaaS platform provider is modeled by **set** of $M/G/K(t)_{\mathbf{C}_{max}}/K(t)_{\mathbf{C}_{max}}$ queueing systems.
  - $K(t)$: the number of function instances at time t.
  - C_{max}: the concurrency limit.
- Aforementioned set depends on the concurrency model adopted by providers.
- **Erlang-B** formula is used.


Warm Pool having capacity n
f_1
f_2
λ
...
f_n
Blocking due to lack of idle function instances
f_{n+1}
f_{n+1}
Request dropped if maximum concurrency limit is reached
Instance added to warm pool increasing its capacity by one and causing a cold start.

The Analytical Model

I formalized algorithms and equations to evaluate the performance of both a **serverless application** and of an **abstract function**.

Aforementioned algorithm depends on:

- The topological properties of the graph representing the application.
- The actual value of transition probabilities of all edges.

The Optimization Problem

- To achieve our goal consisting in finding the best configuration to guarantee QoS constraints, we have to solve an **optimization problem**.
  - It is based on **Multi-Dimensional Multi-Choice Knapsack Problem Formulation**.

The Optimization Problem

Formally, the configuration vector $\mathbf{x}_{\mathscr{C}}$ we want to find is such that:

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

The Optimization Problem

The number of all possible configuration grow very fast.

- An application with:
  - 6 functions.
  - 46 possible memory choice.
  - **Only one** implementation for each function.

  lead to $9.47 \cdot 10^9$ different configurations.

- An application with:
  - 6 functions.
  - 46 possible memory choice.
  - **Two** implementation for each function.

  lead to $6.06 \cdot 10^{11}$ different configurations.

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

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

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

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

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

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

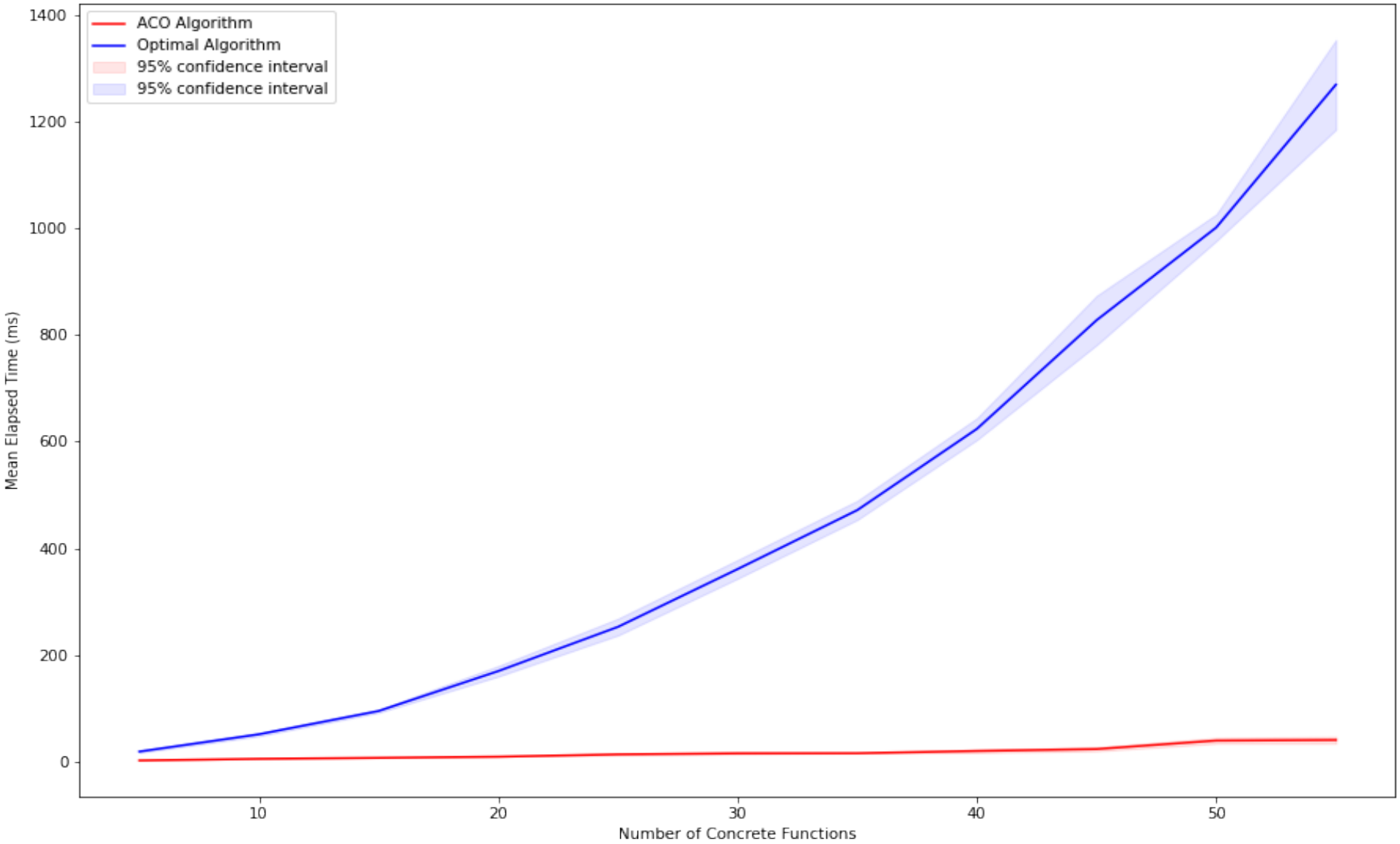
The Heuristic Algorithm

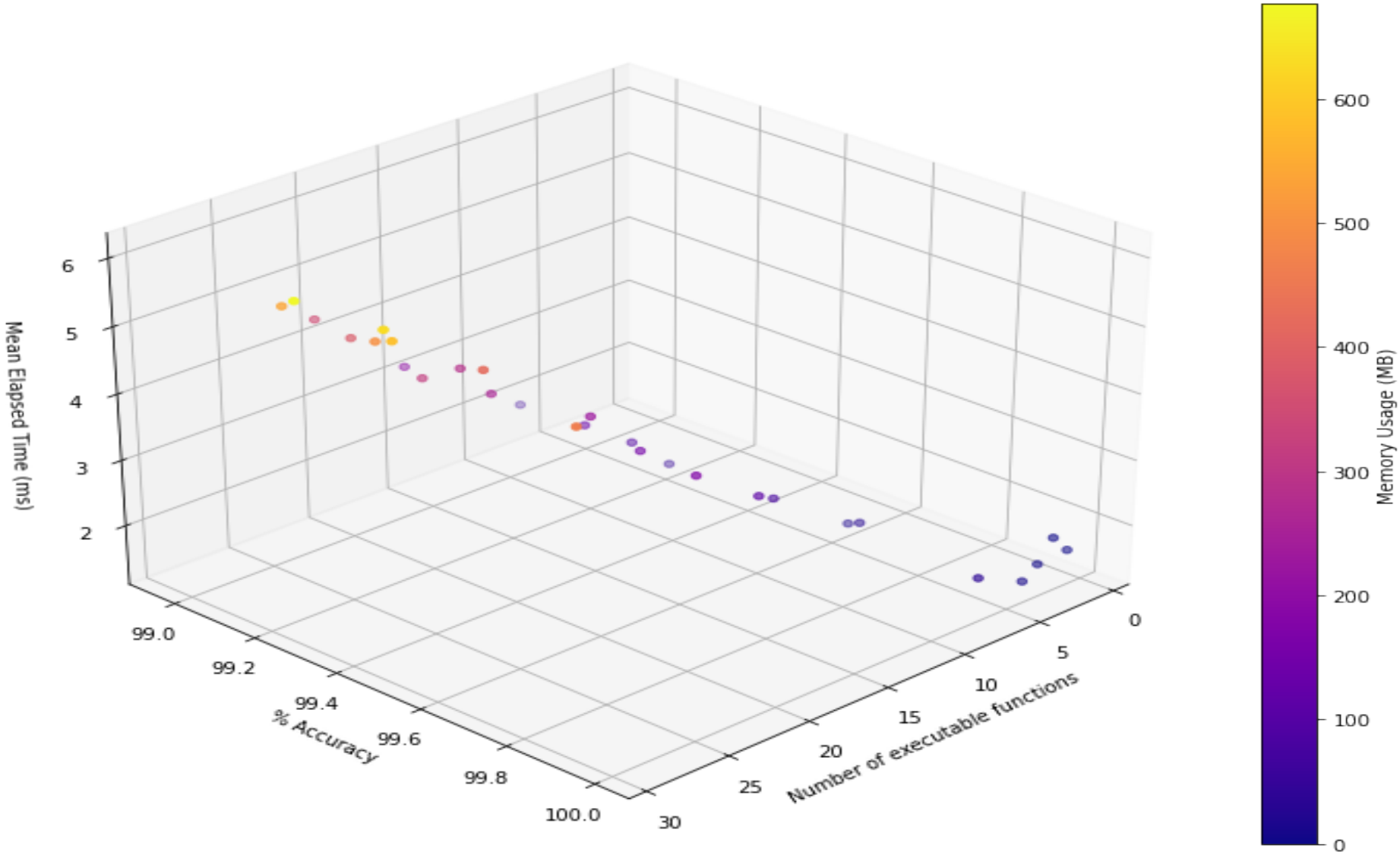
The heuristic algorithm is based on Ant Colony Optimization (ACO), a class of stochastic meta-heuristics: it is called Pre-provisioned Colony Optimization Algorithm with Lazy Pheromone Update

- It is based on a set of computational agents, called artificial ants, which **iteratively** construct a so-called partial solution.
- 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**.
  - **Pheromone Trail**.

The Heuristic Algorithm

- Lazy approach for pheromone trails update on the solution component graph.
- We exploit pre-provisioning tactic to anticipates data needs assuring a lower latency.


ACO Algorithm
Optimal Algorithm
95% confidence interval
95% confidence interval
1400
1200
1000
800
600
400
200
0
Mean Elapsed Time (ms)
10
20
30
40
50
Number of Concrete Functions


Mean Elapsed Time (ms)
6
5
4
3
2
99.0
99.2
99.4
99.6
99.8
100.0
% Accuracy
0
5
10
15
20
25
30
Number of executable functions
600
500
400
300
200
100
0
Memory Usage (MB)

Conclusions

- I presented an analytical model to compute the performance of a generic serverless workflows having multiple concrete functions hosted on multiple FaaS providers.
- I defined an optimization problems formulation to address the problem of finding a suitable configuration in order to meet user defined QoS constraints
- We developed a heuristic algorithm to rapidly solve it.
- We verified the validity of both the proposed algorithms and the analytical model through test and experimental evaluations developing a prototype supporting AWS and OpenWhisk.

Thanks for your attention!

Questions?