
University of Pisa

DEPARTMENT OF INFORMATION ENGINEERING

# Bachelor's Thesis in Computer Engineering

## Automating data collection for feature profiling on FaaS platforms at the Edge

Candidate:
Nicola Beccaceci

Supervisors:
Prof. Giuseppe Anastasi
Prof.ssa Francesca Righetti
Prof. Carlo Vallati

# Contents

# Chapter 1

# Introduction

## 1.1 The crisis of the cloud paradigm

The Internet of Things (**IoT**) is a technological paradigm that refers to the networked interconnection of physical objects equipped with sensors, actuators, computational capabilities, and connectivity. These devices—ranging from environmental sensors to wearables, smart appliances, and connected vehicles—are capable of collecting, processing, and transmitting data in real time, communicating either with each other or with remote infrastructures. The primary goal of the IoT is to extend the Internet into the physical world, transforming ordinary objects into active nodes within a distributed digital ecosystem that can autonomously respond to external stimuli and support automatic or semi-automatic decision-making processes.

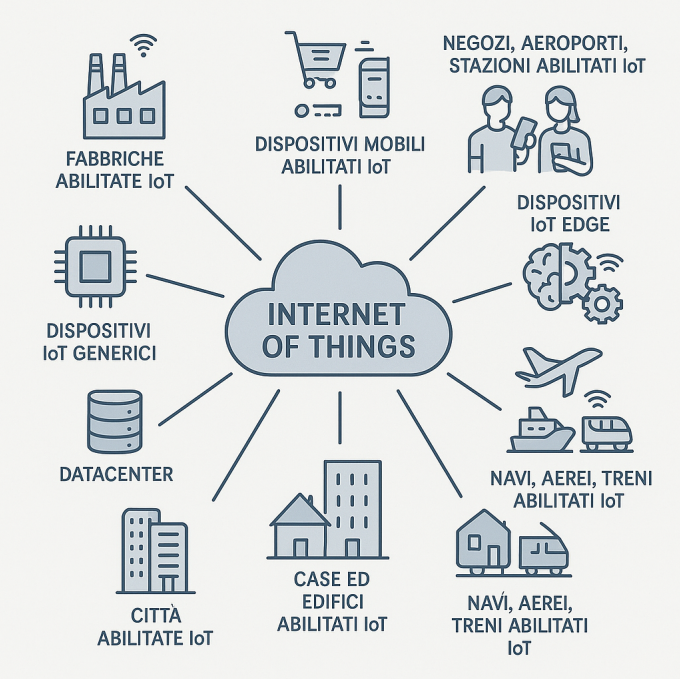


Figure 1.1: Internet of Things: IoT

In recent years, the pervasive adoption of IoT has profoundly reshaped the technological landscape, leading to a proliferation of smart devices that continuously generate and transmit vast volumes of data. According to the International Data Corporation (IDC), by 2025 there will be an estimated 55.7 billion IoT devices in use, generating an unprecedented data volume of approximately 73.1 zettabytes (equivalent to 80,374,299,999,998 gigabytes).

However, this exponential growth should not be interpreted merely as a quantitative phenomenon, as it introduces technical and systemic challenges that would not have emerged with comparable intensity within traditional architectural models. Among the most critical issues is the increasingly pressing need for real-time data processing, a requirement that is particularly evident in IoT systems deployed in industrial, healthcare, transportation, or surveillance contexts, where extremely rapid response times—sometimes on the order of milliseconds—are essential.

Given that physical distance and network congestion introduce non-negligible delays in data transfer and processing, the traditional client-server architecture is inadequate for effectively meeting such stringent latency requirements. While in the past these latencies might have been considered acceptable, for instance in web content delivery or batch processing, they are now incompatible with the operational demands of modern IoT systems, where any delay risks translating into inefficiencies, service disruptions, or, in more critical cases, potential safety hazards.

Compounding this scenario is the very nature of the data generated by IoT devices, which are produced continuously, in a distributed manner, and often in unstructured forms, with a frequency and variety that place significant strain on both network infrastructures and processing systems. Unlike the relatively stable and predictable information flows characteristic of past centralized applications, IoT introduces highly dynamic and difficult-to-predict load patterns, making it impossible to statically provision infrastructure. Instead, systems must be capable of dynamically adapting to the evolving behavior of devices as a function of their environment, context, and temporal factors.

Moreover, since many IoT devices operate in heterogeneous and sometimes hostile geographic environments where stable connectivity cannot be guaranteed, reliance on centralized processing becomes problematic, necessitating the availability of localized computational capabilities to handle at least part of the data directly on-site or in its immediate vicinity.

In parallel, because the expansion of IoT does not follow linear or centrally coordinated growth patterns—new devices are continuously introduced in a heterogeneous and unplanned manner—it becomes essential to adopt horizontally scalable architectures that can autonomously and distributively adapt.

Another critical concern, which becomes even more pronounced with the increasing volume of data generated by IoT, pertains to privacy and data sovereignty. In traditional cloud models, information collected by devices—often relating to user behaviors, habits, locations, or even biometric parameters—is transferred and stored in data centers owned by large private providers, geographically and legally distant from the point of origin. This centralization not only entails risks of unauthorized access or accidental breaches but also raises substantive questions about data ownership and compliance with data protection regulations, which impose stringent restrictions

on the processing and localization of sensitive information.
In critical contexts such as healthcare, industry, or smart cities, this loss of direct control over data represents a significant vulnerability, potentially undermining the overall security of the system.

## 1.2 Edge computing: answers and new challenges

The increasing complexity of IoT systems, together with the critical issues highlighted in the previous section — namely latency, load variability, intermittent connectivity, geographical distribution, and data confidentiality — has clearly demonstrated the inadequacy of a centralized computational model such as the traditional client-server architecture offered by the cloud. In response to these new requirements, the paradigm of **Edge Computing** has emerged, an architectural solution that relocates part of the computational and analytical capabilities toward the edge of the network, that is, in close proximity to the IoT devices generating the data.

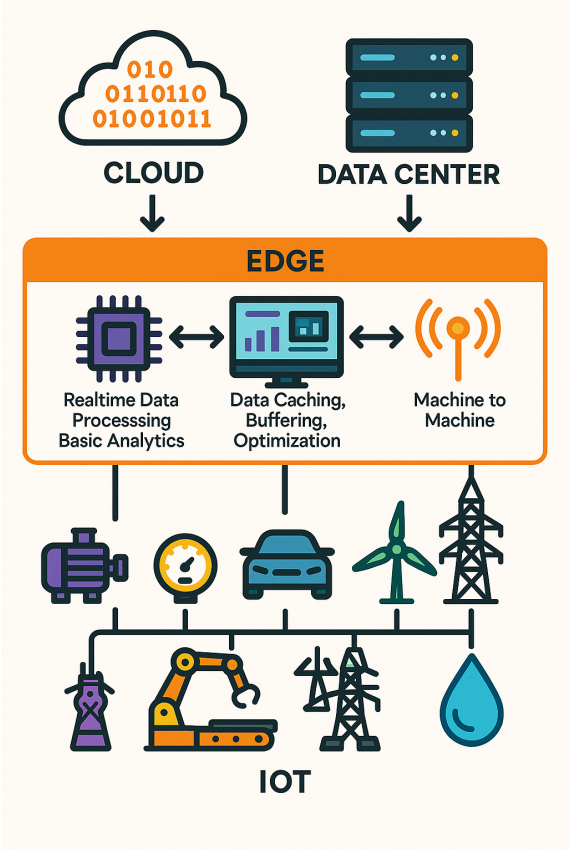


Figure 1.2: Edge Computing

The edge architecture is based on the deployment of locally distributed processing nodes, known as edge nodes, which act as intelligent intermediaries between IoT devices and the central cloud. These nodes are designed to process data in (near) real-time, thereby drastically reducing the need to transmit it to remote data centers, improving system responsiveness and alleviating the burden on the network infrastructure.
This architectural evolution aims to address, in a targeted manner, the main challenges arising from the growth of the IoT ecosystem:

- **Latency reduction**: by moving processing closer to the data source, edge nodes eliminate the dependency on round-trips to the cloud, enabling much faster response times. In contexts such as industrial control or autonomous driving, where even a few milliseconds can be critical, this feature is of paramount importance.
- **Load management and distributed scalability**: through the introduction of internal processing points within the network, edge computing enables a distributed handling of data traffic. Instead of funneling all data flows through a central infrastructure, it becomes possible to dynamically balance the load across multiple edge nodes, thus offering greater flexibility in the face of the unpredictable and localized nature of IoT workloads.
- **Bandwidth optimization**: edge nodes can operate in a partially autonomous mode even under degraded or absent network conditions, allowing devices to continue functioning locally without service interruptions. This is particularly relevant in remote or critical scenarios where reliability is essential.
- **Ottimizzazione della larghezza di banda**: by processing data close to its point of origin, the edge architecture allows only truly relevant data to be transmitted to the cloud, thereby reducing the overall traffic volume and enhancing the network's efficiency.
- **Privacy and data governance**: by bringing processing and analysis as close as possible to the data collection point, the edge architecture significantly reduces the need to transfer large volumes of sensitive data to remote data centers, limiting exposure to third parties and shrinking the overall attack surface. This enables tighter control over the data lifecycle within well-defined boundaries, facilitating compliance with data protection regulations and strengthening transparency toward end users.

However, while the edge architecture introduces substantial improvements, it is not without its own critical issues, some of which stem directly from the distributed and decentralized nature of the paradigm itself.
Energy management emerges as one of the most significant challenges of edge architectures, as decentralized nodes — unlike centralized data centers designed for large-scale efficiency — often operate in uncontrolled environments where power constraints and energy costs are particularly stringent. This distributed configuration, combined with the need to ensure continuous operation in heterogeneous contexts, not only increases the overall consumption but also risks undermining the theoretical benefits of the paradigm when nodes remain active indiscriminately even in the absence of real computational needs.
Secondly, managing a distributed infrastructure such as the edge involves considerably higher complexity than centralized systems, since each node — often operating in heterogeneous and unsupervised environments — requires continuous configuration, monitoring, and updating. This naturally demands the adoption of advanced solutions to ensure automated orchestration of nodes, distributed security, performance monitoring, and predictive maintenance.
Moreover, the effectiveness of edge computing heavily depends on the ability to

predict in advance the workload that each node will need to handle. In the absence of reliable forecasts, it becomes challenging to determine how many nodes to activate, where to position them, and what resources to allocate. Current predictive models, primarily developed for cloud contexts where traffic and load dynamics are more regular and well understood, often prove inadequate for the edge, where variability is much more pronounced and influenced by local factors such as user behavior, environmental conditions, or the operational cycles of specific devices. This lack of effective, edge-specific predictive tools complicates optimal resource allocation, potentially leading to both underutilization and local overloads.

Finally, the hardware and software heterogeneity of edge nodes further complicates the development and deployment of applications. Unlike cloud data centers, where the environment is highly standardized, edge environments often consist of devices with varying computational capabilities, different operating systems, and uneven levels of connectivity and reliability.

In summary, the edge architecture represents a necessary and technologically advanced response to the challenges introduced by the massive spread of IoT. At the same time, however, it opens new fronts for research and development. To fully realize its potential, it is essential to address in a systemic way the issues related to efficient resource allocation, energy management, operational scalability, and dynamic workload prediction, which constitute the core of current and future challenges in distributed computing.

## 1.3 Thesis goals

The introduction of Edge architecture, while representing a significant breakthrough in addressing the operational challenges brought about by the Internet of Things, also introduces a new class of issues that cannot be overlooked.
As highlighted in the previous section, the ability to process data close to its source helps reduce latency, network load, and dependency on connectivity, but it also brings new requirements in terms of management, scalability, and, above all, energy efficiency.
In particular, one of the most critical aspects related to implementing an Edge infrastructure is the need to dynamically activate a number of nodes consistent with the computational load required by IoT devices. An insufficient number of nodes may lead to slowdowns, data loss, or malfunctions in real-time systems, whereas excessive activation results in unnecessary resource waste and a significant increase in energy consumption. Given the growing focus on the sustainability and efficiency of distributed systems, this problem cannot be ignored and must be tackled through a systematic, data-driven approach.
To this end, it is essential to have accurate predictive models capable of estimating in advance the volume of data that will be generated in a given portion of the Edge network, thereby enabling dynamic, proportionate, and energy-sustainable allocation of computational resources. However, while in the cloud computing context such predictive tools are widely developed and supported by an extensive statistical base — thanks to data centralization, traffic historical records, and the use of mature monitoring tools — in the domain of Edge Computing the situation is profoundly different.
To date, there are no consolidated predictive models or reliable statistics concerning the load on edge nodes, nor are there observability tools comparable to those developed for the cloud. This gap is primarily due to the decentralized and dynamic nature of the Edge, the variety of hardware employed, the geographical distribution, and the lack of homogeneous standards for collecting and analyzing system metrics. As a result, edge node allocation decisions are often made in a static, heuristic, or reactive manner, without analytical support based on historical data or reliable forecasts.
This disconnect represents a serious limitation, especially considering that the main challenges of Edge are directly linked to energy consumption and the efficient use of resources. While the cloud can, at least partially, afford some over-provisioning margins thanks to its high capacity and centralized management, Edge requires lean, adaptive allocation solutions guided by precise metrics. Continuing to use — or worse, to extend — statistical models developed for the cloud to edge scenarios proves not only ineffective but potentially misleading, as it ignores the architectural and operational specifics of this paradigm.
It is within this context that the contribution of this thesis is positioned, aiming to fill this methodological gap through the design and implementation of a system for the automated collection, analysis, and monitoring of metrics related to the computational load on edge nodes. The goal is to provide a solid and representative information framework, capable of supporting in the future the development of

predictive models specifically tailored for the Edge domain.
In particular, this thesis will focus on:

- designing an automated data collection system,
- identifying the most relevant metrics for load assessment,
- experimenting with and analyzing data gathered in realistic contexts,
- discussing the implications of the results in terms of dynamic allocation and energy savings.

The ultimate goal is to help lay the groundwork for a new generation of predictive tools explicitly designed for Edge Computing, enabling effective handling of the scalability and sustainability challenges faced by distributed systems in high-density IoT environments.

## 1.4 Thesis Structure

This thesis is organized into seven chapters, each addressing a specific aspect of the work carried out:

- **Chapter 2** – Context
  An introduction to traditional cloud architectures (IaaS, PaaS, SaaS), the serverless and FaaS paradigm, and the concept of cloud without services.
- **Chapter 3** – Technologies Used
  A presentation of the tools adopted, including Docker, Python, and the main libraries employed for monitoring and data analysis.
- **Chapter 4** – System Architecture Design
  A description of the overall system architecture, with particular focus on the main components and their operation.
- **Chapter 5** – Framework Implementation
  An illustration of the key functionalities developed, the code implemented, and the system's behavior during execution.
- **Chapter 6** – Results Analysis
  A report on the collected data, along with comparisons and discussion supported by graphs and quantitative observations.
- **Chapter 7** – Conclusions
  A summary of the thesis contributions and a discussion of possible future developments.

# Chapter 2

# Context

## 2.1 Traditional Cloud Architectures: IaaS, PaaS, SaaS

As highlighted in the previous chapter, adopting a distributed architecture such as Edge Computing represents a response to the inherent limitations of the centralized computational model. To fully understand this architectural evolution, it is necessary to analyze the paradigm that preceded it: **cloud computing**.
Originally designed to provide ubiquitous and flexible access to computational resources, cloud computing has revolutionized IT by offering infrastructures, platforms, and applications as services, delivered over the Internet in a virtualized and scalable manner.
The National Institute of Standards and Technology (NIST) defines cloud computing as a model for enabling convenient, on-demand, ubiquitous network access to a shared pool of configurable computing resources — such as networks, servers, storage, applications, and services — that can be rapidly provisioned and released with minimal management effort or interaction with the service provider.
These resources are provided according to three main service models: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS), each corresponding to a different level of abstraction and responsibility.

**Infrastructure as a Service (IaaS)**
The Infrastructure as a Service (**IaaS**) model represents the lowest level of cloud offering in terms of abstraction. In this scenario, the provider makes available a set of virtualized resources, including servers, storage, networking components, and other essential infrastructure, thereby enabling the creation of customized software environments by installing operating systems, libraries, middleware, and applications, while retaining full control over both the operating system and the software configuration.

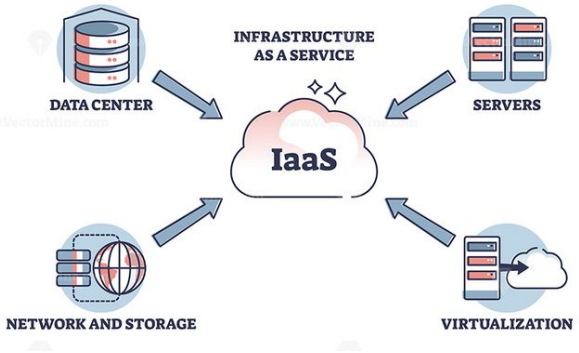


Figure 2.1: Infrastructure as a Service: IaaS

This approach is particularly well-suited when a flexible and scalable infrastructure is required, capable of adapting to specific needs, while at the same time avoiding the burden of purchasing and maintaining physical hardware. For this reason, IaaS is widely employed for development and testing environments, for hosting applications that demand high availability, and for disaster recovery solutions.
Providers such as Amazon Web Services through EC2, Google Compute Engine, and Microsoft Azure are prominent examples of this model, which stands out for several key advantages: the ability to scale resources elastically, increasing or decreasing them dynamically as needed; the adoption of a pay-as-you-go model that allows paying only for the resources actually used, thus optimizing costs; and the flexibility to implement any software environment, adapting to different workloads.
Nonetheless, it should be noted that IaaS requires advanced system administration skills, since the responsibility for configuration, security, and infrastructure management largely falls on the user. Furthermore, while ensuring the availability of infrastructure resources, the provider does not offer application services or integrated development tools, making it necessary to handle these aspects independently.

**Platform as a Service (PaaS)**
The Platform as a Service (**PaaS**) model sits at a higher level of abstraction compared to solutions like IaaS, as it provides a preconfigured environment intended for application development, testing, and deployment. In this model, the provider delivers not only the underlying infrastructure but also an integrated set of development tools, operating systems, databases, middleware services, and application frameworks.


Figure 2.2: Platform as a Service: PaaS

This allows developers to focus exclusively on the application logic, without having to manage the infrastructure or core services, thereby accelerating the development cycle. The pre-established environment enables rapid prototyping, testing, and deployment of applications, significantly reducing time to market, as well as lowering operational costs by eliminating the need to administer servers, databases, or middleware components.
Additional benefits emerge in terms of collaboration, since it becomes possible to work simultaneously on the same platform from different geographic locations, facilitating distributed team dynamics.
However, this model also has some limitations, primarily related to dependency on the provider — a phenomenon known as vendor lock-in — and limited access to environment configuration, which can make integration with legacy systems challenging and offers a significantly smaller margin for infrastructure customization compared to IaaS.
Among the most widely adopted PaaS services are Google App Engine, Microsoft Azure App Services, and Heroku, which are emblematic examples of how this type of cloud offering is implemented at scale.

**Software as a Service (SaaS)**
The Software as a Service (SaaS) model represents the highest level of abstraction within cloud computing, as it provides direct access to complete applications running on infrastructures entirely managed by the provider, eliminating any need for local installation, configuration, or maintenance.
In this context, interaction generally takes place through web browsers or APIs, while the software is consumed as an on-demand service, most often charged under a subscription model. As a result, updates, security, backups, and scalability are fully handled by the provider.

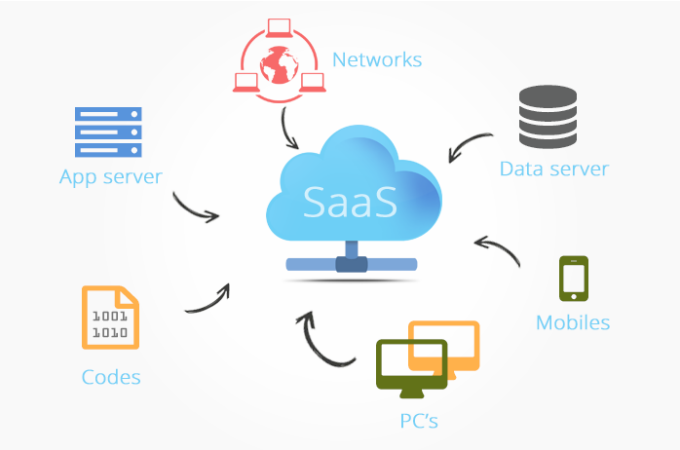


Figure 2.3: Software as a Service: SaaS

These characteristics ensure immediate accessibility from any Internet-connected device and lead to reduced upfront costs, thanks to the absence of perpetual licenses or local infrastructures, while centralized management enables automatic updates.
Nevertheless, adopting SaaS also entails certain drawbacks, mainly due to the limited ability to deeply customize software functionality, potential difficulties in integrating with existing applications, dependency on the provider, and data security concerns, as information is stored on external servers. Well-known examples of SaaS services include Google Workspace, Microsoft 365, Salesforce, Dropbox, and Zoom, which demonstrate how this model is widely implemented across diverse contexts. It should also be noted that IaaS, PaaS, and SaaS models are not mutually exclusive options but rather different service levels that can coexist within the same cloud architecture. Their adoption varies based on multiple factors, such as the type of application, available expertise, customization requirements, and regulatory constraints.
Despite this, the effectiveness of the centralized solutions offered by traditional cloud computing is increasingly being challenged by the emergence of distributed, real-time, and highly localized needs, driven in particular by the Internet of Things (IoT). This highlights the limitations of such architectures — especially in terms of latency, operational autonomy, and data congestion — paving the way for the development of alternative models like Edge Computing, which will be explored in the following sections.

## 2.2 The serverless paradigm and the FaaS model

The evolution of service models in cloud computing has progressively led to an increasing abstraction of the underlying infrastructure. In this process, the serverless paradigm has represented a significant turning point, pushing this abstraction to a level where developers no longer need to worry about managing, configuring, or scaling servers. This approach, which finds its most direct expression in the Function as a Service (**FaaS**) model, has profoundly transformed how applications are designed and executed in the cloud, making it particularly attractive in dynamic and highly variable scenarios such as those typical of IoT.
In the serverless model, applications are broken down into small logical units called functions, which are designed to execute in response to specific events known as triggers and to terminate immediately after execution. The term "serverless" does not imply the physical absence of servers, but rather indicates that infrastructure management is entirely handled by the cloud provider, who automatically takes care of provisioning, scaling, and decommissioning the compute resources.
The user does not define or size the virtual servers on which the code will run but simply uploads the function, specifying its invocation trigger and minimal configuration parameters.

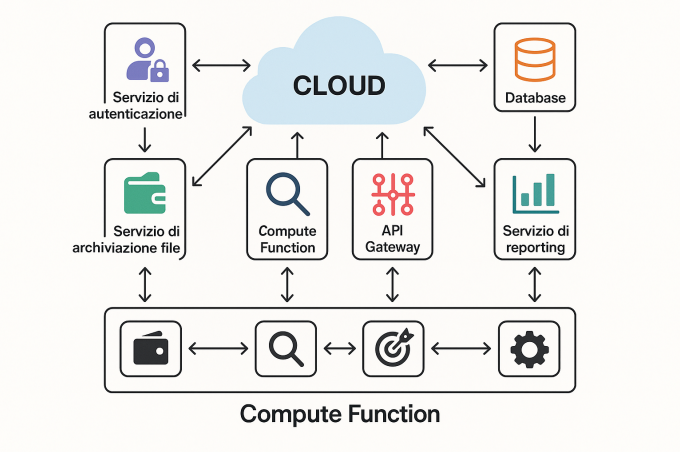


Figure 2.4: Function as a Service: FaaS

This approach offers numerous advantages. Firstly, the economic model is based solely on execution time: the user pays only for the actual time consumed by the function, with granularity down to the millisecond.
Moreover, scalability occurs transparently: as the number of requests increases, the system automatically creates parallel instances of the function, balancing the load without any explicit intervention. The elimination of server provisioning and management processes allows developers to focus exclusively on application logic, reducing time-to-market and simplifying software maintenance.
However, the serverless paradigm also comes with some structural constraints. Functions are designed to be stateless, meaning they do not maintain state between

executions. This necessitates relying on external systems for data persistence or session management, which can introduce additional architectural complexities.
Another critical aspect of the model concerns so-called "cold starts": the first invocation of a function may experience added latency due to the initialization of the execution environment, especially if the function has not been used recently. In centralized cloud contexts, these issues are partially mitigated by the providers' ability to pre-scale instances, but in distributed environments such as Edge Computing—where resources are limited and connectivity is intermittent—they become more pronounced.
The most widely adopted FaaS platforms today—including AWS Lambda, Google Cloud Functions, and Azure Functions—have been developed to operate within highly connected, centralized data centers. In such environments, centralized control and the high availability of resources help mask many of the paradigm's potential inefficiencies. However, when this model is transposed to edge scenarios, these same characteristics risk becoming bottlenecks.

## 2.3 Cloud without services

The evolution of service models in cloud computing has led to a progressive abstraction of the underlying infrastructure, reaching a point where the **serverless paradigm** has marked a particularly significant turning point, as it completely frees developers from the burden of managing, configuring, and scaling servers. This approach, which finds its fullest expression in the Function as a Service (FaaS) model, has profoundly changed the way applications are designed and executed in the cloud, making it extremely attractive in contexts characterized by dynamism and high variability, such as those typical of the Internet of Things.

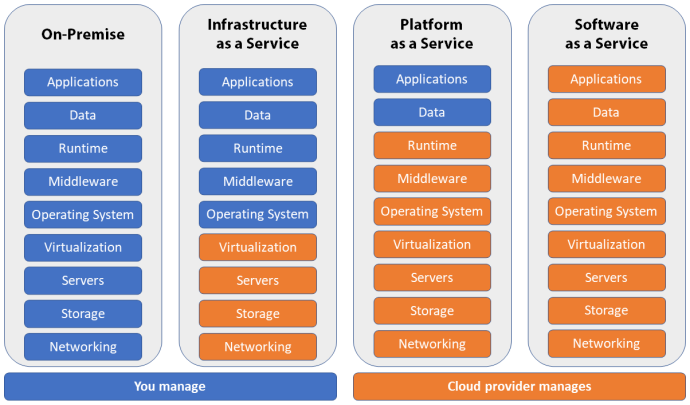


Figure 2.5: Cloud without services

Within the serverless model, applications are typically broken down into small logical units, known as functions, designed to be triggered in response to specific events—so-called triggers—and terminated immediately after execution. The term "serverless" does not imply the physical absence of servers but rather indicates that infrastructure management is entirely entrusted to the cloud provider, who automatically handles the provisioning, scaling, and decommissioning of the necessary compute resources. In this scenario, there is no need to define or size the virtual servers on which the code will run; it is sufficient to upload the function, specify its invocation trigger, and set minimal configuration parameters.
This setup offers a series of benefits, as the economic model adopted is based exclusively on execution time, allowing payment to be proportional to the actual time consumed by the function—often calculated with millisecond-level granularity—while scalability is ensured transparently. An increase in requests automatically results in the creation of parallel instances of the function and load balancing, without requiring any explicit intervention. The absence of provisioning and server management processes also allows developers to focus solely on application logic, thereby reducing time-to-market and simplifying software maintenance.
However, it should be noted that the serverless paradigm also presents some structural limitations, as functions are designed to operate in a stateless manner. This means they do not maintain state between executions, making it necessary to rely

on external systems for session management or data storage, thereby introducing additional architectural complexities. Another critical issue concerns the so-called "cold starts," since the first invocation of a function may take longer due to the initialization of the execution environment, especially if the function has not been recently invoked. In centralized cloud contexts, such issues are partially mitigated by the providers' ability to prepare pre-scaled instances, but in distributed environments such as Edge Computing—where resource availability is limited and connectivity may be intermittent—these problems tend to become more pronounced.
Today's most widespread FaaS platforms, including AWS Lambda, Google Cloud Functions, and Azure Functions, have been designed to operate within highly connected and centralized data centers, where centralized control and abundant resource availability help mask many of the potential inefficiencies of this paradigm. However, these same characteristics risk becoming actual bottlenecks when attempting to transpose the model to edge scenarios.

**Limitations of traditional and serverless architectures**
In traditional cloud architectures, especially in contexts where it is necessary to process large volumes of real-time data generated by distributed sensors, some structural limitations become clearly evident. These are mainly linked to the centralized aggregation of information, which inevitably introduces latency, increases system dependence on network connectivity, and creates bottlenecks when data volumes grow at scale. In such scenarios, the round-trip latency to the data center may prove incompatible with the requirements of time-sensitive processing, while the sheer amount of data produced at the edge is often too large to be continuously transferred to the cloud. Additionally, reliance on services provided by external providers entails phenomena of technological lock-in and raises privacy concerns, since data leaves the local collection environment, creating issues that are particularly relevant in critical IoT applications.
On the other hand, while freeing developers from the burden of configuring and managing servers, the serverless paradigm introduces non-negligible constraints in the IoT domain. The execution of functions is often subject to unpredictable delays due to cold starts, degrading performance in real-time contexts. Furthermore, the platforms themselves—such as AWS Lambda—impose strict limits on the maximum duration of function execution, with typical timeouts on the order of fifteen minutes. Added to this is the need for a constantly active connection to the provider and dependence on proprietary APIs, factors that exacerbate the risk of lock-in and make migration or operation in isolated environments complex.

# Chapter 3

# Technologies used

## 3.1 Docker and Containerization

In light of the considerations discussed in Chapter 2, where it was highlighted how the adoption of a decentralized edge architecture introduces specific challenges related to the efficient, scalable, and replicable distribution of software, it is appropriate to delve deeper into the main technologies used in this work, with particular focus on the containerization paradigm. This paradigm has taken on a strategic role in creating lightweight, modular, and easily portable environments necessary to ensure consistency and repeatability of service execution on heterogeneous edge nodes.
n this context, **Docker** emerges as a key tool thanks to its ability to significantly simplify deployment operations through the use of containers — that is, self-sufficient executable units that bundle the application along with all required dependencies, including libraries, configurations, and runtime files. Based on an operating system-level virtualization approach, containers allow isolation of execution while sharing the host kernel, drastically reducing overhead compared to traditional virtual machines that require the entire guest OS stack. This translates into near-instant startup times, a small memory footprint, and extremely efficient use of underlying hardware resources — crucial aspects in edge contexts where computational capacity and memory availability may be limited.
Moreover, Docker provides an intuitive command-line interface and a comprehensive API for managing images, containers, and virtual networks, facilitating application lifecycle automation and integration with CI/CD pipelines. The layered image model optimizes version management by minimizing duplication of common layers and improving software distribution.
From an isolation standpoint, each container is confined through Linux kernel mechanisms such as namespaces and cgroups, which effectively segregate resources among different processes while allowing them to run on the same host. This achieves a level of separation lighter than that of virtual machines but, in most cases, fully adequate to guarantee security and stability in distributed systems. Thanks to this combination of lightness, portability, isolation, and ease of replication, containerization via Docker proves especially suitable for edge scenarios, where maintaining a high degree of consistency and reliability in service delivery is essential,

even while operating in environments with potentially heterogeneous and constrained resources.

## 3.2 Containers and Virtual Machines

To fully understand the reasons behind the choice of containers, it is useful to compare them with the traditional virtualization approach represented by virtual machines (VMs). VMs emulate the entire hardware of the host system, running a full operating system on each instance. This provides robust isolation and the ability to run different operating systems on the same physical machine. However, emulating the entire stack introduces significant overhead: each virtual machine requires a substantial portion of CPU, RAM, and disk space, slows down startup times, and reduces the density of runnable instances per host.
In this regard, lighter alternatives to full virtualization have emerged over time, such as paravirtualization and, more recently, lightweight virtualization. Paravirtualization is based on modifying the guest operating system to be aware that it is running inside a virtualized environment. In this model, the guest OS avoids executing non-virtualizable instructions by replacing them with calls to specific interfaces exposed by the Virtual Machine Monitor (VMM), called hypercalls. This technique reduces the use of emulation, thus improving performance compared to full virtualization. However, it requires modifying the guest OS kernel, limiting its use to open-source or modifiable systems.
With the advent of operating system-level virtualization, or Operating System Virtualization, an even more efficient form of isolation has been achieved, where neither hardware nor a full operating system need to be emulated. In this paradigm, all containers share the same host kernel but operate in separate user spaces. Technologies such as Linux Containers (LXC) and Docker implement this form of virtualization by leveraging two fundamental Linux kernel mechanisms: namespaces, which isolate resources such as filesystems, network, processes, and users, and control groups (cgroups), which impose limits on resource usage like CPU, memory, and network bandwidth.
The absence of a hypervisor and the shared kernel allow containers to start almost instantly and use minimal system resources. Furthermore, application performance within a container is essentially equivalent to native execution, since there is no penalty related to hardware or OS virtualization. However, this efficiency comes with the trade-off of kernel compatibility: all containers must be compatible with the host kernel. Additionally, although isolation levels are generally adequate, in highly sensitive scenarios the use of VMs may be preferable for stronger separation.

## 3.3 Python Libraries

In light of the considerations discussed in previous chapters regarding edge architecture and the advantages offered by containerization, it became necessary to develop an experimental environment capable of testing these solutions in realistic scenarios.

The main objective was to observe how distributed architectures react to variable loads, evaluating resource usage metrics and execution times.
To achieve this, a modular framework was designed, able to orchestrate containers, monitor resources in real time, and analyze the collected data both quantitatively and graphically. Within the scope of this thesis, a system composed of multiple integrated components was therefore developed: a module for container orchestration, a resource monitoring system, and a set of tools for statistical analysis and result visualization. Python, thanks to its simple syntax and rich availability of specialized libraries, was the natural choice for developing each of these parts.
The **docker-py** library was employed to programmatically interface with Docker, enabling automation of the building and running of containers described in previous sections, as well as monitoring their state and collecting low-level statistics directly without relying on external tools. Within the monitoring module, the **psutil** library played a central role by providing access to detailed information about the operating system, such as CPU, memory, and thread usage, which are essential for accurately profiling the behavior of functions during execution.
IThe acquired data were then organized and stored using **pandas**, one of the most powerful and widely used libraries for managing tabular datasets, enabling structuring samples into DataFrames and serializing them into CSV files to facilitate subsequent analysis. During post-processing, the framework utilized the **statistics** module (particularly the functions for calculating means and standard deviations) to summarize the observed distributions, while **matplotlib** was used to generate graphical representations of the results, such as time plots or resource usage histograms, thus providing immediate visual support for interpreting the experimental data.
Overall, the combination of these libraries allowed the development of a lightweight yet highly versatile system capable of covering all phases of the experimental workflow in an integrated manner: from container orchestration and real-time metric monitoring to subsequent quantitative analysis and result visualization, essential features for evaluating edge architectures in dynamic scenarios with highly variable workloads.

# Chapter 4

# Architectural design

Chapter 3 presented the technologies enabling the execution and monitoring of functions in containerized edge environments. This chapter describes the overall architecture of the developed system, with particular focus on the main components, the design choices made, and the operational flow that—from the request to execute a function—leads to the collection, aggregation, and structured visualization of metrics.

The architecture is structured around a set of tightly interconnected modules that collaborate to manage the entire experiment lifecycle. The core of the system is represented by the Docker image manager, responsible for building container images for each function using the high-level API client.images.build.

This approach, which uses minimal Dockerfiles, drastically reduces image sizes and speeds up the build and deploy cycle—an especially crucial aspect in edge scenarios where image transmission and loading must be fast and lightweight.

Once an image is built or updated, the orchestration module handles the startup, monitoring, and termination of containers. Orchestration is implemented entirely through the docker-py library, without relying on external tools like docker-compose, to ensure maximum programmatic control and flexibility. Each function is executed in detached mode, with the possibility to specify input parameters dynamically. Furthermore, to extend experimentation to high-concurrency scenarios, a dedicated module leveraging threading was included to launch multiple containers in parallel, each monitored independently: the results of parallel executions are saved in separate CSV files, enabling direct comparisons between single and parallel modes.

Supporting this orchestration is a resource monitoring module that relies on a dedicated thread to collect, in real time, metrics related to CPU usage, memory, and host clock frequency. The monitor uses psutil to periodically query the operating system, while container-specific statistics are retrieved via container.stats. All information is progressively updated in a shared dictionary, which remains accessible to the main thread once the container execution ends.

The collected metrics are then organized and serialized into CSV files using pandas, which ensures a consistent tabular format easily reusable for further processing. Each file row includes the timestamp, function name, input parameters, execution duration, and average resource usage, along with the host's available and total memory.

Finally, the analysis and visualization phase is automated through a dedicated script that uses matplotlib to generate plots of the main metrics and scipy.stats to calculate 95% confidence intervals, thus providing statistical and visual support for interpreting the experimental data. The generated graphs are saved in the plots/ folder and immediately highlight differences between execution modes and the behavior of different edge nodes. The described process is summarized in flowchart 4.1, which schematizes the main operational phases of the framework—from the Docker image build stage to the generation of final results.

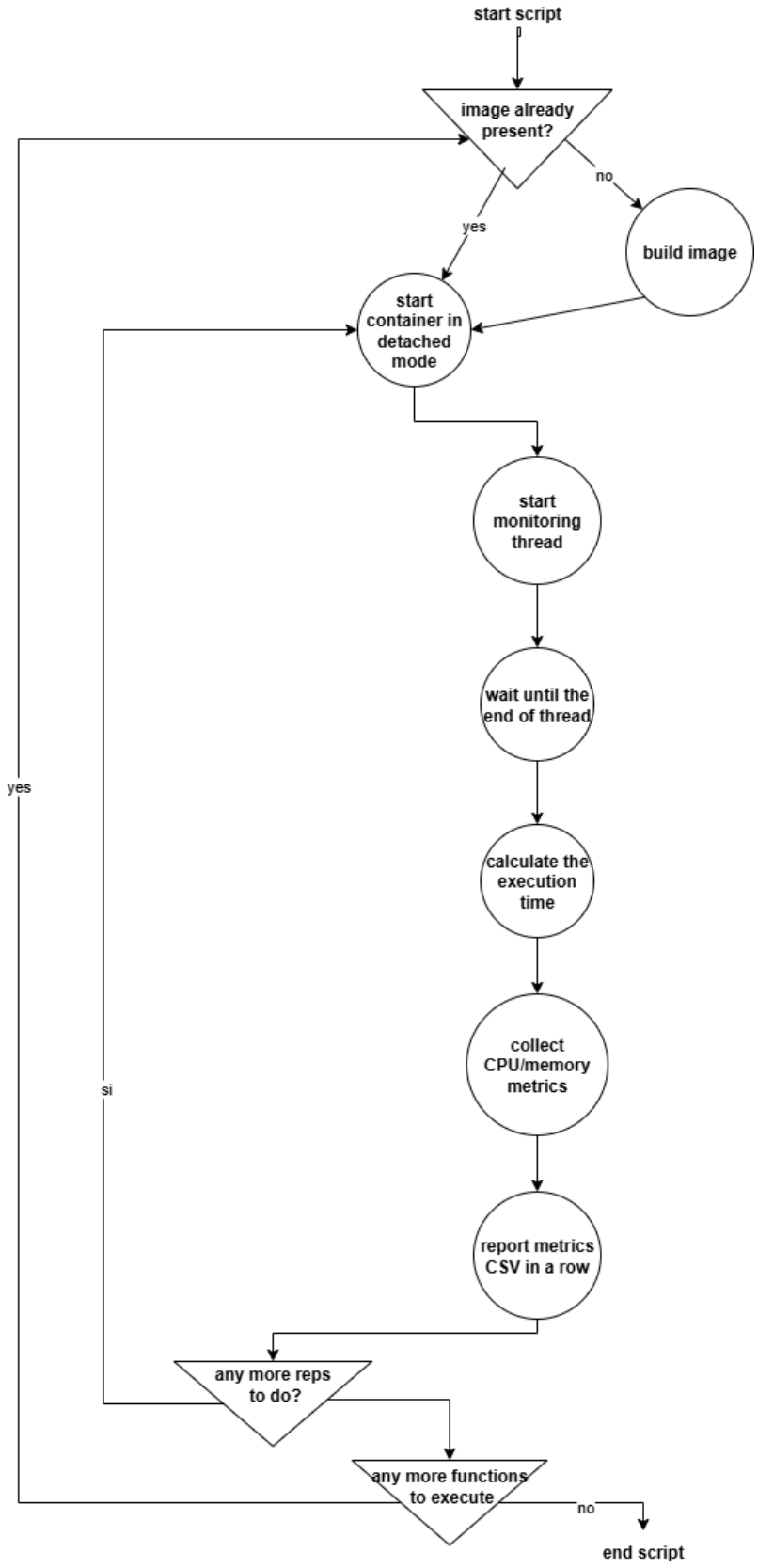


Figure 4.1: Workflow's frameowrk

# Chapter 5

# Framework Implementation

The framework developed in this thesis work is composed of a series of integrated modules, designed to automate all phases of the experimentation process: from building Docker images for each function, to their isolated execution, and finally to the collection and persistence of metrics, followed by statistical analysis and graphical representation of the results.
Below, the implemented functionalities, code operation, characteristics of the tested functions, and the methodologies used to conduct the experiments are described in detail.

## 5.1 Building and managing Docker images

PFor each analyzed function, a minimal Dockerfile was prepared, placed in the root of the workspace (for example: Dockerfile.fibonacci, Dockerfile.hello, Dockerfile.isprime, Dockerfile.ultralytics).
These files define extremely lightweight execution environments, installing only the dependencies strictly necessary for the operation of the individual function, thereby reducing the final image size and minimizing deployment times. As an example, the Dockerfile used for calculating Fibonacci numbers, Dockerfile.fibonacci, is provided:

```
FROM python:3.10-slim
WORKDIR /app
COPY fibonacci.py /app/
```

Listing 5.1: Dockerfile example

The image building process is managed by the monitor.py module through the build_image() function, which leverages Docker's high-level API (client.images.build) to generate the image from the current directory.
During this process, build logs are printed to the console, allowing monitoring of any warnings or errors until confirmation of the image creation.

```
def build_image(client, tag):
    print("Building Docker image...")
    image, logs = client.images.build(path=".", tag=tag)
    for chunk in logs:
        if 'stream' in chunk:
            print(chunk['stream'].strip())
    print("Image built successfully.")
```

Listing 5.2: build_image() function

## 5.2 Execution of functions in isolated containers

Once the Docker image has been prepared, the functions are executed in isolated containers through the run_container_with_input() function, also defined in monitor.py.
This method starts a container in detached mode, optionally passing an input parameter, while the image name and related parameters are dynamically selected based on the specific function. At the same time, a dedicated thread is launched for resource monitoring using the threading library, which collects usage metrics in parallel without interfering with the main program flow. During execution, the monitoring thread continues as long as the container remains active; once the container completes its lifecycle and returns the exit code via wait(), the program waits for the thread to finish using join. Finally, all collected metrics are serialized into a CSV file to enable subsequent analysis.
This approach, in addition to ensuring process isolation and efficient resource monitoring, allows for the easy orchestration of heterogeneous function executions within a complex workflow, keeping the execution and metric collection responsibilities separate.

```
def run_container_with_input(client, image_tag, input_value,
    function_id):
    container = run_container(client, image_tag, input_value)
    stats_data = {}
    monitor_thread = threading.Thread(
        name="monitor_container",
        target=monitor_container_resources,
        args=(container, stats_data)
    )
    monitor_thread.daemon = True
    monitor_thread.start()
    exit_code = container.wait()
    monitor_thread.join()
    write_metrics_to_csv(stats_data)
```

Listing 5.3: run_container_with_input() function

To support scenarios characterized by high concurrency, the parallel_monitor.py script was developed, which simultaneously launches multiple containers, each independently monitored.
This is achieved by instantiating a separate thread for each function to be tested, thus ensuring that the execution and monitoring of each container proceed in parallel without mutual interference. The total number of repetitions is configurable through a variable N, which was set to 10 in the conducted tests, in order to replicate the launch sequence and collect a significant set of metrics.

```
def main():
    client = docker.from_env(timeout=300)
    functions = [
        {"name": "Fibonacci", "image": "fibonacci-app", "input": 20000},
        {"name": "Hello", "image": "hello-app", "input": None},
        {"name": "IsPrime", "image": "isprime-app", "input": 29937646239629496719},
        {"name": "YOLOv8", "image": "yolo-runner-app-ultralytics", "input": "input.jpg"}
    ]
    N = 10
    for i in range(N):
        threads = []
        for func in functions:
            t = threading.Thread(
                target=run_and_monitor,
                args=(client, func["image"], func["input"] if func["input"] else "", func["name"])
            )
            threads.append(t)
            t.start()
        for t in threads:
            t.join()
```

Listing 5.4: main() function of the monitor.py file

## 5.3 Resource monitoring

Resource monitoring is handled by the monitor_container_resources() function, defined both in monitor.py and parallel_monitor.py, to allow its use in the different execution contexts envisioned by the system.
This function runs inside a separate thread and periodically queries Docker's container.stats API, which streams detailed, real-time information on the container's CPU and memory usage. In parallel, psutil is employed to obtain metrics related to the host, such as CPU frequency and available memory, in order to provide a combined view of the load on both the container and the host system.
Particular attention is given to calculating the container's CPU usage as a percentage of the host's CPU, determined by the variation in container CPU cycles recorded across two successive snapshots, normalized by the change in the host's total CPU cycles.
The function progressively updates a shared dictionary, continuously tracking the resources consumed during execution.

```
1  def monitor_container_resources(container, stats_data):
2      for stat in container.stats(stream=True, decode=True):
3          cpu_stats = stat.get("cpu_stats", {})
4          precpu_stats = stat.get("precpu_stats", {})
5          cpu_delta = cpu_stats["cpu_usage"]["total_usage"] -
       precpu_stats["cpu_usage"]["total_usage"]
6          system_cpu_delta = cpu_stats["system_cpu_usage"] -
       precpu_stats["system_cpu_usage"]
7          cpu_usage = (cpu_delta / system_cpu_delta) * 100.0 if
       system_cpu_delta > 0 else 0.0
8          stats_data["cpu_usage"] = cpu_usage
9          stats_data["memory_usage"] = stat["memory_stats"]["usage"]
       / (1024 * 1024)  # MB
10         stats_data["cpu_freq"] = psutil.cpu_freq().current / 1000
       # GHz
11         stats_data["memory_available"] = psutil.virtual_memory().
       available / (1024 * 1024)
```

Listing 5.5: monitor_container_resources() function

## 5.4 Raccolta e salvataggio delle metriche

Una volta conclusa l'esecuzione dei container e completata la fase di raccolta dei dati, le metriche ottenute vengono riportate all'interno di un file CSV tramite la funzione write_metric_to_csv().
Tale funzione si occupa di strutturare i risultati in modo ordinato, inserendo in ogni riga tutte le informazioni relative alla funzione testata, ai parametri di input utilizzati, alla durata complessiva dell'esecuzione, nonché all'impiego medio della CPU e della memoria. Vengono inoltre registrati la frequenza operativa della CPU dell'host e alcuni dettagli sulle caratteristiche hardware del sistema, come il numero di core logici e fisici e la quantità totale di memoria disponibile.
Il file CSV viene creato con le opportune intestazioni nel caso in cui non sia già presente, oppure aggiornato in modalità append qualora esista, così da consentire l'accumulo progressivo dei dati derivanti da esecuzioni multiple. In questo modo risulta possibile costruire un dataset storico delle misurazioni, utile per successive analisi comparative o per la generazione di report. Di seguito viene riportata l'implementazione della funzione write_metrics_to_csv(), che realizza in modo diretto questa logica di serializzazione delle metriche.

```
def write_metrics_to_csv(data, filename="metrics.csv"):
    file_exists = os.path.isfile(filename)
    timestamp = time.strftime("%Y-%m-%d %H:%M:%S", time.localtime()
    )
    data['Timestamp'] = timestamp
    with open(filename, mode='a', newline='') as csv_file:
        fieldnames = ['Timestamp', 'Function', 'Input', 'Execution
    Time (s)',
                      'CPU Usage (%)', 'CPU Frequency (GHz)',
                      'Number of Logical Cores', 'Number of
    Physical Cores',
                      'Memory Usage (MB)', 'Memory Usage (%)',
                      'Memory Available (MB)', 'Host Total Memory (
    MB)']
        writer = csv.DictWriter(csv_file, fieldnames=fieldnames)
        if not file_exists:
            writer.writeheader()
        writer.writerow(data)
```

Listing 5.6: write_metrics_to_csv() function

## 5.5 Description of the tested functions

To validate the framework, several functions of increasing complexity were used:

- **Hello (hello.py)**: simply prints a string, useful for measuring the minimal overhead associated with the container and metrics collection.
- **Fibonacci (fibonacci.py)**: iteratively computes the Fibonacci number at index n, with a typical input of 20000, ideal for CPU-bound tests.
- **IsPrime (isprime.py)**: checks the primality of a very large integer (29937646239629496719).
- **YOLOv8 (yolo_runner.py)**: uses the Ultralytics framework to perform detection, segmentation, and classification inference on images, representing a practical case of AI load on edge nodes.

A snippet from yolo_runner.py shows how the function receives the image file as a command-line argument and processes the same image with three different tasks.

```
if __name__ == "__main__":
    input_file = "input.jpg"
    if len(sys.argv) > 1:
        input_file = sys.argv[1]
    run_yolo("detect", input_file)
    run_yolo("segment", input_file)
    run_yolo("classify", input_file)
```

Listing 5.7: main() function of the yolo_runner.py file

## 5.6 Collection and processing of the obtained metrics

The analysis of the metrics collected during executions is fully automated through the script plot_generator.py, which orchestrates the entire process: starting from reading the CSV files generated during the monitoring phases, it aggregates the data by function and edge node, computes 95% confidence intervals using statistical functions provided by scipy.stats, and produces comparative plots with the matplotlib library. More specifically, the data is loaded into pandas.DataFrame structures, filtered by function and machine, and then subjected to statistical analysis to estimate the mean and its uncertainty via the confidence interval.
This approach makes it possible to visually highlight, in the resulting plots, performance variations not only in terms of their average values but also with respect to their variability.
The generated plots, saved in the plots/ directory, are produced in high resolution thanks to explicit output parameter configuration — specifically with a density of 300 dpi and canvas dimensions optimized to ensure maximum readability — making differences in behavior between the various tested functions, execution modes, and involved edge nodes immediately apparent.
Below is an excerpt of the code for the plot_group() function, which is responsible for building the bar charts along with the confidence intervals computed on the aggregated data.

```
def plot_group(data_dict, group_name):
    for metric_en, metric_label in metrics.items():
        for i, function in enumerate(functions):
            values, ci_values = [], []
            for machine in machine_names:
                df_function = data_dict[machine][data_dict[machine
][“Function”] == function]
                if not df_function.empty:
                    numeric_data = pd.to_numeric(df_function[
metric_en], errors=’coerce’).dropna()
                    if len(numeric_data) > 0:
                        mean, margin =
compute_95_confidence_interval(numeric_data)
                    else:
                        mean, margin = 0, 0
                else:
                    mean, margin = 0, 0
                values.append(mean)
                ci_values.append(margin)
            plt.bar([pos + i * width for pos in x], values, width=
width, yerr=ci_values, label=function, capsize=5)
```

Listing 5.8: plot_group() function:

# Chapter 6

# Analysis of the results

## 6.1 Methodology of the experimental analysis and evaluation metrics

The analysis presented in this chapter, conducted through a systematic approach that evaluates different types of workloads on a heterogeneous infrastructure, aims to understand not only resource efficiency (CPU and memory) but also the scalability of functions as hardware capacity increases and the trade-offs between sequential and concurrent execution—fundamental elements for optimizing deployments in distributed environments.
The studied nodes, characterized by distinct configurations ranging from virtual machines to physical edge nodes, include:

1. Edge node 1 – Virtual machine with 2 vCPUs and 2GB RAM
2. Edge node 2 – Virtual machine with 4 vCPUs and 4GB RAM
3. Edge node 3 – Virtual machine with 16 vCPUs and 31.2GB RAM
4. Edge node 4 – Physical edge node with 4 vCPUs and 4.15GB RAM
5. Edge node 5 – Physical edge node with 8 vCPUs and 15.6GB RAM

The integration of both types of hosts—on one hand, virtual machines with their management flexibility guaranteed by the hypervisor, and on the other, physical edge nodes with their optimized performance and reduced response times—allows for a comparative performance evaluation that takes into account the architectural specificities of each solution.
Performance analysis was conducted following two complementary modes: single execution, where each function was run in isolation within a dedicated container to precisely measure resource utilization metrics such as execution time, average CPU and memory consumption, and the host CPU operating frequency; and parallel execution, using the parallel_monitor.py script, which enabled launching multiple containers simultaneously, each executing a different function, independently moni-

tored via separate threads.
This dual approach allowed for the assessment of the impact of concurrency on the system and for observing workload behavior in real-world scenarios, analyzing metrics such as CPU usage percentage relative to the host system, memory availability, and resource balancing under overlapping loads. To accurately characterize the behavior of applications and infrastructure across the different experimental scenarios, the following main metrics were sampled:

- **Execution Time (s)**: total execution time of each workload, a direct indicator of temporal efficiency.
- **CPU Usage (%)**: average CPU usage percentage during execution, useful for understanding the computational load intensity.
- **CPU Frequency (GHz)**: average CPU usage percentage during execution, useful for understanding the computational load intensity.
- **Number of Logical Cores / Physical Cores**: number of logical and physical cores available on the host, information that relates resource usage to the theoretical capacity of the node.
- **Memory Usage (MB) e Memory Usage (%)**: absolute and relative memory consumption with respect to the host's total memory, to assess the actual RAM resource occupation.
- **Memory Available (MB)**: amount of free memory during execution, indicative of the remaining margin before potential saturation.
- **Host Total Memory (MB)**: total memory of the host, used as a reference to contextualize usage data.

The inclusion of these metrics allowed the construction of a detailed analytical framework, useful not only for highlighting performance differences among the various nodes and execution modes, but also for revealing the implicit trade-offs in the simultaneous management of multiple workloads. This in turn provides insights for optimal resource allocation strategies in distributed edge-cloud contexts.
In light of this, we now proceed to the detailed examination of the plots that precisely illustrate the obtained experimental results.

## 6.2 Single execution analysis

**Execution Time**

The analysis of 6.1 highlights how the behavior of various workloads changes depending on the resources available on the edge nodes. Regarding the Fibonacci function, the execution time is under 5 seconds on all nodes, except for Edge node 4, which shows an average execution time of 10 seconds.

A different picture emerges with YOLOv8, a complex workload that benefits both from parallelism and a good availability of memory: here, moving from Edge node 4 with 4 vCPUs to Edge node 5 with 8 vCPUs results in approximately a 30% reduction in execution time, thanks to the combination of a higher number of cores and sufficient RAM to support its activity.

As for the Hello and isPrime functions, characterized by an extremely light load, execution times remain almost constant and negligible across all nodes, effectively influenced more by the container startup overhead than by the actual available hardware resources.

Confirming these considerations, the following chart directly illustrates the trend of average execution times across the different nodes for each tested function, highlighting the most significant differences.

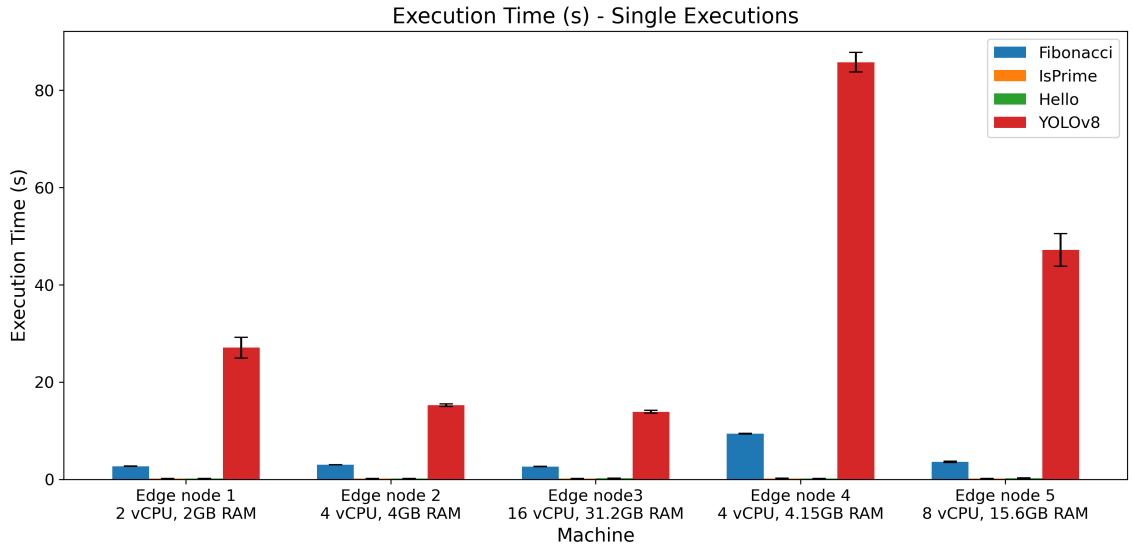


Figure 6.1: plot_Single_Executions_Execution_Time_s.png

**CPU Usage**

The analysis of CPU usage during single execution, derived from Figure 6.2 , shows how the Fibonacci function, which is heavily CPU-bound, occupies about 50% of the available capacity on Edge Node 1 (2 vCPUs), dropping to 25-20% on Edge Nodes 2 and 4 (4 vCPUs), and settling around 10% on Edge Node 5 (8 vCPUs) and Edge Node 3 (16 vCPUs). This trend is consistent with the increase in the number of cores, which distributes the load while essentially keeping the total volume of CPU cycles required by the computation constant.

Regarding YOLOv8, CPU usage is observed at 40-45% on Edge Nodes 1 and 2, decreasing to about 25% on Edge Nodes 3 and 4, and down to 10% on Edge Node 5, confirming a behavior similar to that of the Fibonacci function. In both cases, it appears that single execution does not benefit significantly from the increase in cores, leaving the overall task duration unchanged and simply distributing the load across a broader hardware base.

It should also be noted that the Hello and isPrime functions, characterized by extremely short execution times, do not show significant CPU usage values. In these cases, the monitoring thread fails to reliably sample the metric, since the execution finishes too quickly relative to the measurement interval, resulting in an apparently negligible or zero usage.

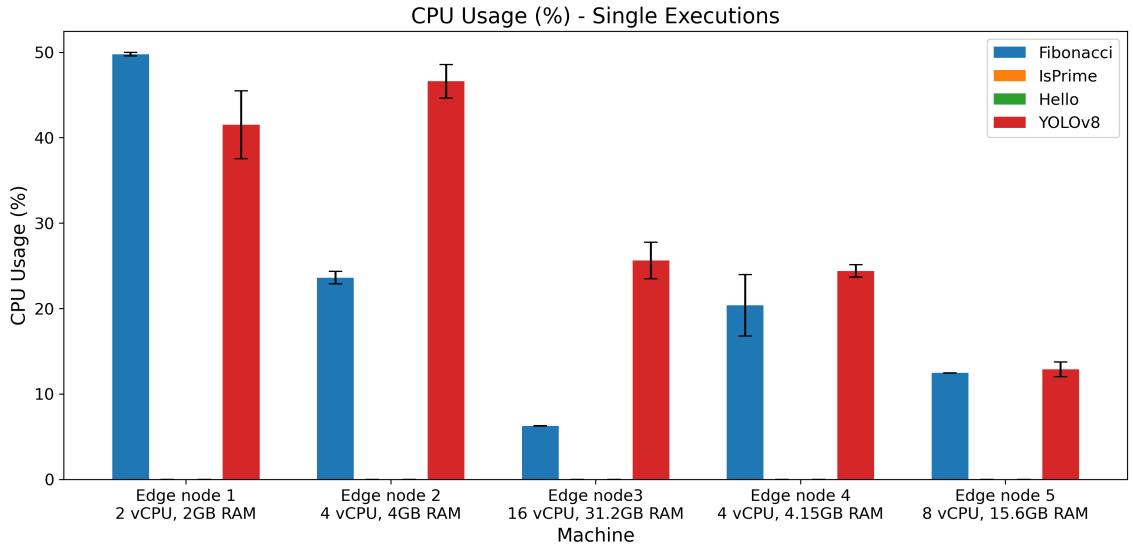


Figure 6.2: plot_Single_Executions_CPU_Usage_perc.png

**Memory Usage**

Regarding memory consumption, the Fibonacci function, typically CPU-bound, shows a modest usage that progressively decreases as the available RAM increases: about 12% on Edge Node 1 (2GB RAM), 5-3% on Edge Nodes 2 and 4 (4GB and 4.15GB RAM), 2% on Edge Node 5 (15.6GB RAM), and less than 1% on Edge Node 3 (32GB RAM), indicating that this workload does not exert significant pressure on memory and mainly scales according to available space.

In contrast, the YOLOv8 function exhibits a much more marked consumption: around 45% on Edge Node 1, decreasing to 25-22% on Edge Nodes 2 and 4, 5% on Edge Node 3, and finally 3% on Edge Node 5, showing how the increase in memory capacity helps reduce the relative load.

Finally, as shown in Figure 6.3, the Hello and isPrime functions have negligible or undetectable CPU usage in the graphs, an expected behavior given the extremely short execution time, which often does not allow the monitoring thread to reliably sample the metric before the task finishes.

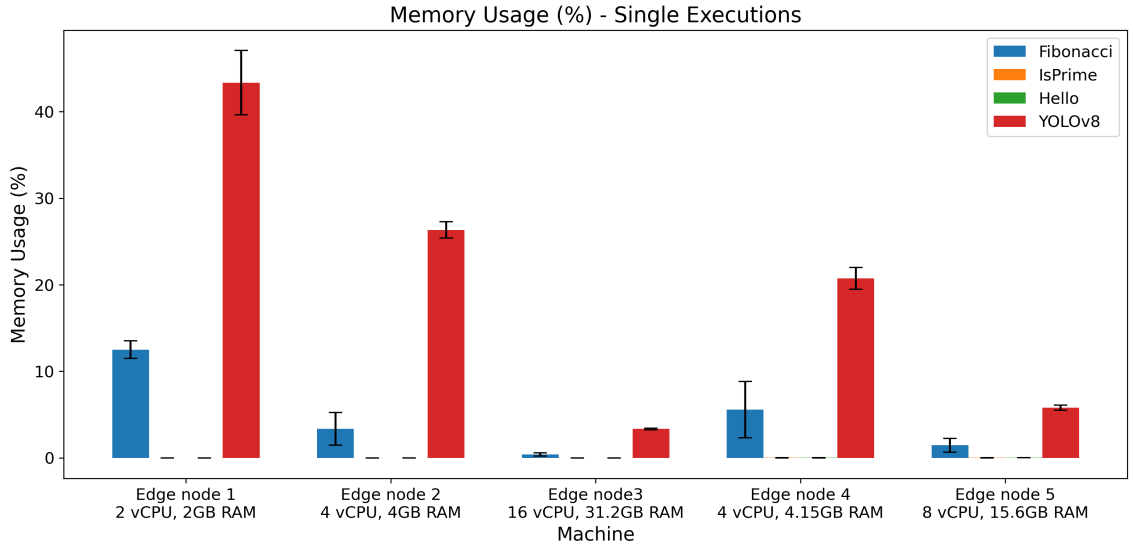


Figure 6.3: plot_Single_Executions_Memory_Usage_perc.png

## 6.3 Parllel execution analysis

**Execution Time**

The analysis shown, referring to 6.4, presents the execution times of various functions run in parallel on the different edge nodes. It is briefly reminded that the Hello and isPrime functions do not show significant execution times since they finish in such a short interval that they mostly escape sampling by the monitoring thread, thus being negligible for quantitative comparison purposes.

Regarding the Fibonacci function, a completion time is observed that progressively decreases from 9 seconds on Edge Node 1 to 5 seconds on Edge Node 2, and down to 3 seconds on Edge Node 3, indicating a gain although not perfectly proportional to the increase in cores. It is interesting to note the exception represented by Edge Node 4, where the time rises to 12 seconds, indicating a possible imbalance in load management, while on Edge Node 5 it returns to 5 seconds, suggesting a stable compromise between the number of cores and overhead.

In the case of YOLOv8, a different behavior emerges: starting from 275 seconds on Edge Node 1, there is a sharp improvement on Edge Nodes 2 and 3 with 50 seconds. However, the time increases again on Edge Node 4, reaching 350 seconds, likely due to inefficient use of memory resources or synchronization, before decreasing to 170 seconds on Edge Node 5, where it still remains far from ideal, highlighting how scalability on many-core architectures requires targeted software optimization.

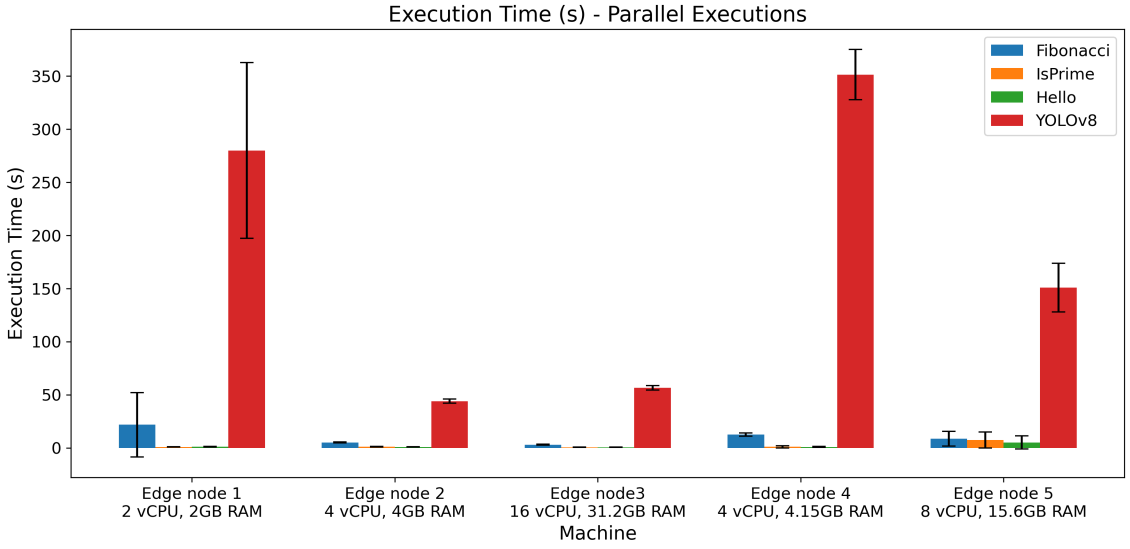


Figure 6.4: plot_Parallel_Executions_Execution_Time_s.png

**CPU Usage**

The following analysis refers to Figure 6.5 and concerns CPU usage during the parallel execution of workloads across different nodes. Below are the CPU usage values for the Fibonacci and YOLOv8 functions, considering the effect of increasing the number of cores.

For the Fibonacci function, parallel CPU usage is around 15% on Edge Node 1 and slightly rises to 17% on Edge Node 2. On nodes with more cores, such as Edge Node 3, usage drops to 7%, while on Edge Node 4 it is 12%, and on Edge Node 5 (8 vCPU) it further decreases to 5%.

In the case of YOLOv8, CPU usage ranges from 17% on Edge Node 1, peaking at 22% on Edge Node 2, and remaining high at 19% on Edge Node 3. On Edge Node 4 it falls to 13%, while on Edge Node 5 it drastically reduces to 4%. Here too, the increase in cores does not translate into linear CPU usage, likely due to load management limitations and potential memory or I/O bottlenecks.

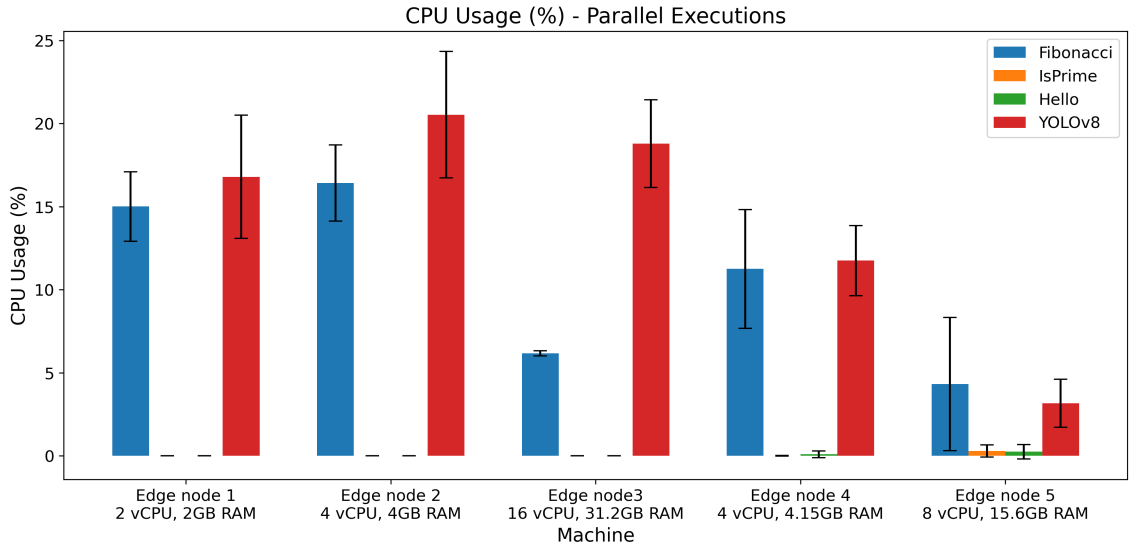


Figure 6.5: plot_Parallel_Executions_CPU_Usage_perc.png

**Memory Usage**

The following analysis, referring to Figure 6.6, examines memory usage during the parallel execution of functions on the various edge nodes.

For the Fibonacci function, memory usage is observed to be around 10% on Edge Nodes 1 and 2, decreasing to 2% on Edge Node 4 and further reducing to 1% on Edge Nodes 3 and 5, highlighting how this CPU-bound workload requires relatively small amounts of RAM even when multiple cores are present.

In the case of the YOLOv8 function, consumption reaches 23% on Edge Node 1, drops to 15% on Edge Node 2, settles around 10% on Edge Node 4, and progressively decreases to 5-3% on Edge Nodes 5 and 3. This trend reflects the workload's ability to more evenly utilize available memory as the nodes' overall RAM increases, limiting saturation phenomena observed on the smaller nodes.

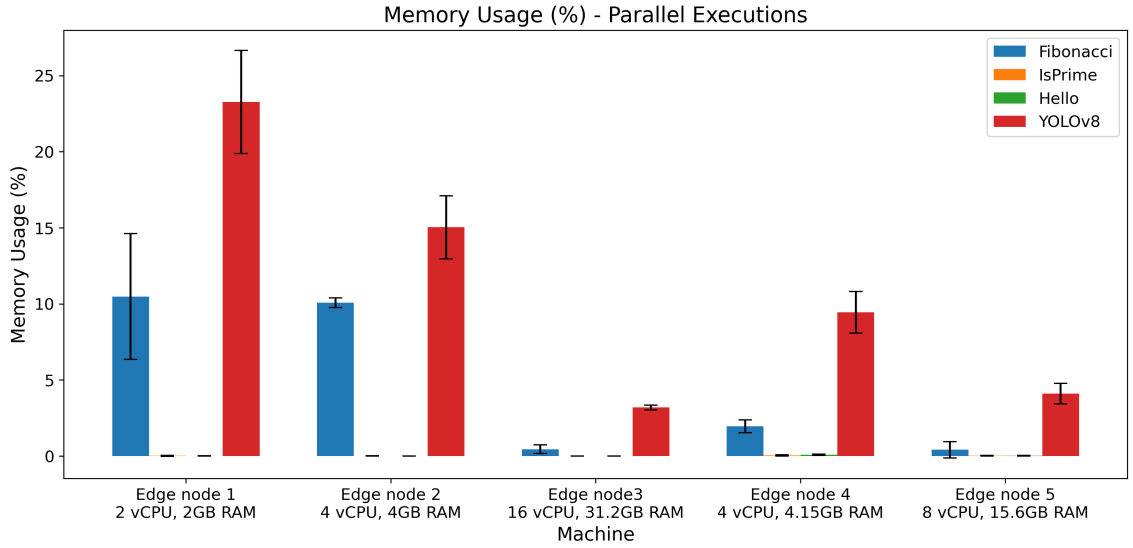


Figure 6.6: plot_Parallel_Executions_Memory_Usage_perc.png

## 6.4 Comparison between single and parallel execution

The overall analysis of the results highlights how adopting parallel execution, compared to single execution, produces different effects on the overall system behavior, both in terms of operation completion time and hardware resource usage. This outlines a scenario where the benefits and limitations of each approach manifest depending on the nature of the function and the infrastructural characteristics of the involved nodes.

Specifically, looking at execution times, it emerges that the parallel approach generally tends to significantly reduce the total duration of the more demanding tasks, as it allows multiple computing units to be used simultaneously; however, this reduction almost never translates into a gain proportional to the number of available cores, because there is inevitably an overhead due to concurrent process management, which adds up with possible inefficiencies in resource allocation and synchronization operations. This means that, while for heavily computational workloads parallelism still brings an improvement in response times, for light or poorly parallelizable workloads single execution sometimes proves equally effective, if not preferable, avoiding introducing management complexity that would not be offset by actual time savings.

Similarly, from the CPU usage perspective, single execution tends to concentrate the load on a smaller portion of the total computational capacity, leading to more evident core saturation. Meanwhile, parallel execution, although distributing the load across a greater number of threads or processes, often fails to achieve full CPU utilization on the more resource-rich nodes, indicating that increasing cores does not automatically translate into a proportional increase in effective parallelism. This phenomenon reveals that beyond a certain threshold, adding computational capacity requires adequate software optimization to prevent part of the resources from remaining unused or overhead due to process/thread scheduling from compromising the expected benefits.

Regarding memory, it is observed that primarily CPU-bound workloads, such as numerical or iterative tasks, maintain a nearly unchanged RAM usage profile regardless of the execution scheme, while more complex applications with intensive data usage benefit from parallel execution, as the workload division tends to distribute memory demand more evenly, reducing peaks on less performant nodes and enabling more efficient overall resource management.

In conclusion, the obtained results confirm that, while parallelism represents a preferred way to speed up the execution of demanding tasks and balance the load among available resources, its real impact is conditioned by the need for careful sizing and appropriate process scheduling to avoid the increase in computational resources resulting merely in overhead shifting rather than a concrete improvement in overall performance.

In light of these considerations, it becomes clear that it is not possible to identify a universal predefined configuration a priori, since the effectiveness of parallel execution compared to single execution strictly depends on the characteristics of the function and the hardware configuration of the edge nodes.

# Chapter 7

# Conclusions

## 7.1 Summary of the work carried out

This thesis work has systematically led to the design and implementation of a comprehensive framework for the deployment, monitoring, and analysis of functions executed in containerized edge environments. Modules were developed to automate the creation of Docker images, to start isolated execution of functions in dedicated containers, and to collect detailed metrics related to CPU usage, memory consumption, and processing times.
The entire system is based on a fully automated pipeline that, through the combined use of libraries such as psutil, docker-py, and threading, along with statistical analysis tools like scipy and matplotlib, enables the collection and processing of data to produce graphs enriched with confidence intervals. This approach ensures a robust quantitative representation of the observed phenomena and allows direct comparisons between different nodes and workloads, consistent with practices adopted in the literature for the evaluation of edge architectures.
The value of the framework lies in its ability to provide a solid methodological foundation to analyze performance in distributed contexts, reducing the operational complexity related to the manual configuration of tests. This makes it possible to perform analyses that are at the same time repeatable, scalable, and applicable to heterogeneous infrastructures, meeting the flexibility requirements typical of modern edge computing scenarios.

## 7.2 Considerations on the obtained results

The detailed analysis conducted throughout this work allowed for a rigorous evaluation of the performance of various workloads executed on a heterogeneous infrastructure composed of both virtualized and physical edge nodes, characterized by different hardware configurations. The tests performed, systematically replicated across multiple iterations, highlighted how the intrinsic characteristics of workloads strongly influence the behavior of computational resources.
In particular, it was observed that strictly CPU-bound functions, such as Fibonacci calculations, tend to benefit only marginally from an increase in the number of cores beyond a certain threshold, due to the inevitable synchronization overheads that outweigh computational gains. Conversely, more complex functions, like YOLOv8, designed for detection and segmentation tasks, demonstrated more favorable scalability on architectures with higher parallelism and abundant memory resources, albeit accompanied by a substantial increase in RAM consumption that, in some cases, reached critical thresholds on less equipped nodes.
These results emphasize the need for a careful balance between hardware resources, application nature, and execution strategy, highlighting how a universal approach is not suitable for edge computing scenarios, which are characterized by high heterogeneity and workload variability.

## 7.3 Possible Improvements and Future Applications

In light of the results obtained, it clearly emerges that the framework developed in this work primarily represents a modular experimental environment capable of automatically executing, monitoring, and measuring the behavior of containerized functions on heterogeneous edge nodes. Equipped with tools for the structured collection of resource usage and execution time metrics, as well as features for statistical and graphical data analysis, this system serves as a valuable support for comparatively evaluating different allocation policies and load distribution strategies. The central contribution of this work lies precisely in the framework's ability to provide a solid methodological basis for testing and comparing scheduling approaches and resource sizing in edge scenarios characterized by significant variability and limited computational resources. The automation of the deployment and execution cycle of functions, combined with the serialization of collected information into easily analyzable formats, simplifies the quantitative analysis of the impact that different allocation methods exert on overall system performance.

From this perspective, particularly interesting are the possibilities of integrating predictive or optimization algorithms—not as intrinsic components of the framework, but as study objects to be evaluated thanks to the monitoring capabilities the system offers. Beyond machine learning models, which would allow estimating and anticipating load peaks to dynamically guide function placement, alternative methodologies can be imagined, such as those based on mixed-integer linear programming formulations used by J-NECORA to optimize service placement along the cloud-edge continuum, or evolutionary strategies based on genetic algorithms and metaheuristics, which are especially effective in tackling multi-objective problems with complex constraints.

Ultimately, the system could naturally be extended through the integration of additional metrics related to energy consumption or end-to-end latency, enabling even more detailed analyses suited to critical contexts such as industrial, healthcare, or smart city applications, where the balance between efficiency and response speed is crucial.

# Bibliography


[1] Introduction to IoT, https://www.researchgate.net/profile/Omkar-Bhat/publication/330114646_Introduction_to_IOT/links/5c2e31cf299bf12be3ab21eb/Introduction-to-IOT.pdf

[2] The relationship between IoT and Big Data, https://www.purestorage.com/it/knowledge/big-data/internet-of-things-and-big-data.html

[3] An Overview on Edge Computing Research, https://ieeexplore.ieee.org/abstract/document/9083958

[4] Performance Evaluation of Container Orchestration Tools in Edge Computing Environments, https://www.mdpi.com/1424-8220/23/8/4008#:~:text=In%20addition%2C%20an%20edge%20service,deployment%20of%20containerized%20applications%20in

[5] Challenges and Opportunities in Edge Computing, https://ieeexplore.ieee.org/abstract/document/7796149

[6] Serverless Computing: A Survey of Opportunities, Challenges, and Applications, https://dl.acm.org/doi/abs/10.1145/3510611

[7] A Decentralized Framework for Serverless Edge Computing in the Internet of Things, https://ieeexplore.ieee.org/document/9193994

[8] Serverless Functions in the Compute Continuum, https://link.springer.com/article/10.1007/s42979-025-03699-7

[9] The State of FaaS, https://ieeexplore.ieee.org/abstract/document/10643918

[10] J-NECORA: A Framework for Optimal Resource Allocation in Cloud–Edge–Things Continuum for Industrial Applications With Mobile Nodes, https://ieeexplore.ieee.org/abstract/document/10886960

[11] Utility-driven virtual machine allocation in edge cloud environments using a partheno-genetic algorithm, https://journalofcloudcomputing.springeropen.com/articles/10.1186/s13677-025-00739-8?utm_source=chatgpt.com

[12] Dynamic resource provisioning in containerized edge systems with reconfigurable edge servers, https://jwcn-eurasipjournals.springeropen.com/articles/10.1186/s13638-025-02450-3?utm_source=chatgpt.com

[13] Docker Documentation, https://docs.docker.com/

[14] Psutil Documentation, https://psutil.readthedocs.io/

[15] Ultralytics YOLO Documentation, https://docs.ultralytics.com/