Politecnico di Torino

Polytechnic of Turin

Computer Engineering

Academic Year 2025/2026

Graduation Session March 2026

Experimental Comparison between GraalVM with Micronaut and Spring Boot in a Microservices Architecture

performance analysis and load testing

Supervisor:
Antonio Vetrò

Candidate:
Giuseppe Maria Barone

Abstract

In the past few years the transition from monolithic applications to microservices has become increasingly relevant due to challenge such as scalability, resilience, and team organization. In this scenario, the role of Spring Boot applications has become essential for developing backend technology for large enterprise systems using Java. This framework helps developers build these applications by introducing abstraction layers and automated configuration mechanisms. However, new frameworks have emerged in recent years. Thanks to the advent of cloud systems, new frameworks have been developed, emphasizing performance, and one of these is Micronaut. Introduced in 2018, Micronaut is a JVM-based framework that is designed for building easily testable microservices and serverless applications.

This thesis explores monolithic and microservices software architectures, examining their respective advantages and disadvantages, and explaining the rationale behind choosing one over the other. It then investigates the two chosen frameworks, paying attention to the underlying aspects that lead to significant performance improvements.

The theoretical part is followed by a practical section explaining the software developed during the internship at Technology Reply, by documenting it using the C4 model and highlighting the differences in development between the two frameworks. The practical section concludes with an evaluation of the software, detailing the choice of each selected metric following the Goal–Question–Metric (GQM) approach. These metrics are evaluated using load testing which progressively increases the number of parallel requests, demonstrating how the system's performance reacts to these changes.

The results indicate that Micronaut performs better than Spring Boot with a smaller memory footprint, while throughput and the average response time remain stable. This demonstrates that Micronaut has more efficient memory management than Spring Boot, which could be a deciding factor in many cases. This study is useful for the company in order to evaluate whether to use Micronaut or Spring Boot for its future projects.

Finally, during the development of this thesis, several areas for improvement were identified to make the application more modular and configurable. In the future, the company intends to evolve the solution by integrating it into an ecosystem of multi-agent applications based on artificial intelligence, further expanding its capabilities.

Table of Contents

List of Figures

Glossary

DBMS
: Database Management System

MS
: microservice

VM
: Virtual Machine

DS
: Distributed System

API
: Application Programming Interface

CLI
: Command Line Interface

HTTP
: HyperText Transfer Protocol

Chapter 1

Overview of Microservices Architecture

Before starting the research, it was important to understand why microservice systems emerged and what needs they address, to analyze their key characteristics in depth and how they fit into the context of software engineering, and giving some real-world examples of microservice systems.

1.1 Introduction to Microservices Architecture

Before diving into microservice architecture, we need to explore the principles behind its architecture and the factors that led to its creation.

1.1.1 Definition and Fundamental Principles

According to Newman [1]: *'Microservices are small, autonomous services that work together'*. In other words microservices are entities that implement a specific business function and are completely autonomous, yet collaborate with others for achieving a common goal.

Microservices represent an architectural approach designed to break down large monolithic systems into smaller, independent components that communicate with each other through lightweight protocols. Each microservice is a small component, which can be developed, tested and distributed in isolation from others. This paradigm prevents the entire information system from being strictly dependent on a single framework or programming language. Each microservice can adopt the technology best suited to its functional and performance requirements in order to perform its assigned business function. Further advantages derive from the distributed nature of microservices: firstly, the possibility of geographical

distribution, as microservices do not necessarily have to be executed within a single physical machine but can be distributed across multiple virtual environments; secondly, greater resilience, since the malfunctioning of one microservice does not compromise the entire application. Regarding their size, there is no fixed standard, nor can one simply rely on lines of code. Typically, a microservice should be small enough to be rewritten in about two weeks"*The code base is small. The service can be rewritten and redeployed in 2 weeks*" [1]. Based on these characteristics, there are six fundamental principles of microservices design [2]:

I. Autonomy Each microservice is independent and can be developed and deployed separately from the others without affecting the overall system. This means that each service has its own runtime and data schema, ensuring greater performance reliability and overall better service quality.

II. Loose coupling The principle of *"loose coupling"* involves reducing the interdependencies between microservices, which allows systems to be decoupled by standardizing on contracts as expressed through business-oriented API [2]. This means that a change within one service does not affect the proper functioning of the others. By communicating through stable APIs, consumers are protected: the internal implementation of a service can change without impacting the external systems that rely on it. Furthermore, this technical independence translates into greater efficiency and agility in the architecture, which helps to reduce coordination costs and achieve faster results.

III. Reuse Reuse continues to be one of the cornerstones of microservice architecture. While it is widely used in the construction of large-scale information systems and not exclusive to microservices, it proves equally valuable in this domain. Indeed, if multiple microservices share common business logic, it is preferable to externalize that call and move it to a dedicated microservice.

IV. Fault tolerance As mentioned by Algirdas Avizienis, *"Fault tolerance means to avoid service failures in the presence of faults."* [3]. In a microservice context, this means that if one component fails, the system must be able to continue functioning with a degradation of performances, rather than collapsing entirely. Fault tolerance is therefore the ability of a service to operate even in the presence of malfunctions, minimizing the impact on the SLA[1]. In microservice systems, communication with the malfunctioning microservice is interrupted by specific protection mechanisms, such as the so-called *circuit breaker*, which allows the

[1]Service Level Agreement: an agreement that defines service quality and availability metrics.

failures of individual components to be isolated, preventing them from spreading and compromising the entire system.

V. Composability Composability is the principle according to which a system can be easily composed, which means a single service can be viewed as an autonomous entity, both as part of a set of microservices and, when combined, can generate new services aggregate with additional functionality. This approach helps to keep the overall architecture less complex and more modular.

VI. Discoverability All APIs exposed by a microservice must be clearly documented and made easily accessible to other teams or microservices. Proper documentation allows consumers to understand the microservices' business and the interfaces it provides. Effective communication of APIs ensures that developers can easily integrate them into the microservices they develop, avoiding the duplication of existing functionality.

1.1.2 Modularity and Service Independence

According to Microsoft's architectural guide, building a microservices application means adopting a design style which aim to break down a complex problem into smaller, autonomous services [4]. To achieve this, each microservice must strictly follow the principle of *separation of concerns*, by performing only the specific task assigned to it and interacting with other microservices.

For example, a microservice architecture should be divided into autonomous components, as illustrated in Figure 1.1.

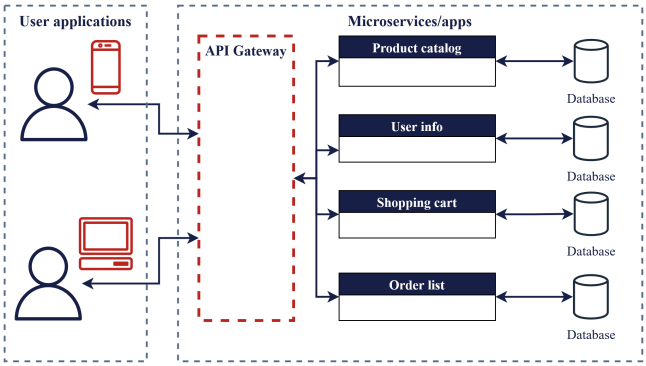


Figure 1.1: Example of a microservice architecture

As shown in Figure 1.1, each microservice handles a specific part of the application, performing particular assigned tasks and writing data to a database to which only it has access. This approach decentralizes breakpoints, making it easier to identify any problems when part of the application is not working correctly.

Furthermore, this structure guarantees the independence of microservices: each service can operate autonomously, even in the absence of others, without knowing or depending on their existence. This independence allows, as discussed in the previous paragraph, the development of microservices with completely different technologies. This is called Polyglot Architecture, which allows engineers to use the best possible technology for different features in development [5].

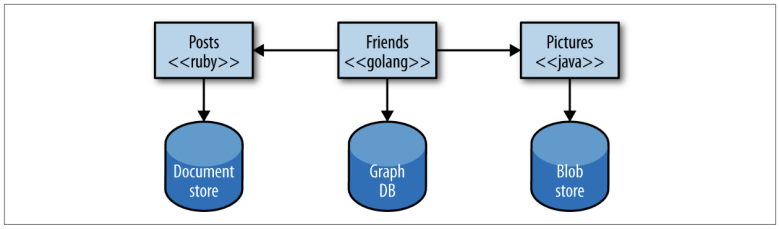


Figure 1.2: Example of Polyglot Architecture in a Microservice Architecture [1]

1.1.3 Scalability, Resilience, and Maintainability

Scalability is a fundamental factor in the design of a microservices system. A system is defined **scalable** if it is able to handle an increasing number of requests and workload without degrading its performance or becoming unstable.

There are two types of system scalability:

- **Vertical scaling (Scaling Up)**: The ability of a system to manage a growing workloads and requests by expanding computational resources of a single node, such as storage, CPU, RAM, and other hardware components. The scaling up approach is limited by currently available hardware resources.

- **Horizontal scaling (Scaling Out)**: The ability of a system to manage the increasing workload by changing how an architectural system is defined, for instance adding the number of nodes deployed in a system. This approach is particularly used in the real world, where distributed systems consist of multiple nodes with limited computational capacity, which guarantee the ability to handle a greater workload by working in parallel.

The horizontal scaling, which has the greatest impact in the real world of distributed systems, can be implemented by several techniques, such as:

- **Partitioning and Work Distribution**: consists of splitting a task into smaller subtasks and distributing them across different nodes of the system in order to balance the workload. In this way, if requests increase, it is possible to add more processes on more machines to handle them effectively.

- **Replication**: involves the ability to create copies of processes or services, distributing the workload or data among these replicas to improve performance. For instance, it is possible to have multiple replicas of the same microservice distributed across different nodes, in order to increase the access speed and availability of the service. Web server clustering[2] is a classic example of how multiple servers can distribute the load and increase availability.

- **Communication Latency Hiding / Limitation**: aims to reduce communication latency between services. Asynchronous communication is often preferred over synchronous communication in order to minimize response times and improve overall system efficiency.

[2]A server cluster is a collective group of servers distributed and managed under a single IP address. [6]

Universal Scalability Law

To understand the limits of system growth, it is essential to provide an introduction on how the system reacts to increased scalability. The **universal scalability law** (USL), proposed by Australian computing systems researcher Dr. Neil Gunther, studies the performance behavior of a computing system as the number of nodes or resources increases. According to the USL, there is a maximum scalable capacity threshold beyond which increasing the number of nodes does not result in a linear increase in performance. For example, suppose we have a server that handles 2,000 transactions per second per node. By increasing the number of nodes to two, we would expect a throughput of 4,000 transactions per second. In reality, this is not entirely true, as their final throughput is slightly less than 4,000 transactions per second. This discrepancy becomes more evident as the number of nodes increases, until a threshold is reached beyond which the system not only stops scaling, but may even show a *retrograde speedup*, i.e. a decrease in performance despite the addition of resources.

The behavior described by the USL is illustrated in Figure:1.3

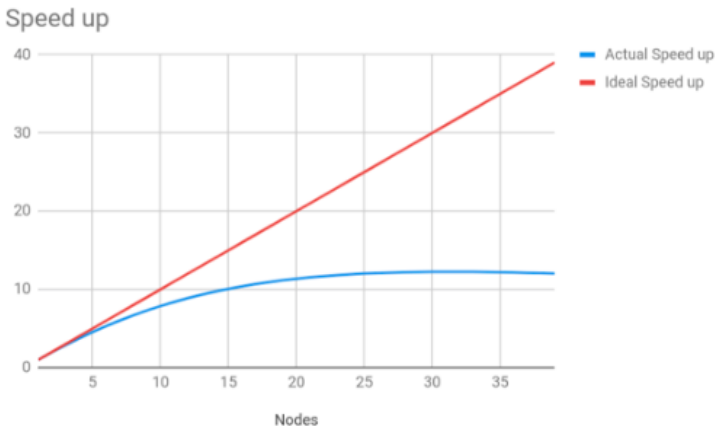


Figure 1: Ideal linear speed up vs actual speed up

Figure 1.3: Graph that shows the trend of a scalable up system [7]

For this reason, the universal scalability law has been proposed, according to which the throughput of a system can be approximated by the following equation (derived from queueing theory) [8]:

$$X(N) = \frac{N \cdot \gamma}{(1 + \alpha \cdot (N-1) + \beta \cdot N \cdot (N-1))} \tag{1.1}$$

where:

- N is the number of concurrent units;
- γ represents the normalized throughput of the system with a single unit;
- α indicates *resource contention*, that is the cost of contention for shared resources (e.g. databases, locks, queues or centralized components);
- β represents the *coherency cost*, that is the cost of synchronization and communication between system units (e.g. message exchange, state replication or coordination between microservices).

Resilience

Resilience is the ability of a system to recover from a state of continuous failure, crucial in microservice systems as it represents how stable and operational a system is. This necessity is amplified in distributed environments, where the high number of interacting services inherently increases the probability of failure due to network latency, hardware degradation, and software faults.

These are the fundamental pillars of resilience in a microservices architecture:

- **Fault Isolation**: A microservices architecture guarantees resilience in the event of a single service failure, ensuring that as an individual service is isolated and prevented from cascading into a complete system outage.
- **Continuous Availability**: The system is divided into smaller services, and it is possible to replicate them in different parts of the globe, which means that a higher level of availability can be maintained even when several services are unavailable.
- **Scalability and Elasticity**: Resilience is closely related to the scalability of a system, because if the capacity of a system increases, it reduces the number of failures it encounters. In addition, if a system can scale autonomously, i.e. adding or removing nodes, resilience will be better guaranteed than in a completely static system that cannot be scaled.
- **Improved User Experience**: The resilience of a system can lead to an improvement in the overall user experience because it ensures that, even in the event of a failure, an application provides the necessary functionality without compromising the overall user experience by implementing the *graceful degradation* technique.
- **Quick Recovery**: Resilient systems can recover from failures automatically or with less manual intervention than a monolithic application. This reduces downtime and all impacts on service.

- **Fault Tolerance**: Resilient systems provide important mechanisms to increase fault tolerance, such as **circuit breakers** (isolating a system), **timeouts**, and **retries** to handle all errors and degrade performance without causing service disruptions.
- **Decentralised Communication**: A resilient system is often based on a system that uses a decentralized approach as a means of communication, such as asynchronous messaging or event-driven architectures. This ensures that in the event of communication problems, the system continues to function.
- **Continuous Testing and Deployment**: A resilient system must undergo rigorous testing and continuous deployment practices. Automated testing, canary deployments, and blue-green deployments help ensure that changes are rolled out safely and do not introduce vulnerabilities or instabilities.

Challenges in Achieving Resilience

A resilient microservices system brings considerable challenges. Some of these are:

- **Distributed Complexity**: In a microservice application, although it is a useful approach, it could introduce complexity in monitoring, debugging, and tracing across the entire system. So, understanding how each service interacts and ensuring fault isolation becomes challenging [9].
- **Inter-service Communication**: Microservices deeply rely on communication between services, often over networks. This introduces network failures, latency, and potential communication bottlenecks, requiring robust communication protocols and error-handling mechanisms.
- **Data Consistency and Integrity**: One of the most challenging aspects in a microservice architecture is maintaining data consistency across each microservice, especially within distributed transactions. A time-consuming feature, which is hard to design and maintain, is ensuring data integrity and synchronization without introducing performance bottlenecks or single points of failure.
- **Resilience Testing**: Testing the resilience of microservices systems is complex and often requires specialized tools and techniques. It demands various failure scenarios, such as network partitions, service outages, or latency spikes. However, it could be a game changer for guaranteeing system stability.
- **Dependency Management**: In big information systems, microservices are strictly interconnected, making them dependent on external services and APIs. Managing dependencies and handling versioning, backward compatibility, and service discovery become critical in maintaining system resilience [9].

- **Scalability and Resource Management**: In highly dynamic environments, the dynamic scaling of microservices to handle varying workloads requires advanced orchestration mechanisms and efficient allocation of computing resources based on demand.
- **Security and Compliance**: Security in microservice environments encompasses the protection of communication channels as well as the implementation of multiple security protocols, such as distributed authorization, access control and authentication management. Additionally, compliance with regulatory requirements further increases the complexity of achieving system resilience [9].

Maintainability

Maintainability how a microservice can be easily modified and deployed without negatively affecting system performance. In a microservices architecture, this quality is strictly linked to service isolation. Breaking a system into independent and self-deployable units allows teams to test their services and make changes independently of other developers, simplifying distributed development [10]. However, as each service is part of a much larger ecosystem while following its own independent development lifecycle, organizational maintainability becomes a critical factor. Although this autonomy allows services to evolve without strictly conforming to centralized standards, it creates risks regarding API stability and documentation. When individual microservices are developed and managed by different teams, technical inconsistencies can lead to significant integration challenges. Consequently, strict governance policies and well-defined versioning strategies are required to prevent integration issues in a heterogeneous microservice environment.

Unfortunately, microservices architecture may suffer from several maintainability issues, particularly in environments characterized by rapid service development, and can lead to sudden growth. The most common issues are:

- **Code Duplication and Divergence**: Since different teams move quickly, codebase often lack a clean designed architecture. As a result, APIs and business logic may change, making API tracking more challenging.
- **Inconsistent Standards**: Different teams adopt different practices for managing microservices, such as logging, error handling, and REST patterns.
- **Tight Coupling Through APIs**: Services may rely too heavily on each other's internal implementations or synchronous APIs [11].
- **Over-Proliferation of Services**: The "micro" mindset could encourage teams to split microservices even more, leading to an over-proliferation of services.

- **Operational Complexity**: Scaling services add complexity to deployment, monitoring, logging, versioning, and rollback procedures.
- **Tooling and CI/CD Gaps**: New services may bypass shared pipelines or monitoring setups.
- **Hidden Technical Debt**: Shortcuts are taken to meet scaling demands or accelerate release cycles.

Maintainability tends to degrade during rapid scaling, as speed is often prioritized over structural integrity. Without enforced standards, appropriate tooling, and architectural discipline, the microservices landscape can become chaotic, costly to modify, and difficult to understand [12].

1.1.4 Case Studies in Real-World Scenarios

Two pillars that have changed the modern world and use microservices architecture in their systems are: Amazon, the global online retail giant; and Uber, where the company has been changed by implementing a new easy way of transport.

Case Study: Amazon

The Amazon case highlights the differences between monolithic and microservices architecture by illustrating the company's shift from a monolithic system to a microservices system. Founded in the early 2000s, Amazon initially developed its retail information service as a large monolithic application, which supported its rapid growth. However, this architectural choice soon required continuous code modifications and upgrades with considerable attention, since in a monolithic application even a smaller error can result in a total system failure. At first, the monolithic architecture brought effective results within Amazon. Nevertheless, as the development team expanded, working on a single, shared codebase became increasingly difficult. For this reason, they decided to change its architectural approach. In 2001, Amazon's engineering team decided to refactor the codebase from scratch, decomposing the monolithic system into smaller, independent subsystems. The adoption of a microservices architecture significantly benefited the company by improving maintainability, enabling independent modification of features and resources, and increasing overall system efficiency. Amazon developers analyzed and decomposed the overall architecture into microservices, a particularly complex approach due to the high degree of coupling. But once the decomposition was completed, services were distributed among different business units. The adopted approach consisted in dividing the system into web service interfaces, assigning them to different teams in order to resolve system bottlenecks. They were successfully resolved due to the reduced size of each team and of the codebase on which

they worked. This organizational structure ensured greater attention to detail for each microservice. For example, separated services were developed for the "Buy" button on a product page, for the tax calculator function, and for other individual functions. Each function was thus implemented as an independent service [13]. The result of this approach is illustrated by the microservice architecture adopted by Amazon in 2008, shown in Figure 1.4.

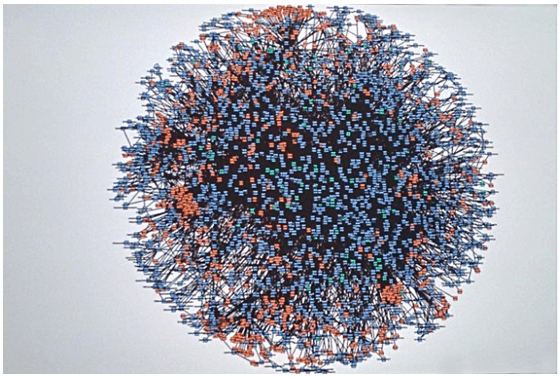


Figure 1.4: Real example of microservice architecture in Amazon [13]

Case Study: Uber

Similar to Amazon, Uber started with a monolithic system, despite being founded in 2010. As in Amazon's case, Uber recognized the limitations of a monolithic architecture; however, the company initially adopted this approach to prioritize development speed and rapid deployment within a single city. In particular, the development team faced challenges with efficient release of new features, bugs fixing, and the rapid support of expanding global operations. Initially, Uber's architecture was structured as follows: Passengers and drivers were connected to Uber's monolith through a REST API [13]. Within the monolithic application, three adapters with embedded APIs handled functions such as billing, payments, and text messages [13]. A single SQL database supported all operations performed on the monolith, and all features were implemented within it.

To address these difficulties, Uber decided to decompose its architecture from a monolithic system into microservices. Subsequently, developers built individual microservices for functions such as passenger management, trip management, and others [13]. All these services are accessible through an API gateway. By moving towards a microservices architecture, the Uber team immediately realized the advantages it offers. First, the system became more maintainable for developers. As a consequence, software evolution accelerated, as fast scaling became easier to achieve. One of the main problems encountered by the Uber team was the

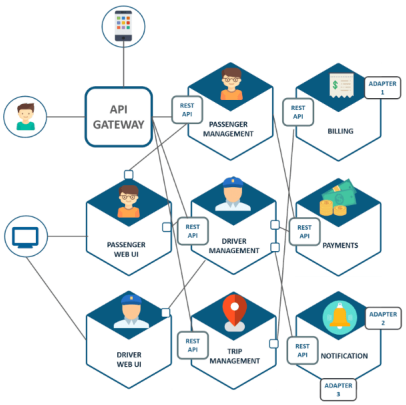
Figure 1.5: Monolithic architecture of Uber [13]

coordination of these microservice teams. For this reason, the company adopted a more documentation-oriented approach, aiming to remove the technical debt that many companies accumulate by writing adequate documentation from the outset. This approach helped prevent confusion among teams and microservices, a problem previously encountered in section 1.1.3. The three key steps for implementing microservice standards at Uber included group buy-in, determining organizational production ready requirements, and making production readiness part of the engineering culture. These requirements needed to be quantifiable and verifiable through testing.

"It is a long process, but it makes a big difference" said Fowler. *"Developers all want to make the best thing they can. Standardization is not a gate, it is not a hindrance. It is something you can hand developers, saying, 'I know you can build amazing services, here's a system to help you build the best service possible.' And developers see this and like it."* [14]. First, the team analyzed the principles that contributed to system availability, such as fault tolerance, documentation, performance, reliability, stability, and scalability. Subsequently, quantitative metrics and business logics accessible to all developers were introduced. Once defined, these metrics were transformed into global standards for every second microservices.

1.2 Monolithic Architecture

Monolithic architecture is one of the oldest and most historically established architectural design patterns in software engineering. It is characterized by a unified and tightly integrated system in which all components are deployed as a single unit. This section outlines the main characteristics and design principles of



(a) System design of microservices in Uber [13]

(b) Actual design of microservices in Uber [13]

Figure 1.6: Representation of microservices in Uber

monolithic architecture.

1.2.1 Structure and Characteristics of a Monolithic Application

In system design, monolithic architecture is defined as a methodology that combines multiple parts of a system into a single codebase. Its origins date back to the early years of the computer era, in the mid-20th century. This approach emerged from the technological constraints of the time: the code was written in low-level languages and developed on a single codebase, and executed on a single machine. Over the years, as technology evolved, new architectures and new programming paradigms emerged. The introduction of object-oriented programming language added abstraction layers over computer architecture, enabling hardware-independent programming. Along the rapid development of networks, new architectures emerged, such as Service-Oriented Architecture (SOA).

In monolithic architecture, all processes are tightly interconnected and they are executed in a single service. This structure, although is simple to develop and deploy, makes the introduction of updates across the entire system more difficult. This approach is well suited for small and simple applications, that require a quick and straightforward initial setup. Despite the gradual shift toward microservices, the monolithic approach may still be preferable in certain scenarios, due to its simplicity and ease of deployment in a single package. These characteristics make it especially suitable for small organizations or development teams with a limited number of members. Monolithic architecture offers several advantages in specific

contexts:

- **Simplicity**: Monolithic architecture enables linear development and deployment. Developers benefit from having the entire system contained within a single package, providing a comprehensive overview of the system.
- **Cost-Effectiveness**: Monolithic architecture can be more economical for start-ups or small-medium companies, as they require less infrastructure overhead compared to distributed architectures.
- **Performance**: Since the entire system is executed on a single machine, monolithic applications may achieve higher performance due to reduced communication overhead between components.
- **Security**: With fewer communication endpoints between services, monolithic systems drastically reduce vulnerable points. When adequate security measures are implemented, this can result in improved system security.

In addition, monolithic architecture provides several features:

- **Single Codebase**: The entire software system is developed, managed, and deployed from a single repository. This approach simplifies program management and implementation. [11]
- **Tight Coupling**: The architectural components are highly interdependent. Since they operate within the same environment, a modification to one often necessitates changes in others, increasing the risk of side effects and regressions.
- **Shared Memory**: Monolithic applications typically share the same memory space, allowing components to communicate efficiently without needing network overhead.
- **Centralized Database**: Unlike distributed systems, monolithic components relies on a single database schema, typically using a single database instance for all data storage needs.
- **Layered Structure**: The system is traditionally organized into horizontal layers (such as Presentation, Business Logic, and Data Access). While this separation improves clarity, it may still introduce dependencies across layers.
- **Limited Scalability**: Since the entire application must be scaled as a whole, scaling a monolithic application can be challenging and often leads to inefficiencies and increased resource consumption.

Monolithic system design focuses on preserving manageability, consistency, and simplicity within a single codebase. Some of the key design principles include:

- **Modularity**: Although a monolithic system is physically unified, it is crucial to structure the code in a modular way to improve maintainability.
- **Separation of Concerns**: Application components are responsible for separating tasks according to the separation of concerns principle. This approach simplifies debugging and improves code organization by clearly distinguishing user interface logic from business and data access logic.
- **Scalability**: Scalability design involves architecting the system to support horizontal scaling, by adding multiple istance of the same application when required. This may include introducing asynchronous processing for resource-intensive operations, employing caching strategies, or optimizing performance-critical components.
- **Encapsulation**: Encapsulation refers to exposing only the necessary interfaces while concealing the internal operations of a component. By enforcing these boundaries, dependencies are minimized, preventing internal changes from propagating throughout the system and simplifying maintenance and evolution.
- **Consistency**: Maintaining consistency in coding styles, architectural patterns, and design principles across the entire codebase ensures clarity and predictability for developers.

1.2.2 Conceptual Comparison with Microservices: Advantages and Disadvantages

The adoption of monolithic and microservice architectures presents both advantages and disadvantages. Microservices architecture, being the result of breaking down systems into smaller ones, is first and foremost a significant advantage when working with a large codebase, making it more manageable and divisible among the various development teams. Furthermore, as this architectural pattern is decomposed into several systems that can be combined with each other, it is much easier to scale and add more services that perform atomic operations than with a monolithic architecture. Microservices are also a more resilient pattern than a monolithic one, as if one service stops working, the overall performance of the system is not compromised. Unfortunately, like any pattern, microservice architectures also have disadvantages, which, if not addressed, can still lead to serious malfunctions.

- **Increased Complexity**: A microservices system must necessarily manage communication among multiple services, which could increase complexity in data management, in particular to consistency and communication. Any change in the response of one service may require corresponding changes in how the calling service interprets that response.

- **Distributed System Overheads**: When adopting a microservice architecture, a network communication between services must be established. This introduces overhead compared to in-process communication in monolithic systems and can lead to increased latency and potential performance issues.

- **Data Management Challenges**: Each microservice is responsible to manage its own data. However, in some cases, resources may be shared across multiple microservices, introducing data consistency and distributed transaction challenges.

- **Increased Deployment and Operational Overhead**: Microservices architecture requires appropriate infrastructure and tooling [15]. Since multiple independent systems are managed, continuous integration and deployment processes can become more complex, requiring advanced monitoring and orchestration solutions.

- **Inter-Service Communication Issues**: Microservices communicate through a network, introducing issues that are typically absent in monolithic systems, such as network failures, latency, and the need for proper API versioning and management.

- **Testing Complexity**: Testing microservices is generally more complex than testing a monolithic application, as it requires validating that each service functions correctly both in isolation and in interaction with other services.

On the contrary, monolithic systems offers advantages that, in certain contexts, may be preferable to a microservices architecture.

- **Simplicity of debugging**: Debugging a monolithic application is relatively simple, as all application logic resides within a single codebase, making it easier to trace the flow of the program.

- **Simplicity of testing**: The entire system can be tested more easily, since breakpoints occur only within a single service rather than across external services.

- **Simplicity of deployment**: The application is deployed as a single file. Often including both frontend and backend components. This reduces deployment complexity and infrastructure requirements.

- **Simplicity of application evolution**: The evolution of a monolithic application is easier to monitor and observe, as the code is clearer and more sequential rather than the monolithic application.

- **Cross-cutting concerns and customizations are used only once**: Monolithic systems also simply the management of cross-cutting concerns such as security, logging and monitoring, because the entire system follows already defined standard.
- **Simplicity in onboarding new team members**: A monolithic system is easier for new team members to understand, as they can have an overall view of a single codebase.
- **Low cost in the early stages of the application**: Monolithic architectures typically incur lower costs during early stages of application development. With all source code located in one place and deployed as a single unit, both infrastructure and development costs are minimized [16].

Of course, monolithic systems also present several limitations, such as:

- **Long Deployment Cycles**: Compared to microservices, monolithic systems require much longer deployment times. Although the architecture is simpler, testing and deployment cycle on all units are needed, which makes the deployment process time-consuming.
- **Risk of Downtime**: In a monolithic system, even a small bug can cause the entire system to go offline.
- **Limited Scalability**: Monolithic systems can generally scale vertically, which makes its extremely inefficient when the number of users grows.
- **Resource Consumption**: Monolithic applications often consume more memory and CPU resources compared to lighter, distributed architectures such as microservices. This approach can lead to reduced efficiency and increased infrastructure expenses.
- **Limited Flexibility**: Making changes to the system can be challenging, as modifications require altering several areas of the codebase. This increases the possibility of introducing errors or inconsistencies.
- **High code coupling**: Although it is possible to keep a clear service structure inside a single repository, practical experience shows that parts of the codebase often evolve into a spaghetti code[3]. As a result, the system becomes more difficult to understand, especially for new team members [16].

[3]Spaghetti code is computer source code that encodes control flow that is convoluted, and therefore, hard to understand.

In conclusion, neither monolithic nor microservices architecture can be considered universally superior. By analyzing the evolution of major technology companies and carefully evaluating the advantages and disadvantages of each approach, it is evident that monolithic architecture remains useful and appropriate in specific cases. In particular, for small companies with limited development teams, constrained resources, and stable traffic requirements, a monolith often represents the most efficient choice. In such scenario, adopting a microservices architecture could introduce unnecessary complexity. Conversely, when a system is expected to grow rapidly, adopting a microservices approach from the outset may be preferable. This approach helps to avoid the extensive refactoring at later stages, as demonstrated by large-scale technology companies such as Amazon and Uber.

1.3 Architectural Patterns and Integration Mechanisms in Microservices

A wide range of architectural design patterns has emerged throughout the history of computer science. However, the most widely used and important ones in a real-world context are: API Gateway, Database per Service, Service Discovery, Circuit Breaker, Backends for Frontends. Each of these architectural patterns can be implemented through different tools, technologies, and algorithms.

1.3.1 API Gateway Pattern

The API gateway pattern provides a single entry point for external systems to access into microservices. Its usage reduces the risk of directly exposing microservices externally, thereby improving security standards. In addition, it minimizes the number of roundtrips between clients and services by collecting responses from multiple services, which significantly improve system performance. The API gateway also ensures separation of concerns by implementing cross-cutting features such as registration and authentication, thus avoiding redundancy and promoting consistency across all microservices.

An API gateway can be implemented either from scratch, or by adopting third-party tools such as Amazon API Gateway, Kong, and Azure API Management [17]. Third-party tools offer several advantages, including high scalability, robust support, regular updates, and reliability. Moreover, they provide advanced services such as caching and monitoring, which would otherwise be challenging to implement from the beginning.

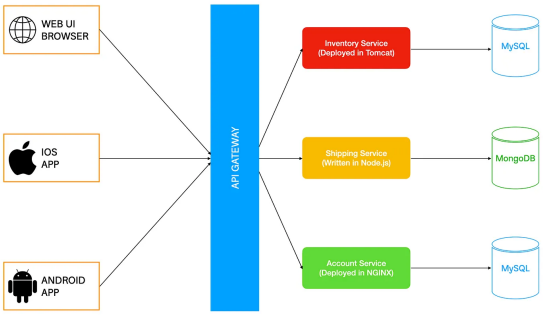


Figure 1.7: API Gateway Architecture [17]

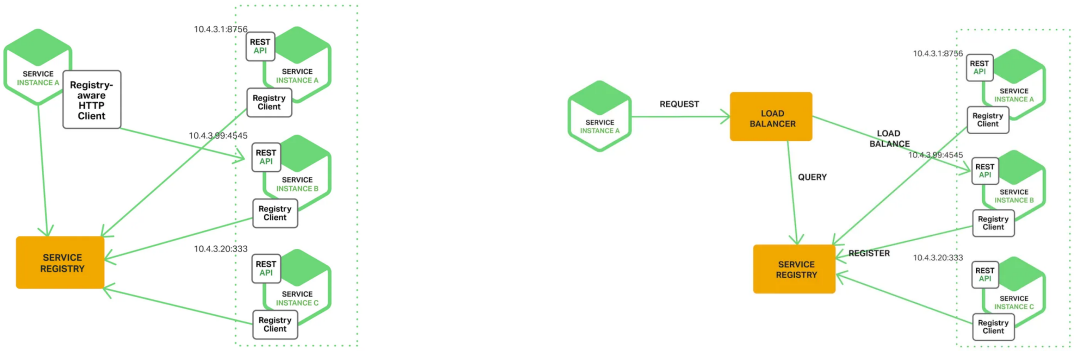
1.3.2 Database per Service Pattern

The Database per service design pattern enforces the principle that each microservice manages its own database. This decentralized approach promotes data encapsulation, autonomy, and scalability, allowing each service to evolve independently without impacting others. Furthermore, it enables each service to adopt the database technology best suited to its requirements, such as SQL, NoSQL, or graph databases. By exposing data exclusively through service APIs, this pattern strengthens security and preserves data integrity, preventing unauthorized access to the database. However, the Database per Service pattern also introduces several challenges. It increases overall system complexity, particularly in terms of data management and consistency. Data duplication may occur, leading to redundancy and synchronization issues. In addition, managing distributed transactions and cross-service queries becomes more difficult, and robust synchronization mechanisms are crucial to ensure data consistency across services.

1.3.3 Service Discovery Pattern

The Service Discovery Pattern is an architectural approach whose purpose is to manage an increasing number of microservices without requiring each service to be aware of all others. As illustrated in Figures 1.4 and 1.6b, tracking microservices in a real-world context is a complex and laborious task. For this reason, the Service Discovery pattern simplifies this process by enabling services to discover each other. Two main approaches to Service Discovery are: **Client-Side Discovery** and **Server-Side Discovery**. In Client-Side Discovery pattern, a "Service Registry" component tracks the locations of available service instances. When a **consumer** needs to access a particular service, it queries the registry to obtain a list of available **provider** locations [18] and then selects one to handle the request. In contrast, the Server-Side Discovery Pattern requests a DNS name. The benefit

is that the responsibility is handled by the deployment platform. However, it introduces a degree of coupling with the deployment platform that manages the Service Registry [18].



(a) Example of Client Side Pattern [18]

(b) Example of Server Side Pattern [18]

Figure 1.8: Representation of two Service Discovery Patterns

1.3.4 Circuit Breaker

The Circuit Breaker architectural pattern is widely adopted in microservices environments to prevent cascading failures within an application. In scenarios where microservices depend on the responses of other services, a failure in one component may lead to a series of errors, which can be avoided by introducing an isolation mechanism that temporarily blocks communication with failing services. The Circuit Breaker pattern usually operates in three states: Closed, Open, and Half-Open. Each state represents a different phase in managing interactions between services [19].

- **Closed State**: In the Closed state, the circuit breaker allows requests to flow normally between services while monitoring the health of the downstream service by collecting and analyzing metrics such as response times, error rates, and timeouts.
- The circuit breaker transitions to the Open state when the monitored metrics breach predetermined thresholds, signaling potential issues with the downstream service. In this state, requests to the failing services are immediately blocked. This isolation prevents cascading failures and allows system stability by ensuring timely feedback to clients, even when services encounter issues.
- **Half-Open State**: After a specified timeout period in the Open state, the circuit breaker enters the Half-Open state, during which a limited number

of "probe[4] requests" are allowed to pass through to the downstream service to test its recovery. If these requests succeed, the service is deemed healthy, and the circuit breaker resets to the Closed state. Conversely, if the probe requests fail, implies that the issue persists, and the circuit breaker reverts to the Open state to prevent further strain.

- **Open State**: When monitored metrics exceed predefined thresholds, the circuit breaker transitions to the Open state. In this state, requests to the downstream service are immediately blocked to prevent cascading failures and to provide fast failure responses.

1.3.5 Backend for Frontend Pattern (BFF)

The Backend for Frontend (BFF) pattern addresses the challenge of supporting different client applications by providing a dedicated backend for each type of client, such as web, desktop, voice assistant, mobile, or IoT devices. While being similar to the standard API Gateway pattern, the BFF approach differs by exposing **multiple specialized interfaces** rather than a single unified entry point. Adopting a BFF

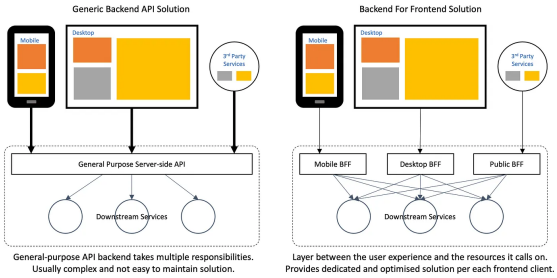


Figure 1.9: Difference between general-purpose API and BFF

architecture enables development teams to accelerate delivery time by providing frontend teams with tailored backends. This approach allows client applications to evolve without impacting other systems or teams, as a separate backend is optimized for a particular case.

[4]A probe request is a limited and controlled test request sent to a downstream service during the Half-Open state of the Circuit Breaker pattern in order to assess whether the service has recovered from previous failures.

1.4 Orchestration and Containerization with Docker and Kubernetes

Containerization has become a fundamental technique in modern microservices architectures. It enables applications to be packaged together with their dependencies and executed in isolated environments. This approach maintains process-level isolation in host operating systems, ensuring the applications' independence and portability from the underlying infrastructure. Containerization is the process of packing applications together and running them in an isolated environment, facilitating distribution across different nodes. This approach addresses the common "it works on my machine" problem, simplifying deployment and eliminating the dependency conflicts.

1.4.1 Deploying Microservice Architecture

Deploying microservice-based systems requires careful consideration of service distribution, orchestration, and resource management. The following discussion focuses on the main approaches and technologies used to deploy microservices effectively in modern computing environments.

Difference between Containers and Virtual Machine

At first glance, containers may appear similar to a virtual machine; however, this assumption is misleading. The fundamental difference between virtual machines

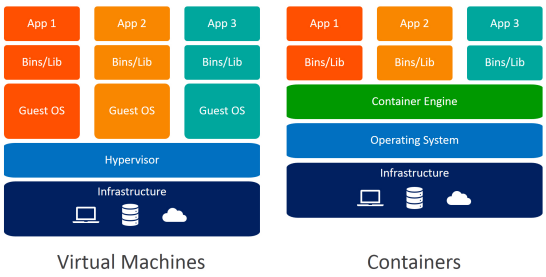


Figure 1.10: Difference between Containers and Virtual Machines [20]

and containers lies in their architectural design. A virtual machine is a technology that enables virtualization of the hardware, allowing multiple operating systems to run on a single machine. This is possible by hypervisor component, which aims for allocating virtual hardware resources such as CPU cores and storage to each VM. In contrast, a container is a virtualization technology which operates at the

operating system level, thus it enables multiple applications to run on the same OS kernel. Other differences are:

- **Resource utilization**: Since VMs have their own OS, they consume more resources, including CPU and RAM [21]. Conversely containers do not require a separate OS for each instance, thus it is less resource consuming.
- **Startup time**: Since the container does not have its own emulating operating system, it achieves higher performance at startup time.
- **Isolation and security**: Generally, VMs provide greater security features than containers due to their isolation level. This is because it runs in an isolated virtualized environment. "A shared kernel introduces a potential security risk: if the kernel is compromised, all containers are vulnerable" [21].
- **Portability**: VMs are difficult to transfer from an operating system to another, because it could have different environmental support. Containers instead are highly portable for its lightweight nature.

Why Container Orchestration is Needed

Container orchestration refers to the automated process of deploying, managing, scaling, and networking containers in production environments. As containerized applications scale across multiple nodes and distributed infrastructures,manual management becomes impractical.

For this reason an orchestration layer is required to ensure reliability and efficiency, by addressing several critical operational challenges:

- **Dynamic Scaling**: Automatically scaling containers up or down in response to real-time workload variations to ensure efficiency.
- **Self-Healing**: Detecting failures and automatically restarting or replacing malfunctioning containers without human intervention.
- **Service Discovery and Load Balancing**: Efficiently routing network traffic to appropriate containers while ensuring high availability.
- **Zero-Downtime Updates**: Managing application updates and rollbacks without interrupting service availability while deploying it.

These reasons make the adoption of this technology mandatory for microservice environments and large-scale applications. The primary advantage of the orchestration environment is Automation, relieving the developer to configure the system with manual actions. However, it could be difficult because of the huge amount of systems built. In this case, the orchestration platform becomes a control plane

that continuously monitors the system and reacts automatically in the presence of issues.

A second benefit of implementing an orchestration system is "High availability" and "failover built-in", which give system uptime guarantees. For instance, if a container fails, the orchestrator automatically detects the crash, restarts the instance, and redirects traffic to healthy containers, ensuring that service interruptions remain transparent to the end user.

Finally, orchestration ensures better "resource utilization". The system distributes workloads efficiently across the available nodes through intelligent scheduling, and the so-called "load balancer". Furthermore, when a node is underutilized, the orchestrator can consolidate workloads and decommission the excess resources to reduce costs due to auto-scaling capability features.

Docker: Basic Concepts

Docker is an OS-level virtualization (or containerization) platform, which allows applications to share the host OS kernel instead of running a separate guest OS like in traditional virtualization [22].

- **Docker image** is an immutable template built from a docker file, a text document containing all the directives that a system must compile in order to assemble the image.
- **Docker container** is an ephemeral system, running on the host machine, that represents an instance of a docker image.
- **Docker registry** is a memory place, often stored in cloud, where it is possible to store images, keeping them versioned.
- **Docker compose** is a tool for running and defining multiple-containers in a docker environment. It is a high level orchestration tool that makes coordination and management of multiple containers simple running them in a single machine, especially in development and testing environments.

Kubernetes: Basic Concepts

Kubernetes, also known as K8s, is an open source system for automating deployment, scaling, and management of containerized applications [23]. The platform Kubernetes does not interact directly with simple containers but operates through a specific abstraction layer designed to manage resources and availability efficiently. At a high level, Kubernetes operates on a **cluster architecture**, a set of machines (physical or virtual) working together. Within this cluster, we can identify two

main roles: the *Control Plane*, which makes decisions as scheduling, and responding to events, and the *Worker Nodes*, which execute the actual workloads.

The core components that constitute the Kubernetes ecosystem are:

- **Pod**: A Pod is the smallest and simplest Kubernetes object. It represents a single instance of a running process in the cluster. Kubernetes runs containers through Pods, which wrap one or more containers (e.g., Docker containers) that share the same storage, network IP, and execution context.
- **Node**: A Node is a worker machine in Kubernetes (either virtual or physical). Each Node is managed by the Control Plane and contains the services necessary to run Pods, such as the container runtime (e.g., Docker) and the Kubelet agent, which ensures the containers are running and healthy.
- **Service**: Since Pods are ephemeral, because they can be created and destroyed dynamically, changing their IP addresses, Kubernetes uses Services to provide a stable networking endpoint. A Service is an abstraction which defines a logical set of Pods and a policy to access them, ensuring that network traffic always finds a running instance.
- **Deployment**: This is a higher-level abstraction that manages the state of Pods. Instead of creating Pods manually, the developer describes a *desired state* in a Deployment (e.g., I want 3 replicas of the nginx application), and the Deployment Controller changes the actual state to the desired state at a controlled rate.

1.5 Data Consistency and Communication between Microservices

Data consistency and communication between microservices are difficult challenges to overcome. However, in the literature there are many solutions to resolve these problems.

1.5.1 Consistency Challenges in Distributed Systems

One of the most significant challenges when dealing with microservices is ensuring data consistency across the entire system. This difficulty stems from the distributed nature of the architecture, which relies on multiple databases running on independent systems, often geographically dispersed.

Unlike monolithic architectures, where the system communicates with a centralized database (simplifying transactional integrity), in a microservices architecture

data is distributed across multiple services. Consequently, ensuring consistency when multiple components need to update related data becomes a complex task that cannot rely on standard database transactions.

To illustrate a typical consistency issue in microservice environments, consider an e-commerce application in which a customer purchases an item. The system must deduct the item from the inventory, process the payment, and confirm the order. If each of these steps is executed by a separate microservice and the payment process fails, the system must ensure that the product is not permanently updated and that the order is not confirmed. This scenario highlights the **consistency problem**: all services must reflect the same final state. In case of failures, the system must avoid inconsistent state and ensure that changes are resolved gracefully.

Data replication across multiple nodes is a common technique in microservices architecture. It is used to improve system performance and reduce latency, allowing users to interact with a replica geographically closer to their location rather than connecting to a distant central server. In the literature, there are various models of data consistency.

- **Strong Consistency**: This model ensures the maximum consistency in all microservices by blocking all data operations until all replicas have the same data. Although strong consistency provides the highest level of reliability, it impacts performance really slowly. So they are therefore used in data-centric environments, where consistency is the main core of applications.

- **Eventual Consistency**: In some applications, strong consistency guarantees are not required, as they are not the core of the infrastructure. Eventual Consistency is suitable for systems characterized by a high volume of read requests and relatively infrequent writes. In this model, updates propagate are slow, and all replicas of a distributed system will eventually converge to the same state if no further updates occur. However, the duration for convergence is not strictly bounded.

- **Continuous Consistency**: Continuous consistency can be measured as the deviation tolerated from strong consistency. This means that consistency is guaranteed within a predefined threshold, beyond which divergence is not permitted.

- **Sequential Consistency**: Introduced by Lamport, this model ensures that all nodes observe operations in the same sequential order, as if there were a global execution time. It is usually used in environments where reordering operations could lead to further consistency problems.

- **Causal Consistency**: Casual consistency is less restrictive than sequential consistency and ensures that the order of operations reflects their causal

relationships, meaning that if one event influences another, all nodes in the system will agree on their ordering.

- **Entry Consistency**: This model ensures that shared data is made consistent only when a synchronization operation occurs, such as acquiring or releasing a lock. Each shared variable (or "entry") is associated with a synchronization object (like a lock or semaphore). It is usually referred to as the critical section of a DS. Consistency properties in entry consistency are expressed in terms of read, write, lock, and unlock operations. Acquiring a lock can only succeed when all writes to the associated shared data have been completed; exclusive access to a lock can only succeed if no other process has exclusive or nonexclusive access to that lock, and nonexclusive access to a lock is allowed only if any previous exclusive access has been completed.

Approaches to Achieve Consistency

To achieve data consistency in distributed and microservices-based systems there are several approaches:

- **Distributed Transactions (2PC)**: Distributed transactions aim to guarantee consistency across multiple services. In particular, it uses the two-phase commit protocol to ensure all services are committed or all are rolled back. While this protocol ensures strong consistency, it introduces high coordination complex and limited distributed systems, making it suitable mainly for data-critical domains such as financial systems.

- **Saga Pattern**: The Saga pattern decomposes transactions into smaller steps. Each service executes its operation and publishes an event. If a failure occurs, compensating actions are triggered to undo previous steps. Sagas can be implemented using choreography, where services communicate through events, or orchestration, where a coordinator service sends commands to saga participants and collects their responses.

- **Event-Driven Consistency**: The event-Driven consistency model is applied in environments where there is a need to update states which are dependent on other states. So a service typically publishes a domain event, usually in a queue, and all other registered services update their state accordingly.

- **CQRS (Command Query Responsibility Segregation)**: CQRS is an architectural pattern which separates responsibilities between command handling and query processing. This separation allows the read and write models to evolve independently and, in some cases, to be persisted in distinct data stores. Consistency between the models is often maintained through asynchronous

event-based mechanisms. Although CQRS offers significant scalability benefits, it typically introduces eventual consistency, which may not be appropriate for applications that require strong consistency guarantees.

- **Idempotency and Retry Mechanisms**: The idempotency and Retry Mechanisms ensure that repeated operations produce the same result. For instance, if a payment request fails and the user makes a second payment, the transaction is processed only once.

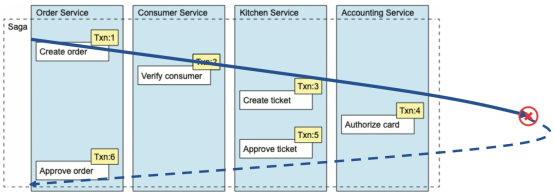


Figure 1.11: Example of saga pattern

1.5.2 Communication Methods: Synchronous and Asynchronous

Unlike monolithic architectures, where a process interacts through direct procedure, communication in microservice applications represents a significant design challenge. Due to the distributed nature of microservices, each they run in their own process and are deployed independently, making service communication really challenging. As a result, the communication must rely on well-defined **communication protocols**. These protocols can be classified into three categories:

- **Synchronous**: Communication based on real-time request and response processes.
- **Asynchronous**: Communication in which services exchange messages without requiring an immediate response.
- **Hybrid**: A combination of synchronous and asynchronous approaches, typically adopted when certain operations require real-time responses while others can be handled in an event-driven manner.

Synchronous

Synchronous communication is primarily based on request-response protocols, most commonly known are HTTP and GRPC. In this model, a client sends a request

to a service and blocks execution until a response is received. If the service does not respond, the request may time out. The HTTP protocol, along with its secure variant HTTPS, is widely adopted in microservices environments. Services expose APIs that allow clients to access and modify resources through different methods such as GET, POST, PUT, PATCH and DELETE. Another form of synchronous communication is the *Remote Procedure Calls* (RPC). As defined in [24], "*A Remote Procedure Call (RPC) is a communication protocol that enables a program to request a service or execute a procedure on a remote server as though it were a local function call*". This paradigm is useful for executing complex calculations or triggering a remote procedure on the server. Although RPC can provide better performance than REST, it is less commonly adopted.

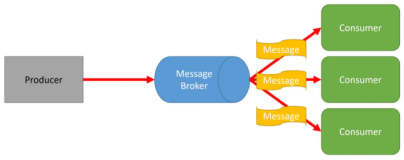
Asynchronous

In Asynchronous communication, a client sends a request without waiting for a response. The key point of this approach is that the client thread is not blocked while waiting for a response.

This approach is typically implemented using the **AMQP (Advanced Message Queuing Protocol)**. This model is especially common in event-driven design patterns, which rely on queues as the primary means of transport and where clients send messages through message broker systems such as Kafka and RabbitMQ. The message producer usually does not wait for a response. These messages are consumed by subscriber systems asynchronously, without waiting an immediate response.

Asynchronous communication can be further categorized into two implementations: one-to-one topology, also called queue, and one-to-many, often referred to as topics. In both cases, the pattern adopted is the publishing and subscribe mechanism, in which infrastructures like event-buses or message brokers are employed; their aim is to publish events between multiple microservices, where communication is achieved by subscribing to these events in an asynchronous way.

(a) Example with exactly one consumer



(b) Example with zero or more consumers

Figure 1.12: Representation of two implementations

Among the most widely adopted tools in microservices are Kafka and RabbitMQ.

Message broker implementations typically involve a broker that manages queues, enabling asynchronous message exchange between services. In this model, an entity known as a publisher emits events into a queue (such as "order-emitter"), while one or more consumers asynchronously process these events.

RabbitMQ is a popular open-source message broker based on the Advanced Message Queuing Protocol (AMQP), which supports additional protocols such as WebSocket and MQTT. In RabbitMQ, there are several exchanges that are supported by:

- **Direct exchange**: it forwards messages to the queues whose binding keys exactly match the routing key of the message.
- **Fanout exchange**: it broadcasts the message to all the matching queues.
- **Topic exchange**: this exchange is connected to a set of queues via binding keys that must be a list of dot-delimited words. Two special wildcards are allowed ('*' and '#'), matching respectively a single word or a sequence of zero or more words.
- **Headers exchange**: its routing decisions are based on message headers rather than routing key.

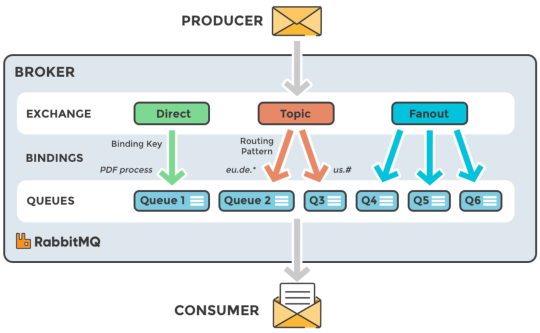


Figure 1.13: Structure of different exchanges in RabbitMQ

Apache Kafka, in contrast, is an event streaming platform designed to operate on a cluster of servers. Its distinguishing features include high scalability, message storage, and fault tolerance. It consists of a message storage, so the arrived messages are stored and indexed by their position within the log. Fault-tolerant is achieved through data replication across multiple brokers, while scalability is supported by the ability to elastically add servers to a Kafka cluster. Topics represent the core of Kafka and are divided into multiple partitions.

Each message is written to a specific partition, enabling parallel consumption and high throughput. Messages are uniquely identified by an *offset* within the

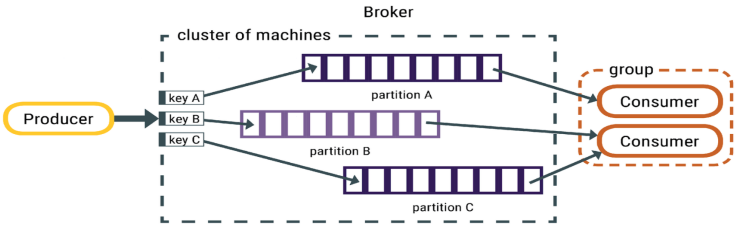


Figure 1.14: Scheme of how kafka works

partition and consist of a key, a value, and a timestamp. Messages sharing the same key are assigned to the same partition, preserving their relative ordering.

Chapter 2

Comparison between Spring Boot and Micronaut frameworks

In modern web development, the use of frameworks is essential for building scalable and maintainable applications. For this reason, the thesis compared two widely adopted frameworks: Spring Boot and Micronaut, highlighting their philosophical and technological differences.

2.1 Technological and Philosophical Comparison

Each framework is examined individually, providing an in-depth comparison of their architectural principles and design philosophies, before presenting a comparative analysis that highlights their respective strengths and limitations.

2.1.1 Motivations For Comparing Spring Boot and Micronaut

Both frameworks are built upon the Java programming language. As widely recognized, Java is an **Object-Oriented Programming (OOP)** language extensively used in web development due to its platform independence, robust ecosystem, and strong community support [25].

The widespread adoption of Java language derives from the necessity to simplify web server development. Prior to the introduction of modern frameworks, Java enterprise application development was managed by technologies which relied on Servlet-based technologies, such as J2EE (Java EE). These technologies required a

highly manual configuration, which make web server project development really complex and error-prone. A notable example is dependency management, which was manually configured by developers; verbose XML files were used to manage dependencies for each component [26]. In a microservices context, building numerous applications from scratch with manual management of all dependencies proved to be particularly challenging. Therefore, the introduction of abstraction layers became essential, enabling developers to streamline application development and reduce the complexity of configuration and deployment.

However, even though Spring Boot is the standard among development teams, they are trying to base their information systems on performance metrics, where factors such as startup time, resource consumption, scalability, and operational efficiency could play a critical role. Thus, many teams are trying to adopt more performative framework such Micronaut and Quarkus, which assert a less resource consumption.

The primary motivation for comparing Spring Boot with Micronaut lies in the shifting technological landscape: the transition from monolithic to distributed systems implies a need for lightweight solutions. Since modern architectures may involve hundreds or thousands of microservices, the use of heavy technologies can lead to significant performance issues.

In cloud-native and serverless environments the lifecycle of an application instance has shifted from static to ephemeral, particularly those using serverless functions or dynamic orchestration platforms such as Kubernetes. Unlike traditional monolithic deployments where uptime is measured in months, microservice instances are frequently created and destroyed due to auto-scaling events, continuous deployment pipelines, and resource optimization strategies. In this context, frameworks with faster startup times and lower memory footprints offer significant operational advantages, mitigating latency during scaling bursts and maximizing infrastructure efficiency.

The most popular framework for enterprise-grade Java microservices is Spring Boot, while Micronaut represents a new generation of frameworks created especially for native execution with GraalVM, quick startup times, and low memory usage. The goal of this study is to specifically assess the performance impact of AOT-oriented approaches (Micronaut with GraalVM) against the conventional, widely used enterprise standard (Spring Boot), which can deliver significant performance advantages over more traditional enterprise solutions. While contemporary JVM frameworks like Quarkus could also have been considered, including additional frameworks would significantly increase the complexity of the experiment without providing proportional benefits, and could introduce factors unrelated to the primary research question.

2.1.2 Use Cases and Philosophical Differences between the Two Frameworks

The two frameworks present difference on their design philosophy and intended use cases. Spring Boot was designed for building web applications and, as reported in the documentation "What is Java Spring Boot?": "Java™ Spring Boot (Spring Boot) is a tool that makes developing web applications and microservices with Java Spring Framework faster and easier" [27]. This framework is the industry standard because of its high availability and ease of use. In fact, it solves the problem of developing server applications by providing ready-to-use production-grade solutions, eliminating boilerplate configuration, and focusing on rapid development [28]. Spring Boot was introduced to replace the traditional Spring Framework, which required heavy XML configuration, making it complex for beginners. Philosophically, it prioritizes developer productivity over resource efficiency, providing features such as [28]:

- **Auto-Configuration**: Autonomous configuration of components based on dependency removes the need for manual XML setup.
- **Easy Maintenance and Creation of REST Endpoints**: The annotations `@RestController`, make the creation of REST endpoints easier.
- **Embedded Tomcat Server**: Spring Boot includes an embedded Tomcat server, eliminating the need for manual setup.
- **Easy Deployment**: The application can be packaged into a single JAR or WAR file that can be deployed easily into a cloud environment.

On the contrary, Micronaut is a JVM-based framework designed specifically for building modular, easy-to-test microservices and serverless applications [29]. The main goal of this framework is to build a web application as light as possible. To achieve it, differently from Spring, Micronaut introduces a new paradigm called **ahead-of-time (AOT) compilation**. The usage of this paradigm, which will be explained technically in the following sections, represents a significant improvement for performance because it allows the framework to precompute dependency injection and other framework-related tasks [29].

Micronaut introduces in its features a new implementation of dependency injection that eliminates the need for reflection at runtime, thereby improving performance [29]. Alongside with GraalVM, Micronaut offers the possibility to build native image support, giving the possibility to build native applications for the host machine and drastically reducing memory usage. Unfortunately, because the program was launched in May 2018, Micronaut does not rely on a huge amount of resources for developing web applications as Spring Boot does.

2.2 Spring Boot Architecture: Runtime Based Approach

Spring Boot is the most used framework for building web server applications using Java programming language. Spring Boot was released in April 2014 to reduce some burdens of developing a Java web application. Indeed, it allowed developers to focus more on the business logic rather than the boilerplate technical code and associated configurations [30]. The Spring Boot framework provides an abstraction layer between the user and the Spring Framework, providing a ready to use application, that the developer can simply start without additional configurations.

2.2.1 The Spring IoC Container

The Spring framework is a modular ecosystem composed of several independent projects, including Spring AOP, Spring ORM, Spring Web Flow, and Spring Web MVC. It gives developers the possibility to use these modules separately for building Java Applications.

The main component of the Spring framework is **Inversion of Control (IoC) container**, which is in charge of building and managing the lifecycle of beans and their dependency. In Spring Boot exist two types of IoC containers, that are **BeanFactory** and **ApplicationContext**. The BeanFactory Container is the most basic container existing, which aims to provide support for dependency injection. The BeanFactory uses Beans[1] and their dependencies' metadata for building and configuring them at run-time. Developers can interact with the BeanFactory to retrieve beans and manage their dependencies programmatically [32].

BeanFactory is represented by the following features:

- **Lazy Initialization**: Beans are created only when it is explicitly requested [33].
- **Lightweight**: Minimal memory footprint is suitable for resource-constrained environments [33].
- **Basic Bean Management**: Supports bean instantiation, configuration, and lifecycle methods such as init and destroy methods [33].

The ApplicationContext is an extension of BeanFactory that provides additional features: event propagation and internationalization support. Furthermore, it offers support for dynamic resource loading, an enhanced observer mechanism that allows

[1] In Spring framework, beans are the objects that form the backbone of your application and that are managed by the Spring IoC container. A bean is an object that is instantiated, assembled, and managed by a Spring IoC container [31].

components to publish and subscribe to application events. It also supports hierarchical structures, enabling child containers to inherit configuration and beans from a parent [34]. Both BeanFactory and ApplicationContext rely on BeanDefinitions to manage beans. A BeanDefinition is a metadata descriptor that contains all the information needed to create and configure a bean instance, such as property values, constructor arguments, and lifecycle methods [35]. When a container instantiates a bean, it reads the corresponding BeanDefinition to determine how to initialize it and inject its dependencies. In Spring, bean configuration is represented by instances of classes that implement the BeanDefinition interface [32].

The BeanDefinitionRegistry is an interface for registries that holds the BeanDefinition, implemented by BeanFactories classes, which internally work with the AbstractBeanDefinition hierarchy to store and manage metadata for all beans [36].

2.2.2 Bean Lifecycle Management

A bean can be identified within BeanFactory with either a unique name or id, and those without them are known as AnonymousBeans.

Each bean follows a well-defined life cycle, which is managed and supervised by the Spring container throughout all stages, from creation to destruction [37]. Bean lifecycle phases are:

- **Container Started**: The Spring IoC is initialized.
- **Bean Instantiated**: In this phase the container create the instance of a bean.
- **Dependencies Injected**: The container injects the dependencies into the bean [37].
- **Custom init() method**: The bean is initialized, if it implements InitializingBean or has a custom initialization method specified via `@PostConstruct` or init-method
- **Bean is Ready**: The bean at this phase is fully initialized and ready to be used.
- **Custom utility method**: In this phase each custom method applied to the bean is executed.
- **Custom destroy() method**: The bean is destroyed only if the bean implements DisposableBean or has a custom destruction method specified via `@PreDestroy` or destroy-method, it is called when the container is shutting down.

It is crucial to note that these steps are executed at runtime, during the application startup.

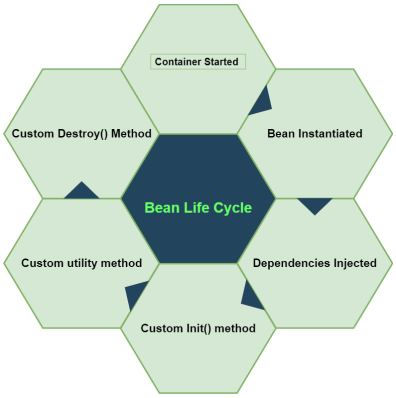


Figure 2.1: API Gateway Architecture [17]

2.2.3 Auto-Configuration and Convention-Over-Configuration

Convention over configuration is a software design paradigm, which aims to decrease the number of decisions that a developer is required to make.

An early example of this principle can be found in the JavaBeans specification, a technology later embraced by frameworks like Spring Boot. Rather than forcing developers to inherit from a complex base class (such as a hypothetical java.beans.Everything), Sun Microsystems designed JavaBeans to provide default behavior for "ordinary objects". For instance, a developer can simply implement a class, and the framework can automatically detect default behaviors such as serialization, UI binding, or dependency injection, without requiring any additional configuration. This allows developers to create lightweight components quickly, overriding specific behaviors only when necessary through specific interfaces.

Spring Boot auto-configuration feature is one of its standout functionalities, allowing developers to build applications with minimal boilerplate code. It is a mechanism that automatically configures Spring application components based on the libraries present on the class path and certain predefined conditions.

This automation is powered by a combination of class path scanning, metadata declarations, and conditional logic. This is due to annotation and runtime checks that determine the most suited configurations to apply.

The auto-configuration process in Spring Boot follows a well-defined lifecycle during application startup, which can be summarized in four key steps:

1. **Enabling Auto-Configuration**: The entry point of this mechanism is the `@EnableAutoConfiguration` annotation, triggered by the class annotated with `@SpringBootApplication`. It is used for starting the loading of auto-configuration classes defined in `META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports`.

2. **Loading Configuration Classes**: Once the auto-configuration is triggered, Spring Boot retrieves a list of potential configuration classes listed in the META-INF folder. Each class is evaluated to determine if it satisfies the conditions specified by its annotations. Configuration classes are loaded in a specific order determined by annotations such as `@AutoConfigureOrder`, `@AutoConfigureBefore`, and `@AutoConfigureAfter`. All of these annotations are used to prioritize the initialization of fundamental components such as: data sources, and web servers, and in order to prevent dependency-related issues.

3. **Applying Conditional Logic**: Spring Boot then evaluates each candidate class to determine if it should be activated. If all the conditions are satisfied, the bean is registered inside the ApplicationContext. For instance, if DataSourceAutoConfiguration is active, Spring Boot creates a DataSource bean using properties defined in "application.properties" or "application.yml". Otherwise, if no database driver is on the classpath, DataSourceAutoConfiguration is skipped entirely.

4. **Resolving Bean Definitions**: This step consist of the resolution of Bean definitions, where Auto-configuration applies conditions to register beans dynamically. For instance, if the user explicitly defines a DataSource bean, Spring Boot detects its presence and skips the default auto-configuration for DataSource. However, overriding default beans may lead to dependency injection issues if not properly aligned with the application's configuration.

To perform what points 2 and 3 described, Spring Boot provides a set of specialized conditional annotations. These allow the framework to decide which specific configuration should be applied depending on the runtime state of the application. The most frequently used annotations are:

- `@ConditionalOnClass`: Check the presence of a specific class on the classpath.
- `@ConditionalOnMissingBean`: Registers only if a similar bean has not already been defined in the ApplicationContext.
- `@ConditionalOnProperty`: Applies a configuration if a specific property is set in the application's configuration files.

These annotations work together in order to help Spring to autoconfigure beans whenever the right conditions are met [38].

2.2.4 Impact on Startup Latency and Memory Footprint

Despite its reputation for convention over configuration, Spring Boot apps often suffer from bloated startup sequences. Most of the impact of the Spring Boot

startup applications is caused by the numerous steps that have been executed [39].

The bottlenecks are caused by:

- **Excessive class path scanning**: If the code has unnecessary packages (e.g., third-party libraries, test packages, etc.) within the scan path, it slows down the context initialization.
- **Too Many Beans or Complex Bean Initialization**: If a project contains hundreds of beans, or complex beans with I/O, reflection, or heavy constructor, the startup time could be compromised, because Spring instantiates all singleton beans at startup.
- **Low Database Connections and Migrations**: ORM and migrations tools, such as Hibernate and Flyway, can considerably delay the application startup. These bottlenecks typically arise from:
  - Slow JDBC connection establishment or timeout issues.
  - Lengthy schema validation or automatic DDL generation.
  - The execution of large or unoptimized Flyway/Liquibase migration scripts, especially when dealing with extensive database schemas.
- **Blocked or Slow External Services During Bean Creation**: Some projects could call external API during the bean initialization, for health checks, config loading, or secret fetching, which slows performances.
- **Too Many Auto-configurations**: Although auto-configuration is a cornerstone of Spring Boot, it can become a disadvantage when "starter" dependencies are included but not fully used. These unused starters may trigger the automatic configuration of beans and services that are not required by the application, leading to unnecessary overhead in both startup time and memory consumption.
- **Blocking Startup Tasks**: Developers often run background tasks (batch jobs, file I/O, cache warming) in CommandLineRunner or ApplicationRunner. These tasks block the main thread unless explicitly made async.

Despite the architectural bottlenecks are caused by the runtime-reflection model, several strategies exist in order to reduce the impact on startup time and memory usage.

Lazy Initialization A possible solution could be the usage of Spring Lazy Initialization. By default, Spring's ApplicationContext instantiates all singleton beans during startup time to immediately detect any configuration anomalies.

However, lazy initialization alters this behavior: instead of creating all beans upfront, the container defers the instantiation and dependency injection of a bean until it is explicitly requested or injected into another active component [40]. Depending on the size of the project, the optimization could speed up the performance and significantly reduce the amount of startup time.

Excluding Unnecessary Auto-configurations The second solution is "Excluding Unnecessary Autoconfigurations", that does not initialize beans that our application doesn't require. This solution could be implemented observing all the beans that are used in startuplogs, and exclude them through `@EnableAutoConfiguration` annotation as it is expressed in listing 2.1.

```
@EnableAutoConfiguration(exclude = {JacksonAutoConfiguration.class,
    JvmMetricsAutoConfiguration.class,
  LogbackMetricsAutoConfiguration.class,
    MetricsAutoConfiguration.class})
```

Listing 2.1: @EnableAutoConfiguration example in Spring

Alternative Servlet Containers The last solution consists of changing the embedded servlet container. By default, Spring Boot uses Apache Tomcat, which is a robust standard but relies on a traditional blocking I/O model. However, more efficient alternatives exist for high-performance scenarios. A notable example is Undertow (the lightweight web server from JBoss/WildFly), which utilizes a non-blocking I/O architecture. Compared to the default configuration, switching to Undertow often results in lower memory consumption and improved response times under heavy load.

JVM and Runtime Tuning Spring Boot is known for its ability to create production-ready applications with extensive features. However, one of the most common complaints of these applications is their memory footprint [41].

While proper memory management is essential to reduce latency, minimize crashes, and improve overall scalability [42], specific tuning of the Java Virtual Machine (JVM) is often required to run Spring Boot efficiently under heavy workloads. Key strategies for improving runtime performance include:

- **JVM Tuning**: It is possible to tune the JVM by assigning an initial (`-Xms`) and a maximum (`-Xmx`) heap size for the application. This defines the boundaries of the memory available for Java objects; however, the **total memory footprint** (Resident Set Size or RSS) will exceed these limits due to non-heap memory usage (Metaspace, Thread Stacks, Code Cache).

- **Embedded Server Threads**: Spring Boot runs an embedded server, such as Tomcat or Jetty. By default, Tomcat has a maximum limit of 200 worker threads, each consuming 1 MB of stack space (in a 64-bit JVM). However, Tomcat creates these thread on demand. In an idle Spring Boot server starts with a minimum of 10 spare threads, but it can scale up to maximum limit under heavy load, potentially consuming up to 200 MB of memory only for thread stacks [43].

- **Garbage Collection (GC) Tuning**: Use the G1 Garbage Collector (G1GC) for most Spring Boot applications, as it provides predictable pause times [42].

- **Reduce Thread Stack Size**: When a thread starts, the JVM allocates a stack memory for it (1MB for each thread). For an application with 1000 threads, this results in 1 GB of committed memory. For small applications, 1 MB per thread is excessive [43]. Fortunately it is possible to reduce the size by the `-Xss` option.

Despite these mitigation strategies, the structural overhead defined by the runtime reflection and dynamic proxying sets a "floor" on resource consumption that is difficult to lower further. While tuning can optimize the application, it cannot eliminate the fundamental costs of the runtime-based architecture. This limitation paves the way for the **Ahead-of-Time (AOT)** approach, introduced by frameworks such as Micronaut.

2.3 Micronaut and the Emergence of a New Paradigm

Micronaut is a "A modern, JVM-based, full-stack framework for building modular, easily testable microservice and serverless applications" [44]. It is used by applications such as Sonar, Mojang and Samsung Smart things. As reported in the official documentation [44]:

"Your application startup time and memory consumption aren't bound to the size of your codebase, resulting in a monumental leap in startup time, blazing fast throughput, and a minimal memory footprint".

This statement highlights the fundamental architectural divergence from traditional frameworks as Spring Boot: the decoupling of performance metrics from code complexity.

2.3.1 Origins of Micronaut: Context and Objectives

The Micronaut framework was developed by the same team responsible for the Grails[2] framework. From the lessons learned while building Spring Boot applications, it aims to handle issues that often make microservice architectures heavy.

The Micronaut framework aims to provide all the necessary tools to build JVM applications, including Dependency Injection and Inversion of Control (IoC), Aspect Oriented Programming (AOP), Sensible Defaults and Auto-Configuration [44]. The combination of these tools, allows Micronaut to facilitate the development of common application typically implemented using Spring Boot, but with a focus to performance features such as: Fast startup time, Reduced memory footprint, Minimal use of reflection, Minimal use of proxies, no runtime bytecode generation and easy Unit Testing.

Moreover, unlike Spring, Micronaut was designed to create applications optimized for serverless and cloud-native environments. It is achieved by using Java's annotation processors [44], usable on any JVM language that supports them, as well as HTTP server and HTTP client.

2.3.2 Compile-time Dependency Injection, AOT, and Reduction of Reflection

Different from Spring, Micronaut uses dependency injection at compile time, using reflection and runtime proxies [45]. This results faster startup times and lower memory usage.

Micronaut AOT is a framework that implements ahead-of-time (AOT) optimizations for Micronaut applications and libraries [46]. This optimization framework focuses on shifting tasks that are normally performed from the runtime to the buildtime. Key optimizations include:

- Pre-parsing configuration files such as YAML and properties.
- Pre-computing bean requirements in order to reduce startup time.
- Performing substitution of classes with optimized versions for a particular environment.

The core of Micronaut AOT, similar to Spring AOT, is too slow the build process in exchange for a faster executable JAR with significantly reduced startup time.

[2]Grails is an open source web application framework that uses the Apache Groovy programming language (based on the Java platform). It is intended to be a high-productivity framework by following the "coding by convention" paradigm, providing a stand-alone development environment and hiding much of the configuration detail from the developer.

This efficiency derives from a deep analysis of the application performed during the build phase.

The Micronaut AOT project consists of 4 main modules [46]:

- **Micronaut-aot-core**: an API provider for implementing "AOT optimizers".
- **Micronaut-aot-api**: expose the public API for interacting with the AOT compiler. It consists of the MicronautAotOptimizer class which is responsible for loading the different AOT modules via service loading, then driving the AOT process.
- **Micronaut-aot-std-optimizers**: implements a number of standard Micronaut AOT optimizers.
- **Micronaut-cli**: the command line tool responsible for calling the AOT compiler. It is recommended to integrate with Micronaut AOT via the CLI tool, so that the process is properly isolated.

Furthermore, Micronaut AOT is a post-processing tool that takes the results of a Micronaut application compiled by developers in order to analyze the application and generate new classes, and resources, used to create binaries. Specifically, the inputs of a Micronaut AOT are:

- The application runtime classpath.
- The Micronaut AOT runtime (including the AOT optimizers).
- The AOT optimizer configuration.

This tool performs a deep analysis of the provided inputs, and generates new classes and resources used to create a new binary, such as jar and native binaries.

The outputs include:

- Generated source files and their compiled versions (e.g., `.class` files);
- Generated resource files;
- Log files, useful for analyzing the steps and any errors during the AOT process;
- A list of resources which should be removed from the final binary (for example, if a YAML file is replaced with Java configuration class, the original YAML file is no longer needed in the final binary).

To avoid reflection problems, Micronaut does not use reflection anywhere in the framework; instead, it uses AOT or Ahead of Time compilation to compute all the annotations and dependencies needed by the code.

2.4 Introduction to GraalVM

GraalVM is an innovative virtual machine tool that enhances the performance and capabilities of Java applications [47]. It was developed as a part of OpenJDK's Graal Project initiated to build a next-generation high-performance polyglot virtual machine [48].

At its core, GraalVM is based on Graal compiler, an alternative "just-in-time compiler" (JIT), which is also written in Java Language. It was specifically designed to compile Java-written applications [49] with the goal of building software that is both extensible and maintainable. This design simplifies the development of complex optimizations while leveraging modern integrated development environments.

2.4.1 GraalVM Architecture

The main goal of GraalVM is to execute different programming language, with high performances. This capability enables interoperability, allowing developers to utilize external libraries across different languages with no overhead. Using different programming languages, helps developers to choose the most appropriate language without losing performance. As stated in Java magazine: "GraalVM is designed with two goals in mind: performance and the ability to support many languages." [50].

GraalVM offers a versatile nature: it can execute programs using the standard JVM in combination with a Just-In-Time (JIT) compiler, but it supports Ahead-of-Time (AOT) compilation in order to create Native Images.

The GraalVM architecture is divided by tiers, which extend support to specific categories of languages. As illustrated in the Figure 2.2:

- The Java HotSpot VM serves as the foundation, utilizing the JVMCI to interface with the compiler.
- The Graal Compiler handles JVM-based languages (Java, Scala, Kotlin).
- The Truffle Framework enables the execution of dynamic languages (Python, Ruby, JS).
- The Sulong component allows the execution of native LLVM-based languages (C++, Rust).

Graal Compiler

The fundamental component of the GraalVM ecosystem is the Graal compiler, a high-performance engine utilized primarily for Just-In-Time (JIT) compilation,

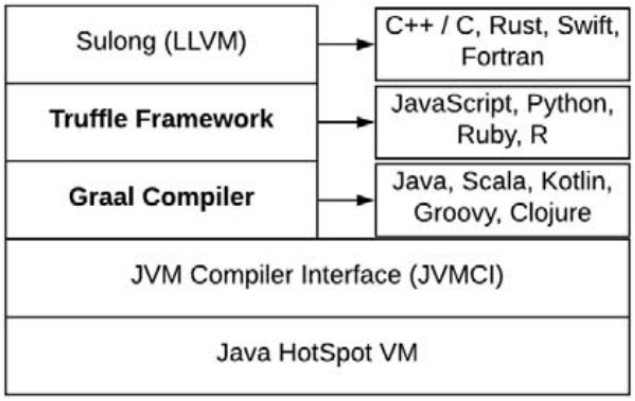


Figure 2.2: GraalVM architecture [51]

but also for Ahead-Of-Time (AOT) compilation. A particular feature of GraalVM is its modular architecture that decouples the Graal compiler from the HotSpot Virtual Machine, allowing Graal to replace the traditional C2 JIT compiler. This separation enables the reuse of existing VM components such as the interpreter, class loading subsystem, garbage collector, and Java Native Interface—without interfering with the compilation process.

The primary objective of Graal VM architecture was to develop a dynamic compiler capable of producing high-quality machine code, simplifying the development of optimization. By lowering the abstraction levels when interacting with low-level languages, Graal achieves these goals without compromising compilation times or memory efficiency [52].

Since the compilation is a complex process, GraalVM organizes its pipeline into two distinct stages. The first stage is *source-specific*, where most of the optimizations are executed. It comprises the Frontend and Graal IR phases, which aim to transform source-specific code into a high-level, platform-independent Graal Intermediate Representation (IR). This transformation is facilitated by tools such as the GraphBuilder or the Truffle framework, followed by the execution of most architectural and platform-independent optimizations.

The source-specific phase is immediately followed by a *target-specific* phase, which consists of Back End[3]. This stage is responsible for translating the High-Level Graal Intermediate Representation (IR) into Low-Level IR (LIR) tailored for the specific hardware architecture [51]. Furthermore, The Backend manages critical low-level tasks such as register allocation and the final generation of machine-specific instructions to be executed by the processor.

[3]It is important to distinguish the compiler back-end from application-level back-end services.

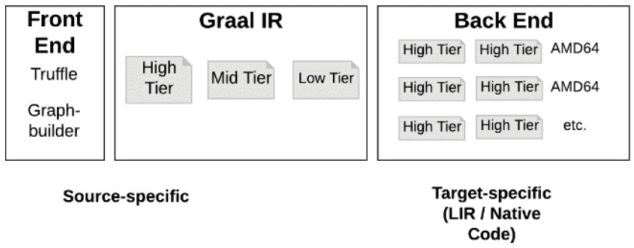


Figure 2.3: Compilation Process within GraalVM [51]

2.4.2 Execution Steps of Graal Compiler

The Graal Intermediate Representation (IR) is a graph-based representation used to model both the control-flow and data-flow dependencies between nodes. In this structure, a node represents a specific program element, such as a constant, an if statement, a loop, a method call, etc. Nodes within the IR could be separated in two groups: fixed and floating nodes. Fixed nodes represent instructions that must follow a strict execution order based on the program's control flow (e.g., branches or loops). In contrast, floating nodes represent values and data-flow operations that do not have a fixed position in the execution sequence, such as arithmetic operations (e.g., addition or multiplication), constant values, and logic comparisons. Unlike fixed nodes, floating nodes do not have a predefined execution position, because they "float" around fixed nodes until the final stages of compilation. [49]

This graph-based design, specifically the separation of data-flow from control-flow, significantly simplifies the application of complex optimizations. For instance, it facilitates Global Value Numbering (GVN), an optimization technique that eliminates redundant computations by identifying and merging nodes that produce the same value [49].

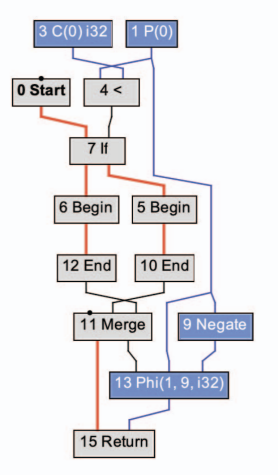
An example of this representation is illustrated in Figure 2.4b which represents how the code is traduced into a Graal IR graph. In this visualization the red links denoted by red edges, connect the fixed nodes, representing the program's strict control flow. Conversely, the data flow is denoted by the blue edges, which connect the floating nodes and illustrate how values move through the application.

2.4.3 Java HotSpot Virtual Machine

The Java HotSpot VM is an implementation of the JVM, initially developed by Sun Microsystems, and later acquired by Oracle. It became the default JVM in Java version 1.3. As already reported in Figure 2.2, the Java Hotspot represents a high-performance VM that connects to the lowest stack of GraalVM architecture [51]. It

```
void abs(int x) {
    int res;
    if (x >= 0) {
        res = x;
    } else {
        res = -x;
    }
    return res;
}
```

(a) Example of Code snipped sample [52]



(b) Representation of Graal IR [52]

Figure 2.4: Example of Graal IR

is optimized to search frequently executed methods, or as it is called "hot spots" in the code in order to apply JIT compiling optimization. The Java HotSpot contains two JIT compilers:

- Server Compiler, called C2
- Client Compiler, called C1

The server compiler (C2) is suited for long-running applications, because it uses highly optimized code and rewrites the components and program dependence graph using a bottom-up tree. While this results in a slower startup time compared to C1, it produces highly optimized code that delivers superior peak performance over time.

The client compiler (C1) is based on control flow graph, and it is suited for desktop applications, where quick startup and responsiveness are prioritized. Indeed, it is known for quick loading, thanks also for the use of interpretation.

JVM Compiler Interface (JVMCI)

The JVM Compiler Interface allows developers to implement a custom version of JIT compiler written in java.

Truffle Framework

A key component within the GraalVM ecosystem is Truffle, an open-source library that enables the efficient execution of multiple programming languages. It simplifies language implementation by automatically deriving high-performance code from interpreters built as self-modifying Abstract Syntax Trees (ASTs). Alongside with the Graal compiler, Truffle represents a significant advancement in language implementation technology, particularly for the modern era of dynamic languages [53].

2.4.4 GraalVM Native Image: the Build Process

The most notable feature of GraalVM is the ability to build Native Images. This ability is particularly disruptive for Micronaut applications, as it allows Java code to transcend the traditional limitations of the Java Virtual Machine (JVM).

As reported in the official documentation of GraalVM:"*GraalVM compiles your Java applications ahead of time into standalone binaries that start instantly, provide peak performance with no warmup, and use fewer resources*" [54].

Static Analysis and the "Closed-World" Assumption

GraalVM Native Image (NI) is a Java bytecode compiler, in which the transformation into a native binary follows a strict "Closed-World Assumption" [55], meaning all code reachable by the application must be processed during the buildtime. This process, which guarantees faster startup time and lower memory footprint, is composed of four main phases [49]:

- **Points-to Analysis:** The builder performs a static analysis starting from the entry point (the *main* method). This process iteratively finds all the reachable parts of the application during the runtime, such as class, methods, fields. After that, any component of the application that is unreachable, is marked as useless, and so it is excluded, which drastically reduces the final memory footprint.

- **Code Initialization:** This is a critical advantage for Micronaut. It starts when there are no more types to be added to type states, and the local fixed point is reached. Developers can choose to initialize specific classes during the build. This means that complex configurations or static data are processed once, and their state is "frozen" into the binary.

- **Heap Snapshotting:** After initialization, the heap snapshotting mechanism phase starts. In this phase the builder captures the state of the objects created in the initialization phase, and stores them in an **Image Heap**. This procedure is particularly useful for building the object graph. The pre-populated heap is

mapped directly into memory at startup, eliminating the "warm-up" period typical of Java applications. After this iterative analysis re-takes the point-to analysis phase, including heap snapshotting, until the system does not find changes compared to the previous iteration.

- **Ahead-of-Time (AOT) Compilation:** The final step is the AOT compilation phase, which is different from the standard Graal compiler workflow. While the Graal compiler features a modular architecture and relies on speculative optimizations that require multiple runtime de-optimizations, AOT compilation performs all optimizations ahead of execution. As a result, parts of the image heap are materialized during the compilation process and embedded directly into the executable. This produces a platform-specific binary with a pre-populated heap, eliminating the need for runtime compilation and reducing startup overhead.

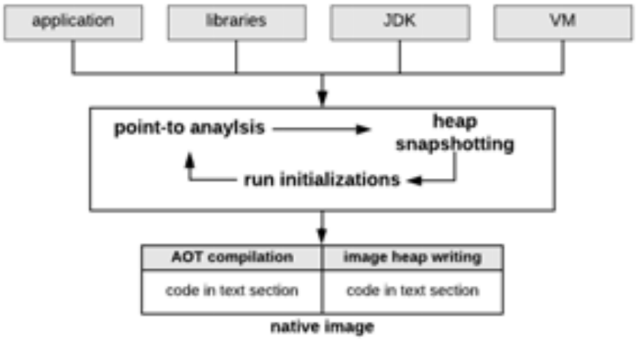


Figure 2.5: Steps of build time process [48]

Comparison between Native Image and CRIU

To understand widespread adoption of GraalVM in cloud environments, it is important to analyze how the system achieves a significant reduction in memory footprint.

GraalVM is particularly effective for managing requests in microservice-based cloud environments, where deployment processes are adopted in order to provide mechanisms to minimize both system startup time and resource consumption. To achieve it, the *Checkpoint/Restore In Userspace* (CRIU) approach can be helpful.

The CRIU was initially developed to provide checkpoint/restore functionality for Linux [56], and later its technology was increasingly used within the JVM ecosystem. It works by "freezing" a running container or application and checkpoint its entire execution state to the physical disk. After that, the saved data can be used any

time to restore the application by resuming execution exactly from the point of the freeze. One of the issues related to the checkpoint/Restore is that it does not reduce the memory footprint due to dynamic class loading and optimization, particularly used in HotSpot VM, in order to compile the code and metadata dynamically [48].

In contrast, GraalVM offers the *Native Image* mechanism. Completely written in Java, Native Image employs Ahead-of-Time (AOT) compilation to produce a standalone executable, using a sequence of complementing mechanisms that are listed in paragraph 2.4.4.

2.5 Theoretical Comparative Analysis

In the literature, there are many comparisons between JVM and GraalVM systems. Most documents include both the "static analysis" and the "dynamic analysis" in order to differentiate steps of the execution.

As [57] state: "software systems can be subjected to static and/or "dynamic analysis". Static analysis is designed to detect bugs, vulnerabilities, and code smells before the software is deployed software into production." This approach examines the source code, deployment, manifest and API documentation; however its disadvantage is the need of language-specific tools, which make difficult to detect errors and bottlenecks on distributed systems. In contrast, the "dynamic analysis" is applied to runtime data collected from systems in production or staging environments, using telemetry data and application logs to monitor the system's behavior. For achieving this, 'dynamic analysis' relies on support tooling such as centralized logging, distributed tracing, and telemetry, used in combination with Jaeger.

The static analysis is mainly performed on the program sources, to inspect it without execution. It analyzes the process through graphs that represent the structure of the code, in order to inspect its quality and design standards. The static analysis could be used for building intermediate representations, such as: Abstract Syntax Trees (AST), Control Flow Graphs (CFG), and Program Dependency Graph (PDG) [49].

Static analysis is not used anymore in microservice environments. As reported in [49]: "Most current static analysis approaches remain distant from microservices, as they consider a single program or codebase." This happens because static analysis analyzes low level programming constructs, such as loops, variables, and local method calls. However, modern microservices rely on higher-level programming practices that emerged with frameworks such as Spring Boot or Micronaut, which use runtime mechanisms such as dependency injection, reflection, and dynamic proxying, which are difficult to track without execution. Consequently, traditional static analysis fails to provide a holistic view of microservice systems because it does

not recognize higher-level abstractions, including distributed components, REST endpoints, and remote procedural calls [49]. To fill this gap, a transition toward "dynamic analysis" and runtime telemetry becomes essential for understanding the system's actual behavior.

2.5.1 Benchmark Methodology

For evaluating JVM, and Graal compiler The Renaissance benchmark and the DaCapo benchmark suites are the most important benchmark suite used

2.5.2 Renaissance Suite Analysis

The first benchmark analyzed is based on the study proposed by [48]. These tests were performed using the Renaissance benchmark suite [58], which is able to test JVM by executing concurrent and parallel workloads. The suite aims to test JIT compilers, garbage collectors, profilers, analyzers and other tools. For the purpose of this analysis, four specific test scenarios were selected:

- *future-genentic*: A workload that executes a genetic algorithm.
- *philosophers*: A variant of "Dining Philosopher" problem used in operating systems for evaluating synchronization and concurrency handling.
- *movie-lens*: A simulation of a movie recommender using Alternating Least Squares (ALS) algorithm for data processing.
- *par-mnemonics*: A workload that solves the phone mnemonics problem leveraging parallel JDK streams.

In this benchmark all tests are warmed-up, which means that a certain number of iterations were performed before the execution of each test. Regarding data persistence, a MySQL database was initialized with tables containing 250 entries to simulate database interactions. The benchmarks test three different JDK configurations to highlight performance differences across compiler versions:

- OpenJDK 64-Bit Server VM (AdoptOpenJDK) (build 25.242-b08) in mixed mode OpenJDK version 1.8.0_242
- OpenJDK 64-Bit Server VM (AdoptOpenJDK) (build 13.0.1+9) in mixed mode with the same javac version
- OpenJDK 64-Bit Server VM GraalVM CE 19.3.1 (build 11.0.6+9-jvmci-19.3-b07) in mixed mode with OpenJDK version 11.0.6

All tests were executed on Mac OS X 10.15.2 x86_64 system, with 2.3 GHz Quad-Core Intel Core i5 and 8 GB 2133 MHz LPDDR3 RAM.

The results illustrated in Figure 2.6, compare the performance of OpenJDK 8, OpenJDK 13, and GraalVM CE 19.3.1. The data demonstrates that in the *movie-lens* scenario, GraalVM significantly performs better than the standard OpenJDK compilers, achieving a reduction in execution time of approximately 31.22% in terms of decreased amount of running time. In other scenarios, the GraalVM compiler performs slightly better than the others.

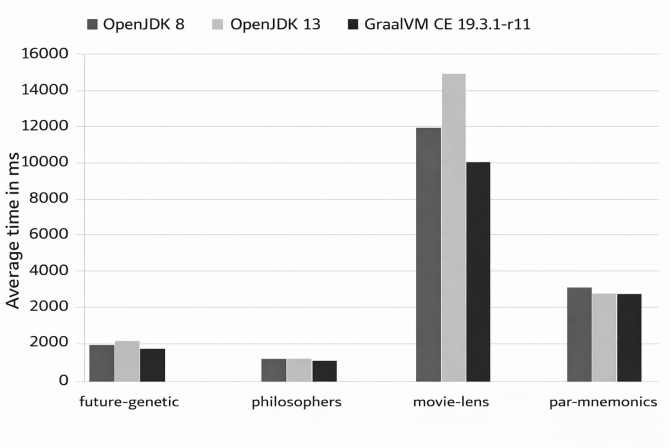


Figure 2.6: Renaissance benchmark results on Graal CE, OpenJDK 8 and OpenJDK 13 [48]

2.5.3 DaCapo Suite Analysis

These benchmarks were performed using DaCapo benchmark suite, a tool built for Java benchmarking and created for performance analysis of JVMs. It included an analysis of JVMs, including compiler and memory management.

The following test cases were selected for this evaluation:

- *h2*: An in-memory benchmark that executes many JDBC transactions, simulating the workload of a banking application.
- *husearch fix*: An application that indexes works of Shakespeare by using Apache Lucene library.
- *xalan*: A benchmark that transforms XML documents into HTML once.
- *jythons*: An application that interprets the PyBench, simulating a Python workload running on the JVM.

In these benchmarks, testing environment used consist on a OpenJDK configuration, with the environment configurations used are [51]:

- OpenJDK 64-Bit Server VM (build 11.02)
- OpenJDK 64-Bit Server VM (build 10.02)
- GraalVM Enterprise Edition (EE) with GraalVM 1.0.0 build 25.192-b12-jvmci-0.53
- GraalVM Community Edition (CE) with GraalVM 1.0.0 build 25.192-b12-jvmci-0.53

The comparative results of these configurations are illustrated in Figure 2.7

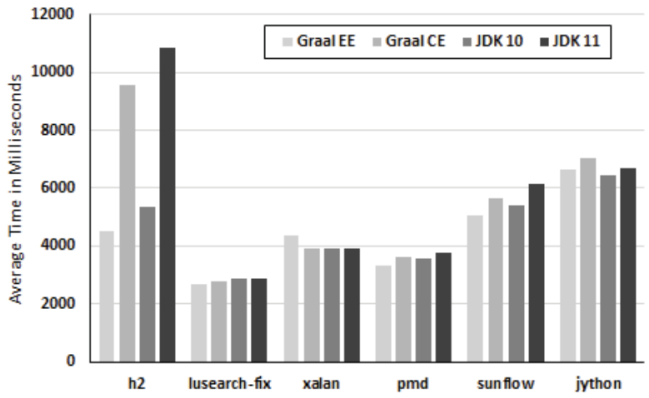


Figure 2.7: Da Capo benchmark results on Graal CE, OpenJDK 10 and OpenJDK 11 [51]

Note that all the reported benchmark reported have been warmed up, before taking the measurement. This initial step is essential because all the system requires time to provide stable result, allowing the JIT compiler to optimize code paths and stabilize CPU caches.

Analysis of Results Each benchmark was performed in 30 iterations, with iterations discarded in order to eliminate startup overhead. As it is showed in Figure 2.7 and as it is also reported in [51], the Graal EE performs better than others in 4 out 6 cases. Specifically, in *xalan* and in *jython* are the only two cases where Graal EE doesn't have the best results. The most notable improvement is observed in the *h2* benchmark, where GraalVM EE achieves a performance increase of approximately 19.8% compared to others.

A closer inspection reveals a correlation with concurrency models. The benchmark test *h2,sunflow*, and *pmd*, that represent multithreaded applications, are

more accurate the others. Considering that, the fastest benchmark results Graal EE, immediately followed by JDK 10 and Graal CE. This suggests that GraalVM is particularly effective at optimizing parallel workloads.

In contrast, in single-threaded applications, such as *jython* benchmark, JDK 10 achieves the best performance, indicating that traditional JVMs may still hold a slight advantage in specific serial execution contexts.

2.5.4 Conclusion of Benchmark

In conclusion, data from both the Renaissance and DaCapo suites demonstrate that the GraalVM framework generally delivers superior performance across a wide range of scenarios. While traditional JDKs remain competitive in single-threaded tasks, GraalVM proves to be the optimal choice for modern, complex, and concurrent workloads.

Chapter 3

Middleware Architecture and Design

After theoretically analyzing the microservice architecture and the comparison between Spring Boot and Micronaut, the following chapter focuses on the practical implementation of the application written using Spring Boot and Micronaut frameworks.

From an overview of the system objective and the rationale behind the proposed solution, the chapter describes the architectural design and the implementation phases using both frameworks. Concluding by describing the deployment strategy on the Oracle Cloud.

3.1 Objectives of the Developed Middleware

The implemented software functions as a middleware, and it was developed in order to facilitate the integration between two heterogeneous company's infrastructures, where each of them exposes distinct and specific APIs. Its peculiarity is to log punctually all the operations that are executed. Through a front-end interface, users can monitor all the system logs, and recover them in case of errors. Consequently, the logging mechanism constitutes the core of the architecture, enabling the immediate identification and resolution of anomalies that are complex and isolated in the communication between systems.

This middleware provides a modern and structured solution for achieving the resolution of possible problems during the integration process between company's enterprise systems, with a focus on:

- **Operation Efficiency**: automatization and centralization of the flux management between systems, in order to reduce errors and intervention times.

- **Scalability and flexibility**: the system is based on microservice architecture, which makes the evolution and the integration of new processes and systems easy.
- **Control and traceability**: the complete monitoring of flux and operation, due to specific components dedicated to orchestration and logging.
- **Logging**: the logs, which are the core of the system, are centralized in a single point. This ensures the facility of consultation, analysis, and make the troubleshooting of anomalies easier.
- **Adaptability to external system**: differently to other solutions already released on the market, this middleware can easily adapt to specific needs of the caller. This ensures to delete implementation costs required from other system for the integration.
- **Transparency**: the middleware aims to be completely transparent, making the integration with the middleware in the communication between the two systems easier.

Batch Aggregation and Optimization The middleware was developed by Technology Reply, the company I worked for, to facilitate the integration between two external enterprise systems: Cyberplan and Microsoft Dynamics 365 Business central.

The specific requirement of this system arose from an integration bottleneck. The source system performs better in sending small, frequent CSV batches, whereas the target system performs significantly better dealing with large, consolidated CSV files.

For this reason the first step of the software is to resolve this discrepancy, by executing the following steps:

1. Start Task: The source system initiates a task with the Middleware, which sequentially signal the target system.
2. Send Batches: The Source system send individual batches to the middleware, which merge them.
3. End the task: The merged CSV is sent to the target system, that will receive a single CSV.

All the operations listed are totally logged by the system. Every network call and data operation is recorded by the system into specific database tables, ensuring that the user can trace and detect problems in case of issues.

3.2 System Architecture

To provide a deep and professional analysis, the software architecture was analyzed using the C4 approach [59]. Known for its professional documentation and visualization of software architectures, the C4 model is a framework that dissects software into four consistent, layered abstractions: **Context**, **Containers**, **Components**, and **Code**, that permits to provide information for different users and purposes [60].

This ensures a clear communication, ambiguity is reduced without compromising consistency or traceability, and promotes more effective documentation that spans business, design, and implementation.

Level 1: System Context The higher level of the C4 hierarchy is the 'System Context' level, visualized through the Context Diagram. This diagram provides a high-level overview of the software system, showing the relationships behind the architectures.

As shown in Figure 3.1, the middleware functions as an integration hub. It receives data from the source system, identified here as *Microsoft Dynamics 365 Business Central*, and communicates directly with the target system, *CyberPlan*.

Figure 3.1: Context diagram of Middleware

Level 2: Container The second level of C4 is called "Container". This level gives a focus on the system, breaking it down into high level technical blocks. As figured in 3.2, it is possible to notice how each element is implemented, and how it interacts with other systems.

As suggested before, the entire system was built on the microservice architecture. As Shown in Figures 3.2, 3.3, the entire system is separated in three main components, that ensures the orchestration, the elaboration of the flux, and the logging of the system.

The Figures 3.2, 3.3 show how the system try to implement the integration between the two companies system, in this case Business Central and Cyberplan, in a microservice environment. All three microservices communicate with a single PostgreSQL database, where all the information between flux, batches and logs are stored inside.

Furthermore, the main components used for this architecture are:

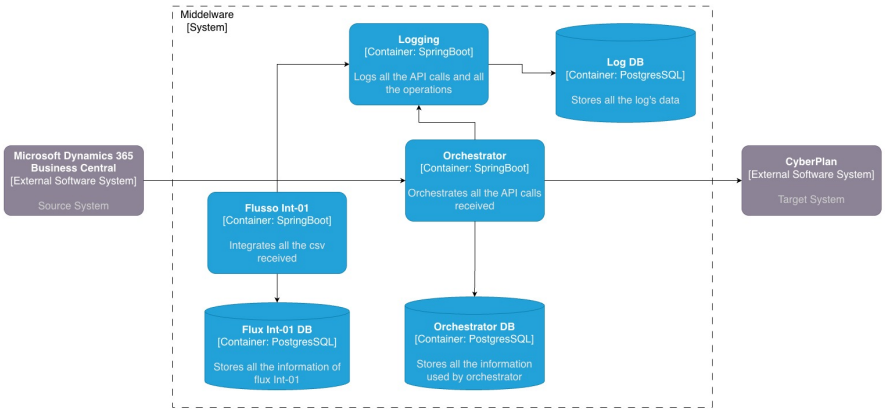


Figure 3.2: Containers diagram of Middleware

1. Orchestrator: The core of the middleware developed for optimizing the management and the coordination between fluxes from and to different company external services. The main features of this component are:
   - **Single point of entry**: it receives all fluxes from different company external systems.
   - **Automatic dispatcher**: it analyzes each flux and redirects it to the most appropriate microservice.
   - **Centralized overview**: it ensures complete governance and traceability of all fluxes.
2. Logging microservice: It is an important component for the application governance. It offers a centralized management of all logs generated from microservices. It ensures a continuous monitoring of the system with a punctual traceability, and the ability to quickly find anomalies inside it. In other words the system permits:
   - **Central log aggregation**: it aggregates and stores the logs from all microservices.
   - **Real-time monitoring**: it permits the view and the immediate analysis of events and system anomalies.
   - **Traceability and auditing**: it ensures the relevant information storing. This data will be used for diagnosis, ensuring the policy of compliance.
   - **Simplified Troubleshooting**: it facilitates the API identification and resolution of potential malfunctions.
   - **Scalability**: it is designed to handle large volumes of data and support the evolution of the application architecture.

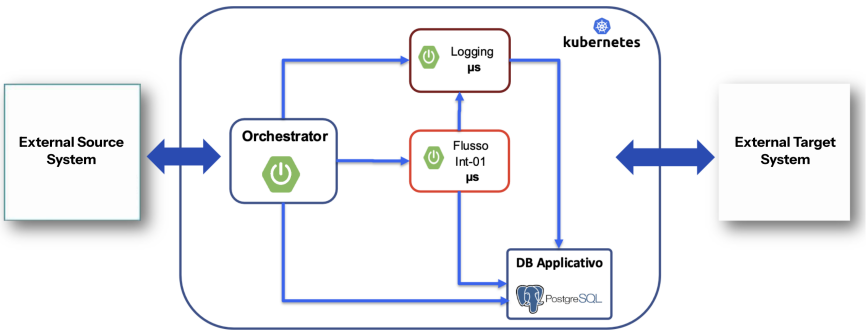


Figure 3.3: Architecture diagram of Microservice System

3. Flux Int-01 (Int-01): it is developed in order to optimize the aggregation and transmission of CSV data from batch API calls, ensuring efficiency and traceability. The system ensures:

   - **Receives API divided in batch**: it manages multiple API calls with CSV binary inside, divided in many batches.
   - **Intelligent Aggregation**: it collects all the calls related to the same entity, joining them in a single CSV, one for each entity.
   - **Transmission to the Target System**: Once the CSV is aggregated, it is sent to the target system.
   - **Full Traceability**: all the API calls and operations are logged to ensure monitoring and precise auditing, leveraging the centralized logging microservice.

Level 3: Component The third level of the C4 model is the "Component", represented by the component diagram that zooms into a single container to show single modules, and how they are related [60]. This diagram is intended for architects and developers, in order to provide a clear map of the internal structure of a container.

The Figure 3.4, 3.5, 3.6 shows the component diagram of each microservice, with their modules and relationships. The first component diagram analyzed is the orchestrator's component diagram 3.4. The diagram exposed a classic MVC architecture that consists on different isolated layers:

- Domain data (Model)
- Presentation (View)
- Control Flow (Controller)

Each microservice follows the same MVC architectures. The adoption of this architecture makes a good separation of roles. The components of a typical Spring Boot web application developed by using the MVC architecture are:

- **Controller**: Controllers in Spring Boot are Java classes responsible for handling incoming HTTP requests and returning an appropriate response. They act as intermediaries between the client (usually a web browser or a mobile app) and the application's business logic.

- **Service**: The service layer typically contains the business logic of the application. It can call other services, or can call many repositories.

- **Repository**: The repository is the layer that interacts with the database. In Spring Data is an interface that represents a collection of operations mapped to a specific domain model or entity.

- **Adapter**: The adapter pattern is a layer responsible to create a bridge between incompatible interfaces. This is used for communication between microservices, or between the middleware and the external system.

In the developed microservice architecture, the Controller depends on the service layer that encapsulates all the core business logic. In this scenario, the service depends also on other auxiliary services to execute specific sub-tasks. This hierarchical design enforces the **separation of concerns** principle, ensuring modularity and maintainability within the application. An example of this dependency structure is illustrated in Figures 3.4 and 3.5.

Regarding exception handling, each API call is managed by a **Controller Advice** component. This module intercepts runtime errors and provides a standardized response body. This ensures a centralized management strategy (Single Point of Responsibility), which significantly simplifies the development and maintenance of the microservice.

Level 4: Code The last level of the C4 model is the Code level, represented by the Code diagram. C4 does not define specific notation at this level, for this reason other modeling tool, such as UML, can be used to represent code structure. The approach used for modelling the code is the Code diagram implemented with UML. As showed in Figures 3.7, 3.8, 3.9, it is possible to notice the classes and the interface of each microservice, with the attributes and methods.

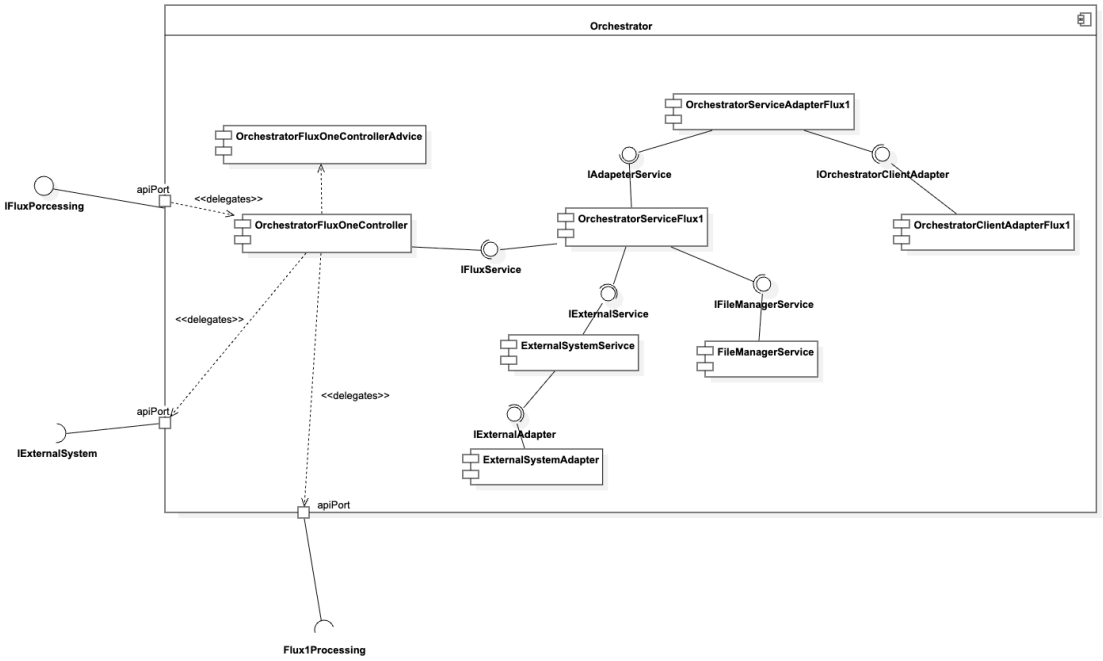


Figure 3.4: Orchestrator's component diagram

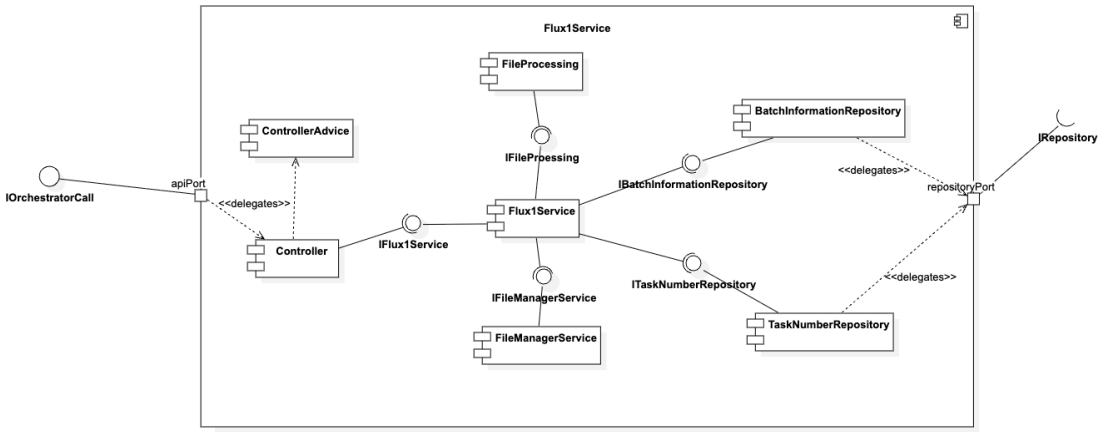


Figure 3.5: Flux1Service's component diagram

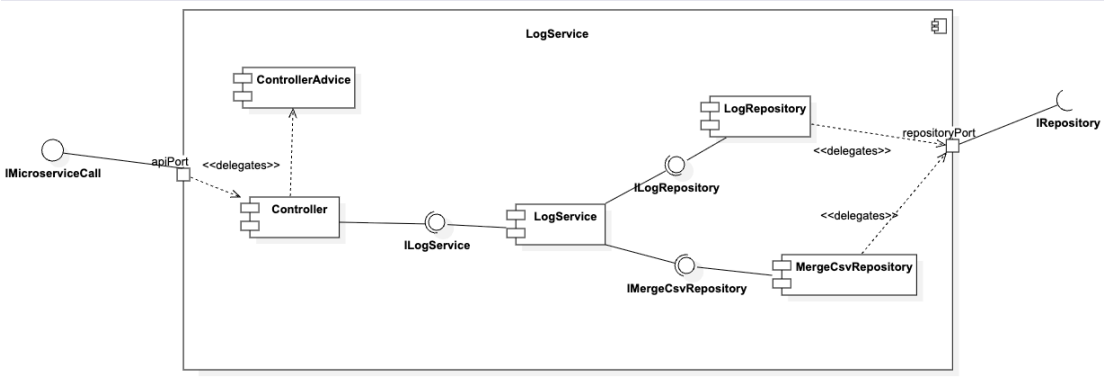


Figure 3.6: LogService's component diagram

3.2.1 Architectural Diagram and Data Flow

For the evaluation purpose, the three microservices must be implemented, and a specific workflow must be executed. The following steps describe the application's workflow, where each number represents a specific step of the application:[1] Remember:

1. The workflow starts when the source system calls the *start task* API call in order to warn the system that batches will arrive.

2. The orchestrator calls the external system APIs in order to inform the target system that a task was started.

3. The orchestrator calls the flux1Serivce, in order to inform that a task number was started.

4. The flux1Service communicates with the database, creating a new task entry, and with the fileManagerService, to create the folders for storing each task. Both orchestrator and flux1Service have a common file system, where all the arrived CSV are stored. Because the purpose is to maintain a clean file management, the file system repository is divided in three folders: inbound, temp and outbound.

5. When a flux is started, the source system sends N batches calling the provided API call.

[1]Remember that the middleware must be transparent, so all the API calls exposed by the orchestrator are the same API calls of the target system.

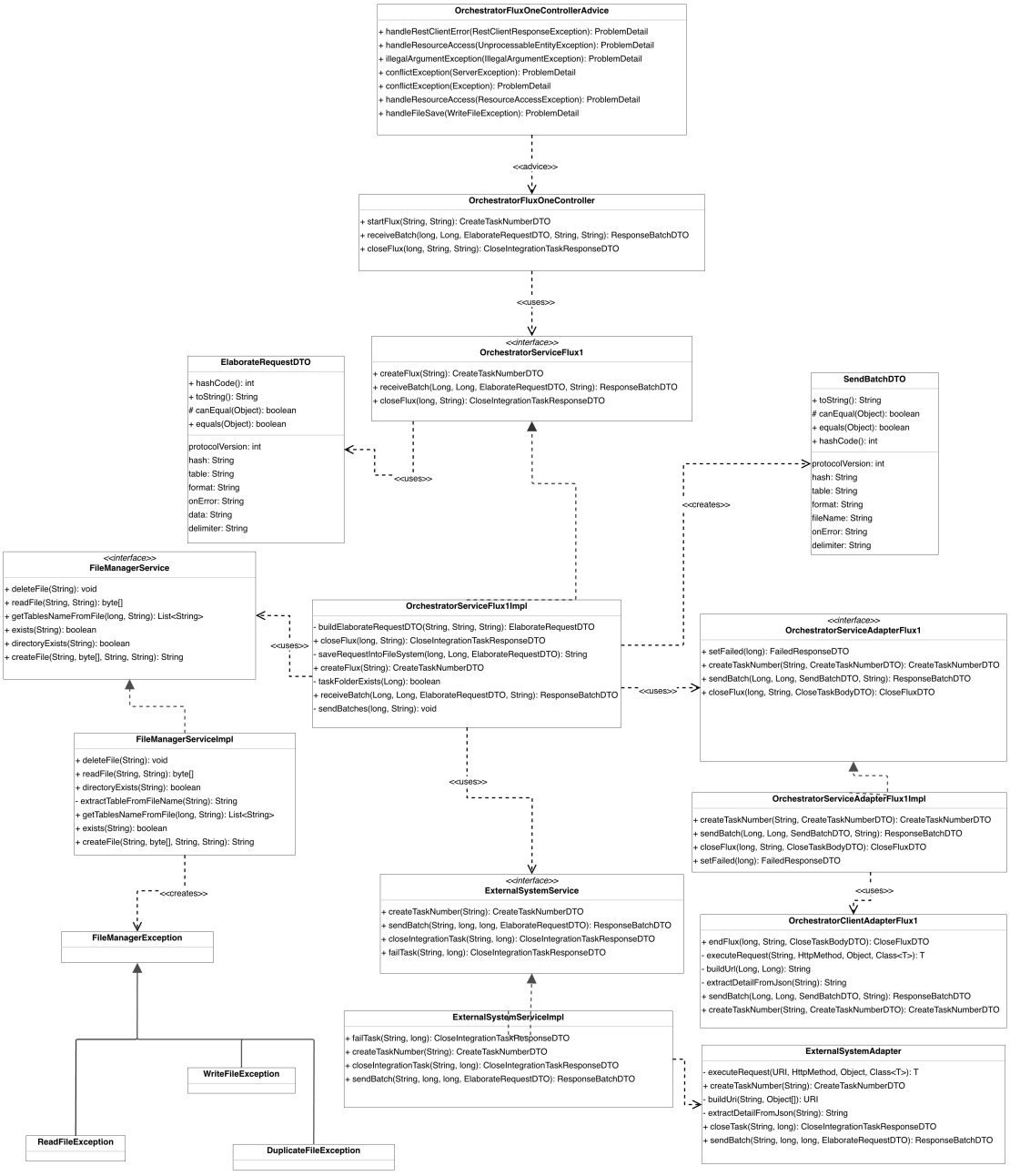


Figure 3.7: Code Diagram of Orchestrator

6. The orchestrator receives the batches containing the CSV encoded in base64 format, then decodes it and saves the CSV in the inbound folder. At the end, it calls the flux1 microservice.

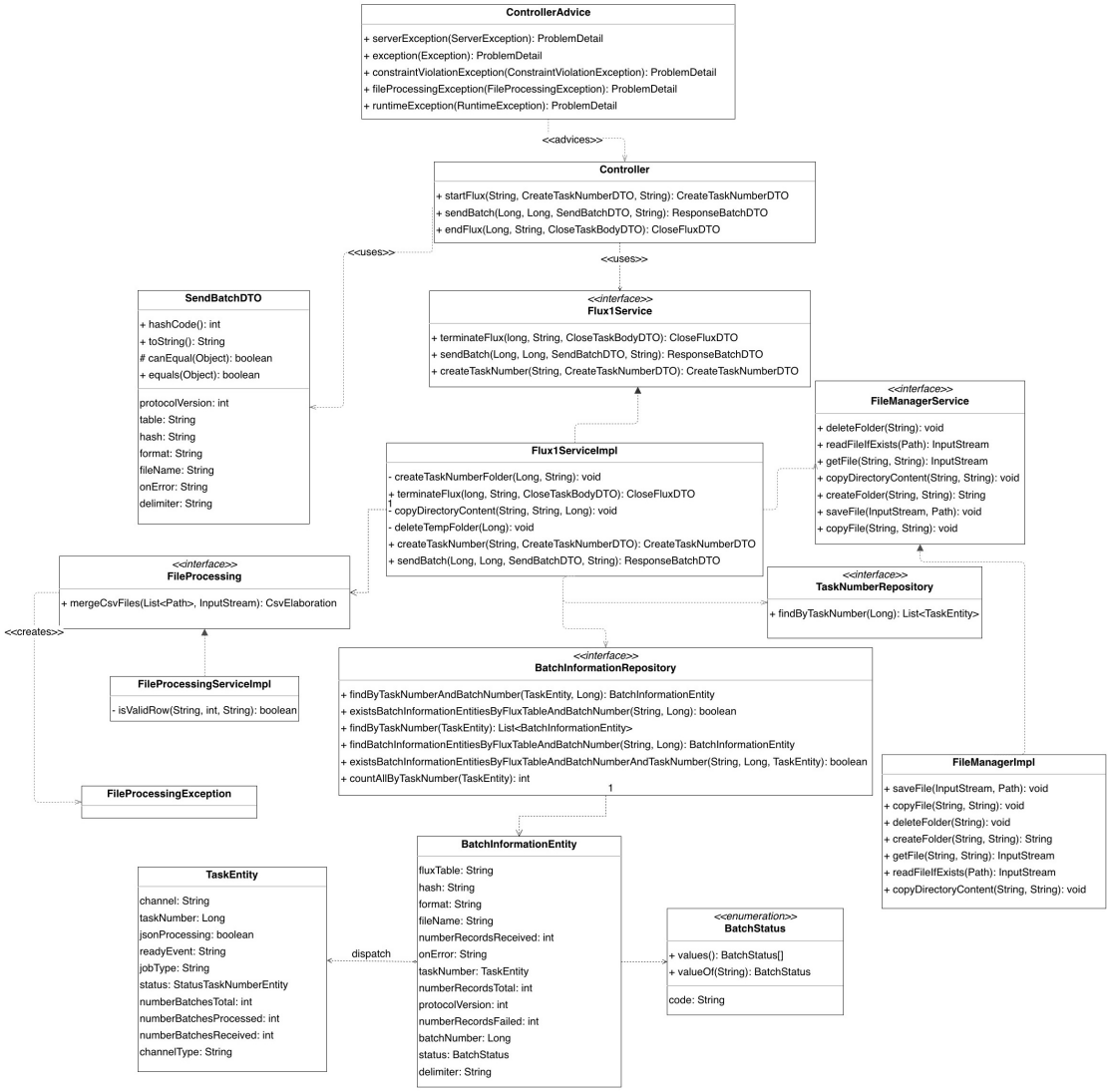


Figure 3.8: Code Diagram of Flux1Service

7. The flux1 microservice moves the file that arrived in the temp folder. If there is another batch with the same flux and the same table name[2], it is merged with the existing file.

8. The flux1 updates all database information about number of batches arrived and number of record processed, arrived or failed.

[2]Each batch has a table name associated, the aggregation of CSV is executed by grouping them by table name

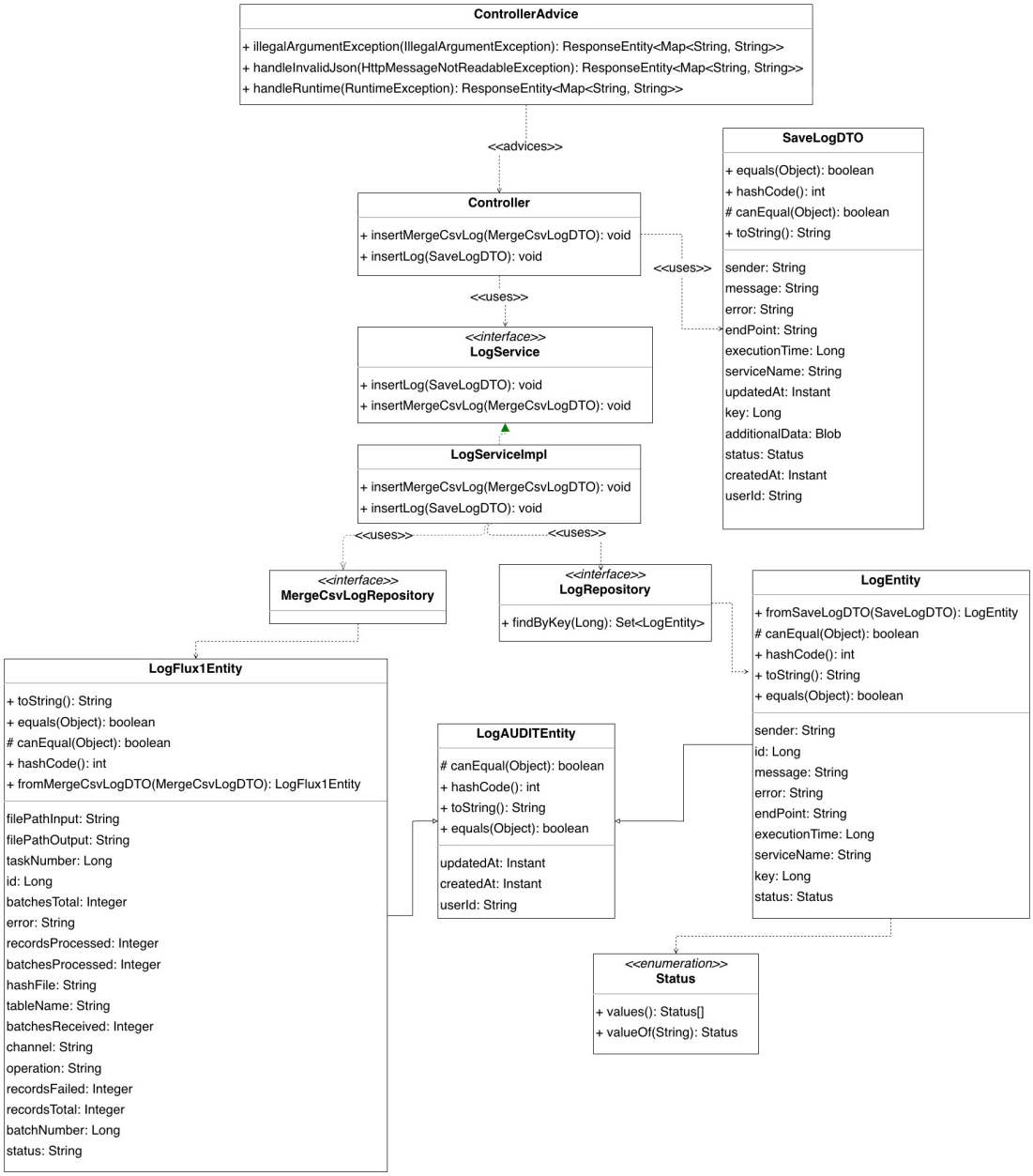


Figure 3.9: Code Diagram of LogService

9. When the 'end task' is called, the orchestrator calls the database in order to update information. Then, it calls the flux1 microservice to warn about the closure of the task.

10. The microservice return all the information about the record and the batches

elaborated.

11. The orchestrator calls the external system in order to send the data received in a grouped and separated CSV.

12. All the executed API calls and operations are logged thanks to the log microservice, by external HTTP calls.

3.2.2 Database Management and Versioning

All the CSV path folder, and all the information about a single flux are stored in a PostgreSQL database, which maintains data in a relation mode. Each microservice relies on different schemas, so that all information are separated. Indeed, each schema is created by different users, one for each microservice, and the user of a single microservice is granted access only by using username and password to the schema that is related to it.

Flux1service Database

The flux one database is composed by the tables:

- BATCH_INFORMATION_ENTITY: Contains all the information about the batches elaborated.
- TASK_ENTITY: Contains all the information about a specific task.

Logservice Database

The flux one database is composed by the following tables:

- GENERAL_LOG: Contains all HTTP call logs of the system.
- LOG_FLUX1: Contains all the log of flux1Service.

Database Versioning

In the development of the microservice, the database schema is not static; rather it evolves as the application code. This continuous evolution is driven by changing requirements and the iterative nature of the development lifecycle. For this reason, in this case, it is important to keep a version of a database to ensure the come back to a previous stable state in case of errors. According to [61], adopting a version control system for the database schema offers several significant advantages for management, such as:

Field Name	Type	Key	Description
id	BIGINT	PK	Unique identifier for the batch
created_at	TIMESTAMP	-	Record creation timestamp
updated_at	TIMESTAMP	-	Last update timestamp
user_id	VARCHAR(255)	-	ID of the user that writes the record
batch_number	BIGINT	-	Sequential number of the batch
format	VARCHAR(255)	-	Format of the batch (e.g., CSV, JSON)
hash	VARCHAR(255)	-	Hash of the batch for integrity check
protocol_version	INTEGER	-	Version of the processing protocol
on_error	VARCHAR(255)	-	Error message
delimiter	VARCHAR(255)	-	Field delimiter used in the batch
task_number	BIGINT	-	Reference to the related task
flux_table	VARCHAR(255)	-	Target table for batch processing
status	VARCHAR(255)	-	Current processing status
file_name	VARCHAR(255)	-	Name of the batch file
number_records_failed	INTEGER	-	Number of records that failed processing
number_records_received	INTEGER	-	Number of records successfully received
number_records_total	INTEGER	-	Total number of records in the batch

Table 3.1: Schema of table `batch_information_entity`

- **Version Control for Database Schema**: The advantage of a schema management is to store the migration scripts in a controller system (e.g. Git). This allows to track all changes and the evolution over time, and to collaborate with other team members.

- **Consistency Across Environments**: The schema management ensures

Field Name	Type	Key	Description
task_number	BIGINT	PK	Unique identifier for the task
created_at	TIMESTAMP	-	Record creation timestamp
updated_at	TIMESTAMP	-	Last update timestamp
user_id	VARCHAR(255)	-	ID of the user that writes the record
status	VARCHAR(255)	-	Current status of the task
channel	VARCHAR(255)	-	Flux channel
job_type	VARCHAR(255)	-	Flux job type
ready_event	VARCHAR(255)	-	Flux property
channel_type	VARCHAR(255)	-	Flux property
json_processing	BOOLEAN	-	Flux property
number_batches_total	INTEGER	-	Total number of expected batches
number_batches_processed	INTEGER	-	Number of batches processed so far
number_batches_received	INTEGER	-	Number of batches received so far

Table 3.2: Schema of table `task_entity`

a consistency across different environments, by applying the same set of migrations. This prevents the risk of issues from different database schema in different environments.

- **Script-Based Migrations**: In order to not be blocked by a specific programming language or framework, it is possible to write a migration using SQL.

In this case a useful tool used for migration management is Flyway.

Flyway is an open source database migration tool that is used to manage, automate, and version database schema changes. It works with most famous databases including MySQL, PostgreSQL, and it is integrated with Java frameworks (e.g., Spring Boot), CI/CD pipelines, and containerized environments.

In order to track the database using Flyway, some configurations are added in the application properties:

```
spring.flyway.enabled=true
```

Field Name	Type	Key	Description
id	BIGINT	PK	Unique log identifier
created_at	TIMESTAMP	-	Event creation timestamp
updated_at	TIMESTAMP	-	Last update timestamp
user_id	VARCHAR(255)	-	ID of the user that writes the record
service_name	VARCHAR(255)	-	Name of the microservice
sender	VARCHAR(255)	-	Caller entity or system
end_point	VARCHAR(255)	-	Invoked API endpoint
status	VARCHAR(255)	-	Call status (e.g., 200 OK, 500 Error)
message	VARCHAR(255)	-	Log message or event description
execution_time	BIGINT	-	Execution duration in milliseconds
error	VARCHAR(255)	-	Stack trace or error code
key	BIGINT	-	Correlation key or business ID

Table 3.3: Schema of table `general_log`

Field Name	Type	Key	Description
id	BIGINT	PK	Unique log identifier
created_at	TIMESTAMP	-	Event creation timestamp
updated_at	TIMESTAMP	-	Last update timestamp
user_id	VARCHAR(255)	-	ID of the user that writes the record
service_name	VARCHAR(255)	-	Name of the microservice
sender	VARCHAR(255)	-	Caller entity or system
end_point	VARCHAR(255)	-	Invoked API endpoint
status	VARCHAR(255)	-	Call status
message	VARCHAR(255)	-	Log message or event description
execution_time	BIGINT	-	Execution duration in milliseconds
error	VARCHAR(255)	-	Stack trace or error code
key	BIGINT	-	Correlation key or business ID

Table 3.4: Schema of table `general_log`

```
spring.flyway.url=\${flux1.flyway.url}
spring.flyway.user=\${flux1.flyway.username}
spring.flyway.password=\${flux1.flyway.password}
spring.flyway.schemas=flux_one
spring.flyway.locations=classpath:db/migration
spring.flyway.baseline-on-migrate=true
spring.flyway.target=latest
spring.flyway.default-schema=flux_one
```

Listing 3.1: Configuration in application.properties

As referred in listing 3.1, for ensuring the validation of schemas, during the application's startup it is provided information such as flywayurl, username and password. If one schema is not registered in the migration file, an error is raised.

3.3 Microservice Implement and Comparison

After modelling the software by using the C4 model, the focus shifts to how the two frameworks are implemented, exploring what are the differences in code implementation.

3.3.1 Project Structure

Each microservice of both frameworks maintains the same structure as they implemented the same standard MVC architecture.

The Figure 3.10 is an example of MVC project structure. The main packages are Entity, Repository, and Controller. Furthermore, each package is tested in the test section, where tests are written in order to guarantee at least the 90% of coverage.

3.3.2 Controller Development and API Analysis

Since both Spring and Micronaut's controller has the same base structure, the listings 3.2, 3.7 shows only their differences. In Micronaut, listing 3.7 is useful for understanding the Micronaut behavior. The difference is the annotation `@ExecuteOn(TaskExecutors.BLOCKING)`, which underline the basic nature of Micronaut.

Indeed, the Micronaut framework is built on Netty, which is designed around an Event loop model and non-blocking I/O. However, it doesn't use a request per thread as Spring Boot, but uses a single thread that manages many requests. This thread could be critical for I/O operations. For this reason, if the architecture is not born to be non-blocking, it is better to insert the Blocking annotation to warn the compiler to move the operation in another thread, specifically used for this type of operation. It is better to use the `@ExecuteOn(TaskExecutors.BLOCKING)` whenever the function makes HTTP requests, database queries, and I/O operation of the file.

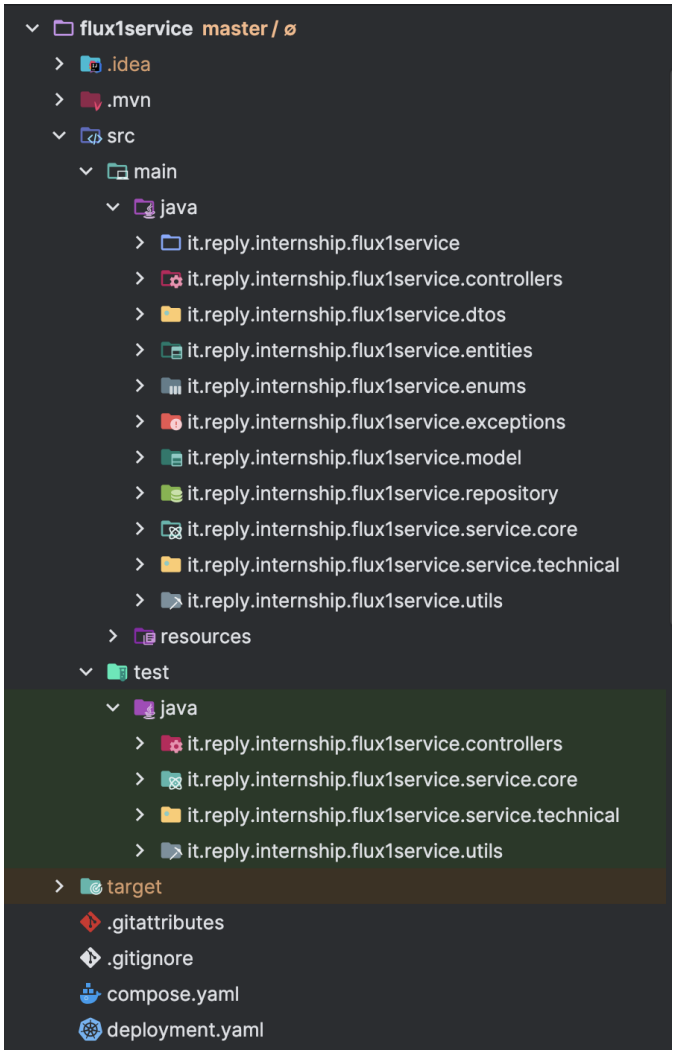


Figure 3.10: Project structure of a microservice

```
@Tag(name = "/", description = "")
@RestController
@RequestMapping("/API/v1/flux1")
public class Controller {

    private final Flux1Service flux1Service;


    public Controller(Flux1Service flux1Service) {
        this.flux1Service = flux1Service;
    }
```

```
    @Operation(
        summary = "start flux request - it creates a flux and return
    the number of the task"
        )
    @APIResponses(value = {
            @APIResponse(responseCode = "201", description = "Created"),
            @APIResponse(
                responseCode = "500",
                description = "Internal Server Error"
                )
    })
    @PostMapping("/{channel}/start")
    public CreateTaskNumberDTO startFlux(
            @PathVariable String channel,
            @RequestBody CreateTaskNumberDTO createTaskNumberDTO,
            @RequestHeader(value = "X-Sender", required = false)
            String sender
    ) {
        CreateTaskNumberDTO response =
        flux1Service.createTaskNumber(channel,createTaskNumberDTO);
        return response;
    }

}
```

Listing 3.2: Spring's controller implementation

```
@Slf4j
@Tag(name = "/", description = "")
@Controller("/API/v1/flux1")
@ExecuteOn(TaskExecutors.BLOCKING)
public class Flux1Controller {

    private final Flux1Service flux1Service;


    public Flux1Controller(Flux1Service flux1Service) {
        this.flux1Service = flux1Service;
    }

    @Operation(summary = "start flux request - it creates a flux and
    return the number of the task")
    @APIResponses(value = {
```

```
            @APIResponse(responseCode = "201", description = "Created"),
            @APIResponse(
                responseCode = "500",
                description = "Internal Server Error"
                )
    })
    @Post("/{channel}/start")
    @LogEvent
    public CreateTaskNumberDTO startFlux(
            @PathVariable String channel,
            @Body @RequestBody CreateTaskNumberDTO createTaskNumberDTO,
            @Header("X-Sender") @Nullable String sender
    ) {
        CreateTaskNumberDTO response =
        flux1Service.createTaskNumber(channel,createTaskNumberDTO);
        return response;
    }
}
```

Listing 3.3: Micronaut's controller implementation

3.3.3 Exception Handling

The exception, handling in both the Spring Boot and Micronaut projects, was managed by an "Advice", ensuring a single point of access to the exception. Indeed, in the project, the custom exception is captured by the advice which provides a custom response body and response code. In Spring Boot the controller advice is implemented by using reflections at runtime. That means, that whenever an exception is thrown, Spring Boot analyzes all the Controller advice, and finds the exception handler that handles it. On the contrary, when Micronaut is annotated by `@Requires(classes = {Throwable.class})`, it generates a class at compiletime (BeanDefinition) to handle that exception. When an exception is thrown, Micronaut calls directly that exception.

```
@Produces
@Singleton
@Requires(classes = {Throwable.class})
public class FluxExceptionHandler implements
    ExceptionHandler<Throwable,
    HttpResponse<FluxExceptionHandler.Message>> {

    private static final Logger logger =
    LoggerFactory.getLogger(FluxExceptionHandler.class);
```

```
    @Override
    public HttpResponse<Message> handle(
        HttpRequest request,
        Throwable exception
        ) {
        if (exception instanceof ServerException) {
                return HttpResponse
                        .status(
                            HttpStatus
                            .valueOf(((ServerException) exception
                            )
                        .getErrorResponse()
                        .getStatusCode()
                        ))
                        .body(new Message(
                                ((ServerException) exception)
                                .getErrorResponse()
                                .getDetails(),
                                "Flux Service Error",
                                ((ServerException) exception)
                                .getErrorResponse()
                                .getStatusCode()
                        ));
        }
        else {
            return HttpResponse
                    .status(HttpStatus.I_AM_A_TEAPOT)
                    .body(
                        new Message(
                            "Unknown Error: " + exception.getMessage(),
                            "Flux Service Unknown Error", 418
                        ));
        }
    }
}
```

Listing 3.4: Micronaut's excepition handling implementation

```
@RestControllerAdvice
public class ControllerAdvice extends ResponseEntityExceptionHandler {
    @ExceptionHandler(ConstraintViolationException.class)
```

```
    public ProblemDetail
    constraintViolationException(ConstraintViolationException exception)
    {
        ProblemDetail problemDetail =
    ProblemDetail.forStatus(HttpStatus.BAD_REQUEST);
        problemDetail.setDetail(exception.getMessage());

    problemDetail.setType(URI.create("/flux/exception/bad-request"));
        problemDetail.setTitle(exception.getMessage());
        return problemDetail;
    }
}
```

Listing 3.5: Spring's controller advice implementation

3.3.4 Service Layer

The core of the software is implemented by the service layer, responsible for calling other service layers in order to perform IO/Task, calling repository, etc. Even if there is little difference of annotation in the service layer, it can significantly change the performances.

```
@Bean
public class Flux1ServiceImpl implements Flux1Service {

    @Value("${fileSystem.baseUrl.inbound}")
    private String inboundFolder;

    @Value("${fileSystem.baseUrl.outbound}")
    private String outboundFolder;

    @Inject
    private LogHelper logHelper;

    @Inject
    private FluxPersistenceService fluxPersistenceService;

    @Value("${fileSystem.baseUrl.temp}")
    private String tempFolder;

    private final TaskNumberRepository taskNumberRepository;
    private final FileManagerService fileManagerService;
    private final FileProcessingServiceImpl fileProcessingServiceimpl;
```

```
 private static final Logger logger =
LoggerFactory.getLogger(Flux1ServiceImpl.class);
 private final BatchInformationRepository batchInformationRepository;

 public Flux1ServiceImpl(
         TaskNumberRepository taskNumberRepository,
         FileManagerService fileManagerService,
FileProcessingServiceImpl fileProcessingServiceimpl,
         BatchInformationRepository batchInformationRepository
 ) {
     this.taskNumberRepository = taskNumberRepository;
     this.fileManagerService = fileManagerService;
     this.fileProcessingServiceimpl = fileProcessingServiceimpl;
     this.batchInformationRepository = batchInformationRepository;
 }


 @Override
 @ExecuteOn(TaskExecutors.BLOCKING)
 public CreateTaskNumberDTO createTaskNumber(
     String channel,
     CreateTaskNumberDTO createTaskNumberDTO
     ) {

     TaskEntity taskEntity = t
     askNumberRepository.save(
             new TaskEntity(
                 createTaskNumberDTO.getJobId(),
                 createTaskNumberDTO.getJobType(),
                 channel
                 ));

     try {

         createTaskNumberFolder(
             taskEntity.getTaskNumber(),
             inboundFolder
             );
         createTaskNumberFolder(
             taskEntity.getTaskNumber(),
             outboundFolder
              );
         createTaskNumberFolder(
             taskEntity.getTaskNumber(),
             tempFolder
```

```
                );

            logHelper.logEventStartTask(
                    taskEntity.getTaskNumber(),
                    "OK",
                    null,
                    channel,
                    MergeCSVOperationLog.FOLDER_CREATION,
                    null
            );

        } catch (FileManagerException e) {
            //save the failed Status
            taskEntity.setStatus(StatusTaskNumberEntity.FAILED);
            taskNumberRepository.save(taskEntity);

            logHelper.logEventStartTask(
                    taskEntity.getTaskNumber(),
                    "KO",
                    null,
                    channel,
                    MergeCSVOperationLog.FOLDER_CREATION,
                    e.getMessage()
            );

            logger.error("Failed to create folders", e);

            throw new FileSystemException(new ErrorResponse(511,
    "Failed to create folders"));
        }

        logger.info("TaskNumber created");
        return new CreateTaskNumberDTO(
                taskEntity.getTaskNumber(),
                taskEntity.getJobType(),
                taskEntity.getStatus().getCode()
        );
    }

}
```

Listing 3.6: Micronaut's service implementation

```
@Service
```

```
public class Flux1ServiceImpl implements Flux1Service {

    @Value("${fileSystem.baseUrl.inbound}")
    private String inboundFolder;

    @Value("${fileSystem.baseUrl.outbound}")
    private String outboundFolder;

    @Value("${fileSystem.baseUrl.temp}")
    private String tempFolder;

    @Value("${logService.service.url}")
    private String baseUrl;


    private final TaskNumberRepository taskNumberRepository;
    private final FileManagerService fileManagerService;
    private final FileProcessingServiceImpl fileProcessingServiceimpl;
    private static final Logger logger =
   LoggerFactory.getLogger(Flux1ServiceImpl.class);
    private final BatchInformationRepository batchInformationRepository;

    public Flux1ServiceImpl(
            TaskNumberRepository taskNumberRepository,
            FileManagerService fileManagerService,
            FileProcessingServiceImpl fileProcessingServiceimpl,
            BatchInformationRepository batchInformationRepository
    ) {
        this.taskNumberRepository = taskNumberRepository;
        this.fileManagerService = fileManagerService;
        this.fileProcessingServiceimpl = fileProcessingServiceimpl;
        this.batchInformationRepository = batchInformationRepository;
    }

    @Override
    public CreateTaskNumberDTO createTaskNumber(String channel,
   CreateTaskNumberDTO createTaskNumberDTO) {

        TaskEntity taskEntity = taskNumberRepository.save(
                new TaskEntity(
                    createTaskNumberDTO.getJobId(),
                    createTaskNumberDTO.getJobType(),
                    channel
                    ));
```

```
try {

    createTaskNumberFolder(
        taskEntity.getTaskNumber(), i
        nboundFolder
        );
    createTaskNumberFolder(
        taskEntity.getTaskNumber(),
         outboundFolder
        );
    createTaskNumberFolder(
        taskEntity.getTaskNumber(),
         tempFolder
        );

    LogHelper.logEventStartTask(
            taskEntity.getTaskNumber(),
            "OK",
            null,
            channel,
            MergeCSVOperationLog.FOLDER_CREATION,
            null,
            baseUrl
    );

} catch (FileManagerException e) {

    //save the failed Status
    taskEntity.setStatus(StatusTaskNumberEntity.FAILED);
    taskNumberRepository.save(taskEntity);

    LogHelper.logEventStartTask(
            taskEntity.getTaskNumber(),
            "KO",
            null,
            channel,
            MergeCSVOperationLog.FOLDER_CREATION,
            e.getMessage(),
            baseUrl
    );

    throw new FileSystemException(
        new ErrorResponse(
            511,
            e.getMessage()
```

```
                    ),e
                    );
        }

        logger.info("TaskNumber created");
        return new CreateTaskNumberDTO(
                taskEntity.getTaskNumber(),
                taskEntity.getJobType(),
                taskEntity.getStatus().getCode()
        );
    }

}
```

Listing 3.7: Spring's service implementation

3.4 Deployment and Infrastructure

The architecture, described in the previous sections and implemented using Spring Boot and Micronaut, was deployed in an oracle cloud environment. This way, it is possible to conduct the load testing without being strictly limited to the machine resources, and being more related to a real use case scenario. After the deployment of each microservice, inside the namespace are installed monitoring tools, grafana e prometheus, in order to monitor and visualize the system for load testing.

3.4.1 Deployment on Oracle Cloud

The deployment of each microservice starts with the creation of the *Dockerfile*, a file containing a sequence of commands executed to create the image. An example is the listing 3.8, where the configuration of the orchestrator is shown[3].

```
# Base Image.
# The application is built to run on Java 21 using the Eclipse Temurin
    JDK.
FROM eclipse-temurin:21-jdk

# Set the working directory inside the container
WORKDIR /app

# Copy the compiled jar into the container
```

[3]The flux1Service and the logService follow the same Dockerfile

```
# The application is compiled as "app.jar" in the "app" folder
COPY target/orchestrator-0.0.1-SNAPSHOT.jar app.jar

# Expose the port 8080 of the container, it will be the default port
    where the http calls arrived.
EXPOSE 8080

#The entry point of the container
# It contains the list of commands that the container must execute when
    it is started
ENTRYPOINT ["java","-jar","/app/app.jar"]
```

Listing 3.8: Dockerfile configuration of orchestrator

After implementing the `Dockerfile`, the container image for each microservice can be built. For this process, it is necessary to use tools such as Podman[4], an open-source alternative to docker. Once the image has been built, it is tagged and pushed to the Oracle Cloud Registry. Subsequently, a Kubernetes manifest, an ".yml" file, is defined to manage the deployment configuration. It is possible to create a file which represents the deployment. Each microservice is equipped with its own *deployment.yml* file, which specifies the container image version and resource requirements.

```
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: sharedflux-pvc
spec:
  accessModes: ["ReadWriteOnce"]
  resources:
    requests:
      storage: 1Gi
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: orchestrator-spring
  labels:
    app: orchestrator
spec:
  replicas: 1
```

[4]https://podman.io/

```
  selector:
    matchLabels:
      app: orchestrator
  template:
    metadata:
      annotations:
        prometheus.io/scrape: "true"
        prometheus.io/path: "/actuator/prometheus"
        prometheus.io/port: "8080"
      labels:
        app: orchestrator
    spec:
      containers:
        - name: orchestrator
          image: fra.ocir.io/fr99gkj3kyca/poc-middleware/v1
          ports:
            - containerPort: 8080
          env:
            - name: SPRING_PROFILES_ACTIVE
              value: "prod"
          volumeMounts:
            - name: sharedflux
              mountPath: /data/sharedflux/flux1
      volumes:
        - name: sharedflux
          persistentVolumeClaim:
            claimName: sharedflux-pvc
---
apiVersion: v1
kind: Service
metadata:
  name: orchestrator
spec:
  type: ClusterIP
  ports:
    - port: 8080
      targetPort: 8080
  selector:
    app: orchestrator
---
```

Listing 3.9: Deployment configuration of orchestrator

As it is shown at Listing 3.9 the *deployment.yml* file is composed of three sections separated by '—'. Each section defines a specific component of Kubernetes

deployment.

- **PersistentVolumeClaim**: The PersistentVolumeClaim (PVC) represents a request of storage. It abstracts the details of the underlying physical infrastructure, allowing the pod to access a specific volume. A claim can request this resource with a specific size and access modes (i.e. they can be mounted ReadWriteOnce, ReadOnlyMany, ReadWriteMany, or ReadWriteOncePod) [62].
- **Deployment**: A Deployment manages a set of replicated Pods. This element check if the real state of pods match with the desired state, described in the deployment section. The Deployment Controller handles the rollout and scaling of the application at a controlled rate. In this case it is defined a single replica set (a single pod), where container is running [63].
- **Service**: A service is an abstraction for exposing a network application that is running as one or more Pods in a cluster. In this case the orchestrator is exposing the port 8080 [64].

3.4.2 Monitoring and Observability Tools

After the deployment of the microservices on the OracleCloud, the development proceeds with the installation of monitoring tools inside the cluster. For this reason, inside the namespace, it has been installed Prometheus and Grafana, using the installation tool helm [65]. After installing the two tools, the cluster looks like in Figure 3.11.

```
giuseppebarone@MacBook-Air-di-Pino ~ % kubectl -n poc-middleware get pods
NAME                                                READY   STATUS    RESTARTS   AGE
flux1service-micronaut-68d857577f-7zqwt             1/1     Running   0          6h12m
grafana-7479d6dff6-hgdf9                            1/1     Running   0          5d3h
logservice-micronaut-74f744c94c-b4l7j               1/1     Running   0          36h
orchestrator-micronaut-8585878cb7-7zz7t             1/1     Running   0          6h10m
output-mockup-service-7648594687-nkccv              1/1     Running   0          179m
postgres-deployment-66bbdfdfd9-8jzvx                1/1     Running   0          5d3h
prometheus-alertmanager-0                           1/1     Running   0          5d3h
prometheus-kube-state-metrics-8459ccf44c-jw69j      1/1     Running   0          5d3h
prometheus-prometheus-node-exporter-kcghg           1/1     Running   0          5d3h
prometheus-prometheus-pushgateway-68757884b8-qv668  1/1     Running   0          5d3h
prometheus-server-f7bd75bb6-9x4n8                   2/2     Running   0          5d3h
giuseppebarone@MacBook-Air-di-Pino ~ %
```

Figure 3.11: namespace with all microservice and monitoring tools

Once prometheus and grafana are configured, prometheus will request all the metrics each 15s, and grafana shows the data collected by prometheus in the dashboard. For evaluation, the software is monitored by two dashboards.

Chapter 4

Metrics and Experimental Methodology

This chapter analyzes and discusses the theoretical rationale behind the choice of using the metrics to evaluate the middleware. Subsequently, it introduces the tools utilized to perform the load testing, as well as the set of methodologies for displaying and extracting the metrics from the test.

4.1 Definition of Metrics

After introducing the implementation of the middleware using the two frameworks and its subsequent deployment on a cloud infrastructure, the evaluation phase represents a key milestone of the project. The evaluation starts by choosing the metrics used to analyze the system. The primary objective of this comparison is to evaluate the software based on performance metrics, with particular attention to resource consumption from the perspective of a system architect. To provide a structured and professional analysis of the choice of this metrics, the approach most suitable is the Goal Question Metric paradigm (GQM) [66], which follows a top-down methodology, rather than a bottom-up one, to provide an incremental identification of observable characteristics.

The GQM is based on the principle that before evaluating software, an organization must specify explicit goals for itself and for the project. Subsequently, the outcomes are used to establish a measurement system, which consists of a set of tailored rules to follow in order to evaluate the software effectively.

The Goal Question Metric is based on three levels [67]:

- **Conceptual level (GOAL)**: This level represents a goal or an objective to be achieved. Examples of objects of measurement include:

  - Products such as Software Requirement Specifications (SRS), Designs, Program or code;
  - Processes such as Testing (Verification and Validation), Designing;
  - Resources such as Hardware and Software.

- **Operational level (QUESTION)**: The level which represents the questions, used to assess the goal.
- **Quantitative level (METRIC)**: The level that can represents metrics, which can be divided in two types:
  - Objective such as LOC (Lines of code), size of module, size of program, etc.;
  - Subjective such as Level of user satisfaction on a scale of 1 to 10.

All the levels listed above compose a hierarchical structure starting with the definition of the goal, the object to be measured, and the view point. The goal is then decomposed into several questions, breaking down the issue into major components, and each question is refined into specific, measurable metrics.

For the purpose of the evaluation, the Figure 4.1 illustrates the GQM approach applied to the developed software.

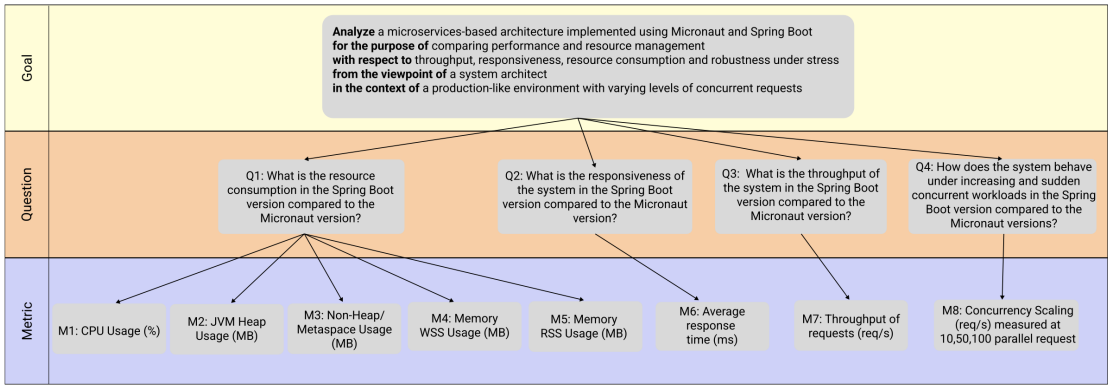


Figure 4.1: Goal Question Metric approach applied to the evaluation of the middleware.

As Figure 4.1 shows, the first step of GQM methodology is to define the goal of the evaluation. In this thesis, the goal is to analyze a microservice-based architecture implemented with Micronaut and Spring Boot frameworks, specifically to evaluate and compare their performance. The system is analyzed from the perspective of a software architect, which focuses on identifying the most efficient framework in terms of resource optimization, scalability, and operational stability. The context

of this evaluation is a production-like environment, deploying the tested software within a cloud infrastructure, simulating a realistic workload of concurrent requests.

4.1.1 Analysis of the Metrics Used

All the metrics used are derived from the questions presented in Figure 4.1, which reflect how the system would be evaluated. In this scenario the system is proposed to be evaluated under:

- responsiveness;
- resource consumption;
- throughput;
- responsiveness to an increasing workload;

The first question addressed is: "*What is the resource consumption in the Spring Boot version compared to the Micronaut version?*", which core question for the evaluation. It evaluates the resource consumption of both the system and the cluster. In this case the metrics of evaluation are:

- **CPU Usage (M1)**: It indicates the percentage usage of a system's CPU during the tests. This indicates how the system is stressed when using one framework than the other.
- **JVM Heap Usage (M2)**: It indicates the amount of memory currently allocated and used by the Java Virtual Machine (JVM) within the heap space for dynamic object storage.
- **Non-Heap/Metaspace Usage (M3)**: It represents the memory used by the JVM for non-heap structures.
- **Memory WSS Usage (M4)**: It refers to the amount of memory actively used by a container. It includes the pages of memory that are currently in use and frequently accessed by the application.
- **Memory RSS Usage (M5)**: It measures how much memory a process is consuming in our physical RAM, to load all of its pages after its execution [68].

The second question proposed is: "*What is the responsiveness of the system in the Spring Boot version compared to the Micronaut version?*". It analyzes the responsiveness of the system, which means how much time the system reacts to an external request. The metric of evaluation that respond to this question is

the **Average Response Time (M6)**. This metric is measured in ms, and it is calculated by a mean between all the response time of the requests.

The third question, "*What is the throughput of the system in the Spring Boot version compared to the Micronaut version?*", is represented by the metric **Throughput (M7)**, which is how many requests a system is able to handle in a specific time period.

The last question,"*How does the system behave under increasing and sudden concurrent workloads in the Spring Boot version compared to the Micronaut versions?*",represents how the system behaves after an increasing and sudden workload, represented by **Concurrency Scaling (M8)** metric.

4.2 Testing and Data Collection Tools

The testing methodology involves the configuration of specific test scripts to ensure repeatable and reliable execution. This is followed by the continuous collection and visualization of the metrics using prometheus and grafana tools.

4.2.1 Apache JMeter: Configuration and Stress Test Scenarios

The test scripts are implemented using Apache JMeter [69]. JMeter is an open-source software written in Java, which is designed to perform load testing, and performance measurement creating operations, if conditions, and thread group via its Graphical User Interface (GUI). The main purpose of the software is the guarantee of test automation, solving the common problems with manual testing such as: consistency and reputability of the tests, that can produce false results. Furthermore, manual testing can be exhausting and time-consuming, potentially causing delays and conflicts during the testing phase [70].

In order to understand all the nomenclature of the tests, the tools used for implementing the automation tests are listed below:

- **TestPlan**: A test plan details a sequence of steps that JMeter will execute during his running. A tests plan consists of one or more Thread Groups, logic controllers, sample generating controllers, listeners, timers, assertions, and configuration elements [71].

- **ThreadGroup**: Thread group elements are the starting point of any test plan. All controllers and samplers must be under a thread group. Each thread will execute the test plan in its entirety and completely independently by other test threads [72].

- **Sampler**: Samplers tell JMeter to send requests to a server and wait for a response. They are processed in order of appearing in the tree [72].

- **Listener**: Listeners provide access to the information that JMeter gathers about the test cases while JMeter runs. Especially in this test situation a "View Results Tree" was used. The "View Results Tree" Listener shows details of sampler requests and responses, and can display basic HTML and XML representations of the response [72].

The advantage of JMeter is its highly customizable interface, which makes the execution of a wide variety of tests possible. In our test case, it is useful for guaranteeing the complete sequence of test calls, which consists of: StartTask, SendBatch, and EndTask.

A proper description of the options used for implementing JMeter is useful for replicating the tests. The following configuration is used to perform the tests explained below:

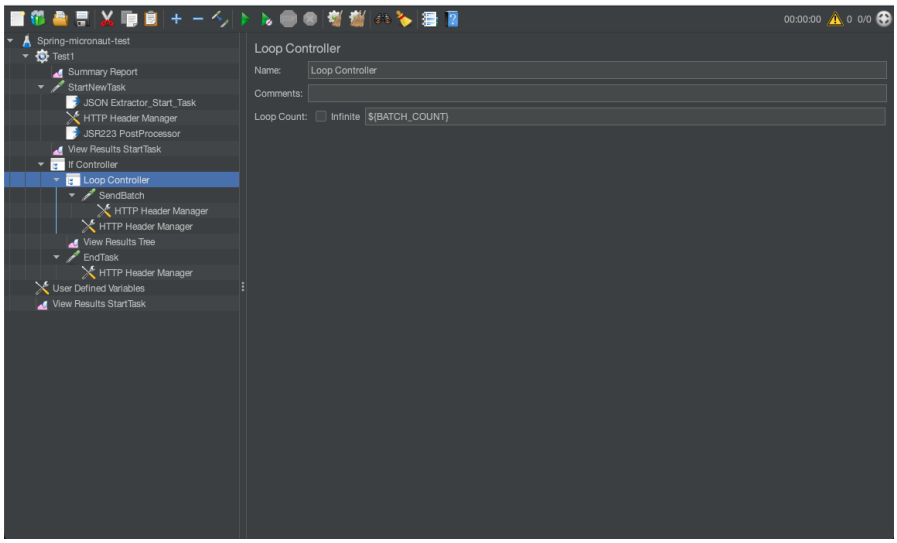


Figure 4.2: Test structure of JMeter

As in Figure 4.2, the structure of the tests are implemented as follow:

ThreadGroup

A thread group can handle more than one user in parallel, and it is used for making parallel requests.

Figure 4.3 is an example of how to configure a thread group. Instantiating a thread group is important to provide information such as:

Figure 4.3: Thread group settings

- **Number of threads**: how many parallel requests can be executed;
- **Rump-up period**: how many seconds need to start all threads gradually. If 10 threads are used, and the ramp-up period is 100 seconds, then JMeter will take 100 seconds to get all 10 threads up and running [73];
- **LoopCount**: how many iterations can be executed;
- **Delay Thread creation until needed**: It is important in order to not end the thread during the execution of the tasks. If it is not set, JMeter creates all the threads immediately, even if it is not used;
- **Duration**: how long the Thread Group runs.

Sampler

Each thread group can contain one or more samplers. An example, used in the tests, is the HTTP Request Sampler that could be customized as shown in Figure 4.4.

The tester in this window could choose how to perform

the test, choose the type of the HTTP request, the url, the port, and other relevant informations that could be useful in order to not make errors caused by the timeout of JMeter, such as:

- **UseKeepAlive**: which enables persistent connections, and allows multiple requests to reuse the same TCP connection.
- **Connect timeout**: which specifies the maximum time JMeter waits to establish a connection with the target server.

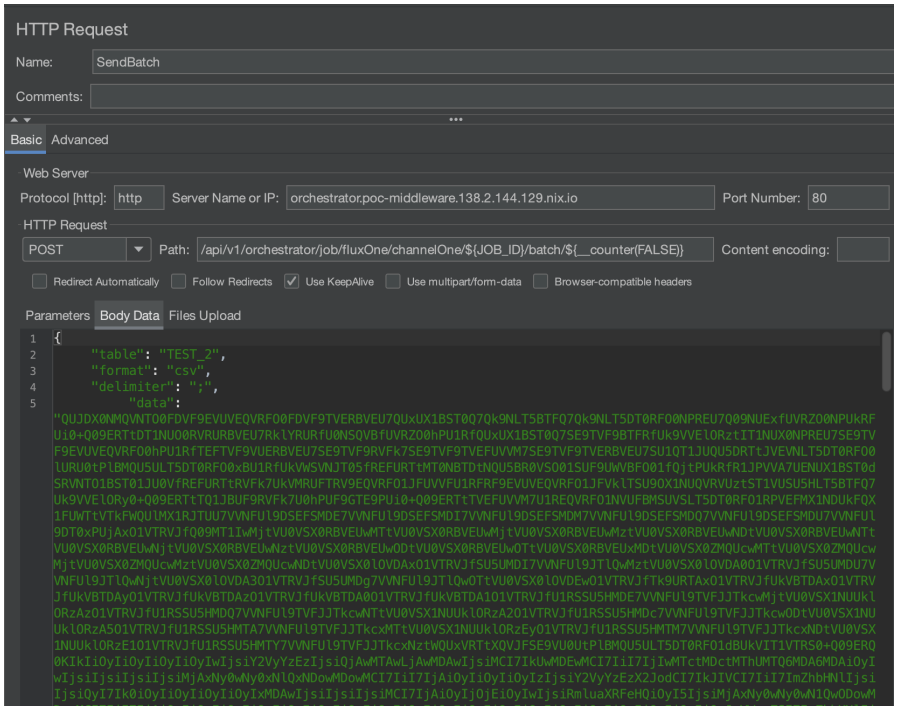


Figure 4.4: Http request settings

- **Response timeout**: which defines the maximum time JMeter waits to receive a response after the request has been sent.

Listener

The two types of listener implemented are: View Results Tree and Summary Report. The first one is used to give a deep analysis to what requests are performed during the load testing, inspecting each request performed. The second listener, Summary Report, is used to give a summary to all the request performed, showing statistics such as the number of samples, success and error percentages, response time, and throughput.

4.2.2 Prometheus and Grafana for Real-Time Monitoring

The tests executed are monitored constantly by prometheus and grafana tools. Prometheus is used in the test environments in order to collect metric information, and grafana for displaying them inside dashboards. The first step to execute is to define the queries that fit all the metrics that we want to collect. As said in [74], "Prometheus provides a functional query language called PromQL (Prometheus Query Language) that lets the user select and aggregate time series data in real

time". It can be configured by adding a range query with a start and an end time, a single timestamp, or a step value to specify the number of queries to be executed within the time range. The query used are listed below:

```
# 1. CPU Usage
max(irate(container_cpu_usage_seconds_total{container!="",
    container!="POD", namespace="<ns>", pod=~"<framework>",
    image!=""}[40s])) by (pod)

# 2. WSS Memory
max(container_memory_working_set_bytes{container!="", container!="POD",
    namespace="<ns>", pod=~"<framework>", image!=""}) by (pod)

# 3. RSS Memory
max(container_memory_rss{container!="", container!="POD",
    namespace="<ns>", pod=~"<framework>", image!=""}) by (pod)

# 4. Avg Response Time
(sum(rate(http_server_requests_seconds_sum{namespace="<ns>",
    pod=~"<framework>"}[10s])) by (pod) /
    sum(rate(http_server_requests_seconds_count{namespace="<ns>",
    pod=~"<framework>"}[10s])) by (pod)) * 1000

# 5. Heap Memory
max(jvm_memory_used_bytes{area="heap", namespace="<ns>",
    pod=~"<framework>"}) by (pod)

# 6. Non-Heap Memory
max(jvm_memory_used_bytes{area="nonheap", namespace="<ns>",
    pod=~"<framework>"}) by (pod)

# 7. Throughput
sum(rate(http_server_requests_seconds_count{namespace="<ns>",
    pod=~"<framework>"}[10s])) by (pod)
```

Listing 4.1: PromQL queries utilized for metrics extraction during the load testing phase.

In listing 4.1, the queries are parametrized respectively by framework and by namespace. The set of queries examines both external and internal performance metrics, specifically exploring container-level metrics, such as Working Set Size (WSS) and Resident Set Size (RSS) memory, alongside JVM memory usage. To capture these dynamics effectively, the queries analyze a 10-second time window and group the results by pod (unlike CPU usage analysis, which requires a larger sampling window).

The most used tool for monitoring systems is Grafana, an open source software that enables you to query, visualize, alert on, and explore metrics, logs, and traces wherever they are stored [75]. Grafana provides sets of dashboards used for displaying metrics. In this case two dashboards are implemented, one for kubernetes and the other for node exporter.

The dashboard, named “K8S dashboard EN", is used for monitoring kubernetes clusters, and starts from the template [76], that are showed in Figures 4.5, 4.6:

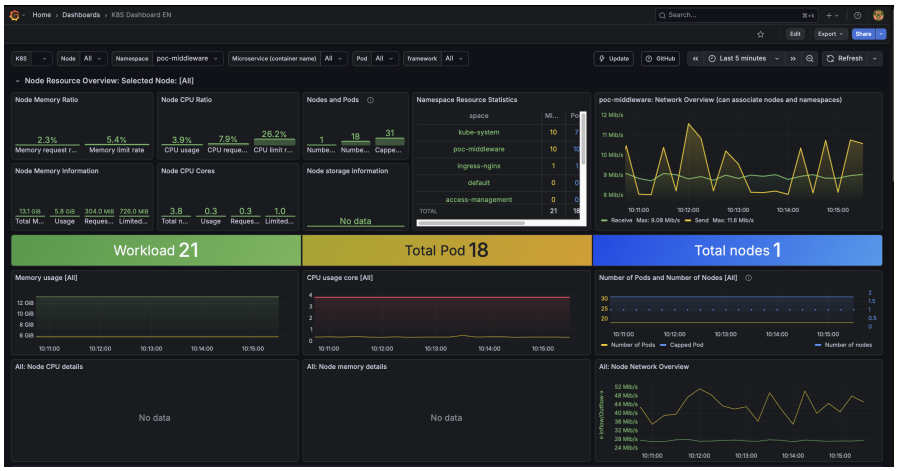


Figure 4.5: K8S dashboard EN

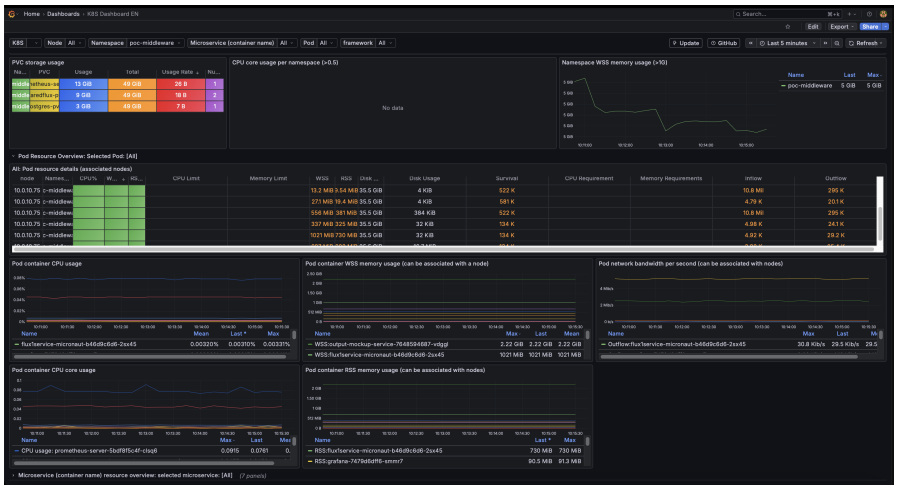


Figure 4.6: K8S dashboard EN

Another dashboard used for displaying further information is the node exporter dashboard, mainly used for a higher level of information. Therefore, it is used for

monitoring all the machine resources. As reported in the guide [77], the Prometheus Node Exporter exposes a wide variety of hardware and kernel-related metrics.

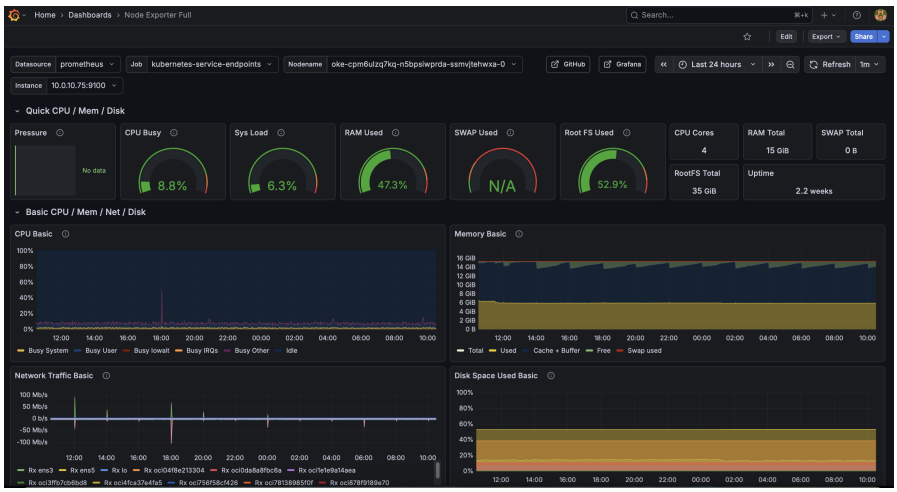


Figure 4.7: Node Exporter dashboard

4.3 Testing Methodology

The explanation of all the configuration tests explores the hardware aspects of the machine and the software configuration that is tested, listing all the versions of the frameworks, and all the hardware configurations. After that, all the test setups are analyzed in order to ensure the repeatability of them.

4.3.1 Test Environment: Hardware and Software Configuration

The test environments include both software and hardware configurations. In order to provide a clear context of the tests performed, it is important to give significant information about the hardware used, and the software version applied.

Hardware Configuration

As in the previous chapters, the hardware configuration utilizes an Oracle Cloud instance. To evaluate both frameworks, the tests are currently executed on a single-node setup. This approach is sufficient for now, as it guarantees simplicity in extracting and calculating the metrics. The node is equipped with 4 CPU cores, 15 GB of RAM, and 35 GB of disk space. Subsequently, other tests are performed in a three node cluster, in order to test the middleware in a real case scenario. In

this case each node is equipped with 2 CPU cores, 15 GB of RAM, and 35 GB of disk space.

Software Configuration

Regarding the applications, they were written using Java 21. The Spring Boot project was built using Spring Boot 3.5.7, and OpenJDK version "21.0.3" LTS version. The Micronaut, instead, was built using Micronaut 4.9.4, and with GraalVM version of GraalVM CE 21.0.2 (Java 21).

4.3.2 Procedures to Ensure Repeatability and Consistency of Results

All testing procedures are structured into three distinct phases: cleaning, environment setup (teardown and deployment), and execution.

1. **Cleaning**: The database and the volume are purged. This ensures that each test starts from a pristine state, without any residual data that could affect the results.

2. **Environment setup**: It consists of shutting down the currently running framework and deploying the other, in order to make the tests as isolated as possible.

3. **Execution**: The test is executed by performing all the three API calls, listed above. After that, for ensuring statistical validity of the results, each test is repeated 30 times. Furthermore, the entire test process is automated using Python scripts. This automation guarantees consistency, and ensures that all test iterations adhere strictly to the exact same execution timing and conditions.

Chapter 5

Experimental analysis and Results

The focus is now on the execution of the load testing, presenting the workload scripts, and the final results collected from the analysis. As reported previously, the testing procedure is performed with 30 repetitions to ensure statistical validity.

5.1 Performance Test Results

The metric results, how they are collected, and the evidence derived from the data, are obtained by performing "Load Testing", which consists of generating incremental requests. At each iteration the number of flows increases, from (1,5,10,50) with a maximum of 100 users, demonstrating how the system handles the gradually increasing workload.

All data generated by Prometheus queries are exported and aggregated into CSV files, to execute automated analysis via custom scripts, in order to show the data in generated graphs.

they represent an average over time of all the values recorded during the testing phases. Since the individual tests start at different times, and the metrics have varying timestamps, the X-axis of the graphs does not represent absolute timestamps, but rather the relative time elapsed from the beginning of each test. This normalization is necessary to correctly calculate and plot the average values over time.

The results are grouped in two main types of graphical representations. The first graph type compares two frameworks for each microservice, providing a detailed view on which microservice suffers most during the test phase; while, the second graph represents the summary of the tests grouped by frameworks, giving details on which framework is more performant.

5.1.1 Load Testing

The load tests are utilized to calculate the metrics M1... M5. The next subsequent paragraphs are divided for number of flows, presenting for each flow the results of the tests. In these specific load tests, it is important to note that the Java Non-Heap memory metric records a value of 0 for Micronaut. This occurs because the Micronaut application is compiled as a GraalVM native image, which does not utilize a traditional JVM Non-Heap memory space. Conversely, Spring Boot is deployed as a standard executable JAR on a traditional JVM. This metric will be used for evaluating the distinction from GraalVM and native image.

Load Testing: 1 Flows

This first scenario performs a single request of flow.

The figure 5.1 displays all the tests grouped by framework. It is evident a general reduction of resource consumption by Micronaut in all metrics that are showed in the graphs. The most significant difference is observed in memory metrics such RSS and WSS, where the Micronaut outperforms Spring Boot, achieving a reduction in memory footprint of around 90.9%. Moreover, a slight difference is noticeable in CPU usage, which stands at 10.9% for Spring Boot and 16.7% for Micronaut.

Regarding the data segmented by pods, Figure 5.2, 5.3, it is possible to notice that the fluxservice handles a heavier workload compared to the other microservices, in both frameworks 5.3, 5.2. In these figures, the CPU usage remains the same for both the frameworks. Regard the memory, Spring Boot exhibits greater variability (higher standard deviation) than Micronaut. In fact, Micronaut achieves less standard deviation instability in M2, and M3.

Furthermore, the relative ranking of the microservices in terms of resource consumption remains completely consistent in both graphs.

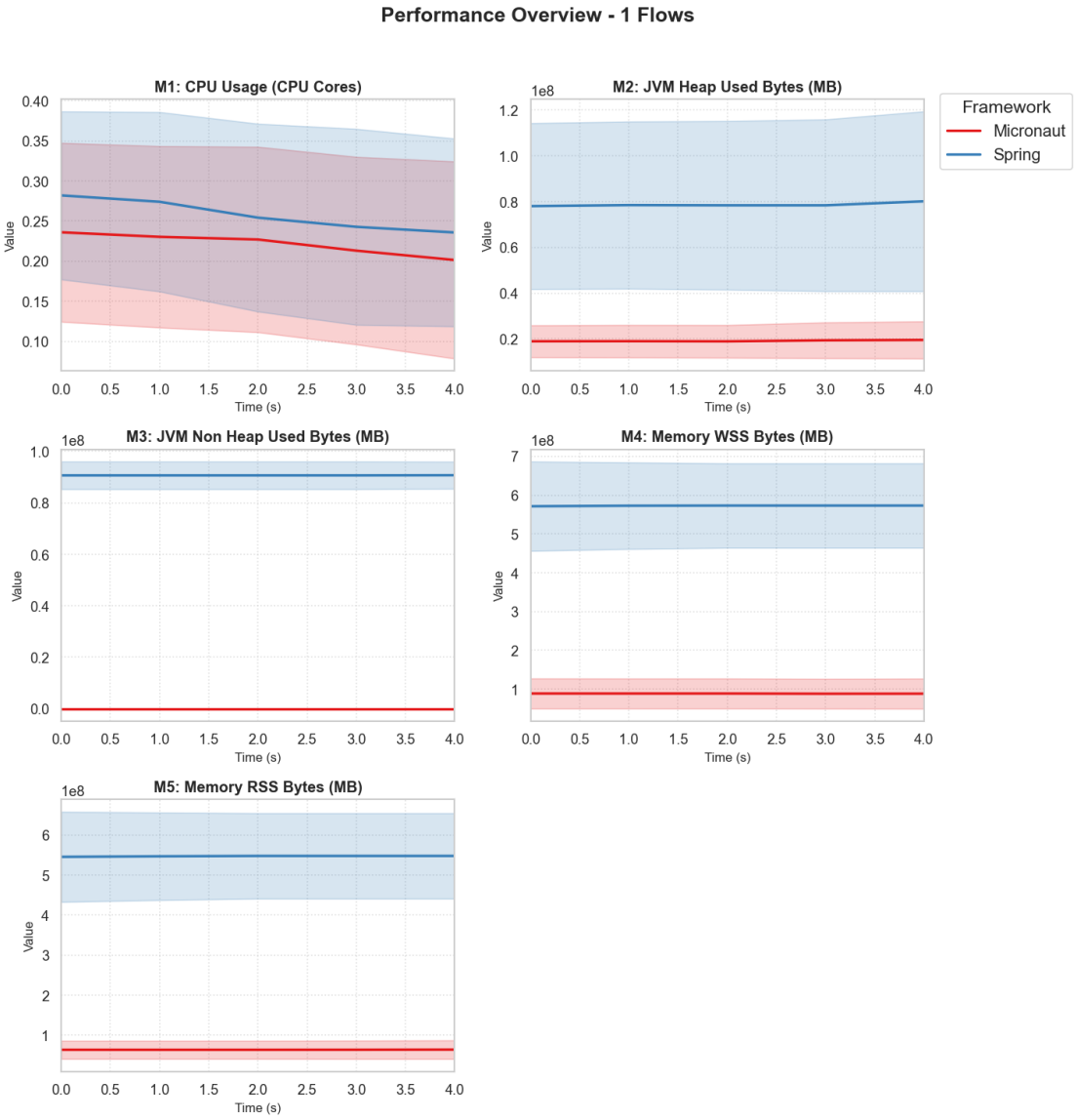


Figure 5.1: Testing 1 Flow; results grouped by microservice

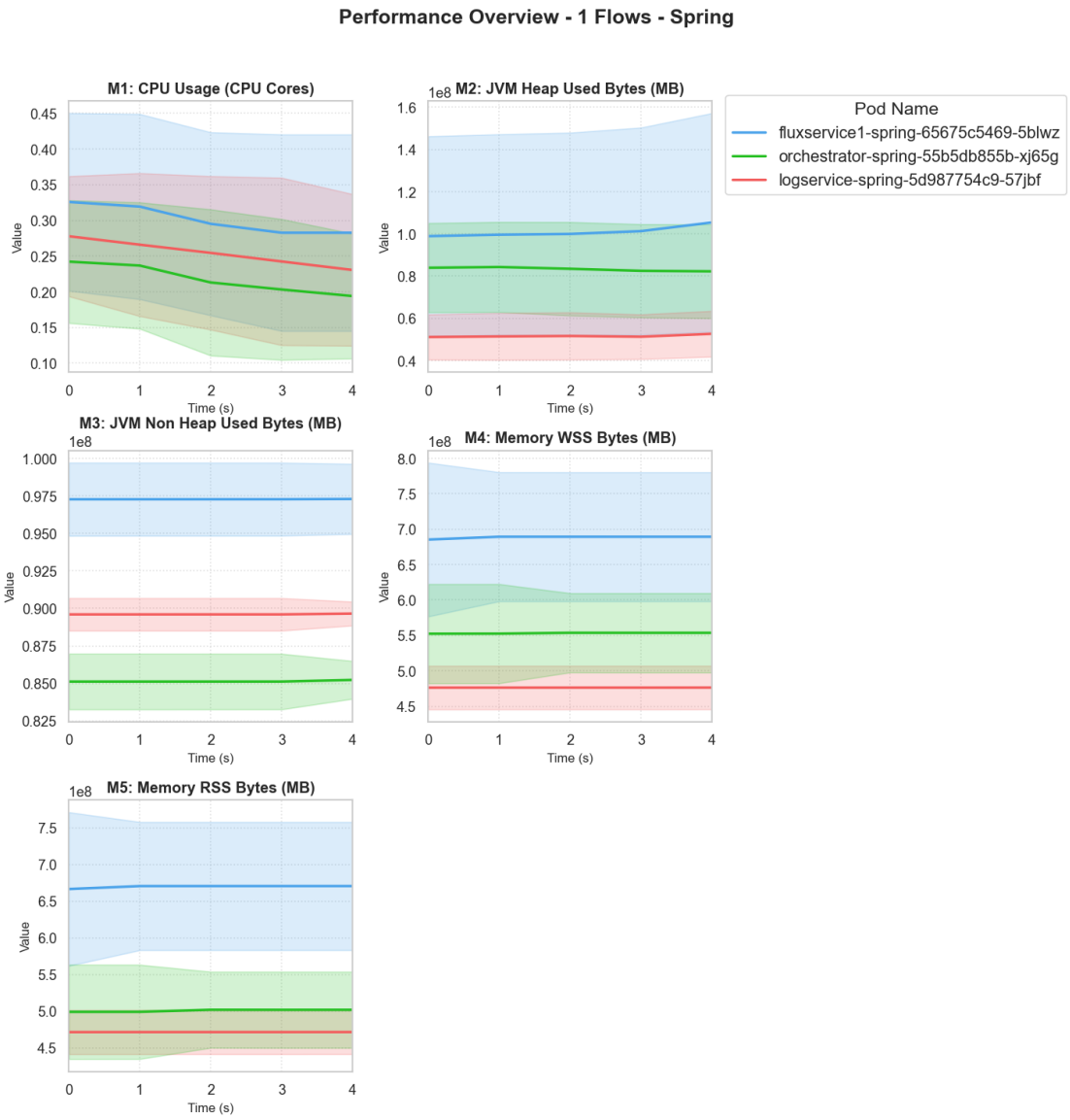


Figure 5.2: Testing 1 Flow with Spring Boot; results grouped by pods.

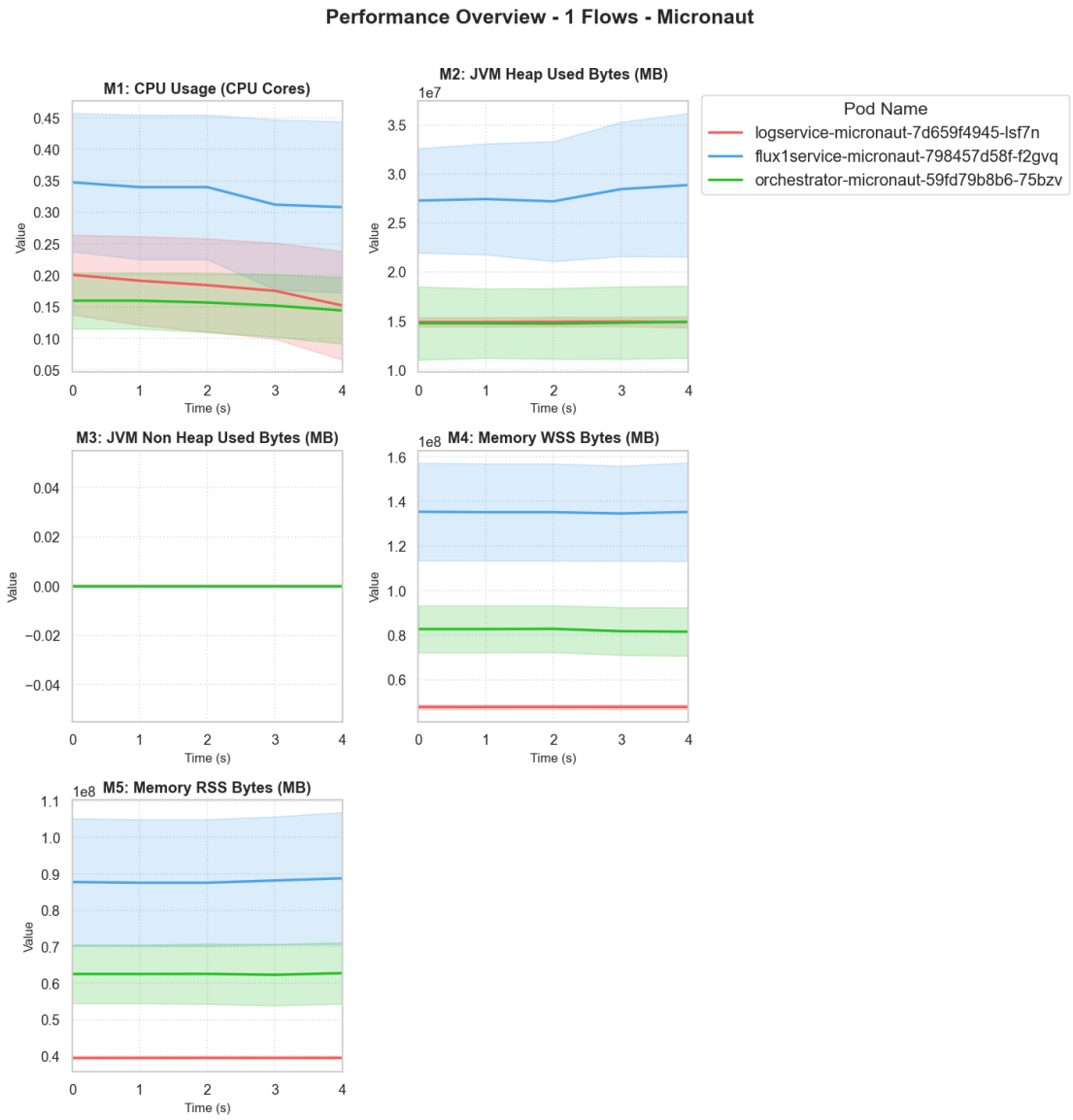


Figure 5.3: Testing 1 Flow with Micronaut; results grouped by pods.

Load Testing: 5 Flows

This scenario performs the execution of 5 concurrent flows. Compared to Figure 5.1, the primary difference in Figure 5.4 lies in metrics M1 and M2; the other metrics do not show significant differences. In this scenario, the CPU usage (M1) has more stability than in the single-user load test. Nevertheless, M2 changes, and shows a slight increment from 0.8 to 0.9 MB. Regarding the performances grouped by pods, it is possible to underline more stability about the metric M2

exhibits greater stability in Micronaut compared to Spring Boot, which experiences a sudden increase.

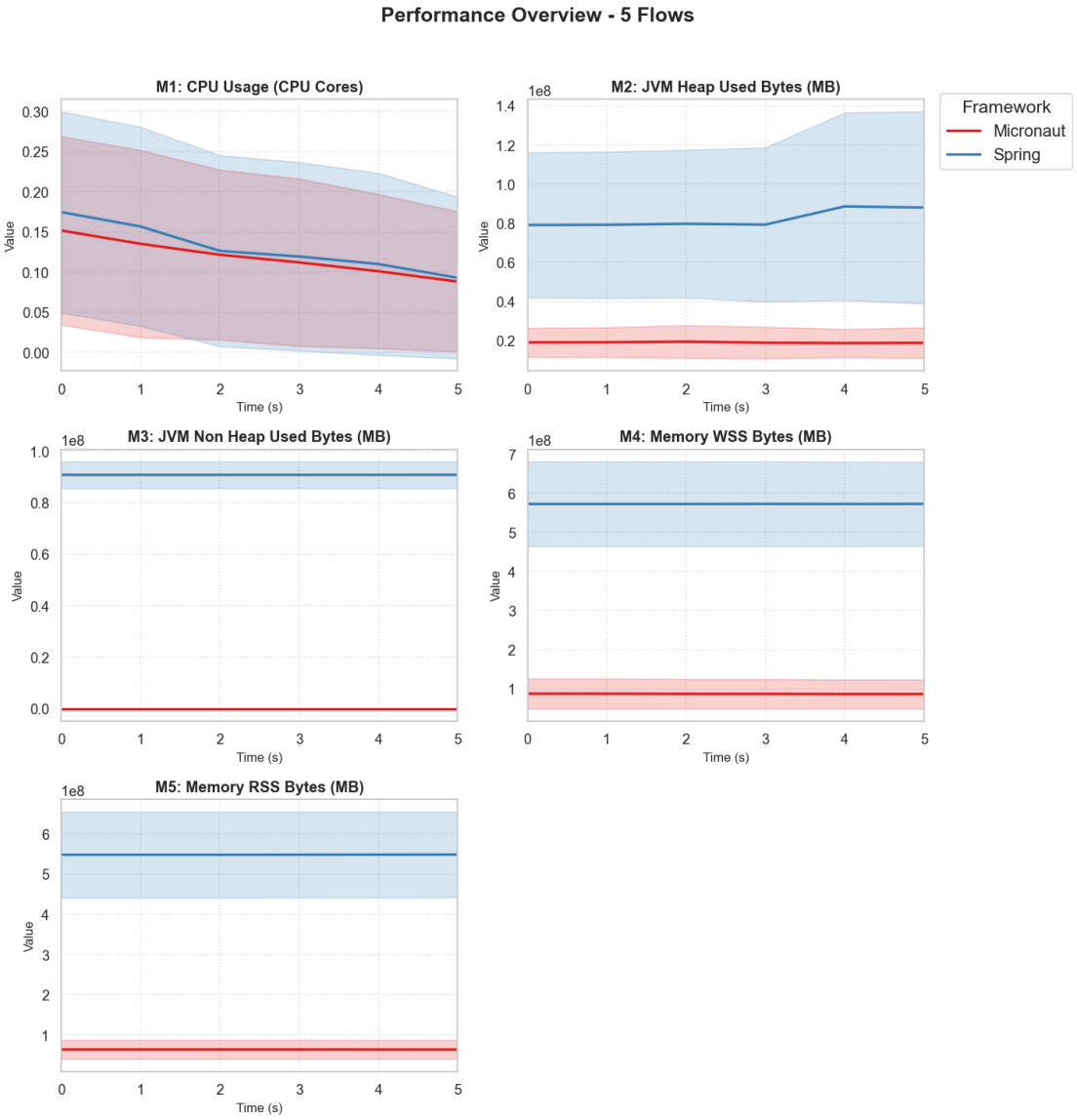


Figure 5.4: Testing 5 Flow; results grouped by microservice.

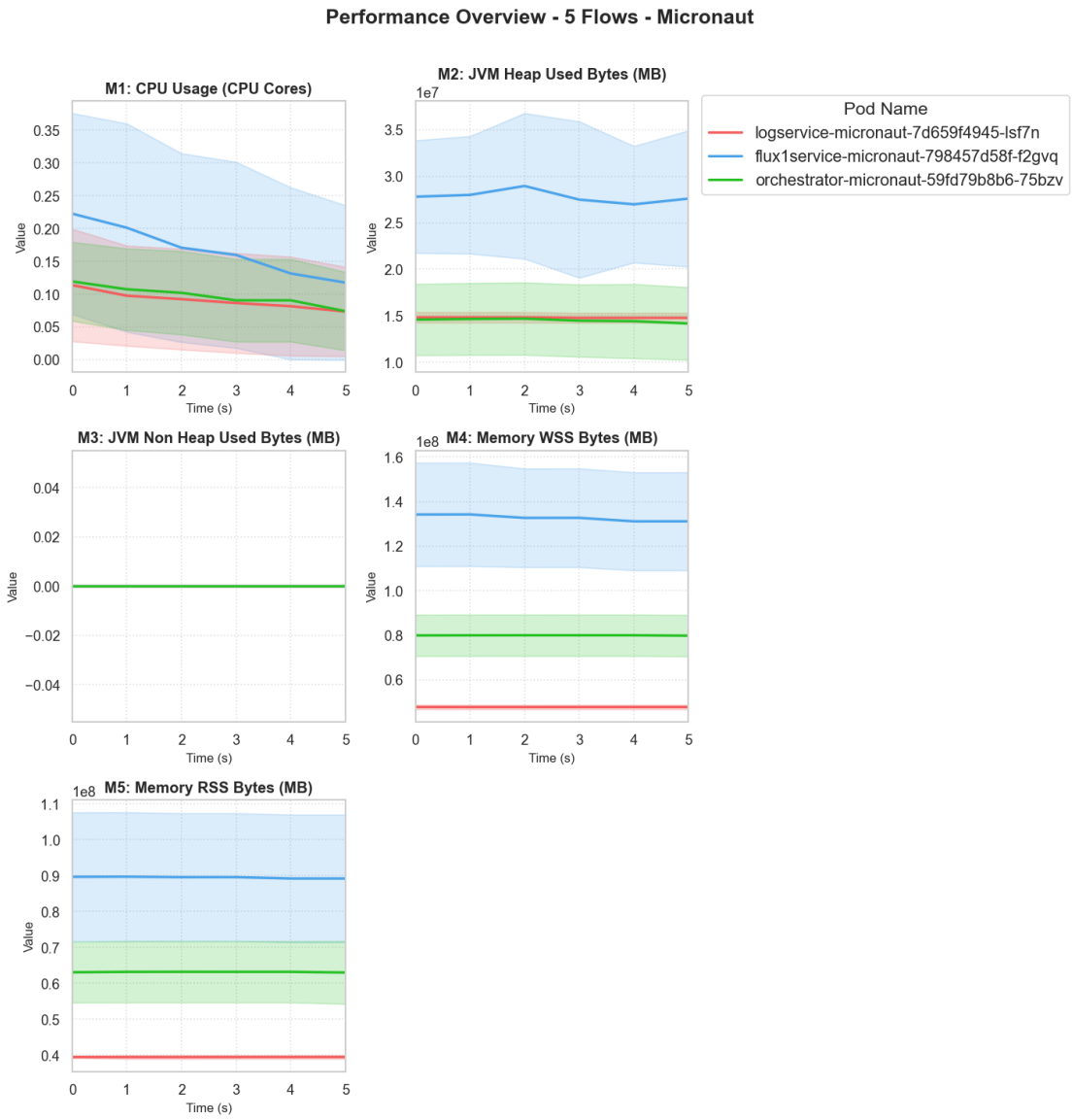


Figure 5.5: Testing 5 Flow with Spring Boot; results grouped by pods.

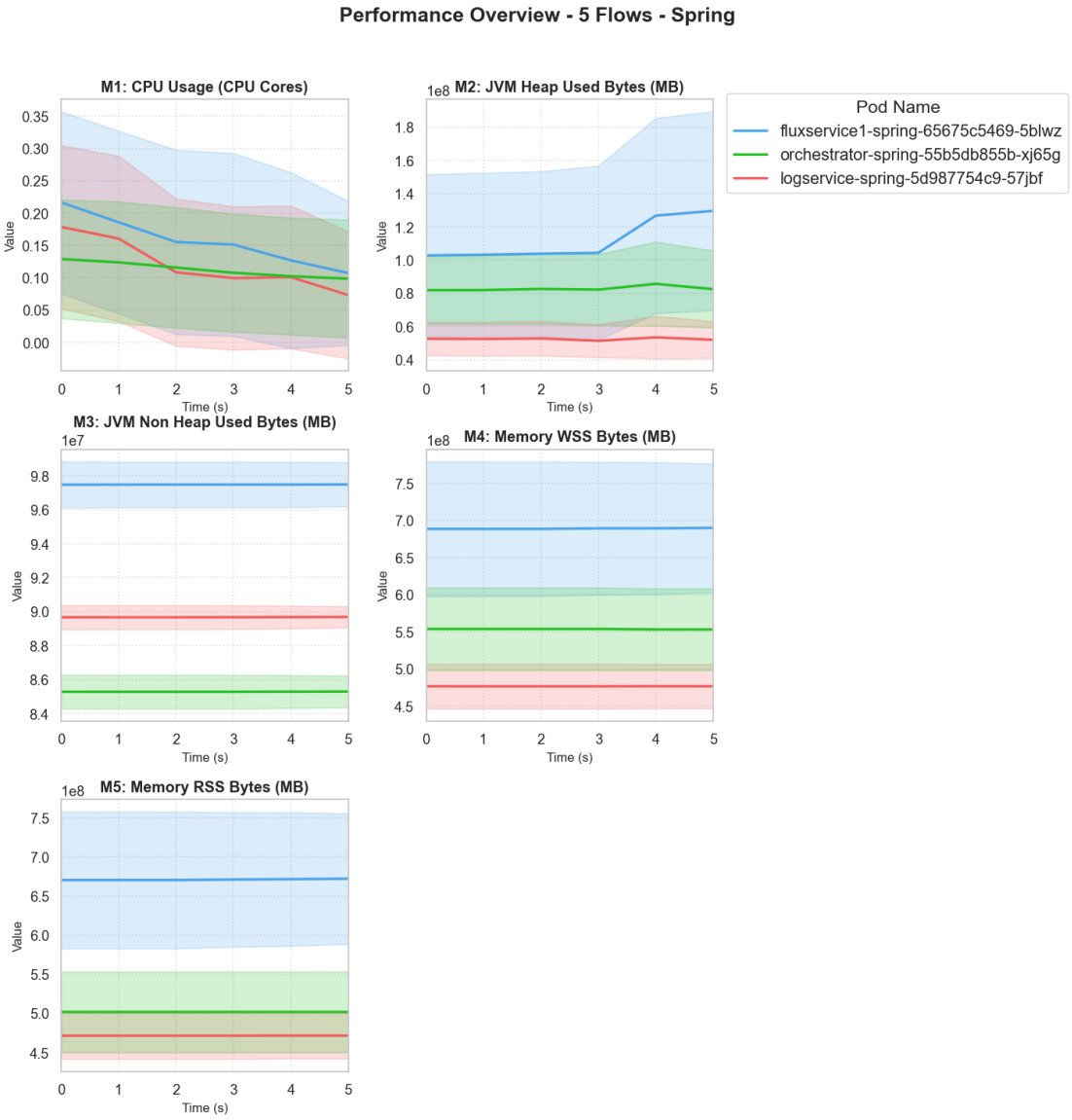


Figure 5.6: Testing 5 Flow with Micronaut; results grouped by pods.

Load Testing: 10 Flows

This scenario performs the execution of 10 concurrent flows. The system, already warmed, handles better a larger quantity of data. However, it exhibits a larger standard deviation compared to the previous tests. Regarding the M2 metric, the average value remains largely unchanged, but it shows greater instability, fluctuating between 1.4 MB to 0.4 MB. The M4 and M5 metrics remain the same as the previous results.

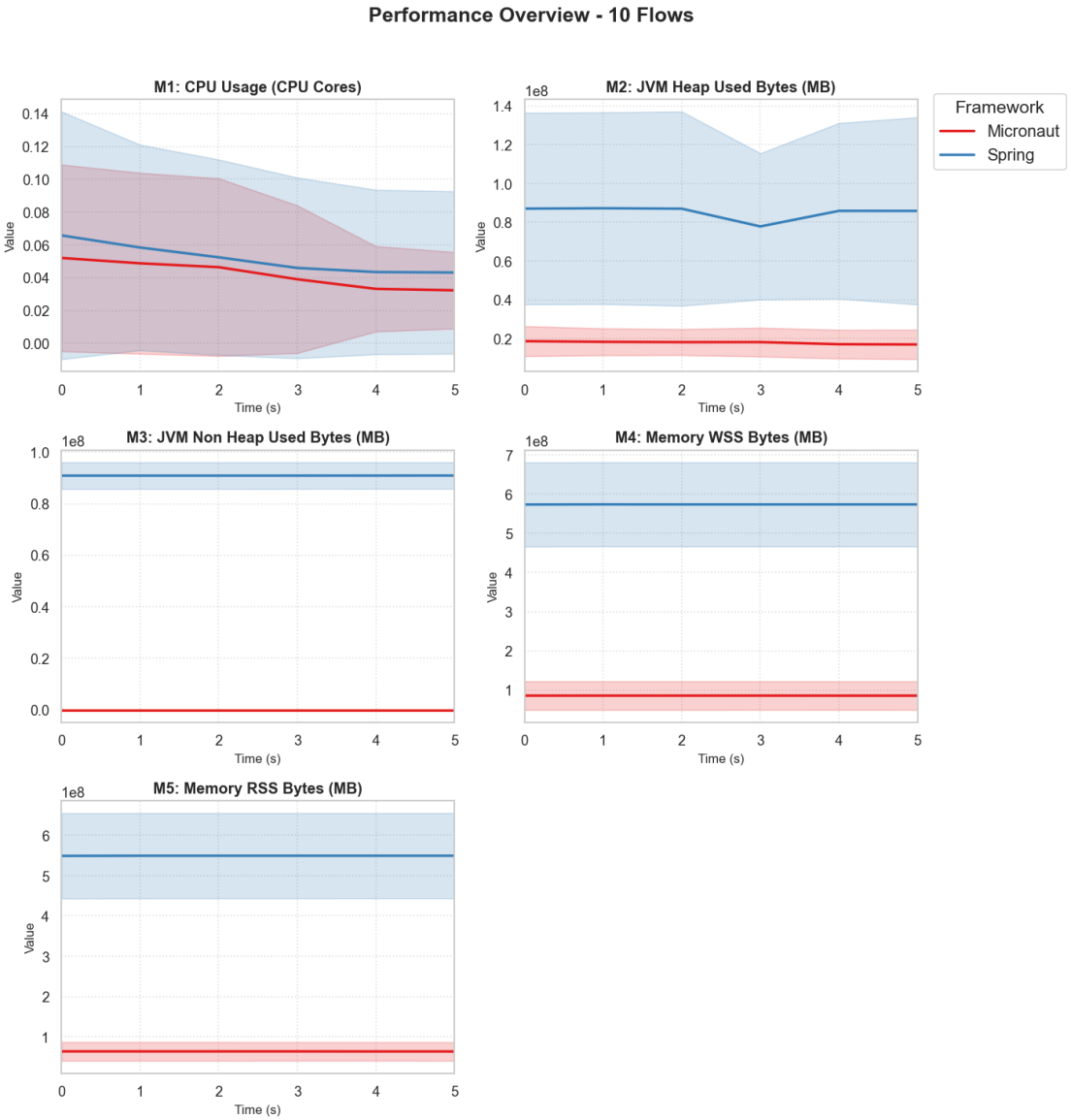


Figure 5.7: Testing 10 Flow; results grouped by microservice.

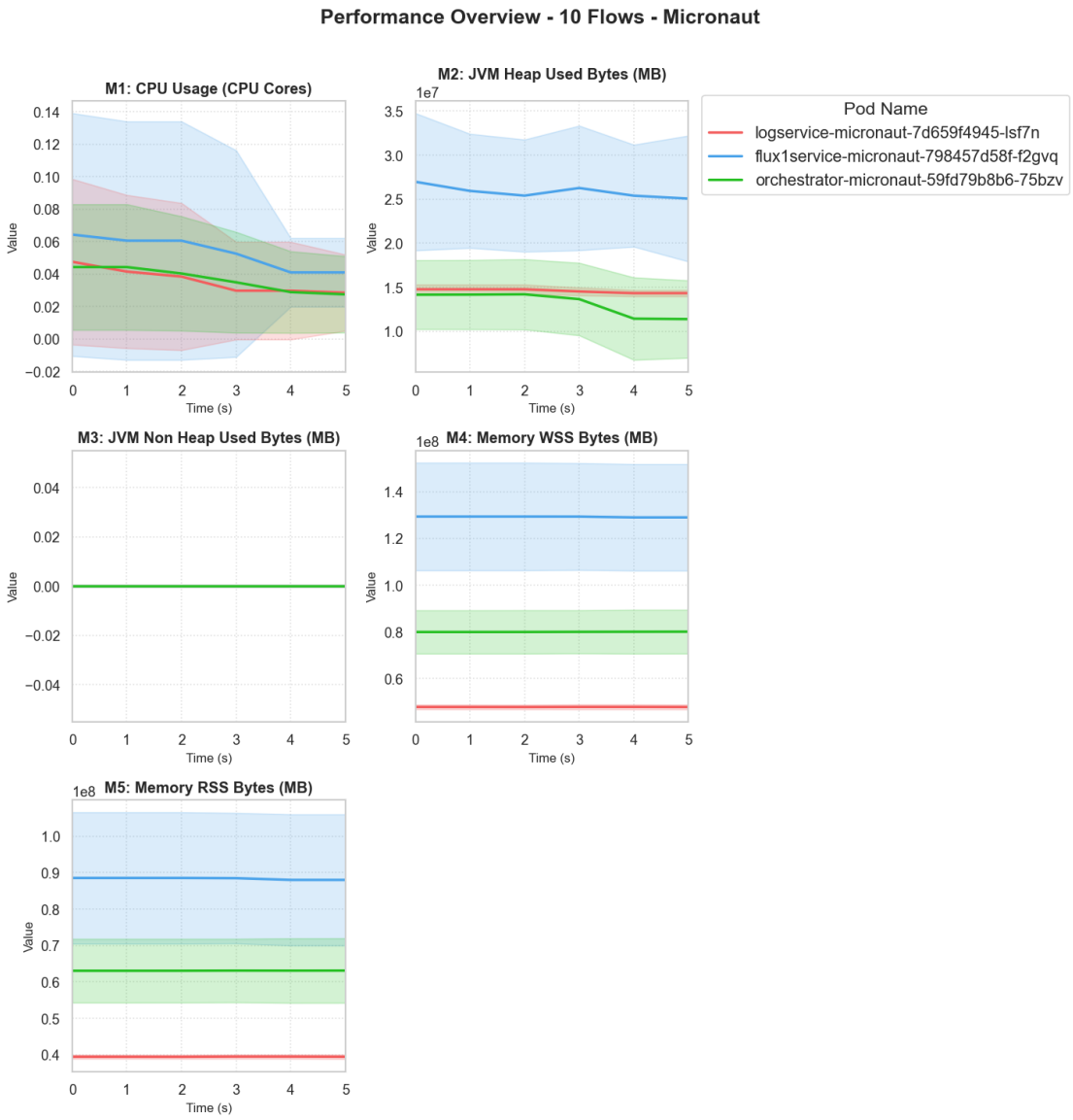


Figure 5.8: Testing 10 Flow with Spring Boot; results grouped by pods.

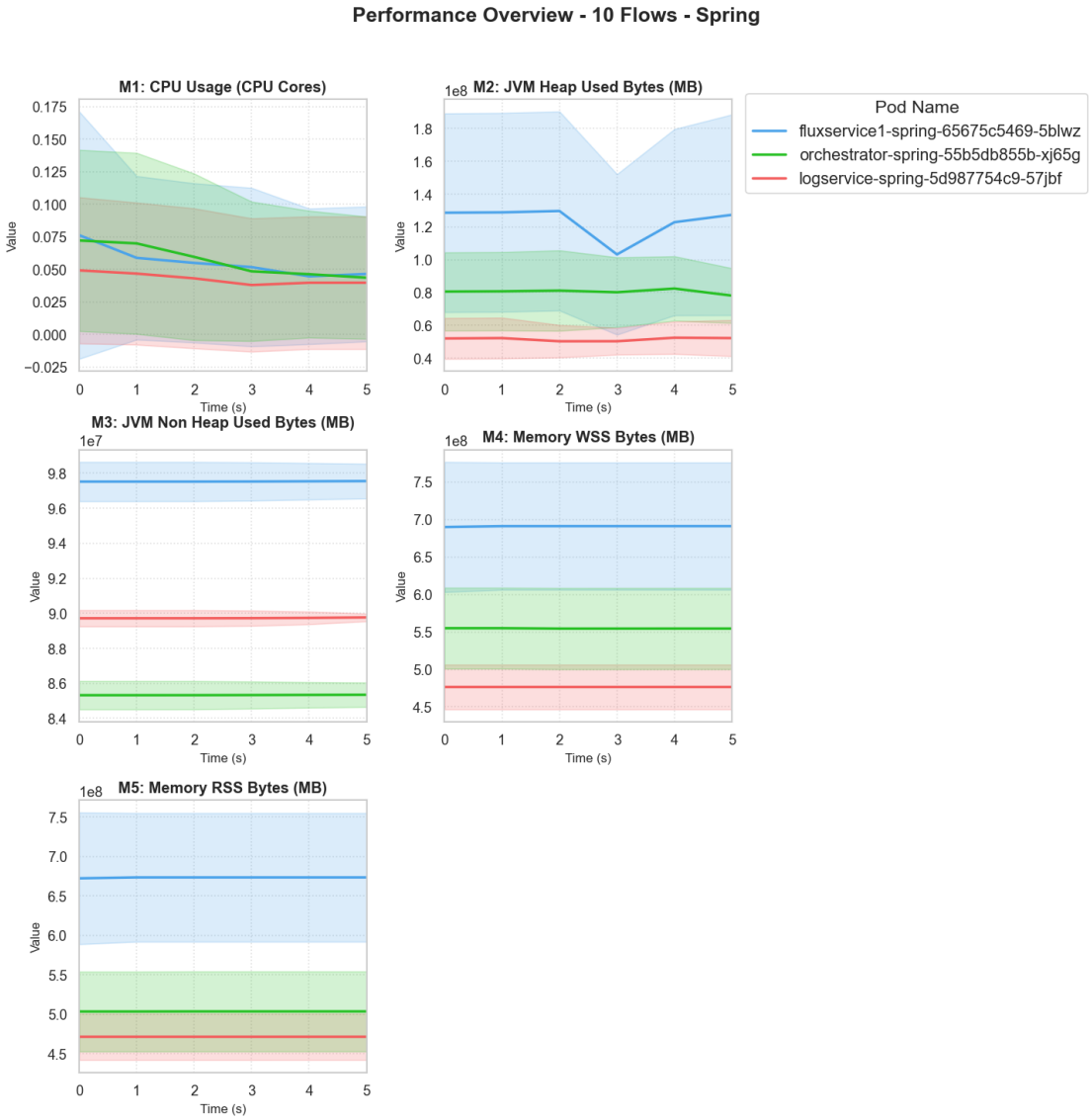


Figure 5.9: Testing 10 Flow with Micronaut; results grouped by pods.

Load Testing: 50 Flows

This load testing is performed with a simulation of 50 current flows. In this case, the use of JVM heap Memory remains approximately the same on average, but with more pronounced fluctuations during the execution. The CPU increases his value up to 0.110 CPU cores for Spring Boot and 0.090 CPU cores for Micronaut. Regarding the memory, the Working Set Size (WSS) experiences a slight, almost imperceptible, downward trend. In the graphs grouped by pods, Figures 5.11 and 5.12, it is possible to observe that in Spring Boot, the Orchestrator performs

more in M2 and M5 compared to the Log service. In contrast, in Micronaut, the results are reversed.

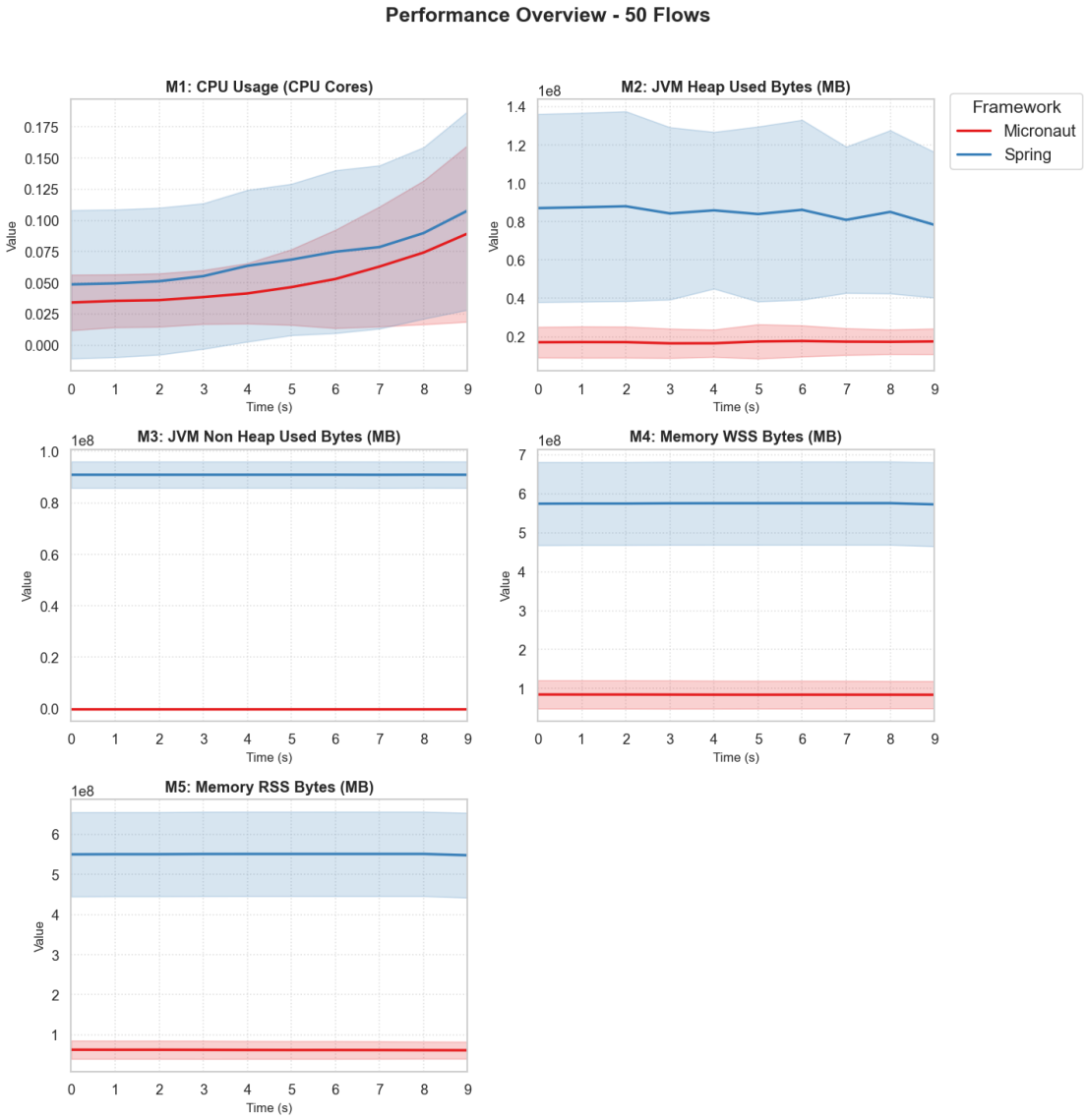


Figure 5.10: Testing 50 Flow; results grouped by microservice.

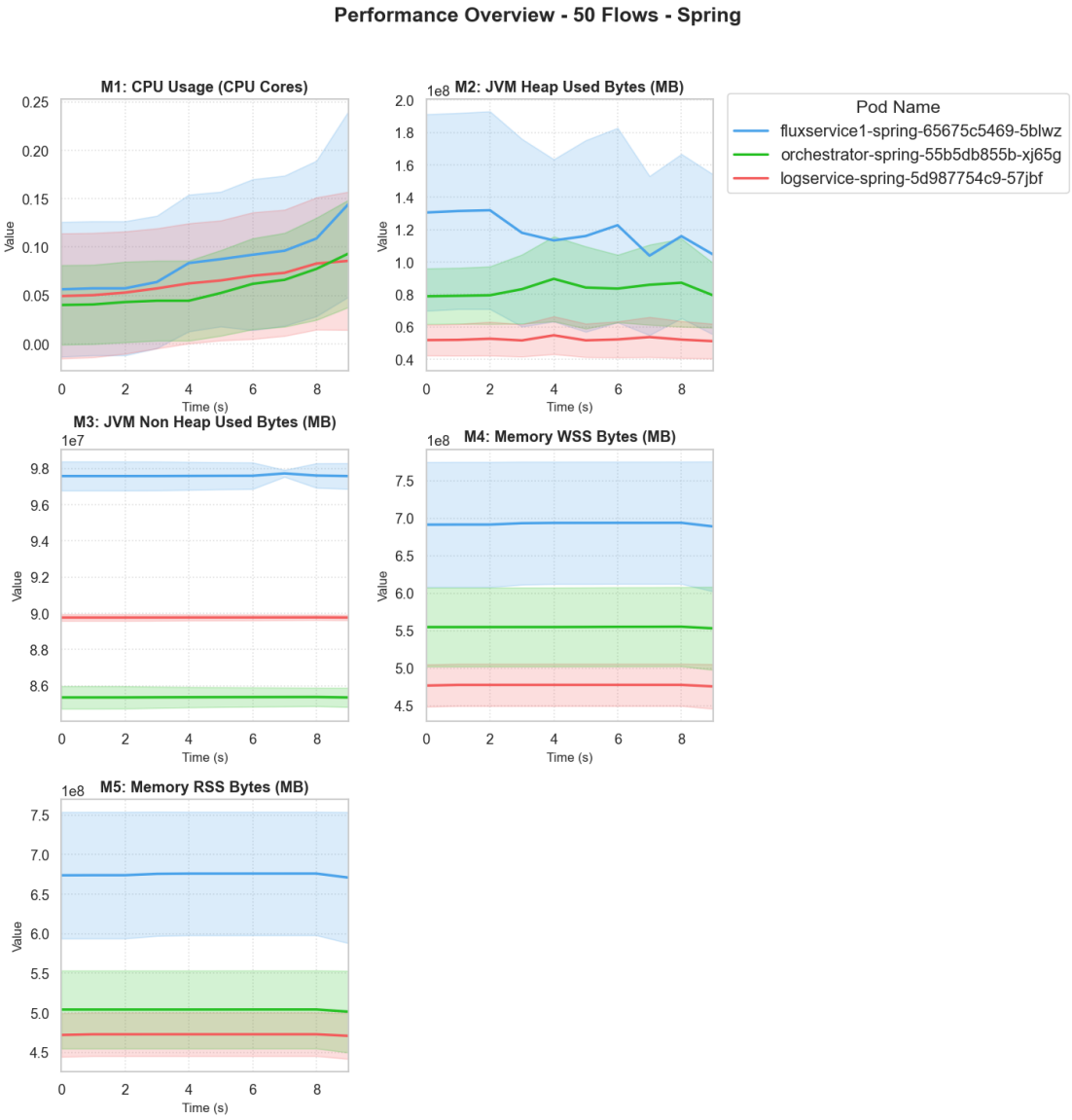


Figure 5.11: Testing 50 Flow with Spring Boot; results grouped by pods.

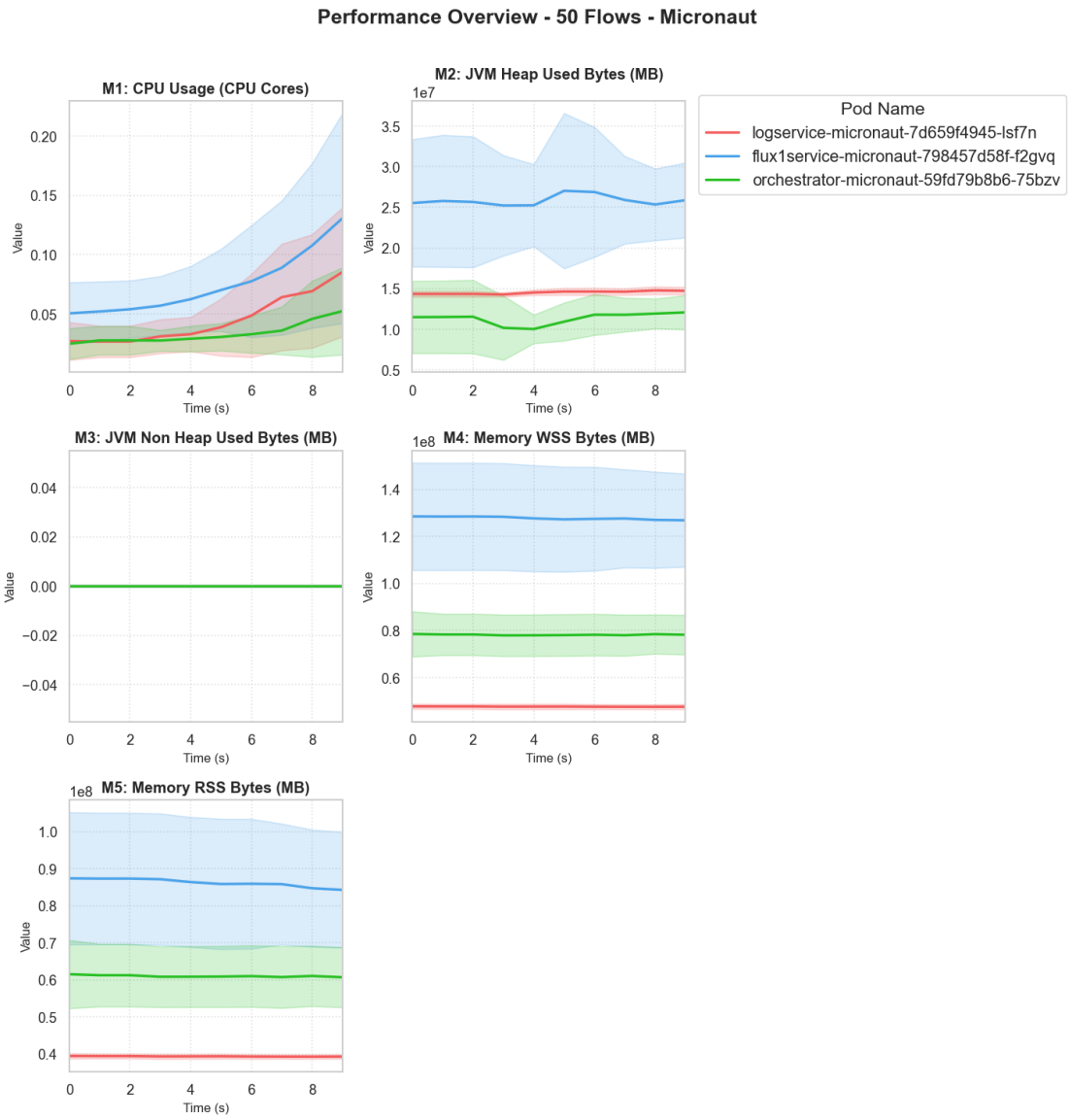


Figure 5.12: Testing 50 Flow with Micronaut; results grouped by pods.

Load Testing: 100 Flows

This Load testing is performed with a simulation of 100 current flows. This is the first scenario where the Spring Boot shows a reduction of the tests duration, which end in 14s compared the 15 seconds of Micronaut. In this case, the JVM Heap Memory usage of Spring Boot has more instability with respect to the previous tests. Moreover, its standard deviation is larger than its Micronaut counterpart. The CPU continues to increase its value up to 0.24 CPU cores for Spring Boot and

0.22 CPU cores for Micronaut. In graphs grouped by pods, Figure 5.14 and 5.15, Spring Boot shows greater stability in metrics M4 and M5 compared to Micronaut which exhibits an increase in these metrics.

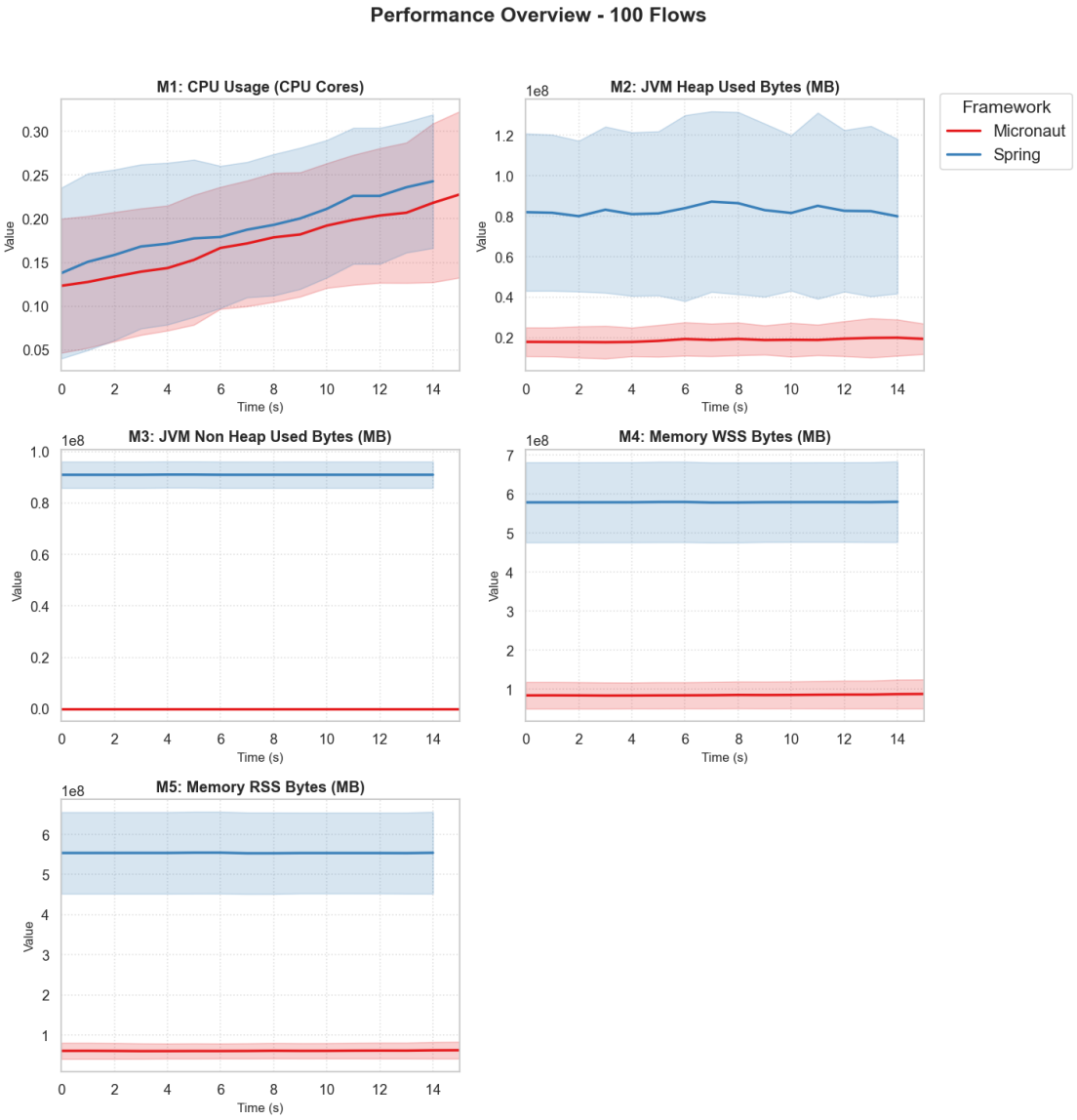


Figure 5.13: Testing 100 Flow; results grouped by microservice.

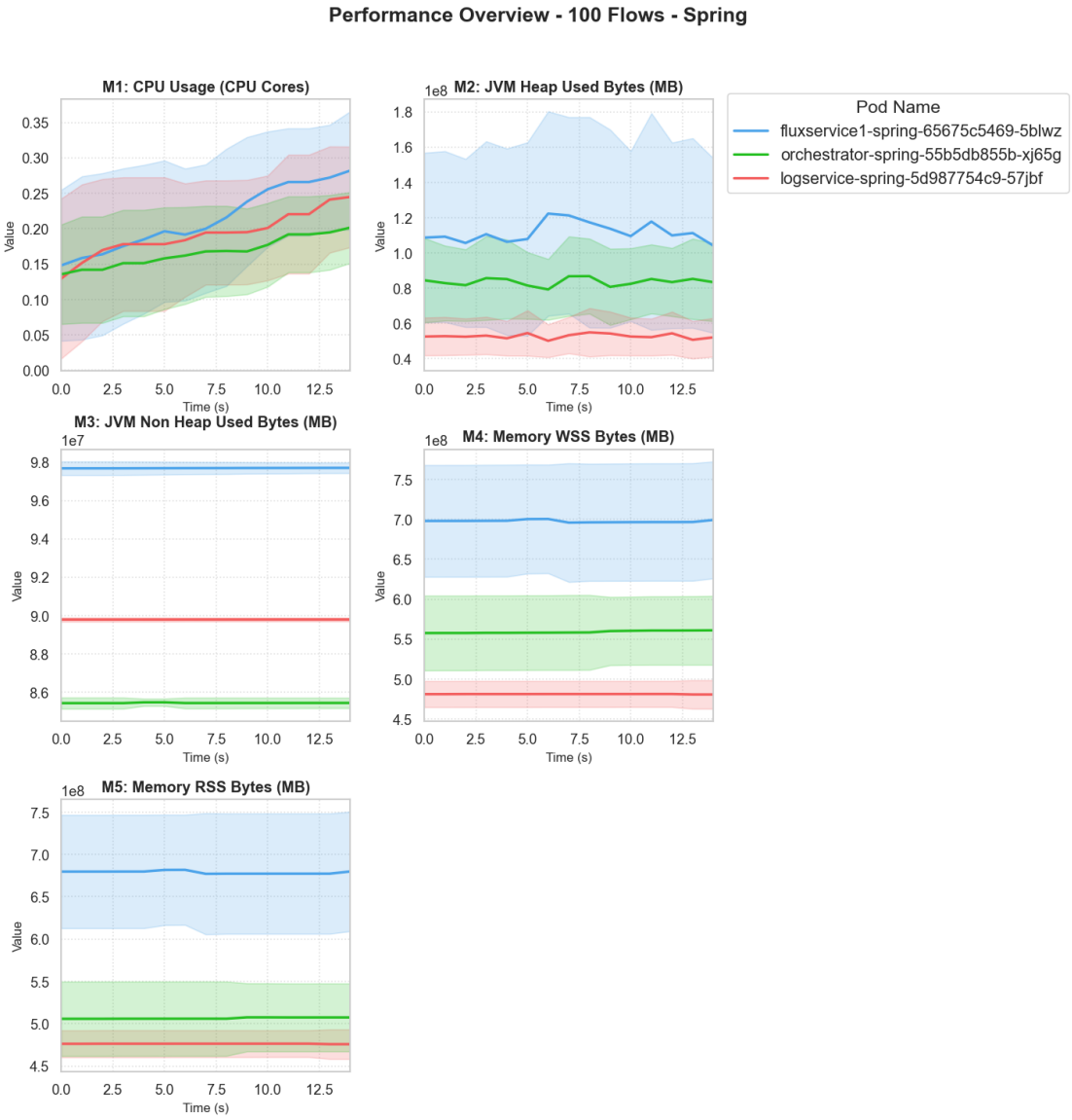


Figure 5.14: Testing 100 Flow with Spring Boot; results grouped by pods.

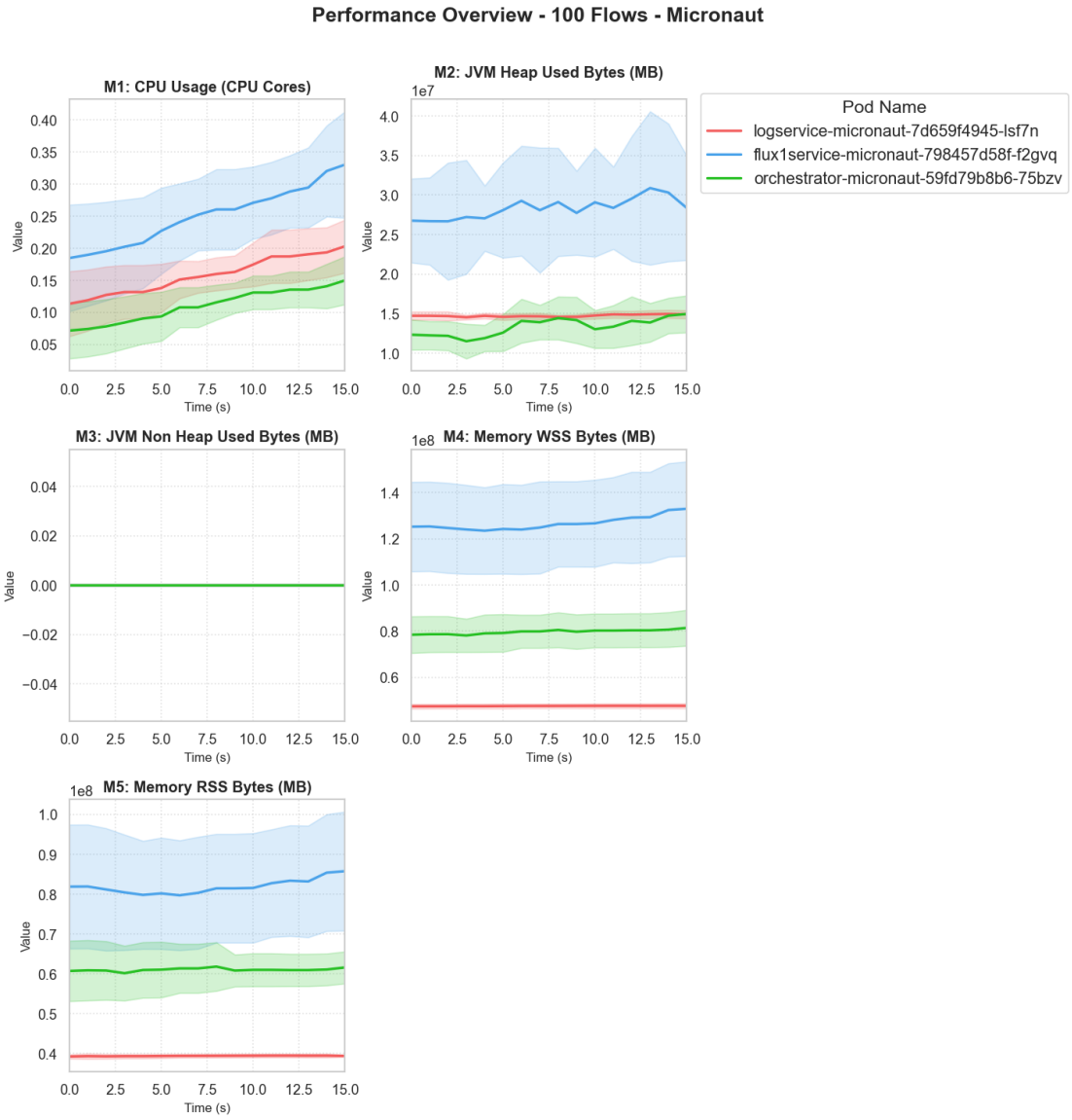


Figure 5.15: Testing 100 Flow with Micronaut; results grouped by pods.

5.1.2 Throughput and Average Response Time

After exposing the differences of performances between GraalVM and JDK, is important to present the throughput and the average response time recorded during the load testing phase. In order to better illustrate the behavior of the metrics is this tests, all the results are grouped by frameworks and plotted with the number of flows in X-axis and the specific metric values on the Y-axis.

Regarding the average response time the increase is approximately linear for both

Micronaut and Spring Boot. Even though Micronaut has generally less response time than Spring Boot, the least performed bedder in the last test (100 flows). Both frameworks perform the same with 1-flow workload recording an average response time of 150 ms. As the concurrent load increases, Micronaut initially scales better than Spring Boot counterpart. The two frameworks end with the same result at 100 flows, performing the test in approximately 1500 ms, which confirms a semi-linear degradation behavior.

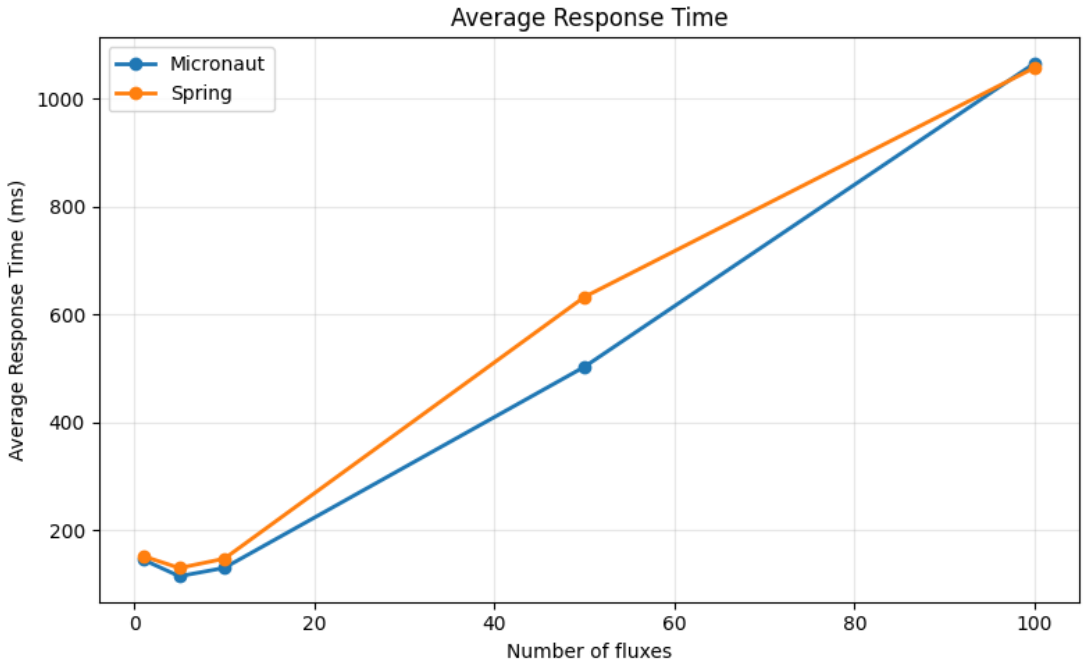


Figure 5.16: Average Response Time: grouped by framework.

Regarding the throughput, the situation is inverted: Spring Boot achieves a higher throughput than Micronaut. Despite this difference, both frameworks exhibit a similar overall trend: Spring Boot's throughput increases from 5 req/s at 1 flow to 57 req/s at 100 flows; while Micronaut, increases from 5 req/s at 1 flow to 52 req/s at 100 flows. This confirms that the trend is not perfectly linear. It is possible to see the divergence point in a 5-flow test, with the gap widening as the workload increases.

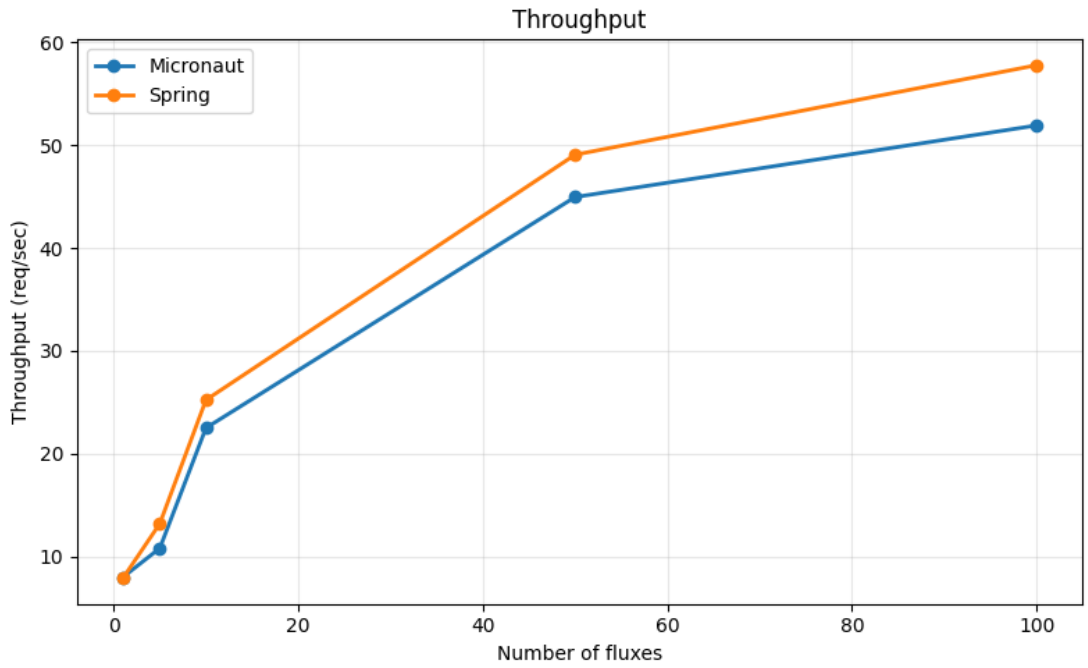


Figure 5.17: Throughput: grouped by framework.

5.2 Comparison with Native and GraalVM architecture

It is relevant for the research provide further information on the advantage of the building of native image, highlighting the difference between an application running with GraalVM and an application compiled as a native image, both implementing the Micronaut framework.

5.2.1 Load Testing

The Load Testing was executed on a GraalVM JVM application to demonstrate the performance benefits of native images compared to traditional VM applications. Figure 5.18 shows an overview of the load testing scenario.

As shown in Figure 5.18, the native image has a significant advantage in RSS, WSS and JVM Heap. However, the most striking result is achieved with the `non_heap_used_bytes` metric, where native image registers a value of 0, having no impact compared to its GraalVM JVM-compiled counterpart. Regarding the CPU usage, the difference is not as evident. An interesting observation is that the standard Micronaut JVM application spends slightly less time executing the

overall test suite than its Micronaut Native counterpart. Regarding the memory, Micronaut with native image performs betters under each metric, achieving, in metric M2, M4 and M5 less than the half of Micronaut compared with GraalVM.

Figure 5.18: Flow's overview with Micronaut; results grouped by pods.

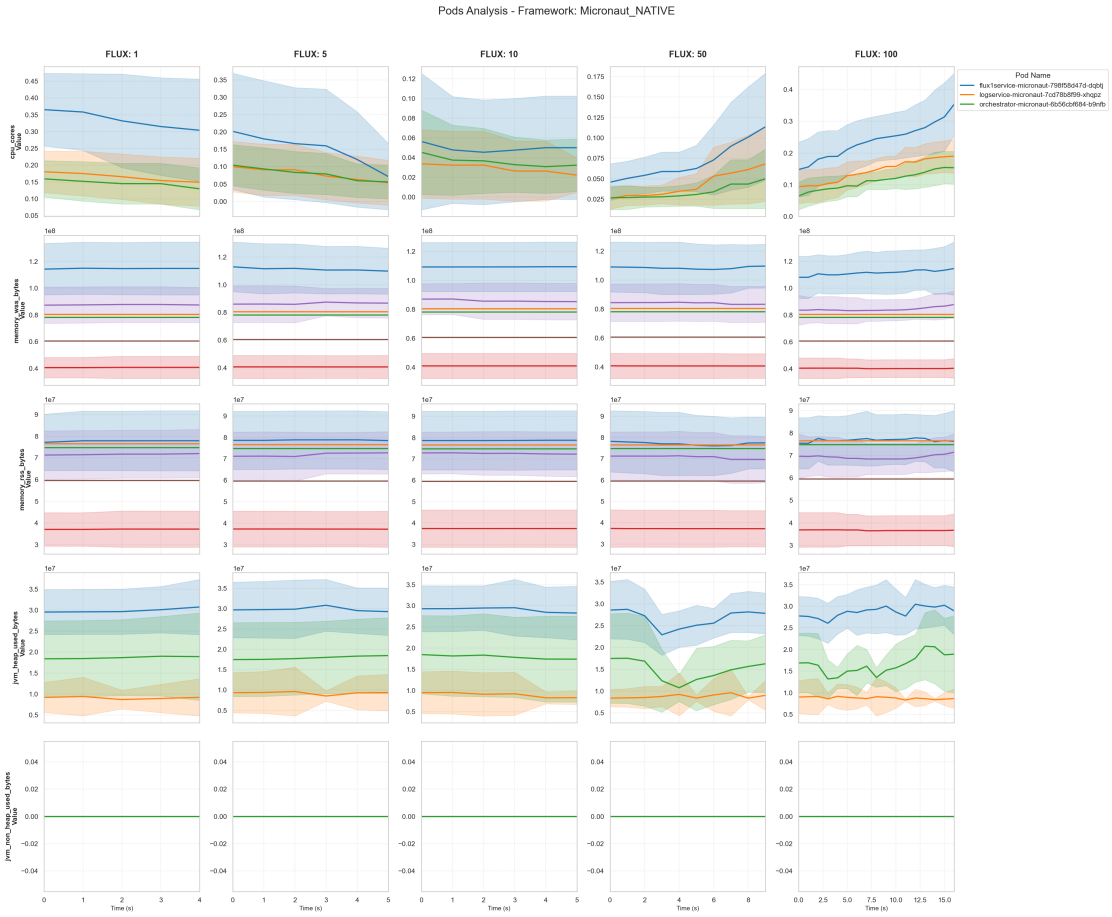


Figure 5.19: Flow's overview; results grouped by microservice.

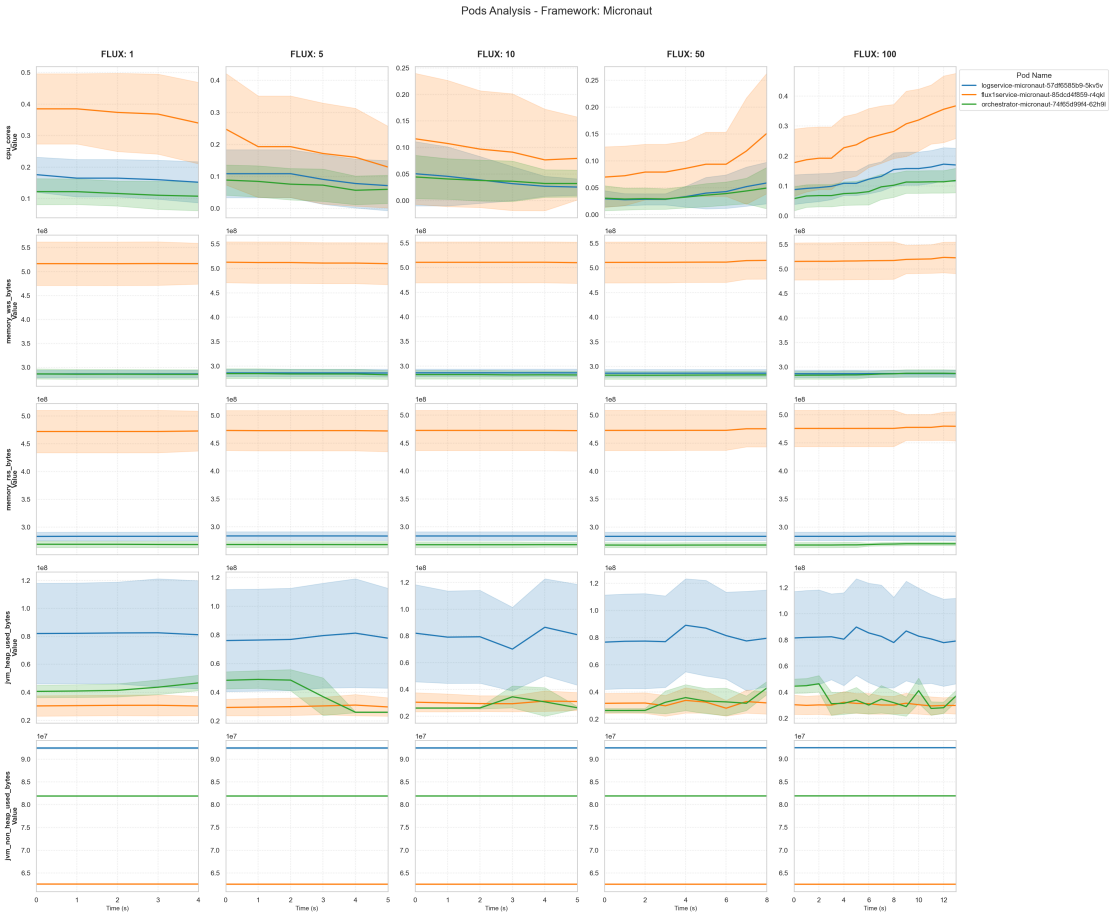


Figure 5.20: Flow's overview with Spring Boot; results grouped by pods.

5.2.2 Throughput and Average Response Time

Regarding the throughput and the average response time, the Micronaut native image exhibits both lower response times and a lower maximum throughput compared to the Micronaut GraalVM JVM image. The response time displays a difference of approximately 10ms at 1 flows, and 300ms at 100 flows. The throughput, instead, follows the same trend for both but results in a performance gap of approximately 8 req/s at the 100-flow mark.

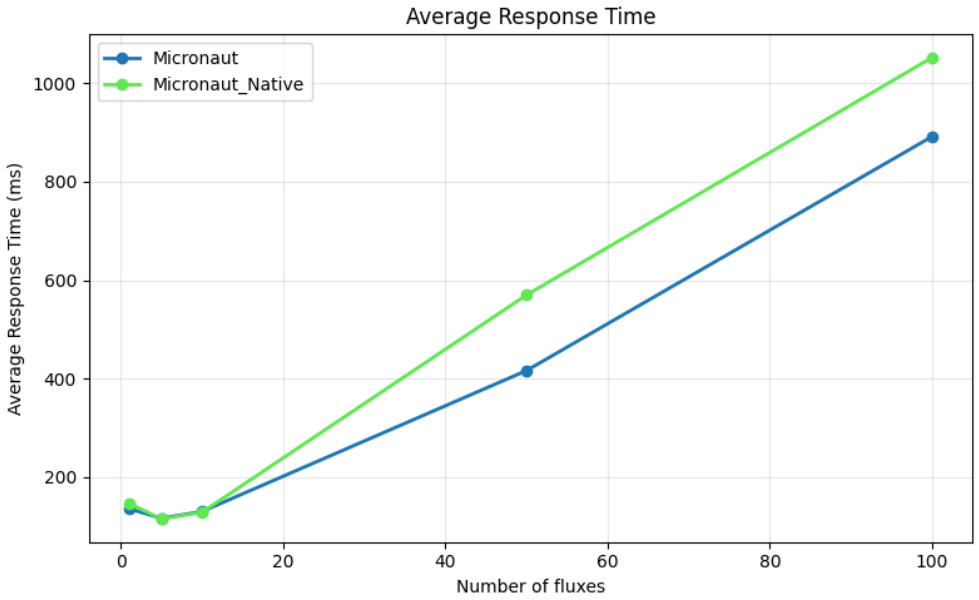


Figure 5.21: Testing 100 Flow; results grouped by microservice.

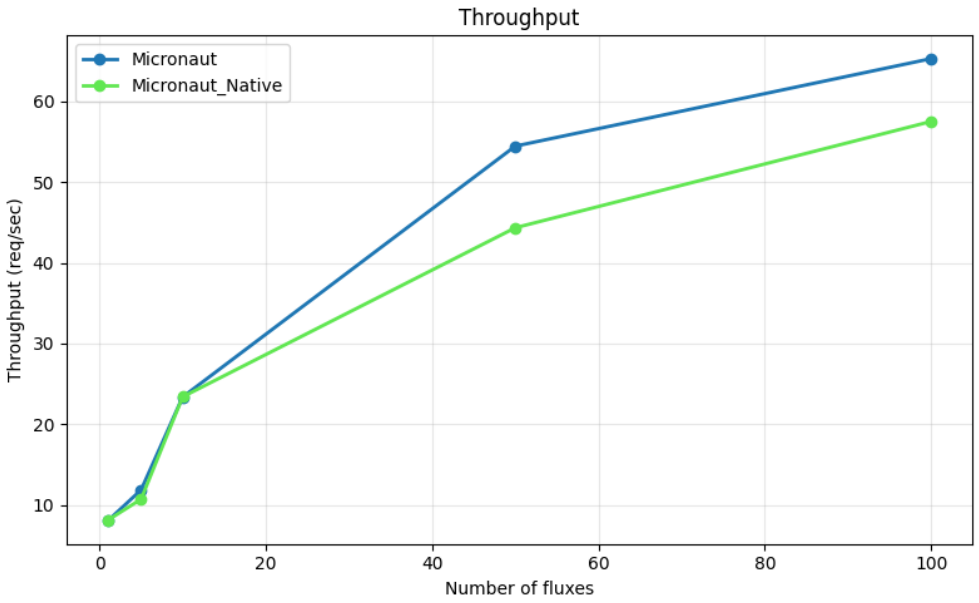


Figure 5.22: Testing 100 Flow with Spring Boot; results grouped by pods.

Chapter 6

Conclusions and Future Developments

6.1 Summary of Experimental Results

By analyzing the load testing results presented in the previous chapter, it is evident that the usage of natively compiled frameworks is significantly more efficient than a Virtual Machine (VM) counterpart. The result of average response time and throughput do not exhibit drastic changes. However, after a rigorous analysis, we can conclude that these specific performance metrics are less dependent on the chosen framework, which has slightly improvements, but it is heavily reliant on the underlying microservices architecture, that is the same for both system.

However, the most substantial difference is observed in memory usage. Across all memory-related metrics (M2, M3, M4, M5), the reduction achieved by the native image is impressive. This outcome strongly supports the adoption of native image in order to minimize the RAM consumption in cloud environment nodes, reducing the costs for maintaining the software. The adoption of Micronaut with native image has both advantages and disadvantages.

The advantages encountered during the development of Micronaut and Native Image were:

- Lack of comprehensive documentation and community resources, due to the recently introduced framework.

- Extended build time required to generate the native executable. This lengthy compilation process can slow down debugging operations. Furthermore, many existing Java libraries lack compatibility with the Ahead-Of-Time (AOT) compilation model, generating runtime errors, and increasing debugging time.

Consequently, the study suggests adopting the combination of Micronaut and GraalVM Native Image, in scenarios where memory performance is critical. The downside is that the development team accounts for the potentially longer development cycles caused by the limited availability of resources, compared to the highly established Spring Boot ecosystem.

6.2 Future Developments

Regarding the future developments, the study serves as evaluation for the company to adopt the Spring Boot or the Micronaut with the Native Image or the JVM approach. As outlined in Chapter 3, the company plans are to evolve this project by adding and managing multiple data fluxes, in order to apply more complex transformations. In the future the company intends to expand the project also by adding a multi-agend ecosystem base on artificial intelligence. This future integration aims to make the software more efficient and intelligent, aligning the infrastructure with current industry standards and technological trends.

Bibliography

[1] Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. Sebastopol, CA: O'Reilly Media, 2015 (cit. on pp. 1, 2, 4).

[2] Ma-Keba Frye. *6 Fundamental Principles of Microservice Design*. 2023. URL: https://www.salesforce.com/blog/microservice-design-principles/ (cit. on p. 2).

[3] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. «Basic Concepts and Taxonomy of Dependable and Fault-Tolerant Computing». In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004), pp. 11–33 (cit. on p. 2).

[4] Microsoft Learn. *Microservices Architecture Style*. 2023. URL: https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices (cit. on p. 4).

[5] Nehal Navlani. *The Power of Polyglot Architecture: A Guide for Modern Organisations*. 2023 (cit. on p. 4).

[6] Melanie Purkis. *What Is Server Clustering?* (Cit. on p. 5).

[7] WSO2 Research. *Scalability Modeling Using Universal Scalability Law*. URL: https://wso2.com/blog/research/scalability-modeling-using-universal-scalability-law/ (cit. on p. 6).

[8] Neil Gunther. *Universal Scalability Law*. URL: http://www.perfdynamics.com/Manifesto/USLscalability.html (cit. on p. 6).

[9] Nitin Khaitan. *Resilient Microservices Design*. 2025 (cit. on pp. 8, 9).

[10] Lorenzo De Lauretis. «From Monolithic Architecture to Microservices Architecture». In: *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. Oct. 2019, pp. 93–96. DOI: 10.1109/ISSREW.2019.00050. (Visited on 12/19/2025) (cit. on p. 9).

[11] GeeksforGeeks. *Monolithic Architecture - System Design*. URL: https://www.geeksforgeeks.org/system-design/monolithic-architecture-system-design/ (cit. on pp. 9, 14).

[12] Shraddha Zladhe. *Maintainability in Microservices Architecture: Principles You Shouldn't Ignore*. 2025. URL: https://medium.com/@shraddhazladhe/maintainability-in-microservices-architecture-principles-you-shouldnt-ignore-3837d655742b (cit. on p. 10).

[13] Terence Bennett. *4 Microservices Examples: Amazon, Netflix, Uber, and Etsy*. 2025. URL: https://blog.dreamfactory.com/microservices-examples (cit. on pp. 11–13).

[14] George Lawton. *How Microservices Patterns Made Uber's Architecture Perform Better*. 2025. URL: https://www.theserverside.com/feature/How-microservices-patterns-helped-Uber-systems-perform-better (cit. on p. 12).

[15] GeeksforGeeks. *Advantages and Disadvantages of Microservices Architecture*. URL: https://www.geeksforgeeks.org/system-design/what-are-the-advantages-and-disadvantages-of-microservices-architecture/ (cit. on p. 16).

[16] Oleksii Dushenin. *Monolithic Architecture. Advantages and Disadvantages*. 2021. URL: https://datamify.medium.com/monolithic-architecture-advantages-and-disadvantages-e71a603eec89?source=rss------java-5 (cit. on p. 17).

[17] Davide D'Antonio. *Microservices Patterns*. 2023. URL: https://medium.com/finanza-tech/microservices-patterns-25da7fb85bd9 (cit. on pp. 18, 19, 37).

[18] Crishantha Nanayakkara. *Microservices Patterns: Service Discovery Patterns*. 2023. URL: https://medium.com/cloud-native-daily/microservices-patterns-part-03-service-discovery-patterns-97d603b9a510 (cit. on pp. 19, 20).

[19] GeeksforGeeks. *Circuit Breaker Pattern in Microservices*. URL: https://www.geeksforgeeks.org/system-design/what-is-circuit-breaker-pattern-in-microservices/ (cit. on p. 20).

[20] *Virtual Machines (VMs) vs Containers: What's The Difference?* URL: https://www.bmc.com/blogs/containers-vs-virtual-machines/ (visited on 03/08/2026) (cit. on p. 22).

[21] *Containers vs Virtual Machines: A Detailed Comparison*. URL: https://www.datacamp.com/blog/containers-vs-virtual-machines?utm%5C_cid=19589720821%5C&utm%5C_aid=152984011134%5C&utm%5C_campaign=230119%5C_1-ps-other%5Ctextasciitilde%20dsa-tofu%5Ctextasciitilde%20all%5C_2-b2c%5C_3-emea%5C_4-prc%5C_5-na%5C_6-na%5C_7-le%5C_8-pdsh-go%5C_9-nb-e%5C_10-na%5C_11-na%5C&utm%5C_loc=

1008585-%5C&utm%5C_mtd=-c%5C&utm%5C_kw=%5C&utm%5C_source=google%5C&utm%5C_medium=paid%5C_search%5C&utm%5C_content=ps-other%5Ctextasciitilde%20emea-en%5Ctextasciitilde%20dsa%5Ctextasciitilde%20tofu%5Ctextasciitilde%20blog%5Ctextasciitilde%20data-engineering%5C&gad%5C_source=1%5C&gad%5C_campaignid=19589720821%5C&gbraid=0AAAADQ9WsEVuM9TzUTmTFtxIsklRBqB5%5C&gclid=Cj0KCQiA6Y7KBhCkARIsAOxhqtMsOg0dA3qhmczYclfzj3uSJfvuQNcsOM6pZzxQ4JOHki4YqP1J2UUaAsqSEALw%5C_wcB (visited on 12/18/2025) (cit. on p. 23).

[22] *What Is Docker?* 23:23:11+00:00. URL: https://www.geeksforgeeks.org/devops/introduction-to-docker/ (visited on 12/18/2025) (cit. on p. 24).

[23] Kyle Penfound, Dagger, Cortney Nickerson, and Kubeshop. *Production-Grade Container Orchestration*. URL: https://kubernetes.io/ (visited on 12/18/2025) (cit. on p. 24).

[24] *Remote Procedural Call (RPC) Mechanism*. 0:00:13+00:00. URL: https://www.geeksforgeeks.org/computer-networks/what-is-rpc-mechanism-in-distributed-system/ (visited on 12/20/2025) (cit. on p. 29).

[25] «(PDF) Engineering Cloud-Native Microservices in Java: A Scalable Approach for Modern Enterprise Software Architectures». In: *ResearchGate* (Sept. 2025). (Visited on 12/09/2025) (cit. on p. 32).

[26] *The Life Before SpringBoot | Spring Framework | A Journey Through Early Java Development*. 2024. (Visited on 12/09/2025) (cit. on p. 33).

[27] *What Is Java Spring Boot? | IBM*. Oct. 2021. URL: https://www.ibm.com/think/topics/java-spring-boot (visited on 12/08/2025) (cit. on p. 34).

[28] *Introduction to Spring Boot*. 1:35:41+00:00. URL: https://www.geeksforgeeks.org/springboot/introduction-to-spring-boot/ (visited on 12/08/2025) (cit. on p. 34).

[29] *Micronaut vs. Spring Boot: A Detailed Comparison*. Aug. 2024. URL: https://dev.to/adaumircosta/micronaut-vs-spring-boot-a-detailed-comparison-4og5 (visited on 12/09/2025) (cit. on p. 34).

[30] Somnath Musib. *Spring Boot In Practise*. M A N N I N G. ISBN: ISBN-13: 978-1-61729-881-3 (cit. on p. 35).

[31] *Introduction to the Spring IoC Container and Beans :: Spring Framework*. URL: https://docs.spring.io/spring-framework/reference/core/beans/introduction.html (visited on 12/12/2025) (cit. on p. 35).

[32] Sachinmewar. *BeanFactory and BeanDefinition in Spring*. May 2023. (Visited on 12/14/2025) (cit. on pp. 35, 36).

[33] Bolot Kasybekov. *Understanding BeanFactory and ApplicationContext in Spring Boot: A Deep Dive*. Mar. 2025. (Visited on 12/12/2025) (cit. on p. 35).

[34] *Spring - Understanding Inversion of Control with Example*. 21:23:18+00:00. URL: https://www.geeksforgeeks.org/advance-java/spring-understanding-inversion-of-control-with-example/ (visited on 12/12/2025) (cit. on p. 36).

[35] *BeanDefinition (Spring Framework 7.0.1 API)*. URL: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/config/BeanDefinition.html (visited on 12/20/2025) (cit. on p. 36).

[36] *BeanDefinition (Spring Framework 7.0.1 API)*. URL: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/config/BeanDefinition.html (visited on 12/14/2025) (cit. on p. 36).

[37] *Bean Life Cycle in Java Spring*. 1:00:35+00:00. URL: https://www.geeksforgeeks.org/java/bean-life-cycle-in-java-spring/ (visited on 12/13/2025) (cit. on p. 36).

[38] Alexander Obregon. *How Spring Boot Auto-Configuration Works*. Oct. 2025. (Visited on 12/14/2025) (cit. on p. 38).

[39] Himanshu. *Spring Boot Is Killing Your Startup Time: Here's Why — And How to Fix It*. May 2025. (Visited on 12/18/2025) (cit. on p. 39).

[40] *Speed up Spring Boot Startup Time | Baeldung*. Dec. 2021. URL: https://www.baeldung.com/spring-boot-startup-speed (visited on 12/18/2025) (cit. on p. 40).

[41] *How to Reduce Spring Boot Memory Usage? | Baeldung*. Sept. 2025. URL: https://www.baeldung.com/spring-boot-memory-usage-optimization (visited on 12/18/2025) (cit. on p. 40).

[42] Tharindu Dulshan. *Boosting Performance: Essential Memory Optimization Techniques for Spring Boot Applications*. Jan. 2025. (Visited on 12/18/2025) (cit. on pp. 40, 41).

[43] *How to Reduce Spring Boot Memory Usage? | Baeldung*. Sept. 2025. URL: https://www.baeldung.com/spring-boot-memory-usage-optimization (visited on 12/18/2025) (cit. on p. 41).

[44] *Home*. URL: https://micronaut.io/ (visited on 12/20/2025) (cit. on pp. 41, 42).

[45] Atiksha Pundir. *Micronaut Framework*. Oct. 2025. (Visited on 03/08/2026) (cit. on p. 42).

[46] *Micronaut AOT*. URL: https://micronaut-projects.github.io/micronaut-aot/snapshot/guide/ (visited on 12/20/2025) (cit. on pp. 42, 43).

[47] Ankita Patel. *Micronaut and GraalVM: Building Native Applications*. Oct. 2024. (Visited on 12/21/2025) (cit. on p. 44).

[48] M. Šipek, D. Muharemagić, B. Mihaljević, and A. Radovan. «Enhancing Performance of Cloud-based Software Applications with GraalVM and Quarkus». In: *2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO)*. Sept. 2020, pp. 1746–1751. DOI: 10.23919/MIPRO48935.2020.9245290. (Visited on 12/21/2025) (cit. on pp. 44, 49–52).

[49] Richard Hutcheson, Austin Blanchard, Noah Lambaria, Jack Hale, David Kozak, Amr S. Abdelfattah, and Tomas Cerny. «Software Architecture Reconstruction for Microservice Systems Using Static Analysis via GraalVM Native Image». In: *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Mar. 2024, pp. 12–22. DOI: 10.1109/SANER60148.2024.00008. (Visited on 12/21/2025) (cit. on pp. 44, 46, 48, 50, 51).

[50] Andrew Binstock, Karin Kinnear, Claire Breen, Jennifer Kurtz, Leslie Steere, Lea Anne Bantsari, Karen Perkins, Simon Roberts, and Mikalai Zaikin. «Java Magazine, May/June 2018». In: (2018) (cit. on p. 44).

[51] M. Šipek, B. Mihaljević, and A. Radovan. «Exploring Aspects of Polyglot High-Performance Virtual Machine GraalVM». In: *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. May 2019, pp. 1671–1676. DOI: 10.23919/MIPRO.2019.8756917. (Visited on 12/24/2025) (cit. on pp. 45, 46, 53).

[52] Richard Hutcheson, Austin Blanchard, Noah Lambaria, Jack Hale, David Kozak, Amr S. Abdelfattah, and Tomas Cerny. «Software Architecture Reconstruction for Microservice Systems Using Static Analysis via GraalVM Native Image». In: *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Mar. 2024, pp. 12–22. DOI: 10.1109/SANER60148.2024.00008. (Visited on 12/23/2025) (cit. on pp. 45, 47).

[53] *Truffle Language Implementation Framework*. URL: https://www.graalvm.org/latest/graalvm-as-a-platform/language-implementation-framework/ (visited on 12/21/2025) (cit. on p. 48).

[54] *Why GraalVM?* URL: https://www.graalvm.org/why-graalvm/ (visited on 12/21/2025) (cit. on p. 48).

[55] *Native Image Compatibility Guide*. URL: https://www.graalvm.org/latest/reference-manual/native-image/metadata/Compatibility/ (visited on 12/25/2025) (cit. on p. 48).

[56] *CRIU*. URL: https://criu.org/Main%5C_Page (visited on 12/23/2025) (cit. on p. 49).

[57] Abdullah Al Maruf, Alexander Bakhtin, Tomas Cerny, and Davide Taibi. «Using Microservice Telemetry Data for System Dynamic Analysis». In: *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. Aug. 2022, pp. 29–38. DOI: 10.1109/SOSE55356.2022.00010. (Visited on 12/31/2025) (cit. on p. 50).

[58] *Renaissance Suite, a Benchmark Suite for the JVM*. URL: https://renaissance.dev/ (visited on 12/31/2025) (cit. on p. 51).

[59] *Home*. URL: https://c4model.com/ (visited on 02/12/2026) (cit. on p. 57).

[60] *C4 Model - The Basics*. Oct. 2024. URL: https://dev.to/rafaeljcamara/c4-model-the-basics-5bk5 (visited on 02/06/2026) (cit. on pp. 57, 59).

[61] Bibhuti Bhusan Sahoo. *Flyway: Advantages and Disadvantages in Database Migration*. Oct. 2023. (Visited on 02/09/2026) (cit. on p. 66).

[62] *Persistent Volumes*. URL: https://kubernetes.io/docs/concepts/storage/persistent-volumes/ (visited on 02/09/2026) (cit. on p. 83).

[63] *Deployments*. URL: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/ (visited on 02/09/2026) (cit. on p. 83).

[64] *Service*. URL: https://kubernetes.io/docs/concepts/services-networking/service/ (visited on 02/09/2026) (cit. on p. 83).

[65] *Helm*. URL: https://helm.sh/ (visited on 02/09/2026) (cit. on p. 83).

[66] Victor R Basili, Gianluigi Caldiera, and H Dieter Rombach. «THE GOAL QUESTION METRIC APPROACH». In: () (cit. on p. 84).

[67] *Goal Question Metric Approach in Software Quality*. 18:49:08+00:00. URL: https://www.geeksforgeeks.org/software-engineering/goal-question-metric-approach-in-software-quality/ (visited on 03/09/2026) (cit. on p. 84).

[68] Robert Edward. *Difference Between Resident Set Size and Virtual Memory Size | Baeldung on Linux*. Dec. 2021. URL: https://www.baeldung.com/linux/resident-set-vs-virtual-memory-size (visited on 03/09/2026) (cit. on p. 86).

[69] *Apache JMeter - Apache JMeter™*. URL: https://jmeter.apache.org/ (visited on 02/18/2026) (cit. on p. 87).

[70] Emily H. Halili. *Apache JMeter: A Practical Beginner's Guide to Automated Testing and Performance Measurement for Your Websites*. 1. publ. From Technologies to Solutions. Birmingham: Packt Publ, 2008. ISBN: 978-1-84719-295-0 (cit. on p. 87).

[71] *Apache JMeter - Apache JMeter™*. URL: https://jmeter.apache.org/ (visited on 03/09/2026) (cit. on p. 87).

[72] *Apache JMeter - User's Manual: Elements of a Test Plan*. URL: https://jmeter.apache.org/usermanual/test%5C_plan.html (visited on 03/09/2026) (cit. on pp. 87, 88).

[73] *Apache JMeter - User's Manual: Elements of a Test Plan*. URL: https://jmeter.apache.org/usermanual/test%5C_plan.html (visited on 03/09/2026) (cit. on p. 89).

[74] *Querying Basics | Prometheus*. URL: https://prometheus.io/docs/prometheus/latest/querying/basics/ (visited on 02/19/2026) (cit. on p. 90).

[75] *About Grafana | Grafana Documentation*. URL: https://grafana.com/docs/grafana/latest/introduction/ (visited on 02/20/2026) (cit. on p. 92).

[76] *K8S Dashboard*. URL: https://grafana.com/grafana/dashboards/15661-k8s-dashboard-en-20250125/ (visited on 03/09/2026) (cit. on p. 92).

[77] *Monitoring Linux Host Metrics with the Node Exporter | Prometheus*. URL: https://prometheus.io/docs/guides/node-exporter/ (visited on 03/09/2026) (cit. on p. 93).