Università degli Studi di Padova

Università degli Studi di Padova

Department of Information Engineering

Master's Degree in
Computer Engineering

Open-source observability solution for microservice architecture

Supervisor:
Dr. Di Buccio Emanuele

Student:
Giovanni Zago
2087645

Co-supervisor:
Dr. Martini Roberto

Academic Year 2024-2025

8/4/2025

To Ester,

if we will ever meet.

Abstract

This thesis presents the design and prototyping of a scalable, open-source observability solution for enterprise IT systems with a focus on microservices architectures. The proposed framework addresses key challenges, including scalability, data correlation, relationship management, and standardization, while integrating smart alerting. A structured approach for deployment, configuration, and governance is outlined to ensure policy compliance and operational efficiency. By enhancing performance monitoring and proactive issue resolution, this work provides a robust, data-driven foundation for modern IT infrastructure observability.

Contents

List of Figures

Chapter 1

Context

1.1 Introduction

Anything we use in our everyday life needs maintenance: our cars, smartphones and even knives. But when it comes to fixing an issue, we need to know what has to be fixed. There's a story of a computer technician who after billing 1000$ a customer to tighten a bolt was asked to justify this high price for such an easy task and he sent the bill with two voices: *hardware support: 1$, knowing what to do: 999$*.

Of course, that's just a story used to explain the importance of expertise, but its moral is strictly connected with the main arguments of this thesis. In fact, today's software, web apps, and any working piece of code can be affected by many different problems: bad implementation, unsupported edge cases, memory overload, bad resource optimization and so on. When it comes to troubleshooting a small to medium-sized project, somehow it can be done manually, just by reproducing the case and debugging the code, maybe in some very small projects the developer can just go through everything line by line.

But what happens when we deal with bigger projects? When an error cannot be reproduced? Sometimes we need to search for an edge case that wasn't expected and handled and we need to act fast because this issue is causing huge inefficiencies and millions of dollars lost every hour. Last year, a production bug of CrowdStrike's platform Falcon (a cloud-based AI-powered endpoint protection) caused approximately 10 billion dollars lost.[Nev24; Wik24] How the bug raised and where was the error located are out of the scope of this thesis, but considering Microsoft is used worldwide, both by private users and companies, the impact of the kernel NULL pointer error released on July 19th 2024 is maybe the biggest of IT infrastructure ever. Bug fix had to

deal with both a huge codebase and a worldwide impact in air transport, finance, healthcare etc. CrowdStrike reverted the update in a couple of hours and detected the problem being some errors in code and more of them in testing.[Cro24]
This is the scenario where **monitoring and observability** prove their value: continuously watching our software in order to take us straight to the point where the error happened if that's the case. Or maybe helping understand how sales are going, how the funnel works and many other useful information: at which time our website has the most viewers? Did people actually check the promo page? What is the average and most usage of our bandwidth and memory?

1.2 Know your system

Monitoring and observability are strictly related and still very different from each other. Monitoring is the capacity to keep an eye on something developers know might happen, they just don't know when it will. Observability is the capacity to have the whole system tracked, also where developers don't even imagine a bug can occur.
Monitoring is typically described as the correct usage of it's three pillars: *metrics, logs and traces*. These three together gave the opportunity to read a system state at every moment and catch any possible bug in real-time or close to real-time. **Observability**, on the other hand, is based on the ability to read the *state of the entire system* and identify any anomaly. «It allows teams to ask fact-finding questions about their data, pursue leads, and generally explore anything they need to clarify their understanding during or after an incident. An observable system allows engineers to look past predefined monitoring alerts and dive deep into all areas of the system to pursue answers they had never thought of before.»[GFL24, pp. 50-51]

1.3 The project

1.3.1 Moviri

This project was supported by Moviri, an international IT consulting company based in Milan. Founded in 2000, Moviri originated as a research spinoff from the Politecnico of Milan and was later incubated at the university's startup accelerator. As a data-driven company, it is firmly commit-

ted to R&D and maintains strong ties with academia—reflected in its subsidiary offices in Padua and Boston, both located in cities renowned for their prestigious universities. Moviri primarily operates in three areas: performance engineering, analytics, and cyber-security, offering both consulting services and dedicated products in each field.
This thesis was developed within the Performance Engineering service line, where I am currently employed. The primary objectives of this unit are performance testing, application performance monitoring, and the optimization and automation of mission-critical applications. A critical aspect of this work is the ability to extract data from systems under test and monitoring. This is a rapidly growing market, with new tools continually emerging and fresh challenges arising from customers. As more systems transition to cloud-based environments and adopt microservices architectures managed with Kubernetes and other tools, Moviri continues to innovate by exploring new opportunities and integrating them according to their relevance to ongoing projects.
The project itself embodies this spirit of continuous innovation, aiming to assess current market offerings—with a particular focus on the open source domain—and determine whether they can be leveraged to enhance service delivery. The technologies, tools, architectures, and data strategies employed in this project reflect a commitment to advancing both knowledge and performance, staying up to date with market trends, and honoring the academic curiosity that originally gave rise to Moviri.

1.3.2 Project aim

The objective of this thesis is the design and prototyping of an integrated Enterprise-level **Observability solution** based on Open-Source frameworks and technologies, together with a comparison of performance and features between different tools. The proposed solution aims to meet the monitoring and analysis needs of modern IT infrastructures, ensuring scalability, advanced data correlation, entity relationship management, and accurate metrics evaluation
The main aspects considered in the project include:

- **Scalability**: the solution must be able to grow flexibly to accommodate infrastructures of different sizes, ensuring optimal performance even in contexts with varying workloads;
- **Standardization**: adoption of standard frameworks and protocols to facilitate

interoperability between different components, also with a view to extension and integration with other solutions, both Open-Source and market-based;

- **Data correlation**: implementation of mechanisms for aggregation and analysis of data from different sources to reduce noise and produce a complete and integrated view of the system;
- **Relationships among Entities**: management and visualization of relationships between various monitored entities, facilitating the identification of dependencies and analysis of the impact of any anomalies;
- **Metrics Evaluation and Customization**: evaluation of metrics available out of the box and the ability to define custom metrics to meet specific business needs;
- **Data Intelligence**: development of advanced artificial intelligence capabilities for automatic problem diagnosis, baseline definition, and intelligent alert generation;
- **Installation, Activation, Configuration, and Governance**: definition of a structured process for installing, activating, and configuring the solution, as well as governance practices to ensure effective monitoring that complies with corporate policies.

1.3.3 Thesis structure

This thesis will firstly go through a description of the context: state of the art and definition of monitoring and observability. Then it will present all the tools that have been tested in this project and the architectures used to deploy web applications, together with a brief explication of their structure and modules.
After this introduction part, will be presented all the different instrumentations, the design choices made and the different data pipeline used. In this part, will also be presented some results, showing what data visualization can do and how these data can be used.
Lastly some comparison between different approaches will be made and some possible future horizons of the thesis.

Chapter 2

Background

This chapter will go through some key concepts project-related. The first thing that will be presented is the **architecture** of a system, using two of the most famous application architectures in IT history. Then, it will go deeper into **observability and monitoring** approaches, in order to have a better focus on the main topic of this thesis.

2.1 Architectures

A crucial part of monitoring and observability is to know the architecture of the system. Imagine a simple task like changing the oil in a car: the overall architecture of a car is essential in order to avoid putting oil in the fuel reservoir. The same goes for software.

The architecture of an application is deeply related to its purpose and how it's developed. An application fully developed by a single person or a team of a few people will easily be designed with regard to continuous meetups between developers, so that any change or decision can be made at any point. When it comes to bigger work teams, even located in different countries or continents, standardization, compartmentalization and versioning becomes crucial. Still, architecture it's a **non-functional requirement**, since any use case can be built on top of every architecture and any architecture can serve the purpose with a good modularity implementation and comprehensive automated tests.

In this thesis we will discuss only about web applications and how observability works for them. For this specific topic, monolithic and microservice architectures are the most used architectures.

2.1.1 Monolithic architecture

> Like many other aging enterprise applications, the FTGO [*Food to Go*] application is a monolith, consisting of a single Java Web Application Archive (WAR) file. Over the years, it has become a large, complex application. Despite the best efforts of the FTGO development team, it's become an example of the Big Ball of Mud pattern. [Ric18, ch 1]

This is a part of a small story working as an introduction that shows why monolithic architecture, although it has been a solid architecture for decades, is no longer always the best one, since nowadays it can easily become a *Big Ball of Mud* [Kos00] [RF06, p. 120]. This architecture is well represented in its own name: a single, comprehensive, «single deployment unit of all code» [RF06, p. 123]. Richards & Ford divides this architecture style into three different styles:

- Layered architecture
- Pipeline architecture
- Microkernel architecture

We will go through just the first one, the most commonly used: layered architecture, also called *n-tiered architecture*. This is maybe the most immediate and easy way to implement an architecture: each level (tier) is strictly connected with the one above and the one below, since it receives requests from upper levels and asks for resources to lower levels. In this image there are three different **possible implementations** of a monolithic application, each one with a different isolation of certain levels. Each layer corresponds to an abstraction of a task or a group of tasks and satisfies a specific business request. For example, the presentation layer needs to communicate with the final user and present what the application is capable of doing; or the database layer is responsible for reading and writing data from and to a database. Each with its own concerns, often this topology tends to reproduce the same structure of the company where the application is designed (as stated in Conway's law [Wik25]).
The peculiarity of this architecture is being all deployed in a **single executable** or WAR file, which means it's easier to deploy, but every minor change affects the entire application: if the FTGO developers have to change some backend logic, also the presentation layer (*aka* the frontend) will be affected by the deployment and users will

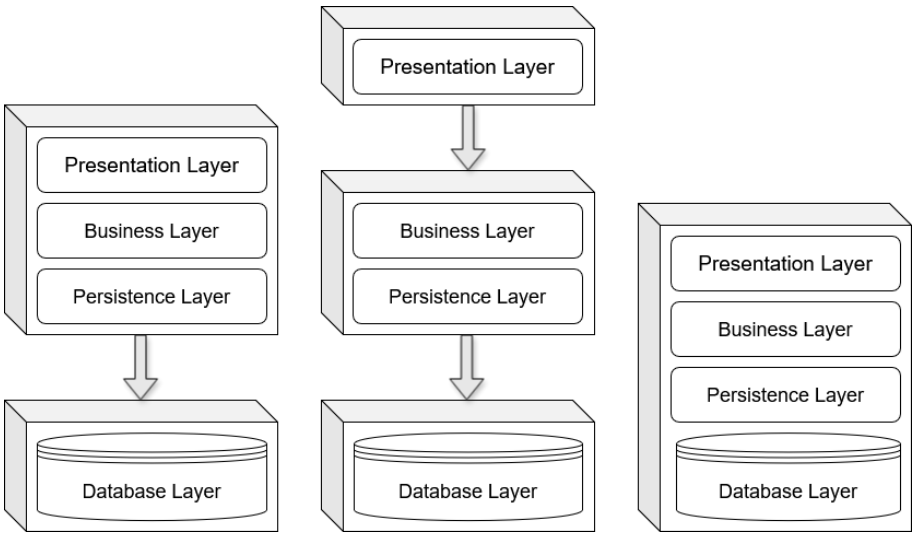


Figure 2.1: Physical topology (deployment) variant.

Source: Cfr. [RF06, p. 134]

not have access to it for a certain amount of time. Furthermore, the deployment of a huge app could take several times and cause many problems or crashing during the deployment. Testing each new version is also time-consuming, since all the application has to be tested and not just a layer or a single service.

This style also has issues when multiple developers teams are working with a different part of the code: **versioning** could be painful if the app is deployed when one of the teams is ready but others aren't. These huge apps also have another practical inconvenience for a developer: they slow down a developer's IDE.

There is also another problem with **scalability**: FTGO app has huge traffic loads in the presentation layer but almost zero in the persistence layer, it would be useful to duplicate just the presentation layer on multiple servers, but it isn't possible, the whole app has to be duplicated and this is very inconvenient on multiple aspects, like expenses and efficiency.

Since every layer is strictly connected with others and the deployment has already mentioned problems, it is very risky to adopt new technologies for just some parts or too expensive to rewrite everything in a newer technology. Similarly, these applications are not fault-isolated: a **bug** can occur in a single module and still cause other modules to crash. It's harder to track exactly where it happened and bug fixes are always urgent, since all modules could be affected by a single module bug.

Of course, this architecture is not still on many applications only for legacy reasons, but it also has some major benefits:

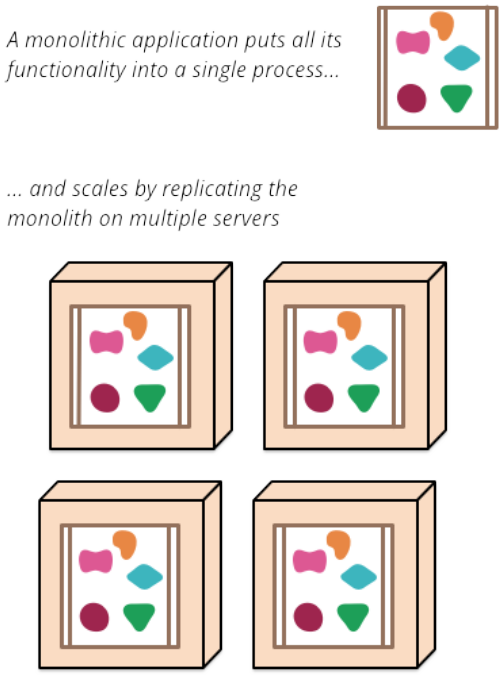


Figure 2.2: Monolithic application scalability.

Source: [FL14]

- Simple to develop: IDEs and other developer tools are focused on building a single application;
- Easy radical changes: changing the code and the database schema is easy to deploy;
- Straightforward to test: although tests are quite slow, developers can write end-to-end tests that work on all layers at the same runs;
- Straightforward to deploy: all a developer has to do is copy the WAR file to a server that has Tomcat installed, and the same goes for scalability.

2.1.2 Microservice architecture

Conversely, microservice architecture focuses on single components, defined as «*a unit of software that is independently replaceable and upgradeable*» [FL14]. The very first perks of this paradigm are shown in a **possible definition** («*Loosely coupled service oriented architecture with bounded contexts*» [Sho16, slide 24]) and in its **scalability** (the «*ability for the system to perform and operate as the number of users or requests increases*» [RF06, p. 58]).

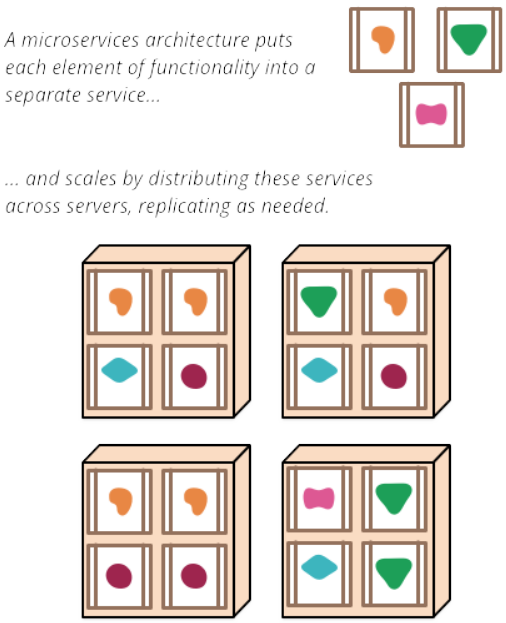


Figure 2.3: Microservice application.

Source: [FL14]

A definition

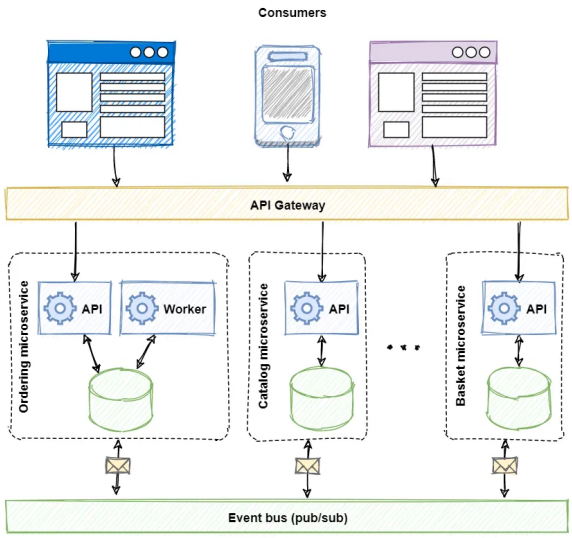


Figure 2.4: The topology of the microservices architecture style.

Source: [AI-25]

Each service represented in this scheme is an **autonomous part of the application** that has its own logic, business and functions, can be updated without the necessity to modify other services and doesn't need to know what others do. Both *loosely coupled* and *bounded contexts* traits are related to this concept of isolation: if you cannot update a

service without updating others, then it's not loosely coupled. The same goes for the context: if it needs to know to much about other services, it doesn't have its bounded context. Of course, as engineers, we all know "too much" it's not a proper definition, but a better definition is yet to be found and right now it's easier to describe it with three categories: **purpose**, **transactions** and **choreography**[cfr. RF06, p. 248]. These categories mean that each service should be functionally cohesive, doing just one task and doing it in full (loosely coupled); reducing communication at minimum, since communication in distributed systems can often cause bugs and various issues (bounded context) and when different services need extensive communication the architect may consider the hypothesis of bounding these services in one larger service (choreography).
As consequences of this architecture, microservices usually have multiple databases and an API layer to communicate. The **data isolation** relies on the same concept of the services themselves: avoid strict coupling between different concepts. The API layer, lastly, is nowadays the most used paradigm to communicate with each service (and mostly also between services).

Scale cube

Another analysis of microservices is offered by [AF10], where the three dimensions of a cube represent three specific features of an architecture, later summarized in a blog in their website[Par25].

X-axis: duplication The X-axis is the most immediate way of scaling, both a monolith and a microservice application: duplicate instances and let load balancer manage requests flow to N instances of the application.

Y-axis: functional decomposing The Y-axis is what really makes the difference between a monolith and a microservice: if there's no functional decomposing (everything is done in the same block of code) it's a monolithic approach, otherwise we can split the application in more and more different services, defining the isolation rule based on each case.

Z-axis: data partitioning The Z-axis, similarly to the X-axis, is about partitioning data, instead of instances. Each instance becomes responsible for a specific subset of the dataset.

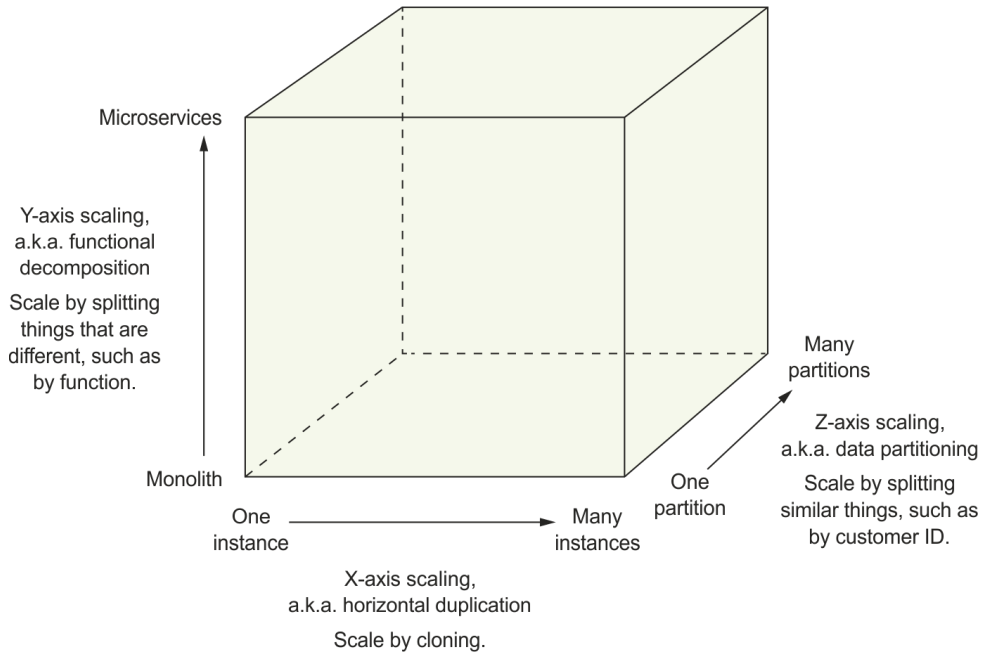


Figure 2.5: The scale cube axis.

Source: [Par25]

2.1.3 Pros and cons

As for every architecture, microservice it's not the right choice for every application, but it has some pros and cons that have to be considered before the design of the application.

On the **pros** side there are multiple aspects related to new technologies and the delivery of a product, such as it enables the continuous delivery and deployment of large, complex applications, services are small and easily maintained, independently deployable and independently scalable. On the developer's side, the microservice architecture enables teams to be autonomous and it has better fault isolation, so it allows experimenting and adoption of new technologies.

On the **cons** side, there are some difficulties due to architectural complexity: finding the right set of services could be challenging; distributed systems are complex, which makes development, testing, and deployment difficult; deploying features that span multiple services requires careful coordination; deciding when to adopt the microservice architecture is difficult itself.

2.2 Delivering

The ability to quickly go to the exact point of a certain problem, bug or feature is not only related to troubleshooting, but also to the actual product delivery paradigms. A **software process model** is a simplified version of a software process, like a general idea functioning as a baseline for a more specific one. In many years of software development, many have been invented and described, but as for architectures, we will go through only a couple of delivering model, maybe the most used and more related to monolithic and microservice architectures just presented.

2.2.1 Waterfall

> This [model] takes the fundamental process activities of specification, development, validation, and evolution and represents them as separate process phases such as requirements specification, software design, implementation, and testing[Som16, p. 45].

The high-level idea behind the waterfall process is to collect all requirements, use cases and possible errors during the first phase of software development, create solid documentation and release the final product, with every phase of the processes directly **dependent by the previous** (this cascade of phases is the origin of its name). In principle all

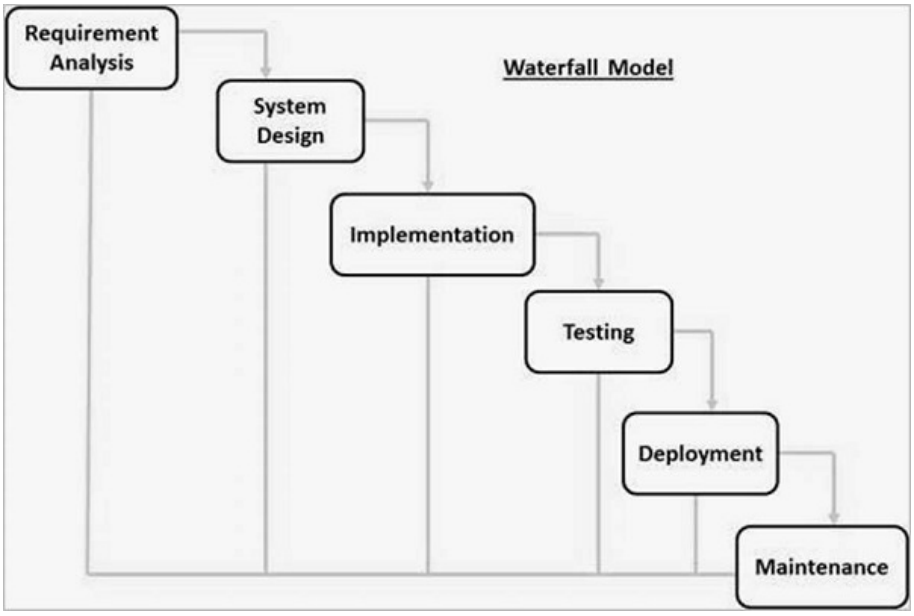


Figure 2.6: Waterfall model.

Source: Tutorials Point: www.tutorialspoint.com/sdlc/sdlc_waterfall_model.htm

requirements are written on a document and approved, only after the sign-off of the document the software is designed. When designing a software, many details come out and requires changes in requirements, that has to be modified and approved again. This strict protocol can create a potentially infinite loop of continuous documentation changes, with possible major delays in product delivery, since only when requirements meet the design and both client and developers agree on them, the real software implementation and testing starts.
This process model is still a good choice when the software is a critical system with high-security standards that cannot be tested without a complete analysis and also with embedded software that needs to communicate with hardware, due to the inflexibility of hardware, every aspect of the analysis cannot be delayed after the first delivery of the product.

2.2.2 CI/CD

> The continuous integration/continuous delivery (CI/CD) pipeline is an automated DevOps workflow that streamlines the software delivery process[SS24].

In this model developers can focus on developing the software and receive **continuous feedback from customers** in a circular pattern where the application is automatically tested and deployed each time. Each developer team works on their branch of code (continuous integration) and they don't need to push it to the master all in one, but every feature can be pushed, tested and deployed alone (continuous delivery), with other teams possibility of getting the new snapshot of the code with a versioning tool like Git. This means that the customer can evaluate the system at every stage of the development to see if it delivers match requirements. CI/CD has three major advantages over the waterfall model:

- the amount (together with costs) of analysis and documentation is significantly less;
- it is easier to get customer feedback and show progress;
- early delivery and deployment of useful software to the customer is possible, at list with bare-bones functionalities.

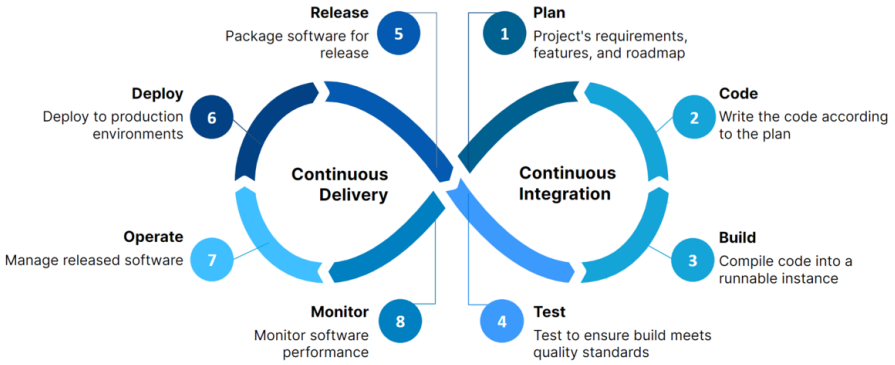


Figure 2.7: CI/CD pipeline.

Source: [Aru24]

CI/CD also has some downsides, such as the lack of documentation or the messy code. Both are not part of the model, but when the application changes quickly it becomes cost-effective maintaining an updated documentation and becomes easier and less time-consuming adding some throwaway code[1], which will easily become spaghetti code. These problems are more effective when the application is huge and a lot of teams work on it, when the software is long-lifetime systems. Large systems need a stable framework or architecture, it cannot be developed incrementally.

2.3 Application state

> Observability and monitoring are often mentioned in the same breath, but they are actually distinct tools and methodologies that can be boiled down to investigating "known" versus "unknown" unknowns[GFL24, p. 51].

A developer team dealing with e-commerce knows there are many edge cases and possible malicious users: every time a new promo is released a hacker can find a path to abuse of it and get free items instead of a discount. This case (and similar) doesn't raise an error, nor a warning and most likely users won't complain about it! But still they need to be captured and fixed, not to mention getting as much information as possible about the hacker. This is the "known unknown": it's predictable someone will try abusing a

[1] «*THROWAWAY CODE is quick-and-dirty code that was intended to be used only once and then discarded. However, such code often takes on a life of its own, despite casual structure and poor or non-existent documentation. It works, so why fix it? [...] Over time, a simple throwaway program begets a BIG BALL OF MUD.*» [Kos00]

new release to find a way to abuse of the application, but it's still unknown how and if it will be possible.
On the other hand, when a good team develops a new feature, all tests are successful and deploys it, the expected result is a fully working application with no bugs whatsoever. In this scenario we are working with a good team and a good team is not naive and knows that «*There's always one more bug*» [This is known as the "Lubarsky's Law of Cybernetic Entomology", a humorous law well documented inside the developers' word. Wal25], it's just they still don't know where it will happen and what consequences it will have. This is the "unknown unknown": something neither predictable nor expected.

2.3.1 Monitoring

Monitoring involves **checking values for predefined thresholds**, creating alerts on possible errors that may occur and runbooks, and so forth. Typically, monitoring is composed of three parts, known as the *three pillars of monitoring*: metrics, logs and traces.

- **log** are structured and unstructured lines of text that a system generates when a specific part of code executes. In general, a log is a record of an event within an application[Usm+22, p. 86913];
- **trace**: (1) A record of the execution of a computer program, showing the sequence of instructions executed, the names and values of variables, or both. Types include execution trace, retrospective trace, subroutine trace, symbolic trace, variable trace. (2) To produce a record as in (1). (3) To establish a relationship between two or more products of the development process; for example, to establish the relationship between a given requirement and the design element that implements that requirement[90, p. 77];
- **metric**: A quantitative measure of the degree to which a system, component, or process possesses a given attribute. See also: quality metric[90, p. 47];

2.3.2 Observability

Observability works by instrumenting your code and **capturing the right level of detail** that lets you answer questions and understand the current state of your system. It's important to underline that, although they are often cited as synonyms, they are actually different things and still not alternatives. Observability's role is not to replace monitoring

systems and neither to be a monitoring system on steroids; observability relies on the same three pillars of monitoring, but use them with a different purpose and context: they are combined to provide four Site Reliability Engineering (SRE) golden signals of observability (latency, traffic, errors, and saturation). **Latency** (request service time) tracks the time taken by an application to process a certain request; **traffic** (user demand) tracks the number of requests received from and send by an application per unit of time; **errors** (rate of failed requests) tracks the percentage (or any other rate) of failed requests in a system, from 404 Not found to response 200 with no real response; **saturation** (overall system capacity) measures how much resources like memory or CPU are being loaded. These are focused on run-time performance and helps alerting, troubleshooting and tuning and capacity planning. An observability system, on top of data exposure, uses then *correlation, topology and incident response* to perform its duties. Both

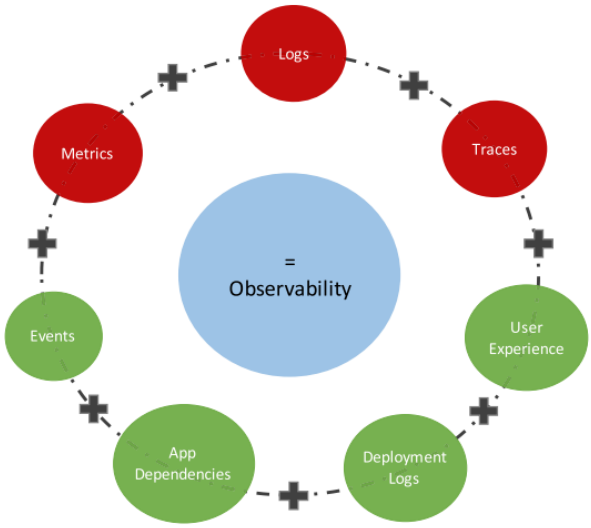


Figure 2.8: Three fundamental pillars of observability (red) plus other data considered vital for observability by different vendors (green).

Source: [Usm+22, p. 86913]

monitoring and observability also allows using data visualization for an easier and faster exploration. Having access to system state via graphs, tables and similar visual tools allows DevOps to operate faster and track all system states with a more proficient, human oriented approach.

2.3.3 Pros and cons

Observability is defined as "*the ability to measure the internal state of a system only by its external outputs*", i.e. the three pillars previously discussed, and provides high-level overviews of the system's health and granular insights into implicit failures. Monitoring, on the other hand, focus on keeping track of a system's health with a predefined set of metrics and logs (i.e., we are looking for a specific collection of failures).
However, in a distributed system, many changes are dynamic, also happening in parallel: we could miss some issues that do not fall into any of the categories of warnings or errors we were looking for. To implement observability, monitoring is a prerequisite.
Monitoring alerts operators about operational failures, whereas observability assists in determining where and why the failure occurred and what triggered the issue. Speaking

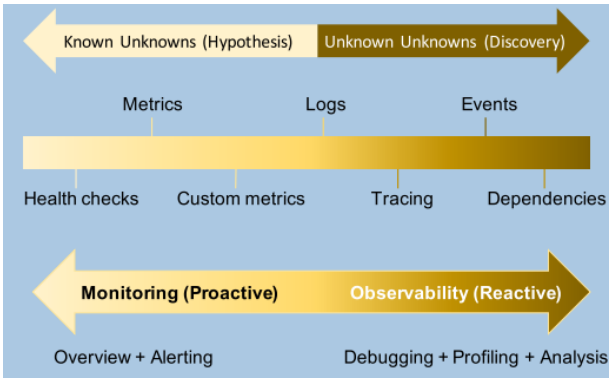


Figure 2.9: Observability vs monitoring. Monitoring is about testing hypotheses, and observability is exploring new discoveries.

Source: [Usm+22, p. 86907]

directly of pros and cons, monitoring has on its side an easier instrumentation: a monitoring system can be implemented in a system with much less effort than an observability system. Furthermore, it's also easier to use, it requests to identify the *known unknown* and monitor it with the three pillars. It has been a valid solution for decades on many different architectures. In a monolithic architecture, an application is frequently in one of two states (up or down) that directly impact customer experience. In microservices, the interconnections of components it's becoming more complex any year, with an impressive growth of the interdependence between components. The root cause of an error in one component is often originated from a state alteration of one or more other components. A specialized, component-specific monitoring context would produce only local information, connect them and walk to the root cause becomes a developer

job, that often has insufficient data to pinpoint the cause of a problem.
On the other hand, observability is for sure harder to instrument and asks a deeper knowledge of the system architecture. This sometimes makes it hard for a company to accept that monitoring could be no longer enough to track system's state, since this "new" approach also requires an increase in effort in development and architecture engineering. But this solution helps to track errors also in more complicated architecture without leaving to developers the arduous task to connect logs from different services. Last but not least, observability is useful mostly when speaking of the *unknown unknown*, preventing to discover issues and bugs only after they made user experience worst enough to report them.
Both monitoring and observability, lastly, needs to grow together with the application. They are not static solutions, as the system is usually not static but changes over time, it has a life cycle, which means a regular update of the software (sometimes the hardware) that will change the system behavior. Another downside, that will not be covered in this thesis, is related to data storage: monitoring and observability systems uses also sensitive data for their purpose: security and integrity of the data are fundamental. They must be stored safely in the system, and when transferred outside, the communication channel has to be secured.

Chapter 3

Tools

In this chapter we will go through technologies and tools used in this project. The common feature of all these tools is being open-source, while most of them have also some enterprise options company oriented. Presented tools are mostly in three categories: telemetry solution, data visualization and application deployment. For the **data visualization** Grafana is at the moment one of the most used product world-wide, since it offers an all-in-one solution for observability that can also be unpacked and be integrated with different tools. Other two tools has been explored in their domain in comparison with the market leader alterative.
For the **telemetry** generation and collection Grafana is still an alternative in its all-in-one package, but this thesis's most analyzed tool is OpenTelemetry, with both no-code agent and its collector.
On the **application deployment** side, we will use as local deployment tools Tomcat together with services deployment and management tools such as Docker and Kubernetes.

3.1 OpenTelemetry

OpenTelemetry (OTel) website documentation reads[1], on what OTel is:

- observability framework and toolkit designed to create and manage telemetry data;
- vendor and tool-agnostic;
- focused on the generation, collection, management, and export of telemetry.

[1] Full documentation at opentelemetry.io/docs/what-is-opentelemetry/.

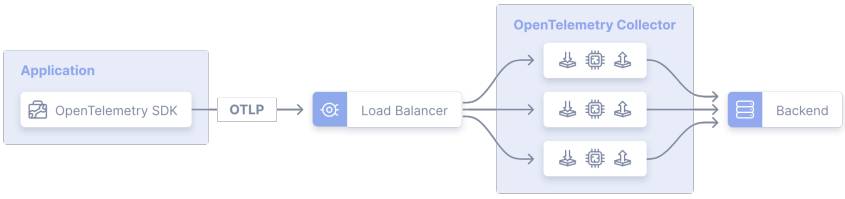


Figure 3.1: OpenTelemetry demo architecture.

Source: OTel demo tutorial: opentelemetry.io/docs/demo/collector-data-flow-dashboard/

OpenTelemetry is an open-source observability framework that provides tools, APIs, and SDKs to collect, process, and export telemetry data such as traces, metrics, and logs. OpenTelemetry has quickly become a standard for monitoring cloud-native applications and microservices, offering unified instrumentation and data collection capabilities and more and more data visualization tools are accepting OTel standard telemetry, making easier for developers to change back end tools without instrumenting a new telemetry exporter in their system. OpenTelemetry extensibility and interoperability are possible thanks to its adhering to open standards such as OpenTracing and OpenMetrics.

3.1.1 OTel collector

An important advantage of OTel it's its vendor-agnostic collector: a way to receive, process and export telemetries without sticking to a specific back end. Easy, small applications can be instrumented to directly send data to designated back end, but when it comes to complex applications with different services and rich architecture, the collector it's the best way to receive all their data, doing some pre-processing and send to one or more vendor.

Although it is possible to install the collector preconfigured with standard receiver, processors and exporter, it's highly configurable with *.yaml* file. A minimum collector configuration needs receivers and exporters, but typically it also has some data processing in the pipeline, as shown above. The OpenTelemetry collector can be installed and deployed in many different ways: it can be 'run in a Docker container, as well as deployed as daemonset and a single gateway instance in a Kubernetes or installed in any operating system. Each of these options has its own pros and cons. Containerized options has more flexibility and are easier to run in a system already running in containers, on top of being prone to updates and only requires the image and the

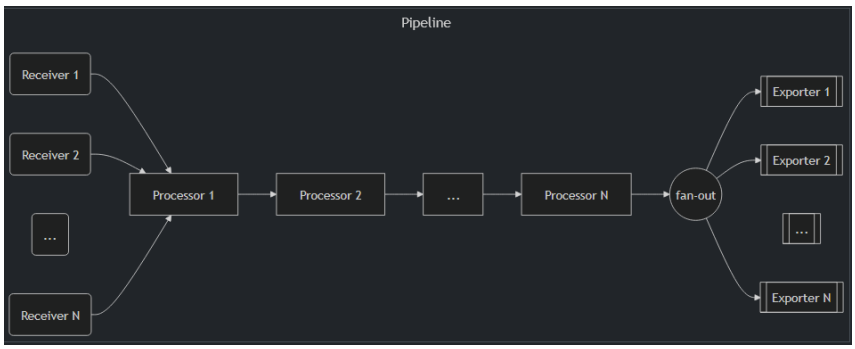


Figure 3.2: OpenTelemetry collector pipeline.

Source: OpenTelemetry architecture documentation: opentelemetry.io/docs/collector/architecture.

configuration file. Installing the collector in a system is probably more suitable for applications running on a hardware machine, with on premise servers or in general for apps deployed in a single host.

3.1.2 OTel agents

Although Open Telemetry has its own SDK and APIs to instrument any code observability at its best, these aren't the most suitable option in many different scenarios. Open Telemetry also offers the option to auto-instrument observability by injecting its agent inside the application[2]. It works with most of the most used languages like Python, Go, Java, .NET etc and automatically detects packages, libraries, request flows and metrics and sends them to the collector thank to environment variables. It's highly configurable both in stand-alone applications and in Kubernetes clusters.

3.2 Grafana

«Grafana open source software enables you to query, visualize, alert on, and explore your metrics, logs, and traces wherever they are stored. Grafana OSS provides you with tools to turn your time-series database (TSDB) data into insightful graphs and visualizations. The Grafana OSS plugin framework also enables you to connect other data sources like NoSQL/SQL databases, ticketing tools like Jira or ServiceNow, and CI/CD tooling like GitLab.» [3]

Grafana is an open-source observability and data visualization platform with a wide set

[2]For further information and supported languages: opentelemetry.io/docs/zero-code/.

[3]Check Grafana official documentation: grafana.com/docs/grafana/latest/.

of tools that covers monitoring and observability at 360 degrees, from instrumenting a system with Prometheus[4], by collecting metrics with Grafana Alloy to data visualization with Grafana Cloud. Originally launched as a tool for visualizing time-series data, it has

Figure 3.3: All Grafana's tools.

Source: Grafana websitegrafana.com

evolved into a powerful multipurpose solution that connects with a wide range of data sources and supports advanced analytics capabilities. At its core, Grafana enables users to create dynamic, customizable dashboards that provide real-time insights into system metrics, application performance, business data, and much more. One of Grafana's

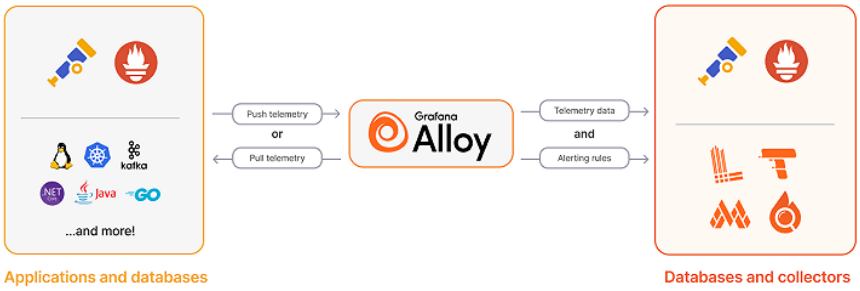


Figure 3.4: An example of Grafana pipeline.

Source: Grafana website - Introduction: grafana.com/docs/alloy/latest/introduction

features is its ability to integrate with numerous data sources, including Prometheus, Elasticsearch, InfluxDB, MySQL, PostgreSQL, AWS CloudWatch, and many others. The platform also supports alerting, enabling users to set thresholds and notifications for critical events or anomalies. Grafana fosters collaboration through its shared dashboards and annotations, making it easy for teams to work together on troubleshooting or performance optimization tasks.

[4]Prometheus is an open-source monitoring and alerting toolkit designed for collecting and analyzing metrics from applications and infrastructure.

3.3 Other data visualization tools

SigNoz

SigNoz is an open-source backend observability tool developed on top of Open Telemetry. As with other backends, it allows visualizing data, troubleshoot the root cause of a bug, send alerts based on threshold on metrics and much more. It has two major advantages: the first is its being built on top of Open Telemetry, which keeps the whole code vendor-free; the second advantage is the possibility of running a Docker stand-alone image in a local machine, together with the opportunity of using a cloud version with full features and support.[5]

OtelBin

Dash0 -a company producer of an observability tool with the same name, unfortunately without a free tier plan- offers an open source Open Telemetry Collector validator and flow visualization: OTelBin. This tool allows the validation of many different versions of the OTel collector and also helps developers to visualize their collector flow: receivers, processors and exporters, all in a human-readable flow. [6]

New Relic

New Relic is a comprehensive observability platform that enables developers and IT operations teams to monitor, troubleshoot, and optimize the performance of their applications and infrastructure. By unifying telemetry data—such as metrics, events, logs, and traces—into a single platform, New Relic provides deep, AI-powered insights that help identify and resolve errors and security issues more efficiently. This unified approach facilitates a clearer understanding of system behaviors, allowing for proactive performance enhancements. For more details, visit the original documentation.[7]

Lumigo

[5] Further details and possible usages at official GitHub page: github.com/SigNoz/signoz.
[6] Official website: www.otelbin.io.
[7] Official website: docs.newrelic.com/

Lumigo is an AI-driven observability platform designed to offer full visibility into cloud-native applications by correlating traces, logs, and metrics. A standout feature of Lumigo is its AI Copilot, which assists developers in troubleshooting microservices more effectively. The Copilot provides real-time synchronization between agents and applications, enabling swift identification and resolution of performance bottlenecks. This intelligent assistant enhances the developer experience by streamlining the debugging process and reducing downtime. For more details, visit the original documentation.[8]

3.4 Kubernetes

Kubernetes (k8s) is a portable, extensible, open source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. Originally designed by Google that made it open-source in 2014, the project is now maintained by contributors worldwide. It takes care of scaling and failover for your application, provides deployment patterns, and more. Kubernetes official documentation list these as most important aspect of k8s, serving three main aspects like Resource management, scheduling and service Management:

- Service discovery and load balancing;
- Storage orchestration;
- Automated rollouts and rollbacks;
- Automatic bin packing;
- Self-healing;
- Secret and configuration management;
- Batch execution;
- Horizontal scaling;
- IPv4/IPv6 dual-stack.

[8]Official website: docs.lumigo.io/

The container orchestration offered by k8s allows treating a set of machines running Docker as a pool of resources. It's one step ahead of deployments made in VMs. Shortly,

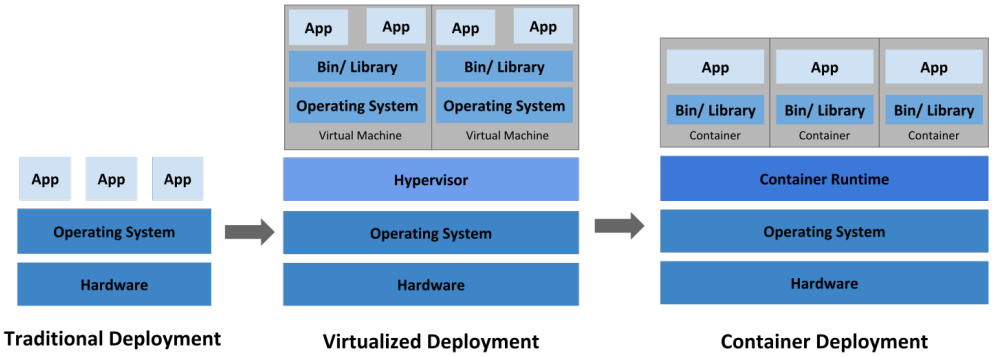


Figure 3.5: The three steps of deployment.

Source: Kubernetes doc overview: kubernetes.io/docs/concepts/overview/

in the first steps of deployment, everything was deployed in hardware server without resource optimization, where one component could have used most of the resources. VMs allowed to limit resources without using multiple hardware servers. Containers are similar to VMs, but they have relaxed isolation properties to share the Operating System (OS) among the applications.

For this project **k3d**[9] has been used as k3s wrapper to run in Docker. In other words, it's on of many tools to manage k8s cluster and interact with each node.

On next paragraps we will present two powerful tools in the Kubernetes word: kubectl and Kubernets dashboard. The first is a command that allows to manage, install and delete pods, to monitor the state of the cluster and to work almost on any front in a k8s application. Kubernetes dashboard is a UI that allows to do monitor many metrics of an application and to apply some changes to an application.

3.4.1 kubectl

`kubectl` is the primary command-line interface used to interact with a Kubernetes cluster. It allows users to manage resources, perform operations on distributed applications, and monitor the status of components within the cluster. With its straightforward and powerful syntax, `kubectl` is a crucial tool for system administrators and developers working in Kubernetes environments.

[9] K3d official website: k3d.io/stable/.

One of the most common operations with `kubectl` is retrieving information about cluster resources. For example, to get a list of active pods, the following command can be run:

```
kubectl get pods -A
```

This command provides an overview of the currently running pods in the cluster, showing their status and other useful details. The `-A` parameter returns all pods in all namespaces (otherwise it would return only pods in the default namespace). Another frequently used command is `kubectl describe`, which offers more detailed information about a specific resource:

```
kubectl describe pod <pod-name>
```

In addition to managing resources like pods, `kubectl` supports the use of configuration files to define and deploy resources, such as when using `kustomize`. `kustomize` is a tool built into `kubectl` that allows users to customize Kubernetes configurations declaratively. It provides a way to manage YAML files for Kubernetes resources, enabling easy modifications for different environments without the need for manual editing of configuration files.
To apply a `kustomize` configuration, the following command is used:

```
kubectl apply -k <path-to-kustomization-directory>
```

In this case, the `-k` flag tells `kubectl` to use the resources defined within the specified directory that contains a `kustomization.yaml` file. This file defines how Kubernetes resources are customized, such as applying patches, adding labels, or configuring namespaces, making it easier to manage configurations across different environments or stages of deployment.
Another key feature of `kubectl` is its ability to perform rolling updates and manage deployments declaratively, although the use of ‘kustomize‘ can also be a part of this process by enabling environment-specific adjustments. Users can define base configurations and create overlays, making `kustomize` a powerful tool for managing Kubernetes resources in a more flexible and reusable manner.

In summary, `kubectl` is an indispensable tool for managing and interacting with Kubernetes clusters. Its integration with 'kustomize' and its support for declarative configuration management make it a versatile and efficient tool for modern cloud-native environments.

3.4.2 Kubernetes dashboard

Among all the k8s tools, implementation, community managed plug-in and everything around it, one particular tool is crucial in this project: the Kubernetes dashboard.[10] It's a UI dashboard that can be installed in a cluster and used to deploy applications, manage resources and many more. For our goal it will be used to monitor cluster resource usage, replicas and errors.
It only supports helm-based installation and the latest image can be installed via command line. Helm is a package manager for Kubernetes, which simplifies the deployment and management of applications on Kubernetes clusters. Helm allows for easier installation, versioning, and management of applications, and provides functionalities like rollback and updates with minimal effort:

```
1 helm repo add kubernetes-dashboard https://kubernetes.github.io/dashboard/
2 helm upgrade --install kubernetes-dashboard kubernetes-dashboard/kubernetes-dashboard
  ↪  --create-namespace --namespace kubernetes-dashboard
3 kubectl -n kubernetes-dashboard port-forward svc/kubernetes-dashboard-kong-proxy
  ↪  8443:443
```

This installation, together with the creation of ad admin role with full access to cluster resource usage allows to monitor the internal state of the cluster.

3.5 Other deployment architectures

Docker

Docker is an open platform for developing, shipping, and running applications. Docker provides the possibility to package and run an application in a loosely isolated environment: the container. The isolation and security lets developers run many containers simultaneously on a given host. Each container already has everything is needed to run the application, without having to rely

[10] This feature is an open-source project: github.com/kubernetes/dashboard.

on package installation and works fine in every machine supporting Docker. With this method, it is possible to abstract software applications from the hardware without sacrificing resources.
Thanks to Docker, writing code and deliver in production are closer, since developers can share container between team members, develop local code, test in container and push the new image to prod container without working on hardware hosts. These features are crucial in CI/CD workflow.

Tomcat

Apache Tomcat is an open source web server developed by a large community together with the Apache Software Foundation that offers a software platform to run Java web app, powering numerous large-scale, mission-critical web applications across a diverse range of industries and organizations, like LinkedIn.
It's quite easy to install and use and allows running Java application locally, like the JPetStore, that we will describe in the next section. For this project I've used Tomcat 9, firstly released in 2016 but still maintained and updated (as I'm writing in February 2025 a new version as been released, Apache Tomcat 9.0.1), supporting Java 8+.

3.6 Web applications

In this section we will go through some details about the two example applications used in this thesis. They are both standard application often used for testing but in the wide offer of similar apps these have a deep difference: one is monolithic and the other uses microservice architecture.

3.6.1 JPet Store

JPetStore 6[11] is a full web application built on top of MyBatis 3, Spring 5 and Stripes. It's an open-source, monolithic, fully working web application widely used for web development's testing and various concepts around JVM. It's prebuild to run on Tomcat or in Docker and has no observability or artificial traffic tools already instrumented.[12]

[11] GitHub repository: https://github.com/mybatis/jpetstore-6
[12] My GitHub repository with configurations: github.com/NonnoPinto/jpetstore-observability.

3.6.2 Online boutique

Online boutique[13] is a cloud-first microservices demo application. The application is a web-based e-commerce app where users can browse items, add them to the cart, and purchase them. It's much more complicated than JPetStore and mostly build in Python and Go. It already has artificial traffic tool instrumented (Locust[14]) and yet maintains its simplicity with it's 11 services written in different languages. Half of the services are already coded with Open Telemetry instrumentation and it presents various possible customizations, developed and maintained by the community.

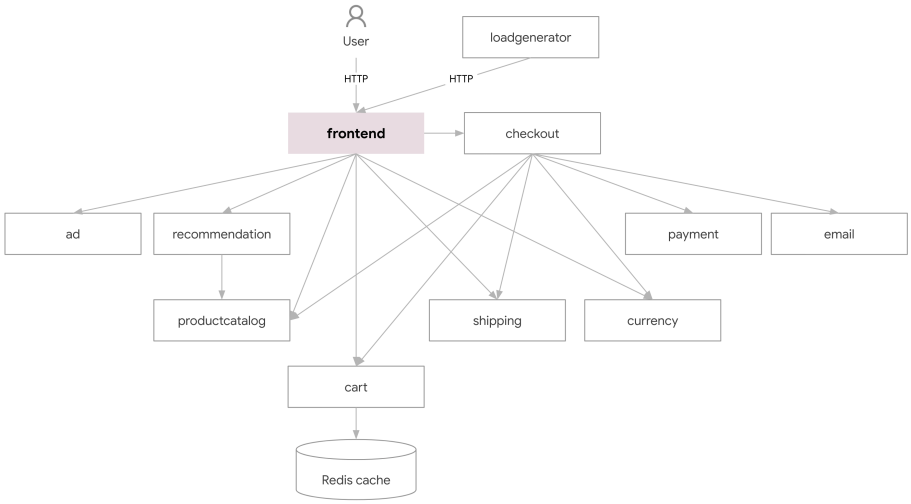


Figure 3.6: Online boutique architecture.

Source: Online Boutique Demo Repository: github.com/GoogleCloudPlatform/microservices-demo

For it's microservices native architecture it has to be deployed with Kubernetes to easily manage each service and their connections.[15].

[13]GitHub repository: https://github.com/GoogleCloudPlatform/microservices-demo.

[14]Full documentation here: docs.locust.io/en/stable/

[15]My GitHub repository with configurations: github.com/NonnoPinto/otel-microservices-demo/tree/main

Chapter 4

Instrumentation

In this chapter, we delve deeper into the thesis of this project. The background information and tools covered in previous chapters serve as the foundation for the project itself, which aims to apply observability theory across multiple complete pipelines. As mentioned, the main goal is to compare different observability approaches and tools and to give a state of the art in observability open source tools. For this achievement, each of the apps has been tested with different tools and deployments: as for application, observability tools can also run locally, on docker or on cloud. We have tested what we find most relevant.

Both applications have been instrumented with different approaches and goals, from a basic approach with a general automatic instrumentation to a metrics-oriented coded instrumentation on specific metrics of interest. Both applications and hardware state were under the umbrella of observability.

All servers and containers has been run in WLS (Windows Linux Subsystem) terminal: Docker only works in Linux OS (except for Docker Desktop, a GUI Docker Management tool that runs a Linux environment behind the scenes) and in general Linux allows an easier a more efficient system for networking (in fact, it's the most used operating system for web servers [W3T22]). For these reason I've decided to work with a Linux environment, in WSL terminal, with Ubuntu distribution.

4.1 OpenTelemetry Java Agent x JPetStore

In this scenario, we used Java **no-code instrumentation** offered by OTel. Basically, a jar file with a Java agent automatically recognizing most of Java packages and

instrumenting OTel telemetries, configured to send data to OTel collector. Since JPet is a standard Java web app, with no custom jar files, Java agent is a perfect fit in this specific context.

The obvious vantage is the near no-code approach, with just some server configuration and collector instrumentation. On the other side, not all packages could be observed or, on the other face of the medal, Java agent instruments observability also in unwanted packages, with an increase in resources consumption.

As for the pipeline shown in the 3.1 the Java agent sends data to OTel collector, operating in a WSL (Windows Subsystem for Linux) terminal and configured to process data with Grafana Cloud configurations and to send telemetries to Grafana Cloud endpoint, together with Dash0 and SigNoz.

4.1.1 Java agent instrumentation

The main part of the instrumentation was, of course, to set the correct Java agent. This requires three steps: download the agent jar file, set environment variables and deploy the app. For the agent I've tested new releases, the last one downloaded and used is v. 2.13.1, released mid-February 2025. Of course any path to store it could be the right one but for simplicity I've just saved it in `./src/lib/` path inside the project. As a result, environment variables are:

```
set JAVA_TOOL_OPTIONS =
↪  "-javaagent:<PATH>\jpet-store\jpetstore-6\src\lib\opentelemetry-javaagent.jar"
set OTEL_SERVICE_NAME = "otel-nocode"
```

While the `OTEL_SERVICE_NAME` gives the name to the service, useful to identify it later after the collector sends the data to one or more backed, the `JAVA_TOOL_OPTIONS` is a crucial key for the instrumentation. Setting these environment variables is the equivalent of launching the application with

```
java -javaagent:<PATH>\jpet-store\jpetstore-6\src\lib\opentelemetry-javaagent.jar
↪  -Dotel.service.name=your-service-name -jar jpetstore.jar
```

but since this application is launched using `Cargo`, setting environment variables was the only option, the only alternative on setting them in the Cargo file (similar to a Makefile). The `javaagent` option indicates where to find the binary to add to the deployment, dynamically injecting bytecode to capture telemetries. It can be used to capture telemetry

data at the "edges" of an app or service, such as inbound requests, outbound HTTP calls, database calls etc. This agent is highly configurable: developers can suppress specific packages, set which type of telemetry to use (for example, a developer could choose to only use metrics, or to combine traces and logs), extend the agent with some personalized options and so on.

The OpenTelemetry Java agent runs inside the same JVM machine running the application and, as any other software agent, needs some resources, known as **agent overhead**. OpenTelemetry declares that this overhead has minimum impact on overall system efficiency, but the final result depends on multiple factors, like physical machine factors or virtualization and containerization. In general, on a small system like the one in test this overhead is minimum and for this project we didn't customized the java agent to fit some resource requirements, but in bigger and more complex system, where the resources can be crucial and the response time determinate the final user experience, some test can be made with minimum java agent instrumentation (for example just the request traces) and with some other fine-tuned instrumentation, in order to fin the best trade off between observability and performance.

4.1.2 OpenTelemetry collector

By default, the java agent sends telemetries to default port endpoints, both in the same network as the application: 4317 for `grpc` and 4318 for `http`, depending on connection protocol of the specific telemetry. Almost everything in the pipeline can be set, also endpoints (we will do this customization in the Online Boutique project to communicate to host machine from inside a container). This endpoint has to correspond to the `receivers` configuration in the OTel collector yaml file. For this project I've used the Linux installation, launching the collector with the `otel-contrib` command with local configuration file and hosted in a single console. Of course the Kubernetes installation wasn't a suitable option since the JPet Store runs in a single docker; same goes for the service installation, less prone to configuration changes and more suitable for stable hosting. The only other valuable option was the Docker image installation, maybe the more immediate and out of the box solution.

Both the Docker image and the Linux command installation were suitable for this use case, the dealbreaker was the access to docker system files. Since OpenTelemetry had to monitor both the app and the resources, having access to Docker resources usage was a

crucial part and accessing them from inside the Docker is much harder than sending to an external collector.
For the deployment, then, it was only about starting the collector with the right configuration and also first validate any new configuration.

- `otelcol-contrib validate --config otel-collector-config.yaml`: validate the collector configuration file. It just checks if all configurations are correctly set, for example if there is an exporter that has no configuration or, more in general, if anything in the pipeline is properly defined. If there's any mistake, the error is returned; nothing is returned otherwise;

- `otelcol-contrib --config otel-collector-config.yaml`: runs the collector with the configuration attached. Some logs are shown about all process starting, more verbosity can be set inside the collector configurations. If there are any error, the collector prints the error message and stops.

As already mentioned, collectors are divided in receivers, exporters and services, although extensions and processors could be configured too. Here we'll break down the collector configuration: I've used the same one for both applications. Here above the full collector pipeline as configured, produced with OtelBin.

Receivers

Receivers are the part dedicated to configure the different host receiving any type of telemetry such as metrics or logs. Of course the most important is the `otlp` receivers, ready to ingest telemetries in OpenTelemetry standard format. But also Prometheus or Zipkin[1], for example, could be configured as receivers, since they have different format and still are widely used. In my collector I've configured, together with the `otlp` receivers, also receivers to collect metrics on the hosting machine like CPU and memory usage and Docker metrics, similar to host machine but on Docker container.

```
receivers:
  otlp:
    protocols:
      grpc:
        endpoint: 0.0.0.0:4317
```

[1] Zipkin is a distributed tracing system that helps track and visualize request flows across microservices to diagnose latency issues.

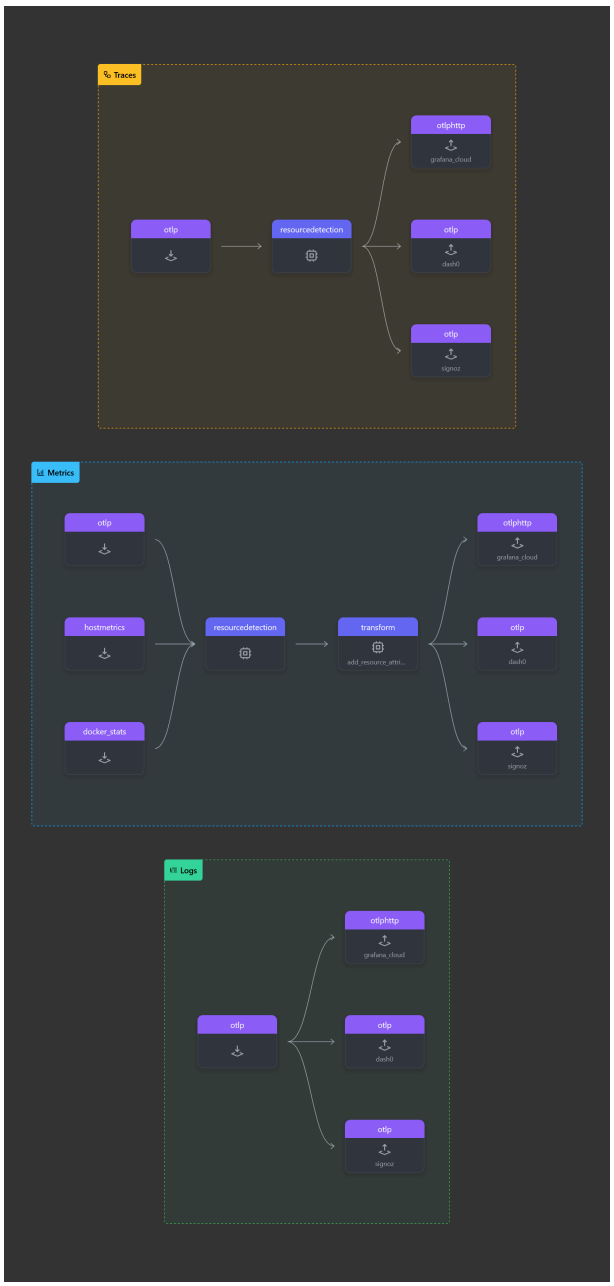


Figure 4.1: Collector pipeline.

```
    http:
      endpoint: 0.0.0.0:4318
hostmetrics:
  scrapers:
    cpu:
    load:
    memory:
docker_stats:
  endpoint: unix:///var/run/docker.sock
  collection_interval: 10s
```

The `otlp` receiver only sets the default port for `grpc` and `http` protocols, referring to `localhost` network. The `hostmetric` receiver role is to scrape metrics from the host

machine: CPU usage, workload and memory usage. Finally, `docker_status` works as the `hostmetric`, but referred to Docker container usage.

Processors

Collector isn't just about receiving and send data, but can also work with data: sometimes an app or the auto instrumentation could send too many data, making it hard to manage them or understand what is useful and what is not. Processors can help in this scope by modifying data, dropping unwanted one and avoid duplicate. Furthermore, most backed solutions with enterprise plans charge companies based on the amount of data ingested (normally in GB): being able to filter those data before the ingestion happens means not just reduce the backed workload, but also means a potentially huge saving in license expending.

```
processors:
  batch:
  resourcedetection:
    detectors: ["env", "system"]
    override: false
    timeout: 2s
  transform/drop_unneeded_resource_attributes:
    error_mode: ignore
    trace_statements:
      - context: resource
        statements:
          - delete_key(attributes, "k8s.pod.start_time")
          - delete_key(attributes, "os.description")
          - delete_key(attributes, "os.type")
          - delete_key(attributes, "process.command_args")
          - delete_key(attributes, "process.executable.path")
          - delete_key(attributes, "process.pid")
          - delete_key(attributes, "process.runtime.description")
          - delete_key(attributes, "process.runtime.name")
          - delete_key(attributes, "process.runtime.version")
    metric_statements:
      - context: resource
        statements:
          - delete_key(attributes, "k8s.pod.start_time")
          - delete_key(attributes, "os.description")
          - delete_key(attributes, "os.type")
          - delete_key(attributes, "process.command_args")
          - delete_key(attributes, "process.executable.path")
          - delete_key(attributes, "process.pid")
          - delete_key(attributes, "process.runtime.description")
          - delete_key(attributes, "process.runtime.name")
```

```
        - delete_key(attributes, "process.runtime.version")
    log_statements:
      - context: resource
        statements:
          - delete_key(attributes, "k8s.pod.start_time")
          - delete_key(attributes, "os.description")
          - delete_key(attributes, "os.type")
          - delete_key(attributes, "process.command_args")
          - delete_key(attributes, "process.executable.path")
          - delete_key(attributes, "process.pid")
          - delete_key(attributes, "process.runtime.description")
          - delete_key(attributes, "process.runtime.name")
          - delete_key(attributes, "process.runtime.version")
  transform/add_resource_attributes_as_metric_attributes:
    error_mode: ignore
    metric_statements:
      - context: datapoint
        statements:
          - set(attributes["deployment.environment"],
          ↪  resource.attributes["deployment.environment"])
          - set(attributes["service.version"], resource.attributes["service.version"])
```

The `batch` processor helps with terminal logging, the `resourcedetection` has the duty to chose what kind of host telemetries to filter and all the other commands are used to reduce the number of data and filter by some keys.

Exporters

On the other side of receivers we found the exporters: their role is to send data to one or more backed service, from the terminal to Grafana.

```
exporters:
  otlphttp/grafana_cloud:
    endpoint: "https://otlp-gateway-prod-eu-west-2.grafana.net/otlp"
    auth:
      authenticator: basicauth/grafana_cloud
  otlp/dash0:
    endpoint: ingress.eu-west-1.aws.dash0.com:4317
    headers:
      Authorization: Bearer auth_<authCode>
  otlp/signoz:
   endpoint: "ingest.eu.signoz.cloud:443"
   tls:
     insecure: false
   headers:
     "signoz-ingestion-key": "<INGESTION_KEY>"
  otlp/signoz/local:
```

```
    endpoint: "0.0.0.0:4319"
    tls:
      insecure: true
  otlphttp/newrelic:
    endpoint: "https://otlp.eu01.nr-data.net"
    headers:
      api-key: "eu01xx8e7332ee3c067a94050e5cd420FFFFNRAL"
  otlphttp/lumigo:
    endpoint: "https://ga-otlp.lumigo-tracer-edge.golumigo.com"
    headers:
      Authorization: LumigoToken t_f7c527a2942b493db40b7
```

Each exporter has the configuration for the specific backend service. For some exporter the path is quite straight forward, like for the Docker image of SigNoz, where the only configuration needed are the port and avoiding tls security check. Other exporter needs a key, like SigNoz cloud or Dash0. Others are so specific they need a full authentication: Grafana needs an extension in order to send data to its Cloud option.

Extensions

Extensions are, as the name suggests, some add-on to the process. In this project I've used two extensions: the extension to log in to Grafana Cloud and zpages extension, a `localhost` minimalist UI to navigate through telemetries.

```
extensions:
  zpages:
    endpoint: "localhost:55679"
  basicauth/grafana_cloud:
    client_auth:
      username: "<userID>"
      password: "<pwd>"
```

Service The service is the core configuration of the collector: it decides, for every telemetry, which receivers have to be used, how to process the data and where to send them. In this snippet it shown one possible configuration where telemetries are sent to Grafana, Dash0 and SigNoz cloud. Due to hardware restrictions, I've mostly used one or at most two different exporters at the time.

```
service:
  extensions: [basicauth/grafana_cloud, zpages]
  pipelines:
```

```
    traces:
      receivers: [otlp]
      processors: [resourcedetection]
      exporters: [otlphttp/grafana_cloud, otlp/dash0, otlp/signoz]
    metrics:
      receivers: [otlp, hostmetrics, docker_stats]
      processors: [resourcedetection,
      ↪  transform/add_resource_attributes_as_metric_attributes]
      exporters: [otlphttp/grafana_cloud, otlp/dash0, otlp/signoz]
    logs:
      receivers: [otlp]
      processors: []
      exporters: [otlphttp/grafana_cloud, otlp/dash0, otlp/signoz]
```

Collector usage Since I've mostly used the collector with logs in debug level, to compare received telemetries and what telemetries backed shows, for most of the project happened that I had tons of log always printing in the terminal, making almost impossible to catch any error. The full command I've come up to run the collector is, then:

```
nohup otelcol-contrib --config otel-collector-config.yaml > collector.log 2>&1
```

The `nohup` command allows the process to run in background and the `> collector.log 2>&1` command prints every log (together with process errors) in a file called *collector.log*. Still, the amount of log was a problem and any software cannot navigate the file due to its dimensions and its continuos new lines appended. The solution was a small Python script that checks the output files to creates a backup and clean the file every 10000 lines (customizable).

```
import time
import shutil
import os
from datetime import datetime

LOG_FILE = "collector.log"
CHECK_INTERVAL = 1
MAX_LINES = 10000

def count_lines(filepath):
    #Lines count
    with open(filepath, "r", encoding="utf-8", errors="ignore") as f:
        return sum(1 for _ in f)

def rotate_log():
```

```
    #Creates a copy using the timestamp and cleans the original file
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    backup_file = f"./bk/{LOG_FILE}_{timestamp}.log"
    shutil.copy(LOG_FILE, backup_file)
    open(LOG_FILE, "w").close()
    print(f"[{timestamp}] Log saved as {backup_file} and resetted.")

def monitor_log():
    #Monitors log file and rotate after reaching the limit
    while True:
        if os.path.exists(LOG_FILE):
            line_count = count_lines(LOG_FILE)
            if line_count >= MAX_LINES:
                rotate_log()
        time.sleep(CHECK_INTERVAL)

if __name__ == "__main__":
    print("Log monitor started...")
    monitor_log()
```

4.2 OpenTelemetry x Online Boutique

As previously shown, Online Boutique has a much more complex architecture and it's deployed on Kubernetes, so with a clustered microservices development. In this case, a crucial point is the usage of host and `IP`: each service has its own port inside the cluster network, each cluster has its `IP` that can be easily be referred within the cluster but has a different `IP` when being referred from outside it. In the image below, we can see a k8s application deployed with a virtual machine and how communication works from the host machine to the nodes, between the nodes and from the outside internet to the nodes: each node has a different `IP` to be used from the host machine, they communicate with each other using a cluster `IP` and every `IP` is different from the tipycal `localhost IP`. The same problem issued before exists on reaching the host machine network, the typical `localhost`, `127.0.0.1` or `0.0.0.0` aren't a suitable option when it comes to communicate from inside the cluster to outside the cluster. In this case, then, the issue is to find the external address of the host machine and it's accessible via command line with the `IP` command: `ip a | grep inet`. With `ip a` command the terminal lists all the IP hosted in the machine and the `| grep inet` filters the result showing only the line with the IP and hides the details. Here is an example result without the filter:

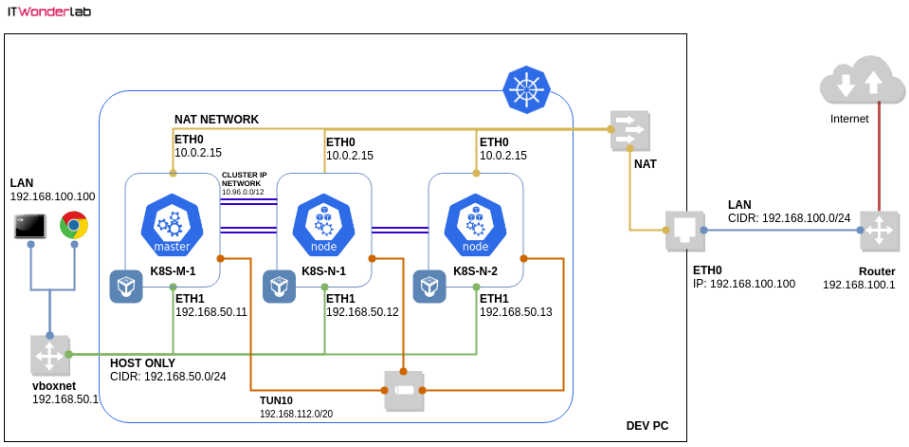


Figure 4.2: Networking in a Kubernetes Cluster using VirtualBox.

Source: Source: www.itwonderlab.com/nodeport-kubernetes-cluster/

```
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default
↪  qlen 1000
    link/ether 00:15:5d:eb:e1:19 brd ff:ff:ff:ff:ff:ff
    inet 172.29.92.175/20 brd 172.29.95.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::215:5dff:feeb:e119/64 scope link
       valid_lft forever preferred_lft forever
```

This is one of the networks listed in the terminal, the one of our interest: it's the host machine IP reachable from containers. As stated before, the OpenTelemetry collector receivers with `grpc` protocol works at port `4317`; the exporter injected in the application running in a cluster, then, should be `172.29.92.175:4317`.
In order to use the auto-instrumented agents in a Kubernetes app, there a few steps to follow, where the order matter!

- install a cert-manager in the cluster;
- install the OpenTelemetry operator in the cluster;
- install the services with the reference to the operator.

4.2.1 Cluster installations

Let's break it down to the actual passages needed. Of course the first step is to create a cluster:

```
k3d cluster create my-cluster -p "8081:80@loadbalancer" --api-port localhost:8080
↪  --k3s-arg "--disable=traefik@server:0"
```

This command is always the first step, since it create the cluster, the main environment for Kubernets to run. Among with standard command as the `cluster create`, there are three parameters that works in order to make application accessible and make possible to communicate from inside to outside the cluster.

- `-p "8081:80@loadbalancer"`: This makes the application accessible from the browser. The load balancer is a Kubernetes service that, as the name suggests, distributes incoming requests between different replicas of each service. With this parameter, I am binding the cluster's port 80 (the default port for the load balancer) to port 8081 on the host machine, so that the application can be accessed via a browser from the host machine;
- `--api-port localhost:8080`: This exposes the Kubernetes cluster's API on the default localhost port. This allows the OTel (OpenTelemetry) collector running on the host machine to access the Kubernetes API without any additional configuration, enabling the collector to communicate with the cluster;
- `"--disable=traefik@server:0"`: This disables Traefik, which would normally be deployed as the default ingress controller in a cluster created with k3d. Disabling Traefik provides two key benefits related to resources and ports. First, since the project does not require Traefik, it's unnecessary and would only consume resources. Second, Traefik uses the same ports as the OTel agent, which could cause conflicts. Disabling it avoids these conflicts and prevents the need for manual port configuration (which would otherwise be required for the OTel agent and services communicating with it).

Now the cluster exist, it's important to install a cert-manager for OTel agent to work properly, then the OTel agent itself. For the cert-manager I've used the recommended one for Kubernetes, the cloud-native certificate manager for Kubernets and OpenShift. Both the cert-manager and the OTel agent are directly installed from the latest release on GitHub.

```
kubectl apply -f https://github.com/cert-manager/cert-manager/releases/latest/downloa
↪  d/cert-manager.yaml
```

```
kubectl apply -f https://github.com/open-telemetry/opentelemetry-operator/releases/la
↪ test/download/opentelemetry-operator.yaml
```

Now all the fundamental installation for instrument observability in the app have been made, these are the pods actually running in the cluster.

```
NAMESPACE                     NAME                                                READY   STATUS    RESTARTS   AGE
cert-manager                  cert-manager-584674cf4-lscwh                        1/1     Running   0          3h32m
cert-manager                  cert-manager-cainjector-77b6dc6654-wtrdf            1/1     Running   0          3h32m
cert-manager                  cert-manager-webhook-68f89ddf6c-8956f               1/1     Running   0          3h32m
kube-system                   coredns-576bfc4dc7-8cfw6                            1/1     Running   0          3h47m
kube-system                   local-path-provisioner-6795b5f9d8-8rsl2             1/1     Running   0          3h47m
kube-system                   metrics-server-557ff575fb-9qzpk                     1/1     Running   0          3h47m
opentelemetry-operator-system opentelemetry-operator-controller-manager-6995c444cb-nkrpw 2/2     Running   0          3h31m
```

Three namespaces has been created: the kube-system, handling Kubernetes options configurations and functioning; the cert-manager, to handle the certification needed by the OTel agent and opentelemetry-operator-system for the OTel agent itself.

4.2.2 Agent injection and Kubernetes dashboard

When injecting the auto-instrumentation in Kubernetes, the order matter: the OTel instrumentation has to be already deployed when services are installed, because the annotations in service yaml that points what to search for injection works only at first installation, not on further customization or any successive change. This deployment installs an instrumentation in the cluster with all the specs on languages and environment variables, when needed.

```yaml
---
apiVersion: opentelemetry.io/v1alpha1
kind: Instrumentation
metadata:
  name: auto-instrumentation
  labels:
    app: otel-auto-instrumentation
spec:
  exporter:
    endpoint: 172.29.92.175:4317
  propagators:
    - tracecontext
    - baggage
  sampler:
    type: parentbased_traceidratio
    argument: "1"
  python:
    env:
```

```
      - name: OTEL_EXPORTER_OTLP_ENDPOINT
        value: "http://172.29.92.175:4318"
      - name: OTEL_SERVICE_NAME
        value: "recommendationservice"
      - name: ENABLE_TRACING
        value: "1"
      - name: OTEL_PYTHON_LOGGING_AUTO_INSTRUMENTATION_ENABLED
        value: "true"
      - name: ENABLE_PROFILER
        value: "0"
  java:
    env:
      - name: OTEL_EXPORTER_OTLP_ENDPOINT
        value: "http://172.29.92.175:4317"
      - name: OTEL_SERVICE_NAME
        value: "adservice"
      - name: ENABLE_TRACING
        value: "1"
      - name: OTEL_JAVAAGENT_LOGGING
        value: "true"
      - name: ENABLE_PROFILER
        value: "0"
```

This yaml has the instructions to install the OTel injection for java and python, with tracing and logging enabled but profiler disabled. The OTel collector has to be specified since this service will run inside the cluster, but the collector has already been started outside the cluster. The IP, then, is the bridge from the cluster to the host machine network (i.e., the localhost). Now that the auto-instrumentation has been installed, the application can be installed in the cluster, following GitHub repository instructions on the README.

Before installing the application one last installation is needed, the Kubernetes dashboard, with helm.

```
helm repo add kubernetes-dashboard https://kubernetes.github.io/dashboard/
helm upgrade --install kubernetes-dashboard kubernetes-dashboard/kubernetes-dashboard
↪  --create-namespace --namespace kubernetes-dashboard
```

At this point there are no accessible functionalities from a normal browser, but everything is ready to deploy the application, observability and Kubernetes dashboard.

4.2.3 Application and pre-build observability

The first step is now to deploy the application in it's basic configuration, without the coded observability. The yaml file deployed has few different things from the Google's original repository. First, both the java service (adservice) and the python service (reccomandationservice) has the annotation for the OpenTelemetry injection:

```
[...]
spec:
  selector:
    matchLabels:
      app: recommendationservice
  template:
    metadata:
      annotations:
        instrumentation.opentelemetry.io/inject-python: "true"
[...]
spec:
  selector:
    matchLabels:
      app: adservice
  template:
    metadata:
      annotations:
        instrumentation.opentelemetry.io/inject-java: "true"
[...]
```

A key difference in the deployment yaml is the creation of a ServiceAccount with admin role, able to access any metric of the cluster, to send them to dashboard. This implementation is divided in three steps: first, the creation of the ServiceAccount named `admin-user`, then the creation of an admin role with administration privileges inside the cluster, lastly the binding of the account and the role.[2]

```
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: admin-user
  namespace: kubernetes-dashboard
---
apiVersion: rbac.authorization.k8s.io/v1
```

[2] In this thesis I've shared just partially the configuration used. For the entire configuration refer to the GitHub repository: github.com/NonnoPinto/otel-microservices-demo/blob/main/release/kubernetes-manifests.yaml.

```yaml
kind: ClusterRole
metadata:
  name: admin-user
rules:
  - apiGroups: ['']
    resources: ['nodes/stats']
    verbs: ['get', 'watch', 'list']
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: admin-user
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
  - kind: ServiceAccount
    name: admin-user
    namespace: kubernetes-dashboard
---
```

With the below installation command all the services are installed in the cluster, the admin user is created and bound with admin privileges. Now the application is navigable at `http://localhost:8081/`.

```
kubectl apply -f ./release/kubernetes-manifests.yaml
```

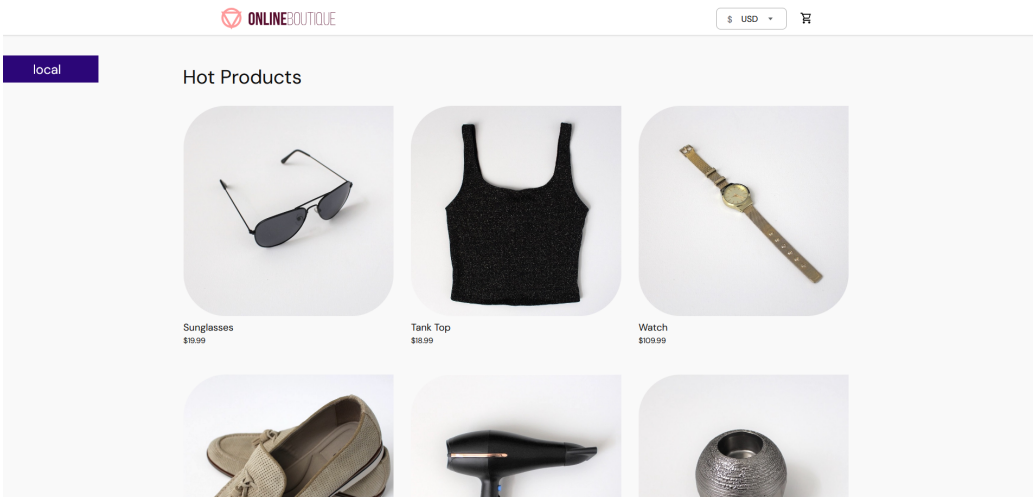


Figure 4.3: Online Boutique - home page.

Now it's good practice to check if the injection worked properly, with the command `kubectl describe pod <pods-name>` and check in the output description if the

OpenTelemetry auto-instrumentation has been successfully attached to the pod. Here an example of the attended result (the description is much longer, but these two lines are the one that state the right output of the injection):

```
Events:
  Type    Reason     Age    From       Message
  ----    ------     ----   ----       -------
  Normal  Created    5m55s  kubelet    Created container
  ↪  opentelemetry-auto-instrumentation-java
  Normal  Started    5m55s  kubelet    Started container
  ↪  opentelemetry-auto-instrumentation-java
```

Last installation for the application is to enable the pre-build services observability with the kustomization, where the annotations used for the auto-instrumentation needs to be replicated otherwise would be lost and to inject would be removed. As an example, is here reported the kustomization of just one service, the same is repeated for each observable service. Below, the command to install the kustomization.

```
# currencyservice - tracing
- patch: |-
    apiVersion: apps/v1
    kind: Deployment
    metadata:
      name: currencyservice
    spec:
      template:
        spec:
          containers:
            - name: server
              env:
              - name: COLLECTOR_SERVICE_ADDR
                value: "172.29.92.175:4317"
              - name: OTEL_SERVICE_NAME
                value: "currencyservice"
              - name: ENABLE_TRACING
                value: "1"
              - name: ENABLE_LOGGING
                value: "0"
              - name: DISABLE_PROFILER
                value: "1"
```

```
kubectl apply -k ./kustomize
```

Lastly, we can expose the Kubernetes dashboard port outside the cluster and create a token for the admin-user to use as login for the dashboard

```
kubectl -n kubernetes-dashboard port-forward svc/kubernetes-dashboard-kong-proxy
↪  8443:443
kubectl -n kubernetes-dashboard create token admin-user
```

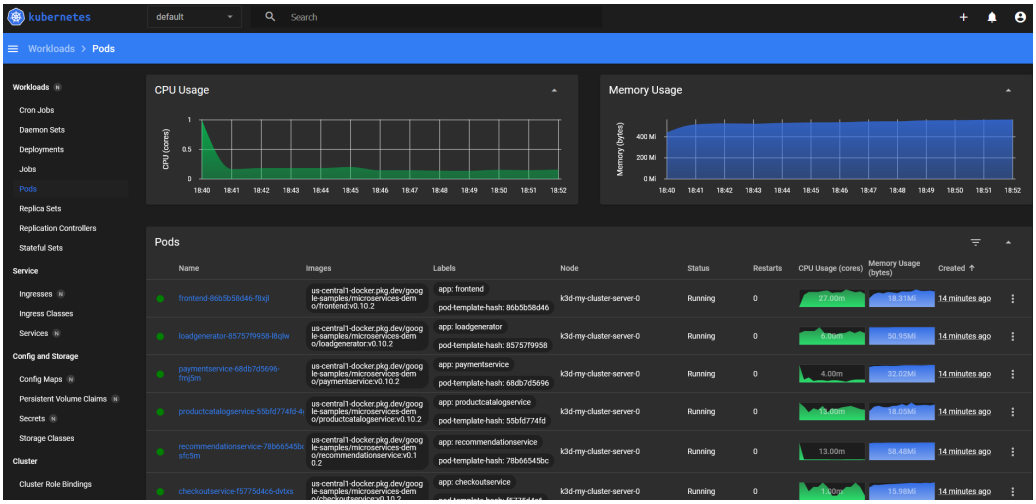


Figure 4.4: Kubernets Dashboard - Pods.

4.2.4 Kubernetes metric monitoring

I've also chosen to "duplicate" the k8s dashboard data in the observability pipeline: k8s hostmetrics. This deployment allows OTel to manage also metrics about resource usage, pods replicas, container status and other metrics related to k8s state. In order to obtain these metrics I had to create an admin role and bind it to some internal access permissions, the actual deployment that does the job (in the `yaml` below) and an internal OTel collector, otherwise I would have had to give admin permission to an external agent, not a best practice. This collector deployed inside the cluster collect the metrics and sends them to the external collector, as already explained before. Below some significant snippet from the essential configuration done to enable k8s monitoring.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: otelcontribcol
  labels:
    app: otelcontribcol
```

```yaml
data:
  config.yaml: |
    receivers:
      k8s_cluster:
        collection_interval: 5s
      kubeletstats:
        collection_interval: 5s
        initial_delay: 1s
        auth_type: "serviceAccount"
        endpoint: "https://${env:K8S_NODE_NAME}:10250"
        insecure_skip_verify: true
        node: '${env:K8S_NODE_NAME}'
        metrics:
          k8s.pod.cpu.usage:
            enabled: true
          k8s.pod.cpu.node.utilization:
            enabled: true
        [...]
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: otelcontribcol
  labels:
    app: otelcontribcol
spec:
  replicas: 1
  selector:
    matchLabels:
      app: otelcontribcol
    [...]
    spec:
      serviceAccountName: otelcontribcol
      containers:
      - name: otelcontribcol
        image: otel/opentelemetry-collector-contrib
        args: ["--config", "/etc/config/config.yaml"]
        volumeMounts:
        - name: config
          mountPath: /etc/config
        imagePullPolicy: IfNotPresent
        resources:
          limits:
            cpu: 200m
            memory: 128Mi
        env:
          - name: K8S_NODE_NAME
            valueFrom:
              fieldRef:
                fieldPath: spec.nodeName
```

```
57          [...]
```

4.2.5 Kubernetes Horizontal Pod Autoscaling

The last important aspect, core functionality of a microservice architecture, is the possibility of dynamically change the number of pods of a particular service (named replicas) based on the workload. Kubernetes can manage this configuration with the horizontal pod autoscaling (hpa): a configuration that sets boundaries of CPU usage and replicas number. Based on these metrics, k8s manage to replicate (or delete) some pods. For example, with the command `kubectl autoscale deployment frontend --cpu-percent=10 --min=1 --max=7`, k8s will try to keep the CPU percentage usage of the frontend service under the 10%, using at most 7 replicas of the pod, to balance the single pod usage. This functionality, already analyzed in the architecture sections, is an important aspect to avoid bottlenecks, crashes due to resource usage or anything in the scope of a single pod handling too many request at the same time and much more. From a high level perspective replicas are calculated with this formula

$$\text{desiredReplicas} = \left\lceil \text{currentReplicas} \times \frac{\text{desiredMetricValue}}{\text{currentMetricValue}} \right\rceil$$

where the ratio between desired metric and actual metric (taken as an average usage from each replica) determinate the multiplier for the current replicas. For example, if the ration is 2 because the usage is double what it's expected, the number of replicas will be two times the current number. Since this job is done dynamically, it is possible that the number of replicas keeps fluctuating frequently. This is usually referred to as thrashing, or flapping. When also hpa is installed in the cluster for some target deployments, everything is set and ready to go.
In our environment I chose to install hpa in four services, all with desired metric base on CPU usage. The `yaml` installation for all of them are like this one, with different max replicas and CPU usage thresohold.

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: adservice
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
```

```
    name: adservice
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 60
```

Chapter 5

Observability in practice

This chapter will present the final steps of the research, core and goal of the project. As previous chapters have shown, observability is a spectrum of many different things, approaches, tools and possible utilization. It's almost impossible to give a comprehensive view of it and it's outside the perimeter of this work. In this chapter I'll present the most relevant views and insight I had on the data analysis of the two applications, in order to prove that observability give a real boost to web app development and to show different sides of this word. For each back end service used and tested, I'll show different insights, depending on what each of them have to offer: everyone have a different focus and using all of them in the same way would mean trying to force them in what they are not. For example, NewRelic is deeply focused on navigation into traces and metrics, with some prebuilt easy dashboards and an always-ready query IDE. On the other side, SigNoz (maybe due to its younger age), has less navigation skills and appears more dashboard-oriented. In each case, the main focus will be trying to underline the link between workload and pod scaling and how observability helps in this investigation.

First to be presented are the three pillars and metrics, the main data used in this project, with a list and some insights on what aspects of JPet Store and Google Online Boutique were under the observability implementation. Then, how data were visualized by some default views and in some dashboards, result of the work done. On the last part some possible alerting.

For its nature, logging is mostly strictly related to application processes and development debugging. Logs are a sharp tool when it comes to finding the null reference in your application that hides in some inheritance or unchecked variables, also they come to help when specific errors are triggered. But for this thesis very few logs have been

instrumented and they will not be presented in any of the tools for a couple of reasons: they best work when developed inside the code -and I've used projects already implemented- and they don't give meaningful insights to and application context (they work better in small, specific, cases).
When dealing with such a huge amount of data, the first step is to find some meaningful field to categorize them and focus on the project's goal such as scaling and system observability. We decided to look for a correlation between user load, resource usage and pod scaling (for the microservices architecture) and for the best possibility to navigate through each metric or cluster of metrics to explore the system or to find the root of a problem.
In following sections I'll go through some key features and approaches I've tested without claiming to be exhaustive on all the developments made, let alone on tools capabilities.

5.1 Pillars

5.1.1 Metrics

Metrics are not just a monitoring pillar, but also a fundamental part of observability, where they no longer are used only with thresholds and alerts, but they can be navigated, connected and show unknown bugs. Every application produces tons of different metrics that can be exported. Let's consider only resources metrics for a single node Kubernetes application: there are memory usage, memory limit, memory request metrics, same for the CPU and these metrics have multiple layers like system, container, node and pod! These are just the very basic metrics of a very basic application, with more complex applications (i.e. multi-node) they grow exponentially, since for every node there is a replica of all metrics container-related. There's more: for every metric there could be the real-time gauge, average over the last few minutes and a percentage, adding another multiplier to their complexity.
In a real-world application, an application with any medium complexity architecture would create so many metrics that the agent overhead would have a significant impact in performances both in the application and in observability. Here a crucial part is played by configurations used: it is in fact possible to drop many different metrics based on their name, data length or even limit the amount of resources used by the agent. All these

configurations, if tuned with the right amount of data, can significantly decrease this overhead and still give reasonable observability. Data filter can be done at the agent level to avoid creating these metrics and have less impact on the application or can be done at collector level, in this case the agent still creates all the metrics but multiple collectors with different configurations can be used depending the backend characteristics and capabilities: a tool running on an AWS server or running in an old server on premise have different scaling capabilities and could benefit of different metrics configurations. I've tested many different metrics, some were the default metrics, others had to be declared. I didn't have to drop metrics due to the small sizes of the entire system, but there are some metrics that were more interesting than others, for my project. An example over all the others is the system resources metrics: everything has been run in my PC so tuning the hardware limits wasn't any profitable, as well as out of the scope of the thesis. All the system resources metrics have been tested but not used, since it was not something relevant for my use case.

Table 5.1: Metrics Table.

Name	**Type**	**Description**
container_cpu_utilization_ratio	gauge	CPU usage ratio of the container.
container_memory_percent_ratio	gauge	Percentage of memory used by the container.
container_memory_usage_total_bytes	gauge	Total memory usage by the container.
container_network_io_usage_tx_bytes_total	counter	Total transmitted network bytes by the container.
container_network_io_usage_tx_dropped_total	counter	Total dropped transmitted packets by the container.
k8s_container_cpu_request	gauge	CPU requested by the container in Kubernetes.
k8s_container_memory_request_bytes	gauge	Memory requested by the container in Kubernetes.
k8s_deployment_available	gauge	Number of available pods in a deployment.

Name	Type	Description
k8s_deployment_desired	gauge	Desired number of pods in a deployment.
k8s_hpa_current_replicas	gauge	Current number of replicas in HPA.
k8s_hpa_desired_replicas	gauge	Desired number of replicas in HPA.
process_runtime_jvm_cpu_utilization_ratio	gauge	CPU utilization by JVM process.
process_runtime_jvm_memory_usage_bytes	gauge	Memory usage by JVM process.
system_cpu_load_average_1m	gauge	System CPU load average over 1 minute.
system_cpu_utilization_ratio	gauge	System CPU utilization ratio.
system_memory_usage_bytes	gauge	Total system memory usage.
system_memory_utilization_ratio	gauge	System memory utilization ratio.

5.1.2 Traces

The key comparison with metrics was the one made with traces, used as sum of traces more than as details on the navigation. This information has been used to estimate the user load for each service (or for the entire application), that is just a part of the workload an application has to process, but it's probably the most variable one: if keeping a server up or sending up logs are near-constant resource usage operations, user interaction may be absent for hours (an Italian kid's game website isn't expecting much traffic during nighttime), create spike at a specific time (like at the opening of ticket sales for the Taylor Swift concert) or constantly increase the load due at an unexpected event (like the constant increase of users on Netflix due to Covid-19 pandemic).
Traces brings much information and some are more related to the project than others: the path and the service (for the Online Boutique project). Using these information I had the possibility to check the amount of traces for each service and track the paths that

both the pre-installed synthetic traffic and Locust traffic were following. The latter tool will be explained later in detail in this paragraph.

A simple view on traces can be done with the zpages extension, a simple web hosted GUI to navigate inside telemetries. It has many different capabilities, such as listing traces, in their pure format:

```
When                        Elapsed (sec)
----------------------------------------
2025/03/20-00:47:01.429172     2.061110 trace_id: 62886aab361e41a6ed8ba09ef09b6bef
↪  span_id: a754eff0691bf681 parent_span_id: ed8339ecbc9f8ff8
                                        Status{Code=Unset, description=""}
                                        Attributes:{http.method=GET,
                                        ↪  http.status_code=200, http.url=http://%2F
                                        ↪  var%2Frun%2Fdocker.sock/v1.25/containers/
                                        ↪  0991d51de11ee901cd0427d89b806c27ee82c070f
                                        ↪  9f460445c369eff33aa74c0/stats?stream=0,
                                        ↪  net.peer.name=/var/run/docker.sock, user_
                                        ↪  agent.original=OpenTelemetry-Collector
                                        ↪  Docker Stats Receiver/v0.0.1}
```

But what makes traces so important is their representation of the workload of an application: more traces directly represents a higher workload, that's why a traces time chart is fundamental in observability, since it helps to understand the performance spikes. To visualize the traces I've used multiple tools, in order to compare them and verify their reliability. The source of truth for this data was Locust time chart: Locust is an open-source performance testing tool used to simulate users for load testing. It's highly scalable and scriptable in Python, making it versatile for testing various systems and protocols. With a web-based interface, it provides real-time insights into test results, allowing users to control and analyze the load generation. It also allows visualizing some insights about the test and a small recap at the end. Its python script can be manually developed, or it can be automatically produced from a HAR file[1]: in this project JPet has been tested with a file produced from a HAR file, Online Boutique with the load generator file instrumented with a double wave test for a total test time of 10 minutes. This second configuration gives a similar result: Other results are shown, but they will be presented in next chapters.

[1] A HAR (HTTP Archive) file is a JSON-formatted file that records detailed information about network requests and responses made by a web browser. It is generated from the browser's Developer Tools and is commonly used for debugging performance issues and analyzing web traffic.

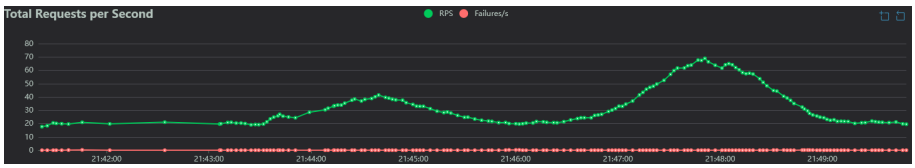


Figure 5.1: Locust requests time chart.

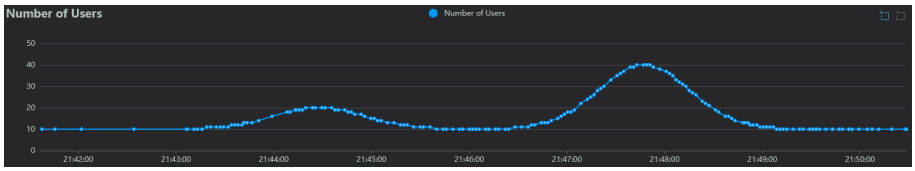


Figure 5.2: Locust users time chart.

5.2 Data analysis and navigation

The very first mandatory analysis is to understand what data I'm dealing with: recognize different type of data from different parts of the system.

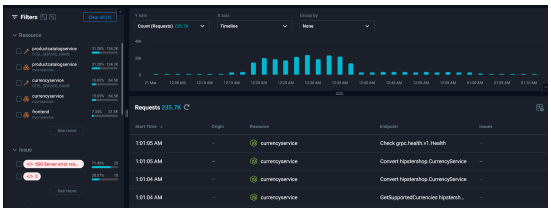
5.2.1 Traces

One first feature in most tools used is the service correlation map, here are some screens. Another fundamental aspect is the ability to navigate these traces. All of them also

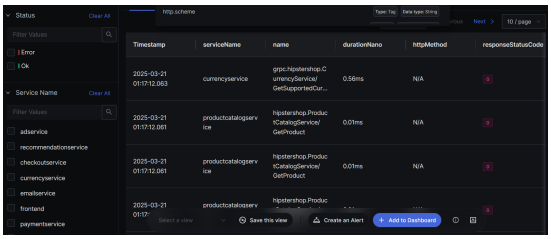
(a) Lumigo (b) SigNoz

Figure 5.3: Service map.

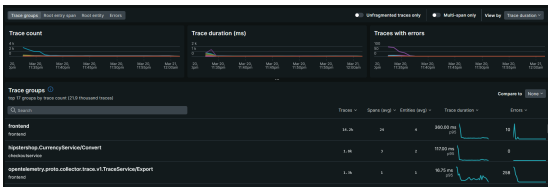
allows to further investigate each trace with standard information as the URL, the http request, response etc. But also they allow to connect the trace with its context and in some cases also to understand possible errors and their root cause. Lumigo also offers this possibility with an AI assistant. On the other hand, both NewRelic and SigNoz



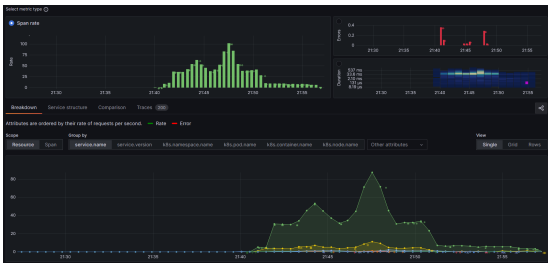
(a) Lumigo



(b) SigNoz

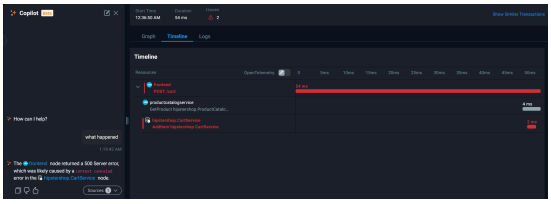


(c) NewRelic



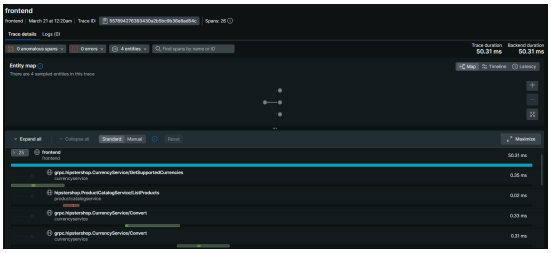
(d) Grafana

Figure 5.4: Traces navigation.



(a) Lumigo

(b) SigNoz



(c) NewRelic

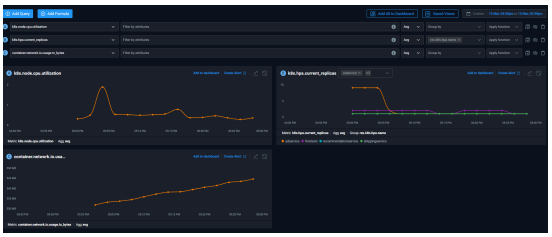
(d) SigNoz

Figure 5.5: Traces analysis.

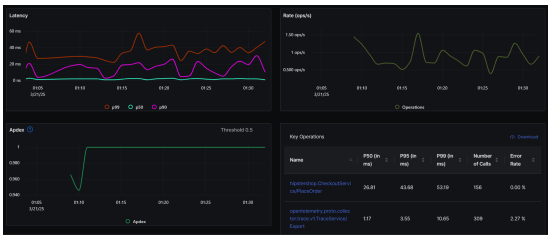
offers this option manually. These are just few of the capabilities of the tools I've tested. Each of them is very different, some are quite new, some are in the market since many years, but all of them has their features to see, work and investigate traces.

5.2.2 Metrics

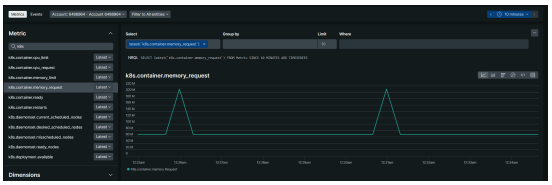
Something quite similar happens for metrics, where still all the tools allow searching, navigate and investigate. Since metrics could be almost everything happening in a system, here each of them uses different approaches. Lumigo let the user create very quick dashboard-like investigation, NewRelic has different possible ways, like choosing a metric and go deep inside or, feature shared with SigNoz, a more service-focused analysis, where the service is the first step and user can deep dive inside the service metrics.



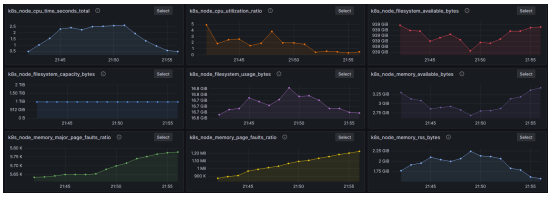
(a) Lumigo



(b) SigNoz



(c) NewRelic



(d) Grafana

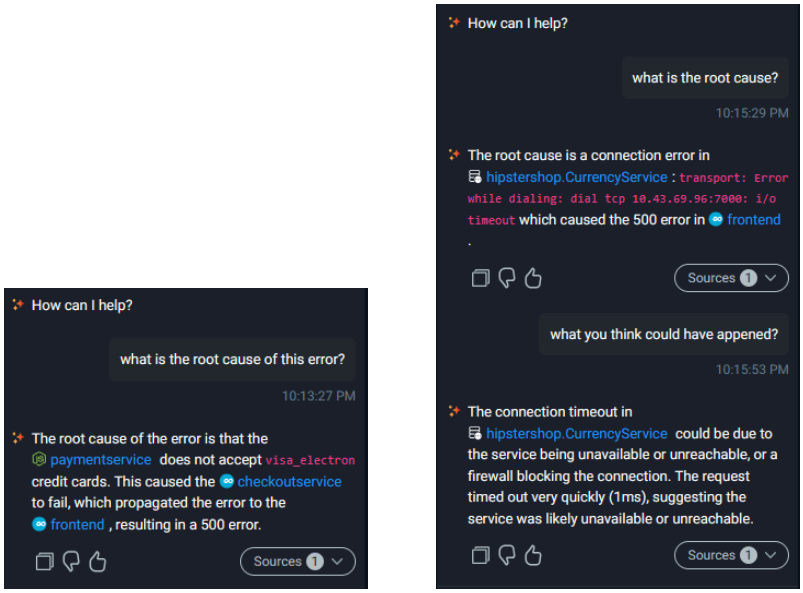
Figure 5.6: Metrics navigation.

AI assistant

All the tools provide various ways to facilitate navigation and quickly identify any anomalies in the system, mainly through graphics and colors. In this field, an honorable mention goes to Lumigo, which offers an AI assistant even in its free tier. Grafana and New Relic, like many others, are also incorporating AI into their tools: Grafana's AI assistant is only available in enterprise plans, while New Relic's AI mainly serves as a query builder assistant.

Lumigo, on the other hand, provides an AI analysis tool accessible to all users, including those on the free plan. This tool analyzes transaction errors and can answer questions like a chatbot. Here are two examples of error analysis: on the left, an error caused by a service configuration, where AI Copilot quickly identified the root cause in seconds; on

the right, an error I intentionally triggered by shutting down `currencyservice`, and once again, AI Copilot found the root cause in seconds.



(a) SigNoz (b) Grafana

Figure 5.7: Lumigo AI Copilot.

5.3 Data Visualization

Another useful approach, possibly more in between observability and monitoring, is to visualize data with dashboards composed of panels illustrating any type of aggregation or list of telemetries. Dashboard are particularly useful for their high level of customization, since they are like a white canvas where each dev or team can paint their own needs. Any standard dashboard offers different panels like single number, table, time chart, bar chart, pie chart and some more advanced dashboards offer also more specific panels like heat map, linear distribution and much more. Dashboards are mostly

(a) SigNoz (b) Grafana

Figure 5.8: Dashboards.

interesting since they can show just the desired telemetries, with comparison, aggregation or formulas. A row metric cannot answer the question “What is the CPU usage peak for each service?”, a dashboard panel with the right query can. Dashboard has a different goal compared with telemetries navigation: they are made to watch on specific aspects of a system and they are designed with the exact required panels with the information for that team: a high level manager wouldn’t probably be interested in navigate inside the CPU spikes and the node scalability, but it would prefer to monitor costs of Cloud host and detect possible saving strategies. This is a following step, something to produce after the metric navigation and analysis, it’s the result of hours spent inside the data.

Chapter 6

Results

This project has walked me through the state of the art of observability with a small touch of some future possibilities. In this final chapter I'll try to summarize the different aspects seen in the project and draw some conclusion and future work opportunities. Firstly, I'll give feedback on the different tools used to pursue this project's goal, summarizing the experience with each of them. Then, I'll provide a brief overview of a major problem with microservices: autoscaling. Later, with the help of the previously shown charts and data, I'll discuss the project's contribution to data correlation, offering some insights. Lastly, I'll present some conclusions and possible future work on this topic.

6.1 Tools analysis

This project brought me to a lot of different tools, some were useful, and some were less suitable for the goal; some were out of the box and some were quite intricate to set up fully; finally, some had a straightforward usage, strictly connected with a few different possible usage, some had a huge variety of possible applications in different fields and contexts. Here I'll try to return an overview on most of the tools I've used for the thesis. Not all the tools I've used have had a central role in the development of the project, some were just a support for the bigger project and that's why I'll focus only on the tools that have their core on observability and scaling.
I'll start with the main character of this project: OpenTelemetry, the new standard of observability. Then, a comparison of data visualization and metrics analysis tools: Grafana, SigNoz, Dash0 and New Relic. All of them played a key role on this

observability journey and they all have their perks and downsides. In the end, an honorable mention to some testing and scaling tools that deserves a mention: Locust and Kubernetes HPA.

OpenTelemetry

Maybe the main character of this thesis, the core of all the implementation, OpenTelemetry confirmed its fame and goes beyond. It's not just a standard for telemetries, it's also a powerful tool to inject agents for many different languages (and the list is continuously growing) and the collector it's on its own another amazing configurable collector. It has many different possible implementation, from the telemetries that can be coded or auto-injected to the collector, that works in Docker, Kubernetes, Linux system and much more.
In this project I've configured a few different collectors and have to admit that when the logic is understood it's pretty straightforward and it's possible to customize it to receive, filter and send data with low effort. It's continuously growing integration in all the observability and monitoring tools is for sure helping this standard growing.

Data visualization

Every tool shown in this thesis is the result of research and tests; some tools have been found to be interesting enough, but others didn't have distinctive features to enter the project. Grafana was the main tool in the first place: it's used worldwide, with million of users and many companies entrust it for their application observability, it also has a strong community. As expected, I've found it a complete tool, still growing with new integration (like the one published as I'm writing, for the OTel metrics exploration[1]) and a huge data integration from different possible instrumentation like Prometheus, Loki, Tempo, OpenTelemetry and much more but also with the possibility to use its own agent and collector (called Grafana Alloy). I've also found some lack of possibilities in the free tier, the most important was the limitation of 1000 traces per dashboard, a limit that restrict the time range to just few seconds already in a small application. Despite all the other tools that used an API key to authorize the collector to send data, Grafana has its own extension to authorize the collector and this is an extra factor of security, but could

[1]The release blog here: grafana.com/blog/2025/03/10/grafana-drilldown-first-class-opentelemetry-support-now-available-for-metrics/.

also be a higher grade of difficulty into data integration with OpenTelemetry or other tools outside the Grafana ecosystem.

Another mature tool is NewRelic, with its nearly 20-year history and the vast array of options available on the website. It spans from dashboards to metric navigation, from OpenTelemetry integration to custom agent for different architectures like Kubernetes or AI models. It covers all the observability necessities and allows free tier's user to navigate the application status in an intuitive interface. Some possible improvements are the website performance and some option available also with OpenTelemetry implementation: many options only work, in fact, exclusively with the NewRelic agent.

Two younger tools are SigNoz and Lumigo. SigNoz, born less than 5 years ago, has an incredible intuitive interface, and the Docker Standalone possibility is a unique opportunity to run an observability tool without subscription or enterprise plans. In the matter of open-source availability, SigNoz is probably the best tool exposed in this thesis, since the GitHub repository has the whole application and it's continuously updated from the community, which is also quite active in the Slack support channel (i.e. a major release was made during the project and the update made some huge changes that broke my architecture; the Slack support community helped with the solution and also listened to my suggestions - a documentation change was made under my suggestions). It still has a lack of options when compared to more mature tools but the continuous update and discussion inside the community show promise and great potential.

Lumigo, with its almost ten years of experience, has been the most particular tool I've tested. It is developer oriented and focused on data correlation, and having the AI Copilot in free tier makes it a *unicum* compared with other tools. Setting the AI Copilot aside, it has all the basic tools useful to navigate inside logs, metric and traces, a strong orientation to Kubernetes architecture and offers a proprietary agent to start the data pipeline with almost no configuration.

Testing

Locust has proven to be an exceptionally convenient and powerful tool for performance testing, offering both out-of-the-box ease of use and extensive configurability. The tool provides a straightforward setup process, enabling users to quickly create and execute basic load tests. What sets Locust apart, however, is its flexibility in configuring more

advanced tests. The ability to simulate numerous virtual users is a useful feature, and it is highly customizable, allowing testers to control the number of users and the rate of requests. This allows for the testing of various system capacities and load conditions, providing highly accurate insights into performance under different usage patterns. One of its key strengths lies in its ability to easily generate tests from external configuration files, such a HAR. This integration enables the automation of load tests based on predefined paths, simplifying the process of replicating real-world traffic conditions and stress-testing systems in a controlled, repeatable manner.

6.1.1 The autoscaling drama

When discussing microservices, one of the main advantages often highlighted is the ability to scale nodes and pods. While this may seem straightforward and beneficial, the reality is more complex. Scaling itself is a significant challenge with inherent limitations. Although its purpose is to optimize resource usage and enhance performance, a superficial approach could even lead to worse performance and resource wastage.
The first step inside this rabbit hole was a GitHub answer to an issue on k8s hpa[aze21]: the author went through all the experiments and the necessary calculation he did to demonstrate that in order to have a decent hpa autoscaling there are some conditions. The issue was about hpa not scaling down, although there was effectively a decrease in resources request from the container, but this user also brought to light the opposite possibility: hpa continuously changing the number of pods, at the slightest change (a phenomenon also known as flapping). To stabilize autoscaling without having unnecessary pods two settings are needed: set the limit on CPU usage (not on memory usage) and set currentMetricValue $\times$ 2 $\leq$ desiredMetricValue, where the metric should, indeed, be the CPU usage. Another blog post with a deeper study on hpa problems and some possible workarounds and solution is the third and final part of a three-part series on k8s hpa on Medium[Sek20]: here the author investigates different areas of hpa, underling its limits (the potentials have been explained in previous chapters of this blog series, with the cited one serving as the concluding chapter.). The article lists different aspects and solutions, here I'll try to report what mostly concern this thesis.
Normally, hpa takes some time to duplicate a pod (autoscaling lag) and with a high desired metric value, there is the chance that the pod hits the 100% of the metric usage before creating a replica. An immediate solution is to lower the desired metric value, but

this option will create more pods than really needed, leading to higher costs. This is the first compromise to accept: finding the best balance between pods replicas and expense savings. Another limit of hpa is the aggregation of the metrics: usually the resource usage is taken on a time span (like 30s) and this option leads to the inability to detect usage spikes, where the high resource requests lasts for a smaller time span. The solution here is to increase the metric resolution, but this choice leads to a greater observability overhead. Again, there are compromises to accept.

More aspects are well explained in the article and the conclusion is a hard pill from the author:

> Enabling HPA is not the same as having a working autoscaling solution[Sek20].

This quote leads directly to another paper on the topic, where the authors proposed a reactive autoscaling algorithm: "PROMETHEE is a multi-criteria decision algorithm that enables the establishment of an outranking between various alternatives"[MCD24, p. 4]. This algorithm is based on five different criteria: the number of allocated CPUs, the size of allocated memory, the size of allocated storage, energy consumption, and number of running Kubernetes objects. Based on these metrics, Promethee decides which node has to be scaled up (or down) in order to maximize (or minimize) the specified criterion, based on a preference function to be given by developers.

One last mention goes to another team that used different ML approaches (SVM-Linear, SVM-Poly, MLP and Linear Regression) to predict the CPU usage by user load[RT24]. In their study they started by analyzing the behavior of the system in order to gather some data. As case study, they had the same of this thesis' project: Google Online Boutique in a Kubernetes cluster with Locust as performance testing tool, observability instrumented via OTel, Prometheus and Grafana. They gave same data to each algorithm and tested their result with different user request loads (from 2000 to 10000 users). Then, they used Locust to measure the real CPU usage on user load and see which ML approach gave the minimum error on prediction. Turns out MLP had the lowest error on prediction, giving the more realistic metrics in advance, metrics used to tune a better hpa and save resources.

6.1.2 Data correlation

All the tool that have been presented offers many options for data correlation, at least queries and graphs allows users to describe the system under the convenient aspects and track the state with the desired granularity. But having only these features would relegate them to the monitoring stage, inside the *known unknown*. All of them, instead, had the options to observe dependencies, error chain, service API calls and much more, one linked to another. Not just the opportunity to always go deeper in each metric or trace details, but also to see what has been impacted and where it comes from. NewRelic has the more obvious example: while exploring a metric, it proposes related metrics, based on the name: if I'm analyzing the `k8s_container_cpu_request` metric, it shows all the k8s metrics and the CPU related metrics. Or, in Lumigo, traces are well explained not only with their source, destination duration and other basic information, but it also provides the series of spans that the trace has generated and all their information.
This approach allows easier discover of error root cause or resources usage. On this last topic, NewRelic has a dedicated option to monitor resource costs, if the data pipeline is managed by NewRelic's agent.
Another possible approach in data correlation and root cause finding could have been using span pattern recognition[Bel+24], where most frequent patterns are marked as correct, and any deviation from the pattern is considered an error. In the project the team recognized patterns and sub-patterns as normal state of the system and detect any possible error with this analysis. It turns out that

> The use of sequential pattern mining enables the isolation of anomalous behavior patterns and facilitates the identification of their root causes[Bel+24, p. 46].

6.2 Conclusion and future work

This project encompassed multiple broad objectives, each of which could have constituted an independent research endeavor. However, the unifying concept was the development of a complete data pipeline within specific constraints. The study highlights how observability is a rapidly evolving field: during the course of the project, numerous new features were released in OpenTelemetry and various observability tools. As the project progressed, it expanded significantly, to the extent that I chose not to focus on

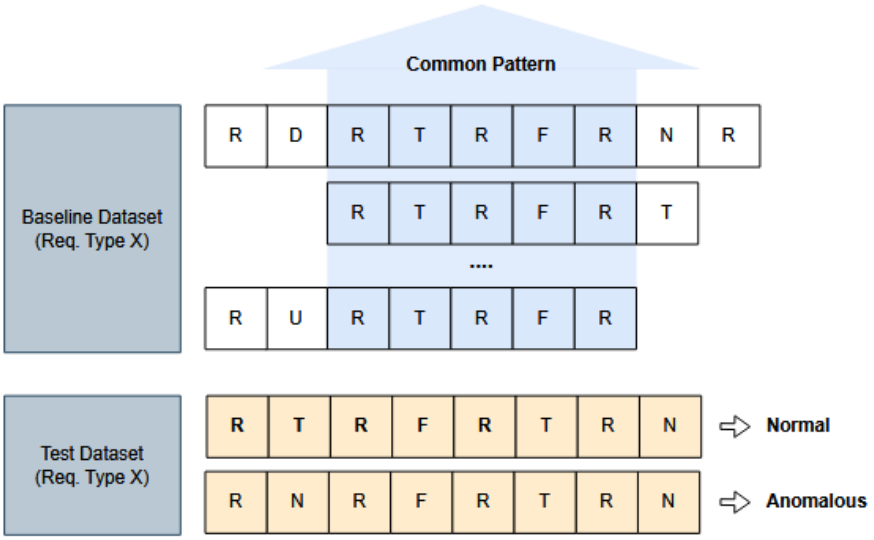


Figure 6.1: An example of a possible error in pattern recognition model.

Source: [Bel+24, p. 43]

sharing extensive results on the JPet Store, instead prioritizing the analysis of a more complex system. The results demonstrate that modern observability solutions enable a deep understanding of system behavior. Furthermore, once an observability tool is selected based on system requirements, the use of multiple different tools does not appear to offer significant advantages.

In this project, I analyzed and tested various approaches, revealing certain recurring patterns. Enterprise-level observability tools typically offer out-of-the-box configurations, with agents that can be easily deployed in a Kubernetes environment, automatically transmitting data with minimal setup. However, efficiency requires effort: achieving comprehensive and precise observability instrumentation—while maintaining sensitivity to overhead—necessitates architectural planning and deliberate choices regarding what should be observed. For medium-to-large applications, I recommend using an open-source, vendor-agnostic telemetry solution like OpenTelemetry. It offers a high degree of customization (beyond what standard tool agents provide) and facilitates seamless switching between different backends, as demonstrated in my thesis. That said, an enterprise-level solution like Grafana's all-in-one platform provides better support and stability but enforces reliance on a single tool, making future transitions more challenging over time.

This project primarily focuses on observability within Kubernetes-based microservices applications, where the analysis has been most detailed. In such a system, I was able to

visualize key metrics, such as the service handling the highest number of requests and the overall service map, which are fundamental insights available in any observability tool. Even these basic observations can inform critical decisions, such as determining which services should have the highest resource limits and which should maintain minimal resource usage before scaling.

The project is fully documented and reproducible, thanks to the use of open-source, free-tier components. It can serve as a foundation for various observability-related studies, whether focused on telemetry collection or system scaling.

Future developments could involve deeper exploration of specific components within the observability pipeline. Potential directions include implementing custom telemetry solutions, enhancing the collector to modify, filter, or discard specific data, or conducting an in-depth analysis of a single backend, evaluating its full range of features, advantages, and limitations. The breadth of topics covered in this project inevitably resulted in a lack of deep exploration in certain areas, which could be addressed in future research. Another compelling direction would be a “meta-analysis” of the collector itself, investigating its scalability and overhead impact.

Acknowledgements

At the end of this journey, I can say that I did not walk alone.
First and foremost, my deepest thanks go to Moira: my wife-to-be, my friend, my main supporter, and my family. I also want to express my gratitude to the family that has been with me every step of the way: thank you, Mom, Dad, Antonio, Anna, and Alessandra, and thank you for the wonderful family we are building together.
At UniPD, I am grateful to have found true friends. Without them, these years would not have been the same (and probably, some exams would have delayed me even longer). I'd like to mention them in no particular order: Francesco, my brother in Christ, Lorenzo, Riccardo, and Marco. A huge thanks to many others who have helped me throughout this adventure: none of you were ever taken for granted, and each of you is a blessing from God.
Last but not least, thank you, Dr. Di Buccio, for your understanding and patience. Thank you, Roberto, for believing in me and for all the support you've built around me at Moviri. You made this possible.
Finally, I would like to thank God for the life He has given me and for the people He has placed by my side.

Bibliography

[90] "IEEE Standard Glossary of Software Engineering Terminology". In: *IEEE Std 610.12-1990* (1990), pp. 1–84. DOI: 10.1109/IEEESTD.1990.101064.

[AF10] Martin L. Abbott and Michael T. Fisher. *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise*. Addison-Wesley Professional, 2010.

[AI-25] AI-Simplified. *Distributed Systems and Microservices: Guide to Streamlined Software Architectures*. 2025. URL: https://medium.com/@AI-Simplified/distributed-systems-and-microservices-guide-to-streamlined-software-architectures-2782673e5808 (visited on 03/25/2025).

[Aru24] Udensi Fortune Arua. *The Role of the CI/CD Pipeline in Cloud Computing*. 2024. URL: https://www.civo.com/blog/the-role-of-the-ci-cd-pipeline-in-cloud-computing (visited on 03/23/2025).

[aze21] azeer-esmail. *Autoscaling issue in Kubernetes*. 2021. URL: https://github.com/kubernetes/kubernetes/issues/78761#issuecomment-1075814510 (visited on 03/07/2025).

[Bel+24] Adel Belkhiri et al. "Towards Efficient Diagnosis of Performance Bottlenecks in Microservice-Based Applications (Work In Progress paper)". In: *Companion of the 15th ACM/SPEC International Conference on Performance Engineering*. ICPE '24 Companion. London, United Kingdom: Association for Computing Machinery, 2024, pp. 40–46. ISBN: 9798400704451. DOI: 10.1145/3629527.3651432. URL: https://doi.org/10.1145/3629527.3651432.

[Cro24] CrowdStrike. *Incident Root Cause Analysis*. 2024. URL: https://www.crowdstrike.com/wp-content/uploads/2024/08/Channel-

File-291-Incident-Root-Cause-Analysis-08.06.2024.pdf (visited on 12/29/2024).

[FL14] Martin Fowler and James Lewis. *Microservices: a definition of this new architectural term*. 2014. URL: https://www.martinfowler.com/articles/microservices.html (visited on 01/01/2025).

[GFL24] Ian Gorton, Liz Fong-Jones, and Alf Larsson. "Observability Q&A". In: *IEEE Software* 41.1 (2024), pp. 50–54. DOI: 10.1109/MS.2023.3330234.

[Kos00] Raph Koster. *The History of MUDs*. 2000. URL: http://www.laputan.org/mud/ (visited on 01/01/2025).

[MCD24] Tarek Menouer, Christophe C□rin, and Patrice Darmon. "Reactive Autoscaling of Kubernetes Nodes". In: *Proceedings of the 4th Workshop on Flexible Resource and Application Management on the Edge*. FRAME '24. Pisa, Italy: Association for Computing Machinery, 2024, pp. 31–38. ISBN: 9798400706417. DOI: 10.1145/3659994.3660310. URL: https://doi.org/10.1145/3659994.3660310.

[Nev24] George V. Neville-Neil. "Building on Shaky Ground: We owe it to the world to make systems work safely and reliably." In: *Queue* 22.5 (Dec. 2024), pp. 12–18. ISSN: 1542-7730. DOI: 10.1145/3703127. URL: https://doi.org/10.1145/3703127.

[Par25] AKF Partners. *The Scale Cube*. 2025. URL: https://akfpartners.com/growth-blog/scale-cube (visited on 01/01/2025).

[RF06] Mark Richards and Neal Ford. *Fundamentals of Software Architecture. An Engineering Approach*. Microsoft Research, 2006. URL: https://mrce.in/ebooks/Software-Fundamentals%20of%20Software%20Architecture.pdf (visited on 01/01/2025).

[Ric18] Chris Richardson. *Microservices Patterns*. Manning Publications, 2018.

[RT24] Adam Rubak and Javid Taheri. "Machine Learning for Predictive Resource Scaling of Microservices on Kubernetes Platforms". In: *Proceedings of the IEEE/ACM 16th International Conference on Utility and Cloud Computing*. UCC '23. Taormina (Messina), Italy: Association for Computing

Machinery, 2024. ISBN: 9798400702341. DOI: 10.1145/3603166.3632165. URL: https://doi.org/10.1145/3603166.3632165.

[Sek20] Sasidhar Sekar. "Autoscaling in Kubernetes: Why Doesn't the Horizontal Pod Autoscaler Work for Me?" In: *Medium* (2020). URL: https://medium.com/expedia-group-tech/autoscaling-in-kubernetes-why-doesnt-the-horizontal-pod-autoscaler-work-for-me-5f0094694054 (visited on 03/07/2025).

[Sho16] Randy Shoup. *The Evolution of Microservices*. 2016. URL: https://learning.acm.org/binaries/content/assets/leaning-center/webinar-slides/2016/evolutionofmicroservices_webinarslides.pdf (visited on 01/03/2025).

[Som16] Ian Sommerville. *Software Engineering*. 10th. Pearson, 2016.

[SS24] Stephanie Susnjara and Ian Smalley. *CI/CD Pipeline*. 2024. URL: https://www.ibm.com/think/topics/ci-cd-pipeline (visited on 01/03/2025).

[Usm+22] Muhammad Usman et al. "A Survey on Observability of Distributed Edge & Container-Based Microservices". In: *IEEE Access* 10 (2022), pp. 86904–86919. DOI: 10.1109/ACCESS.2022.3193102.

[W3T22] W3Techs. *Usage Statistics of Operating Systems for Websites*. 2022. URL: https://w3techs.com/technologies/overview/operating_system (visited on 02/28/2025).

[Wal25] James Walker. *Laws of Computer Programming*. 2025. URL: https://www.cs.oberlin.edu/~jwalker/humor/lawsOfComputerPrograming.html (visited on 01/06/2025).

[Wik24] Wikipedia. *2024 CrowdStrike-related IT outages*. 2024. URL: https://en.wikipedia.org/wiki/2024_CrowdStrike-related_IT_outages (visited on 12/22/2024).

[Wik25] Wikipedia. *Conway's Law*. 2025. URL: https://en.wikipedia.org/wiki/Conway's_law (visited on 01/01/2025).