Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea Magistrale in Informatica

Math Department - Master Degree in Computer Science

Implementation and evaluation of a container-based software architecture

Master's thesis

Supervisor

Prof. Tullio Vardanega

Student

Alberto Simioni - 1106663

Academic Year 2016-2017

Abstract

Recent advances in fields such as Cloud Computing, Web Systems, Internet of Things and Distributed NoSQL DBMS are enabling the development of innovative enterprise information systems that significantly increase the productivity of end users and developers.

The aim of this thesis is to explore the new opportunities that these new technologies are bringing to the enterprise world. The new opportunities are explored by investigating the scenario of a medium-sized worldwide-trading company, Fiorital S.p.A.

The thesis presents the design of a software architecture for the future information system of the company. The architecture is based on the usage of the Container technology and of the Microservice architectural style. Containers have empowered the usage of Microservices architectures by being lightweight, providing fast start-up times, and having low overhead.

Candidate technologies for the implementation of the proposed software architecture are singled out, and the selection rationale is presented. This thesis provides an evaluation of both the candidate architecture and the technologies through the implementation of a prototype and the application of synthetic workloads that mimic stressful use scenarios. The results show that, in spite of the relative immaturity of some of the candidate technologies, the information system's candidate architecture is appropriate and that a company like Fiorital would considerably benefit from it.

Acknowledgements

I would first like to thank my family for the continuous support received throughout my years of studying. This accomplishment would not have been possible without them.

I would like to thank my thesis supervisor Prof. Tullio Vardanega for his feedbacks and cooperation during the thesis work.

Finally, I would like to thank my friends for accepting nothing less than excellence from me.

Padua, July 2017 Alberto Simioni

Contents

List of Figures

List of Tables

Chapter 1

Problem statement

The motivation behind my thesis work originates from a project that the University of Padua, specifically the Computer Science course, is conducting together with the company Fiorital S.p.A.

Fiorital is a company of approximately 300 people, that works on the import and export of fish. They work with many partners that are distributed around the globe. The company has been growing a lot during the last years and they want to continue to grow. The core business of the company is not related to ICT. But nowadays, almost all the companies need to be up to date with the opportunities that Information Technology (IT) is bringing. Fiorital is aiming to improve its business processes by leveraging the possibilities that the current-state IT technologies are offering. The company wants to use IT to make the tasks of its employees more efficient by supporting their operations, by managing the company-related information and by providing insights into the decision-making processes.

To achieve these goals Fiorital has contacted the University of Padua. Fiorital and the University decided to start a project together. In this project, the university helps Fiorital in the designing of an innovative information system. The name of the project is Smart Enterprise (SE) project.

To illustrate the project, the chapter starts by describing the company Fiorital S.p.A. and their business activities. Having an idea of the company's activities helps understanding the motivations behind the SE project. The project requirements are tailored to the specific needs of the company. Later, the chapter describes the SE project and what is my role in the project.

1.1 The company

Fiorital S.p.A is a company founded in 1979 and based in Venice. They perform import and export activities in the fish market. They operate mainly with wholesalers and mass retail channels.

The company deals with different providers and transportation companies. They import from providers scattered around the globe and they sell to customers in the Italian and European market. According to different needs, the products are transported directly to the clients or are processed and packaged at the company's headquarters in Venice, after which they are shipped to reach Italian and European consumers. Figure 1.2 shows a representation of the providers from which Fiorital imports and the means of transport that are used to bring the goods to the headquarters. It's essential to

Fiorital to utilize the JIT (Just-In-Time) methodology in the majority of the activities performed. This methodology is necessary given the type of goods that are traded by the company: fish usually has the necessity to arrive at destination in few days in order to maintain its freshness.

Figure 1.1: Fiorital logo [1]

At the headquarters in Venice Fiorital has a production division for the processing and production of ready-to-cook and ready-to-eat fish products for the retail sector.

Besides the import and export operations of the company, Fiorital has recently started to perform Business to Consumer (B2C) activities. The company has opened a restaurant where the customers can taste the fish that comes directly form the Venice headquarters. Fiorital has also opened a corner store inside a supermarket chain, where the customers can buy and eat the ready-to-eat food that the company packages at its production department.

1.1.1 Company structure

The company is divided into different departments that are responsible for different tasks. The following list describes the main departments and their functions:

- **Purchasing department**: deals with the management of the purchasing orders of fish from the providers. The providers of Fiorital are very numerous. Therefore, the department is subdivided into different divisions that manage different subsets of the providers depending on the providers' locations;
- **Logistic department**: deals with the planning and the control of the transportations of the fish products. The transportations are related to the purchasing orders and the sales orders. The department is divided into two divisions, one that manages the inbound transportations, the other manages the outbound transportations;
- **Sales department**: deals with the management of the sales orders of fish from the customers. The customers of Fiorital are very numerous. As for the purchasing department, the sales department is subdivided into different divisions each managing a different subset of the products that come from different zones of the world;
- **Loading and unloading platform**: deals with the warehousing of the products at the Venice headquarters. The platform situated at the headquarters is not a real warehouse, it's a small warehouse where the products should remain for a short time. To preserve the freshness of the products, it's essential that the fish doesn't remain on the platform for a long time;

- **Quality control**: this department performs quality assurance checks on different activities of the other departments. The operators control the compliance of the products and activities against different quality standards;
- **Administration**: this department deals with the management of the payments and invoices related to purchasing/sales orders and the transportation companies.

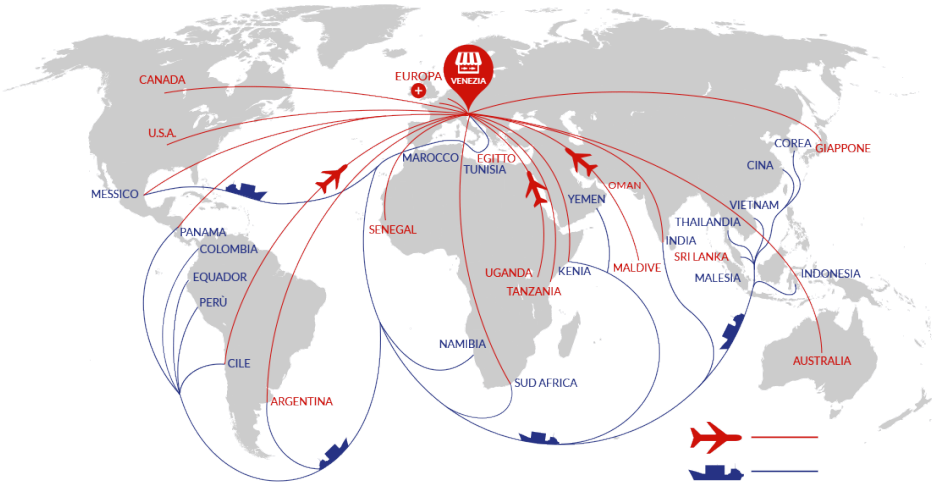

Figure 1.2: Import and export routes of Fiorital [2]

The company Fiorital S.p.A. has a structure that is typical for a company that does import and export of goods and direct sales to clients.

1.2 The Smart Enterprise Project

This section provides an overview of what Fiorital together with the University wants to obtain with the implementation of the SE project.

The project started in June 2016 and it's still ongoing. Several master students and three professors of the University of Padua have worked and are currently working on the project.

The main motivation behind the SE project is the will of Fiorital to renovate its IT system. The current IT system of the company is deployed in a data center within the company. The data center has different software programs installed that compose the information system of the company. Operators working in different departments have the need to exchange information among each other. Many times, the information coming from one department is the input for all the activities that another department has to perform. The programs that the operators use aren't designed to exchange data among each other because each of them was developed for helping a specific business process of the company. As the information system grew, the company realized the necessity to make these programs cooperate and exchange information.

This fact causes different problems for the operators of the company. Operators in different departments don't use the information system to communicate, instead, they communicate face-to-face or using phone calls. One consequence is that much informa-

tion is not registered inside the IT system and it's hard to retrieve the motivations of some past business decision.

Furthermore, the company is rapidly growing and changing its business processes, whereas, the information system wasn't updated while these changes were happening. The extension of the system with new features is tricky. Integrating new features into the system is a complex operation, the system is composed of proprietary programs and it is not simple to integrate it with new features. Furthermore, updating the production environment of the system is a long manual operation.

All these facts highlighted the need for a new IT system that should be more flexible and provide better features to the company's operators.

In addition to the problems just mentioned, there are other considerations that are encouraging the renovation of the system. Numerous trends in the IT sector are bringing recent technologies that are driving innovation inside the companies. Some of these trends are Cloud Computing, Internet of Things (IoT), Machine Learning, Robotics and so on. Fiorital, with this project, wants to explore and understand which are the opportunities that these recent technologies could bring for them. It's fundamental for a company like Fiorital to explore these opportunities to continuously improve its business processes by making them more efficient and effective.

1.2.1 Project Objectives

The previous section described which are the motivations that pushed the company in the creation of the new information system. This section explains which are the main goals of the project. Goals that can be achieved by implementing a new information system. In particular, the section describes what features and what characteristics the new information system should provide and have.

An information system is the Information and Communication Technology (ICT) that an organization or a company utilizes, and also the way in which the human operators interact with this technology in support of business processes. The main components of a computer-based information system are: hardware, software, data, networks and procedures.

The following list presents the objectives of the project by describing the features and characteristics of the new information system:

- **Versatile**: the company has many departments that perform various business processes and activities. The company has a complex structure that will likely continue to change in the future. Therefore, the departments have many different requirements, the information system should be able to satisfy the majority of these requirements even the ones that may arise in the future. There are different causes that may originate the showing up of new requirements. Figure 1.3 shows which are these causes. Fiorital may change its organizational structure or its business processes in the future. In these cases, the system should be updated in order to represent the changes by adapting the features that the system provides. Fiorital may also change its business strategies, in this case, new types of data analyses may be required. Lastly, the data in which the company is interested in are likely to change frequently;

- **User interface accessible from different devices**: the goal is to allow a large number of devices types interacting with the system. A user may use his phone, tablet, laptop and/or a desktop workstation to interact with the information system. In this way, the operators of the company are not restricted only to their

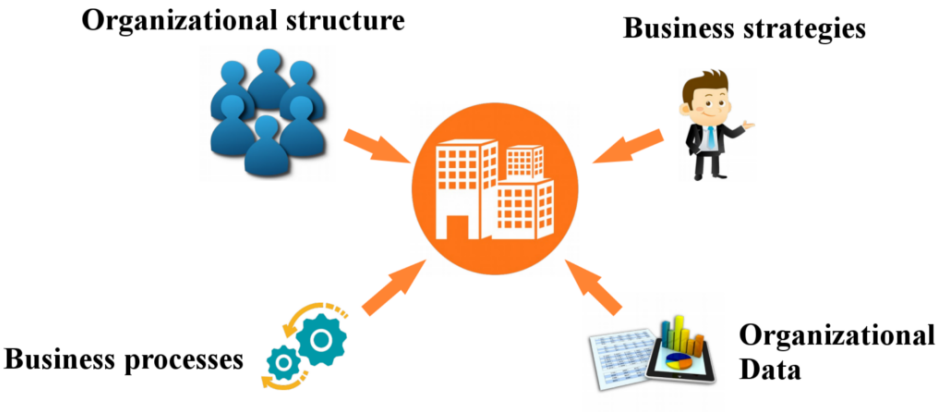

Figure 1.3: Factors that may cause requirements changes on the information system.

workstations and they can interact with the system also when they are not in their offices and even when they are outside of the company's headquarters;

- **Easily accessible information**: Fiorital wants to reduce the number of operations that an operator should perform in order to access the information relevant for him. An operator working in a certain department is interested only on a subset of the whole information and of the features that system provides.

  The user interface should be designed so that an operator sees only the data and operations that may be useful for him. The system decides what to show to the operator based on the role of the operator and on his previous actions.

  Moreover, the system gives the operator the possibility to receive live updates about events that happened inside the information system. These events may be caused by the actions of other operators or by the interaction of the information system with external information sources. This feature reduces the number of operations that a user performs to retrieve some information.

  Without having live updates a user has to continuously check for the presence of recent updates, causing an important waste of time. Also, the user notices the presence of the update later than with the live update feature.

  The user may be provided with a dashboard where he can see the live updates in real-time and where he can select which events and information should be presented on the dashboard;

- **Integration with new sources of information**: the SE project aims at obtaining a system where new data sources can be easily added. An example of data source that can be added to the system is the information generated by sensors connected to a newly-installed machinery inside the company production department. Another example is the information about the flight schedules of a particular airline company that Fiorital may use to transport the goods;

- **Operators' tasks automation**: this objective concerns the automation of certain tasks that the operators perform at the platform and production departments

inside the Venice headquarters. There are situations where an operator has to manually insert information into the system.

An example is the weighing of the pallets that are being loaded and unloaded from or into a truck. Currently, the operator uses a balance to weigh the pallet and then he manually inserts the information through the user interface of the current system. A more reasonable approach would be that the balance has computing and networking capabilities in order to send automatically into the system the information;

- **Real-time analyses**: the type of data analyses that are usually performed by an enterprise information system are Business Intelligence (BI) analyses. BI technologies provide historical, current and predictive views of business operations. The analyses are performed periodically, each week, month or year and they can be used to support a wide range of business decisions ranging from operational to strategic.

  One objective of this project is to have the system performing real-time analyses that can be directly useful for the business processes of the company.

  One example is the automatic generation of products' catalogs for the Fiorital's customers. The catalogs can be calculated using the information about the customers' past purchasing orders and the live information about the current products' availabilities. This type of analyses needs to be performed in real-time because the information about the products' availabilities is available during the execution of the operators' business activities. The operators may retrieve this information by contacting the Fiorital's providers. As soon as the availabilities information is present in the system, the catalogs should be generated. Figure 1.4 shows the information flow for the scenario just described.

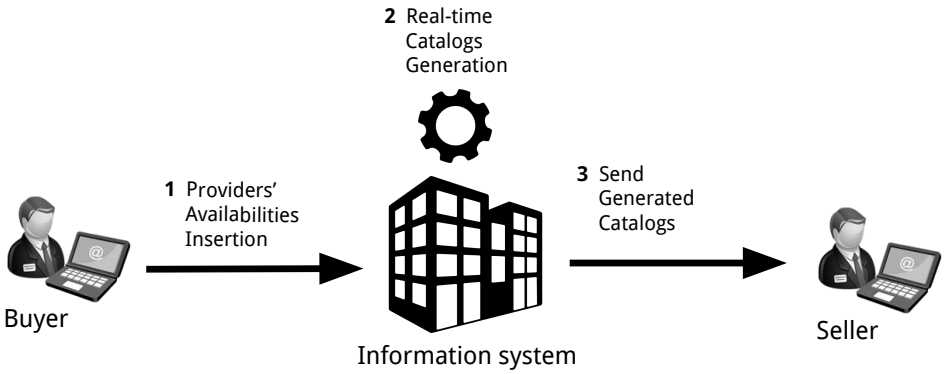

Figure 1.4: Catalogs generation flow

Another example is the automatic creation of an unloading plan. Unloading plans concern the unloading of pallets from a truck. The unloading plan specifies where the pallets should be positioned inside the platform. Automatically generating an unloading plan has the benefit of optimizing the space usage of the platform, increasing the total amount of pallets that can be stored. Another benefit is to position each pallet in a good place based on which is the next operation that should be performed on the pallet.

There are other scenarios where this type of real-time analyses could be useful. One objective of the project is to identify these scenarios and implement algorithms that compute the results in short time. These algorithms may be implemented through the usage of Machine Learning techniques and Combinatorial Optimization techniques.

Table 1.1 lists the set of objectives and places an identifier on each of them.

Project's objectives
O1 - Versatile
O2 - User interface accessible from different devices
O3 - Easily accessible information
O4 - Integration with new sources of information
O5 - Operators' tasks automation
O6 - Real-time analyses

Table 1.1: Project's objectives list

Three professors of the University of Padua are working on the SE project. One professor is responsible for the design of the system architecture. One professor is working on the real-time analyses that require Machine Learning techniques for the implementation. The last professor is working on the real-time analyses that require Combinatorial Optimization techniques for the implementation. One employee of Fiorital, a former student at the University of Padua, is responsible for the analysis of the business processes of the company and the coordination of the project between Fiorital and the University. The analysis he is conducting is useful for the people of the university that need to understand the business context of Fiorital in order to perform their activities.

1.3 My role in the project

This section explains which is my role within the project and what I want to achieve with my thesis work.

My thesis work concerns the design of the system architecture. I've carried out the work under the supervision of the Professor Tullio Vardanega. The goal is to design an architecture that satisfies the objectives of the project that I described in section 1.2.1. Even though these objectives may look simple at a first glance, there are many aspects that require a careful design of the architecture. The system should be accessible from any kind of devices, should allow the execution of complex real-time algorithms, the architecture should be as much agile as possible in order to allow easy modifications and integrations and the performance should never degrade. A more detailed list of features that are fundamental to the system architecture is presented in section 2.1.

1.3.1 Thesis objectives

The thesis work aims at finding the best architecture for the specific use case of Fiorital. The architecture, as shown in chapter 2, uses technologies that were published recently and their usage for the development of new enterprise systems hasn't been studied much in the academic research.

One objective of the thesis is to study and evaluate the technologies that can be useful for the Fiorital use case. This document presents a comparison of different technologies showing in which contexts they are useful for the implementation of the new information system.

Another goal of the thesis is to design the system architecture and to find a set of architectural properties that are useful to evaluate the effectiveness of the architecture. The design and the discussion of the software architecture are based on this set of architectural properties.

The IoT architecture of the system is an important part of the system. The goal *O5 operators' tasks automation* would clearly benefit from the presence of IoT devices inside the headquarters of Fiorital. The network architecture of the IoT devices and which sensors and actuators to use, are aspects that should be carefully designed. However, the system IoT infrastructure is not part of the thesis work, due to time-constraints. The architecture designed within the thesis work has to allow the extension and integration of the system with the IoT part.

The last goal of the thesis is to implement a prototype of the designed architecture in order to create a set of results that shows the effectiveness of the architecture. The prototype should be an implementation of the designed software architecture that represents the specific scenario of Fiorital.

The results will be based on a set of metrics. The chosen metrics give quantitative estimations of the architectural properties used to evaluate the system architecture.

1.4 Outline

The rest of the document is organized as follows. Chapter 2 discusses the architectural properties that the designed system architecture should have. Then, it describes the designed candidate architecture, explaining how it should be able to satisfy the architectural properties. Eventually, the chapter discusses which are the technologies that could be useful for the implementation of the system and presents the chosen technologies.

Chapter 3 presents an experimental evaluation of the architecture and of the chosen technologies. It describes the implementation of the prototype and presents a set of results based on the set of metrics that evaluate the candidate architecture.

Chapter 4, the last chapter, summarizes the outcomes of the work and provides insights for future improvements.

Chapter 2

Candidate architecture and technologies selection

The main objective of the thesis is to design a proper architecture for the new information system of the SE project.

This chapter starts by giving a brief explanation of what is a software architecture and by introducing a bit of terminology that is useful in the following sections and chapters.

In their book, K. Bittner and I. Spence [3] give a simple and precise definition of what is a software architecture:

"Software architecture encompasses the set of significant decisions about the organization of a software system including the selection of the structural elements and their interfaces by which the system is composed; behavior as specified in collaboration among those elements; composition of these structural and behavioral elements into larger subsystems; and an architectural style that guides this organization. Software architecture also involves functionality, usability, resilience, performance, reuse, comprehensibility, economic and technology constraints, trade-offs and aesthetic concerns."

The definition describes a software system as composed of structural elements and of their interfaces. In the rest of the document these structural elements are called as the *components* of the architecture. Whereas, the term *interface* doesn't change. Another term that is used in the rest of the document is *architectural style*. An architectural style is a family of architectures that share certain characteristics. There are different type of styles. More than one style could be used to design a software architecture. When starting to design a software architecture it's crucial to decide which styles should be used.

The next section presents a set of architectural properties that are central for the design of the information system. The section explains why these properties are fundamental to have for an architecture that wants to satisfy the project's objectives presented in section 1.2.1.

These properties are crucial also for the implementation and the evaluation of the candidate architecture. The results presented in 3 measure the prototype's compliance to the architectural properties.

After presenting the architectural properties, the chapter presents the candidate architecture of the system and a discussion about the technologies that have been chosen for the implementation.

2.1 Architectural properties

The chosen set of properties are those that are extremely hard to achieve for an information systems implemented with traditional technologies, rather than all the properties that an information system should have. Therefore, the following list of properties is not a complete list of architectural properties that an information system should have, but rather, a set of properties commonly hard to achieve and that can bring an important improvement to the system.

Each property may concern the service provider or the application user or both. The properties are decomposed into various sub-properties. Each sub-property concerns only a single actor:

- **Responsiveness**: refers to the specific ability of a system or functional unit to complete assigned tasks within a given time [4]. This property aims at assuring that the system continues to have the normal response times also when the load increases. The users should not experience latencies due to the increased usage of the system. The high load is caused by the requests performed by the company's operators, by third-party systems and by IoT devices. The workload experienced by the system may vary during different time intervals. For example, Fiorital has more operators that are working during some specific days of the week, because the customers' requests of fish are higher in those days. Moreover, each year, the sales of fish increase a lot during the second half of December. The system has to take into account the following aspects that create workload variability:

  - The number of provided software services can increase and decrease. Furthermore, a service generates and receives different amounts of requests over time;
  - The amount of analysis on the data varies over time;
  - The number of users of the system;
  - The number of IoT smart things that compose the system varies over time. Consequently, the information they generate over time varies as well.

  The system's users shouldn't experience any performance degrade, they should be able to perform the usual operations with the usual execution times.

  Generally, a system, that aims in handling a high number of requests without experiencing any latency, needs to use more hardware resources to compute the requests. It is possible to overprovision the system with a large amount of hardware resources to maintain the responsiveness of the system. However, hardware resources are expensive both to buy and to keep in execution (electricity and cooling devices). To reduce the costs, the usage of the computing resources should be optimized. A component of the system should utilize an increased amount of hardware resources, only when the workload is too high for the currently available hardware resources. A component of the system should always try to optimize the usage of the computing resources.

  It should now be clear how the system showld be responsive, but at the same time it should be cost-efficient, maximizing the utilization of the hardware resources.

  The responsiveness property can be subdivided into the following three sub-properties:

– **Service Level Agreement (SLA) Preservation**: this property aims at maintaining the service level for the application user of the service. Therefore, this property concerns the application user. A user wants guarantees that the system always provides the same desired performances, even when the workload is high. The guarantees provided by the SLA concern the response times of the user's requests. The response time of a user request should never cross a certain threshold.

  This property is fundamental to an information system of a company. The operators of Fiorital need to use the system for their business activities. Having a system with bad response times leads to a waste of time for the company's operator and consequently to an economic loss for the company;

– **Queue Length Minimization**: the users of the system perform concurrently various requests that are received by the system. Each component of the system can execute various requests in parallel. Anyway, the number of requests that a component can execute in parallel is limited, for example by the Central Processing Unit (CPU) power available to the component. Consequently, the system should have a set of queues where to store the requests that are waiting to be executed. A request that waits for a certain amount of time inside a queue and then is executed returns its results later than a request that is executed immediately without being stored in a queue. Bigger is the size of the queue when the request arrives, more is the time that a request spends being inside a queue and consequently increases the response time of the request. Figure 2.1 gives an abstract representation of a requests queue where four requests are waiting to be executed. The component executing the requests is the CPU and it is occupied executing other requests.

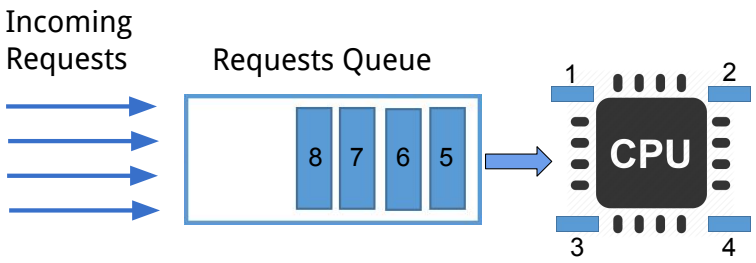

Figure 2.1: Representation of a requests' queue.

  This property concerns mainly the service provider. An application user is not interested in requests' queues and shouldn't even be aware of their existence. While the service provider should minimize the length of the queues to provide better response time for the users' requests;

– **Throughput Maximization**: throughput is the number of events that take place in a given amount of time. Each component that manages the requests should be as efficient as possible to maximize the usage of the hardware resources. A consequence of maximizing the throughput is that the overall costs for the the hardware resources are reduced. Therefore, the system is more cost-efficient. Also this property concerns the service provider that wants to reduce the expenses for the IT infrastructure. The

application user of the service is totally not interested in the cost of running the hardware infrastructure.

Figure 2.2 shows graphically the subdivision of the responsiveness property in its sub-properties. Each one has a pertinence of concern relative to one between the user and the service provider. The users of the system are the company's operators. The service providers of the system are the developers and the IT staff of the company.

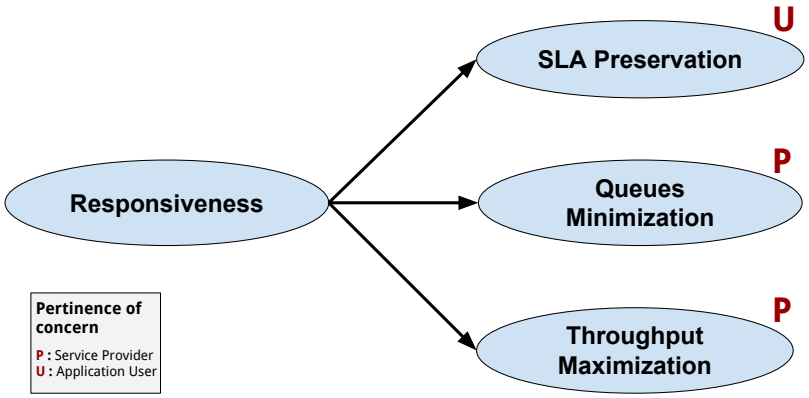

Figure 2.2: Responsiveness decomposition into sub-properties.

- **Reactiveness**: this property indicates how a system is implemented using an event-driven flow of control, rather than a centralized one.

  With centralized control, one component has the overall responsibility to control, start and stop the other components. There is usually, a control component that takes the responsibility for managing the execution of the other components.

  With event-based control, each component can respond to externally-generated events from other components or from the system's environment. Event-based systems are driven by externally generated events where their timings and the arrival order is outside the control of the components that process the events. The availability of new information drives the logic of the computations forward. Each component propagates the information changes to the other components that are interested in the information that is changed.

  With a reactive approach, the problem that is being modelled is decomposed into multiple discrete steps. Each step executes in an asynchronous and nonblocking fashion, and at run-time, they bind together to produce a workflow execution.

  The Reactiveness property is decomposed into the two following sub-properties:

  - **Asynchronous Communication**: the system architecture is composed of different components. At this stage the system architecture is not already defined. However, we can assume that the components of the architecture are executed on multiple computing nodes and on multiple software processes. The components communicate with each other by sending messages and requests forming a distributed system.

A system becomes more resource-efficient when the components' communication is asynchronous. The communication between components can be synchronous or asynchronous. With synchronous communication, the sender of a message blocks its execution until it's acknowledged of the arrival of the message to the destination. With asynchronous communication, the sender transmits a message to the destination, without blocking its execution.

Asynchronous communication improves the overall responsiveness of the system because the components never stop their execution to wait for an acknowledgement or an answer. Figure 2.3 gives a representation of this aspect. A sample execution flow is represented at the left using synchronous communication and at the right using asynchronous communication. Each column indicates a different component or process. The total execution time for both the approaches is equal but, synchronous, blocking communication (left) is resource-inefficient and easily bottlenecked. The asynchronous approach (right) conserves valuable resources, indeed there is no computational time wasted in waiting for a response.

This property concerns the service provider. Asynchronous communication is a mechanism that the provider utilizes to provide a system that has a better responsiveness;

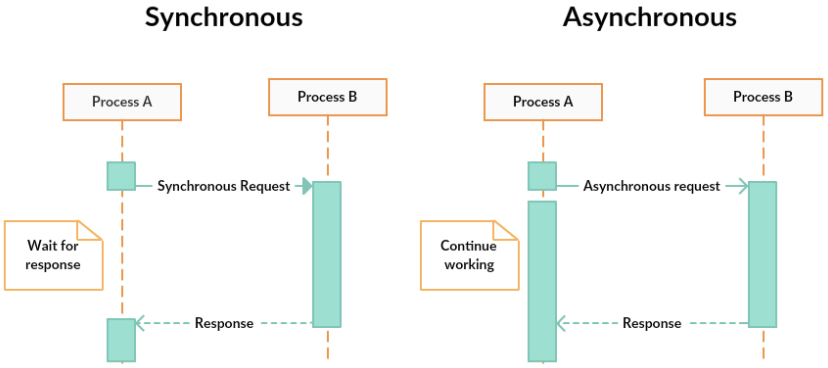

Figure 2.3: Execution flows comparison between synchronous and asynchronous communication.

– **Liveness**: the sub-property indicates that the components of the system are as much as possible up to date about the information that is relevant to them.

The traditional enterprise information systems have relatively awful internal search capabilities. As a result, operators take far too long to find things, despite the enormous IT investments in capturing and storing corporate data, which is then unable to be located.

One component of the system is the control dashboard that the operators use to interact with the system. The operators use the control dashboard to receive new information from the system, to insert data and to send commands to the system. The goal of the liveness property is to have a control dashboard always updated with the most recent information that is relevant to the operators.

There are two project objectives that benefit from having a system with the liveness property. These project objectives, presented in section 1.2.1, are the *Easily accessible information* and the *real-time analyses*. The dashboard is always continuously updated with the newest information. Consequently, the operators don't have to perform any action to find the information they are interested in and they are notified almost immediately about the presence of the new information. Moreover, the results of the real-time analyses are available to the operators as soon as possible.

It should be already clear that this property is valuable from the perspective of the application user. The user profits by having a dashboard that shows the live information that is relevant to him.

Figure 2.4 shows graphically the subdivision of the reactiveness property in its sub-properties;

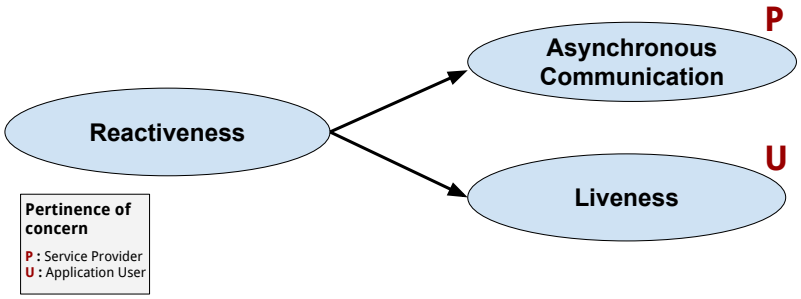

Figure 2.4: Reactiveness decomposition into sub-properties.

- **Agility**: the term indicates the ease in applying changes in the development and production environment of the information system. As shown in 1.2.1 one objective of the SE project is to have an information system that is versatile and that can easily assimilate changes. The team that develops the system, generally, cannot predict how a business will need to evolve over time and therefore they cannot initially build the perfect system. There are many factors that cause the necessity of applying changes to the system. These factors, that arise from the specific needs of the company, were presented in figure 1.3 in chapter 1 and they are: organizational structure, business strategies, business processes and organizational data. Moreover, there are others, more technical, factors that arise the need for applying changes. Examples of these changes are the necessity to update the Operating System (OS)es of the system's computing nodes, the need to fix a set of software bugs, the need to improve the performance or the security of a set of components or the integration of new features.

  Applying updates and maintaining the system is part of the duty of a system administrator. Medium-sized and big-sized companies have different system administrators in their IT departments. The normal operational functions of a system administrator include tasks such as periodic maintenance, updates, and monitoring. These issues must be kept in mind in early stages of planning. When you look at the complete life cycle of an information system, only a modest part of that life cycle is spent building the features of the service. The vast majority of the life cycle is spent operating the system. Yet traditionally, the operational

functions of software are considered lower priority than features, there is general less awareness of their importance. Managing and operating the environment in which the system runs is just as important as implementing the application itself. Configuring the OS, the networks, the database, and web servers is extremely important for an information system to function optimally.

Some of the normal operations conducted throughout an infrastructure life cycle are the following [5] [6]:

- **Account provisioning**: the system administrator adds accounts for new users, removes the accounts of users that are no longer active, and handles all the account-related issues that come up in between (e.g., forgotten passwords). When a user should no longer have access to the system, the user's account must be disabled. All the files owned by the account should be backed up and then disposed of, so that the system does not accumulate unwanted baggage over time;
- **Adding and removing hardware**: when new hardware is purchased or when hardware is moved from one machine to another, the system must be configured to recognize and use that hardware;
- **Performing backups**: it is the system administrator's job to make sure that backups are executed correctly and on schedule. It must be possible to back up and restore the service's data while the system is running;
- **Startup and Shutdown**: the service should restart automatically when a machine boots up. If the machine is shut down properly, the system should include the proper operating system hooks to shut the service down properly. If the machine crashes suddenly, the next restart of the system should automatically perform data validations or repairs before providing service;
- **Installing and upgrading software**: when new software is acquired, it must be installed and tested, often under several operating systems and on several types of hardware. It must be possible for software upgrades to be implemented without taking down the service. Some systems can be upgraded while running, which is riskier and requires careful design and extensive testing;
- **Monitoring the system**: large installations require vigilant supervision. System administrators regularly ensure that email and web services are working correctly, watch log files for early signs of trouble, make sure that local networks are properly connected, and keep an eye on the availability of system resources such as disk space;
- **Monitoring security**: the system administrator must implement a security policy and periodically check to be sure that the security of the system has not been violated. On low-security systems, this chore might involve only a few basic checks for unauthorized access. On a high-security system, it can include an elaborate network of traps and auditing programs.

The operations above mentioned are recurring operations. It's very important to automate as much as possible these operations. The desire for automation is motivated by three main goals: more precision, more stability, and more speed. Other factors, such as increased safety, increased capacity, and lower costs

are desired side effects of these three basic goals. For a system administrator, automating the work that needs to be done should account for the majority of his job. Manual work has a linear payoff. That is, it is performed once and has benefits once. By comparison, time spent automating has a benefit every time the code is used. The payoff multiplies the more you use the code.

For creating automation there are many products, languages, and systems. The capabilities of these tools range from basic to quite high level. Professional administrators spend much of their time writing scripts. Some examples of these tools are [6]:

- **Shell Scripting Languages**: provide the commands one can type at the operating system command-line prompt. It is easy to turn a sequence of commands typed interactively at the Bash or PowerShell prompt into a script that can be run as a single command;
- **Interpreted Scripting Languages**: interpreted languages designed for rapid development, often focusing on systems programming. Some common examples include Perl, Python, and Ruby. They all have large libraries of modules that perform common system administration tasks such as file manipulation, date and time handling, transactions using protocols such as HTTP, and database access;
- **Configuration Management Languages**: domain-specific languages created specifically for system administration tasks.

The information system of the SE project should require a small amount of manual administrative operations for applying changes. Moreover, the system's updates should be applied with a minimal downtime.

The changes that are more problematic are those that involve the communication and the interfaces between more than one component. The system should support run-time reconfiguration of running groups of services and applications. Run-time reconfiguration is essential for the Fiorital's information system that needs to cope with continuous changing requirements. The system shouldn't be switched off to apply the reconfigurations.

A component of the system may receive requests and perform other requests. A set of requests that a component provides is a "provided interface" of the component. While, a set of requests that a component needs to correctly function is the "required interface". Provided interfaces represent the services offered to the component's environment. Required interfaces represents the services the component needs from its environment. These two types of interfaces are building blocks that ease the task of composing various components. Moreover, they make it feasible to replace components in running configurations. A configuration defines how each building block (components and interfaces) is combined to form an architecture describing correct component connections, component communications, interface compatibility and that the combined semantics of the whole system result in correct system behavior [7].

Communication between the interfaces of different components can take place only if the interfaces have been bound. Required interfaces can be bound only to provided interfaces and vice versa. Figure 2.5 shows an example of provided and required interfaces.

I have decided to decompose the agility property in two sub-properties:

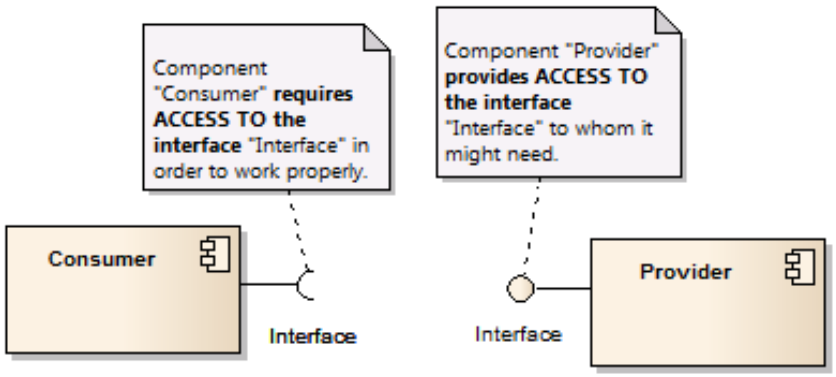

Figure 2.5: Provided and required interfaces explained.

- **Components addition agility**: indicates the ease in applying additions of new components into the system at run-time. To dynamically start a new component, different deployments operations are usually necessary. The goal is to maximize the automation of these operations and therefore reducing the number of manual operations that a system administrator has to perform. Another goal is to minimize the downtime of the system during the addition of the component.

  An important aspect of inserting a new component is the binding between the component's required interfaces and the correspondent provided interfaces that should already be present in the environment. It is necessary that all the required interfaces of the component are already present inside the production environment otherwise, the component will not work correctly. The new component should describe which are the provided interfaces it needs and a set of parameters that indicates the SLA requirements per each interface. The providers of those interfaces should receive the SLA requirements of the new component and adapt their deployments in order to support the new requirements. Operations like dispatching, load balancing and auto-scaling may be needed. The required interface usually has SLA requirements concerning the response times. The new component has also a set of provided interfaces. The component should describe which are these interfaces, in order to make them available to the system's environment;
- **Components removal agility**: indicates the ease in applying removals of existent components at run-time. When a component is dynamically removed from the system, it is necessary that all the components that required the provided interfaces of that specific component have been already updated so that they don't need that interface to correctly work. These operations aims at removing the component's bindings. Also in this case the goal is to maximize the automation of the removal operations and to minimize the downtime of the system during the removal of the component.

Figure 2.6 shows graphically the subdivision of the agility property in its sub-properties.

Both the sub-properties concern the service provider. The application user is not interested in the system's implementation details.

The agility architectural property is useful for achieving three of the project's objectives mentioned in section 1.2.1. These objectives are the *integration with new sources of information* objective, the *versatile* objective and the *operators' tasks automation*. It is simple to add new sources of information or to implement new features to an agile system.

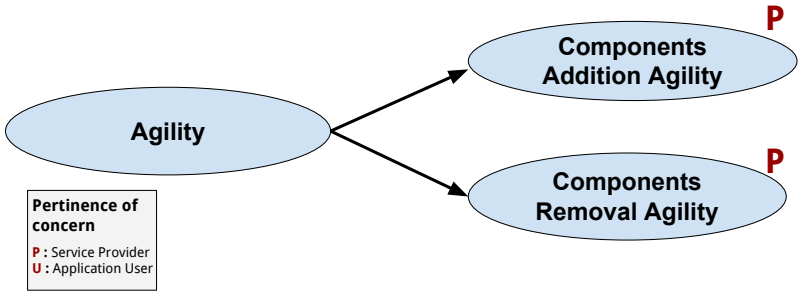

Figure 2.6: Agility decomposition into sub-properties.

Table 2.1 presents a summary of the properties and their decomposition into sub-properties.

Property	Sub-property	Pertinence of concern
P1 - Responsiveness	P1.1 - SLA Preservation	Application user
	P1.2 - Queue Length Minimization	Service provider
	P1.3 - Throughput Maximization	Service provider
P2 - Reactiveness	P2.1 - Asynchronous Communication	Service provider
	P2.2 - Liveness	Application user
P3 - Agility	P3.1 - Components Addition Agility	Service provider
	P3.2 - Components Removal Agility	Service provider

Table 2.1: Properties decomposition summary

In the above list of architectural properties, five out of six project's objectives were mentioned. The project's objectives are listed in chapter 1. A system's architecture that possesses the properties before described, helps in achieving the project's objectives.

Table 2.2 shows a tracing list from properties to project's objectives. The tables shows per each property which are the objectives that the property helps in achieving.

One objective is not mentioned in the above list *O2 - User interface accessible from different devices*. This objective, as shown later in this chapter, is enabled by the candidate architecture of the new system and the technologies that implement it.

The objective "O5 - Operators' tasks automation" is not completely achieved with *P3 - agility*. This objective is enabled by having an agile system were new IoT devices may be added as new information sources that need to communicate with the information system. An example is the automatic weighing of the fish pallets. The operators place the pallets over a smart scale that automatically inserts the weighing information into the system. However, as already mentioned in section 1.3.1, the IoT infrastructure is out of the scope of the thesis. The network architecture of the IoT devices and which sensors and actuators to use, are aspects that should be carefully designed.

Properties	Project's objectives	Description
P1 - Responsiveness	O3 - Easily accessible information	A responsive system enables the users to always find quickly the information and to receive quickly the live updates about events that happened inside the information system.
	O6 - Real-time analyses	A responsive system enables to have quick real-time analyses even when the system's load is high.
P2 - Reactiveness	O3 - Easily accessible information	The user dashboard is always continuously updated with the newest information. Consequently, the operators don't have to perform any action to find the information they are interested in and they are notified almost immediately about the presence of the new information.
	O6 - Real-time analyses	The results of time real-time analyses are available to the operators as soon as possible.
P3 - Agility	O4 - Integration with new sources of information	It is straightforward to add new sources of information or to implement new requirements with an agile system.
	O5 - Operators' tasks automation	New features can be easily added to an agile system. Some of these new features may be oriented in integrating new IoT devices into the system.
	O1 - Versatile	Adding and removing new features to an agile system, can be performed frequently, with a low effort and a minimal downtime. Moreover, many administrative operations are automatic.

Table 2.2: Properties to project's objectives tracing

The properties above mentioned are not easy to obtain using traditional technologies, but they are important as they help to obtain a better system that fulfils the project's objectives. The technologies and the architectural style that I have decided to use to implement the system are described in the next sections.

2.2 Candidate architecture

This section presents a description of the designed architecture for the Smart Enterprise project. The goals listed in section 1.2.1 and the architectural properties presented in section 2.1 are fundamental for the design of the system.

The first subsection describes which are the hardware components of the system. The structure of the hardware resources is important for the design of the system's architecture. It allows understanding which are the principal information flows.

The second subsection presents the architectural style that is used to organize the

components of the candidate architecture. It explains why that specific architectural style is appropriate for the SE project. Deciding the architectural style is the first fundamental decision point. It is the decision of choosing what kind of architecture to build. It might be a microservices architecture, a more traditional N-tier application, or a big data solution. There are benefits and challenges to each.

Section 2.2.2 shows that the microservice architectural style is suitable for the SE project, by considering the desired architectural properties discussed in section 2.1

2.2.1 Hardware Components

Figure 2.7 shows graphically which are the hardware components of the system. The description of the components is described in the following list:

- **Smart Things**: are small computing devices inside the factory and the platform of Fiorital's main building. They continuously sense data about machinery and the environment. They are also able to actuate commands to modify the state of the machinery and of the environment;

- **Data center**: provides services to the users, acts as a mediator between the other components and records all the information that is relevant to the company. Figure 2.7 shows the data center divided into servers and database. The data center is a unique logical component, but to better describe the information flows it is important to be aware of the presence of these two different layers. The server layer is responsible for the computations. While the database is responsible for the persistent storage of the system data.

  The servers are the computing nodes that perform computations and provide services to the users of the system.

  The database is the persistence level component of the system. It stores persistently the information that is relevant to the company;

- **Operators' devices**: the company's operators can access the services provided by the system, through any kind of device that has access to the internet and that can show web documents. The operators may be people that are working in the main building of the company, or even people that are working inside other buildings (e.g. corner stores and restaurants). Moreover, the operators are able to access the system even when they are outside the company's buildings. The system allows the users to work remotely when they are at home or other places. To enable this feature, the system needs mechanisms of user authentication and message encryption (e.g. Virtual Private Network (VPN));

- **Third-party data source**: the system allows the connection to itself of data sources provided by other companies and organizations.

The diagram in figure 2.7 shows how operators' devices, third-party information sources and smart things have to use the data center to exchange information between each other. The system doesn't allow the smart things and the external information sources, to communicate directly with the devices of the operators. By having all the communications passing through the data center, it's possible to record each action and decision. Moreover, it's easier to assure security when each source has only a single link to the data center.

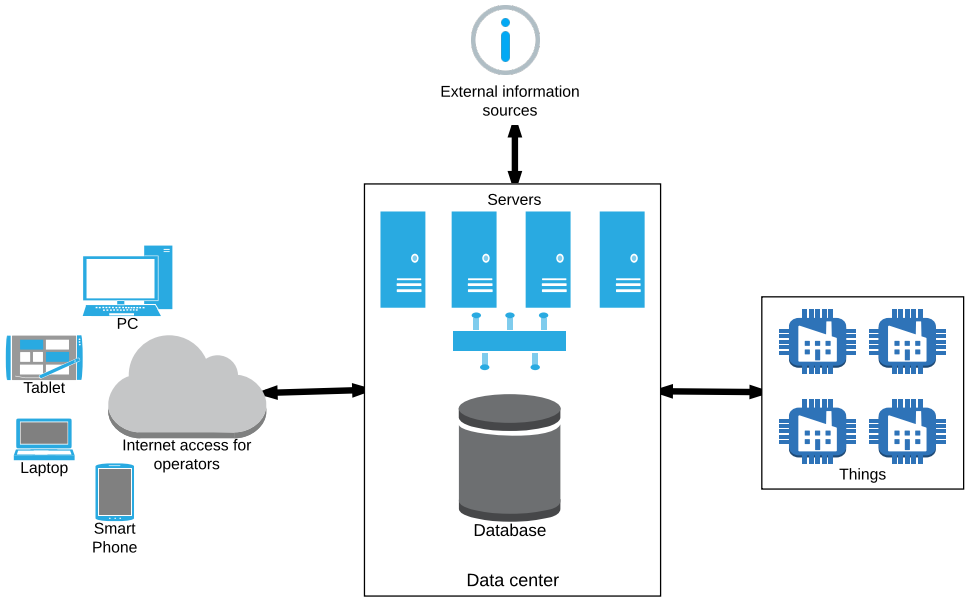

Figure 2.7: System's hardware components.

2.2.2 Microservice architectural style

This style is described by Martin Fowler as "an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies" [8]

Microservice architectures involve the breaking up of an application into tiny services. Each microservice performs a single task that represents a small part of the business capability. The system is built up by using microservices as building blocks.

The beginning of the current chapter gave a definition of what is a software architecture. The terms *architectural style*, *components* and *interfaces* are part of the definition. In this section, there is a claim stating that microservices is an appropriate *architectural style* for the SE project. There are others architectural styles. However, this document focuses on showing why the microservices approach fits the Fiorital's use case rather than, making a complete comparison of which is the best architectural style in this context. With a microservice approach, the *components* of the software architecture are mostly services. A Microservice architecture breaks down the architecture into services as the primary way of componentizing the system. Services are out-of-process components who communicate with a mechanism such as a web service request, or remote procedure call. Services are independent of each other and are completely autonomous, and therefore, changes in the individual services won't affect the entire application. Each service collaborates with other services through an Application Program Interface (API). Therefore, with microservices the *interfaces* are remote procedure calls that form different APIs.

The microservice architectural style has been gaining popularity in the last few years, around 2014. The phenomenon is explained by the availability of new technologies. In particular the advent of the container technology. As stated in the given definition at the beginning of the current section, the microservices should be independently deployable. Consequently, there should be dozens of microservices that should run isolated from the others. The traditional way for running isolated and independent processes was to use Virtual Machine (VM)s. However, a VM adds an important overhead to the execution of a single service. An entire guest OS is virtualized inside a VM. Having dozens of microservices running inside dozens of VMs is not a feasible approach. There are too many OSes running on the system. The waste of resources is too big for having a system with good performance.

The container technology, that is deeply explained in section 2.3.5 helps in resolving the issue. A dozen of containers can run a dozen of microservices that are all isolated and independent from each other. The container technology doesn't require any dedicated guest OS running per each microservice. Instead, a container is very lightweight and has a minimal overhead on the normal execution of a service. Without the container technology, the microservice architectural style wouldn't be a feasible architectural design for an information system.

The microservice architectural style is a specialization of Service-Oriented Architecture (SOA) used to build flexible, independently deployable software systems. Microservices keep many of the principals of SOA, like separation of concerns, loose coupling, statelessness and reusability. The main difference between microservices and SOA is the granularity of the services. The word microservice highlights how the services have to be fine-grained. Microservices should be independently deployable, while SOA services are often implemented and deployed inside single monoliths.

To explain the microservice style it's useful to compare it to the monolithic style: a monolithic application built as a single unit[8]. One fundamental feature of the SE system is exposing a user interface to the company's operators. Enterprise applications are usually composed of three components: client-side user interface (front end), the database and the server-side application (back end). In the monolithic style, the back end is a monolith, a unique application. Where any changes to the system involve building and deploying the entire server-side application.

Figure 2.8 shows a graphical representation of monolithic and microservices architectures. The monolithic version is a single application composed of different libraries. While on the other side there are different microservices, where each of them is a different independent component that can be uploaded to a different computing node. Hence, each microservice is an independently deployable application. This aspect is very important and as discussed later has positive outcomes concerning *responsiveness* and *agility*.

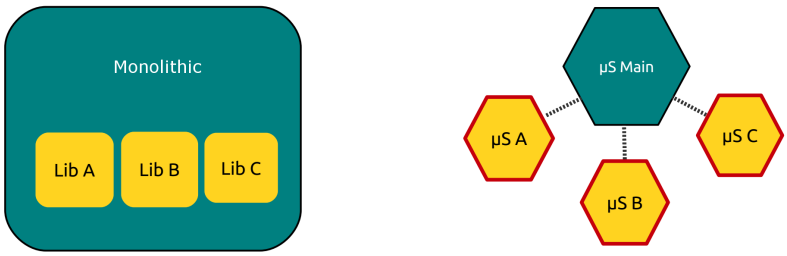

Figure 2.8: Monolithic and microservices architectures

Another important characteristic of microservices is the decentralized data management. Microservices decentralize data storage decisions. While monolithic applications prefer a single logical database for persistent data. Microservices prefer letting each service manage its own database, either different instances of the same database technology, or entirely different database systems. Figure 2.9 shows the data management comparison between monolith and microservice style.

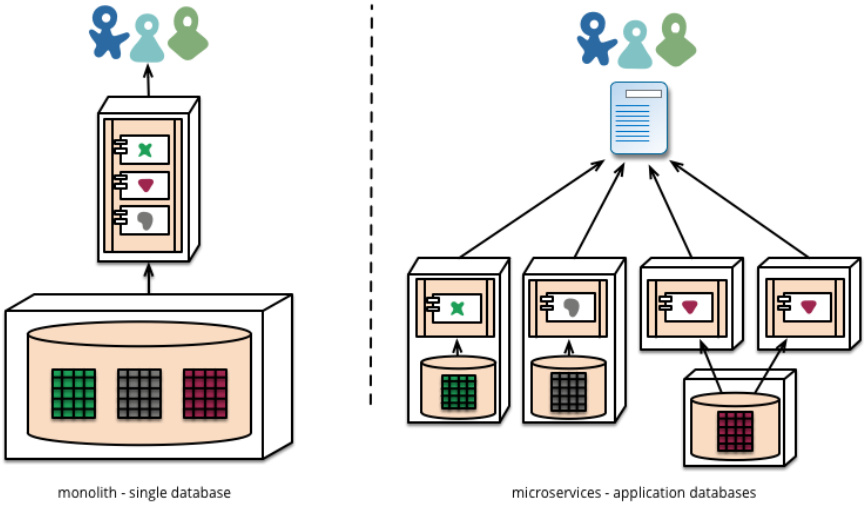

Figure 2.9: Data management comparison between monolithic and microservices approaches.[8]

Decentralizing responsibility for data across microservices has implications for managing data updates. The common approach to dealing with updates has been to use transactions to guarantee consistency when updating multiple resources. This approach is often used within monoliths.

Using transactions like this helps with consistency, but imposes significant temporal coupling, which is problematic across multiple services. Distributed transactions are notoriously difficult to implement and as a consequence microservice architectures emphasize transactionless coordination between services. Without transactions, there can't be strong consistency and therefore microservices use eventual consistency and problems are dealt with by compensating operations.

Choosing to manage inconsistencies in this way is a new challenge for many development teams, but it is one that often matches business practice. Often businesses handle a degree of inconsistency in order to respond quickly to demand, while having some kind of reversal process to deal with mistakes. The trade-off is worth it as long as the cost of fixing mistakes is less than the cost of lost business under greater consistency[8].

The main benefits of microservices architectures are:

- **Ease of deployment**: when one feature of the system is changed, using the monolithic approach requires all the entire monolith application to be redeployed. Whereas, in the microservice approach only the microservice that has been changed needs to be redeployed. With microservices, we can make a change to a single service and deploy it independently of the rest of the system. This allows us to get our code deployed faster. If a problem does occur, it can be isolated quickly to an individual service, making fast rollback easy to achieve. This aspect

makes the overall system more agile. Services can be easily dynamically added and removed from the system at run-time. Therefore, microservices are a good choice concerning the sub-properties *3.1 Components addition agility* and *3.2 Components removal agility*.

- **Independent scalability**: another benefit of the microservice approach is that the scalability of the system is much more efficient. It's possible to scale out only the microservices that are under heavy load, while maintaining the same number of replicas for the microservices that have a normal workload. Different microservices can have different usage patterns, with some microservices needing to perform more read requests, and others with mostly write requests. This aspect highlights even more the benefit of the possibility to scale each of the services independently.

  In the monolithic approach, all the entire application needs to be replicated when a single component is under heavy load. It's important to note that the scalability by replication is possible thanks to the characteristic of the services of being stateless. Without the stateless characteristic of a microservice, it wouldn't be possible to scale it out by replicating it with new instances. Some synchronization and consistency errors would arise. A stateless service doesn't hold any data. Stateless services let the database layer handle the maintenance of a client's state. Since they don't have any data, all the service instances are identical.

  Microservices improve the scalability of the system and therefore also the system's *responsiveness (P1)* is better. Scalability helps in *preserving the SLA (P1.1)*. The users' requests maintain good response times also when the traffic burden is high. Scalability prevents also the presence of long requests' queues *(P1.2)*. When there are many requests. instead of queueing them, the system creates new instances, increasing the execution parallelization. The number of instances shrinks when the number of requests is lower. This scalability mechanism *maximizes the throughput (P1.3)* without wasting any hardware resource. Figure 2.10 shows the scalability difference between the two approaches;

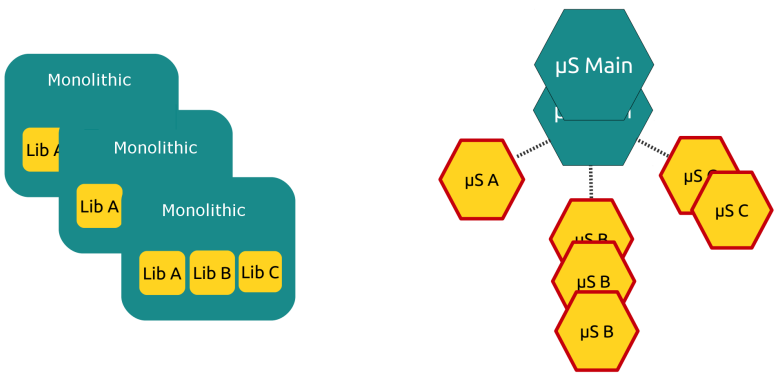

Figure 2.10: Scalability comparison between monolithic and microservices architectures

- **Resilience**: the possibility to replicate independently each microservice improves also the fault tolerance of the system. When a service is in an erroneous state, it's possible to replicate or restart only the service with the erroneous state.

With the monolithic approach, all the application has to be restarted. By having microservices, if one component of a system fails, but that failure doesn't cascade, you can isolate the problem and the rest of the system can carry on working [9]. Handling the fault tolerance of the microservices requires attention but taking care of it appropriately makes the whole system much more stable and fault-tolerant by design;

- **Technology heterogeneity**: with a system composed of multiple, collaborating services, we can decide to use different technologies to develop each different service. This allows the developers to pick the right technology for the context of each specific service, rather than having to select a single generic technology that in most cases, is the technology that the developers know better.

  If one part of our system needs to improve its performance, we can decide to change the technologies of just one microservice. In the monolithic approach changing technologies requires the entire system to be reprogrammed;

- **Organization around business capabilities**: when looking to split a large application into parts, often management focuses on the technology layer, leading to UI teams, server-side logic teams, and database teams. When teams are separated along these lines, even simple changes can lead to a cross-team project taking time and budgetary approval. The microservice approach to division is different, splitting up into services organized around business capability. Such services take a broad-stack implementation of software for that business area, including user-interface, persistent storage, and any external collaborations. Figure 2.11 shows the concept: at the left, there is a team organized around a monolithic approach, at the right, a team organized around microservices;

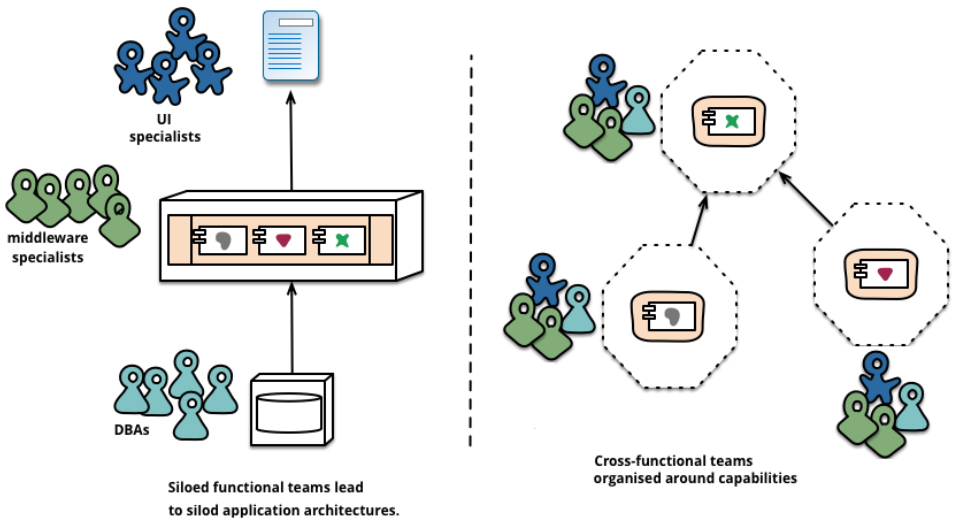

Figure 2.11: Team organization comparison between monolithic and microservices approaches.[8]

The benefits' list helps understanding why microservices are a good choice. However, there are also some downsides that are very important to consider when deciding

whether to use microservices or not:

- **Design complexity**: the division of the system into many microservices is a complex operation that needs particular attention and should be carefully designed before starting the implementation of the system. If the division into microservices is done wrong, it could result in a high network traffic and the communication interfaces being too complex to use;
- **Distribution**: microservices use a distributed system to improve modularity. But remote calls are slow. If a service calls six remote services, each calling other six remote services, these response times add up to some very high latency. A mitigation to this characteristic is to use asynchrony for remote calls. If a service makes six asynchronous calls in parallel, it is now only as slow as the slowest call instead of the sum of their latencies. This can be a big performance gain. Asynchronous calls are also in line with the property *Asynchronous Communication (P2.1)*;
- **Eventual consistency**: microservices add eventual consistency issues because of their usage of decentralized data management. Microservices require multiple resources to update, and distributed transactions are banned (for good reason). Developers need to be aware of consistency issues, and figure out how to detect when things are out of sync. The monolithic style isn't free from these problems. As systems grow, there's more of a need to use caching and replication to improve performance, and cache invalidation is another important problem;
- **Operational Complexity**: the main downside with microservices, is the high number of administrative operations that need to be carried out to deploy, configure and update the system. The microservice architectural style has a bigger number of components and infrastructure requirements compared to the monolithic one. The number of releases increases, they are more frequent but with minor effect. The beginning of the section, describes microservices as "independently deployable by fully automated deployment machinery". This sentence highlights how the operations of deployment should be fully automated. A lot of frequent manual operations to deploy the services of the system would cause a high increase in the number of configuration mistakes.

  It's clear how there is the need for tools that help the developers and the system administrators to automate operations. Automating operations is fundamental task for having a system with the *agility (P3)* property. The most important features that the tools should provide are:

  - **Rapid Provisioning**: the system should be able to fire up new servers in a matter of minutes or hours. The operations to start the new server should be as much automated as possible. To obtain this feature the Cloud Computing paradigm may be used;
  - **Monitoring**: with many loosely-coupled services collaborating in production, things are bound to go wrong in ways that are difficult to detect in test environments. As a result, it's essential that a monitoring regime is in place to detect serious problems quickly;
  - **Continuous delivery**: is the ability to get changes of all types, including new features, configuration changes, bug fixes and experiments, into production, or into the hands of users, safely and quickly in a sustainable way.

The goal is to make deployment, whether of a large-scale distributed system, a complex production environment, an embedded system, or an app, predictable, routine affairs that can be performed on demand [10]. Continuous delivery is essential for a serious microservices setup. There's just no way to handle dozens of services without the automation and collaboration that continuous delivery fosters. Operational complexity is also increased due to the increased demands on managing these services and monitoring[8].

2.2.3 Smart Enterprise software architecture

The previous section presented the microservice architecture as the architectural style to use for the design of the Smart Enterprise system. The architecture is based on the microservice architectural style jointly with the layered architecture pattern. The latter is a client–server architectural pattern in which presentation, application processing, and data management functions are physically separated. The set of clients in the SE system are formed by the company's operators using the control dashboard, the external data sources and the smart things. Figure 2.12 shows an example of the pattern using the most common implementation that consists of four layers: presentation, business, persistence and database.

Each layer in the architecture forms an abstraction around the work that needs to be done to satisfy a particular business request. For example, the presentation layer doesn't need to know or worry about how to get customer data. It only needs to display that information on a screen in particular format [11].

The presentation layer in the SE system corresponds to the software executed by the user interface of the company's operators, by the external data sources and by the smart things. The presentation layer interacts with the system business layer.

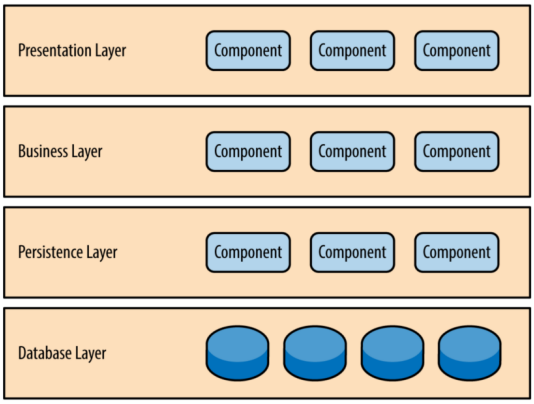

Figure 2.12: Layered architecture pattern.[11]

The business layer and the persistence layer are organized as microservices within the system. The business layer exposes APIs to the presentation layer and is the part of the system that encodes the real-world business rules that determine how data can be created, stored, and changed.

The only layer that is executed on the clients' devices is the presentation layer. The other layers are all executed inside the Fiorital's datacenter. This approach is called thin-client. The client is used mainly for displaying the user interface and to receive the user input. The microservices pattern requires this type of architecture. The

services are all executed inside servers and not on the clients' devices. The presentation layer sends the user input to the business layer. While the business layer sends the requested data to the presentation layer. The thin-client approach has the benefit of easier maintenance and cheaper upgrades [12]. It's easier for system administrators to update only the servers in the datacenter than updating both the servers and the software on the clients' devices. Moreover, the clients benefit from not having to submit their devices to the software upgrades. A single operator may have 4 or 5 different devices that he uses to interact with the system. Today modern web browser helps in having thin clients that don't need to be manually updated.

The persistence layer provides to the business layer simplified access to the data stored in the persistent storage. Different services at the persistence layer may use distinct logical databases to store the data. The database layer represents a typical Database Management System (DBMS) and may be composed of different logical databases. In a microservices architecture beyond having multiple logical databases, there may be also different physical DBMS. This section shows an architecture having only one DBMS.

Figure 2.13 shows part of the SE system software architecture. Each major component is denoted with a number contained in a yellow circle. The diagram presents both the multitier and the microservices design patterns.

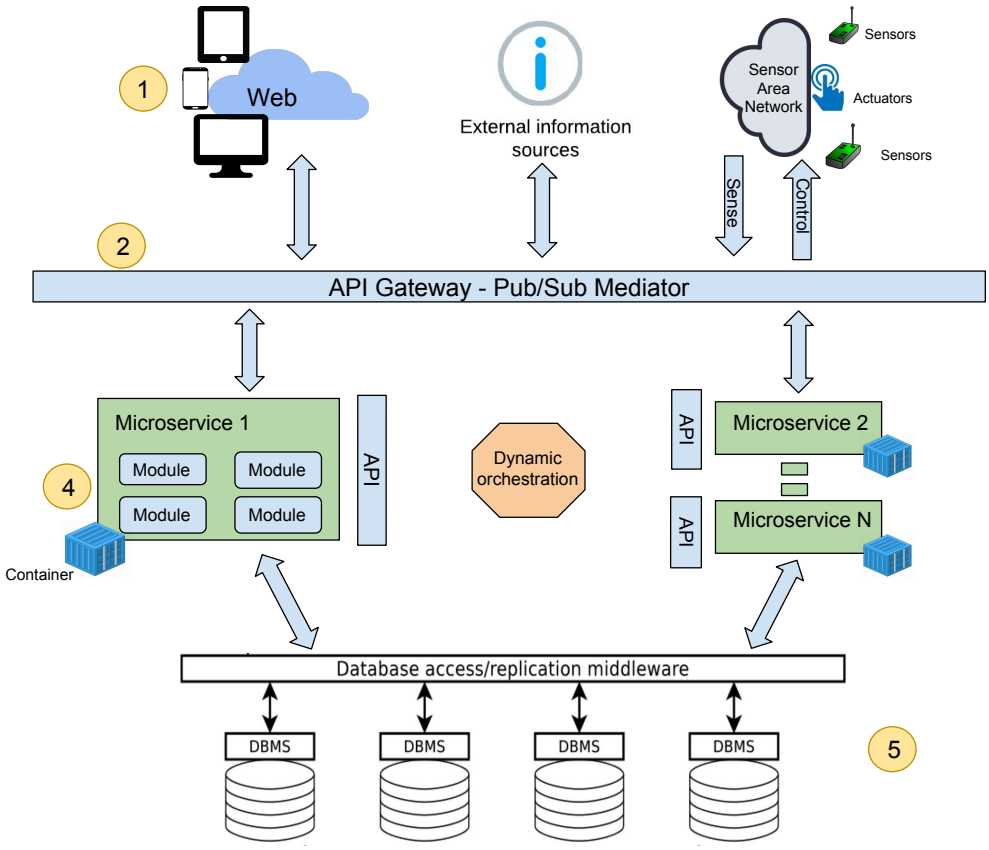

Figure 2.13: Smart Enterprise Tiered Architecture

Presentation layer (1) At the top level (1) there is the user interface of the system, the external information sources and the smart things. This set of components corresponds to the presentation layer. The user interface is executed inside the operators' devices.

Business and persistence layer (4) The business layer and the persistence layer are organized into microservices (4) that expose APIs. The persistence layer services expose their APIs to the services of the business layer. Moreover, services in the persistence layer may perform calls to the other services in the persistence layer. Similarly, the communication between microservices should be performed through remote calls using the connectionless protocol. It would be inefficient to have a connection between each pair of services that need to communicate. Remote calls are slow compared to local function calls. As explained in the microservices section 2.2.2. A mitigation to this characteristic is to use asynchrony for remote calls. Asynchronous calls are also in line with the property *Asynchronous Communication (P2.1)*.

Each microservice runs inside a dedicated container that isolates the microservices between each other. The container technology is fundamental for the effectiveness of a microservices architecture, as explained in section 2.2.2. The candidate architecture includes also a component called dynamic orchestrator. The orchestrator is an essential component for managing the execution of the microservices. The microservices run on a set of servers. The role of the orchestrator is to schedule the services/containers on the different computing nodes. The orchestrator is dynamic because it has a dynamic autoscaling mechanism. The orchestrator continuously checks the run-time load of all the microservices and adapts the number of replicas of each microservice to the load they are experiencing. This characteristic of the architecture is helpful for making the overall system more responsive.

Database layer (5) The database layer (5) is shown in the diagram as a DBMS with different nodes forming a distributed database. The database layer exposes an API to the persistence layer. This API is usually composed of a query language for retrieving and inserting data in the database. The choice of whether to use one or more DBMS doesn't influence the general architecture shown in figure 2.13. Even with only one DBMS, more logical databases can coexist in the system.

The persistence layer communicates with the database layer by performing remote calls or queries to the databases. The communication should be asynchronous. As described in 2.1, asynchronous communication improves the efficiency of the microservices. Asynchronous calls are in line with the property Asynchronous Communication (P2.1).

The diagram shows the DBMS as composed of multiple nodes. The microservices are scalable by using replication because they are stateless. The database layer is stateful and different techniques for scalability should be used. However, the database should be scalable through techniques of data partitioning or through function partitioning techniques (e.g. one instance for write operations, other instances for read operations). Figure 2.14 shows the three different scalability approaches for obtaining horizontal scalability.

API gateway - pub/sub mediator (2) The component called API gateway - pub/sub mediator (2) is shown in the diagram as one component but it's actually formed by two different components: the API gateway and the pub/sub mediator. In

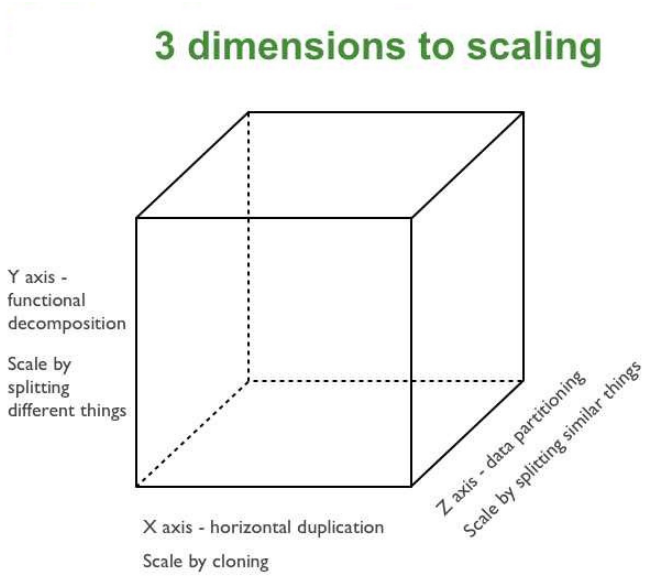

Figure 2.14: The scale cube. [13]

a microservice architecture, there are different services running on different servers. An important design goal for microservices is to hide the fact that there are multiple servers and services. Partitioning into services can change over time and should be hidden from clients. In other words, client applications running on remote machines should have no need to know anything about the internal organization of the cluster.

This access transparency is offered by means of a single access point. The solution is to implement an API gateway that is the single entry point for all clients. The API gateway knows the location of all the microservices. The gateway helps to insulate the clients from how the application is partitioned into microservices and to insulate the clients from the problem of determining the locations of the service instances. Using an API gateway is a common design pattern for the microservice architectural style[14].

The gateway forms the entry point for the microservices, offering a single network address. The gateway, by inspecting a request's payload, can take informed decisions on where to forward the request to. The gateway operates at the level of the application layer (of the networking protocol stack). For example, in the case of web services, the switch can eventually expect an HTTP request, based on which it can then decide who is to process it. This technique is often called content-aware request distribution [15].

The gateway provides also authentication and encryption features. For what concerns the encryption, the gateway uses the Secure Socket Layer (SSL) protocol between itself and the clients. The gateway is also responsible for the authentication of the clients, the clients send their credentials to the gateway that authenticates the requests, and forwards them to the other services. To do this it uses access tokens (e.g. JSON Web Token) that securely identifies the requestor in each request to the services[16].

As the name *API* gateway states. The gateway is helpful for the remote requests of the clients. In a microservices, architecture it is typical to use a connectionless protocol where clients make requests to servers and not vice versa. This approach is not good for a system that wants to have an interactive control dashboard. One architectural

property presented in section 2.1 is *P2.2 liveness*. This property requires having a control dashboard always updated with the most recent information. For a system to obtain the liveness property through a connectionless protocol, it should make use of a polling technique. Polling is bad because there are latency and efficiency issues. Many clients' devices continuously checking the presence of new information waste a lot of resources. Furthermore, the new information arrives later because usually there is a polling interval between two different calls. A system using a push connection-oriented communication protocol doesn't have the efficiency and latency issues. The services send directly the information updates to the clients when they are available. The API gateway is not designed for a connection-oriented communication protocol. For this reason, the system needs another component that acts as a single entry point for all clients for the connection-oriented protocol. This component is a publish/subscribe mediator that allows clients and services to communicate through the publish-subscribe pattern.

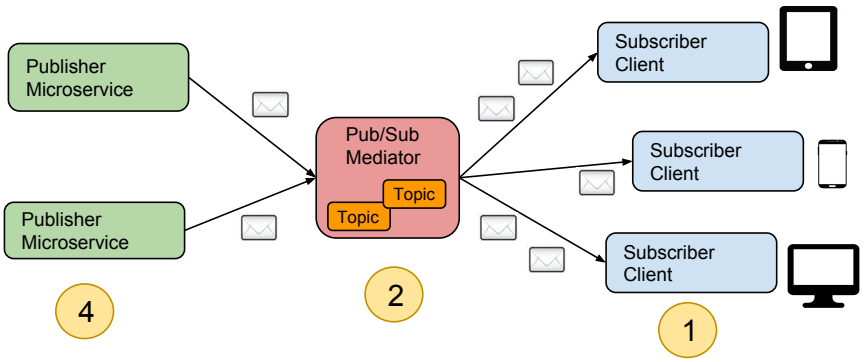

Figure 2.15: Publish-subscribe communication pattern.

Publish–subscribe is a messaging pattern where senders of messages, called publishers, do not send the messages directly to specific receivers, called subscribers, but instead categorize each message with a topic. A publisher has no knowledge of which subscribers will receive his messages. Figure 2.15 provides a graphical representation of the publish-subscribe communication pattern.

Similarly, subscribers express interest in one or more topic and only receive messages that are of interest, without knowledge of which publishers, if any, there are. The mediator is an intermediary message broker that allows this type of communication. The broker receives at run-time the subscriptions from the clients and when a message with a certain topic arrives from a service, it performs the filtering operations and sends the message only to the clients that are interested in that message.

Thanks to the the publish-subscribe protocol, the interactive dashboard is highly customizable by each operator. An operator can change dynamically what to see on his dashboard. The presentation layer reacts to the requested changes by adding and removing subscriptions to different topics.

It is important to repeat that the publish-subscribe communication is used by the microservices to send live updates to the presentation layer. The presentation layer should perform remote calls to the microservices using the connectionless protocol. Similarly, the communication between microservices should be performed through remote calls using the connectionless protocol.

Load balancers (3) and scalability The system is designed to be scalable in all its components. Figure 2.16 shows how the scalability of the microservices works. Each microservice can be scaled out or scaled in depending on the number of requests that are being processed. Each microservice has a server-side load balancer (3) that forwards each request to an instance of the microservice. A load balancer should check the load of each instance, adapting the number of replicas of a microservice with the current load.

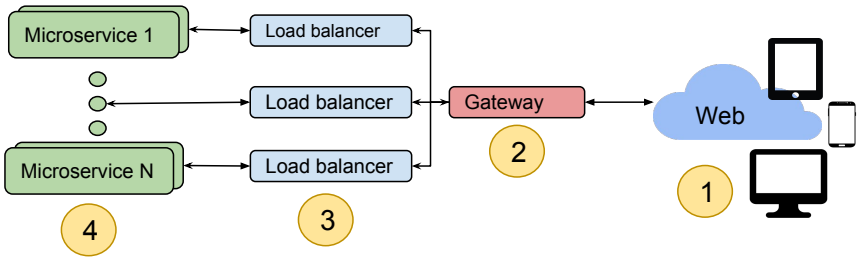

Figure 2.16: Microservices scalability through load balancers.

Figure 2.16 shows a configuration where the gateway is a single point of failure. The gateway that performs the content-aware distribution needs to do a lot of work. There should be a mechanism to scale also the gateway. One approach could be to put a fast transport-layer load balancer in front of the gateway. Consequently, the gateway can be scaled out with new instances when the load increases and the load balancer distributes equally the load to the gateway instances. This approach has the advantage of preserving the single network address as the access point for the presentation layer.

2.2.4 Tracing architectural properties to candidate architecture

The architectural properties described in section 2.1 are mentioned several times throughout the description of the candidate architecture. Different components or characteristics of the candidate architecture help in obtaining the architectural properties. Table 2.3 shows a summary of the relations between architectural properties and the candidate architecture characteristics. The table shows how each architectural property is covered by some characteristics of the candidate architecture.

2.2.5 Technological requirements

Throughout the discussion of the architecture and of the microservice architectural style, many requirements for technologies emerged. Some examples are the distributed DBMS, the API gateway, rapid provisioning and so on. The candidate architecture is complex and is composed of different layers, components, communication protocols and interfaces. The next section presents the technologies used to implement the candidate architecture. Before presenting the technologies choice, it's useful to list all the technological requirements imposed by the candidate architecture. Table 2.4 shows the technological requirements.

Properties	Candidate architecture characteristics
P1 - Responsiveness	- Microservices independent scalability - Dynamic orchestrator - Load balancers - Scalable database layer - Scalable gateway
P2.1 Asynchornous communication	- Asynchronous remote calls among microservices - Asynchronous remote queries to the database
P2.2 Liveness	- Publish-subscribe communication from microservices to the presentation layer
P3 - Agility	- Microservices ease of deployment - Rapid provisioning of computing and storage resources - Services Monitoring - Continuous delivery of updates

Table 2.3: Properties to candidate architecture characteristics tracing.

Tech. Requirement	Description
TR1 - Rapid Provisioning	The system should be able to fire up new servers in a matter of minutes or hours. This feature is essential for a system based on the microservice architectural style.
TR2 - Monitoring	With many loosely-coupled microservices collaborating in production, things are bound to go wrong in ways that are difficult to detect in test environments. As a result, it's essential that a monitoring regime is in place to detect serious problems quickly.
TR3 - Continuous delivery	The goal is to make the deployment routine affairs that can be performed on demand [10]. Continuous delivery is essential for a serious microservices setup. There's just no way to handle dozens of services without the automation and collaboration that continuous delivery fosters.
TR4 - Microservices automatic scalability and load balancing	Microservices can be scaled by replication. There should be a technology that automatically checks the hardware resources usage of each microservice. It should scale the service out or in when needed and load balance the requests among the different service instances.

Tech. Requirement	Description
TR5 - Communication interface and protocol	Each microservice exposes APIs to other microservices or to the presentation layer. Therefore, there is the requirement for a standard way to define these APIs. The interfaces are accessed through a connectionless application-layer networking protocol. Both the interface and the protocol technologies should be widely used and standardized in order to allow easy integration with other technologies and external systems.
TR6 - Asynchronous communication	As described in the candidate architecture, the communication should be as much asynchronous as possible. The presentation layer and microservices should be implemented using technologies that use asynchronous communication.
TR7 - API Gateway	The component (2) of the candidate architecture is the API gateway. The gateway should support the connectionless communication protocol and provide different security features. Developing a proprietary gateway would be very time-consuming. It is more convenient to find an open-source technology.
TR8 - Publish-subscribe protocol and mediator	Another component of the candidate architecture is the pub/sub mediator. The protocol used to communicate should be connection-oriented to allow the microservices sending push messages to the presentation layer.
TR9 - Distributed database	Figure 2.13 shows a DBMS distributed into more nodes. The DBMS technology should be scalable through techniques of data partitioning or through function partitioning techniques.

Table 2.4: Technological requirements.

2.3 Technologies overview and selection

This section presents the set of technologies that are used to implement the candidate architecture of the SE project. The ending part of the previous section presented a list of technological requirements. The technological choices are constrained by these requirements. The set of technologies should fulfil all the requirements. This section describes the selected technologies and motivates all the choices. However, this section is not a complete analysis of the technological state of the art. The technological scope is very wide, as shown in the rest of the section. The chosen technologies and tools range from graphical libraries to DBMS to infrastructure provisioning and management. Due to the limited amount of time available for the thesis work, a complete analysis of the technological state of the art is not present in the document.

The first four technological requirements of the list in section 2.2.5 are related to IT operations. The requirements present complex operations and therefore they should be highly automated. To understand the benefits of automation it is useful to take a look at a typical use case. The manual approach to setting up a typical server environment

includes steps like these:

- Wait for approval;
- Buy the hardware;
- Install the OS on the computer;
- Connect to and configure the network;
- Get an IP address;
- Allocate the storage;
- Configure the security;
- Install and deploy a database;
- Deploy the server application on the computer.

The above list is just a summary of the main operations carried out by system administrators. During the last decade, cloud computing emerged and helped resolve many of the problems related to IT operations. Almost all of the steps in the manual approach became automated, making life much easier for system administrators.

Cloud computing is defined by National Institute of Standards and Technology (NIST) as: "Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" [17].

Cloud computing is an important paradigm, that have reached a pervasive adoption by companies for their IT services. The main motivations behind the adoption of cloud computing use today is both technological and economic. Cloud technology allows taking a very expensive asset, and delivering its capabilities to individual users for a small amount of money per month. The biggest benefit of cloud infrastructure comes from disposable virtual servers that can be easily provisioned and reprovisioned, automatically. IT operations teams realized that it is far more expensive to debug and fix a faulty virtual server than to replace it with a new server.

Cloud computing at the Infrastructure as a Service (IaaS) level provides a set of services accessible through the web. These services allow creating hardware resources like computing, networking and storage. The cloud services can be tuned (e.g. number of cores, storage amounts, network bandwidth) and can be configured for security (e.g. public IPs, firewalls, VPNs) and integration between the web services (e.g. load balancers, auto-scaling, monitoring, deployment). The cloud did not remove the necessity of IT operations. With cloud computing, the system administrators have to configure, install, upgrade, monitor, perform backups, startup and shutdown the system by using the interfaces provided by the different cloud services. The cloud provides higher abstractions for hardware and networking resources. These higher abstractions allow to creating complex configurations where the system becomes highly available, scalable and upgradable. Cloud computing alleviates some of the problems application administrators face in their existing hardware and locally managed software environments. However, the rapid increase in scale, dynamicity, heterogeneity, and diversity of cloud resources necessitates having expert knowledge about the cloud landscape.

Traditional IT administrators use sequential scripts to perform a series of tasks (e.g. software installation or configuration). However, this is now considered an antiquated technique in a modern cloud-based environment. The new way is to use orchestration tools [18]. These tools define the policies and service levels through automated workflows, provisioning, and change management. This creates an application-aligned infrastructure that can be scaled up or down based on the needs of each application. Orchestration also provides centralized management of the resource pool, including billing, metering, and chargeback for consumption. For example, orchestration reduces the time and effort for deploying multiple instances of a single application. The process of orchestration typically involves tooling that can automate all aspects of application management from initial placement, scheduling and deployment to steady-state activities such as update, deployment and health monitoring functions that support scaling and failover [19].

Orchestration differs from automation in that it does not rely entirely on static sequential scripts but rather sophisticated workflows.

Cloud computing is the base for the entire technology stack of the SE project. The cloud computing paradigm introduced the concept of orchestration. Cloud computing benefits from the usage of a virtualization technology called software containers. Cloud, orchestration and containers are the three infrastructure technologies used to obtain the first four requirements (rapid provisioning, monitoring, continuous delivery, services scalability). The next subsections provide a description of each technology. The four layers of the SE system architecture (presentation, business, persistence, database) run on top of the cloud infrastructure. The chosen technologies for the implementation of the layers were selected considering that they should operate in a cloud environment.

The description of the technologies starts by providing more details about cloud computing. The other technologies are presented by following the order of the architecture layers. Web technologies are described first because they regard the presentation layer (client-side) and the implementation of the services (server-side). After that, the API gateway and the publish-subscribe mediator technologies are presented, because they simplify the communication between the presentation layer and the services. Next, containers and containers orchestration are shown because they help to manage the operations of a set of microservices running over a cluster of computing nodes. Eventually, the DBMS technology for the database layer is presented. The technologies dissertation order is shown in figure 2.17. The figure shows how the order of the dissertation follows the architectural layers.

2.3.1 Cloud computing

The fundamental technology that powers cloud computing is virtualization. Virtualization software makes it possible to run multiple operating systems and multiple applications on the same server at the same time [20].

Commonly with cloud computing the IaaS provider gives to their customers a set of virtual machines, each of them running its own instance of an operating system, where the customers can install their applications and the dependencies required by the applications. One problem with this approach is that the installation and configuration process on the virtual machines is usually hard to automate and error-prone. Furthermore, each virtual machine installed on a physical computing node adds a considerable overhead due to the presence of an entire OS on every single virtual machine. The biggest benefit of cloud infrastructure comes from disposable virtual servers that can be easily provisioned and reprovisioned. This feature fulfils

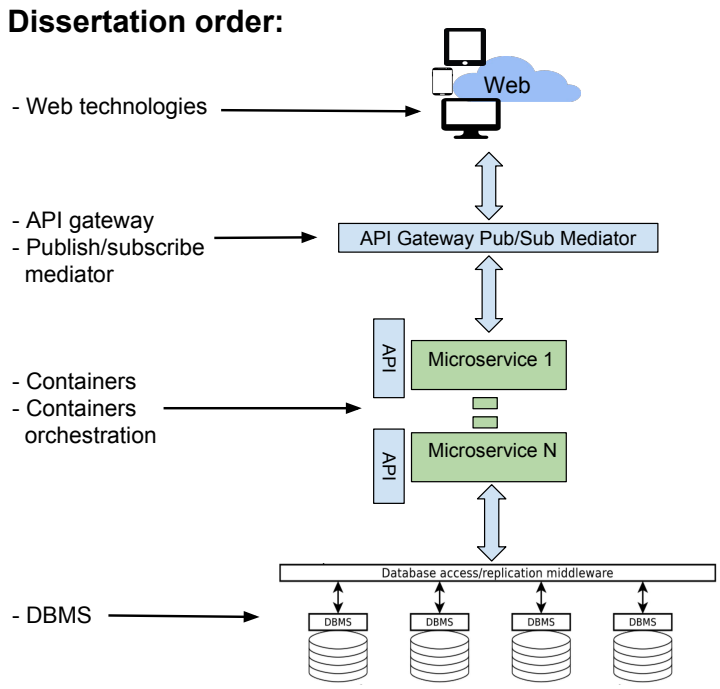

Figure 2.17: Technologies dissertation order follows the architectural layers.

the technological requirement *TR1 - rapid provisioning*. A more recent approach, that solves the main problems of virtual machines, is using containers instead of virtual machines for the deployment of the applications and of their dependencies. Section 2.3.5 explains in detail what a container is and how they compare to virtual machines.

One essential characteristic of cloud computing is to provide computing resources in a manner which the running applications can scale rapidly outward and inward with the proportional demand of users.

Scalability Scalability is defined as: "the ability to handle increased workloads by repeatedly applying a cost-effective strategy for extending a system's capacity." [21]

The property can be obtained in two different ways:

- **Vertical scalability**: the ability to increase the capacity of existing hardware or software by adding resources. Vertical scaling is limited by the fact that an application can only get resources as big as the size of the server;
- **Horizontal scalability**: the ability to connect multiple hardware or software entities, such as servers, so that they work as a single logical unit [22].

Horizontal scalability is more effective than vertical scalability as it is not limited by the size of a single server. However, a system that uses horizontal scalability needs to be more carefully designed as it becomes a distributed system.

Horizontal scalability may be obtained through the replication of the application mechanism. As the workload grows new instances of the applications are started in

order to have more resources that are executing the workload. The applications that want to leverage scalability by replication have to be designed so that the application's instances are executed in a coordinated fashion, with each instance executing a portion of the workload/requests.

The candidate architecture has stateless services that need to be scaled by replication. Therefore, the cloud computing paradigm helps to fulfil the requirement *TR4 - Microservices automatic scalability and load balancing*. Each instance of the service executes a portion of the incoming requests. Each request should be independent from the previous requests so that it's not important which instance of the service is handling the current request.

A typical architecture for a system running in the cloud is shown in figure 2.18. Most of the today's cloud platforms are structured in two layers ("tiers"). The first tier runs services that receive incoming client requests and are responsible for responding as rapidly as possible. Directly behind this is a second tier of which include databases and other applications used by the components of the first tier.

Services running in the first tier are expected to be stateless: they can communicate with other stateful services, but any storage they maintain is expected to be purely temporary and local. However, a typical system needs to store state information and data. This is the role of the second-tier were DBMS are employed. The components running in tier two are central to the responsiveness of tier-one services and they don't have to become a bottleneck for the responsiveness of the system. Usually caching, replication and sharding techniques are utilized to keep the whole system responsive.

The typical architecture for cloud system is in line with the candidate architecture shown in figure 2.13. Layer one corresponds to microservices of the business and persistence layers. Layer two corresponds to the database layer.

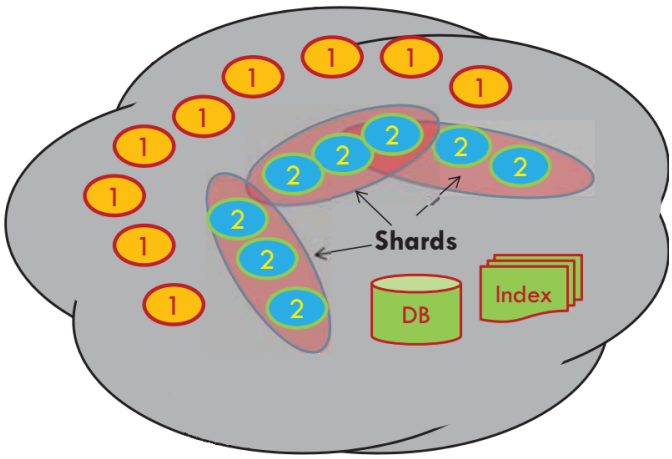

Figure 2.18: Typical architecture of a cloud-based application. [23]

Scalabillity can be "elastic" when the provisioning and de-provisioning of the resources are automated by dynamically measuring the workload that the application is handling. Elasticity was defined as: "the degree to which a system is able to adapt to workload changes by provisioning and deprovisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible" [24]. The first-tier, that is composed of stateless services,

largely benefits from elastic scalability as this tier should respond as rapidly as possible, even when there are intense workloads.

With automatic scalability, there is no need to make a long-term prediction of the workload demand in advance. Long-term predictions are not easy to formulate and are usually wrong or not accurate, leading to over-provisioning and under-provisioning situations. Both the situations causes economic losses for the service provider. Over-provisioning is a waste of hardware resources, while under-provisioning degrades the quality of the provided service, that may cause customers to stop using the service. Figure 2.19 shows graphically a comparison between manual horizontal, manual vertical and elastic scalability.

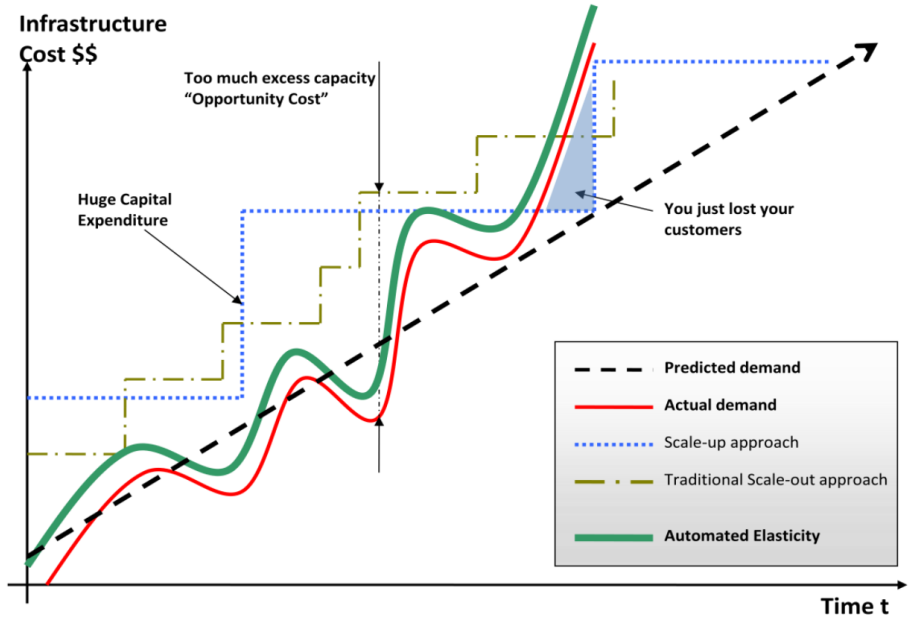

Figure 2.19: Elastic scalability vs manual scalability. [25]

Traditional infrastructure generally necessitates predicting a number of computing resources that the application will use over a period of several years. If the prediction under-estimates, the applications will not have the horsepower to handle unexpected traffic, potentially resulting in customer dissatisfaction. If the prediction over-estimates, the system is wasting money with superfluous resources. The on-demand and elastic nature of the cloud approach (Automated Elasticity), however, enables the infrastructure to be closely aligned (as it expands and contracts) with the actual demand, thereby increasing overall utilization and reducing cost.

Elasticity is one of the fundamental properties of the cloud. Elasticity is the power to scale computing resources up and down easily and with minimal friction [25]. To implement elasticity there should be an auto-scaling system that uses a set of performance metrics to dynamically measure the instances resources usage. Moreover, the auto-scaling system has a set of algorithms that dynamically analyze the metrics. The auto-scaler must be aware of the economic costs of its decisions, which depend on the pricing scheme used by the provider, to reduce the total expenditure [26].

An extensive list of those metrics is provided in [27], for both transactional (e.g.

e-commerce websites) and batch workloads (e.g. video transcoding or text mining). They could be easily adapted to other types of applications:

- **Hardware**: CPU utilization per VM, disk access, network interface access, memory usage;
- **General OS Process**: CPU-time per process, page faults, real memory (resident set);
- **Load Balancer**: size of request queue length, session rate, number of current sessions, transmitted bytes, number of denied requests, number of errors;
- **Application server**: total thread count, active thread count, used memory, session count, processed requests, pending requests, dropped requests, response time;
- **Database**: number of active threads, number of transactions in a particular state (write, commit, roll-back);
- **Message queue**: average number of jobs in the queue, job's queuing time.

Note that, when working with applications deployed in the cloud, some metrics are obtained from the cloud provider (those related to the acquired resources and the hypervisor managing the VMs), others from the host operating system on which the application is implemented, and yet others from the application itself.

The quality of the auto-scaling mechanism can be measured using two fundamental aspects:

- **Speed**: the speed of scaling up is defined as the time it takes to switch from an under-provisioned state to an optimal or over-provisioned state. The speed of scaling down is defined as the time it takes to switch from an over-provisioned state to an optimal or under-provisioned state;
- **Precision**: the precision of scaling is defined as the absolute deviation of the current amount of allocated resources from the actual resource demand [24].

The provisioning speed can be drastically improved by using containers instead of virtual machines. As shown in section 2.3.5 containers are up to 10 times faster to bootstrap than VMs.

The precision of the auto-scaler can be improved by using better algorithms that analyse the resources usage metrics. The analyses consist of processing the metrics gathered directly from the monitoring system, obtaining from them data about current system utilization, and optionally predictions of future needs. Some auto-scalers do not perform any kind of prediction, they just respond to the current system status: they are *reactive*. However, some others use sophisticated techniques to predict future demands in order to arrange resource provisioning with enough anticipation: they are *proactive*. Anticipation is important because there is always a delay from the time when an auto-scaling action is executed (for example, adding a server) until it is effective (for example, it takes several minutes to assign a physical server to deploy a VM, move the VM image to it, boot the operating system and application, and have the server fully operational). Reactive systems might not be able to scale in case of sudden traffic bursts (e.g. special offers or the Slashdot effect). Therefore, proactivity might be required in order to deal with fluctuating demands and being able to scale in advance [26].

Cloud computing technology selection - OpenStack Cloud computing is a fundamental technology for the new information system. Having disposable computing resources improves drastically the agility and the responsiveness of the system. However, the company Fiorital prefers to have an in-house hardware infrastructure rather than using the infrastructure of a cloud provider. One reason for this decision is the presence of an already existing datacenter inside the main building of the company. Meaning that the company could install the new information system inside the datacenter without spending any additional cost for renting hardware infrastructure from a cloud provider. Another reason is the strong dependency on a stable and powerful internet connection that the headquarters of the company should possess in order to have a system that is always available to the company operators. Currently, Fiorital doesn't have an internet connection stable enough that guarantees complete availability.

Therefore, the decision is to use an open-source cloud environment on top of the datacenter of the company. This cloud technology is called OpenStack. OpenStack is an open-source solution for creating and managing cloud infrastructures [28], originally developed by NASA and Rackspace. The open-sourceness of this package, being free to download, gives to small players the possibility of deploying small cloud infrastructures. OpenStack exploits well-known open-source components and libraries, and manages both computation and storage resources on the cloud to enable dynamic allocations of VMs. OpenStack offers a large set of services. A customer can choose which service to install inside its own datacenter. The foundational and most popular services are: Nova to manage the computing resources, Neutron for managing the networking resources, Swift and Cinder for managing the storage and Glance to provide and manage a set of VM images. OpenStack is the most used open-source cloud platform and is being used by thousands of companies [29]. The API Service is the frontend to the OpenStack Cloud: it receives user requests and translates them into cloud actions. The offered functions span from VM control to authentication and authorization. In particular, this service exports such functionalities through web services: user requests are delivered by HTTP messages with XML-based payload, converted into suitable commands, and dispatched to the final internal component. To avoid vendor lock-in, and ease service migration toward different cloud providers, OpenStack exports a set of functions compliant with the ones offered by different vendors, such as Amazon [30].

2.3.2 Web technologies

The information system should be accessible from all kinds of user devices, mobile phones, tablets, laptops and desktops. The choice of deciding what technologies and protocols to use for developing the information system is crucial. The presentation layer technologies should be implemented using technologies that run on as many types of devices as possible. This is one of the project objectives mentioned in section 1.2.1. Furthermore, as stated by the technological requirement *TR5 - Communication interface and protocol*, the choice of which protocols and interfaces technologies to use, should be oriented on allowing a broad number of different devices to be able to interact with the business layer.

Web enabled devices have become extremely commonplace, with mobile devices leading the way, but this fact brings to developers a challenge. The challenge is the fragmentation which includes both device fragmentation and operating system fragmentation across the mobile environment. Devices with different processing, memory, communication and displaying capabilities are examples of device fragmentation [31]. There are different companies with their own platforms, running different operating

systems and requiring expertise in each development environment, making it hard and costly to address multiple devices.

What all platforms have in common, is the increasing compliance to the web standards brought to them through modern web browsers. Technological achievements, such as the improvement of modern browsers and the standardization of HTML5, led to the creation of advanced web applications, offering features only available to native applications before.

Web Applications Web applications can be defined as "software systems based on technologies and standards of the World Wide Web Consortium (W3C). They provide web-specific resources such as content and services through a user interface, the Web browser" [32]. HTML5 is a markup language, used for structuring and presenting content for the World Wide Web and a core technology of the Internet, maintained by W3C [33]. HTML5 is also used as a simplifying term, for addressing a family of other related web standards and technologies, like CSS3 and JavaScript together, with which it represents the complete package, or idea, that is HTML5 [34]. One advantage of mobile applications is that they don't need to be installed on the users' devices. They can be used on-demand through the usage of the web browser. Traditional native applications require the device's OS to periodically check for application's updates and perform their installation, replacing the previous version of the application. On the other hand, web applications, that are accessed on-demand, are always up-to-date when the user accesses them, without any update installation.

Mobile web applications are software that use web technologies, i.e., once again JavaScript and HTML5, to provide interaction, navigation, or customization capabilities. Mobile web applications run within a mobile device's web browser. This means that they are delivered on the fly via the internet and they are not separate programs that need to be installed and stored on the mobile device. The main drawback of mobile web applications is that lack on user experience and performance.

As Java applets and Flash aren't supported on mobile devices, they can no longer be considered as cross platform solutions. The only viable solution left for cross-platform mobile application development is HTML5 and the other modern web technologies, associated with it.

Structuring the graphical interface of the system as a web applications enable to have a system that can be used from all type of devices. This is one of the project objectives mentioned in section 1.2.1.

Single-page web applications In order to increase the performance and offer a user experience similar to that of a native application, web applications can be developed as single-page applications. Single-page applications load the required resources either through a single page load, or load appropriate resources dynamically, in response to user actions. Single-page applications are based on HTML5, on structured JavaScript in both client and server side and on the responsive design technique. These technologies allow to have an interface with no page reloads at any point and thus the user experience is differentiated from the usual web page browsing experience [35].

Web services and REST APIs The presentation layer is implemented using web technologies. The microservices on the business layer should support the HTTP protocol, as it is the foundational communication protocol for the web. The services, therefore, can be called web services. Web services promote the reusability of software

components in internet based applications. They communicate through technologies such as XML, JSON and HTTP.

An advantage of web services is to allow organizations to work together since they are platform and language independent. Web services communicate through HTTP. However, HTTP was designed to transfer hypertexts and not to perform remote calls to services. Web services need a standard way for designing their APIs.

Representational State Transfer (REST) provides an architectural style for implementing web services and their interfaces. REST web services adhere to web standards and use traditional HTTP request and response mechanisms such as the GET, POST, PUT, DELETE request methods.

REST services are resource-centric web services. They revolve around the concept of a resource, and only a limited set of the HTTP operations are allowed to be performed on the resource objects. REST services use Uniform Resource Locator (URL)s to identify resources. In a REST based web service, the GET method fetches information about the resource and the POST method updates a resource or adds an entry, while DELETE will remove an object. Typically, REST web services return the data in the XML or JSON format, but they can also return it via HTML or plain text.

In addition to their inherent simplicity, REST web services are stateless and their GET method operations can be cached by HTTP caches between the client and the web services. This allows to offload heavy web traffic over to reverse proxy servers to lighten the load on your web services as well as the databases.

REST web services can be used directly from a web server or embedded in programs. When used in browser-based applications, the browser can locally cache the results of REST operations when the web server in invoked via a GET request.

AJAX The presentation layer should communicate asynchronously with the microservices, as described by the technological requirement *TR6 - asynchronous communication*. Asynchronous JavaScript and XML (AJAX) is the technology that underlies a single-page web application on the browser. AJAX is a non-blocking way for clients to communicate with a web server without reloading the page. Web applications use dynamic content, where the web pages are generated on the fly in response to search requests or button clicks by the users. AJAX lets the web browser ask the server for new data asynchronously.

Real-time web The microservices should have a way to send push messages to the presentation layer. This need is described by the technological requirement *TR8 - Publish-subscribe protocol and mediator*. Furthermore, real-time web requires full-duplex communication. However, HTTP was not designed to support real-time, the server is not able to send push messages to the client. With the emergence of JavaScript and AJAX, polling techniques enabled near real-time communication. Depending on the polling interval there is a trade-off between latency and efficiency [36]. Another way to achieve real-time communication is the Flash XMLSocket implemented in the Adobe Flash platform [37]. While Flash tends to be installed in most desktop browsers, Adobe has no support in mobile devices. Recent studies overcome the real-time communication problem by using the WebSocket protocol of Internet Engineering Task Force (IETF) and the Web Real-Time Communication (WebRTC) protocol[38].

Chaniotis et al. has examined the problems of modern web application development, as well as client-server real-time communication possibilities, concluding that newer tools such as Node.js, the WebSocket protocol and WebRTC aid in realizing the Real-Time Web [36]. Kai Shuang and Feng Kai, have compared different methods of server

push technologies, and have evaluated their delays and unnecessary connection costs. They have concluded that WebSocket is superior to all the other methods examined [39].

The WebSocket protocol is the choice for the connection-oriented communication protocol required by *TR8*. However, the protocol doesn't support directly publish-subscribe. To obtain the publish/subscribe technology there is the need for a publish/-subscribe protocol that can run on top of the WebSocket Protocol. Web Application Messaging Protocol (WAMP) is an open standard WebSocket subprotocol that provides two application messaging patterns in one unified protocol: Remote Procedure Calls + Publish/Subscribe. Its design goal is to provide an open standard for soft real-time message exchange between application components and ease the creation of loosely coupled architectures based on microservices [40].

WAMP was designed with first-class support for different languages in mind. Nothing in WAMP is specific to a single programming language. As soon as a programming language has a WAMP implementation, it can talk to application components written in any other language with WAMP support. There are libraries for the protocol for all the major programming languages.

WAMP is the chosen technology for the implementation of the publish/subscribe connection-oriented communication protocol.

WAMP needs a broker (mediator) to work properly. The broker technology is shown in section 2.3.4.

Server-side technologies The server-side technology concerns the implementation of the web services in the business and persistence layers. An important technological requirement for these layers is *TR6 - asynchronous communication*. The communication between microservices and from microservices to the DBMS should be asynchronous. Following the technology for the implementation of the services is described. The high concurrency of web server architectures is of high importance for the responsiveness of web applications. Application developers who deal with multiple I/O sources, such as networked servers handling multiple client connections, have long employed multithreaded programming techniques. Such techniques became popular because they let developers divide their applications into concurrent cooperating activities. This promised to not only make program logic easier to understand, implement, and maintain but also enable faster, more efficient execution [41]. Actually, the most deployed web server is Apache and the server side language of choice for web applications is PHP [42]. They are both reported to have around 40% market share. However, PHP was never meant to be used to write complex web-based applications. Apache, on the other hand, is incapable of scaling linearly with the number of CPU cores.

Even though many developers have successfully used multithreading in production applications, most agree that multithreaded programming is anything but easy. It's fraught with problems that can be difficult to isolate and correct, such as deadlock and failure to protect resources shared among threads.

Developers also lose some degree of control when drawing on multithreading because the OS typically decides which thread executes and for how long.

A newer approach is to use event-driven programming. This approach offers a more efficient, scalable alternative that provides developers much more control over switching between application activities. In this model, the application relies on event notification facilities such as the *select()* and *poll()* Unix system calls and the Linux *epoll* service. Applications register interest in certain events, such as data being ready to read on a particular socket. When the event occurs, the notification system notifies

the application so that it can handle the event. Asynchronous I/O is important for event-driven programming because it prevents the application from getting blocked while waiting in an I/O operation [41].

A development platform that uses the event-driven programming approach and that has gained a lot of usage is Node.js [43]. The detailed description of Node.js can be found at the official website of the technology [43].

Node.js technologies is the chosen technology for implementing the web services of the information system. The web services of the information system are I/O intensive applications. In order to implement such an architecture, which is heavily I/O bound and asynchronous by design, demanding in high concurrency support, efficient servers are required. The event-driven approach of Node.js seems rather better than a threaded one.

2.3.3 API gateway

The API gateway is a component of the candidate architecture shown in section 2.2.3. The gateway acts as a single access point for the remote calls of the presentation layer. It dispatches the requests to the microservices based on the type of request. The presentation layer communicates with the microservices through HTTP calls. The gateway is an application layer component, meaning that it can inspect the payload of the HTTP requests. This technique is often called content-aware request distribution.

The chosen technology is Kong: a scalable, open source API Layer (also known as an API Gateway, or API Middleware). Kong runs in front of any REST API and is extended through plugins. It provides SSL encryption features, authenticates the users and passes an Access Token containing information about the users to the services.

Kong was originally built at Mashape to secure, manage and extend over 15,000 Microservices for its API Marketplace, which generates billions of requests per month. Backed by NGINX with a focus on high performance, Kong was made available as an open-source platform in 2015. Under active development, Kong is now used in production at hundreds of organizations from startups, to large enterprises and government departments [44].

The candidate architecture requires that the gateway is scalable. Kong is distributed by nature, Kong scales horizontally simply by adding nodes. Figure 2.20 shows the overall architecture of Kong.

2.3.4 Publish-subscribe mediator

The publish-subscribe mediator is a component of the candidate architecture. The mediator corresponds to the broker of the publish-subscribe protocol. The chosen connection-oriented protocol for the publish-subscribe pattern is WAMP, as shown in section 2.3.2. There are different implementations of brokers for WAMP. The most used one is Crossbar [45]. Crossbar.io is an open- source networking platform for distributed and microservice applications. It implements the open WAMP, is feature rich, scalable, robust and secure. Crossbar.io maintained by Crossbar.io GmbH.

2.3.5 Containers

Containers are a recent technology that provides operating-system-level virtualization. The main difference with the traditional virtualization techniques is that there is no need for an hypervisor layer that provides the virtualization features. Instead, the

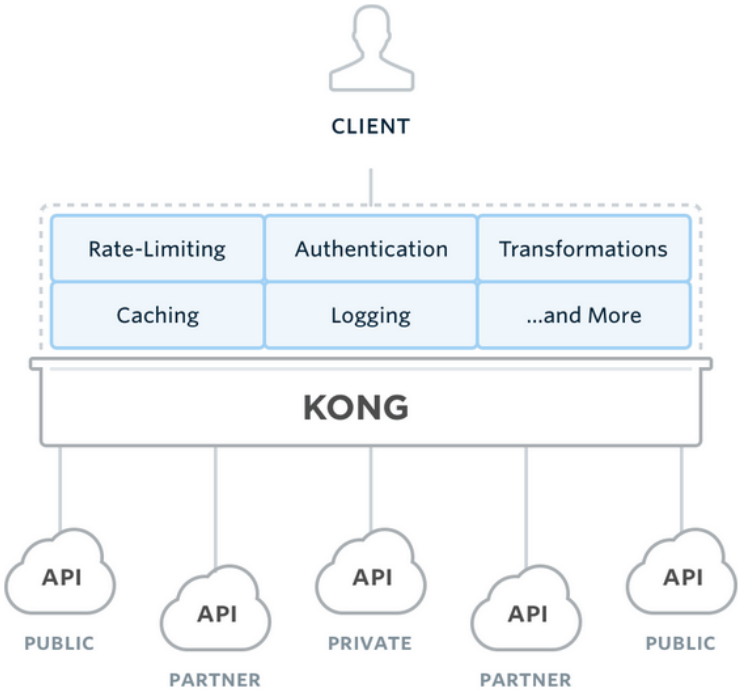

Figure 2.20: Kong architecture. [44]

kernel and the OS provide the virtualization features. Containers in the context of a microservices architecture, as the one of SE project, are useful to package, deploy and run each single microservice. A container is easily mapped to a service.

Traditional virtualization allows a computing node to run multiple isolated operating system images at the same time. This feature is useful in many use cases and in particular for the enterprise sector. A typical use case is where a company has an email server, a Web server, an FTP server and others. For reliability reasons, it's better to install each service on a different machine. By placing each service on a different machine, if a service crashes, at least the other ones are not affected. This aspect is good also for what concerns security, if a service has security flaws, these flaws have no effect on the other services.

Instead of installing each service into a different computing node, virtualization can be leveraged so that each service is installed inside a different VM. Virtualization helps substantially by reducing the amount of computing machines needed and consequently it lowers the costs of supporting a complex computing environment. As reported by A. S. Tanenbaum [46]: "The reason virtualization works, however, is that most service outages are due not to faulty hardware, but to ill-designed, unreliable, buggy and poorly configured software, emphatically including operating systems."

Another important benefit brought by virtualization is the simplified software dependencies management. The traditional way of deploying applications is to install the applications on a host machine using the OS package manager. This approach had the disadvantage of entangling the applications' executables, configuration, libraries, and lifecycles with each other and with the host OS. By using virtualization technologies it's

possible to build immutable virtual-machine images to achieve predictable deployments and updates rollouts and rollbacks. This technique is called *Immutable Server* [47]. These features help in achieving the technological requirement *TR3 - Continuous delivery*.

The container technology helps to resolve the same issues tackled by the VM technology. But, containers are more efficient in many aspects by resolving some of the downsides related to VMs. Before showing the advantages of containers compared to VMs, there is a description of what containers are and how they are implemented.

Application containment is a relatively new technique of operating-system-level virtualization for running multiple services (containers) in isolation on a single computer. Instead of virtualizing full virtual hardware computers like hypervisors do, container-based isolation is done directly at the kernel level. Guest processes run directly on the host kernel, and thus need to be compatible with it.

Containers have been implemented on top of Linux-based OSes. They mostly depend on two features offered by the kernel: cgroups and namespaces. Control Groups provide a hierarchical grouping of processes in order to limit and/or prioritize resource utilization [48]. Namespaces can isolate processes by presenting them with an isolated instance of global resources [49].

While the two features above are probably the most important to implement a container system on Linux, other kernel features can be used to further harden the isolation between processes. For example, the Linux Containers (LXC) system uses six different features of the kernel to ensure that each guest is isolated from the rest of the system and cannot damage the host or other guests [50].

In traditional hypervisor-based virtualization approaches, the guest is a full operating system. With containers, the guest can be as limited as a simple program. A container can also host a full operating system, with the limitation of sharing the same kernel as the host. This is called a system container, while a container running a single application is called an application container. Running a program directly inside the container has the advantage of removing the overhead created by having a second OS executing on top of the host OS [51].

Figure 2.21 shows the difference between container-based and hypervisor-based virtualization. In the container architecture, there is only one OS that is running. A detailed comparison between VMs and container is shown in shown in the paper [52].

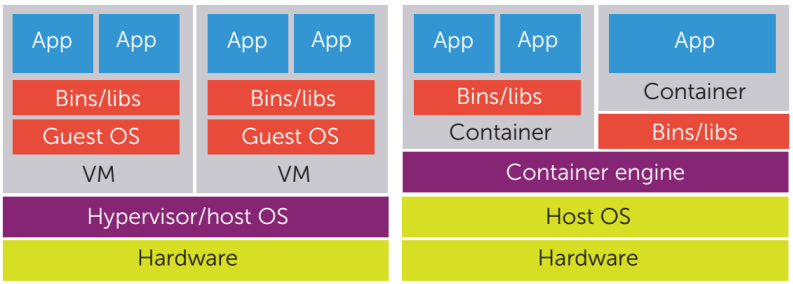

Figure 2.21: Comparison of virtualization between virtual machines and containers.[51]

With most container systems, it is possible to ship an image of an application as a single file. The image contains the application along with all its dependencies. It can be used to create a new container, providing fast deployment. This feature is helpful for obtaining the continuous delivery technological requirement.

Container technology choice - Docker The most widely used container technology is Docker. Docker is a cross-platform container system implementation. Docker is the chosen technology for obtaining the containers. The description of the Docker technology is shown in the official website of the Docker documentation [53].

Docker is the de facto standard among the container technologies. Many other technologies for managing containers are based and support only Docker. Moreover, Docker provides a set of very useful features for building and running containers on single hosts. The Docker ecosystem simplifies certain tasks related to application development. One of its key strength is predictable deployments. Thanks to it, we know that an application that works on one computer A will work as expected on another computer B. The Docker centralized registry provides ready-to-use images that accelerate the start of a project.

Serverless computing A new recent paradigm is gaining traction in the cloud environment. Containers enable developers to readily spin up new services without the slow provisioning and runtime overheads of virtual machines. However, also with container the notion of a server is central. Servers have long been used to back online applications, but new cloud-computing platforms foreshadow the end of the traditional backend server. Servers are notoriously difficult to configure and manage [54], and server bootstrap time severely limits an application's ability to quickly scale up and down.

Many operations have to be carried out to deploy and configure the scalability of the services running inside the containers. Furthermore, with containers an instance of the service is always running even when there is no traffic. Serverless goes a step further where you don't even need to think about how much capacity you need in advance.

As a result, a new model, called serverless computing, is ready to change the construction of modern scalable applications. Instead of thinking of applications as collections of servers, developers define applications with a set of functions (Function as a Service (FaaS)) with access to a common data store. FaaS takes the microservice approach of splitting a monolith a step further by breaking things down even smaller. Figure 2.22 shows a representation of this concept.

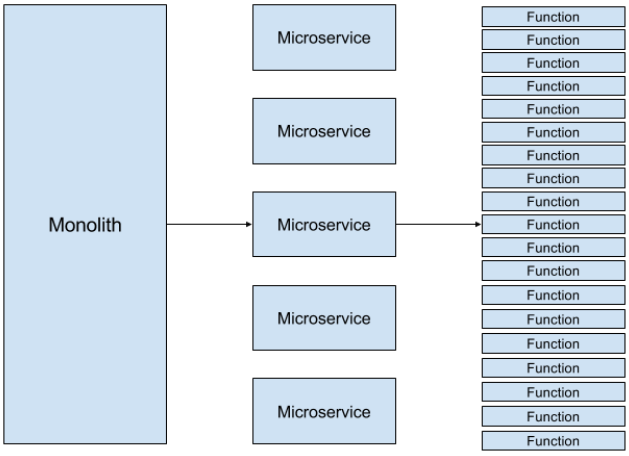

Figure 2.22: Monolith decomposition into functions. [55]

With serverless computing there is no need for operations carried out by the application owner. The scalability of the functions is managed completely by the cloud provider. The functions are very fast to start-up (a few milliseconds). Therefore, only a single API gateway needs to continuously run, while the services can be started only when a request comes in.

From an application owner perspective, you only pay for the time that your function(s) are running and since you don't have to run an app 24/7 anymore, this can be a good cost savings.

An excellent example of this model is the platform Amazon Lambda [56]. The serverless model has many benefits compared to more traditional, server-based approaches. Lambda handlers (Amazon Web Services (AWS) notion) from different customers share common pools of servers managed by the cloud provider, so developers don't have to worry about server management. Handlers are typically written in languages such as JavaScript or Python; by sharing the runtime environment across functions, the code specific to a particular application will typically be small, and hence it is inexpensive to send the handler code to any worker in a cluster. Finally, applications can scale up rapidly without needing to start new servers.

Handlers are inherently stateless, so they are integrated with a cloud-based database service.

AWS Lambda, allows the deployment of microservices without the need of managing servers. This service is designed to offer a per request cost structure, meaning that developers only have to worry about writing individual functions to implement each microservice and then deploying them on AWS Lambda.

Figure 2.23 shows the evolution of virtualization technologies and how they improve the productivity of the development teams.

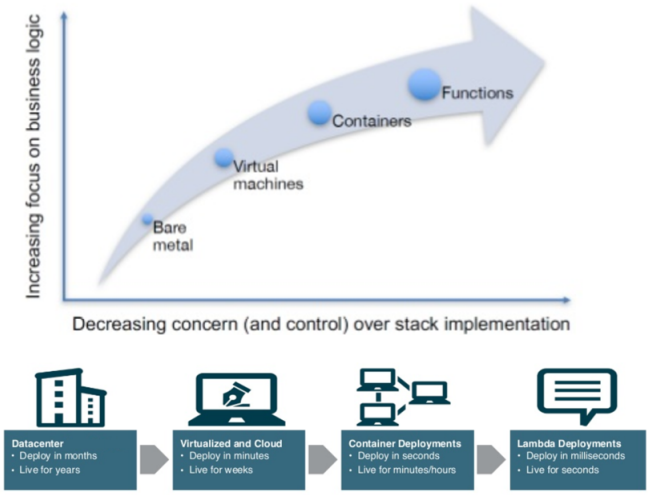

Figure 2.23: The virtualization techniques evolution. [57] [58]

The serverless technology is provided by different cloud providers (Amazon Web Service's Lambda [56], IBM Bluemix's OpenWhisk [59], Google Cloud Platform's Cloud Functions [60], and Microsoft Azure's Functions [61]). However, there are no open-source platforms that enables the developers to use this paradigm in a private

cloud.

The serverless technology has some very good properties. However, there are some drawbacks that make it not suitable for the SE project:

- **No support for WebSocket**: some things you just can't do with functions. Function can't keep an open WebSocket connection with the presentation layer;
- **Slower response time**: a microservice will almost always be able to respond faster, since it can keep connections to databases and other components open and ready;
- **Vendor lock-in**: currently, there is a lack of open-source tools to implement the serverless paradigm. The SE project should run on the Fiorital's infrastructure and not on the platform of a specific cloud provider.

2.3.6 Container orchestration

Cloud technology provides the customer with a set of resources that can be requested on demand. Containers are the technology that enables to virtualize the hardware resources and install different applications on them that are isolated from each other. Hence, the customer is provided with a set of machines connected through a network, where each machine may run one or more server/application. This configuration is called by A.S.Tanenbaum and M.V. Steen as a server cluster [15]. In most cases, a server cluster is logically organized into three tiers, as shown in figure 2.24. The configuration shown the in the figure is similar to the architecture of the SE project.

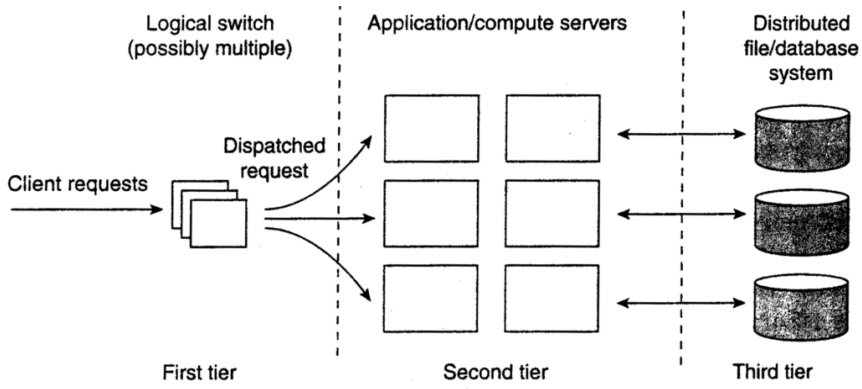

Figure 2.24: The general organization of a three-tiered server cluster. [15]

.

A server cluster should appear to the outside world as a single computer. However, when it comes to managing a cluster, the situation changes dramatically.

Orchestration tools provide lifecycle management capabilities to complex, multi-container workloads deployed on a cluster of machines. By abstracting the host infrastructure, orchestration tools allow users to treat the entire cluster as a single deployment target [62]. The process of orchestration typically involves tooling that can automate all aspects of application management from initial placement, scheduling and deployment to steady-state activities such as update, deployment and health monitoring functions that support scaling and failover. The most basic feature of an orchestration tool is the provisioning, or scheduling, by negotiating the placement of

containers within the cluster and launching them. This process involves selecting an appropriate host based on the configuration. A fundamental feature that orchestration tools provide is declarative configuration. This feature allows teams to declare the blueprint for an application workload and its configuration in a standard schema, using languages such as YAML or JSON. These definitions also carry crucial information about the repositories, networking (ports), storage (volumes) and logs that support the workload. The configuration file allows specifying the order to adopt to lunch or stop a set of containers. The declarative configuration approach allows orchestration tools to apply the same configuration multiple times and always yield the same result on the target system. Another important aspect is container discovery. In a distributed deployment consisting of containers running on multiple hosts, container discovery becomes critical. For example, web services need to dynamically discover the database servers, and load balancers need to discover and register web services.

The Docker technology combined with an orchestration system allows to obtain a production environment where the following operations can be fully automated:

- Building the images of a service inside a container image;
- Sending the image to the images repository;
- Downloading the images to the hosts that should run it;
- Running the images as a Docker container;
- Connecting the container to other containers (cluster of containers);
- Sending traffic to the containers;
- Monitoring the running containers in production.

CaaS The development of the orchestration tools based on containers have lead to the birth of the Container as a Service (CaaS) paradigm. CaaS is an IT managed and secured application environment of infrastructure and content provided as a service (elastic and pay as you go, similar to the basic cloud principles), with no upfront infrastructure design, implementation and investment per project, where developers can (in a self-service way) build, test and deploy applications and IT operations can run, manage and monitor those applications in production. From its original principles, it is partially similar to Platform as a Service (PaaS) in the way that resources are provided "as a service" from a pool of resources. What's different in this case is that the unit of software is now measurable and based on containers[63].

Orchestration technologies The ecosystem of applications around Docker has exploded in the last years [64], with contributions in many areas such as Continuous Integration/Continuous Delivery (CI/CD), application packaging and Container Orchestration Platforms (COPs). Figure 2.25 shows the large number of technologies that have been developed for the Docker/container ecosystem.

Indeed, there are many applications for managing the execution of containers across a cluster of hosts. There are three important open-source projects that provide a wide set of container management features. All of them have been released recently and are being intensely updated with new features. None of them has reached a stable implementation stage:

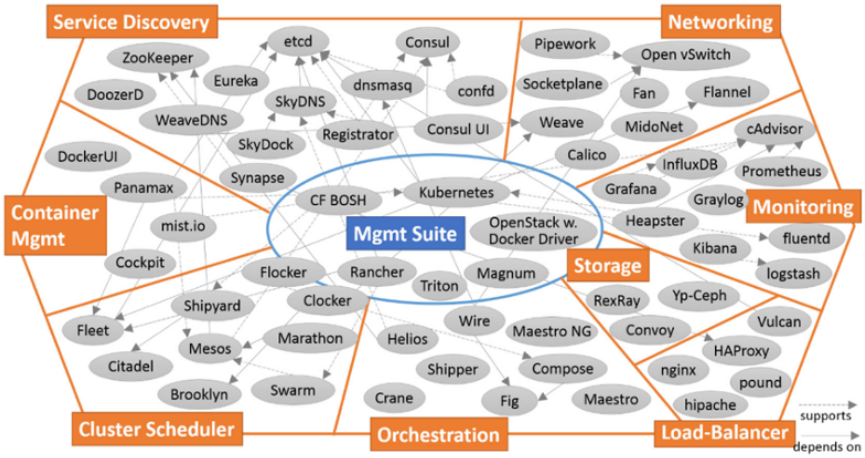

Figure 2.25: Docker ecosystem with dependencies. [64].

- **Docker Swarm**: Docker also has additional tools for container deployments, including Docker Machine, Docker Compose, and Docker Swarm. At the highest level, Machine makes it easy to spin up Docker hosts, Compose makes it easier to deploy complex distributed apps on Docker, and Swarm enables native clustering for Docker;
- **Kubernetes**: originally developed by Google, now supported by the Cloud Native Computing Foundation. Kubernetes is an open-source orchestration system for Docker containers. It handles scheduling onto nodes in a compute cluster and actively manages workloads to ensure that their state matches the users' declared intentions;
- **Apache Mesos**: can be used to deploy and manage application containers in large-scale clustered environments. It allows developers to conceptualize their applications as jobs and tasks. Mesos, in combination with a job system like Marathon, takes care of scheduling and running jobs and tasks.

Container orchestration systems are very useful for improving the agility property desired for the new information system. Orchestration systems automate many aspects of the development lifecycle such as update, deployment and health monitoring functions that support scaling and failover.

Technology choice - Kubernetes The idea is to install a container orchestration platform on top of a set of OpenStack virtual machines. In this way, the system can be designed and deployed as a set of containers, where each container is a different component of the architecture. The resulting infrastructure configuration is a set of containers that are running on top of a cluster of virtual machines. The set of containers is managed by the Kubernetes container orchestration platform. Although Docker is the de-facto standard for containerization, there are no clear winners in the orchestration space. Docker Swarm has few simple features and it's the native option inside the Docker platform, resulting in the most flexible approach and the easier to use, it is a good choice for simple web/stateless applications. The SE project is a complex project with a lot of different requirements. Swarm is very bare bones at

the moment and for complicated, larger-scale applications to production it's better to choose one of Mesos/Marathon or Kubernetes. The decision is to use Kubernetes because is more feature rich and mature of the two [64]. To decide which container orchestration platform to choose among the three option, I carried out just a short investigation for restricted time reasons. The explanation given before doesn't want to be a complete explanation of which technology is the best for the SE project.

2.3.7 NoSQL

The candidate architecture has a database layer. The database layer stores the organizational data of the company and all the information managed by the information system. This section describes why NoSQL databases could be a good fit for the new information system. In recent years new types of DBMS, different from traditional relational SQL databases have been introduced and have been extensively used. These new set of DBMS fall under the category NoSQL. The need for NoSQL DBMS is driven by some challenges presented in building modern applications:

- Development teams are abandoning the waterfall development cycle. Now small teams work in agile sprints, iterating quickly and pushing code every week or two, some even multiple times every day;
- Applications that once served a finite audience are now delivered as services that must be always-on, accessible from many different devices on any channel, and scaled globally to millions of users;
- Organizations are now turning to scale-out architectures using open-source software, commodity servers and cloud computing instead of large monolithic servers and storage infrastructure [65].

Figure 2.18 shows the typical two-tiered architecture for cloud systems. The second tier is responsible for the storage of data. A DBMS is typically used to store data and it should be scalable as it should not become a bottleneck for the first tier. The DBMS in the second tier are intrinsically stateful applications. Therefore, in the second tier, different scaling techniques have to be used than the ones used in the first tier.

Traditional SQL DBMS require complicated techniques to scale outside of a single server and they usually scale with low performances. As a consequence, the NoSQL DBMS that are easier to scale are being much more used in the cloud environment. NoSQL databases give up a massive amount of functionality that a SQL database always provides. Things like automatic enforcement of referential integrity, joins and Atomicity, Consistency, Isolation, Durability (ACID) transactions are not present in NoSQL databases.

In practice, relational databases always have been fully ACID-compliant. Though, the database practice also shows that ACID transactions are required only in certain use cases. For example, databases in banks and stock markets always must give correct data. Databases that do not implement ACID fully can be eventually consistent. In principle, by giving up some consistency, DBMS can gain more availability and greatly improve scalability. This fact can be understood by mentioning the Consistency, Availability, Partition tolerance (CAP) theorem defined by E. Brewer [66]. The CAP theorem states, that for any system sharing data it is impossible to guarantee simultaneously:

- **Consistency**: means that whenever data is written, everyone who reads from the database will always see the latest version of the data;

- **Availability**: each node can perform any operation and the operation terminates in an intended response;
- **Partition tolerance**: means that the database still can be read from and written to when parts of it are completely inaccessible [67].

In cloud-based applications using horizontal scaling strategy it is necessary to decide between C and A. Usual NoSQL DBMS prefer to loose C over A and P. Priority of availability has an economic justification. Unavailability of a service can imply financial losses. A database without strong consistency means, when the data is written, not everyone, who reads something from the database, will see correct data; this is usually called eventual consistency.

A recent transactional model for databases having eventual consistency has been formulated: Basically Available, Soft state, Eventual consistency (BASE) [68]. The availability in BASE corresponds to availability in CAP theorem. An application works basically all the time (basically available), does not have to be consistent all the time (soft state) but the storage system guarantees that if no new updates are made to the object eventually (after the inconsistency window closes) all accesses will return the last updated value. Availability in BASE is achieved through supporting partial failures without total system failure. Eventual consistency means that the system will become consistent after some time.

Many NoSQL databases have been developed recently, more than 225 [69]. Figure 2.26 shows which are the NoSQL databases that are being mostly used by software developers. The chart shows per each DBMS over time how many LinkedIn users have inserted the database as a possessed skill in their personal profile.

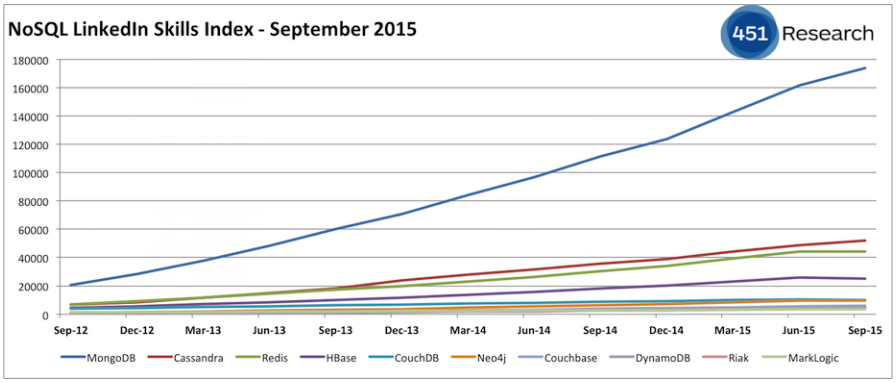

Figure 2.26: Developers NoSQL knowledge by LinkedIn skills [70]

NoSQL databases are important for obtaining the two properties of responsiveness and agility. As explained before, NoSQL databases scale better than SQL databases. Better scalability means that it's easier to provide the property SLA preservation. Regarding agility, NoSQL databases usually don't have mandatory schemas for the data models. This characteristic enables the developers to easily change the data format used by the information system without spending much time in modifying existing schemas and data.

NoSQL technology choice - MongoDB Among the set of existing NoSQL databases, the database that is being used the most is MongoDB. Organizations

of all sizes are adopting MongoDB because it enables them to build applications faster, handle highly diverse data types, and manage applications more efficiently at scale.

Development is simplified as MongoDB documents map naturally to modern, object-oriented programming languages. Using MongoDB removes the complex Object-Relational Mapping (ORM) layer that translates objects in code to relational tables [71]. A detailed description of the MongoDB technology can be found at the official website of the technology [72].

2.3.8 Tracing architectural requirements to technologies

The set of chosen technologies satisfies the set of technological requirements that were shown in section 2.2.5. Table 2.5 shows per each requirement which are the technologies that fulfil the requirement.

Tech. Requirement	Technologies
TR1 - Rapid Provisioning	Cloud computing - OpenStack
TR2 - Monitoring	Containers orchestration - Kubernetes
TR3 - Continuous delivery	Docker containers Containers orchestration - Kubernetes
TR4 - Microservices automatic scalability and load balancing	Cloud computing - OpenStack Docker containers Containers orchestration - Kubernetes
TR5 - Communication interface and protocol	REST APIs for the interface HTTP communication protocol
TR6 - Asynchronous communication	AJAX Node.js
TR7 - API Gateway	Kong
TR8 - Publish-subscribe protocol and mediator	WAMP Crossbar broker
TR9 - Distributed database	NoSQL MongoDB database

Table 2.5: Tracing architectural requirements to technologies

Chapter 3

Experimental evaluation

Chapter 2 presented the candidate architecture of the system and the chosen technologies for the implementation of the architecture. The architecture is quite complex. There are many components partitioned into different layers and various employed technologies. The thesis work wants to obtain a good architecture for the SE project. To assess the effectiveness of the architecture, an evaluation is necessary. The user experience that the system offers is an integral constituent of the system's overall effectiveness. If the application users of the interactive, Internet-based system experience slowness or other difficulties with it, they will be less likely to use the system in the future and they will work less efficiently, regardless of the amount of total processing and storage resources that the system possesses. It is thus in the best interest of the service provider to offer a consistent user experience at all times, even under adverse conditions such as system updates or high demand.

It's essential to implement a prototype of the candidate architecture to obtain a set of results that evaluate the effectiveness of the architecture. A task of the thesis work is the implementation of the prototype. The prototype should be an instantiation of the candidate architecture, composed of services and data that are useful for the Fiorital scenario. The set of services provided by the prototype should reflect the company's structure.

The design of the architecture aims at satisfying the architectural properties shown in section 2.1. The assessment of the architecture has been obtained by using a set of metrics that evaluate the foundational architectural properties. The chosen metrics give quantitative estimations of the architectural properties that can be used to assess the system architecture. To obtain a set of quantitative results, a series of executions of the prototype have been carried out. During the executions, the prototype has been stressed with some synthetic workloads.

The obtained results have highlighted that the candidate architecture has good performance. It scored good responsiveness results mainly thanks to its scalability. The scalability property is obtained through the usage of the three different scalability techniques: horizontal duplication, functional decomposition and data partitioning. These techniques were described in section 2.2.3 and are recalled in 3.2. The functional decomposition technique is the one with the major positive impact on the system's responsiveness. The reactiveness of the architecture is also excellent. All the components communicate asynchronously and the applications users receive the push information on their live dashboards very quickly. The re-configuration of the architecture can be performed quite smoothly. The re-configuration can be carried out with a low number

operations and in most cases, they can be performed at run-time with zero downtime. A more detailed evaluation of the candidate architecture is shown in section 3.2.

The majority of the chosen technologies are also appropriate. The Web technologies, Node.js, Docker and MongoDB have been proven as valuable, mature and efficient technologies that allow having a high-quality information system. Kubernetes, Kong API gateway and the Crossbar publish-subscribe mediator have many good features that are essential for the system implementation. However, they present some flaws, that slightly hinder the quality of the system. Kubernetes requires a complex installation process and lacks maturity. Kong presents an undesired behaviour that is not in line with the candidate architecture. Crossbar lacks development support and scalability. All these aspects are extensively covered in section 3.3.

The chapter starts by presenting the set of metrics and the criteria that are used for assessing the effectiveness of the candidate architecture. Then, the second and the third sections present a critical evaluation of the candidate architecture and of the used technologies. Section four shows the construction of the prototype and its experimental setup. Eventually, the last section shows the complete experimental results.

3.1 Evaluation criteria

The first step for establishing the evaluation criteria is to choose a set of metrics that can be used to obtain quantitative estimations of the architectural properties presented in section 2.1. The following lists presents the metrics by going through the architectural sub-properties:

- **SLA preservation**: aims at maintaining the service level for the application user even when there is a high workload. The service level can be measured by analysing the response times of the application users' requests. The response time refers to the time that elapses between when a user presses a button till when the full result data is received by the user's device. This is also called End-to-End response time [73]. End-to-End response time metric quantifies how long the user must wait for a response to a request. The variance of response times ("jitter") quantifies the degree to which the response time for a particular action, such as a page request, varies during a test interval. It is possible to describe this variance via metrics such as the standard deviation of the response time from the mean, or by providing graphical representations such as box-and-whiskers plots of the observed response time distributions. SLAs usually specify a response time criteria that must be met. A predicted value Q can be used to form a response time SLA of the form "the web API operation responds under Q milliseconds at least p percent of the time"[74]. If Q is a correct prediction, the probability of the request next measured response time being greater than Q is $1 - (0.01p)$. If the time series consists of independent measurements, then the probability of seeing n consecutive values greater than Q (due to random chance) is $(1 - 0.01p)^n$. For example, using the 95th percentile, the probability of seeing 3 values in a row larger than the predicted percentile due to random chance is $(0.05)^3 = 0.00012$.

  Amazon describes response time requirements for internal services in terms of the 99.9th percentile, even though it only affects 1 in 1,000 requests. This is because the customers with the slowest requests are often those who have the most data on their accounts because they have made many purchases, they're the most valuable customers [76]. Figure 3.1 shows a representation of mean,

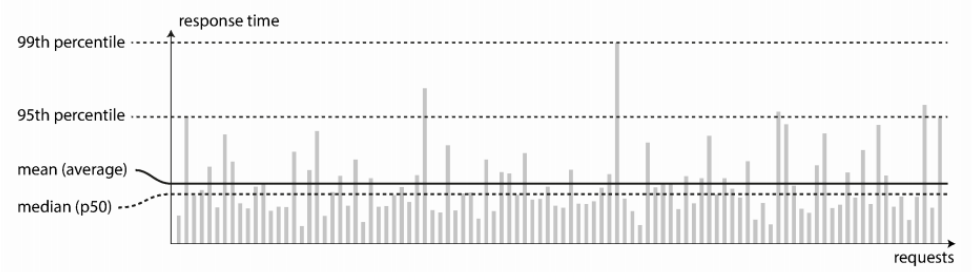

Figure 3.1: Illustrating mean and percentiles: response times for a sample of 100 requests to a service. [75]

median and percentile on the requests response times. It's important to keep those customers happy by ensuring the website is fast for them: Amazon has also observed that a 100 ms increase in response time reduces sales by 1% [75].

The property SLA preservation clearly concerns the application users.

- **Queue length minimization**: each service of the system can execute various requests in parallel. Anyway, the number of requests that a component can execute in parallel is limited, for example by the CPU power available to the component, by the available RAM and I/O. Consequently, the system has a set of queues where to store the requests that are waiting to be executed. Each service instance possesses a requests queue, figure 3.2 shows that each service instance has a requests queue. A service provider wants to provide a system that is responsive. One mechanism is to minimize the length of the queues. A system with short queues is more responsive than a system with long requests' queues. There are two useful metrics to measure the burden of the queues. One is the queue length over time. The other is the waiting time of the requests. The property and the metrics clearly concern the service provider. The application user should not even be aware of the presence of the queues;

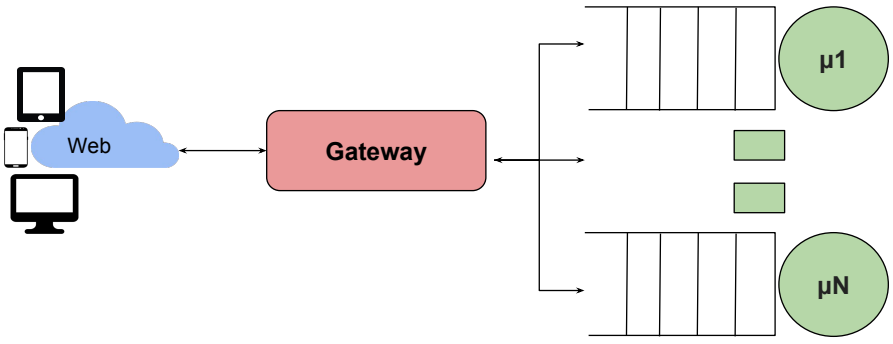

Figure 3.2: Illustration of the services' queues.

- **Throughput maximization**: the service provider to provide a better responsiveness should maximize the throughput of the system. Each service should

be as efficient as possible to maximize the usage of the hardware resources. A consequence of maximizing the throughput is that the overall costs for the hardware resources are reduced. Therefore, the system is more cost-efficient. The throughout can be measured both at the system level and for each service. A good metric is the number of served requests of requests inside a time interval. An example is the number of served requests per second. The throughput metric concerns the service provider, that wants to maximize the efficiency of the system;

- **Reactiveness**: this sub-property states that the different components of the system should communicate through asynchronous requests and messages. This property is not quantifiable into a metric. The property can be assured by using architectural components and technologies that communicate asynchronously; this aspect was illustrated in section 2.2.3 and 2.3. The reactiveness property concerns how the system is implemented and therefore is important only from the point of view of the service provider;
- **Liveness**: the property allows the application users to receive push information coming from the system's services. This property concerns the applications users, who benefit from having always up-to-date information on the dashboard. The source to destination time is a good metric for this property. The source is the user or the smart thing that sends a request to the information system. The destination is the receiver of the push message sent by the information system. The message concerns the request made by the source. An example of the scenario is shown in figure 3.3, where user A sends a requests and user B receives the notification.

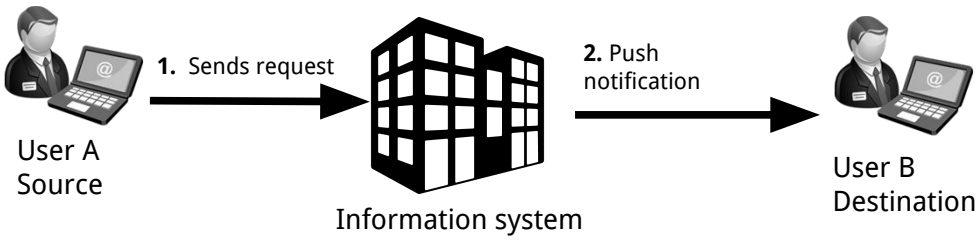

Figure 3.3: Source and destination for the liveness property.

- **Agility**: the term indicates the ease in applying changes in the development and production environment of the information system. The two sub-properties of agility are components addition agility and components removal agility. Both additions and removals change the configuration of the system. A configuration defines how each building block (components and interfaces) is combined to form an architecture describing correct component connections, component communications and interface compatibility. One helpful metric in this context is the system downtime when there is a dynamic re-configuration of the system. The re-configuration may be both an addition or a removal. The downtime can be measured by the amount of time that the system or a portion of the system are not working properly. The other metric is the number of operations that a system administrator has to perform to apply a re-configuration. The agility property concerns the service provider because an agile system helps in simplifying the system administrators' tasks.

Table 3.1 summarizes the techniques to evaluate each property.

Architectural property	**Metrics**	**Pertinence of concern**
P1.1 - SLA preservation	End-to-End response time	Application user
P1.2 - Queue length minimization	Queues length Requests waiting time	Service provider
P1.3 - Throughput maximization	Requests per time interval	Service provider
P2.1 - Reactiveness	Architectural and technological observation	Service provider
P2.2 - Liveness	Source-to-Destination response time	Application user
P3 - Agility	Downtime Number of re-configuration operations	Service provider

Table 3.1: Tracing architectural properties to quantitative metrics.

3.2 Candidate architecture critical evaluation

This section presents an evaluation of the candidate architecture shown in section 2.2.3. The evaluation is based on the results obtained from the exposure of the prototype to synthetic workloads and from the implementation process of the experimental setup. The details about the experimental setup construction are given in section 3.4 and the full experimental results are presented in section 3.5.

The system's prototype has been implemented using the architecture and the technologies shown in chapter 2, the construction of the experimental setup and the simulations' results helped in obtaining an evaluation of the candidate architecture. The evaluation of the architecture is based on the properties and on the metrics shown in section 3.1. The architecture is effective and with good results for what concerns the three properties responsiveness, asynchronous communication and agility.

3.2.1 Responsiveness

The candidate architecture has the capability of being horizontally scalable in all its components. Scalability helps in making the system responsive. The candidate architecture achieves horizontal scalability by using three different techniques. The three techniques are:

- **Horizontal duplication**: creates multiple replicas of a component/service that are behind a load balancer. This mechanism is automated by the autoscaling mechanism of the dynamic orchestrator;
- **Functional decomposition**: splits the application into multiple, different services;

- **Data partitioning**: each server runs an identical copy of the code but it is responsible for only a subset of the data.

Figure 3.4 shows the three techniques represented as three different dimensions. The

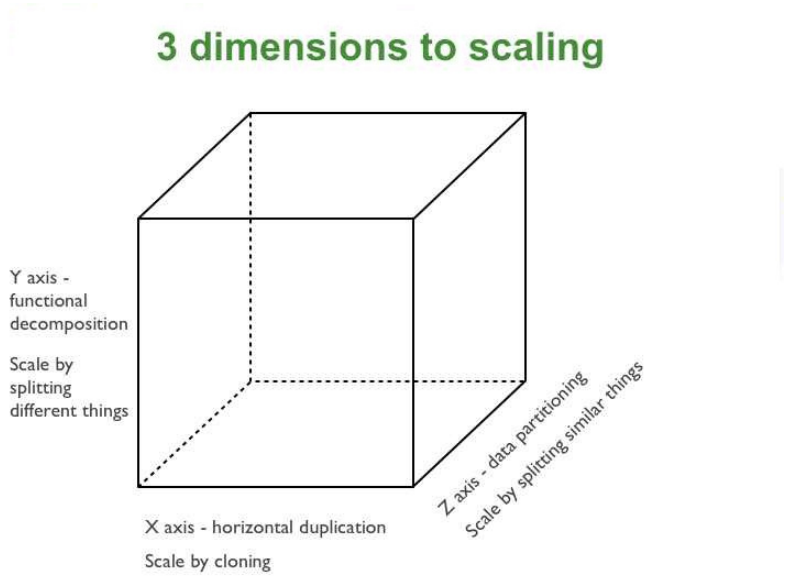

Figure 3.4: The scale cube. [13]

implemented prototype doesn't test the effectiveness of the data partitioning techniques. The database layer is not implemented in the prototype. The horizontal duplication and the functional decomposition were tested during the simulations. The simulations results show that functional decomposition makes the overall system responsive. The prototype is composed of 18 microservices, each of them is potentially deployable on a different computing node. The number of microservices in the real system would be even higher, as not all the departments are represented by the prototype. The high number of microservices allows the system to leverage the hardware resources provided by a cluster of computing nodes. The orchestrator manages the scheduling of the different containers/services on the cluster of computing nodes, trying to balance the resource usages among the nodes. Figure 3.5 shows the CPUs and RAM usages during the executions. The orchestrator tries to leverage the resources of all the nodes in a balanced way. The nodes named s1 and s2 have fewer cores and RAM than the other 3 nodes. This is why their resources usage is lower.

The simulations show also that the automatic horizontal duplication is effective. The main problem with this technique is the design of the autoscaler. Obtaining an efficient autoscaler for a given service is not an easy task; section 2.3.1 presented the various metrics that may be used. Determining which metrics and logic to use for a service depends much on the specific service's characteristics. A bad autoscaler may not detect the burden that a service is experiencing. Furthermore, the high number of different microservices increases the complexity, as different services may require different autoscaling logics. The majority of the IaaS-level autoscalers only use the CPU and the RAM usage as metrics for deciding when to scale a service. The simulations highlighted how other two metrics would be much more effective: the length of the

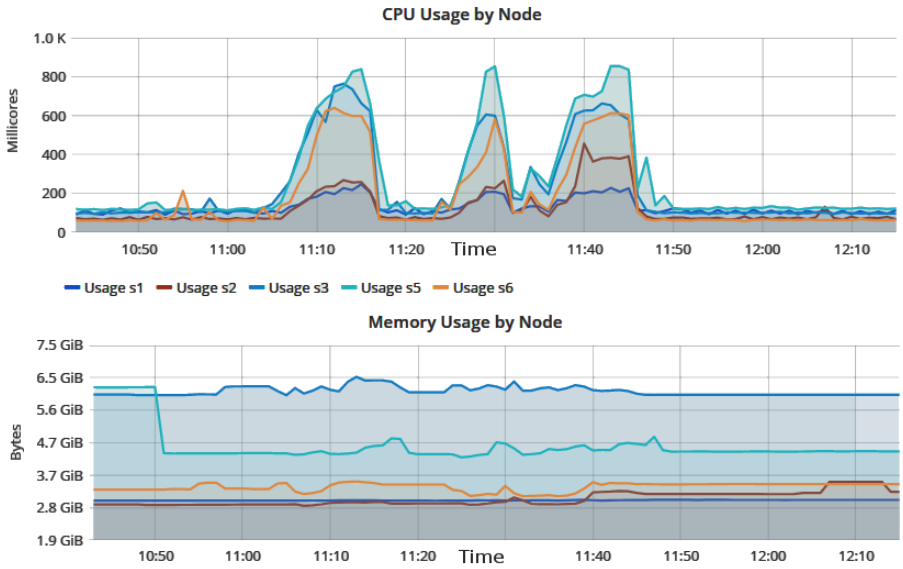

Figure 3.5: Nodes resources usage during the simulations.

requests queues and the mean response times.

The API gateway component is central for the system's scalability. It is essential for hiding to the presentation layer the functional decomposition into different microservices. The gateway itself can be scaled by duplication, as it shouldn't become the system's bottleneck

The simulations show that the candidate architecture is able to manage high requests burden in an efficient way through autoscaling. The dynamic autoscaling helps in maintaining low response times and low queue length, but, at the same time, it helps in maximizing the throughput of the components. When the load is low, only one instance of a service is running, otherwise, there would be a waste of resources causing a non-optimal throughput.

The requests burden of the simulation is based on the number of operators of Fiorital and on their typical operations.

One drawback of the microservice architecture is the high RAM consumption. The system is composed of dozens of microservices. Each of them is an independent container with a web server and its own memory address space, the same libraries are loaded into different containers multiple times. This aspect increases the RAM consumption compared to an application running a single web server. A single container running a Node.js microservice requires around 100 MB of RAM when there is no traffic. A single container running the Kong API gateway requires 300 MB of RAM. A single container running the Crossbar mediator requires 100 MB of RAM. The nodes are also running the Docker daemon, the Kubernetes orchestrator and the host OS. The Kubernetes orchestrator needs a dedicated node to run the orchestrator master node that coordinates the slave nodes. The master node doesn't run any microservice and therefore increases the overall system resources consumption. However, when the system needs to be scaled, only the services under heavy load will be scaled, while in the monolith approach all the application has to be scaled out. A single service needs a small amount of RAM (100 MB), while the entire application would require much

more.

3.2.2 Reactiveness

This property is composed of the two sub-properties *asynchronous communication* and *liveness*. The candidate architecture is effective for both the sub-properties. The presentation layer communicates asynchronously with the business layer, in turn, the business layer communicates asynchronously with the persistence layer. The user requests go through many steps and they all execute in an asynchronous and nonblocking fashion. The gateway and the mediator forward the messages between the microservices and the presentation layer, they also perform their operations asynchronously. Asynchronous communication improves the overall responsiveness of the system because the components never stop their execution to wait for an acknowledgement or an answer. The candidate architecture is composed of many independent services. The fact that they are independent helps in obtaining an efficient system that uses asynchronous communication among its components.

The drawback of distributed asynchronous services is that programming them becomes harder. It's harder mainly because the developers are used to the synchronous programming model where the execution workflow follows the written source code line by line. Another reason that adds complexity is that distributed systems are harder to program, remote calls are slower than local calls and are always at risk of failure. The queries to the database and the remote requests to other services are asynchronous. The mechanism that allows receiving the responses to the asynchronous requests/queries is the callback. It is a function that is executed when a certain event is detected. In the case of a remote request, the event is the arrival of the response to the remote request. A service that receives a certain user request may perform asynchronously different remote requests, making the overall execution workflow more complex.

Regarding the liveness property. The executed simulations highlight that the publish-subscribe mediator allows the application user to receive quickly the live information on the dashboard. The microservices send asynchronously the updates to the application users through the publish-subscribe mediator, that is lightweight and can easily handle the amount of open connections for the number of Fiorital operators. To measure the liveness of the system the source-to-destination response time metric was used. The source-to-destination response time in the simulations corresponds to the same values of the end-to-end response times. This aspect highlights the adequate liveness of the system. All the applications users interested in the topic receive the update as fast as the user that caused the update.

3.2.3 Agility

The agility results show that the system can be updated dynamically without taking down the system and consequently with zero downtime. Furthermore, the number operations that are necessary to change the system components' configuration is low. It is helpful to recall what is a system's configuration: defines how each building block (components and interfaces) is combined to form an architecture describing correct component connections, component communications, interface compatibility and that the combined semantics of the whole system result in correct system behavior [7].

The system's agility benefits a lot from its architecture. The subdivision into many microservices, the dynamic orchestration and the presence of the API gateway make

the system very agile.

To add new features to the API of a microservice, the dynamic orchestrator, is able to achieve zero downtime. The orchestrator performs the update by creating a new instance of the updated container/service, it instructs the load balancer to route the new requests only to the new instance, it waits a certain amount of time to allow the old instance to respond to all the processing requests and finally turns down the old instance. To add a new service to the system, it's even easier, there is no old instance to replace but only a new service that is dynamically instantiated. After creating the new service, the gateway is informed about the presence of the new service and starts dispatching requests to it.

To remove some API calls from a microservice interface, the operations of the dynamic orchestrator are similar. The main difference is that the gateway or the microservice itself, should respond to the new requests requiring the old interface with a message that signals that the interface is not any more supported.

The agility of the system benefits from the microservices' characteristic of being independently deployable: simple services are easier to deploy, and since they are autonomous, are less likely to cause system failures when they go wrong. The gateway is an essential component for the system's agility, as it decouples the interface required by application layer from the microservices partitioning. The gateway is scalable and its routing logic can be updated dynamically without requiring any restart and downtime.

The orchestrator is essential for automating many re-configuration operations. A full description of the operations required for performing some re-configurations is described in section 3.5.

The main drawback to the system's agility is the architecture complexity. The architecture is made of many distributed components that require different technologies. The operation teams needs to be fully prepared on the architecture and on the technologies. There are lots of services to manage, which are being re-deployed regularly.

3.3 Technologies critical evaluation

This section presents an evaluation of the technologies chosen for the implementation of the architecture. The technologies were discussed in section 2.3. The evaluation is based on the results obtained from the exposure of the prototype to synthetic workloads and from the experimental setup's implementation process. The technologies evaluation order used in the next paragraphs is the same used in section 2.3.

Web - HTTP and WebSocket The web technologies used in the prototype implementation are the protocols HTTP and WebSocket. The HTML and JavaScript user interface is not part of the system's prototype, therefore the technologies were not properly tested. However, they are widely used and standardized technologies for the implementation of web applications. The protocol HTTP together with the REST interfaces is used for doing remote calls from the presentation layer to the microservices and also for the communication from a microservice to another microservice. The HTTP protocol is compliant with the candidate architecture features, it's a rich protocol, highly standardized with a broad usage among companies. The API gateway that works at the application layer, inspects the HTTP requests. The HTTP protocol is the most supported protocol by the technologies that provide an API gateway.

The WebSocket protocol is a constrained choice for obtaining push messages in web applications. The WebSocket is highly standardized and has a great support by web browsers and web servers. However, the WebSocket protocol doesn't provide publish-subscribe features. For this reason, the WAMP protocol is used on top of the WebSocket protocol. The former is an open standard WebSocket subprotocol. The protocol provides the required publish-subscribe features. However, it's a protocol that is not much used by developers and companies. As a result, there are libraries for many programming languages but they are not fully tested and with a low-quality documentation. At the moment, there is no valid alternative to WAMP as a publish-subscribe protocol built on top of the WebSocket protocol.

Kong API Gateway The Kong gateway is the technology used for the implementation of the API gateway. It is a feature-rich technology that can be dynamically configured by adding and removing publicly exposed APIs. It is possible to write scripts that can be executed dynamically for managing the gateway parameters configuration. The gateway supports a high number of requests and can be scaled by duplication.

The simulations highlighted a problem with the technology. When there is a high number of requests, some microservices may degrade their performance and reply with higher response times. The Kong gateway notices these delays and acts consequently by queueing the requests. This approach is in contrast with the system's candidate architecture. The gateway limits the requests, but this approach hides to the dynamic orchestrator the high burden that the microservices are experiencing. Therefore, the orchestrator doesn't scale dynamically the microservices. The consequence is that the requests' queues of the gateway may continue to grow, making the overall system much less responsive when there are high requests loads.

A better approach would be to have the gateway directly integrated with the dynamic orchestrator. In this way, the orchestrator would have precise knowledge about the live response time of the microservices. This metric would be useful for the autoscaling mechanism of the system.

Crossbar Mediator The Crossbar technology acts as a publish-subscribe broker for the WAMP protocol. Crossbar handles very good the the push messages sent by the microservices. It doesn't add any delay to the communications. There are two problems with the technology. The first is that it's supported by a single small company that could possibly stop supporting it in the future. The other problem is that the technology can't be easily scaled by replication. At the moment, the possibility to scale it is given only in the enterprise version of the technology.

Node.js The Node.js technology is used to implement each web microservice. The event-driven asynchronous approach of Node.js is really good for the system's architecture. The web server of Node.js is light and is very responsive. One Docker container running a Node.js microservice requires 100 MB of RAM, that is a low amount of memory. A dozen of microservices instances may run on a computing node without any problem. Furthermore, Node.js uses only one thread for executing the event-loop. This Node.js characteristic enables one instance of a microservice to consume resources up to one core. This aspect facilitates controlling the CPU usage of the microservices.

Node.js has good support for the HTTP protocol and the WAMP protocol.

Docker The Docker technology is really good for providing containers. The Docker images are quite small and the containerization overhead in running the services is limited. The Docker technology is also very fast in packing the microservices images. The microservices are similar to each other, they all use the same Node.js libraries. When packaging a microservice, the layered filesystem of Node.js reuses the majority of the layers from another microservice image, reducing considerably the time spent in packaging the image. The same consideration is valid also when uploading the image to the Docker registry, only the changed layers are uploaded to the Docker registry. Usually, only one layer is changed, and a few KBs are uploaded to the Docker Registry.

The installation of the Docker registry is not very simple. It requires a valid SSL certificate, otherwise, it won't work properly. Furthermore, Kubernetes has to be instructed about the presence of the Docker registry. It would be more straightforward to have the orchestration system providing directly the container registry as a unique platform.

Kubernetes Kubernetes manages efficiently the orchestration of the containers on the set of available virtual machines. It provides straightforward abstractions for the management of stateless services. It is very simple to create a microservice with a load balancer on top, an autoscaler that scales in and out the number of instances and an entry on the internal Domain Name System (DNS) service provided by Kubernetes. The operations for adding and removing microservices are highly automated and with zero downtime. The same considerations apply for updating the interfaces of existing microservices. Kubernetes allows to monitor the health of the containers and to see their logs.

There are some flaws in the current Kubernetes technology (v1.5). The provided autoscaler mechanism uses only the CPU usage metric. The decision about when to scale in/out is only based on the current CPU usage. The autoscaler should use other important metrics such as the RAM usage, the requests queue length and the response times. Another flaw, that was already mentioned above, is the lack of automatic integration with the API gateway and with the Docker Registry. The orchestrator should provide these two components as a whole platform.

One more flaw is the complicated installation process of the Kubernetes technology on a cluster of computers. Kubernetes claims to be a technology that improves the portability across different IaaS cloud and bare-metal environments. The YAML declarative configuration files of Kubernetes, the Docker images are claimed to be portable on any infrastructure that has Kubernetes installed. This claim is correct. However, the portability is harmed by the complexity of the technology installation. In addition to being complex, the required installation operations vary a lot according to the infrastructure environment that is being used. These flaws should be resolved in the next versions of Kubernetes. The technology is being rapidly developed and some solutions to the aforementioned flaws are being developed. However, the fast development that Kubernetes is undergoing isn't a good characteristic for a company's information system.

MongoDB The MongoDB database was installed in the experimental setup as a replica set of three instances. Each instance is a Docker container with an attached persistent volume. However, the simulations didn't test the performances and the operational agility of the database. An evaluation of the technology is not possible within the current status of the project.

3.4 Experimental setup

To obtain quantitative results for the metrics shown in section 3.1 a prototype of the system is necessary. Furthermore, a group of synthetic workloads should be run on the prototype. The executions are more meaningful if they represent the Fiorital's scenario. The prototype should be based on the functional needs of the company, the data of the prototype should have the size of the expected company's data and the number of simulated users should be close to the Fiorital's number of operators.

The goal of the setup is to create an environment where different tunable executions can be performed. The executions generate a set of results regarding the metrics shown before. The setup is composed of two different parts. One part is the prototype of the system that implements the candidate architecture shown in section 2.2.3. The second part is a script that generates a workload by sending requests to the prototype. The implemented prototype doesn't have the user web dashboard. The prototype receives requests generated by the script that doesn't run inside a web browser. This technique is common for an experimental setup and it's called benchmarking [73]. That means to create a script or otherwise controlled execution of selected functions in an application. The script is run automatically and periodically at an interval that will provide a reasonable sample of the application response time. The main advantage of this technique is that script response time can be more accurately measured than the actual production environment of the application. An issue with the approach is that it only provides an approximation of the application response times. A further problem with this approach is that it often measures the application over a limited subset of the network. This can cause the response time measurement to be skewed (either too low or too high) [73]. The next two sections show the two parts of the setup: the prototype and the benchmarking script.

3.4.1 Prototype

This section describes the implemented system prototype. The architecture and the technologies used in the prototype implementation are the same described in chapter 2. One objective of the prototype is to record the quantitative results of the workload experienced during the simulations. Therefore, the architecture shown in 2.2.3 is enriched with new features for recording the metrics' results. The illustration of the prototype starts by describing the OpenStack environment utilized and the above construction of the Kubernetes cluster. Later, the illustration follows the order of the architectural layers: the API gateway and the publish-subscribe mediator technologies, the microservices and the persistence layer.

OpenStack environment All the computations and installation are performed on the CloudVeneto OpenStack cluster [77]. The OpenStack cluster is provided by the University of Padua and by the Padua-located branch of the INFN (National Institute of Nuclear Physics). The cluster is composed of 12 computing nodes PowerEdge M620 (each equipped with: double processor Intel Xeon E5-2670v2 2.5GHz, 25M Cache, 8.0GT/s QPI, Turbo, HT, 10 Core and 160GB di RAM 1866MT/s), 4 management nodes PowerEdge M620 (each equipped with: double processor Intel Xeon E5-2609 2,40GHz, 4 Core, cache 10MB, 6,4GT/s QPI and 32GB di RAM 1600MHz). The total available storage space is 90 TB, the majority of the space is provided by 41 Hard Disk Drive (HDD)s and 7 Solid-State Disk (SSD)s.

The OpenStack services that are provided by the cluster are:

- **Keystone**: Authentication, authorization and service discovery;
- **Glance**: VM images management;
- **Nova**: Virtual machines management;
- **Neutron**: Networking;
- **Cinder**: Block volume storage;
- **Heat**: Orchestration;
- **Ceilometer**: Customer billing, resource tracking, and alarming capabilities.

Different research projects are run on the cluster. Each project has assigned resources' quotas. The SE project has assigned 20 VCPU, 40 GB RAM, 200 GB for Volume (e.g. non-ephemeral) storage and 1 public IP. The two most useful OpenStack services for the prototype implementation are Nova for the creation of disposable VM and Cinder for the creation of network-attached persistent volumes.

Kubernetes cluster The prototype runs on top of a group of virtual machines provided by the CloudVeneto OpenStack:

- 1 Kubernetes Master node with 2 VCPU and 4 GB of RAM;
- 3 Kubernetes Slave nodes with 4 VCPU and 8 GB of RAM;
- 2 Kubernetes Slave nodes with 2 VCPU and 4 GB of RAM;

The cluster is running the version 1.5 of the Kubernetes orchestrator. All the services and components of the system are run on the Kubernetes cluster. Each of them runs as a separate Docker container. The orchestrator is responsible for the scheduling of the different containers on the slave nodes. The master node doesn't run any application container. It runs software that is essential for ensuring the proper operation of the cluster. To modify and deploy containers on the cluster it is necessary to interact with the master. Each instance has private IP that is internal to the SE project inside OpenStack. The Master node has been attached also to the single public IP provided by the OpenStack environment. The Kubernetes installation is aware of the presence of the public IP and it is possible to install and run containers that are open to the Internet.

Kubernetes provides three fundamental components for the candidate architecture:

- **DNS service**: there is a system container that runs an internal DNS server. The microservices and the other components can lookup the cluster internal IP address of the other containers. Every container defined in the cluster is assigned a DNS name. Multiple instances of the same service/container have the same DNS entry, and there is a load balancing mechanism in front of the instances;
- **Load balancers**: Kubernetes allows to create multiple instances of the same container and having a load balancer that manages the inbound requests. The load balancer works at the TCP/IP level, different TCP connections may be routed to different container instances;

- **Autoscaler**: this component allows to create automatically new container instances when the current instances are under heavy load. The load is measured by tracking the current CPU usage. The provided autoscaler mechanism works only by checking the CPU usage, but, there are many different situations where other metrics should be used, as illustrated in section 2.3.1.

Another component that is fundamental for the Kubernetes cluster is the Docker registry. The registry is not provided by the Kubernetes framework. It is part of the Docker platform. The public registry of Docker is the Docker Hub Registry. It provides ready-to-use images of common operating systems and applications. While by default Kubernetes searches the public registry, it is fundamental for the Fiorital information system to maintain a private registry, otherwise, the container images of the system would be public. The images may contain proprietary code or components internally to Fiorital.

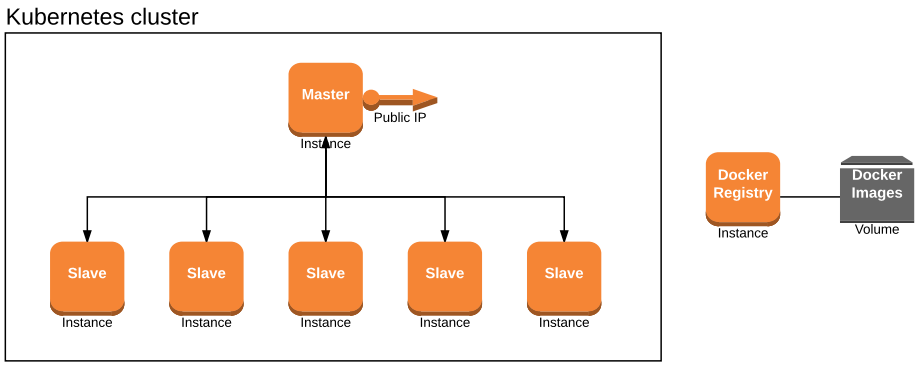

Figure 3.6: Cloud resources of the prototype.

Kong API Gateway and Crossbar mediator The Docker private registry is installed on a separate OpenStack VM instance. The images are stored inside a persistent network-attached volume provided by the OpenStack Cinder service. The volume is attached to the VM running the private Docker registry. The volume is necessary because, otherwise, any error that leads to a failure of the Docker Registry VM would cause the lost of all the Docker images.

Figure 3.6 shows the set cloud resources utilized by the prototype for the Kubernetes cluster and for the Docker Registry.

The API gateway and the publish-subscribe mediator are components that allow the presentation layer to communicate with the microservices (business and persistence layers). The two technologies that provide an implementation of these two components are Kong and Crossbar, as described in section 2.3.3 and 2.3.4. Both the technologies provide predefined public Docker images that allow a rapid installation on the Kubernetes cluster.

Both components should be accessible from the public Internet to allow the users to use the SE system wherever they are. Consequently, both the components are accessible through the public IP address.

Kong routes the HTTP requests from the presentation layer to the microservices. Kong routes the requests to the different services based on the URL of the requests.

Kong, to dispatch a request, uses the internal DNS service of Kubernetes to find the address of the load balancer of the service that should manage that request. Figure 3.7 illustrates the interaction between Kong and the DNS service.

Kong has to be instructed about what URL paths are managed by each service.

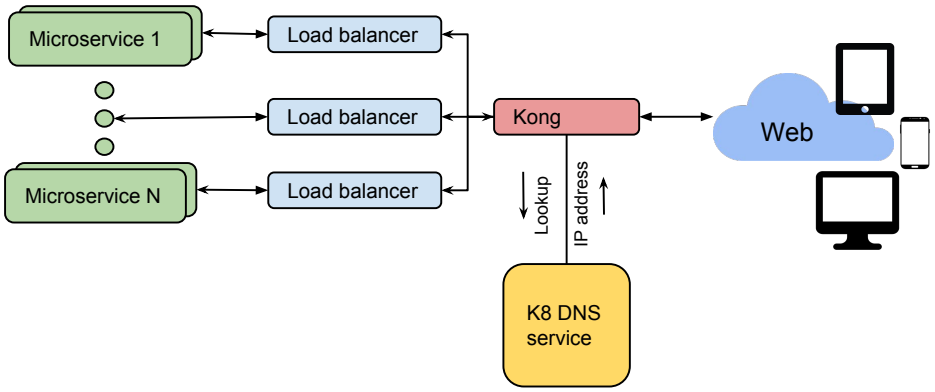

Figure 3.7: Kong gateway and the DNS service communication.

The Crossbar mediator sends push messages from the microservices to the clients connected to the system through the WebSocket and WAMP protocols. Crossbar maintains an open WebSocket connection with the presentation layer clients. The clients can subscribe to different topics they are intrested in. Furthermore, the microservices that need to send push messages to the clients keep the WebSocket and WAMP connection open with Crossbar. The mediator doesn't need to interact with the DNS service, because the services search and send messages to the mediator and not vice versa.

Microservices mesh The prototype should be based on the needs of Fiorital's and represent closely the company's scenario. Therefore, the prototype is composed of a mesh of microservices that represents the Fiorital's department structure. Furthermore, the APIs provided by the microservices represent the typical needs of the Fiorital's operators.

The prototype represents the needs of 4 core Fiorital's departments. the following list gives a description of the 4 departments and of their tasks:

- **Purchasing department**: deals with the management of the purchasing orders of fish from the providers. At the start of the day, the purchasing department usually asks to the suppliers the fish supplies availabilities and prices. The real purchasing orders are performed in a second moment, usually when there is certain number of Fiorital's customers interested on some fish products. Therefore, the main operations of a buyer are the insertion of new availabilities inside the system, checking the trend of the selling orders and inserting new sales orders;

- **Selling department**: deals with the management of the fish sales orders coming from the customers. The selling departments prepares and send a set of personalized catalogs for their customers, based on the past sales and on the current supplies availabilities. Later, they insert the sales orders into the system;

- **Logistical department**: deals with the planning and the control of the transportations of the fish products. The transportations are related to the purchasing orders and the sales orders. Fiorital doesn't own the trucks or boats used for the transportation, they usually rely on a set of transportation partners that provides transportation services for different routes. The transportations can be inbound when the product arrives at the Fiorital's platform from a supplier, or outbound when the products arrive at a customer site. Fiorital manages directly the platform, while they rely on the partner's warehouses for intermediate goods storing. The logistical operators manage in the system the information about the inbound and outbound transportations. The operators are also interested in receiving the information about the confirmed purchasing and sales orders;
- **Platform department**: deals deals with the warehousing of the products at the Venice headquarters. The platform situated at the headquarters is not a real warehouse, it's a small warehouse where the products should remain for a short time. The main interaction with the system for the platform operators is preparation of the schedules. It can be an unloading schedule for unloading the wares into the platform or loading schedules for loading the wares into an outbound truck.

As explained in section 2.2.3 the microservices are subdivided into two layers. The business layer and the persistence layer. The two layers subdivision is a typical technique in SOA, where there are entity and task services. A task service (business layer) is a form of business service with a functional context based on a specific business process. As a result, task services are not generally considered agnostic and therefore have less reuse potential than other service models. Task services are generally named after the business process they represent [78]. An entity service (persistence layer) is a form of business service with a functional context that is derived from one or more related business entities. Examples of business entities include order, client, timesheet, and invoice. Because their functional boundary is based on business entities, entity services are naturally agnostic to business processes. This allows them to be repeatedly reutilized in support of multiple tasks and business process, positioning them as highly reusable services [79].

The implemented microservices mesh is composed of both entity and task microservices. In total 18 microservices have been implemented. Eight of them are entity microservices, such as "Availabilities", "Catalogs", "SalesOrders". Ten of them are task microservices. Each task microservice is assigned to a specific department and part of its name is its department name.

Each service has an own execution time, that may differ from the execution times of the other services. In section 3.5 the execution times of all the microservices during the simulations is shown. The execution time of an API call is an estimation of the size of the data in the Fiorital's scenario. To each API endpoint a synthetic workload has been assigned.

Microservices implementation All the prototype's microservices are implemented with Node.js. The technology possesses a web server called Express. Each microservice exposes a REST interface to the upper layer. The handler of an API endpoint simulates an estimated execution time for that REST endpoint. The simulation of the execution times is performed through two mechanisms. The first is an empty *for* cycle that performs a certain amount of iteration (1000, 10000, ...). This methodology is used to simulate computational workload of the API endpoint. The second methodology is

used to simulate the interaction of the handler of the API endpoint with the database. The interaction is asynchronous, as the standard approach for Node.js. The simulation of this interaction takes place through a timeout mechanism. The timeout is an asynchronous mechanism that sets the arrival of an event inside the Node.js loop with a minimum delay.

One metric that should be recorded by a Node.js microservice is the requests queue length and the waiting time. The Express web server of Node.js doesn't provide access to its requests queue. For this reason, a microservice has been implemented using two Node.js processes instead of only one. The first process, the master, acts as a front-end that receives the remote requests and his duty is to only forward the requests to the second Node.js process, the slave. The slave process executes the handler of the remote requests and sends back the results to the master process. The master process sends back the results to the request's sender. With this technique, the master node can keep track of the queue length and of the waiting time. Figure 3.8 illustrates the master and slave processes.

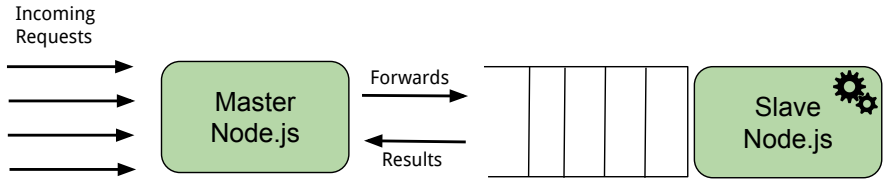

Figure 3.8: Master and slave Node.js processes of a microservice.

Every microservice records its response time, the queue length, the queue waiting time and its throughput. The data recorded by each microservices are gathered into one single ad-hoc process that acts as a logger. The microservices send their results to the logger.

MongoDB setup MongoDB is running inside the Kubernetes cluster with one replica-set of 3 replicas. Each replica is a different container with a network-attached volume for persistent storage. To properly configure the MongoDB replica set on Kubernetes a guide provided by the MongoDB website has been followed [80].

3.4.2 Benchmarking script

The script is executed on a dedicated virtual machine. It is implemented in Node.js. The script simulates the action of the Fiorital's operators. The operators can be of four different types, each type belongs to a different department. The total number of simulated application users is 80. This number is the sum of the maximum number of operators for each department that are working simultaneously during a typical working week at the Fiorital headquarters. The number of operators for each department and the performed actions of each operators are following described:

- **Buyers - 15**: they belong to the purchasing department. A buyer in the benchmarking script performs repeatedly the following actions:
  - Opens an availability for a supplier;

  - Adds 8 times, 8 different products to the supplier's availabilities;
  - Reads the supplier status;
  - Opens a purchasing order;
  - Adds 7 different products to the order;
  - Confirms the order;
  - Reads the order status.

- **Sellers - 30**: they belong to the selling department. A seller in the benchmarking script performs repeatedly the following actions:
  - Opens a new catalog for a customer;
  - Adds 8 times, 8 different products to the catalog;
  - Reads the catalog status;
  - Opens a sales order;
  - Adds 7 different products to the sales order;
  - Confirms the order;
  - Reads the order status.

- **Logistical operators - 20**: they belong to the logistical department. A logistical operator in the benchmarking script performs repeatedly the following actions:
  - Receives a push notification of a new confirmed order;
  - Creates a new trip;
  - Fills the trips with wares 8 different wares;
  - Confirms the trips;
  - Reads the status of a warehouse.

- **Platform operators - 15**: they belong to the platform department. A platform operator in the benchmarking script performs repeatedly the following actions:
  - Receives a push notification of a new confirmed trip to the Fiorital headquarters or leaving from the Fiorital headquarters;
  - Creates a new loading/unloading plan;
  - Fills the plan with loading/unloading steps;
  - Confirms the plan.

The operators' actions are simulated by the benchmarking script by calling the APIs provided by the prototype's microservices. Each simulated operator performs its set of actions periodically. There is a tunable interval time between one API call and the next one. The execution of the benchmarking script determines the start of a simulation. The execution time of the script can be configured.

The simulated operators aren't all created immediately, but, their number is gradually increased during the executions until the maximum number of 80 operators is reached. Therefore, the number of operators grows from 0 to 80. The increase may be, logarithmic, linear and exponential. This is another parameter that can be configured at the start of the simulation.

The script, while executing, measures the End-to-End response time of each request. The script measures also the source-to-destination response time of the actions that produce push messages received by other operators. Figure 3.9 shows the architecture configuration of the complete experimental setup with the name of the technologies used.

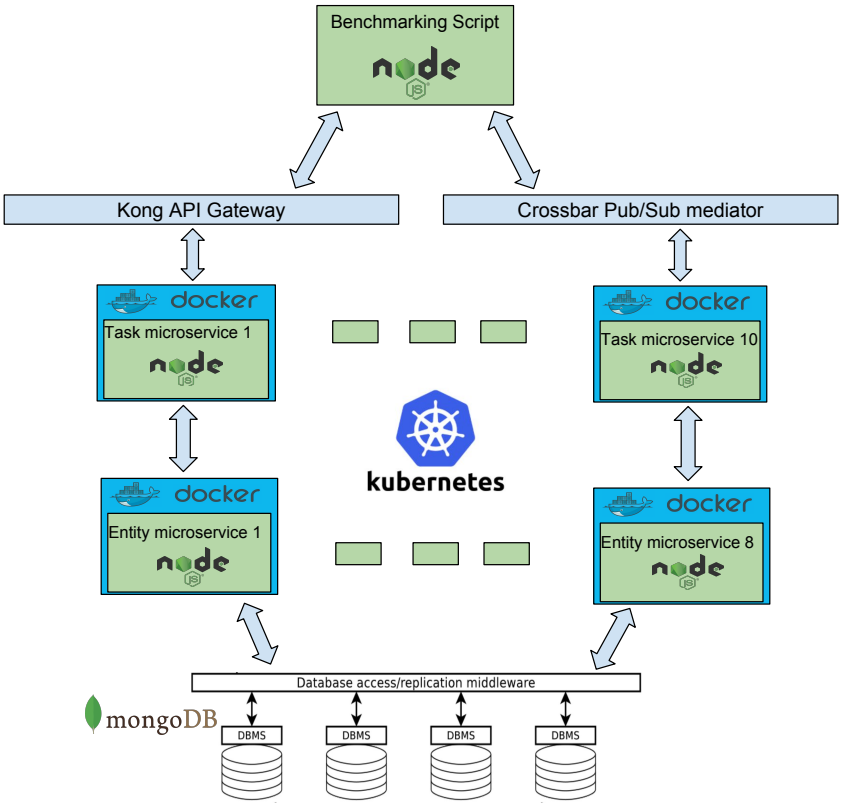

Figure 3.9: Architecture, components and technologies of the experimental setup.

3.5 Experimental results

The previous section described the experimental setup used to perform the simulations and to collect the results. This section presents the results of the executions of the synthetic workloads. The results are based on the metrics shown in section 3.1 and they measure the responsiveness, reactiveness and agility of the system. The results show that the architecture and the technology are effective and they score good results concerning the three architectural properties. The results are shown partitioned by architectural property. Indeed, this section has three subsections responsiveness, reactiveness and agility. The results highlight a problem with the Kong API gateway technology. This problem is shown and discussed in the responsiveness subsection.

The setup is composed of the prototype and of the benchmarking script. During the simulations, different sets of results have been obtained. The benchmarking script

allows configuring different parameters, as explained in section 3.4.2. All the execution have been executed for 10 minutes. The parameters that change between different executions are two. One is the operators' requests period (T), that has been set between the interval [6 seconds, 250 milliseconds]. This parameter allows increasing or decreasing the overall workload experienced by the prototype. The second parameter is the type of function used for spawning the simulated operators during a simulation. At the start of a simulation the number of operators is 0, then, it gradually grows to 80 over time. The increase from 0 to 80 is regulated by a function that can be logarithmic, linear or exponential. One feature of the system's architecture is the autoscaling mechanism, with this mechanism the system should adapt its resources usage based on the current workload. With the exponential function, the workload generated by the script grows exponentially. This aspect poses the system and its autoscaling mechanism under a heavier stress.

Before showing the results it's helpful to understand the flow of a single request through the components of the prototype. Figure 3.10 shows the typical flow of a single request sent by the benchmarking script. The request has a final answer (6.a) but generates also a push message sent back through the Crossbar publish-subscribe mediator (6.b). Each request arrives first to a task microservice (2), that performs some computations. Then the task microservice asks for data to an entity microservice (3). The entity microservice replies with the data (4). The task microservice performs other computation with the data and sends back the results (5).

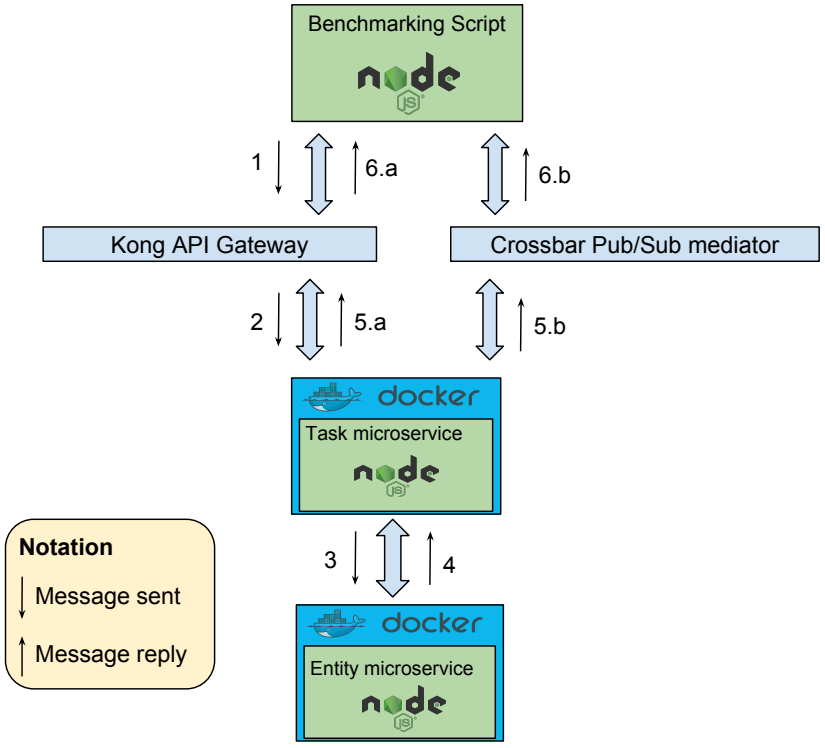

Figure 3.10: Message flow of a single request through the component of the prototype.

The measured End-to-End response time is measured by the benchmarking script.

The script measures the time elapsed between the dispatching of message (1) and the arrival of message (6.a). The source-to-destination response time is also measured by the benchmarking script. The script measures the time elapsed between the dispatching of message (1) and the arrival of the push message by Crossbar (6.b). Each microservice records their execution times per each request and step of the request. The task microservice records its execution times in the two steps (2,3) and (4,5). The entity microservice records its execution times in (3,4).

3.5.1 Responsiveness

One metric to evaluate the system's responsiveness is the End-to-End response time. It is helpful to understand which are the different timing steps that compose the final End-to-End response time. Figure 3.10 shows the steps. The execution times among the microservices is different. For this reason, their response time have been recorded as well throughout the simulations. The messages sent through the gateway and the mediator should have a fixed response time. Likewise, the delays imposed by the network, the networking protocols, and network virtualization protocols of OpenStack and Docker should have a certain size with values in a short interval. It is important to give a correct estimation of the delay time for evaluating the End-to-End response time.

Delay time estimation The estimation of the delay time has been carried out through a system's simulation where the synthetic workload is low. The End-to-End response time shouldn't be affected by any intense workload delay and therefore they should be more stable. To calculate the mean delay time the following formula has been used:

$$\mu(DT) = \mu(E2E) - 3 * \mu(MET)$$

Where DT = Delay Time, E2E = End-to-End response time, MET = Microservice Execution Time. The MET is multiplied by three because a single request goes through 3 steps where there are microservices calculations, as illustrated in figure 3.10. The synthetic workload used for calculating the DT has a single operator request period (T) of 6 seconds. The users' increment is linear. The period is quite big, the other simulations have been carried out with a lower period. The expected results should show a quite stable response time, however, different API calls have different execution times, so the response times for different requests types should be different. The difference shouldn't be too large and the maximum E2E response time shouldn't overcome 500ms. The $3 * \mu(MET)$ value should be lower than the value $\mu(E2E)$.

Figure 3.11 shows the boxplots for the E2E response time and the MET (Microservice execution time). The mean values are 93ms for the E2E response time and 16ms for the MET execution time.

Therefore, the mean delay time is:

$$\mu(DT) = \mu(E2E) - 3 * \mu(MET) = 93ms - 48ms = 45ms$$

The delay time accounts for almost half the total E2E response time. This is reasonable considering that a request flows through different steps with many remote components and services. The request spends much time inside the network links, being processed by the networking protocols and through the gateway. Figure 3.11 shows that the E2E response times vary inside a certain range of timings. This is a normal behaviour as different API calls have different execution times. There are also no very big values in

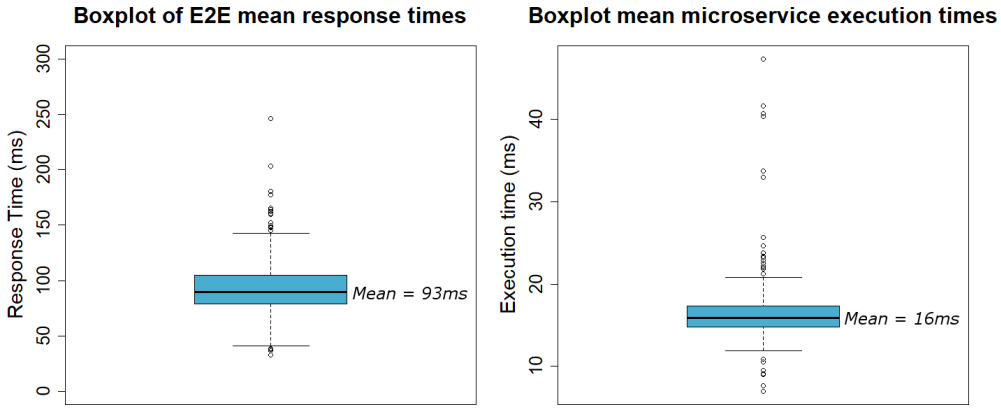

Figure 3.11: Boxplots of E2E response times and MET.

the E2E response times, also this aspect is normal given that the load of the executed simulation is pretty low.

Services execution times The prototype is composed of 18 microservices. It's helpful to see their mean execution times when the load is low. That execution time can be used as a reference for checking the degraded performance when the load becomes higher. The same synthetic workload of the previous results can be used, since the load of the execution is low. Each microservice has different API endpoints that can have different execution times. Therefore, it's expected to see microservices that have different execution times values for different requests. A single API call instead should have execution times that are all close to the same value. Figure 3.12 shows a boxplot of the execution times per each microservice.

There are some boxplots that have a wide range of values, others have values that are in a short interval. The services that have a short interval of values are composed of API endpoints that have a similar execution time. The services that have a wider range of values are composed of API endpoints with different execution times. For example, the first microservice "Availabilities" in figure 3.12 is composed of three endpoints that have different execution times. Figure 3.13 shows the three different execution times of the three endpoints. At the left, a boxplot with the execution times of each endpoint is shown. As expected, the three endpoints have response times that are all close to the same value. That value is different between them, that explains why in figure 3.12 the service "Availabilities" has a quite wide range of values. The three endpoints have different numbers of requests over time. The benchmarking script performs periodically a set of requests to different API endpoints over time. The diagram on the right shows that the response time per each API endpoint remains almost constant over the whole execution.

Synthetic workload - Linear - T = 2sec, 1sec, 500ms This paragraph shows the responsiveness results for three executions. The executions use a linear load increasing function. They differ from each other by the request's period used by the 80 simulated users. The benchmarking script sends requests for 10 minutes (600 seconds). The simulated users send requests with a specific period. For example in the execution with $T = 2sec$ each user sends a request every 2 seconds. Therefore 80 users sending requests each 2 seconds, meaning that the overall rate of requests sent by the script

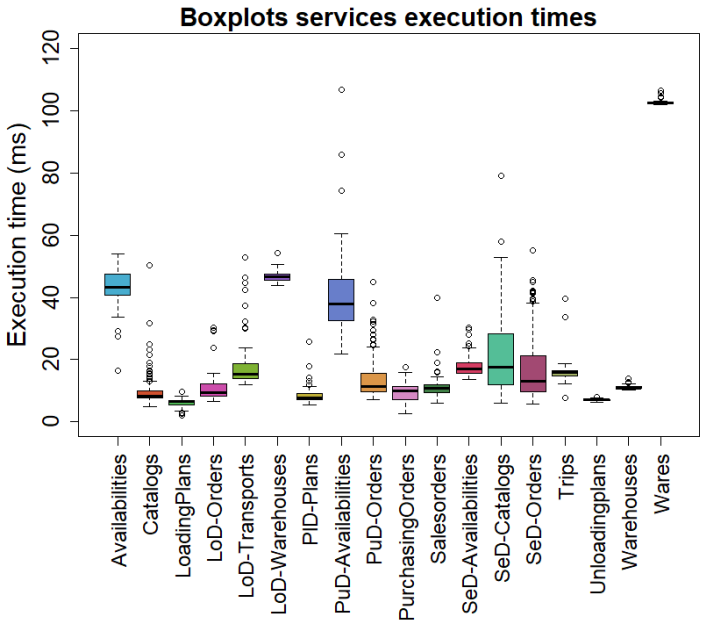

Figure 3.12: Boxplots of the microservices execution times.

is 40 requests/second. However, as shown in section 3.4.2, some users start sending request after a certain push event arrives. The logistical and the platform operators do their action in response to the events that notifies the confirmation of an order and the confirmation of a trip. For this reason, the requests rate should be lower, as there may be many logistical and platform operators that don't have any task to perform. It is expected from the simulations that the system's throughput doubles between one simulation and the next one. The response times should remain similar between different executions. The response times of a single execution should remain close to the same value over the execution time span.

Figure 3.14 shows the E2E response times of the executions and the overall system throughput of the executions. The throughput for the execution with period $T = 2sec$ is lower than 40 requests/second. The execution ends with a throughput around 33 requests/second. As expected, the throughput doubles between one execution and the next one. It is possible to see that for all the executions the throughput grows linearly until half of the time span (5 minutes). This behaviour is correct because the maximum number of system's users is reached around that moment. After that moment, the number of users remains constant at 80. The charts shown a spline of the single points. In all the charts there is a point per second. The E2E response times of a single second is the mean of the response times that took place in the time interval of that specific second. The throughput of the execution with T=2sec has points that follow two diverging lines. This aspect is caused by the fact that each user sends a request each two seconds, while the requests are aggregated per single second. Indeed, the other two executions with T= 1sec, 500ms don't present the characteristic. The mean response times remain quite similar throughout the simulations. However, the lines are wavy. The waving is caused by the changing periodical actions performed by the simulated users. In certain time spans a simulated operator performs requests to a certain API endpoint, while later, it makes requests to another endpoint. This aspect

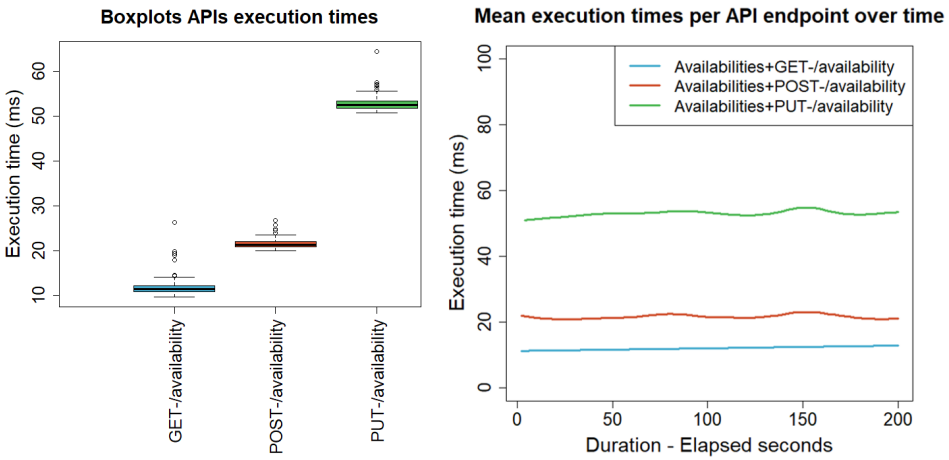

Figure 3.13: "Availabilities" microservice API endpoints execution times.

is clearly visible in figure 3.15.

The chart on the right shows the throughput of a subset of the system's microservices in the execution with $T = 2sec$. The chart shows that the throughput of some services follows the wavy pattern of the E2E response time. Specifically, it is caused by the seller operators. That makes different API calls in different time intervals. However, this aspect causes also short delays in the execution of the microservices. The services that are experiencing more load have longer response times and longer queue lengths. This is visible in figure 3.16. The queue lengths shown are sampled by the maximum queue length of a service in the interval of a second. Both the queue lengths and the execution times have the same waving of the services throughput. Meaning that many requests are causing some services to slow down. The slow downs aren't particularly severe, and the maximum queue lengths remain shorts.

The responsiveness results of the three executions show that the system remained responsive throughout all the three executions.

Synthetic workload - Linear - T = 250ms This execution has the same parameters of the three previous executions apart from the period $T = 250ms$. Each simulated user makes 4 requests per second. It's expected that the system throughput doubles compared to the execution with $T = 500ms$, while the E2E response time should remain similar to the previous executions. Figure 3.17 shows the E2E response time, the system throughput and the microservices execution times. By looking at the E2E response times, it's clear that the system is performing badly. The response time continues to grow linearly until the benchmarking script stops sending requests (600 seconds). However, after the 600 seconds of executions, other delayed replies continue to arrive at the benchmarking script. The delays grow to almost 4 minutes at the end of the execution. The delayed replies start to show up around the 200th second. Around the same moment, the system throughput stops to grow and actually it slowly decreases with a much bigger variance. The third chart shows the mean execution times of the microservices. Even tough the microservices start having delays around the same instant, the delays are not comparable to the E2E execution times. What is causing the delays is the Kong API gateway. The gateway notices the delays in the execution times of the microservices and the high number of requests coming from the

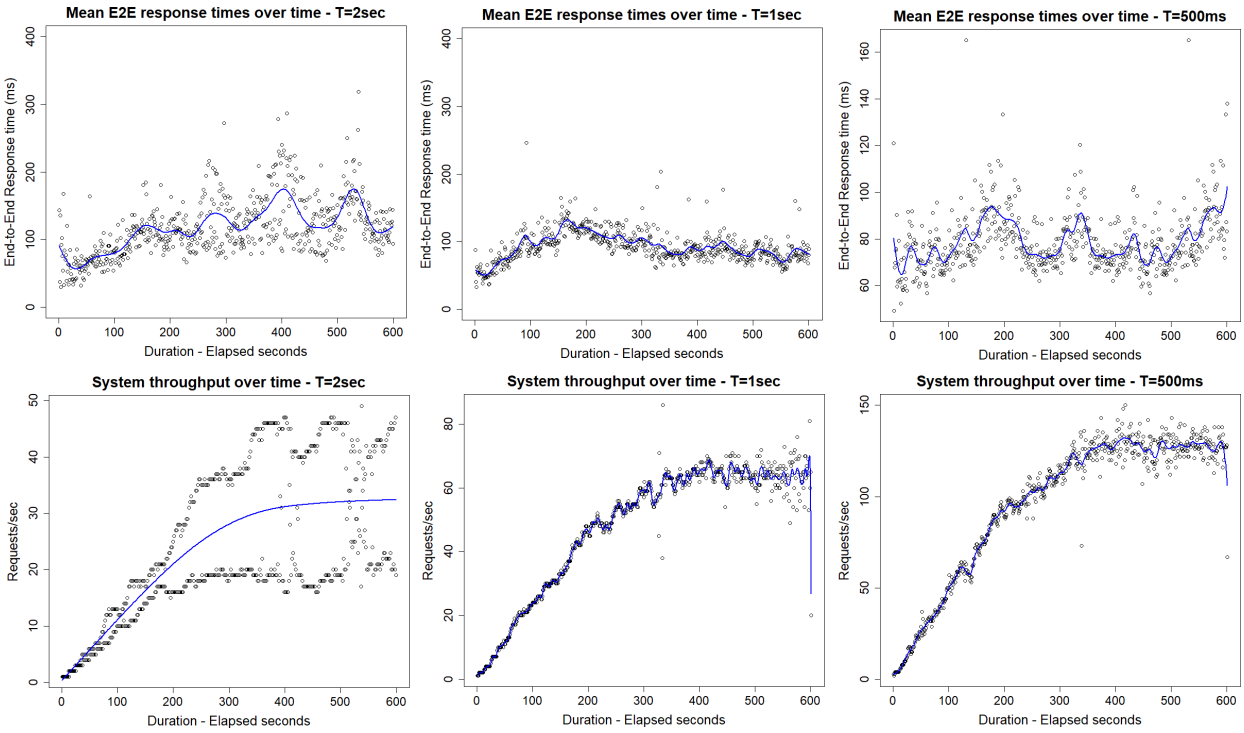

Figure 3.14: End to End response time and throughput - T= 2sec, 1sec, 500ms

same IP. Therefore, it starts queuing and rate-limiting the requests coming from that IP to avoid overloading the services of too many requests. It looks like that Kong is rate-limiting the system to a value around 160 requests/second. This behaviour is not in line with the candidate architecture of the system. It's a characteristic of the Kong technology. The behaviour isn't in line with the architecture because it hides from the orchestrator the high load that the microservices are experiencing. Consequently, the autoscaling mechanism doesn't work properly.

It is helpful to investigate deeper which microservices are introducing delays in the system. Figure 3.18 shows the execution time, throughput and queue length of a set of microservices. There is only one service that is experiencing delays. It is named "SeD-Catalogs", meaning that it's a task service of the Selling Department that deals with the customers' catalogs operations. The shape of the line of its execution times is very similar to the mean execution times of the microservices in figure 3.17. This aspect highlights that the "SeD-Catalogs" microservice is the main microservice that is slowing down the system. It is the microservice that receives the higher number of requests during the execution. The simulated seller operators are 30. The other types have less simulated operators. The "SeD-Catalogs" service is also the only one with long requests queues. At the end of the execution, after the 600 seconds, the services that are not involved with the selling department consume faster the queued requests by the Kong gateway. The third chart shows that four services involved with the selling department have more throughput at the end of the of the simulation. This is because there are more requests queued for that department than for the others. This also explains the peak in the response times in the first chart of figure 3.17. The response times are suddenly higher because the requests that have been queued longer

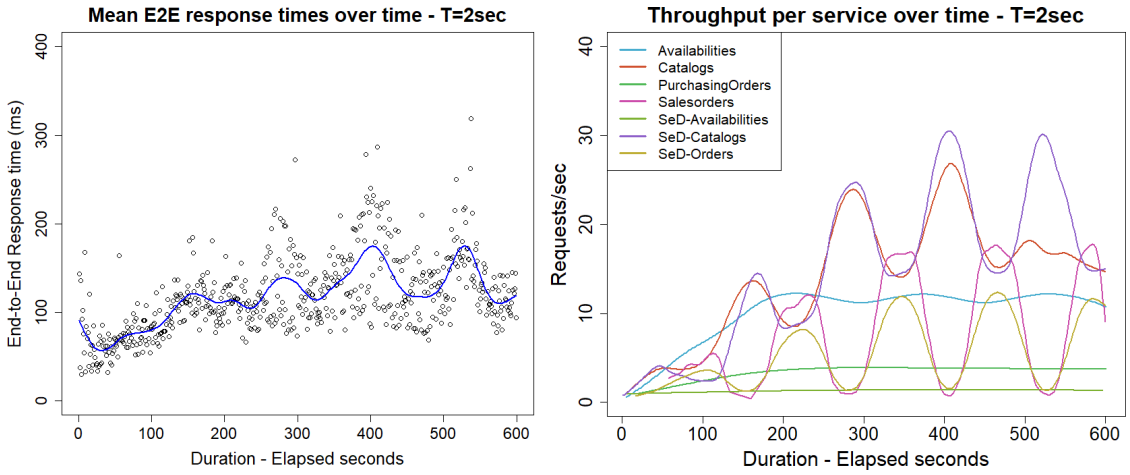

Figure 3.15: E2E response time compared to services' throughput - T= 2sec

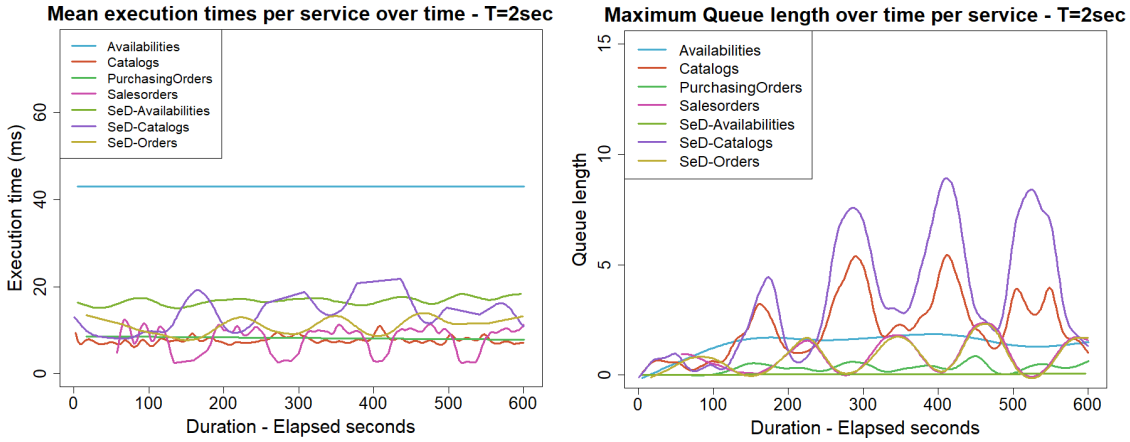

Figure 3.16: Services execution times and queue lengths - T= 2sec

by the Kong gateway are being drained off from its queue.

Synthetic workload - Exponential - T = 500ms The execution discussed in the previous paragraph showed that the prototype doesn't work adequately when the requests period is 250ms. The execution shown in this paragraph stresses the system by using an exponential users generation function. The number of users grows exponentially from 0 to 80 (the maximum). The requests period used is $T = 500ms$, that is the lowest described period with which the system works adequately.

The load generated by the benchmarking script should increase to the maximum in less than a minute. It's expected to see delays in the response times in the first part of the simulation. Later, the autoscaling mechanism should stabilize the response times to the normal values. The execution times of the microservices should be higher in the first part of the execution time, for the same reason.

Figure 3.19 shows the End-to-End response time, the system throughput and the microservices execution times throughout the simulation time span. The E2E response time and the microservices' execution times, as expected, present a significant increase

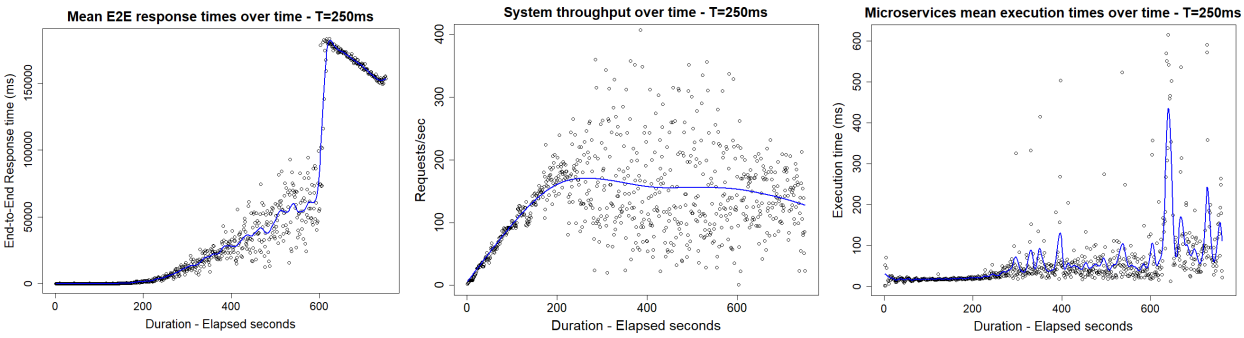

Figure 3.17: E2E response time, system throughput and microservices execution times - T = 250ms.

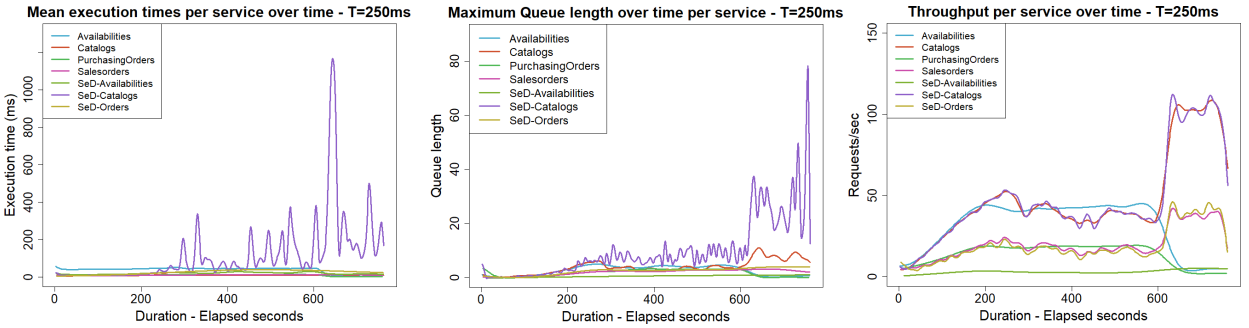

Figure 3.18: Execution time, throughput and queue length of a subset of microservices - T = 250ms.

during the first part of the simulation. The autoscaling mechanism is able to handle rapidly the quickly increased workload. The chart of the throughput shows how the workload increases exponentially during the first 60 seconds of the simulation. The overall system behaviour throughout the simulation is in line with the expected behaviour.

3.5.2 Reactiveness

Only one of the two sub-properties of the reactiveness property has a quantitative metric that measures its quality. The sub-property is liveness the corresponding metric is the S2D (Source-to-Destination) response time metrics, as discussed in section 3.1. The source-to-destination response time metric measures the elapsed time between the performed request of the operator A and the receipt of the corresponding push notification at the operator B. A good source-to-destination response time for a certain request should be very close in value to the E2E response time of that same request.

To check the liveness property of the system the E2E response time of two execution is compared to the S2D response time of the same two executions. One simulation is the one with exponential workload increase and period $T = 500ms$. The other one with linear workload increase and period $T = 1sec$. It is expected that the two types of response times have the same pattern of values in both the simulations. However, the two types of response time may be a bit different because not all the API endpoints generate a push message to another operator.

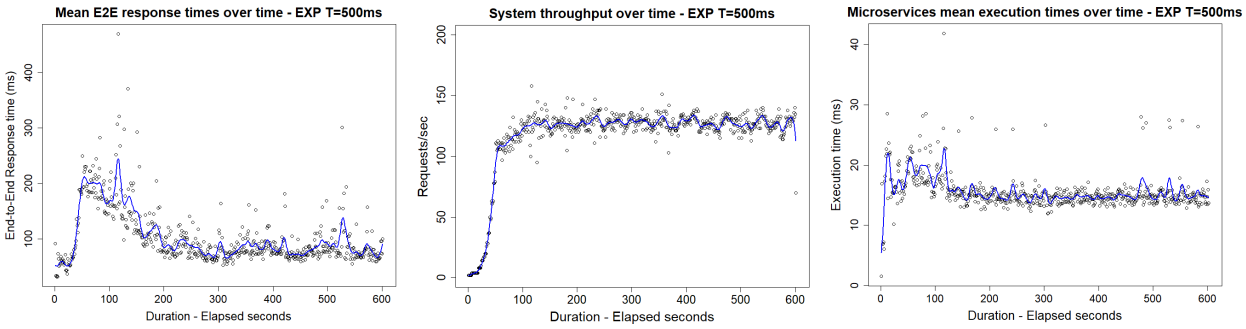

Figure 3.19: E2E response time, system throughput and microservices execution times - Exponential growth and T = 500ms.

Figure 3.20 shows the comparison of the S2D and the E2E response times in the two simulations. The response times are similar in both the executions, showing a good S2D response time. The S2D response time has a mean value that is slightly lower than the E2E response time. The motivations are two. The first motivation is that in the source code, the push message is sent by a microservice to the Crossbar mediator before the actual reply that goes through the Kong gateway. The second motivation is that not all the API endpoints generate a push message. For example, the service "Wares" that has the longest execution time, as shown in figure 3.12, doesn't generate any push message. Therefore, the overall S2D response time results a bit lower than E2E response time.

3.5.3 Agility

The two metrics for measuring the agility of the system are the downtime and the number of required re-configuration operations, as described in section 3.1. To measure both of them it's helpful to think about a scenario where the configuration of the system should be updated. The first paragraph shows the scenario and results of the downtime metric. The second paragraph shows the scenarios and the required re-configuration operation.

Downtime Evaluating the system's downtime can be performed by updating dynamically one service of a running simulation. The goal is to show that the update of a service causes no system's downtime. The orchestrator technology helps in achieving this goal. One running service is updated dynamically by creating the new instance, routing the new requests to the new instance, wait for the old instance to reply to all the requests and finally tearing down the old instance. These operations are performed autonomously by the dynamic orchestrator Kubernetes. The chosen execution is the one with a linear increase of the workload and a request period $T = 1sec$. The chosen service for the update is the "SeD-Catalogs" service. It is the service that throughout the execution has been experiencing the stronger workload among the set of microservices. The update is performed after 4 minutes of execution when the load generated by the benchmarking script has reached the maximum.

It's expected to see no request that has a very big E2E response time. No request should be lost by the system during the update. At the moment of the update, two instances of the service should be running, as the autoscaler should have already created another instance for the "SeD-Catalogs" service. The two new instances, that

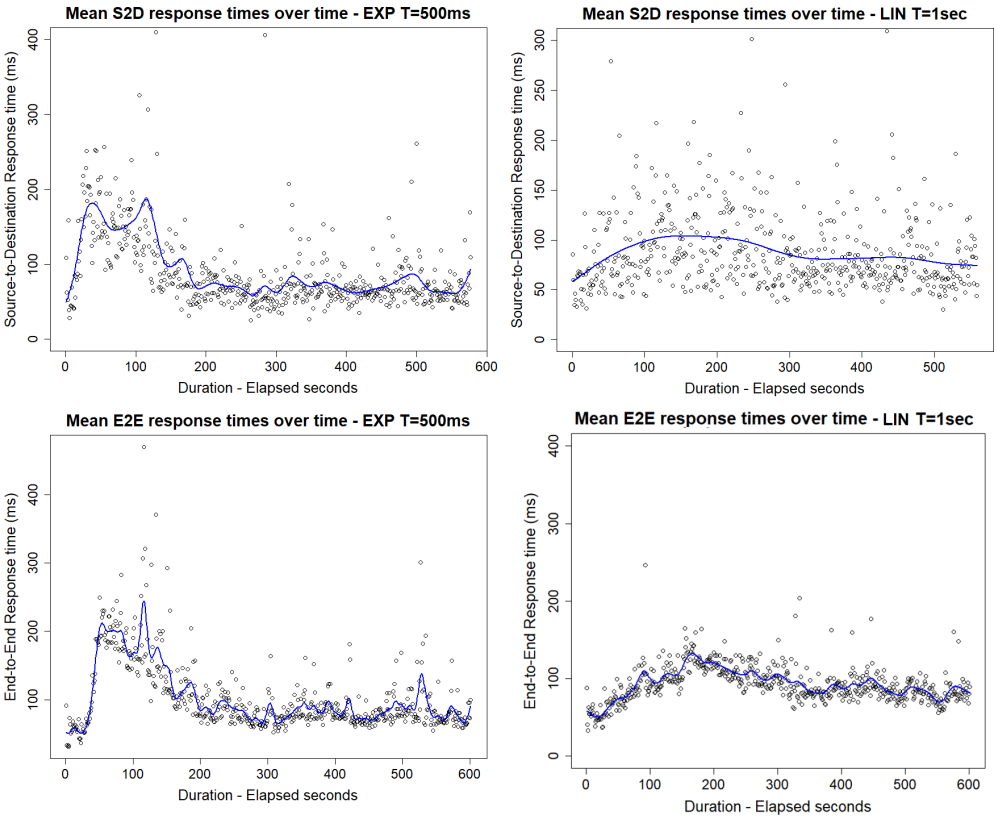

Figure 3.20: Comparison between S2D and E2E response times in two simulations.

replace the two old ones, should be created with a short time interval of separation. The dynamic orchestrator replaces gradually the running instances.

Figure 3.21 shows the E2E response time and the microservices execution time throughout the described execution. The two vertical red lines indicate when a new updated instance of the "SeD-Catalogs" service is created. The fact that there are two red lines, confirms that at the moment of the update two instances of the service were running. Both the charts show that there is a slight peak on the timings. It means that the replies to the requests made to the "SeD-Catalogs" service are presenting some short delays during the execution. However, no request gets lost and there are no errors during the execution. This fact highlights that the service has been updated without any downtime. Figure 3.22 shows that during the update, the queue of the "SeD-Catalogs" service gets slightly longer during the update. It means that Kubernetes, the dynamic orchestrator, makes some queuing of the incoming request for the service. It allows a smooth transition between the two versions of the service.

Number of re-configuration operations Two scenarios have been selected for showing the required re-configuration operations. One where a new component is inserted inside the system. The other scenario involves the removals of two microservices.

The addition scenario concerns the insertion into the system of a new entity microservice. For example, in the Fiorital's context, the system could be extended for supporting the planning of new production orders at the Fiorital's headquarters. Some

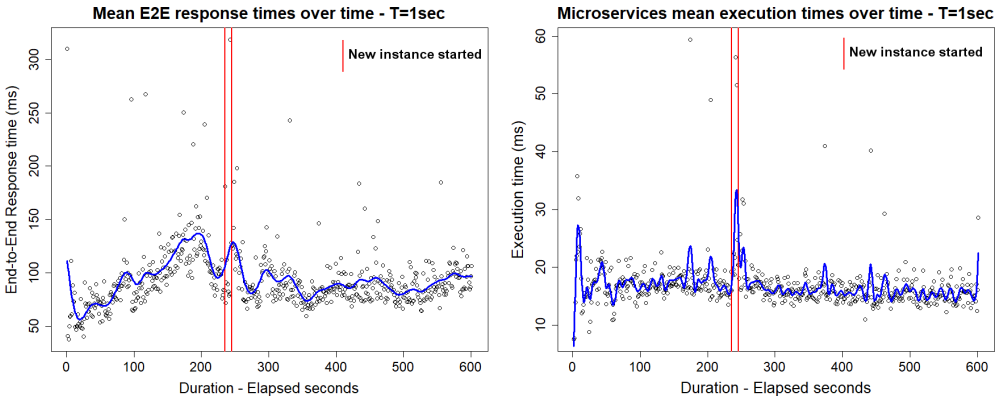

Figure 3.21: E2E response time and microservices execution times during a microservice update.

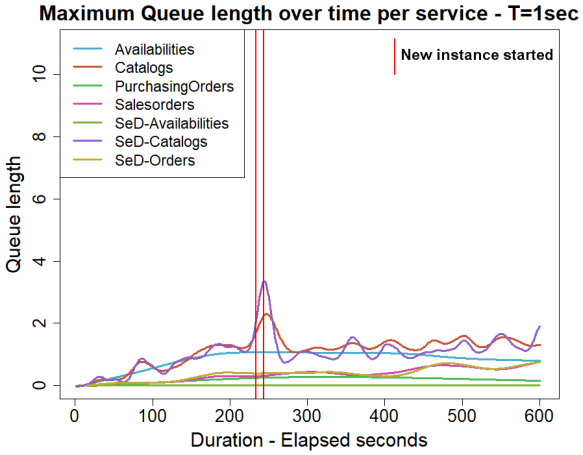

Figure 3.22: Queue lengths of a subset of microservices during an update.

of the wares arrived at the company's platform may be subject to production activities. Some goods are packaged into different boxes, to make them ready to be directly sold by the mass market retailers. The implemented prototype doesn't currently support the feature. For adding this new feature a new entity service called "ProductionOrders" should be inserted in the system. The task service platform department should be enriched with new API endpoints to make the features available to the platform operators. In order to perform the described changes the following manual operations are needed:

1. Creation of the Docker image for the new service "ProductionOrders" and update of the Docker image of a task microservice regarding the platform department;
2. Push the two images to the Docker Registry;
3. Creation of a Kubernetes declarative configuration file, describing the new container to be inserted;
4. Give a start command to the Kubernetes console for starting the new service

"ProductionOrders";

5. Give an update command to the Kubernetes console for updating the task microservice.

All the operations can be performed dynamically while the system is running having zero downtime. The operations are few in number, only five manual operations for inserting and updating two containers. The number of operations can be even lowered by having more automatic continuous deployment tools.

The second scenario is the one regarding the removal of the components. The scenario presumes that some features should be no more provided by the system. Therefore, one task service should be removed. The scenario presumes that also an entity service should be removed because it was used only by that particular task service.

The operations required for applying the re-configurations are:

1. Instruct the Kong API gateway not to forward anymore requests to the task microservice that is being removed. All the requested can be redirected to an endpoint that replies saying that those APIs aren't supported anymore;

2. Removal of the task microservice using the Kubernetes console;

3. Removal of the entity microservice using the Kubernetes console.

The operations can be performed dynamically while the system is running. The API gateway is a fundamental component for the removal operations, as it decouples the API interfaces from the microservices that are providing it. The number of required operations is low and they can be performed quickly by a system administrator.

Both the addition and the removal of components can be performed quickly and dynamically, improving the overall agility of the system.

Chapter 4

Conclusions and outlook

4.1 Recalling the thesis objectives

This thesis addressed the design of the software architecture for the new information system of the company Fiorital S.p.A. The company and the university are collaborating through the Smart Enterprise project that has the goal of designing an innovative information system. The project started in June 2016 and it's still ongoing. The main motivation behind the SE project is the will of Fiorital to renovate its IT system.

Traditional information systems present different flaws that cause several troubles to the system administrators and to the application users. Traditional systems require the system administrators to perform repetitive manual operations, some of them may require the production environment to be stopped, causing periods of downtime. Enterprise information system needs to be constantly updated for supporting the changing business activities performed by the company. Therefore, the system administrators have to perform frequently the maintenance operations. Another problem with the traditional systems is the lack of efficient scalability, that may cause the application users to experience degraded performance. Lastly, the traditional information systems have two flaws that the make the overall user experience worst. The first is that they are typically developed with technologies that don't provide its users with live dashboards that get updated autonomously by using push messages. The second flaw is that the information system is accessible only from company's devices that are placed in the company's offices. The SE project wants to obtain a system that overcomes these flaws. The list of the project's objectives is recalled in table 4.1

Several master students and three professors of the University of Padua have worked and are currently working on the project. The role of my thesis in the SE project is to design a good system architecture and to choose the technologies for implementing the Fiorital's new information system. The company wants to explore and understand which are the opportunities that the recent technologies could bring to them. The thesis objective can be summarised in the following three objectives:

1. **Architecture design**: *What is an effective software architecture for an information system that should satisfy the project's objectives?*
   The architecture of the software system is its backbone. It should be designed before starting to develop the system. Software architecture is about making fundamental structural choices which are costly to change once implemented.

2. **Technologies selection**: *Which are the best technologies for the implementation*

Project's objectives
O1 - Versatile
O2 - User interface accessible from different devices
O3 - Easily accessible information
O4 - Integration with new sources of information
O5 - Operators' tasks automation
O6 - Real-time analyses

Table 4.1: Recalling of the project's objectives list

of the system's architecture?
Given the designed software architecture, a set of technologies should be chosen for the implementation of the system and of its architecture. The technologies should be open-source and effective for the system that is going to be implemented. As shown throughout the document, the majority of the chosen technologies have been published recently and their usage for the development of new enterprise systems hasn't been studied much in the academic research. This fact highlights the need to study the technologies within the Fiorital scenario.

3. **Architecture and technology evaluation**: *Are the architecture and the technologies choices satisfactory?*
The last goal of the thesis is to implement a prototype of the designed architecture. The prototype is helpful for creating a set of results that shows the effectiveness of the architecture and of the chosen technologies. The prototype should be an implementation of the designed software architecture that represents the specific scenario of Fiorital.

4.2 Reviewing the accomplishments

This section discusses how the three goals presented in the previous section have been satisfied through the thesis work

1. **Architecture design**: the thesis work accomplished in finding a good architecture design for the new information system of Fiorital. Various activities have been carried out for obtaining the architecture. Firstly, the Fiorital scenario has been analyzed and studied. The structure of the departments, the needs of the departments and the current IT system have been studied. Then, a set of architectural properties, that the architecture should possess, have been defined. The architectural properties have been chosen by considering the project's objectives. The chosen architecture is based on the microservices architectural style and the container technology. The architecture has been designed by trying to achieve the best results in respecting the predefined architectural properties. The candidate architecture is profoundly different from the architecture of the traditional information system. The new architecture presents concepts such as

dynamic orchestration, containerization, distributed database and push messages to the end user. All these characteristics are helpful for accomplishing the SE project objectives;

2. **Technologies selection**: a complete list of technologies for the candidate architecture has been chosen. The architecture and its composing architectural layers require the usage of disparate technologies. The thesis describes a set of chosen technologies that are appropriate for the implementation of the candidate architecture. The selection has been carried out without making a complete analysis of the technological state of the art. Due to the fact that the technological scope is very wide. The chosen technologies and tools range from graphical libraries to DBMS to infrastructure provisioning and management. Cloud computing (OpenStack), Orchestration (Kubernetes), Docker containers and NoSQL database (MongoDB) are indispensable technologies for the implementation of the candidate architecture. Especially for improving the responsiveness and the operational agility of the system. All the technologies are open-source and can be hosted inside the existing datacenter of Fiorital.

3. **Architecture and technology evaluation**: the architecture and the technologies have been exhaustively evaluated. A set of quantitative results has been obtained. The results show that the candidate architecture is effective for the Fiorital's scenario. The majority of the chosen technologies are also appropriate. However, the Kubernetes, the Kong and the Crossbar technologies present some flaws, that slightly hinder the quality of the system. However, the technologies research activities didn't find any alternative technology that should perform better than the three aforementioned technologies. The evaluation has been carried out by utilizing the implementation of a prototype and the execution of a set of synthetic workloads on it. The executions produced a set of results using some quantitative metrics that are based on the predefined architectural properties. The results helped in achieving the final evaluations.

The thesis work individuated a valid candidate architecture for the new information system of Fiorital. The architecture can be a compelling architecture for many other enterprise scenarios where an information system is needed. The architecture is a good starting point for the future implementation of the new information system. Some of the technologies individuated are still not stable enough for being used in an enterprise environment. One example technology is Kubernetes. However, it is reasonable to think that the technologies will soon be stable enough.

4.3 Project outlook

This last section of the thesis presents briefly which are the limitations of the current thesis work and how they can be used as motivations for future improvements. These additional tasks could have been helpful to evaluate further the candidate architecture or for extending the architecture with new useful features. Secondly, this section presents some insights for future works that have been enabled by the thesis work.

4.3.1 Thesis limitation

IoT architecture One of the goals of the SE project is to have a system that automates as much as possible the operations of the company's operators. As discussed

throughout the thesis, the IoT devices would be helpful for making automatic some of the operations that are currently performed manually by the operators. The IoT part of the system has been only partially considered. The designed architecture within the thesis work allows the extension and integration of the system with the IoT devices. The network architecture of the IoT devices and which sensors and actuators to use are aspects that should be carefully designed. However, the system IoT infrastructure is not part of the thesis work, due to time constraints. A collection of IoT devices should be installed into the Fiorital headquarters and possibly also in some of the other warehouses that are controlled by Fiorital. The devices should all be connected to the Internet or the local network of the company, in order to communicate with the company's datacenter. The IoT field is continuously growing and gaining attention by companies. They are widely adopted in the management of the companies warehouses.

Layered database The distributed NoSQL DBMS used in the candidate architecture is a valid option for storing the data of all the departments of Fiorital. However, there is one exception. The accounting department of Fiorital should not be exposed to the eventual consistency characteristic of the NoSQL databases. For this reason, there could be an improvement to the current design by using an architecture where the database layer is divided into two sub-layers. The first, called adaptive layer, that runs the NoSQL database. The other one, called stabilized layer, running an ACID SQL database. The adaptive layer allows the system to have all the features described throughout the thesis, like scalability and agility. The second layer should be used only by the accounting department and should be populated with consolidated data, that are not going to change in the future. The accounting department should not be dealing with data that are eventually consistent. Some example data are the completed sales/purchasing orders.

The idea is to have a component that listens to the data that are being inserted into the adaptive database layer. The role of this component is to select which is the consolidated data that can be stored in the stabilized database layer. Figure 4.1 shows a draft of the layered database architecture. The figure shows the two databases and two additional components. One component, called "Stream generator", listens to the data that are being inserted/updated into the adaptive database layer. By listening to these data it generates a stream of data that can be listened and analyzed by other system components. The "Stream generator" allows selecting which data should be part of the generated stream. The second component, called "Stream analyser" reads the generated stream and selects which data are the consolidated data that should be stored into the stabilized database.

The layered database would allow the accounting department to use the same information system used by the other company's departments. Figure 4.1 shows just an idea of how to structure the layered database. However, a broader analysis and the design of the structure should be carried out.

Distributed testbed The benchmarking script used to generate the synthetic workload runs on a single node and in a single process. However, the application users are normally using the information system from disparate computing devices. Running the workload by using a testbed composed of various computing nodes would be a more precise simulation of the workload.

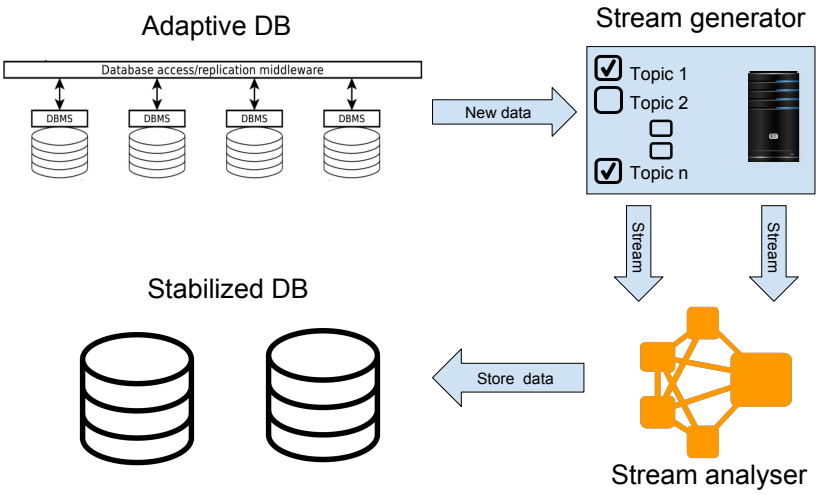

Figure 4.1: Draft of the layered database architecture.

4.3.2 Insights for future works

Data analysis One project's objective is to provide real-time analyses. The information system has a lot of data about the business activities that have been carried out in the past or that are currently ongoing. There are many situations where the Fiorital's operators would benefit from having the system performing autonomously and rapidly some data analyses. These analyses may suggest to an operator which suppliers to contact on a certain day or which customer should be willing to buy many products. There are other scenarios where this type of real-time analyses could be useful. One objective of the project is to identify these scenarios and implement algorithms that compute the results in short time. This type of analysis can be performed through machine learning techniques. There is a team of the university that is currently working on the data analysis part of the SE project.

The implemented prototype allows testing the data analysis directly on it. In this way, the data analysis team can create and test algorithms that are integrated with the rest of the system. Different containers performing different data computations may be easily installed inside the existing prototype.

Autoscaling metrics As shown throughout the thesis, the implemented system would highly benefit from the presence of a dedicated autoscaling mechanism. The current implementation uses only the CPU usage metric for determining when the microservices should be scaled in or out. This single metric is not sufficient for having an effective autoscaler. The current prototype implementation can be used as a starting point for designing and implementing a dedicated autoscaler. The new autoscaler may use different metrics such as the requests queue length, the execution times and the number of requests received.

Monolith migration The proposed candidate architecture is based on the microservice architectural style. The features that the system provides have to be divided into a mesh of microservices. Where each microservice has a defined and coherent duty. Performing this subdivision into microservices from the start may cause to create unbalanced and non-coherent microservices. It's problematic to have a clear overview of all the features the system should provide at the beginning of its design. It's more likely that new features will be added to the system while its development is ongoing.

For this reason, it a common strategy to design and implement first a monolith system. In this way, the domain can be fully studied before performing the subdivision into microservices. Fiorital is currently implementing its new information system without using the microservice architectural style. However, the subdivision into microservices of the current system under development could be an interesting future work to carry out. The monolith information system under development should present the majority of the features that the Fiorital's information system needs. Therefore, a complete design of the needed microservices can be performed by analysing the features provided by the monolith.

Acronyms

ACID Atomicity, Consistency, Isolation, Durability. 53, 92

AJAX Asynchronous JavaScript and XML. 43

API Application Program Interface. 21, 27, 29–32, 34, 36, 41–43, 45, 49, 55, 58, 63–66, 68, 71, 72, 74, 75, 77–80, 83, 86

AWS Amazon Web Services. 49

B2C Business to Consumer. 2

BASE Basically Available, Soft state, Eventual consistency. 54

BI Business Intelligence. 6

CaaS Container as a Service. 51

CAP Consistency, Availability, Partition tolerance. 53, 54

CPU Central Processing Unit. 11, 59

DBMS Database Management System. 28, 29, 32, 34, 36, 38, 44, 53, 54, 92

DNS Domain Name System. 67, 69, 71

FaaS Function as a Service. 48

HDD Hard Disk Drive. 68

IaaS Infrastructure as a Service. 35, 36, 62, 67

ICT Information and Communication Technology. 4

IETF Internet Engineering Task Force. 43

IoT Internet of Things. 4, 8, 10, 18, 92

IT Information Technology. 1, 3, 4, 11–13, 34–36, 51, 89, 90

LXC Linux Containers. 47

NIST National Institute of Standards and Technology. 35

ORM Object-Relational Mapping. 55

OS Operating System. 14, 15, 22, 35, 36, 42, 44, 46, 47, 63

PaaS Platform as a Service. 51

REST Representational State Transfer. 43

SE Smart Enterprise. 1, 3, 5, 7, 9, 14, 16, 20–22, 27, 28, 34, 36, 46, 50, 52, 53, 57, 69, 70, 89, 91, 93

SLA Service Level Agreement. 11, 17, 24, 54, 58, 59, 61

SOA Service-Oriented Architecture. 22, 72

SSD Solid-State Disk. 68

SSL Secure Socket Layer. 30, 67

URL Uniform Resource Locator. 43, 70, 71

VM Virtual Machine. 22, 40, 41, 46, 47, 69, 70

VPN Virtual Private Network. 20, 35

W3C World Wide Web Consortium. 42

WAMP Web Application Messaging Protocol. 44, 45, 55, 66, 71

WebRTC Web Real-Time Communication. 43

Bibliography

[1] "Image - logo fiorital." http://www.fiorital.com/. Accessed: 2017-04-18.

[2] "Image - fiorital trading routes." http://www.fiorital.com/it/index/animaecuore#logistica. Accessed: 2017-04-18.

[3] K. Bittner and I. Spence, *Managing Iterative Software Development Projects*. Addison-Wesley Professional, 2006.

[4] M. Weik, *Computer Science and Communications Dictionary*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[5] E. Nemeth, G. Snyder, T. R. Hein, and B. Whaley, *Unix®and Linux®System Administration Handbook*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 4th ed., 2010.

[6] T. A. Limoncelli, S. R. Chalup, and C. J. Hogan, *The Practice of Cloud System Administration: Designing and Operating Large Distributed Systems, Volume 2*. Addison-Wesley Professional, 1st ed., 2014.

[7] A. Johnsen and K. Lundqvist, *Developing Dependable Software-Intensive Systems: AADL vs. EAST-ADL*, pp. 103–117. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[8] "Martin fowler - microservices." http://martinfowler.com/articles/microservices.html. Accessed: 2017-05-27.

[9] S. Newman, *Building Microservices*. O'Reilly Media, Inc., 1st ed., 2015.

[10] "Continuos delivery website." https://continuousdelivery.com/. Accessed: 2017-06-07.

[11] M. Richards, *Software Architecture Patterns*. O'Reilly Media, Inc., 2015.

[12] J. Nieh, S. J. Yang, N. Novik, *et al.*, "A comparison of thin-client computing architectures," tech. rep., Technical Report CUCS-022-00, Department of Computer Science, Columbia University, 2000.

[13] M. L. Abbott and M. T. Fisher, *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise*. Addison-Wesley Professional, 1st ed., 2009.

[14] "Microservices gateway pattern." http://microservices.io/patterns/apigateway.html. Accessed: 2017-06-07.

[15] A. S. Tanenbaum and M. v. Steen, *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006.

[16] "Microservices access token pattern." http://microservices.io/patterns/security/access-token.html. Accessed: 2017-06-07.

[17] P. Mell, "The nist definition of cloud computing,"

[18] R. Ranjan, B. Benatallah, S. Dustdar, and M. P. Papazoglou, "Cloud resource orchestration programming: Overview, issues, and directions," *IEEE Internet Computing*, vol. 19, pp. 46–56, Sept 2015.

[19] "Private cloud automation, orchestration, and measured service." http://www.networkcomputing.com/cloud-infrastructure/private-cloud-automation-orchestration-and-measured-service/912817173. Accessed: 2017-06-07.

[20] R. Sampathkumar, *Disruptive Cloud Computing and It: Cloud Computing Simplified for Every IT Professional*. Xlibris Corporation, 2015.

[21] C. B. Weinstock and J. B. Goodenough, "On system scalability," tech. rep., DTIC Document, 2006.

[22] D. Beaumont, "How to explain vertical and horizontal scaling in the cloud." https://www.ibm.com/blogs/cloud-computing/2014/04/explain-vertical-horizontal-scaling-cloud/, 2014. Accessed: 2017-04-08.

[23] "Cloud computing slides at cornell university." http://www.cs.cornell.edu/courses/cs5412/2016sp/slides/. Accessed: 2017-04-18.

[24] N. R. Herbst, S. Kounev, and R. H. Reussner, "Elasticity in cloud computing: What it is, and what it is not.,"

[25] J. Varia, "Architecting for the cloud: Best practices," Jan. 2011.

[26] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *J. Grid Comput.*, vol. 12, pp. 559–592, Dec. 2014.

[27] H. Ghanbari, B. Simmons, M. Litoiu, and G. Iszlai, "Exploring alternative approaches to implement an elasticity policy," in *2011 IEEE 4th International Conference on Cloud Computing*, pp. 716–723, July 2011.

[28] "Openstack cloud." https://www.openstack.org/. Accessed: 2017-04-29.

[29] "Openstack usage by companies." https://www.openstack.org/user-stories/. Accessed: 2017-04-29.

[30] A. Corradi, M. Fanelli, and L. Foschini, "Vm consolidation: A real case based on openstack cloud," *Future Gener. Comput. Syst.*, vol. 32, pp. 118–127, Mar. 2014.

[31] S. Amatya and A. Kurti, *Cross-Platform Mobile Development: Challenges and Opportunities*, pp. 219–229. Heidelberg: Springer International Publishing, 2014.

[32] G. Kappel, E. Michlmayr, B. Pröll, S. Reich, and W. Retschitzegger, *Web Engineering – Old Wine in New Bottles?*, pp. 6–12. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004.

[33] "W3c - html5." https://www.w3.org/TR/html5/. Accessed: 2017-04-18.

[34] A. Juntunen, E. Jalonen, and S. Luukkainen, "Html 5 in mobile devices – drivers and restraints," in *Proceedings of the 2013 46th Hawaii International Conference on System Sciences*, HICSS '13, (Washington, DC, USA), pp. 1053–1062, IEEE Computer Society, 2013.

[35] M. Mikowski and J. Powell, *Single Page Web Applications: JavaScript End-to-end*. Greenwich, CT, USA: Manning Publications Co., 1st ed., 2013.

[36] I. K. Chaniotis, K.-I. D. Kyriakou, and N. D. Tselikas, "Is node.js a viable option for building modern web applications? a performance evaluation study," *Computing*, vol. 97, pp. 1023–1044, Oct. 2015.

[37] "Xmlsocket, as3." http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/net/XMLSocket.html. Accessed: 2017-04-18.

[38] J. Lundar, T.-M. Grønli, and G. Ghinea, "Performance evaluation of a modern web architecture," *Int. J. Inf. Technol. Web Eng.*, vol. 8, pp. 36–50, Jan. 2013.

[39] K. Shuang and K. Feng, "Research on server push methods in web browser based instant messaging applications," *JOURNAL OF SOFTWARE*, vol. 8, no. 10, p. 2645, 2013.

[40] "Why wamp?." http://wamp-proto.org/why/. Accessed: 2017-06-10.

[41] S. Tilkov and S. Vinoski, "Node.js: Using javascript to build high-performance network programs," *IEEE Internet Computing*, vol. 14, pp. 80–83, Nov 2010.

[42] "Web server survey." http://news.netcraft.com/archives/2013/08/09/august-2013-web-server-survey.html, 2013. Accessed: 2017-04-28.

[43] "Node.js." https://nodejs.org/en/. Accessed: 2017-04-28.

[44] "Kong api gateway github." https://github.com/Mashape/kong. Accessed: 2017-06-10.

[45] "Crossbar.io." http://crossbar.io/. Accessed: 2017-06-10.

[46] A. S. Tanenbaum, *Modern Operating Systems*. Upper Saddle River, NJ, USA: Prentice Hall Press, 3rd ed., 2007.

[47] K. Morris, "Martin fowler - immutableserver." https://martinfowler.com/bliki/ImmutableServer.html, 2013. Accessed: 2017-04-07.

[48] P. J. P. Menage and C. Lameter, "cgroups." https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt, 2015. Accessed: 2017-04-07.

[49] "namespaces(7), linux programmer's manual." http://man7.org/linux/man-pages/man7/namespaces.7.html. Accessed: 2017-04-07.

[50] S. Graber, "Lxc 1.0: Security features." https://stgraber.org/2014/01/01/lxc-1-0-security-features/, 2014. Accessed: 2017-04-07.

[51] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, vol. 00, pp. 171–172, 2015.

[52] K.-T. Seo, H.-S. Hwang, I.-Y. Moon, O.-Y. Kwon, and B.-J. Kim, "Performance comparison analysis of linux container and virtual machine for building cloud," 2014.

[53] "Docker documentation." https://docs.docker.com/. Accessed: 2017-06-28.

[54] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with openlambda," in *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'16, (Berkeley, CA, USA), pp. 33–39, USENIX Association, 2016.

[55] "What is serverless computing and why is it important." https://www.iron.io/what-is-serverless-computing/. Accessed: 2017-06-10.

[56] "Aws lambda." https://aws.amazon.com/lambda/. Accessed: 2017-04-08.

[57] "From vm to container to serverless." https://www.slideshare.net/welkaim/from-vm-to-container-to-serverless. Accessed: 2017-06-10.

[58] "Evolution of microservices - craft conference." https://www.slideshare.net/adriancockcroft/evolution-of-microservices-craft-conference. Accessed: 2017-06-10.

[59] "Welcome to bluemix openwhisk." https://new-console.ng.bluemix.net/openwhisk/. Accessed: 2017-04-08.

[60] "Google cloud platform, cloud functions." https://cloud.google.com/functions/. Accessed: 2017-04-08.

[61] "Microsoft azure functions." https://azure.microsoft.com/en-us/services/functions/. Accessed: 2017-04-08.

[62] "From containers to container orchestration." https://thenewstack.io/containers-container-orchestration/, 2016. Accessed: 2017-04-10.

[63] C. de la Torre, "Containerized docker application lifecycle with microsoft," dec 2016.

[64] R. Peinl, F. Holzschuher, and F. Pfitzer, "Docker cluster management for the cloud - survey results and own solution," *Journal of Grid Computing*, vol. 14, no. 2, pp. 265–282, 2016.

[65] "Nosql databases explained." https://www.mongodb.com/nosql-explained. Accessed: 2017-04-13.

[66] E. A. Brewer, "Towards robust distributed systems (abstract)," in *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, (New York, NY, USA), pp. 7–, ACM, 2000.

[67] J. Pokorny, "Nosql databases: a step to database scalability in web environment," *International Journal of Web Information Systems*, vol. 9, no. 1, pp. 69–82, 2013.

[68] D. Pritchett, "Base: An acid alternative," *Queue*, vol. 6, pp. 48–55, May 2008.

[69] "Nosql databases." http://nosql-database.org/. Accessed: 2017-06-10.

[70] "Developers' nosql skills." https://blogs.the451group.com/information_management/tag/nosql/. Accessed: 2017-04-18.

[71] "Why use mongodb instead of mysql?." https://www.mongodb.com/compare/mongodb-mysql. Accessed: 2017-04-30.

[72] "Mongodb documentation." https://docs.mongodb.com/manual/. Accessed: 2017-06-28.

[73] T. R. Norton, "End-to-end response-time: Where to measure?,"

[74] H. Jayathilaka, C. Krintz, and R. Wolski, "Service-level agreement durability for web service response time," in *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 331–338, Nov 2015.

[75] M. K., *Designing Data-Intensive Applications*. O'Reilly Media, 1st ed., 2017.

[76] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 205–220, Oct. 2007.

[77] "Cloudveneto openstack cluster." http://cloudveneto.pd.infn.it/ and https://cloud.cedc.csia.unipd.it/User_Guide/index.html. Accessed: 2017-06-10.

[78] "Task service." http://serviceorientation.com/soaglossary/task_service. Accessed: 2017-06-25.

[79] "Entity service." http://serviceorientation.com/soaglossary/entity_service. Accessed: 2017-06-25.

[80] "Mongodb installation on kubernetes." https://www.mongodb.com/blog/post/running-mongodb-as-a-microservice-with-docker-and-kubernetes. Accessed: 2017-06-25.