Università di Pisa

Department of Information Engineering

Bachelor’s Degree in Computer Engineering

Creation of a Function as a Service framework in Python for rapid testing of scheduling algorithms

Advisors
Carlo Vallati
Francesca Righetti
Giuseppe Anastasi

Candidate
Tommaso Molesti

October 9, 2025

Contents

1 Abstract

The evolution of cloud computing has consolidated the Serverless paradigm and the Function as a Service (FaaS afterwards) model as the dominant approaches for developing scalable applications. However, the performance of these systems is linked to two challenges: the latency introduced by the “cold start” phenomenon and the efficiency of scheduling algorithms in distributing the workload. Evaluating different strategies for managing containers and resources in controlled and reproducible environments is essential for optimizing these platforms.
This thesis addresses this need through the design and implementation of a FaaS framework in Python, conceived as a testbed for the comparative analysis of scheduling algorithms. The system architecture is based on a central API Gateway, developed with FastAPI, which orchestrates a cluster of nodes for the execution of functions, with which it communicates via SSH. User functions are executed as Docker containers, allowing realistic scenarios to be simulated. A key element is modularity, which allows different scheduling policies and container lifecycle management strategies to be dynamically integrated and compared, implementing and analyzing Cold, Pre-warmed, and Warmed invocation modes.
The experimental analysis conducted to test the developed system made it possible to quantify the impact of each strategy on execution latency and resource usage (CPU and RAM). The results demonstrate how warming techniques can drastically reduce response times and how the choice of a scheduling algorithm that is aware of the status of the nodes influences load balancing. The developed framework has proven to be a flexible and reproducible tool that can facilitate the study and optimization of performance in FaaS systems.

2 Introduction

2.1 The evolution of cloud computing and the Serverless paradigm

Cloud computing has revolutionized the world of IT, offering a flexible alternative to the traditional approach. In the past, to create an application, a company had to purchase and physically maintain its own servers, a costly and rigid process known as on-premise infrastructure. The cloud has introduced a new model: computing resources, such as servers, storage, and networks, have become a service accessible on demand via the Internet, with a cost model based on actual consumption (pay-per-use).
This change has proven beneficial for companies of all sizes, from startups to multinationals, thanks to its great flexibility. Instead of waiting weeks to purchase a new server, a development team can obtain a virtual one in minutes. This is possible because cloud service providers manage enormous amounts of shared resources. When a user needs a server, it is automatically assigned to them using the resources of the shared infrastructure.
The latest major step in this direction is the Serverless paradigm. The basic idea is to take abstraction to the next level: completely eliminating the need for developers to think about servers. With Serverless, code is only executed when a specific event occurs (such as a user clicking on an app). The necessary resources are created on the fly, used for as long as strictly necessary, and then deleted. This approach promises to further reduce costs and increase efficiency, leaving developers with a single task: focusing on writing the logic that makes their application work.

2.2 The Function as a Service model and how it differs from Serverless

The most common way to implement the Serverless philosophy is through the FaaS model. Well-known platforms such as Amazon AWS Lambda, Google Cloud Functions, and IBM Cloud Functions are examples of FaaS services.
With this approach, an application is no longer seen as a single large block of code that runs constantly on a server, but is broken down into many small independent "functions." Each function is a piece of code specialized in a single task, which lives inside a lightweight, isolated software container. When an event requires that task to be performed, the FaaS platform takes care of everything: it starts the container with the function, executes the code, and, once the result is obtained, shuts everything down.
The advantages of this model are:

- Speed of development: developers can write and update small functions much more quickly than an entire application.
- No server management: the responsibility for maintaining, updating, and protecting servers is entirely delegated to the service provider.
- Automatic scalability: if a function is called thousands of times per second, the platform automatically creates thousands of copies to handle the load, without any manual intervention.
- Costs: you only pay for the time the code is actually running.
- Flexibility: you can write each function in the programming language best suited to perform that specific task.

Initially, the terms FaaS and Serverless were almost synonymous. Over time, the concept of "Serverless" has expanded. Today, FaaS is considered the computing engine of the Serverless world, but it is not the only component. The term "Serverless" also includes other fully managed services, such as databases, messaging systems, or storage platforms, where the user does not have to worry about managing any servers. In summary, an entire Serverless architecture can be built by combining different FaaS functions with other managed services.

2.3 Scheduling in FaaS environments

Once a FaaS platform receives a request to execute a function, it must make a fundamental decision: on which of the many servers available in the cloud will that code actually be executed? The software component that makes this decision is called a scheduler.
The scheduler's main task is to direct each individual execution request to the most appropriate computing node at that precise moment. The choice is not random, but guided by certain objectives that determine the efficiency of the platform:

- Minimize latency: the most important goal for the end user is to get a response as quickly as possible. An effective scheduler tries to send the work to a node that can execute it quickly.
- Maximize resource utilization: from the cloud service provider's perspective, it is critical that servers are not left idle when there is work to be done, nor are they overloaded. The scheduler must intelligently balance the load to use resources efficiently.
- Fairness: in a multi-user system, the scheduler must ensure that no request is "forgotten" or penalized in favor of others.

In FaaS environments, there are also other challenges to face, the biggest of which is undoubtedly managing "cold starts." Typically, when a specific serverless function is not invoked for a certain period of time, the cloud provider destroys the environment in which it is running to optimize costs, save energy, and avoid allocating resources unnecessarily. This optimization obviously has a direct impact on performance. When a subsequent user calls that function, the platform must start a new execution environment from scratch to host it. This startup time, which is not negligible, introduces significant latency in the response, a delay that is called a "cold start."
A FaaS scheduler must therefore not only consider how busy the various servers are (in terms of CPU and RAM), but must also know the readiness status of a function on a given node. It must know whether a "warmed" container already exists and is ready for immediate execution, or whether a node is in an intermediate "pre-warmed" state, with the function image already downloaded to the cache but with the container not yet active. The choice of scheduling algorithm is a critical factor that directly impacts the performance, costs, and user experience of a FaaS platform.

2.4 Objectives

The main objective of this thesis is the design and implementation of a FaaS framework, conceived as a test bed for analyzing and comparing the performance of different scheduling algorithms in a controlled and reproducible environment.
The aim is to develop a tool that allows us to study how different resource management and load distribution strategies affect system performance.
To achieve this goal, the following sub-objectives have been defined:

- Modular framework: build a functional architecture consisting of a central gateway and distributed execution nodes, with a design that allows different scheduling strategies to be easily "assembled" and replaced, as if they were interchangeable modules.
- Quantify the impact of cold start: implement and analyze the three different container management modes (Cold, Pre-warmed, and Warmed) to measure in a concrete and numerical way the latency introduced by cold start and the effectiveness of pre-warming strategies.
- Compare scheduling algorithms: implement and compare different scheduling policies to evaluate their effectiveness in load balancing. Develop a measurement and analysis system: create an automatic mechanism for collecting key performance metrics (execution time, CPU and RAM usage) and for generating reports and graphs that facilitate the interpretation and discussion of the results obtained.

3 State of the art

3.1 Containerization technologies: Docker

A container is a standard unit of software that packages an application's code, along with all its dependencies (such as libraries and configuration files), ensuring that it runs quickly and reliably in any computing environment. The most widely used technology for creating and managing containers is Docker.
The Docker ecosystem distinguishes between two main concepts: images and containers. An image is an executable, static, self-contained package that contains all the elements necessary to run an application, such as code, runtime, and system libraries. A container is the running instance of an image, created and managed by the Docker Engine.
Containerization ensures that software behaves consistently and predictably, regardless of the underlying infrastructure. Containers isolate the application from its operating environment, ensuring consistency between development, testing, and production environments.
The container paradigm differs substantially from that of Virtual Machines (VMs). While VMs are based on hardware virtualization, requiring a complete guest operating system for each instance, containers are based on operating system-level virtualization, sharing the host system's kernel. This fundamental architectural difference makes containers lighter, more portable, and more efficient, with shorter startup times.

3.2 Performance issues in FaaS: latency and cold starts

One of the main performance issues in FaaS platforms is startup latency. Before it can process an event, the platform takes some time to initialize the function instance. This latency is not constant, but can vary significantly, from a few milliseconds to several seconds, depending on various factors.
It is necessary to distinguish between two initialization scenarios: "warm start" and "cold start." A warm start occurs when the platform reuses an instance of the function and its container, which have been kept active by a previous event. In this case, latency is minimal. The main problem lies in the cold start, which occurs when it is necessary to create a new container from scratch, start the function's host process, and load the code. This process introduces a considerable delay, which is the main concern in terms of performance.
The actual duration of a cold start can depend on many variables, some of which are under the developer's control. Among the most influential factors are the number and size of imported libraries, the complexity of the code, and the specific configuration of the function's environment, such as allocated memory.
However, one critical factor is the frequency with which cold starts occur. This is directly related to the volume and type of application traffic. An application that processes a constant stream of events will keep its instances "hot", making cold starts very rare. Conversely, a function that is invoked sporadically will almost certainly undergo a cold start with each invocation, because FaaS providers deactivate inactive instances after a few minutes to optimize resource allocation.
The impact of cold starts always depends on the context and use case. For high-volume asynchronous systems that process large amounts of data in the background, the initial latency of some invocations may be completely negligible. Conversely, for synchronous and interactive services that require low-latency responses, such as APIs for web or mobile applications, even a small delay can significantly degrade the service and therefore the user experience.
Startup latency is an important trade-off in the FaaS model. The need to test performance with realistic workloads is therefore essential to determine the feasibility of a serverless solution for a specific use case.

3.3 Status management strategies

Over time, various strategies have been developed to manage the lifecycle of execution environments (typically containers). These strategies represent different trade-offs between cost optimization and latency reduction. There are three main states in which a function instance can be found: Cold, Warm, and Pre-warmed.

3.3.1 The Cold state (Cold start)

The Cold state represents the starting condition of a function that has not been invoked recently. When a request for a function in this state arrives, the FaaS platform must perform a series of steps before it can process the event. This process includes provisioning a new container, initializing the runtime, and loading the function code and its dependencies. The entire sequence introduces significant latency, known as a "cold start." This approach is the most cost-effective, as resources are only allocated when strictly necessary and you do not pay for periods of inactivity, but it offers the worst performance in terms of latency.

3.3.2 The Warm state (Hot start)

After serving an invocation, the FaaS platform may decide to keep the execution environment active for a short period of time, waiting for subsequent requests. An instance in this condition is in the Warm state. If a new request arrives while the instance is still "warm", the platform reuses it, skipping the entire initialization process and moving directly to code execution. This results in very low latency, a phenomenon known as a "warm start." However, the duration of this state is not guaranteed. After a period of inactivity that can vary, the provider deallocates the instance to free up resources, returning it to the Cold state.

3.3.3 The Pre-warmed state

For latency-sensitive applications where cold starts are unacceptable, a third strategy known as "pre-warming" has been introduced. This strategy ensures that a specific number of instances of a function are initialized and kept constantly in the Warm state, even before the first request arrives. This way, when traffic arrives, it finds a pool of environments ready to respond, effectively eliminating the possibility of a cold start. This strategy ensures maximum performance and predictable latency, but introduces an additional cost: you pay to keep instances active, regardless of whether they are processing requests or not.

3.4 Scheduling algorithms in distributed systems

Scheduling in a distributed system consists of assigning processes to the various available computing nodes. The most common and relevant approach for this context is load balancing, whose purpose is to distribute the workload among the nodes to maximize the overall throughput of the system. The goal is to keep each node equally busy, avoiding some being overloaded while others remain idle.
Load balancing algorithms are mainly divided into two categories:

- Static Algorithms: load distribution occurs without taking into account the current state of the system. The decision is made a priori. The main advantage of these algorithms is their simplicity.
- Dynamic Algorithms: they make decisions based on the current load of each node. They dynamically redistribute work from overloaded nodes to less busy ones, offering superior performance, especially when task execution times are highly variable. Although more complex to design, they are usually more efficient.

Another difference concerns the decision-making architecture, which can be centralized or distributed.

- Centralized Architecture: a single node is responsible for all allocation decisions. This approach is efficient because all information about the system status is concentrated in a single point, but it has lower fault tolerance: if the central node fails, the entire system crashes.
- Distributed Architecture: the scheduling logic is distributed among the various nodes of the system, which collaborate to make decisions. This model is more resilient and has no single point of failure, but it can introduce greater complexity and communication overhead.

In this project, a centralized architecture was chosen, where the API Gateway is the single node responsible for all decisions.

4 Framework design and architecture

4.1 Requirements

To meet the main objective of the project, a series of functional and non-functional requirements that the framework must possess have been defined. These ensure that the system is not only functional, but also robust, easy to use for experimentation, and open to future expansion.

4.1.1 Functional requirements

They describe the specific operations that the system must be able to perform.

- Function and node management: the framework must expose APIs to allow dynamic registration of both the functions to be executed and the compute nodes that make up the cluster. The gateway must maintain a registry of these entities in order to orchestrate executions.
- Configurable scheduling: it must be possible to implement and select different scheduling policies without having to modify the core logic of the gateway.
- Support for invocation states: the system must explicitly implement and manage the three main invocation modes (Cold, Pre-warmed, and Warmed) to quantify the impact of the "cold start."
- Metrics collection: the collected data must be saved in easily analyzable formats, such as text tables and graphs.

4.1.2 Non-functional requirements

They define the qualities of the system, influencing the user experience and maintainability.

- Modularity and extensibility: the architecture must be built in such a way that each part performs a specific task, thus facilitating the modification or addition of new features without impacting the rest of the system. Adding a new scheduling policy or node selection strategy should not require invasive changes to the existing code, but should be limited to implementing a new class that complies with a predefined interface.
- Reproducibility: The entire environment, consisting of the gateway, nodes, and client, is defined in code using Docker. A startup script "cleans up" the system state before each execution, stopping and removing residual containers from previous executions to ensure an identical starting point for each test.
- Portability: thanks to the use of containerization, the framework must be agnostic with respect to the underlying operating system. The entire infrastructure can be run on any machine on which Docker Engine is installed, ensuring portability between different development and testing environments.

4.2 General architecture

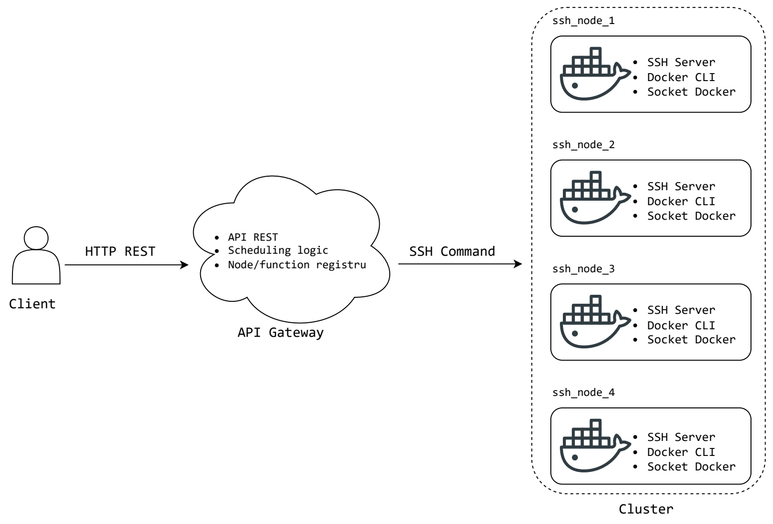

Figure 1: General architecture diagram of the framework

The framework architecture was designed following a distributed microservices model, with the aim of ensuring modularity, extensibility, and reproducibility of experiments. The system consists of several containerized entities that work together to orchestrate the recording and execution of functions. Each component has a specific and well-defined role, contributing to the creation of a controlled and flexible testing environment.
The heart of the system is the API gateway, a web server developed with the FastAPI framework in Python. It acts as the entry point and brain of the infrastructure. All operations, from registering a new function or node to requesting execution, pass through its REST APIs. The gateway maintains the state of the system, keeping track of active nodes and available functions. Its main responsibility is to apply the configured scheduling policy to decide, for each incoming request, which execution node is best suited to process it, completely abstracting this logic from the client.
Execution nodes are distributed components that physically execute function code. Each node is a Docker container based on a minimal Ubuntu image, running an SSH server to accept commands from the gateway. A key feature is that these nodes mount the host system's Docker socket. This allows them, even though they are isolated containers, to orchestrate the startup and management of other containers on the same host machine. When the gateway selects a node, it sends it the Docker commands necessary to execute the requested function via SSH.
The client is not an end user, but a Python script designed to orchestrate test sessions automatically. Its task is to simulate a workload by sending requests to the gateway. Typically, the client performs a cycle of operations: first, it registers the execution nodes and functions that will be tested, then it sends a predefined number of invocation requests sequentially. The client operates with complete transparency with respect to the status of the set of execution nodes: it simply invokes a function by its name, without knowing where or how it will be executed.
To simulate realistic test scenarios and quantify the impact of cold starts, two custom Docker images were created for the functions. The first, `custom_python_heavy`, is a "heavy" image that includes Python scientific libraries such as pandas, scikit-learn, and tensorflow, characterized by significant startup time and memory footprint. The second, `custom_python_light`, is a "lightweight" image with minimal dependencies, representing a leaner workload. Using these two images allows us to evaluate the behavior of scheduling algorithms under different operating conditions and measure the impact of image size on execution latency.

4.3 Communication protocol

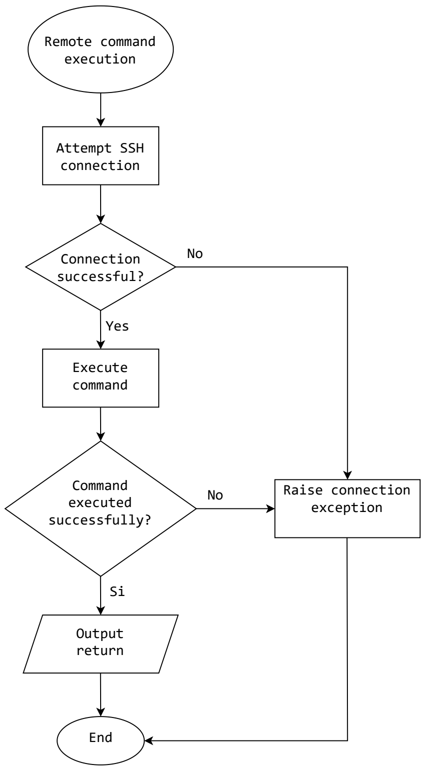

Figure 2: Flowchart for executing a remote command

The SSH (Secure Shell) protocol was chosen for communication between the API gateway and the distributed execution nodes. This choice was motivated by the standard nature of SSH in Linux environments and its ability to provide an encrypted channel for executing commands on remote machines. All that is required is for an OpenSSH server to be running on each execution node, as configured in the respective Dockerfile.
To implement this protocol in the gateway, the Python library `asyncssh` was used, as it integrates natively with Python's `asyncio` framework, the same one on which FastAPI is based. This is important for the gateway's performance. Since remote operations such as downloading a Docker image (`docker pull`) or executing a function can take a significant amount of time, the asynchronous approach allows the gateway to handle multiple connections and SSH commands concurrently, without ever blocking its main loop. This way, the gateway remains responsive and can continue to process other incoming requests while waiting for a remote operation to complete.
The communication flow is managed by the `run_ssh_command` function. When the gateway needs to send a command to a node, this function establishes an SSH connection using the credentials provided when the node was registered. Since the execution nodes are temporary containers, known host checking (`known_hosts=None`) has been disabled to simplify and speed up the SSH connection. This avoids the complexity associated with managing SSH keys. Once the connection is established, the command is executed on the remote node, and its output is captured and returned to the gateway. This mechanism gives the gateway the full control it needs to orchestrate the lifecycle of function containers across the infrastructure.

4.4 Modular policy design

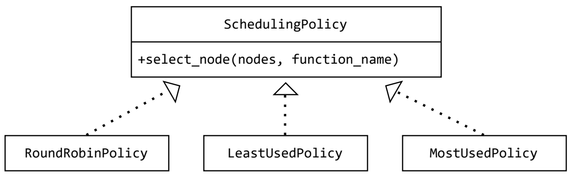

Figure 3: Class diagram for scheduling policies based on the Strategy design pattern

One of the objectives of this thesis is to create a framework that acts as a testbed for analyzing and comparing the performance of different scheduling algorithms. To meet this requirement, the architecture must allow different scheduling strategies to be "mounted" and replaced quickly and easily. This principle is the basis of the non-functional requirement of modularity and extensibility defined above. The design of the policies must therefore allow anyone who wants to extend this project to implement a new algorithm and integrate it into the system with minimal effort, without having to modify the central logic of the gateway.
To achieve this level of flexibility, the architecture adopts the "Strategy" design pattern. This software design pattern allows you to define a family of algorithms, encapsulate each of them in a separate class, and make them interchangeable. In the context of our framework, each specific scheduling algorithm is implemented as an independent "strategy." The gateway can be configured to use any of these strategies without altering its internal operation.
The core of this design is the definition of a common interface that each policy class must comply with in order to be considered valid and interchangeable. Although Python does not require explicit interfaces like other languages, the contract is defined by the signature of each policy's main method: `async def select_node(nodes: Dict[str, Any], function_name: str)`. This method accepts as parameters the list of available execution nodes and the name of the function to invoke. Its sole responsibility is to execute the specific logic of the algorithm it implements and return the name of the node it has selected as most suitable for execution.
Integrating this design into the gateway is simple and effective. When the server starts, a concrete policy is instantiated in the variable `DEFAULT_SCHEDULING_POLICY`, and its object is maintained for the duration of the session. The logic that manages invocation requests is completely decoupled from the specific algorithm: it simply invokes the `select_node` method on the `DEFAULT_SCHEDULING_POLICY` variable, which contains the instance of the scheduling policy chosen when the gateway was started.
The advantages of this approach:

- Extensibility: to test a new scheduling algorithm, simply create a new class that complies with the `select_node` interface. No changes to the gateway's core code are required.
- Maintainability and testability: the logic of each algorithm is isolated in its own class, making the code cleaner, easier to maintain, and allowing each strategy to be tested independently.
- Experimental flexibility: changing the scheduling algorithm for an entire test session boils down to modifying a single line of code at the point where the policy is instantiated.

5 Implementation details

5.1 Gateway implementation

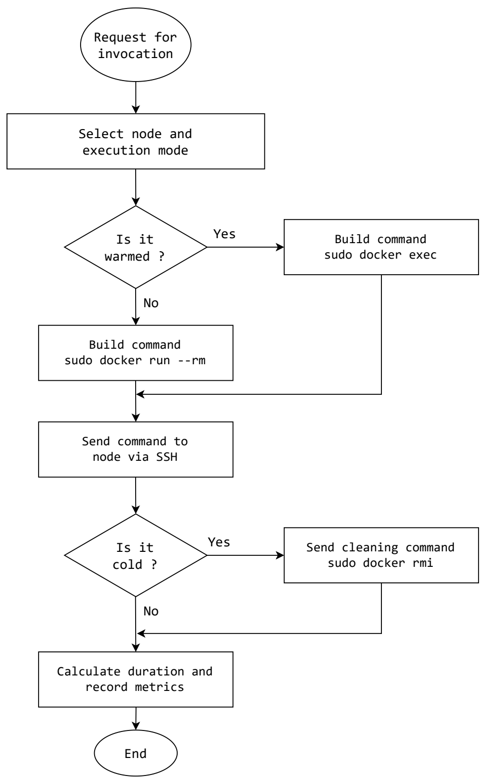
Request for invocation
Select node and execution mode
Is it warmed ?
Yes
Build command sudo docker exec
No
Build command sudo docker run --rm
Send command to node via SSH
Is it cold ?
Yes
Send cleaning command sudo docker rmi
No
Calculate duration and record metrics
End

Figure 4: Flowchart of function call request management

The gateway is the brain of the entire framework. It is an asynchronous web application built using FastAPI. It has three main responsibilities: exposing REST APIs for system management, maintaining the current state of registered nodes and functions, and orchestrating each individual invocation by applying scheduling policies.
State management is implemented through simple in-memory data structures defined in the `state.py` module. The `function_registry` and `node_registry` dictionaries contain information about registered functions and nodes, while `function_state_registry` keeps track of the state (e.g., warmed, pre-warmed) of a function on a given node. To ensure the persistence of results between different testbed executions, the gateway implements a `startup_event` function which, when the server starts up, checks for the presence of a pre-existing `metrics.csv` file and, if present, loads its contents into the `metrics_log` list in memory.
The gateway interface is defined by three main API endpoints.
The first two, `/nodes/register` and `/functions/register`, manage component registration. While node registration simply saves credentials, function registration triggers a more complex process: after saving the function data (Docker image and command), it immediately invokes the `apply` method of `SCHEDULING_POLICY`. This step is crucial because it is here that the warming strategy (if configured) is applied based on the global variable `WARMING_TYPE`, for example, by starting the preloading of the image or the creation of a persistent container.
The most important endpoint is `/functions/invoke/{function_name}`, which orchestrates the execution of a function.
The flow is as follows:

- Node selection: the `NODE_SELECTION_POLICY.select_node` method is invoked, which implements the high-level strategy (e.g., giving priority to nodes with warmed containers). This call returns the name of the selected node and the actual execution mode (Cold, Pre-warmed, or Warmed).

- Command construction: based on the execution mode returned above, the gateway dynamically constructs the Docker command to be executed. If the mode is Warmed, a `sudo docker exec` command is assembled to execute the code on an already active container. For Cold and Pre-warmed modes, a `sudo docker run --rm` command is constructed to start a temporary container.

- Remote execution: the command is sent to the selected node via the `run_ssh_command` function. The text output returned by the function (a confirmation string "Ok") is captured and recorded in the gateway logs for debugging purposes. Printing the calculation result is avoided, as it could be very long and clog up the terminal output.

- Cleaning the image after Cold invocations: once a Cold execution has finished, the gateway performs an additional step to ensure the purity of future tests. A `sudo docker rmi` command is sent to remove the Docker image just used from the node. This methodological choice is useful to ensure that any subsequent Cold invocation on the same node is forced to re-download the image, preventing execution times from being altered by a cached version.

- Metrics: once the response is obtained, the gateway calculates the total execution time and invokes the `log_invocation_metrics` function to record all the metrics of the call (node used, execution mode, time taken, CPU/RAM usage) in the `metrics.csv` file.

5.2 Execution node structure

The execution nodes are the framework's workers, responsible for the actual execution of the function code. Each node is an independent containerized instance, defined by a Dockerfile and orchestrated by the docker-compose.yml file. To ensure a clean and standardized environment, each node is based on an official Ubuntu image.
Within this minimal environment, the software packages essential for its operation are installed: the OpenSSH server, which allows the gateway to connect and send commands, and the Docker suite (`docker.io`), which provides the command line interface for managing containers. An important architectural choice is to mount the host system's Docker socket inside each node. This allows a container (the node) to communicate with the host's Docker daemon and to start, stop, and manage other "sibling" containers on the same machine. This avoids the complexity of having to run a Docker daemon inside each node.
The runtime configuration of each node is managed by the `entrypoint.sh` script.
When launched, this script performs a series of operations:

- SSH user creation: a non-privileged user, `sshuser`, is created with a default password. This user is used by the gateway for all SSH connections, avoiding the use of the root user.

- Docker permissions configuration: the user `sshuser` is added to the system's docker group. To overcome any permissions issues in test environments, `sudo` access without a password is configured specifically for the `/usr/bin/docker` command.

- Metrics script generation: the script creates an executable file in `/usr/local/bin/get_node_metrics.sh`. This script is designed to read resource usage information directly from the Linux kernel system files located in `/sys/fs/cgroup/`. It can automatically detect the version of `cgroups` (v1 or v2) in use on the host, ensuring portability. Its task is to calculate the percentage CPU and RAM usage of the node itself and return the data in a JSON format that is easily interpreted by the gateway.

To ensure reproducibility and control of experiments, strict resource limits are imposed on each execution node via the `deploy` configuration in the `docker-compose.yml` file, limiting the maximum CPU and RAM usage.

5.3 Implementation of scheduling policies

Scheduling policies are at the heart of the framework's decision-making logic. As mentioned in the chapter on design, each policy is a separate Python class that implements the logic for selecting the most suitable node for executing a function. All policies are defined in the `policies.py` file and share a security mechanism: before considering a node for execution, they check its RAM usage, discarding it if it exceeds the `RAM_THRESHOLD` threshold defined in `state.py`. This condition prevents work from being assigned to nodes that are already overloaded.

5.3.1 RoundRobinPolicy

The `RoundRobinPolicy` is the implementation of a static scheduling algorithm, whose goal is to distribute the load evenly and cyclically among all available nodes, without considering their current workload.

- Logic: the class uses `itertools.cycle` to create an infinite iterator that loops through the list of registered nodes. To dynamically handle the addition or removal of nodes during test execution, the policy maintains a local cache of node names (`_nodes_cache`). At the beginning of each call to `select_node`, it compares the current list of nodes with its cache. If it detects a difference, it recreates the cyclic iterator with the new list of nodes, ensuring that the system is always up to date.
- Selection process: when it needs to select a node, the policy extracts the next element from the iterator. At this point, it contacts the chosen node to retrieve its CPU and RAM metrics using the `get_metrics_for_node` function. If RAM usage is below the threshold, the node is returned as the winner. Otherwise, it is discarded and the process continues with the next node in the iterator until a suitable candidate is found or all available nodes are exhausted.

5.3.2 LeastUsedPolicy

The `LeastUsedPolicy` implements a dynamic and informed scheduling algorithm. Its purpose is to actively balance the cluster load by sending new requests to the node that, at that precise moment, has the lowest CPU load.

- Logic: this policy requires a comprehensive and up-to-date view of the status of all nodes before a decision can be made. To obtain this information, it uses an auxiliary method `_get_all_node_metrics`. This method creates a list of `asyncio` tasks, one for each node to be queried, and executes them in parallel using `asyncio.gather`. This approach drastically reduces waiting time, as SSH calls for retrieving metrics are executed simultaneously rather than sequentially.
- Selection process: once the metrics have been obtained from all nodes, the policy filters the list to keep only the "eligible" nodes, i.e., those that meet the RAM constraint. If there are no eligible nodes, the selection fails. Otherwise, it applies Python's `min()` function to the remaining candidate group, using the value of the `cpu_usage` field as the comparison key. This identifies and returns the node with the lowest CPU usage value.

5.3.3 MostUsedPolicy

The `MostUsedPolicy` is the opposite of the `LeastUsedPolicy`. It is a dynamic algorithm designed to select the node with the highest CPU load, provided that its RAM is below the threshold. Although counterintuitive for traditional load balancing, this policy has been implemented for experimental purposes:

- Allows you to study the behavior of the system when the load is deliberately concentrated on a few nodes.
- It is useful for testing strategies where the goal is to saturate the resources of one node before moving on to the next, for example to optimize energy costs.

- Logic and selection process: its implementation is identical to that of `LeastUsedPolicy`. It uses the same mechanism based on `asyncio.gather` for parallel metric collection and the same RAM filter. The only difference is in the final line of the selection, where the `max()` function is used instead of `min()` to identify and return the node with the highest `cpu_usage` value.

5.4 Implementation of invocation strategies

While scheduling policies handle load balancing between nodes, the framework implements higher-level logic to manage invocation strategies, with the goal of minimizing latency, especially that resulting from cold starts. This logic is managed by a series of classes, defined in `policies.py`, which implement a Chain of Responsibility design pattern.
Node selection is not performed directly by a load balancing policy, but by a selection meta-policy defined in `main.py` by the variable `NODE_SELECTION_POLICY`. This class is the first link in the chain and initiates a hierarchical selection process. The goal is to find a node that can execute the function with the lowest possible latency.
These are the steps in the process:

- `WarmedFirstPolicy`: this is the first link in the chain. Its responsibility is to check whether a container already exists in the Warmed state for the requested function. It queries the `function_state_registry` dictionary. If it finds a node on which the function is marked as Warmed, it selects it and returns it, interrupting the chain. This guarantees absolute priority to executions with minimum latency (`docker exec`). If no warmed container is found, the policy does not make a decision, but delegates the request to the next link in the chain, i.e., `PreWarmedFirstPolicy`.

- `PreWarmedFirstPolicy`: this second ring is activated only if the previous one has failed. Its logic is similar to the above: it checks the `function_state_registry` for a node on which the function is in the Pre-warmed state. This state indicates that the function's Docker image has already been downloaded to the node (`docker pull`), ensuring a faster startup (`docker run`) than a full cold start. If it finds a node that meets this condition, it selects it and returns it. Otherwise, this policy also fails and delegates responsibility to the last link in the chain, `DefaultColdPolicy`.

- `DefaultColdPolicy`: this is the final link in the chain and acts as a fallback. It does not perform any checks on the status of the containers. Its function is to invoke the basic scheduling policy (e.g., `RoundRobinPolicy`, `LeastUsedPolicy`, or `MostUsedPolicy`), contained in the variable `DEFAULT_SCHEDULING_POLICY`. The latter will then select a node based on its own algorithm. This is the strategy that leads to Cold execution, where there is no guarantee that the image is already present on the node.

5.5 State management and invocation flow

This chapter provides an overview of the complete path of a function within the framework. The life cycle of a function is divided into two phases: an initial phase of registration and state preparation (which occurs only once) and a second phase of invocation and execution (which is repeated for each request).

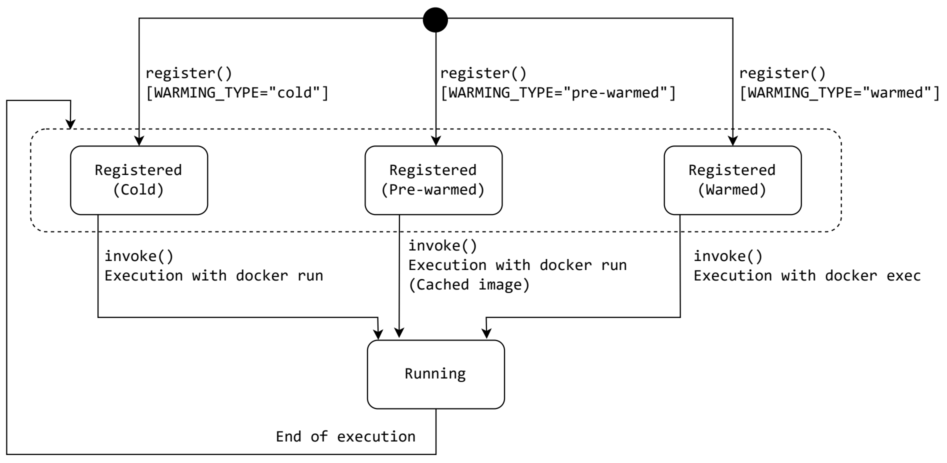

Figure 5: State diagram of the life cycle of a function in the framework

5.5.1 Registration and preparation of the status

The lifecycle of a function begins when the client sends a request to the gateway's `/functions/register` endpoint. This is not a simple data save, but an active process that defines the initial state of the function in the cluster, based on the gateway's global `WARMING_TYPE` configuration. The `StaticWarmingPolicy` class interprets this configuration and prepares the nodes accordingly.
Three scenarios are possible:

- Warmed configuration: if the framework is in Warmed mode, the `StaticWarmingPolicy` selects a node (using the default scheduling policy) and sends it the command to start a persistent container (`sudo docker run -d ... sleep infinity`) via SSH. Once the operation is complete, the status of the function on that node is recorded as Warmed in the `function_state_registry`.
- Pre-warmed configuration: the policy selects a node and sends it the command to download the Docker image of the function (`sudo docker pull`). The image is then stored in the node's cache, and the status of the function is recorded as Pre-warmed.
- Cold configuration: no action is taken on the execution nodes. The gateway simply records the existence of the function in the `function_registry`. No information is added to the `function_state_registry`, indicating that there are no pre-warmed instances.

5.5.2 Invocation and execution

When the client sends a request to the `/functions/invoke` endpoint, the execution phase begins. At this point, the cluster state has already been prepared, and the gateway's task is to choose the most efficient execution path possible, following the chain of responsibility logic described above (`WarmedFirstPolicy`).
The result is one of the following execution paths:

- Warmed path (minimum latency): `WarmedFirstPolicy` finds a Warmed instance in the `function_state_registry`. the gateway constructs a `sudo docker exec` command and sends it to the corresponding node. This is the fastest path overall, because it skips both container creation time and image download time.
- Pre-warmed path (reduced latency): if no Warmed instances exist, the `PreWarmedFirstPolicy` finds a Pre-warmed instance. The gateway constructs a `sudo docker run --rm` command. Execution is slower than the Warmed path because it requires the creation of a new container, but it is still faster than a full cold start because it avoids the image download time, which is already present in the node's cache.

- Cold path (maximum latency): if none of the previous checks are successful, the `DefaultColdPolicy` selects a node using the load balancing policy. The gateway sends a `sudo docker run --rm` command. This is the situation that generates the highest latency, since the execution time includes both the creation of the container and the entire time it takes to download the Docker image from the registry.

6 Experimental analysis and results

6.1 Test methodology and environment

To ensure the validity and reproducibility of the results, all experiments were performed following a rigorous methodology and in a well-defined hardware and software environment. Each test session was automatically orchestrated by the `start.py` script, which ensures an identical starting point for each execution. Before starting the infrastructure, the script performs a complete cleanup of the Docker environment, removing any containers and images left over from previous executions, and then starts the entire system using `docker-compose up`.

6.1.1 Hardware and software

The tests were performed on a MacBook Pro computer equipped with an Apple M4 processor, 16 GB of RAM, and macOS Tahoe operating system. The containerization environment was managed by Docker Engine version 28.4.0, with Docker Compose version 2.39.2. The entire framework code, including test functions and orchestration scripts, was executed using the Python 3.13.7 interpreter.

6.1.2 Framework configuration

The test infrastructure defined in the `docker-compose.yml` file consists of a central API Gateway and four execution nodes. To simulate an environment with limited resources and to ensure consistency of results across different tests, stringent hardware limitations were imposed on each execution node: a maximum of 0.50 CPU cores and 512 MB of RAM. For each experimental scenario, the client was configured to perform a total of 100 invocations for each registered function, in order to collect a sufficiently large data sample for statistical analysis.

6.1.3 Testing and workload functions

To simulate heterogeneous workloads and isolate the impact of different latency components (initialization vs. execution), two test functions were defined:

- Function `fibonacci-image-big-func-small`: this function associates the "heavy" image (`custom_python_heavy`) with a light computational load (`COMMAND_LIGHT`). The aim is to maximize the impact of the cold start, making the container download and startup time the dominant component of the total latency.
- Function `fibonacci-image-small-func-big`: this function associates the "light" image (`custom_python_light`) with a heavy computational load (`COMMAND_HEAVY`). In this scenario, the initialization overhead is minimal, and the measured latency is almost entirely attributable to the actual execution time of the code on the CPU.

The actual computational load of both functions is a calculation of the Fibonacci series. The complexity of the calculation is defined in the client using two separate commands, `COMMAND_LIGHT` and `COMMAND_HEAVY`, which pass the number of iterations as a command line argument to the `loop_function.py` script executed within the container.

6.2 Evaluation metrics

To conduct a quantitative and objective analysis of the different scheduling and invocation strategies, a set of metrics was defined. These metrics are automatically collected by the framework for each individual function invocation and recorded in a CSV file (`metrics.csv`), allowing for detailed analysis and graph generation. The metrics were divided into a primary metric, which evaluates the performance perceived by the user, and two secondary metrics, which describe the internal state of the system and the effectiveness of load balancing.

6.2.1 Primary metric: execution latency

Execution latency (execution time) is the main performance indicator of the framework. It is defined as the total time interval between the moment the gateway receives an invocation request and the moment the execution of the function on the remote node is completed. This metric is critical because it represents the end-to-end latency of the service, i.e., the actual delay that an end user or another service would experience.
The analysis of this metric has two objectives:

- Compare the average performance of the different strategies by calculating the arithmetic mean of the execution times for each test scenario.
- Analyze the statistical distribution and consistency of response times using box plots, which allow you to view the median, quartiles, and any outliers.

6.2.2 Secondary metrics: resource usage

To evaluate the effectiveness of scheduling algorithms in distributing the workload, two secondary metrics were defined that describe the status of a node at the time of selection:

- CPU usage (%): indicates the percentage of load on a node's CPU, measured immediately before the scheduler assigns it the execution of the function.
- RAM usage (%): indicates the percentage of RAM occupied on the node, measured immediately before the task is assigned.

This data is collected via a script executed on each node, which reads information directly from the Linux kernel's `cgroups` file system to obtain an accurate measurement of the resources consumed by the node's container. These metrics do not directly measure the performance of a single invocation, but they are important for verifying whether a scheduling policy is effectively balancing the load efficiently.

6.3 Test scenarios and results

To fully evaluate the framework's performance, two main test scenarios were defined. The first scenario focuses on analyzing the impact of different invocation strategies (Cold, Pre-warmed, Warmed) on latency. The second scenario compares the effectiveness of scheduling policies (`RoundRobinPolicy`, `LeastUsedPolicy`, `MostUsedPolicy`) in balancing the workload between nodes. Each of these scenarios was run for both test functions (`fibonacci-image-big-func-small` and `fibonacci-image-small-func-big`) in order to analyze the behavior of the system under both cold start and computational load conditions. For each scenario, the results obtained will be presented separately for each function.

6.3.1 Scenario 1: Analysis of invocation strategies

- Objective: the purpose of this experiment is to measure and quantify the impact of cold start on execution latency by comparing the performance of the three container management states implemented in the framework: Cold, Pre-warmed, and Warmed.
- Configuration: to isolate the effect of the invocation strategy, the scheduling policy was kept constant for all tests, using `RoundRobinPolicy`. Three separate and independent test sessions were performed for each of the two functions, modifying the configuration variable `WARMING_TYPE` in the gateway each time.
- Expectations: the initial hypothesis is that there is a statistically significant difference between the latencies of the three modes. The Cold mode is expected to have the highest average latency and the greatest variability, due to the time required to download the image and initialize the container. The Pre-warmed mode should show significantly lower latency, eliminating the image download time. The Warmed mode should guarantee the lowest and most consistent latency of all, as execution takes place on an already active container.
- Results: the data collected confirms the expectations, highlighting the impact of warming strategies on latency. The analysis is performed separately for the two functions, as the results show significant differences related to the nature of the workload.

Function `fibonacci-image-big-func-small` (Cold Start dominant). As shown in graphs 6 and 7, latency in this scenario is dominated by the initialization time of the heavy image. The average execution time in Cold mode is 6.542 seconds. Switching to Pre-warmed mode, which eliminates only the image download time, reduces latency to 5.4022 seconds, an improvement of 17.42%. Warmed mode, which also eliminates container startup time, further lowers latency to 5.2647 seconds, resulting in a 19.52% speed increase over Cold mode. The box plot shows greater variability and the presence of outliers in Cold mode, compared to a much more compact and predictable distribution for Warmed mode.

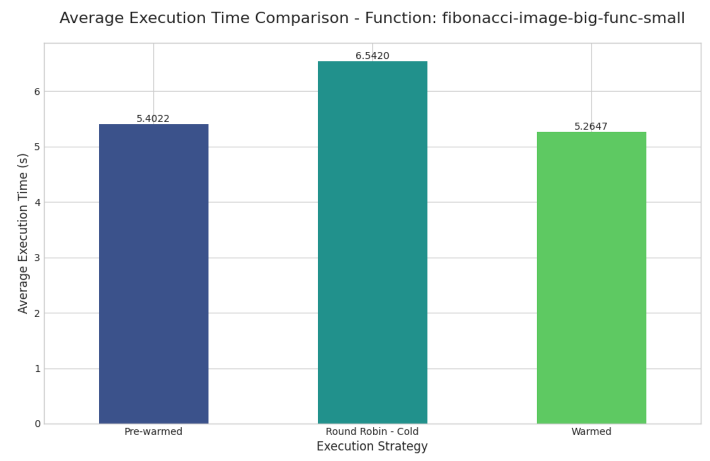

Figure 6: Comparison of average execution time (s) for the image-heavy function

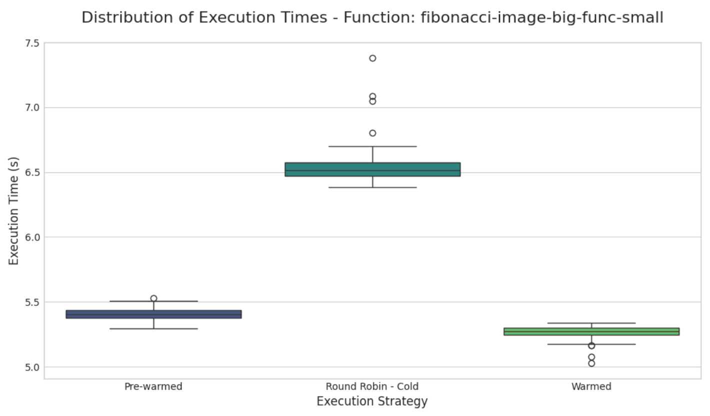

Figure 7: Distribution of execution times (s) for the image-heavy function

Function `fibonacci-image-small-func-big` (dominant CPU load). In this scenario, where initialization overhead is minimal and latency is mainly determined by long computation times, warming strategies show a positive but lower percentage impact. As shown in graphs 8 and 9, the average time in Cold mode is 13.3288 seconds. Pre-warmed mode reduces the time to 12.2149 seconds (an improvement of 8.36%), while Warmed mode drops to 12.0481 seconds, with an overall improvement of 9.61% compared to Cold mode. Again, Warmed mode is not only the fastest, but also the one with the most predictable and consistent time distribution.

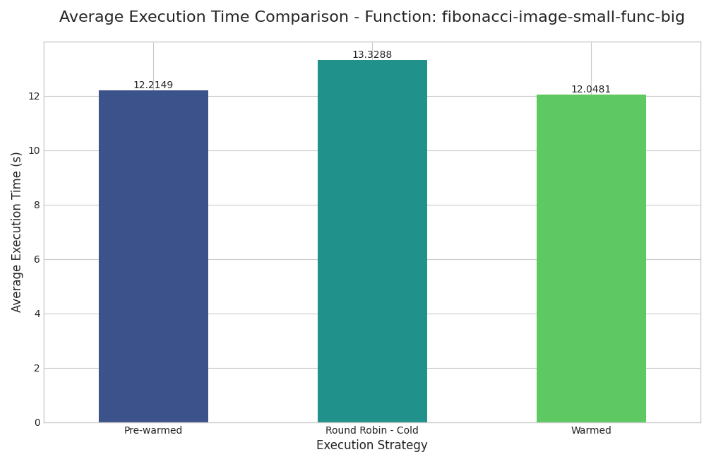

Figure 8: Comparison of average execution time (s) for the computationally intensive function

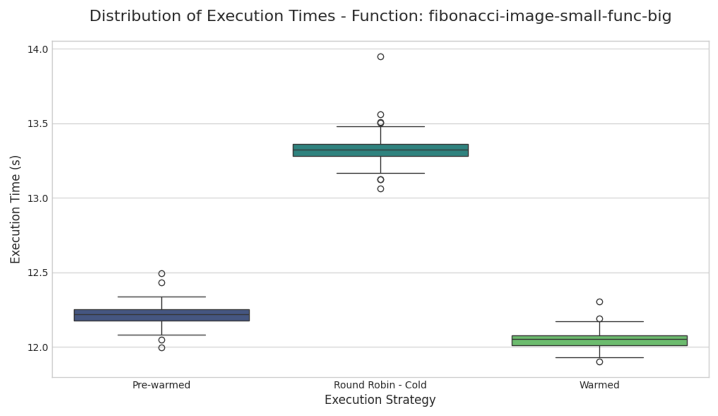

Figure 9: Distribution of execution times (s) for the computationally intensive function

6.3.2 Scenario 2: Comparison of scheduling policies

- Objective: this experiment aims to compare the effectiveness of a static scheduling policy (`RoundRobinPolicy`) with two dynamic and informed policies (`LeastUsedPolicy` and `MostUsedPolicy`) in balancing the workload and influencing the overall performance of the system.
- Configuration: to ensure that node selection was not influenced by pre-existing states, all tests in this scenario were performed in Cold mode. Three separate test sessions were conducted for each of the two functions, modifying only the default scheduling policy in the gateway.
- Expectations:
  - `RoundRobinPolicy`: expected to distribute invocations almost perfectly evenly among the four nodes.
  - `LeastUsedPolicy`: should exhibit smarter behavior, tending to favor nodes with lower CPU usage, balancing the load.
  - `MostUsedPolicy`: the opposite behavior is expected, concentrating the load on the most used node to deliberately create an unbalanced load scenario.

- Results: the collected data confirm expectations regarding load distribution and reveal a measurable impact of policies on latency consistency. The analysis is presented separately for the two functions.

Function `fibonacci-image-big-func-small` (Cold Start dominant) In this scenario, the `RoundRobinPolicy` distributes the 100 invocations (25 per node) equally, and the `LeastUsedPolicy` achieves an almost identical result, demonstrating its effectiveness in keeping the system balanced (Graph 10). The `MostUsedPolicy`, as expected, concentrates the load, assigning 35 invocations to `ssh_node_1`. From a latency perspective (Graph 11), the three policies show almost identical median latency. However, the `LeastUsedPolicy` has the most compact distribution, indicating more consistent and predictable performance. Conversely, the `MostUsedPolicy` shows the greatest variability and the presence of outliers, confirming that load concentration leads to less stable performance.

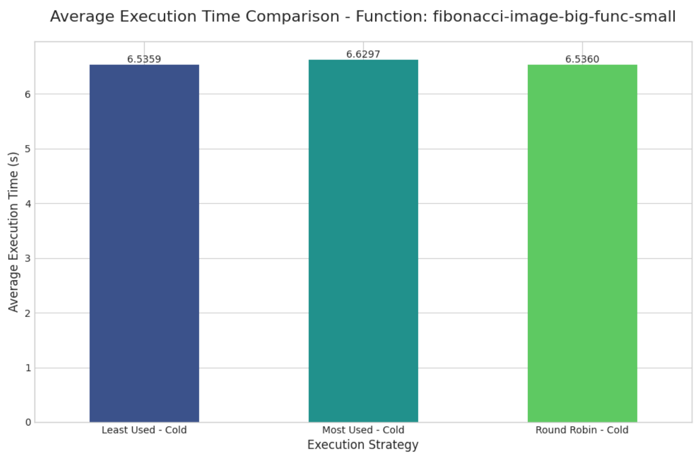

Figure 10: Distribution of invocations for the image-heavy function

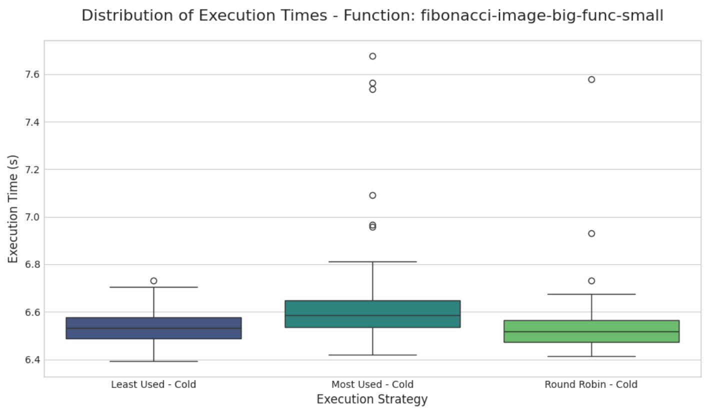

Figure 11: Distribution of execution times for the image-heavy function

Function `fibonacci-image-small-func-big` (Dominant CPU load) Even with a heavy computational load, the load distribution results are consistent (Graph 12). The `MostUsedPolicy` shows even more extreme behavior, assigning 42 invocations to `ssh_node_4`. The impact on latency (Graph 13) follows the same trend observed previously. Although the average latency is similar between policies, the `LeastUsedPolicy` proves to be the most stable strategy, with the narrowest distribution of times. The `MostUsedPolicy` is again the policy with the highest variability, suggesting that the concentration of computationally intensive tasks on the same node leads to a degradation in performance predictability.

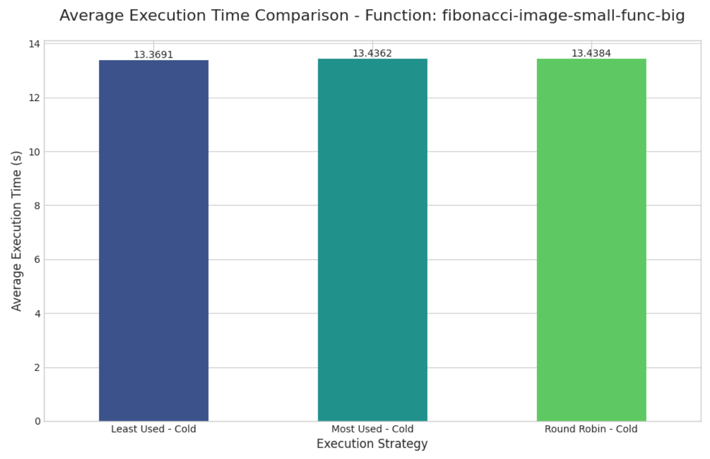

Figure 12: Distribution of invocations for computationally intensive functions

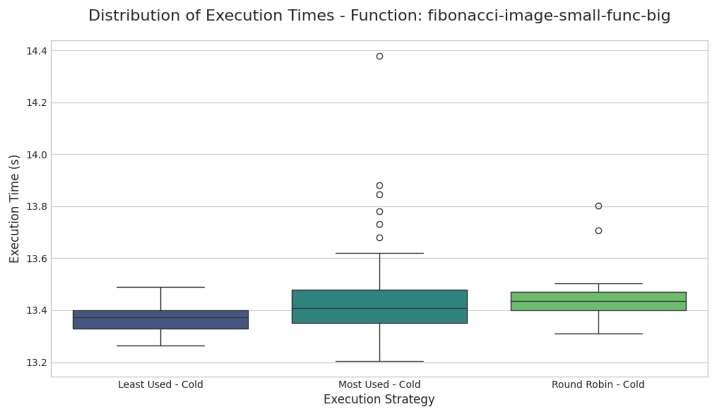

Figure 13: Distribution of execution times for the computationally intensive function

6.4 Discussion of results

The analysis of the results presented in the previous chapter allows us to draw meaningful conclusions about the effectiveness of the various strategies implemented. In this section, the data will be interpreted to explain the phenomena observed, highlighting the trade-offs between the various solutions and linking the results to the initial objectives of the thesis.

6.4.1 Scenario Analysis 1

The results of Scenario 1 demonstrate that warming strategies are the most critical factor for reducing latency in FaaS systems, confirming the initial hypotheses.
For the image-heavy function, switching from Cold to Pre-warmed mode resulted in a clear improvement. This gain is almost entirely attributable to the elimination of Docker image download time, which in this case represents a significant portion of the total latency. Switching further to Warmed mode offers a smaller but still significant improvement, due to the elimination of container creation and initialization time.
The analysis of the computation-heavy function is still important. Although the percentage improvement is lower, the absolute time saved is similar. This shows that the benefit of warming is consistent, but its relative impact decreases as the execution time of the function increases and the size of the image decreases.
Warmed mode is confirmed as the solution with the best overall performance and the most predictable and consistent results. The choice between the three strategies is therefore a compromise: Cold mode optimizes costs at the expense of latency, while Pre-warmed and Warmed modes trade greater resource usage for a significant and measurable reduction in response times.

6.4.2 Scenario Analysis 2

Scenario 2 highlighted an equally important aspect: the impact of scheduling policy on performance stability and predictability.
The load distribution results validated the design of the algorithms: RoundRobinPolicy proved to be a static and fair distributor, while `LeastUsedPolicy` and `MostUsedPolicy` dynamically adapted their decisions based on CPU load, balancing it in the first case and concentrating it in the second.
Although the average latency did not vary dramatically between the three policies (probably due to the sequential nature of the test load), the distribution of response times changed significantly. For both functions, `LeastUsedPolicy` produced the most compact distribution, with a smaller range between minimum and maximum values. This suggests that, even under low load conditions, the choice to avoid nodes with CPU loads contributes to more consistent performance.
In contrast, the `MostUsedPolicy` worsened predictability, increasing the variability of response times and the number of outliers. This demonstrates that poor load balancing not only risks overloading nodes, but also introduces an element of unpredictability into the latency perceived by the user.

7 Conclusions

7.1 Summary of the work carried out

This thesis addressed the need for flexible and reproducible tools for performance analysis in FaaS systems. The growing adoption of the serverless paradigm has made it important to understand phenomena such as cold starts and the impact of scheduling algorithms, which directly affect user-perceived latency and infrastructure efficiency.
To address this need, a FaaS framework in Python was designed and implemented from scratch, specifically conceived as a testbed. The architecture, based on a central API Gateway and containerized execution nodes controlled via SSH, proved to be robust and flexible. The adoption of established design patterns ensured high modularity, making the system easily extensible.
A rigorous testing methodology was implemented, based on two distinct function profiles to isolate and analyze the impact of initialization latency versus computational load. The developed framework allowed for orchestrating controlled experiments to compare different scheduling policies and container management strategies.
Finally, the experimental analysis allowed for comparing the different strategies implemented.
The results quantitatively confirmed the effectiveness of warming strategies, which reduced average latency in scenarios dominated by cold starts. Furthermore, it was highlighted that the choice of scheduling policy mainly impacts performance stability: although under low load conditions the average latency does not undergo significant variations, dynamic policies such as `LeastUsedPolicy` have been shown to offer greater consistency and predictability of response times compared to a static approach.
The work carried out not only validates the theories analyzed, but also provides the community with a practical tool for future research in the field of serverless system performance.

7.2 Current framework limitations

As a prototype designed for research purposes, the framework has certain limitations, dictated by the need to prioritize simplicity and controllability of the test environment over the typical requirements of a production system.

- Security: credentials for SSH communication between the gateway and nodes are managed in clear text, and host verification is disabled. This configuration is completely inadequate for a real deployment.
- State persistence: although metric results are saved to CSV files, function and node availability logs are kept in memory. This means that every time the gateway is restarted, the system state is lost and must be rebuilt from scratch via client registration calls.
- Static node scalability: the test cluster architecture (the number of execution nodes) is defined statically in the `docker-compose.yml` file. The framework does not support dynamic scalability, i.e., adding or removing nodes at runtime.
- User interface: interaction with the framework is only possible via API and scripts (`client.py`), there is no graphical user interface (GUI) that could simplify the configuration of experiments and the visualization of results.
- Error handling: the error handling logic is basic. If a function invocation fails on a node, the error is logged, but no advanced strategies are implemented, such as an automatic retry mechanism on another node.

7.3 Future developments

The above limitations pave the way for many interesting future developments, which could transform the framework from a research prototype into an even more powerful and comprehensive tool.

- Security and usability improvements: a first step could be to strengthen the communication protocol by introducing SSH key-based authentication. A simple web interface could then be developed for managing experiments and viewing results dashboards.
- Advanced monitoring: the current metric collection system could be enhanced by integrating tools such as Prometheus for time-series data collection and Grafana for creating interactive dashboards, offering a much more comprehensive, real-time view of the system status.
- Extension of scheduling policies: the framework lends itself as a basis for implementing and testing more sophisticated scheduling algorithms. Predictive strategies based on machine learning could be explored, which attempt to estimate the execution time of a function or the future load of the nodes.
- Multi-language support: to increase the versatility of the testbed, support could be extended to functions written in other programming languages, allowing the performance of different runtimes in the FaaS environment to be analyzed.
- Concurrent request management: this would allow the framework to be tested under more realistic stress conditions, evaluating not only the latency of individual operations, but also the total throughput of the system. This would allow the behavior of scheduling policies to be analyzed in resource contention scenarios and the responsiveness of the gateway to be measured when it has to handle a large number of SSH connections and Docker commands in parallel.

Finally, the project's source code will be made available under an open source license. It is hoped that this will encourage students, researchers, and enthusiasts to use, modify, and extend the framework, collectively contributing to research and understanding of the performance of serverless systems.

8 Bibliography

- IBM - Cos'è il cloud computing?
  https://www.ibm.com/it-it/think/topics/cloud-computing
- Wikipedia - Cloud Computing
  https://it.wikipedia.org/wiki/Cloud_computing
- Italcom - L'evoluzione del Cloud Computing: dalle virtual machine ai container fino al serverless
  https://www.italcom.it/levoluzione-del-cloud-computing
- Red Hat - Cos'è il Function-as-a-Service (FaaS)
  https://www.redhat.com/it/topics/cloud-native-apps/what-is-faas
- Cloudfare - What is serverless computing?
  https://www.cloudflare.com/learning/serverless/what-is-serverless
- Red Hat - Understanding containers
  https://www.redhat.com/en/topics/containers
- Docker - Use containers to Build, Share and Run your applications
  https://www.docker.com/resources/what-container
- Martin Fowler - Serverless Architectures
  https://martinfowler.com/articles/serverless.html
- Geeks for Geeks - Scheduling and Load Balancing in Distributed System
  https://www.geeksforgeeks.org/computer-networks/scheduling-and-load-balancing-in-distributed-system
- AsyncSSH - Asynchronous SSH for Python
  https://asyncssh.readthedocs.io/en/latest
- Wikipedia - Strategy Pattern
  https://en.wikipedia.org/wiki/Strategy_pattern
- Wikipedia - Chain of Responsibility Pattern
  https://en.wikipedia.org/wiki/Chain-of-responsibility_pattern