TÉCNICO LISBOA

Microservices Architecture for Gaming Industry Companies

Ensuring Scalability and Availability for Gaming Companies with Microservices and Hybrid Cloud

Sofia Maria Machado Estrela

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisors: Prof. João Nuno de Oliveira e Silva
Eng. Jorge Daniel Machado Vila Boa

Examination Committee

Chairperson: Prof. Teresa Maria Sá Ferreira Vazão Vasques
Supervisor: Prof. João Nuno de Oliveira e Silva
Member of the Committee: Prof. João Carlos Antunes Leitão

January 2021

Declaration

I declare that this document is an original work of my own authorship and that it fulfils all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

I would like to thank Chilltime and Daniel for bringing me this opportunity – it is not everyone that gets to make their thesis in such an interesting topic – while making me part of the team, providing me with all the resources to perform this work. I would also like to thank to each and every member of the Chilltime family for all your support throughout the year: Thank you for every meeting, every feedback and every joke in our daily scrum meetings.

I would also like to extend my gratitude to my supervisor Prof. João Nuno de Oliveira e Silva for accepting to work in this project and all invaluable input given (and patience) while guiding me to create this report.

Finally, a heartfelt thank to my parents and partner for not only their support in this semester, but for all strength given in the last five years that has made it possible to be where I am today.

Abstract

The video gaming industry has seen fast growth since its emergence in the 1960s. The surge has been accompanied by numerous technical challenges for both developers and administrators. It is an industry characterised by an increasingly demanding target audience, requesting high availability and frequent software updates. Thus, a company's infrastructure can play an essential role in the performance of their products and the productivity of their developers. In the recent years, the concept of microservices has reshaped the architectural field, with their modularised and distributed approach, having been successfully applied to massive multiplayer games. Nevertheless, more often than not, companies offer multiple games, meaning that microservices can have more to offer if the architecture is designed with a company-wide view. In this research, Microservices Architecture for Gaming Industry Companies (MAGIC) is proposed as an answer for businesses wanting to escape their limiting monolithic infrastructures and benefit from microservices to successfully impact both Quality of Service (QoS) for players, and the development and deployment cycles for the employees. One of the key concepts of MAGIC is the centralisation of common components across video games like player management, leader-boards and others. Another crucial feature in this proposed architecture is its ability to scale the most resource-intensive tasks to the cloud. By using a hybrid approach, companies can get the best of both worlds – potential infinite scaling while efficiently using the already existing company resources.

Keywords

Video Gaming Industry; System Architecture; Microservices; Scaling; Hybrid Cloud

Resumo

A indústria dos videojogos tem apresentado um rápido crescimento desde que surgiu na década de 1960. Este aumento tem sido acompanhado por variados desafios técnicos tanto para programadores como para administradores. Esta indústria tem um público-alvo extremamente exigente que requer alta disponibilidade dos serviços e atualizações recorrentes de *software*. Assim, a infraestrutura de uma empresa pode ter um papel fundamental na performance dos seus produtos e produtividades dos seus programadores. Nos últimos anos, os microsserviços têm revolucionado a área de arquitetura de sistemas, com a sua abordagem modularizada e distribuída, tendo sido utilizada com sucesso em jogos multijogadores massivos. No entanto, é comum as empresas disponibilizarem múltiplos jogos, o que significa que os microsserviços têm ainda mais para oferecer se a arquitetura for desenhada para a empresa enquanto um todo. Nesta trabalho de pesquisa, MAGIC (*Microservices Architecture for Gaming Industry Companies*) é proposta como uma solução para empresas que procuram ultrapassar os limites das suas infraestruturas monolíticas e de beneficiar dos microsserviços para impactar tanto a qualidade de serviços para os seus utilizadores como os ciclos de desenvolvimento e distribuição para os seus empregados. Um dos conceitos principais da MAGIC é a centralização de componentes comuns aos vários jogos, tais como a gestão de jogadores e tabelas de pontuações. Outra característica crucial é a possibilidade de escalar as tarefas computacionalmente exigentes para a nuvem. Utilizando uma abordagem híbrida, as empresas obtêm o melhor dos dois mundos – potencial infinito de escalabilidade e uma utilização eficiente dos recursos existentes.

Palavras Chave

Industria de Videojogos; Arquitetura de Sistemas; Microserviços; Escalabilidade; Nuvem Híbrida

Contents

List of Figures

List of Tables

Acronyms

MAGIC	Microservice Architecture for Gaming Industry Companies
API	Application Program Interface
OOP	Object Oriented Programming
QoS	Quality of Service
MUD	Multi-User Dungeon
FPS	First Person Shooters
RTS	Real Time Strategy
MMORPG	Massive Multiplayer Online Role-Playing Game
CD	Continuous Delivery
LB	Load Balancer
SOA	Service-Oriented Architecture
CD/CI	Continuous Development/Continuous Integration
IPC	Inter-Process Communication
REST	REpresentational State Transfer
HTTP	HyperText Transfer Protocol
ACID	Atomicity, Consistency, Isolation and Durability
BASE	Basically Available, Soft state, Eventual consistency
2PC	Two-Phase Commit
CQRS	Command Query Responsibility Segregation
ACL	Access Control List
VM	Virtual Machine
OS	Operating System
JAR	Java ARchive

CPU	Central Processing Unit
IaaS	Infrastructure as a Service
PaaS	Platform as a Service
CaaS	Container as a Service
FaaS	Function as a Service
IaC	Infrastructure as Code
DB	Database
CRUD	Create Read Update and Delete
UML	Unified Modeling Language
P2P	Peer-to-Peer
AMQP	Advanced Message Queuing Protocol
TCP	Transmission Control Protocol
IP	Internet Protocol
HMAC	Hash-based Message Authentication Code
AWS	Amazon Web Services
DLX	Dead Letter Exchange
TTL	Time-to-Live
ACK	Acknowledges

1

Introduction

Contents

1.1 Introduction and Motivation

In the 1960s, the gaming industry witnessed the boom of pinball arcade games. Since then, this industry has incurred numerous changes and enormous growth. From arcades to video games, to the more recent dedicated consoles and mobile games, evolution has been a constant – facing new challenges and opportunities, as the hardware evolves [1].

Multiplayer games date back to when users would take turns playing on the same physical machine. In 1993, with the Internet and the World Wide Web phenomenon, the gaming industry increased its growth rate again with the appearance of a new type of video games: networked gaming. Soon, online multiplayer games became very popular, enticing players to compete or collaborate, now in different physical devices [2, 3]. Nowadays, in a hyper-connected world where constant availability has become the new normal, mobile games have initiated a 'New Era' of gaming. A new class of hyper-casual-gamers emerged from the non-gamer population, playing video games on their smartphones [4] and thus drastically increasing the size of the video gaming market.

This rise in the number of players and platforms has created a number of challenges for the development and maintenance of online games, among which:

- Scalability – As games have an increasing number of users, the system needs to scale accordingly, increasing the resources available to tasks that, due to the load on the server, are having issues to perform correctly and timely. Moreover, games often have high peak times, where a high level of responsiveness is still expected to be maintained [5].

- High availability – Users expect to be able to play at any time of the day, with minimal delay. When latency and jitter are an issue, users rapidly lose interest in the game [6].

Due to these challenges, companies are required to rethink their infrastructures to ensure that they can always offer the best possible experience to their users.

The study here presented is conducted in collaboration with Chilltime, a Portuguese game development company also developing apps for other companies. Their biggest product is World War Online an online real-time strategy game that can be played both in the browser and smartphones (Android and iOS).

Like many game development companies, with the growth of the offered games and rise in the number of players, some performance issues surfaced, as the current structure revealed to be inadequate for the demand, especially in peak times. Moreover, no developer had a complete understanding of the system, due to its size and unstructured design, which, in turn, made work less efficient and of inferior quality. Concerns also arose as to fault tolerance, which, in their initial infrastructure, was mostly done manually.

At the commencement of this study, Chilltime had a few servers available for production purposes in a data centre. The company also had several servers in their offices, used to aid the team building and testing software updates. Their production structure was centralised and consisted of a monolithic infrastructure running on top of Linux servers, using Apache and MySQL.

In this study, Microservice Architecture for Gaming Industry Companies (MAGIC) – a distributed architecture which can be used to overcome the current challenges associated with online multiplayer games – is presented and its implementation exemplified.

The architecture proposed in this study offers a more distributed approach, using the concept of microservices to provide a modular and loosely coupled system, that can be effortlessly extended and updated, as well as easy to scale and distribute when needed. This architecture also helps companies deal with peaks in its load: offloading the exceeding requests to the cloud has become a possibility. Each service has a clear interface, which other services can interact with, as well as a clear objective/set of data that becomes that service's only concern.

To effectively apply microservices, the current monolithic system needs to be broken down into several smaller services. This decomposition cannot be copied from other companies – it is profoundly related to the industry and business model. Moreover, other support systems need to be ensured to improve the security, observability, and fault tolerance on the overall architecture.

Thus, in the present research, the chosen architecture – MAGIC – is being presented, as well as being explained how it follows the company's business model and how it was implemented to solve the issues from the previous infrastructure. Chilltime is now able to have a more flexible and scalable system, without the performance and resilience issues that were driving users away.

1.2 Organisation of the Document

In Chapter 2, it is explored the background theory around the gaming industry as well as the current architecture paradigms used – monolithic and microservices – along with their patterns of implementation, and respective challenges and countermeasures. On top, cloud services and providers are also analysed to deal with auto-scaling.

In Chapter 3, the requirements and objectives of Chilltime will be presented, drawing the foundations to Chapter 4 where the proposed architecture is explained using the C4 model approach. Throughout Chapter 5, the details of implementations are revealed and justified, as well as the next steps for the company. They are then evaluated in Chapter 6 to understand how they contribute to the objectives and requirements of Chapter 3.

Finally, Chapter 7 outlines and discusses the conclusions drawn from this research project.

2

Background

Contents

In this chapter, the relevant background information for this report is presented. It begins with the evolution of video games and the types of games that exist currently, to set context and introduce the specific constraints of the industry. It is followed by an in-depth exploration of the current architecture paradigms, focusing on the existing microservices patterns. Finally, this chapter finishes with a brief overview of the cloud services possibilities, advantages, as well as its disadvantages.

2.1 The Gaming Industry

2.1.1 Video Games Evolution

To better comprehend how games work today, it is essential to understand their origins. According to [2] and [3], throughout the history, there have been a few distinctive phases that have shaped this industry:

- 1958 – To make a tour to a laboratory less boring, the first video game was created using oscilloscopes and potentiometers to make tennis for two. Other similar games appeared around the same time, where the hardware itself was the game.
- 1961 – *Spacewar* was developed in a PDP-1 minicomputer and it's considered to be the first programmed video game.
- 1972 – Atari launched the first coin-operated machines with the game *Pong*, creating the first commercial video game.
- 1972 – The first home console was launched – Magnavox Odyssey – by a TV manufacturer.
- 1970s – Often designated as the video game's golden age, the 70s enabled the fast growth of the video gaming industry, with more consoles, hundreds of games and coin-operated machines.
- 1977 – Apple launched the first home computer, with a BASIC interpreter, opening the way for game development in non-dedicated devices.
- 1980 – Nintendo's *Game & Watch* was the first commercial hit with handheld devices. The combination of being small, cheap and portable promoted the beginning of the mobile video games revolution.
- 1980s – Emergence of the first networked multiplayer games, played in university networks or modem-vs-modem;
- Early 1990s – With the appearance of internal filesystems, it was then possible to reuse code and better organise the chaos of game developing. This encouraged a revolution in terms of how

game development was done, with the introduction of several data formats and behaviour systems. Promptly, generic game systems were created and the first game engines developed.

- 1990s – Due to the World Wide Web phenomenon, multiplayer games saw massive adoption. By adding other players to the game, many users found them more rewarding than their singleplayer counterparts.
- 2000 – With the appearance of the first cell phones with Java, opportunities for game development emerged. But the real advantage of mobile phones is that they are communication devices, which encouraged more online and multiplayer games development.

As it can be inferred, the three main business models in the gaming industry are: coin-operated machines in a paid-per-play model; dedicated consoles that after having the games purchased, could be played repeatedly; and personal, non-dedicated devices that could also run purchased games continually. The last two models remain until today while coin-operated machines have entered in decline.

For this research work, the most relevant types of games are the networked ones. In [7] and [3], it is presented a more in-depth history of this type of games:

- Multi-User Dungeon (MUD) games – MUD games are text-based multiplayer games in a persistent world. Players could interact with the world using some text actions such as ”move north” or ”pick/drop object”, as well as interact with other players in the same network. These games were implemented using a client-server architecture, where the client was a simple telnet session.
- First Person Shooters (FPS) games – FPS games like *DOOM* were played modem-to-modem. Players could connect and play real-time against each other in a non-persistent world. Originally, the implementation of these games was accomplished with a peer-to-peer topology using Ethernet broadcast (and thus affecting non-playing devices in the same network).
- Real Time Strategy (RTS) games – When Kali Inc. released software that enabled users to play games across the Internet allowing distant friends to play together, RTS games gained popularity with their intuitive interface, alongside a good AI and good network support to ensure that players are engaged in the game.
- Massive Multiplayer Online Role-Playing Game (MMORPG) – These games are very similar to MUD games, yet with graphics and much more functionality. They can be very network intensive and complex in code and architecture.

2.1.2 Types of Video Games

According to [8], the most popular way to classify video games is using genres. Genre classification is based on user observation of objective characteristics. It can be extremely useful in assisting users to

find similar games and helps advertising products more clearly. Some examples of genre classifications are Sandbox, Action-Adventure, First Person Shooter, Massive Online Multiplayer Role-Playing Games, Real-Time Strategy Games.

As reported by [9], the problem with game genres is the lack of mutual exclusivity or joint exhaustivity (to make sure there is one and only one classification for each game). Moreover, while the games' genre classification can be useful to aid the users in understanding the purpose of the game, it is usually not adequate to classify games regarding the technology running behind, making this classification not ideal for this research. The solution is, then, to present and define the most relevant characteristics of games in the context of this project and use their combination to classify the constraints referred.

Local and Networked Games

Networked games (or online games) can be defined as games that connect to another player's device or a central server [3]. Currently, most games are played online, even if singleplayer, with a connection to a server (if only to provide backup). A local game, on the other hand, is a game that can be played without connection to remote devices. Some examples are simple mobile games that can be played without Internet or video games in portable handheld consoles.

Singleplayer and Multiplayer Games

Singleplayer refers to games that are not influenced by other players' performances. Multiplayer refer to games where the other player's performance influences your own performance, for example, when playing a car's race against a friend. A game is also considered multiplayer when there are features like leader-boards, even if players do not play directly or simultaneously against each other, since these features enhance the competitiveness spirit in players and can therefore influence other users invested time in playing the game.

One common misconception is that multiplayer games are all networked games. In reality, multiplayer games also encompass having two input devices in a split-screen or having users playing in turns [3].

Mobile Games

Mobile games generally refer to the games that are played in smartphones, but the concept can be extended to other portable consoles.

World Persistency

As reported by [10], game world persistence exists when the virtual world created in the game exists without users needing to be present. In opposition, a virtual world is non-persistent when users need to be present for it to keep developing. Note that this definition is separated from data persistence, which refers to having the state of the world maintained even when servers crash.

2.1.3 State of the Art and Related Work

In this section, the related work done for architectural design in gaming companies is being analysed.

In terms of architecture, there is relevant research done in the context of MMORPG or FPS games, due to their number of concurrent players and network sensitivity. Solutions were created to send a reduced number of messages while still being able to synchronise the users in real-time, providing a seeming infinity world. Their infrastructure is defined by being Peer-to-Peer (P2P) or client/server.

Leaning to a P2P approach, there is the Mercury model [11] that designed a scalable and efficient event communication system, Colyseus [12] with a distributed architecture for a better user experience and Hydra [13] with an efficient network architecture with fault tolerance. While P2P had the potential to be less costly and quicker in terms of network latency it had disadvantages: challenging to control cheating; scalability issues with a large user base; unstable environment due to the lack of control over the client's latency and hardware resources, that required several complex fault tolerance mechanisms.

With these problems in mind, research was done regarding hybrid architectures: using both P2P and client/server, trying to collect the benefits of both approaches for the case of MMORPG [14].

The architecture presented by this research does not have elements of P2P – with the growth in network speeds and computing power, a client/server architecture is considered to be more secure, easier to code, and more flexible in terms of included functionalities. Architectural patterns such as microservices and the increasing adoption of cloud providers features, have enabled companies to scale more quickly and offload tasks to computing nodes closer to their end-users for an improved network latency.

On the other hand, research efforts regarding software architecture have provided solutions for processing, object modulation, data distribution, caching and other techniques to deal with the number of concurrent players and heavy processing [15–17]. These techniques are fundamental for efficient processing. However, they mainly focus on a monolithic architecture (with all the disadvantages associated) and on-premise implementations.

Regarding the usage of microservices in video games, there is research concluded in respect to the integration of game engines as microservices in the mobile cloud [18] as well as back-end architectures based on microservices for scalability gains and MMORPG games [19, 20].

While the last research topic referred is significantly related to the research done here, the scope is very different, as this work focuses on a company-wide architecture, rather than just one product. This is important to consider as many gaming companies offer more than one game and therefore, an aggregated architecture that centralises similar components will enable a more straightforward development, maintenance and even expansion. Moreover, this research work also discusses the usage of cloud providers to offload requests in a hybrid cloud environment for better resource management.

2.2 Microservices

When dealing with the creation of a project, the software has both functional requirements – related to the specific behaviour of the system – and quality attributes – describing performance attributes of a system [21]. While any architecture can be used to satisfy functional requirements, when it comes to the quality requirements, it is the architecture that can make a considerable difference [22].

Before proceeding any further, it is essential to introduce the concept of patterns. Patterns consist of four elements: a specific name, a common and generic problem, a description of a reusable solution, and the trade-offs and consequences of that solution. These patterns reveal proven strategies to solve common problems with well-known drawbacks [23].

In this section, the concept of a microservice architecture is being presented, starting by its definition and advantages/disadvantages, followed by the overall process of designing a microservice infrastructure. After, the most relevant microservice patterns are being introduced, regarding challenges related to communications, data handling, development cycle and ensuring a production-ready deployment.

2.2.1 Origin and Overview of Microservices

The microservice architecture surged as a natural response to the problems faced by growing companies with monolithic infrastructures. In essence, as described by Irakli Nadareshivili [24] it is about "building solutions with speed and safety at scale".

Monolithic Architecture

Monolithic architecture refers to the traditional way of serving software applications by encapsulating all functionality in one application [25]. This approach has been used by many companies worldwide including Netflix and Amazon, and it has several advantages – at least in elementary applications – like being simple to develop, easy to change, and straightforward to test and deploy [26]. Nonetheless, problems start to surge as companies expand, the most common being [22, 27]:

- With the codebase growth, it becomes increasingly difficult and time-consuming to perform changes (such as fixing bugs and implementing new features) as the code and features become progressively more dispersed throughout the existing code.
- Applications become too complex and too large for a single developer to fully understand, resulting in inferior code quality, which will in turn contribute to make later changes even more complicated. This may also increase the support that new developers need until they become mature members.
- Since a single change can influence other non-related features, the deployment cycles will take progressively a longer time due to the extended testing scope, more probability of bugs to fix and, consequently, difficulties in ensuring Continuous Delivery (CD) and a truly agile process.

- To scale a monolithic application, a Load Balancer (LB) is used to distribute traffic between several instances, scaling the system as a whole. This means that even if there is only one specific feature that requires more capacity, the entire system is replicated.
- Nowadays, it is crucial for companies to update and experiment with new technologies. This process can be complex, time-consuming and involve high risk which increases the probability of dealing with an increasingly obsolete technology stack.
- Reliability is difficult to guarantee since the whole system is affected should something happen to a part of a software or the server.

Service-Oriented Architecture (SOA)

The microservices architectural pattern appeared as a natural response to these problems – companies pursued more goal-oriented structures rather than solely a solution for a specific issue. The first concept that appeared (and deeply influenced microservices) was SOA [27]. SOA promotes the separation of features in small services, loosely coupled to each other, using network messages to communicate between them. This approach can deal with some of the problems of the monolithic infrastructure and promotes the reusability of software as well as easier maintenance and easier replaceability of services. The problem with SOA is the lack of clear direction on what is the best way to implement this architecture. Contrarily, the microservices architecture emerged from real-world usage and scenarios, creating a more refined approach on SOA's fundamental principles.

Microservices Architecture

A microservice is an autonomously developed and independently deployable component that implements useful functionality and has a bounded context with a very clear usage Application Program Interface (API) and supports interoperability through message-based communication [22, 24]. Microservices ensures two key concepts: Loose coupling and high cohesion [27]. The first ensures a degree of isolation between services, meaning that when updating or fixing a service, it is not required to change any other service. The second ensures that everything similar or related is positioned together, the rest being in a completely separate environment.

The benefits of microservices include:

- An effective way of scaling – Other than the horizontal scaling (by cloning) [28] there can be separation of work by responsibility and/or requesters. This separation of work is achieved by microservices and enables the optimal scaling process by scaling only what is needed. This incurs in reduced both infrastructure costs and the risk of capacity-related service outages.
- Improved fault isolation (Resilience) – Due to the modularity and loose coupling of services, the fault's area of impact can be contained as opposed to the monolithic application [29].

- Autonomous teams – Different teams can have specific specialisations, minimising the collaboration between teams, as services are very independent. This enhances the teams' efficiency and minimises scheduled system-wide downtime, which in turn ensures better availability [24].
- Technology heterogeneity and optimisation for replaceability – Since making changes is more effortless and involves less risk (due to the bounded scope of each service), companies have the opportunity to explore new software and languages, eliminating technology lock-in. This enables choosing the right tools for the right job, as well as readjust more efficiently to changes [27].
- Composability – ensure the reduction of development time through increased opportunities for reusability over time [24].
- Ease of deployment – since there is no need to test the whole system for every modification made, only the service that has been changed – this ensures that the company can take advantage of Continuous Development/Continuous Integration (CD/CI) [27].
- Increased Agility – with the ease of deployment and replaceability, it is easier for companies to adapt to changes and try new paths and features [24].
- Less complexity in the code – with more independent services that are smaller in size, it is easier for developers to work with the code, fixing bugs and extending/adding functionalities quicker [24].

However, microservices are no "silver bullet" and are accompanied by some disadvantages and real challenges in its implementation or migration from monolithic infrastructures. Some of the most substantial disadvantages of microservices are:

- A microservice architecture adds a lot of complexity to the infrastructure when compared to monolithic systems [26], creating challenges in terms of monitoring, testing, debugging flows as well as on the implementation of some of the microservices key features like distributed transactions and inter-service communication that can be challenging to both plan and execute.
- The increased number of services and the addition of a communication layer between them, will further add new sources of failure for the system that need to be dealt with [27].
- Migrating a system from monolithic to microservices may be a very time-consuming process, that requires the involvement of every team in its planning and execution [26]. It may also involve code refactoring for each service. It can take several years for the migration to be done as it is an incremental process, done in the addiction of the work that happens day-to-day.
- While unit testing (separated service testing) is easier to implement and automate, global testing can be more complicated to coordinate than as a monolithic infrastructure [29].

2.2.2 Decomposing an application in microservices

As explained by Chris Richardson in his book [22], defining an architecture is not a process to be followed mechanically, as it involves some creative thinking. Chris Richardson suggests the following three steps to decompose the application:

1. Start by defining all key requests done to the whole system, both commands to trigger actions and requests for information.
2. With the requests, decompose the architecture by routing the requests to different components.
3. Finally, determine each service's API in order to fulfil all the key requests of the first step.

While doing the division, it is also advised to follow the Single Responsibility Rule and Common Closure Principle (both rules borrowed from Object Oriented Programming (OOP) principles [30]). The main obstacles performing the decomposition are the number of calls (as too many requests can compromise the network latency), the choice of communication paradigms and finally how to maintain data consistency, issues that will be addressed later in this chapter.

2.2.3 Inter-Process Communication

As opposed to a monolithic infrastructure where components are invoked via function calls, microservices uses Inter-Process Communication (IPC) to interact [31]. Ensuring reliable and efficient communication is a crucial step in the planning process, impacting the availability and network latency of the services provided. There are two main paradigms to consider: synchronous and asynchronous.

In this section, both paradigms are presented along with their patterns for implementation. Finally, their advantages and disadvantages are compared to understand better when each should be used.

Synchronous Communication

This type of communication is defined by having the client making a call to a remote server and then blocking until the operation is completed. This is the easiest way to think and plan communication systems and it is greatly related with the one-to-one request/reply communication paradigm (where the client initiates the requests and waits for the server response). It expects a timely response from the service (as it is not ideal to have a blocked client for an extended period of time), so this approach is very sensitive to network conditions. In less ideal network situations, the blocking time can lead to unresponsive client applications [27].

For microservices, REpresentational State Transfer (REST) is the most relevant synchronous communication pattern. It is an architectural style for networked applications and when a service is based on REST it is considered a RESTful service. To ensure a RESTful service, the next six key constraints

should be respected [32]: client-Server architecture; uniformed interface; layered system; cache; stateless; (optional) code-on-demand.

With these characteristics REST enables loosely coupled applications that can effortlessly scale and work with internet intermediaries (like cache, proxies and load balancers) to ensure efficient use of the Internet [22,33]. The payload of the request and responses can be sent in various formats. Due to these advantages and simple usage and configuration (it does not demand any message broker service), REST has had a massive adoption on the Web.

On the other hand, the most significant constraints of REST applications are the requirement for stateless communications and usage of restrictive interface names; being only capable of request/reply communication; having only one-way communication (from client to server) that can lead to polling; the client needs to know the locations of the resources, meaning that some mechanism of service discovery needs to be implemented to ensure scalability and fault tolerance and, finally, being complicated to fetch multiple resources in a single request.

Asynchronous Communication

In this type of communication, the caller does not wait for the server to complete. This approach is aligned with event-based communication (where the client does not expect a response) [27].

Asynchronous communication is also capable of doing one-to-many communications like publish/-subscribe, which is a one-to-many version of the event-based notification, and publish/asynchronous response, where there is a limited waiting time for interested services to show interest by sending an asynchronous message back to the sender [31].

The most relevant asynchronous communication pattern is messaging [34]. In this paradigm, the sender writes the message to a channel and the receiver reads that message from the same channel. The message itself includes headers with metadata as well as a body with content and is sent to the sending port (open in a message broker) and, inside, directed to a channel. The consumer is connected to a receiving interface in the broker and receives the messages. The channel can be either point-to-point or publish/subscribe [35].

The benefits of messaging include loose runtime coupling (as opposed to synchronous communication); improved availability (due to the existence of a message broker); several communication paradigms supported (such as request/reply, notifications, publish/subscribe and more) [22]; reduced unnecessary network traffic between application such as REST polling calls [33]; no need to implement a service discovery mechanism when using a message broker and, finally, better option for long-running jobs and in low latency networks, as it does not block the client's application.

Regarding the drawbacks, they are: request/reply is more complex to implement than in a synchronised communication along with the additional complexity in the infrastructure due to the message brokers which might need additional expertise and resources [27].

2.2.4 Data Distribution, Consistency and Queries

In monolithic applications, databases and transactions are ensured to have Atomicity, Consistency, Isolation and Durability (ACID) properties, but this is not possible when considering distributed applications as the transactions may be done between databases across different services. As such, according to the CAP theorem – which states that from consistency, availability and network partition tolerance, only two can be assured [36] – consistency tends to be sacrificed and, therefore, it is not possible to ensure ACID transactions.

When it comes to data in distributed systems, each service should ideally have its own database (which can differ in type). Transactions can be ACID inside each service, but, when data transactions cross boundaries, isolation cannot be guaranteed unless the database is shared between services (which will tighten the coupling between the services) [33]. The solution is replacing ACID properties by Basically Available, Soft state, Eventual consistency (BASE).

Data Transactions

There are two main options when dealing with data transactions: Two-Phase Commit (2PC) [37] – which ensures ACID – and sagas [38] – which only ensure BASE.

2PC has a first step where every node in the transaction votes whether to commit or to abort. In the second step, each vote is counted and if all were in favour of committing, a message is sent to the participating nodes to commit, while if there is at least one abort, the message requests for the nodes to abort [37]. This warrants that every database in the transaction either commit or rollback. There are two main problems with 2PC: it is not supported by NoSQL databases, it is complex to implement using message brokers and the usage of synchronised calls can impact the availability and scalability of the services. As a result, when the complexity of the system requires, architects turn to sagas [38].

A saga is composed of various local transactions in sequence. When a distributed transaction starts, the first local transaction updates the database and triggers an event to the next local transaction in line. In case a local transaction fails, then the saga begins the execution of new local transactions that compensate/undo the previous local transaction performed in the saga [22].

There are two main saga patterns: Choreographed sagas and Orchestrated sagas. In choreographed sagas, each node of the transaction connects to the next node. In contrast, in orchestrated sagas there is the orchestrator concept, a centralised supervisor node that will notify each local transaction on what to do and manage the outcomes [27].

This pattern only ensures ”eventual consistency”. But sagas benefit from a higher scaling potential due to the services becoming less coupled during runtime, improving both availability and scalability that would be lost with a 2PC approach. On the other hand, sagas are considerably more complex to implement, being considered one of the most challenging aspects of the microservices migration [22].

Regarding the orchestration saga, companies using this should be mindful as to prevent the orchestration node from becoming a central logic point. Instead, using a choreographed approach can be more decoupled. The orchestration, however, facilitates the job of monitoring, logging and debugging [27].

Making Queries

Traditional data persistence use database's tables to map info and current state. The most limiting factor of this approach is the lack of history on the state changes, which has to be done in separate tables and is generally not "rollback-friendly" [22].

Event sourcing is an alternative approach that, instead of storing the structure's state, it stores the events that lead to the current state [39]. For performance purposes, a rolling snapshot (a state projection) can be maintained. There are various benefits to this approach: data is immutable and should there be an error, it is easier to issue a compensating change instead of trying to rollback the latest alteration, there is full auditability, and there is the possibility of calculating state projection at any given point in time, without special coding [24].

However, event sourcing alone is not be enough to solve the problem of ensuring data persistence, avoiding database sharing (and tight coupling) and ensuring data isolation and encapsulation. When dealing with requests that require data that is spanned across multiple services/databases, they can become quite complex using the traditional approach – API composition pattern [40]. With API composition, the requester sends messages to the service's APIs endpoints to ask for the needed information. With the response, it can combine the data received to get the final required information. While this approach is rather elementary in terms of understanding and implementation, if applied in more extensive queries, it may become inefficient in terms of memory [22].

Alternatively, there is the Command Query Responsibility Segregation (CQRS) pattern [41], which defends that requests and updates should be dealt separately. When using CQRS, one service can subscribe to the events of other services and, with an event sourcing approach, it will be able to maintain a rolling snapshot of all relevant information. This way, when interested services request that distributed information, it can query only one service, using the rolling snapshot to directly retrieve the response. This enables the complete separation of the data models on each service, keeping them independent and loosely coupled. The biggest downside is the complexity involved (both in event sourcing and CQRS), and for that reason, they should only be implemented in the presence of clear benefits [27].

2.2.5 Impact on the Development Cycle

CD/CI in microservices can be more easily achieved, which in turn reduces the time needed to deploy changes. However, for these advantages to be possible, there needs to be some automation involved, as well as operational maturity.

The following principles are essential to guarantee the correct deployment cycle in microservices [33]: continuous integration, continuous testing, continuous delivery and continuous monitoring.

Continuous integration implies that developers should commit their work frequently to the main branch. The most significant advantage is that errors can be detected earlier. Also, by having frequent integrations, the number of changes per integration is small. The sooner errors are detected and the smaller the changes involved, the easier and less costly for the company to find and fix them.

Continuous (automated) testing is beneficial due to being able to avoid unexpected disruption of business, ensuring more reliable applications, as well as being also very effective in creating proactive problem prevention.

As for continuous delivery – performing small releases frequently –, most of the advantages are related to the existence of building pipelines which automate the process of deploying the developer's code to production, executing the several phases of testing and ensuring the new release is production-ready before deployment [27]. These pipelines in microservices should be simple to use and available on-demand for teams.

Finally, the continuous monitoring principle ensures that the services are functioning optimally after the releases. Moreover, it enables using real-time reports to find bottlenecks and the most impactful optimisations. With the correct automation, thresholds can be established for the number reported, that once exceeded, can notify the company about the existing problems.

Testing

Testing is a crucial feature in a microservices architecture, that enable fast and continuous delivery in addition to increased agility and quicker reaction to changes. Microservices bring more flexibility and opportunities when designing the testing process when compared to monolithic. There are different types of testing differing in the scope of the analysis done [22]:

- Unit tests -— includes testing to classes and its functionality;
- Integration tests -— comprises testing to verify the service integration with other infrastructure;
- Component tests —- includes testing for an individual service or feature.
- End-to-end tests —- Acceptance tests for the entire application. If end-to-end testing goes smoothly, the service can be deployed to a production environment.

The bigger the scope of the tests, the longer they run, and the slower they reveal their results. Because of this, testing should be done in phases and in an iterative manner, starting from the smaller scoped testing until end-to-end validation. By having different phases, the errors can be spotted quicker than in a monolithic application where testing is usually done mostly on the biggest scope of the application. Due to gradual increase in the scope, it can be easier to find the error when it is detected.

Before closing this section, it is worth mentioning that testing is a highly context-dependent feature of microservices and thus, the systems should be built regarding the company, customer and how teams and developers work. Furthermore, testing can be a highly Central Processing Unit (CPU) intensive task, so these systems should be designed with the appropriate cautions [33].

Building Pipelines

Building pipelines allows for continuous testing and deployment process, by automating and chaining together the different phases of testing, and ensuring developers' changes are deployed to production. While it may not be possible to provide fully automated pipelines in every situation, it is essential to take into consideration that the more automated they are, the quicker is the release of new features.

In monolithic infrastructures, new releases tend to be a joint effort from several teams, that can take days to be tested and ensured it is production-ready. With microservices and this approach to testing, pipelines mechanisms should be accessible by teams directly, that can launch releases independently, without fear of breaking the application.

Deployment

Microservices are independently deployable, allowing the partial scaling of only the services in need, contrarily to a monolithic infrastructure where the application is scaled as a whole [22]. This operational flexibility can open up deployment options, profoundly affecting the scaling costs and efficiency.

Cloud providers also play a role. If a service is deployed on-premise, scaling hardware can be extremely costly due to the upfront costs of physical servers. It also need to be available before it is needed, running the risk of having idle hardware for too long and, consequently, not being worth the investment. Another option, with less up-front costs, is deploying the services in the cloud [24].

Nowadays, there are many deployment platforms available. In the next paragraphs, the most common ways of deployment are being presented and compared.

The first deployment being explored is using a language-specific packaging format in a physical server [42]. To deploy a language-specific package (like Java ARchive (JAR) for Java applications), the server needs to be configured, the packages copied and the service started. These steps can be automated in a deployment pipeline that previously builds, tests and invokes the production environment management service to deploy the newest version. Although this approach is fast to deploy and can ensure an efficient resource utilisation, there is no ability to control the resources used by the application on the server, no isolation between instances, no encapsulation of the language used and the production environment management service can be complex to implement [22].

A more advanced deployment option are Virtual Machines (VMs) [43]. They offer the emulation of a computer system and there can be several virtualised environments in the same server, but from the perspective of the processes running inside, it is as if they are running alone on a physical server. To

deploy services as VMs, the package is copied to an Operating System (OS) image, creating a new image that can efficiently run in various VM managers of different servers. This approach offers the encapsulation and isolation that was lacking in the previous one. Moreover, this can be deployed in the cloud with Infrastructure as a Service (IaaS). However, VM can be worse regarding resource utilisation and the process of creating an image can be slow, impacting the deployment process duration [22].

Another deployment strategy is the usage of containers [44]. Containerisation is a deployment technology which is more recent and lightweight. Just as VMs, containers are virtualised environments that encapsulate and isolate the services, having their own root filesystem and networks ports. However, due to its virtualisation mechanism, it does not suffer from the slow OS boot time at the launch of the container instance. Comparing to VM they are significantly fast at creating images and distributing them, by storing in registries. The biggest drawback of containers is that the company using them is still responsible for their administration unless they use cloud Container as a Service (CaaS) solutions [22].

It is worth mentioning the existence of container orchestration software [45], that treat physical machines as pools of resources and can run the needed instances of each service on top. The orchestrator is in charge of all resource management, scheduling – selecting where is the best place to run the service – and service management. It also ensures that both instances and servers are healthy, providing fault tolerance mechanisms. Another benefit is their ability to perform updates on services with zero downtime, by creating new instances of updated containers and only stopping the older version when the new ones are considered healthy. In case of problems, a rollback can easily be done.

Finally, the last deployment platform explored are serverless solutions [46]. Every deployment platform that has been described previously requires some system administration knowledge. Moreover, even when using the cloud provided implementations, the company needs to pay for VMs and containers running even if they are idle. Serverless is a deployment pattern that consists of using public cloud serverless deployment mechanisms. Benefits include the easy integration of these services with other cloud providers' services, the elimination of many infrastructure administration tasks, and a more flexible approach with a usage-based pricing. However, there are some drawbacks: there can be high-latency in the response due to the time taken by cloud providers setting up the application to start, and the limited application for long-running services and asynchronous messaging [22].

2.2.6 Production Environment Considerations

One pattern that is commonly observed in the majority of microservices' implementation is the API Gateway [47]. This service is generally observed in the edge of the microservice infrastructure and represents the overall interface that users can access. This node is essential to provide security (as it is the first node accessed by users) and should be able to route requests to the according services, therefore having some type of service discovery mechanism [24].

In terms of security, even public APIs must have protection mechanisms in place. As for routing, API gateways need to be able to send requests to the right node. Even with all dynamic address attribution features in the microservices implementation for scaling and fault tolerance, there needs to be a discovery mechanism that will allow to find the services and route the requests correctly.

Other than API Gateway, for an application to be production-ready, there are some additional features that need to be taken into consideration: security and observability [22].

Security

The three most important security features are authentication, authorisation and auditing. Authentication defines the ability to identify a user, while authorisation referes to making sure a user can access requested information or has the authority to ask for a specific task. Finally, auditing is about tracking the users' action inside the app to detect security issues and help with customer support.

To authenticate a user, the application typically verifies the user's credentials. To implement authorisations, implementations usually use role-based security mechanisms to aggregate similar users and create Access Control List (ACL) in order to grant these roles access to the needed resources. A critical pattern that is adopted in microservices implementation is Access Token pattern where the API Gateway that authenticates the user will add a header to the requests with the access token of the client, through which it is possible to identify the user and its roles [48].

Observability

There are several ways of analysing what is happening – as well as what has happened – inside the services. Due to the distributed architecture, logs and metrics are distributed as well, and it may be challenging to have a system-wide perspective. This feature is essential if the company wants to be able to debug the errors that will inevitably occur in the production system and might span across various services [22]. In microservices, observability is ensured by having centralised services that receive this information from services, to store and even to react to the data collected [27].

The first pattern is the log aggregation [49]. It is a centralised node that collects logs generated by every service. Usually, this can be done by implementing an event sourcing mechanism that will publish logs to the logging service, manage them and, possibly, add searching and filtering mechanisms on top, for easier comprehension. When certain events are received, this service can also send an alert.

Another observability pattern is the health check API service [50].In order to detect cases where the service becomes unhealthy (for example, the server is working, but the application has stopped) the health check API can help as it does periodic checks on the system's services health. To do so, every service must implement a `GET /health` API endpoint, returning if the service is healthy.

Additionally, the Application Metrics services pattern [51] is another observability pattern that collects metrics from each service, providing aggregation, visualisation and alerting. Like the logging services,

an event sourcing approach can be used to send metrics to this service, which will then be parsed and put in graphics. The visualisation can help with problem detection as well as analysing the current state of the system and finding patterns and bottlenecks, facilitating profiling and debugging.

Finally, distributed tracing is a pattern that allows debugging request flows [52]. As requests may travel thought various services, it can make its debug complicated in case of errors. To trace requests, each request should be given a unique ID and the information on its flow needs to be collected in a centralised node that provides filtering and visualisation, helping to understand the requests' flow.

2.3 Using Cloud Providers

Cloud services are considered to be the next stage in the evolution of the Internet. Throughout the next chapter, its impact will be evident and the different cloud services available will be presented.

2.3.1 Cloud Services

Despite the name, the cloud itself is composed of interconnected data centres, spanned across the globe, with networks, storage, several services and many interfaces that have enabled the creation of a new business model: to sell computing as a service, much like water and electricity utilities [53]. There is a lot of variety in cloud services and their most significant advantages lie in offering services and servers at scale that are highly available, with ingrained fault tolerance and infinite potential scalability, while being able to deploy those around the world. If a company wants to achieve these capabilities, there is a lot of infrastructure and management involved, and consequently, it would be a serious investment. But with cloud systems, the price paid by the companies is related only to what is consumed by them.

Cloud can be divided into public and private cloud. Public cloud is provided by cloud providers like Microsoft, Google or Amazon. In contrast, private cloud is the adaptation of the ”resource pool” point of view that is characteristic of cloud services through platforms like OpenStack [54]. There is also the concept of hybrid cloud solutions, where companies use their on-premise resources but expand them to the cloud to extend functionality or to have infinite scale potential.

Public cloud runs in a multi-tenant environment (servers are shared between clients) and end-users do not control where and how exactly their services are being run on the provider's data centre, as a result of the abstraction layers built-in these services. An exposed API can be used to access the resources and run commands. Other than the potential infinite scaling and utility pricing, with the public cloud the user can also ”outsource” infrastructure expertise and focus on the core competencies of their own businesses. Although, as microservices, cloud computing in no “silver bullet” and these benefits come with a cost: the loss of control on part of the users regarding the way their services are deployed, secured and available [55].

2.3.2 Cloud Computing

One of the most important cloud service provided is cloud computing, which offers deployment platforms for companies substituting on-premise data centre servers. There are different cloud computing options, providing different layers of abstraction on the underlying system.

Infrastructure as a Service (IaaS)

This service provides the thinnest layer of abstraction. As all physical management is hidden, companies use APIs to interact with their outsourced system. IaaS provides VMs and, therefore, administration expertise is still needed to install, manage and patch solutions on the system [55]. IaaS support additional features like having a LB routing traffic between VMs instances as well as auto-scaling options. Payment is related to the type of resources available in the VMs allocated (such as CPU, memory, disk space) as well as how much time instances are kept online.

Platform as a Service (PaaS)

On the next level of abstraction is PaaS, which abstracts the OS layer, removing the system administrator role [55]. PaaS is directed for developers, hiding all hard labour regarding servers, networks, operating system, storage and even more complex solutions like logging, monitoring, caching, email and similar tasks. This way, the developer only has to focus on the business logic. PaaS users have less control on the application environment and developers are confined to the languages and libraries supported by the service. On the other hand, the cloud vendor will provide a highly available, fault-tolerant and scalable application without hassle for the company.

Container as a Service (CaaS)

CaaS [56] is a newer solution that stands in the middle of IaaS and PaaS . Due to being a virutalisation mechanism, much like VMs, the proximity to IaaS is trivial to explain. But containers can also be considered for its positive impact on both development and deployment aspects, focusing on a more CD/CI-based approach. Their lightweight images represent the concept of Infrastructure as Code (IaC), which results in a deployment option that shares many similarities with PaaS but without the constraints of needing the cloud provider to support specific languages or frameworks [57].

Function as a Service (FaaS)

FaaS refers to cloud services that run code but without the developer needing to deploy the server application [58]. As a trigger is sent to the cloud provider, the function will start performing and the payment is calculated based on trigger and computing resources used while processing. Benefits of FaaS include diminishing the time spent on deployment, easy usage, low cost, potential infinite scaling and no

infrastructure maintenance. However, there are drawbacks when using this approach: no control over the infrastructure; not designed for long-running processes; complex to develop for statefull functionality; possible start-up time to consider when triggering a function which is not ideal for time-sensitive tasks. On a final note, when considering a cloud provider for FaaS, it is critical not only to consider the languages and platforms offered but also the triggers supported by each vendor.

Serverless

Serverless computing is a deployment platform where the user does not manage any underlying infrastructures, as if there were no servers involved [58]. All infrastructural choices are hidden under a layer of abstraction. The concept of serverless is significantly related to FaaS, sometimes the terms even being used interchangeably. However the distinction is essential, as recently there have appeared certain serverless services that run on containers, offering the same abstraction layer but for container images instead of actual code, such as Amazon Web Services (AWS) Fargate [59].

2.3.3 Cloud Storage

There are various solutions in the cloud to store data. To choose the right service, the characteristics of the data (such as performance, volume, retention period) need to be taken into consideration. There are two main types of Databases (DBs) to consider [55]: relational DBs and non-relational (or NoSQL) DBs.

Relational DB are the most familiar choice of DBs for most developers. Cloud Services offer the exact same capabilities as the local versions: ensuring ACID transactions and Create Read Update and Delete (CRUD) querying. As for NoSQL DB, only BASE transactions are ensured but they are more easily scalable than relational DBs. NoSQL DBs have various implementations suitable for different tasks, such as Key-Value Store, Column Store, Document Store and Graphs.

Inserting data in the cloud should be a pondered decision due to security. Ultimately, it is a decision that boils down to the confidence that a company deposits in the cloud providers. The vendor lock-in is also a significant factor to consider as while uploading data is free, the download/migration of high volumes of data is a very costly process, making it difficult to change the provider.

2.3.4 Other Services

Apart from computing power and data, cloud providers offer various services regarding three main areas: security (such as API Gateways, Access Control and Single Sing On solutions); monitoring (which includes logging and metrics systems as well as container orchestrators); and AI (which incorporates Translators and Text-to-speech APIs). New functionalities are added every year by providers, allowing companies to outsource several areas of their work and allowing them more availability to focus on prospering their businesses.

3

Objectives and Requirements

Contents

As it was previously stated, this research work was done in collaboration with Chilltime. This chapter is introduced by an analysis of the initial situation of the company, followed by the obstacles faced that cause the company to seek change. Afterwards, the main objectives for the architecture reformulation are presented, and from there the requirements and constraints to be considered when making architectural decisions are derived.

3.1 Objectives

Chilltime is a game development company based in Portugal. As briefly mentioned in Chapter 1, they offer several games that are available on different platforms (browser, mobile and desktop). Moreover, they also develop other products in areas unrelated to gaming for other companies. Over the years, the company's user base has grown, and the services provided have outgrown the initial expectation.

Chilltime has three main games: World War Online (RTS browser game), Marble Adventures (mobile puzzle game) and Soccer Avatars (mobile quiz game).

The first is the oldest game of the company, has the biggest user base and suffers the most from the limitations of Chilltime's architecture. In World War Online, players battle against each other with the goal of building the biggest empire. An example of a battle can be seen in Figure 3.1.

Figure 3.1: Example of a match in World War Online

There is a singleplayer version and a multiplayer version of these battles. The battle itself is a turn-based event, that when triggered for singleplayer – which is played against an artificial player – is immediately processed. In the case of multiplayer, once the battle is triggered, it is only scheduled for processing two minutes later, giving the opportunity for the player in defence to change the positioning of their units to better protect the area. This match processing mechanism was initially developed in

PHP, on the backend, but was recently renovated to a Unity server. Unity [60] is a professional game engine used for developing games that is able to target several platforms (such as Linux servers) and easily integrate with 3D models, animations and physics. Figure 3.1 corresponds to this newest version of the battle engine.

Players can also interact between each other in cooperative relationships, by participating in squads, that collectively try to help each other defeating other players and squads. These alliances are one of the most significant features that keep players engaged in the game.

Chilltime also interacts with their players via email and app notifications to both mobile phones and desktops. These communication channels are essential to keep players updated and engaged.

The original architecture for Chilltime's products was monolithic. As such, in the office there were a few servers available for building and testing purposes. Additionally, the company had four production servers in a data centre:

- One running a webserver, with the multiple web applications and API available;
- Another running the database service;
- Other with the pre-production environment for testing, as well as some cronjobs monitoring;
- An idle one.

This infrastructure posed some challenges for Chilltime. The majority of these production servers are not new, requiring updates – a challenging task to execute without making the services unavailable in the given architecture. Secondly, there were also no redundancy, failure recovery or backup mechanisms, which would result in very convoluted situation in case of a server failure.

As a result of the company's growth, some efficiency problems appeared due to the increased load on the servers. As a consequence, the games were slower to respond to users and it was harder to update or create new features – in fear of increasing the load on the servers further. Scaling this system was a hard process, as buying a bigger and newer machine was beyond possibility considering financial constraints. Players would use the delays to their advantage, cheating to achieve higher scores.

Difficulties in updating the operating systems (and consequently third party applications) also led to security vulnerabilities in the servers, as they become outdated. In addition, as the code base of the provided services increased, no one in the company could understand it completely, making changes to the existing code or databases a complicated affair for the developers, especially the new ones.

Lastly, although many games in Chilltime have similar structures, many were not being recycled in terms of code (users and logins, for example) culminating in distinctive implementations, which was not efficient for developers to work with.

Facing these challenges, Chilltime decided to update its infrastructure to a more modularised and flexible one, that could accommodate the services' needs and the company's business growth.

Before defining the objectives for this reformulation, it is also worth analysing the types of customers that Chilltime (and the gaming industry in general) has, in order to make sure the architecture can take care of their demands. Some interesting facts about gamers' behaviours are:

- When users experience delays in a website, they are less likely to return. It also impacts negatively on the user's opinion about the company and its security [61];
- The time users spend inside a game is strongly influenced by factors such as delay and jitter on the server's responses to submitted requests and actions [6].
- Some users cheat, taking advantage of existing limitations of the services to have an edge over other players and situations. Cheating can have several negative impacts for the company such as a decrease in sales and revenue and a loss of non-cheating players due to the unfair advantage that cheaters have [62].

Quality of Service (QoS) is, therefore, a significant concern to keep users engaged and to ensure their return in the future. Another conclusion is the importance of ensuring that the services work correctly when under load, protecting data inconsistencies during data transactions, in order to avoid weak points that might be manipulated for cheating purposes.

Based on this information, it is now possible to determine that the main objectives for the new architecture of Chilltime are to:

- Ensure high availability of the services;
- Ensure no unnecessary delays;
- Ensure that even in peak times the performance is smooth;
- Quickly recover in case of a server or service failure;
- Make it easier to maintain the servers and software;
- Make it easier to maintain and extend the code base;
- Have no significant upfront costs either in the new architecture or in the modernisation process.
- Have an architecture that is easy to extend by developers, without needing to hire a system administrator or other additional expertise.

3.2 Requirements

In Section 3.1, the problems faced by Chilltime were presented and, from there, the goals for the new architecture were derived. In this section, a set of requirements is being defined for the system, based on the previous discussion. As such, the requirements are:

R1 – The system should guarantee isolation between components

This requirement exists to contain the impact of an error. This implies that when an error occurs, it should not crash other adjacent services, diminishing the effect of the error in the users.

R2 – The system components must auto recover in case of failure

This requirement is to ensure high availability of the services. Services and servers are bound to fail, and this should be expected, not fixed upon occurrence. The system needs to detect the error and take the appropriate measures to correctly restart in a consistent state.

R3 – The system should not have internal polling requests

Polling request is the act of periodically asking a service if there are new tasks. This method introduces delays that may be prolonged up to the total time period between the pooling requests, and may congest the network and/or databases unnecessarily.

R4 – The components should be deployable off-site

In case a server stops working, the components should be able to be deployed in the cloud, not needing to wait for the arrival of a new server to recover from the problem.

R5 – The battle system should be able to auto-scale off-site

During peak times, the request load might surpass the threshold of what the servers on-premise are able to handle. In this case, match processing – a very processing-intensive services – should run in the cloud instead, in an hybrid system, to avoid the degradation of the experience provided to the users. Other than worse user experience, the increasing load has the probability to create errors and inconsistencies that can open doors for cheating in the game.

R6 – The battle system should scale up to 500 simultaneous requests without degrading its performance

Considering the rate of battles in Chilltime's games, in order to have five hundred simultaneous matches the company needs an enormous active user base. Therefore, if the battle engine mechanism is able to handle this number without the significant downgrades in the battles performance, the company is confident that the system is robust and reliable.

R7 – The system should host multiple games simultaneously

As many gaming companies, Chilltime offers several games and the system should be designed with that in mind. For example, this suggests that there may be services inside each game offering similar capabilities as in others, such as users, logins, leaderboards and send emails and other types of notifications. In Chilltime these services were being implemented separately in every project, making its maintenance a hindrance for developers. As such, the new architecture should be designed to identify and group all identical services in a single one to ensure easier maintenance.

R8 – The system must have monitoring and logging services

This requirement's objective is to ensure that problems are detected early, as well as to ensure an easier time debugging. Furthermore, monitoring may also aid in finding the bottlenecks of the system and helping developers direct their efforts to the most impactful place while ensuring better performance for users.

R9 – The system must increase testability and observability

One problem of monolithic infrastructures is its difficulty to test it in a modular way, and the new architecture should not only make sure that there can be unit testing but also that it can be done automatically. Moreover, there should be systems that can help retrieve information on what is happening inside in order to facilitate development and debugging.

R10 – The system must be able to send messages to players via the adequate channel

The system must incorporate a communication service that is able to send messages back to the player in the most adequate channel – for Chilltime it means email and app notifications.

R11 – The communication system must be able to incorporate new channels

The communication system should be extensible if the company wants to adhere to new communications channels. The processes of adding a channel should be simple and quick to setup.

4

Architecture

Contents

In this chapter, the developed architecture – MAGIC – is being presented. This architecture aims to create a company-wide structure that complies with the objectives and requirements described in Chapter 3 while having in consideration the business functionalities required by Chilltime. Since the majority of the components are believed to be essential to many gaming companies, any company in the industry can use this infrastructure as a starting point to define their own structure. In the case eventual adaptations need to be done, they should be simple to add on account of the modularity and loose coupling obtained by the use of a microservice approach, creating an easily extensible design.

To ensure a good understanding of the system and its advantages, the C4 model is being used to provide graphical support on explaining the service. As presented by Simon Brown [63], this model aims to present software systems in four different views, that offer distinct perspectives on the system, with a simple and flexible notation. Although this system was created to present software architecture, it is easily extensible to the explanation of a system architecture. The C4 model is comprised of four layers of abstraction [64]:

- context layer – It aims to frame the system in the context of its users and its interaction with external services. The focus is to provide a high-level view of the target system, presenting the interactions with the outside.
- container layer – It focuses on the main components/services offered by the system. The internal relationships and interactions with external components are also explained.
- component layer – It is represented by several diagrams that offer a more detailed view on each of the containers defined in the second layer and focuses on the components that ensure the provided functionality works correctly as well as their relationships with each other.
- code layer – It is where the implementation details are presented. Originally, as this model was designed for software architecture, the fourth layer was the Unified Modeling Language (UML) of each component. For this report, this layer presents the technology stack used, containers distribution, and communication patterns chosen. These diagrams are analysed in Chapter 5.

4.1 First Layer – Context

As mentioned, this layer defines the context of the application. The diagram for the first layer is presented in Figure 4.1.

Starting by the leftmost components – the users – there exists three main types:

- Anonymous - These users do not have an account in the company, although they are able to access some public web pages to perform certain tasks, for instance, to create an account or get more information about the company.

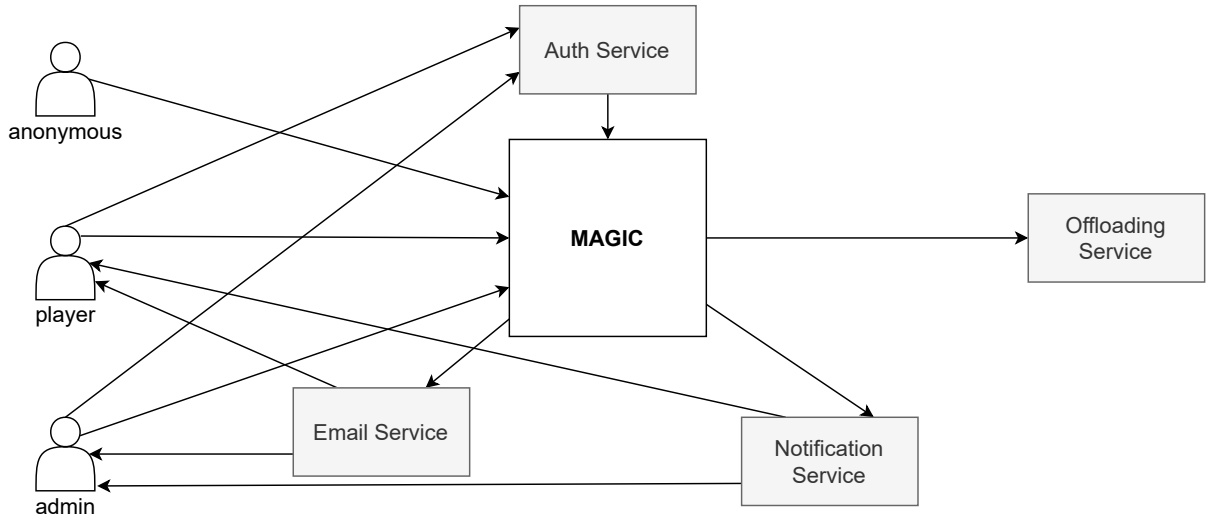

Figure 4.1: Context Abstraction Layer

- Players - These users have an account in the company and are authorised to access the games web pages and APIs.

- Administrator - These users work for the company and have a more unrestricted access to the APIs, the game's web pages and support web pages – to access monitoring and logging information or to access internal data or tools in order to, for example, solve customer issues.

Referring again to Figure 4.1, these users communicate with the central component – MAGIC. This component represents the companies' infrastructure, including the game engines, the web-pages, APIs and back-office support services.

The users (in specific the ones registered in the company) are also able to communicate with the Auth Service – the authentication and authorisation service. This service manages the user's authentications and access roles and, therefore, should ensure that the users are not impersonating someone else, accessing exclusively to what their roles permit. The authentication and authorisation service also needs to communicate with the companies' system to ensure the service provided.

It was also included in this diagram two external services for the system to communicate with the users: email service and app notification service. Both these services receive requests from MAGIC and will then interact with the users, provided they possess an account in the company.

Lastly, there is the offloading service. This service is supplied by cloud providers and refers to the API requests sent by MAGIC in order to dispatch the exceeding load to run on the cloud environment, in a hybrid cloud approach.

4.2 Second Layer – Containers

This layer will focus on MAGIC main components that interact with the outside services and provide the company's functionalities. The diagram for the second layer is presented in Figure 4.2.

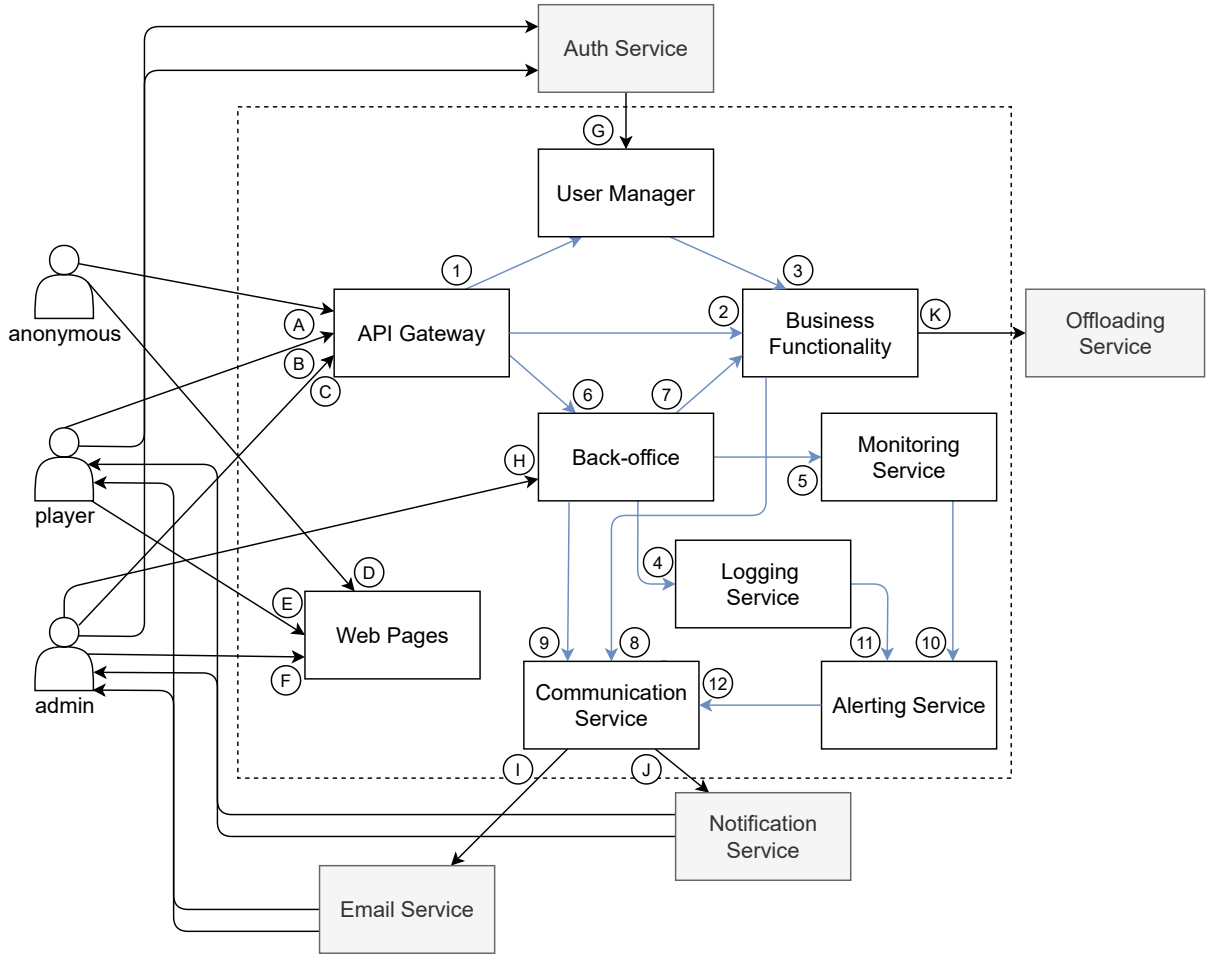

Figure 4.2: Container Abstraction Layer

The external components were those presented in the previous section. Regarding the internal components, they're as follows:

- API Gateway – Conjugated with the Auth Service, this service should provide the first barrier of any API request to the system. It should have a service discovery mechanism that it can use to route the requests to the correct service before sending the answer back to the client.
- Web Pages – This component contains all web services provided by the company which are accessible by the users.
- Communication Services – This component contains interfaces that can be accessed by the other

components in the company, which hide the implementation behind the connections to the external email/notification service and trigger the sending of these types of messages to the correct users.

- User Manager – This component aims to connect the information concerning the users (for example, email, address, ID) with the information on access tokens and identifier from the external Auth Service.
- Business Functionality – This component contains all services regarding the specific functionality of the company which, in this case, is related to gaming. All game-related data and functions are inside this component.
- Back-office – This component represents a set of web pages and respective backend functions with the authorisation to perform special queries on the data in the business functionality services. This way, the administrators can access monitoring and logging information and perform actions related to the administration and management of the products.
- Logging – This component includes the mechanisms to retrieve logs from the services in the company and storing them in a way that is easy to filter and analyse.
- Monitoring – This component includes the mechanism that verifies if the services in the company are online, healthy and functioning properly, by collecting metrics and performing health checks.
- Alerting – This component analyses the output from the logging and monitoring and triggers alarms when certain errors happen or thresholds are surpassed.

All these components relate to each other and to external services. In Figure 4.2, internal relationships are blue and marked with a number and external interactions are black with capital letters.

Starting with the API Gateway component, it receives requests from the users (A, B and C). These requests are then identified/validated in the User Manager (1) and then sent to the specific service that will provide the answer inside the Business Functionality (2). The users also connect with the Web Pages component (D, E and F), that provide the websites and the list of API requests to fetch the dynamic information.

Moving on to the User Manager service, it receives identification and validation requests by the API Gateway. It also receives information from the external Auth Service, defining the relationship between the access token provided in the external service and the user identifier inside the company infrastructure. It is worth mentioning that this user identification is made in relationship to the whole company, independently of the game they are connecting to. The relationship between the users and the games is only analysed inside the Business Functionality component, hence the connection between these two services (3).

The administrator user, benefiting from more unrestricted access to the system, can also access the Back-Office system (H) directly. This service provides a visualisation platform of the Logging and Monitoring system (4 and 5) but can also receive administrator's requests from the API Gateway (6) to retrieve restricted information or to trigger protected administrator actions by sending the appropriate messages to the Business Functionality component (7). If these restricted requests involve sending some message to the client, the service can solicit that request to the Communication Service (8).

It is also worth analysing the Business Functionality component. As it has been discussed, this component receives requests from the API Gateway (2), from the User Manager (3) and the Back-Office (7). These requests are directly related to the data and action from games that can be played in the company. Most of these requests will be executed inside the component, and the ones that require a response are then sent back from the path where they were received. But some need to be forwarded to the Communication Service (9), to ensure the user is notified of a certain event.

The Communication Service will then send the formatted messages to the specific communication channels chosen – in this architecture either email or notification (I and J, respectively). Other than the communication channel, the Business Functionality component might also need to request services from a cloud provider, if it decides that the existing local resources are not enough to ensure timely execution of the tasks. In this case, some requests are sent to the Offloading Service for processing (K).

Finally, there is the Alerting Service. This component analyses the data received in the Monitoring and Logging components (10 and 11, respectively) and, in case anything happens that is worth mentioning to the administrators (such as an unhealthy service), an email can be sent to the appropriate person, by having the Alerting Service send a request to the Communication Service (12).

On a last note, although it is not represented in the Figure 4.2 in order to produce a more understandable image, there is one last set of communication between components that is not represented. The Logging and Monitoring services are connected and receive information from every service represented in the diagram. The messages received contain information on the actions performed by the system as well as a way to understand if every service is online and healthy.

4.3 Third Layer – Components

The third layer focuses on the components that allow the containers presented in the second layer to perform as desired. This means that each component of Figure 4.2 has a respective diagram of layer three, in the C4 model. Since all components except the Business Functionality are considered to be implemented using a standard or one block approach, it is not worth creating the diagram or doing a more in-depth explanation of the services. Regarding the Business Functionality components, the layer three diagram of this container is presented in Figure 4.3.

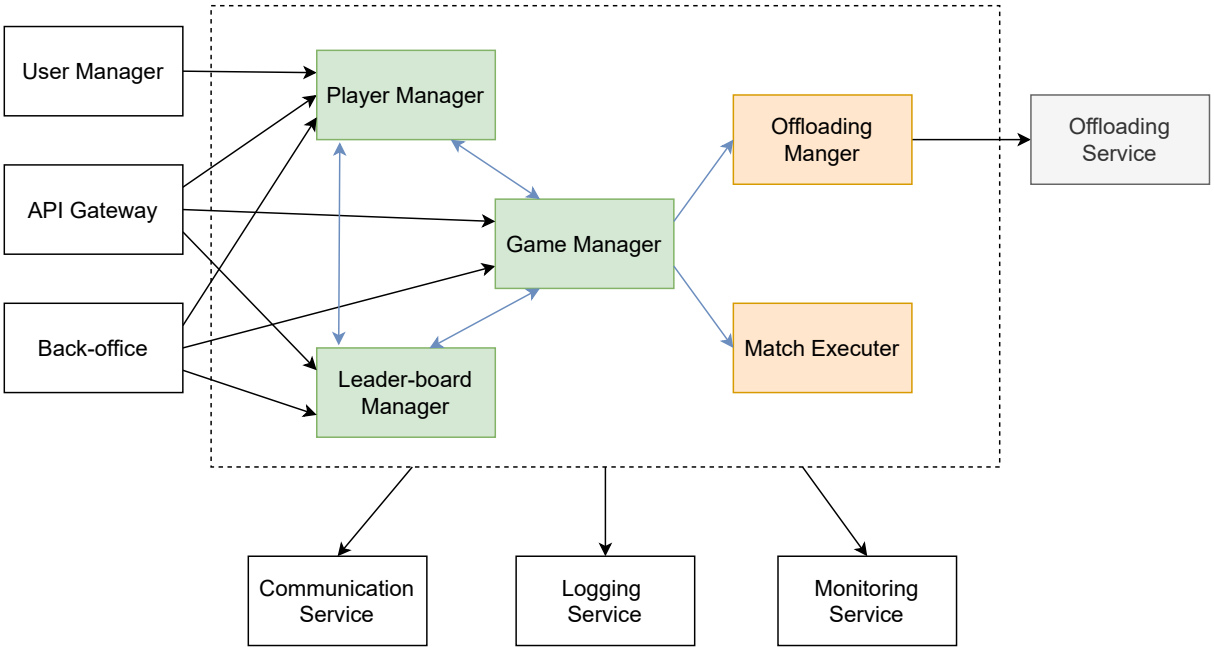

Figure 4.3: Components Abstraction Layer – Business Functionality

It is possible to divide the Business Functionality into two layers: the data layer (represented by the green components) and the engine layer (represented by the orange components).

One important aspect to notice in the figure, is that each green manager is responsible for an aspect of gaming for all games produced in the company. The concept of player (with nickname, avatar, scores) is present in any of these games, and therefore, by having these components centralised in a Player Manager, it is possible to centralise much of the services to be able perform all similar requests like changing the nickname or increasing the score after a game. The same can be said to the Leader-board Manager, whose implementation mechanisms are very similar in every game.

Moving on to the Game Manager, this service includes all information about past and ongoing matches, indexed by the specific game being played. This service also comprises information about maps and digital worlds in the game (if applicable). This component is the only one that can connect to the orange engine layer components.

Regarding the engine layer, these are responsible for processing the moves requested by players and returning the result of the turn or the match to the Game Manager. If a match ends, the scores are updated in the Player and Leader-board Managers. The existence of two types of workers in the engine layers lies with the possibility of task offloading in the cloud. This means that the Game Manager will consider the current state of the system by analysing metrics – such as the number of matches being processed, CPU usage percentage or the results of load prediction mechanisms – to determine if the requested move can be processed locally (in the Match Executor) or if it has to be directed to the Offloading Manager to be processed in the cloud.

Focusing on the relationships with external components, the User Manager may interact with the Player Manager, to understand what games does a user has access, for example. Both the API Gateway and the Back-office Services can connect to any of the three data layer managers to retrieve and update information or submit a move request by a user.

Regarding the Communication Services, it can accept requests from any component inside this diagram to, for example, send a notification to a player when he changes the rank on a leader-board. Logging and monitoring can also interact with any of the blocks, to collect the activity performed by the services and workers and to ensure they are performing as expected. These relationships were omitted from the diagram, to ensure it remained comprehensible.

4.4 Business Functionality API

To finalise the presentation of this infrastructure, the last thing left to explain is the main service API. By presenting the API, it is easier to understand what capabilities are incorporated in this architecture, as well as to understand the flow of the requests in the service. This API is exported by the API Gateway, having the requests then sent to the appropriate Business Functionality component.

It is worth noticing that, although the presentation of this API uses an HyperText Transfer Protocol (HTTP) REST formulation, it can be implemented in any format desired. Also, even though only the Business Functionality API is being presented, all services in the second layer (represented in Figure 4.2) have their own interfaces. However, since their functionality is standard, it is not being explored in this chapter. Moreover, request options like output pagination and filtering may be implemented on top of this API, but are not referred throughout this report.

Finally, the presented API is to be considered generic. It will not incorporate exactly all resources included in the responses or requests, since many of these values depend on the game itself. The score value is a good example, as it comprises the player's virtual possessions and punctuation displayed in the leader-boards. In World War Online, for example, the score is made of the user's rank, the number of iron and units in the player's possession. Regarding the Marble Adventures, the score is the number of stars archived in each puzzle level and the number of lives left to play the game.

The API is divided in two: the players API and the admin API, each related to the requests made by their respective users' type.

4.4.1 Player API

These users are characterised by having full access and control over information requests and action triggers on their own accounts, but limited access to other user's information. Moreover, all the requests

referred in this section should be developed with the appropriate formatting, filtering and pagination techniques in order to be suited for the web pages or mobile applications needs.

- `/player/me`
  - `GET` – retrieves information about the requesting user from Player Manager.
  - `PUT` – updates a value in the profile on Player Manager.
  - `DELETE` – removes the user's account, the request being forwarded to to the Player, Game and Leader-board Manager.
- `/player/:player_id`
  - `GET` – requests the Player Manager fir information on another user, with a more limited output than `GET /players/me`.
- `/player/me/invites`
  - texttttGET – Player Manager returns all friendship requests either sent or received.
- `/player/me/invites/:invite_id`
  - `GET` – obtains further information on the invite from the Player Manager.
  - `DELETE` – rejects or cancels an invite from the Player Manager.
  - `PUT` – a previously received request or update a sent one from the Player Manager.
- `/player/:player_id/invites`
  - `POST` – request sends an invite to a player, in the Player Manager.
- `/players/me/friendships/`
  - `GET` – etrieves a list with all the user's friendships, from the Player Manager.
- `/players/me/friendships/:friendship_id`
  - `GET` – fetches more information on a friendship, in the Player Manager.
  - `PUT` request modifies a parameter in the friendship, in the Player Manager.
  - `DELETE` – removes the relationship between the players, in the Player Manager.
- `/games/:game_id/matches`
  - `GET` – lists all invited matches from the Game Manager.

  - `POST` – creates a new match of the defined game in the Game Manager. May required connection to the Player Manager if the game supports inviting friends to play.

- `/games/:game_id/matches/:match_id`
  - `GET` – receives the current state of the match from the Game Manager.

- `/players/me/games/:game_id/matches`
  - `GET` – requests list of the user's matches, by composing information from Player Manager and Game Manager.

- `games/:game_id/matches/:match_id/players/me`
  - `DELETE` – removes a player from a game in the Game Manager.
  - `POST` – post a new move on the game in the Game Manager, being then sent to the Match Executor or Offloading Manager for processing. Afterwards, the new state of the game is updated in the Game Manager and, if the game finishes, the score is updated in both the Player and Leader-board Manager.

- `/leaderboards/game/:game_id`
  - `GET` – presents the leader-boards' list of the game referred, from the Leader-board Manager.

- `/leaderboards/:leaderboard_id`
  - `GET` – provides the list of player and respective positions on a certain leader-board, from the Leader-Board Manager.

- `/players/me/game/:game_id/leaderboards`
  - `GET` – retrieves the information about all positions of a player in all leader-boards of the selected game from the Leader-board Manager.

4.4.2 Administrator API

The admin user differs from the player user by not having access to playing the actual game. But it can meddle with other player's games and profiles, as well as retrieve any kind of information, with unrestricted access.

- `/players/games/:game_id`
  - `GET` – retrieves the list of all players in the game from the Player Manager.

  - `POST` – creates a new user in the Player Manager, collaborating with the external User Manager, Player Manager, Game Manager and Leader-board Manager.

- `/players/:player_id`
  - `GET` – retrieves the player's information from the Player Manager.
  - `PUT` – changes a player's profile, in the Player Manager.

- `/invites`
  - `GET` – sends the current invitations list of the users, from the Player Manager.

- `/invites/:invite_id`
  - `GET` – retrieves the information on the invite from the Player Manager.
  - texttttPUT – changes some information on the invitation, in the Player Manager.
  - `DELETE` – cancels the invite in the Player Manager.
  - texttttPOST – either accepts or rejects an invitation in the Player Manager.

- `/players/:player_id/invites`
  - `GET` – returns a list of a user's invites from the Player Manager.

- `/players/:player_id/friendships`
  - `GET` – returns the list of friendships of a specific player from the Player Manager service.

- `/friendships`
  - `GET` – returns all friendships, handled by the Player Manager.

- `/friendships/friendship_id`
  - `GET` – requests more information on a friendship from Player Manager.
  - `DELETE` – blocks a friendship in the Player Manager.
  - `PUT` – requests to change any friendship information from the Player Manager.

- `/games/:game_id/matches`
  - `GET` – gives a list of all the matches available in a game from the Game Manager.

- `/players/:player_id/matches/`
  - `GET` – retrieves information on the matches of a player from the Game Manager.

- `/players/:player_id/match_invites`
  - `GET` – from the Game Manager, retrieves all invitations for matches the player received.
- `/players/:player/games/:game_id/matches/:match_id`
  - `DELETE` – enables an admin to remove a player from a match in the Game Manager.
- `/games/:game_id/matches/:match_id`
  - `GET` – retrieves the current game state, from the Game Manager.
  - `DELETE` – cancels and closes the game, in the Game Manager.
- `/leaderboards/:leaderboard_id`
  - `GET` – retrieves the players' ranking in a leader-board from the Leader-board Manager.
  - `POST` – recalculates the leader-board positions on the Leader-board Manager.
- `/leaderboards/games/:game_id`
  - `GET` – outputs all leader-boards of a game from the Leader-board Manager.
- `/leaderboards/:leaderboard_id/players/:player_id`
  - `GET` – asks the Leader-board Manager for the position of a player in a leader-board.
- `/leaderboard/games/:game_id/players/:player_id`
  - `GET` – retrieves all positions of a player in all leader-boards of a game, from the Leader-board Manager.

5

Implementation

Contents

In this chapter, the implementation of the proposed architecture is being explained.

It is a complex process to change the infrastructure of an existing company. Several tasks are involved in successfully breaking down the monolith, with some authors referring that it is a never ending evolution, which keeps changing in order to adapt to the company's current needs [24]. The reason for its complexity revolves around having employees to learn new technologies and software and time to refactor code. Current internal processes – from testing to monitoring – are reformulated, in a company-wide effort that requires careful coordination between teams.

Chris Richardson presents a pattern for an incremental conversion called strangler application [22], which consists in having a new microservice application developed around the legacy monolithic one by progressively absorb features from it, until it disappears.

The keys to a successful adaptation are planning and opportunity: choosing the right time to separate services from the monolithic application. Usually, it can be done when these services are raising problems or need functional updates. The establishment of new features also generates great opportunities to continue evolving the system.

Regarding Chilltime's implementation process, it was no different. There has been considerable progress, always very depended on the company's circumstances. It is worth noticing that during the period of this research, not all elements have been migrated to the new infrastructure, due to the constraints referred to previously. So, in this section, it will be discussed the process of implementation of the services done to date, as well as an implementation plan for the services further on.

5.1 Technology Stack

In this section, the language and software packages chosen for the implementation of MAGIC on Chilltime are being presented.

Docker

Docker was introduced by 2013 and quickly gained track on the open source community [65]. It offered the first mature implementation of container management, claiming the benefits referred in Chapter 2 and more: significant productivity gains in the DevOps area due to the IaC approach, and an easy management and adaptability of the deployment environment, which was not possible with VMs.

Docker works by having a daemon on the server that manages not only the container instances in the host, but also the networks and volumes, even doing monitoring – by doing periodic health check requests to the various instances. On top of the Docker, Docker Compose can be used in order to configure services [66], by grouping the configurations on the various containers that constitute that service, and being able to control these groups of containers as a whole. Docker Compose also facilitate

the scaling process of certain workers, as the number of instances can be easily changed with the `docker-compose` command. Finally, one last advantage is the restart policy options, that determine what happens when there is a problem in a container – it can be set to `no`, `always`, `on-failure` and `unleass-stopped` – helping with fault-tolerance.

Kubernetes

When a system has Docker installed on more that one server, an orchestrator can be used to configure and determine the desired state of the system and implement more robust fault tolerance mechanisms, by managing collectively the various hosts with Docker.

One of the most popular orchestrators for containerised applications is Kubernetes [45]. It supplies software that successfully builds and deploys distributed systems. It also ensures high reliability, even when part of the system crashes. It guarantees high availability, even during software rollouts and maintenance. And, finally, it ensures scalability, by efficiently using the existing resources – all this with straightforward configurations. This is possible because Kubernetes offer several key features. The first is the immutable infrastructure from containers, meaning that the structure does not evolve gradually but rather is defined in a configuration file. The second is about declarative configuration, which extends the immutability concept to the Kubernetes configuration. Rather than configuring the steps to create the architecture, Kubernetes requests only a declarative description of the desired state of the system. Kubernetes will employ all mechanisms to ensure that the state is kept as requested. The final feature is about self-healing systems because, since Kubernetes understand the desired outcome, it can continuously adapt the system – regarding the runtime variables, errors that occur, or even when a container needs to be replaced for a newer version without downtime.

RabbitMQ

As for message brokers, the chosen was RabbitMQ – an open-source, lightweight yet extremely powerful and versatile message broker [67]. RabbitMQ is platform and vendor neutral, has client libraries in most languages, and holds some layers of security. Therefore, RabbitMQ has become a prevalent choice for companies that require this service. RabbitMQ is based on Advanced Message Queuing Protocol (AMQP), which defines three abstract components that create the message routing behaviour:

- exchanges – routes messages to queues;
- queues – data structure that stores messages;
- bindings – routing rules for exchanges.

As it will be seen in this chapter, the configuration of these parameters can create several types of communication paradigms between services, making RabbitMQ a very flexible tool, ideal for the Chilltime infrastructure.

Traefik

Traefik is an open-source edge router that receives requests on behalf of the system, distributing (and load balancing) them over the correct components [68]. This edge router is capable of two types of routing: Layer 4 (Transmission Control Protocol (TCP) – based on the Internet Protocol (IP) address and ports) or Layer 7 (HTTP – based on the hostname and path). It integrates natively with several technologies like Docker and Kubernetes, enabling service discovery options and also providing an easy setup and configuration.

Elastic Stack

There are several open-source tools designed for observability tasks, but Elastic Stack (formerly known as ELK Stack) stands out. As defined by Pravah Shukla and Sharath Kumar [69], "Elasticsearch is a realtime, distributed search and analytics engine that is horizontally scalable and capable of solving a wide variety of use cases. At the heart of Elastic Stack, it centrally stores your data so you can discover the expected and uncover the unexpected.".

To deploy an Elastic Stack, there are three main components. Logstash centralises the collection and transformation of data, supporting many different types of inputs, including Docker and RabbitMQ. Logstash outputs the received data to the next component: Elasticsearch. This component stores all data collected in Logstash, providing search and analytic capabilities in a scalable way. Finally, Kibana corresponds to the visualisation tool of Elasticsearch. There are several types of graphs that may be built and interacted with in the interest of aiding the visualisation of patterns and relationships between data, being provided to the user in a website. The biggest advantages of Elastic Stack are being schemeless, document-oriented, easy to operate and scale as well as being resilient. There are also several client libraries which are essential to easily add new sources of logging and monitoring.

Node.js

Based on JavaScript, Node.js is an event-driven language that can produce highly scalable servers using an event loop software architecture [70]. This architecture also reduces the complexity of writing code for concurrent programming, while still offering an excellent performance. To top the event-driven approach, Node.js also provides several non-blocking clients and libraries, making it ideal for connecting to external services as well.

Jenkins

Jenkins is a tool that can construct deployment pipelines that is able to build, test and deploy the company's software [71]. It can easily be integrated with Kubernetes to automatically deploy the new versions on the system, being an extremely interesting option to automate the deployment cycle for the company.

5.2 Implementation to Date

In the next section, the implementation steps for MAGIC in Chilltime are explored in more detail. As referred, not all the products and services have been migrated to the new infrastructure. In Figure 5.1 it can be observed the architectural diagrams from the second layer of the C4model, in which the blue services have been completely implemented while the yellow components have been only partially done.

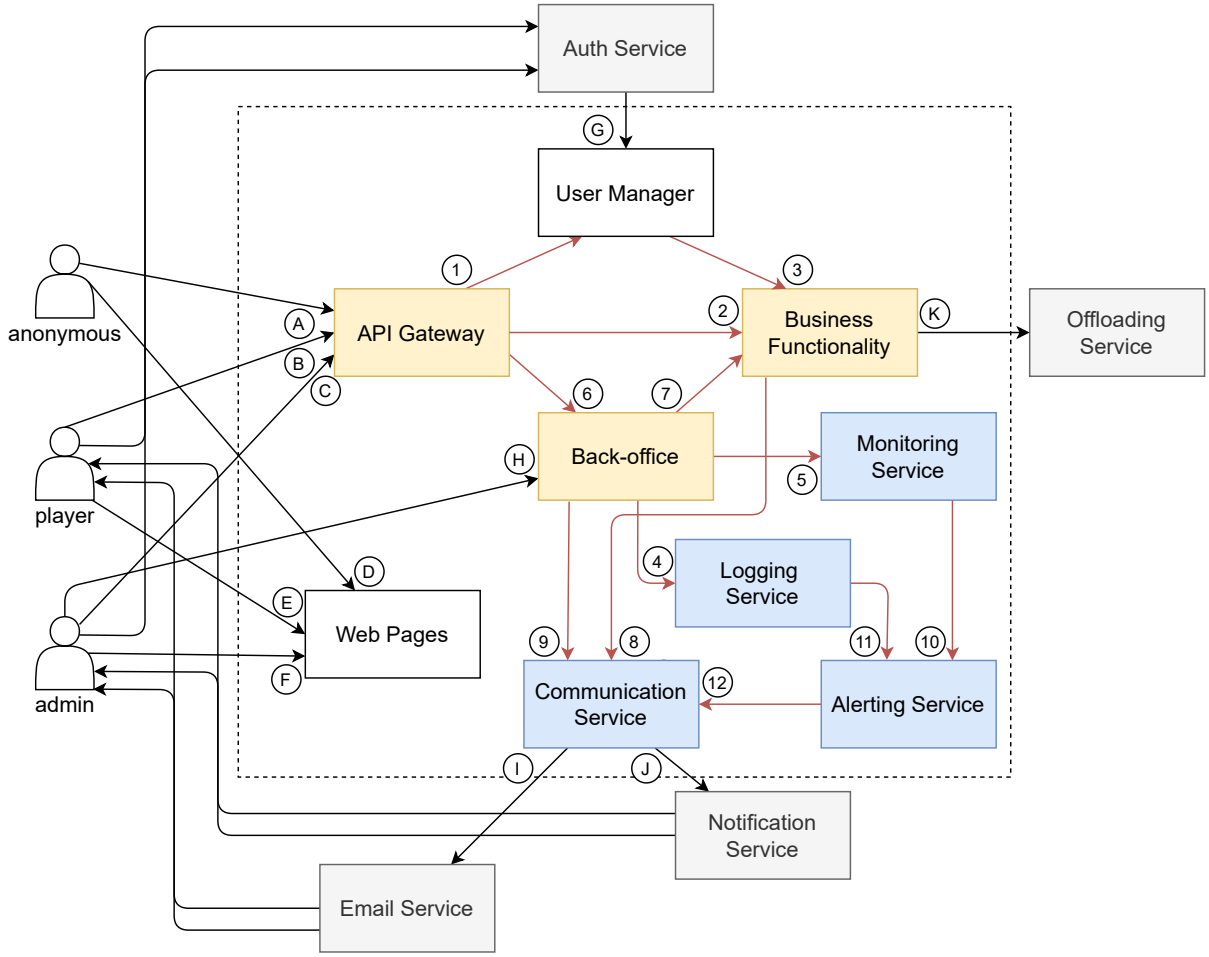

Figure 5.1: Implementation State of the Container Layer (Second Layer)

Regarding the Business Functionality, the implementation state of the third layer of components can be seen in Figure 5.2 where, again, the blue components have been totally implemented while the yellow ones have only been partially covered. Moreover, as it will be explored in more detailed later, the services implemented in this diagram, have only been for the World War online game, as it was the one having updates done – as the battle engine migration to Unity referred in Chapter 3 – which created the perfect opportunity for this feature's migration.

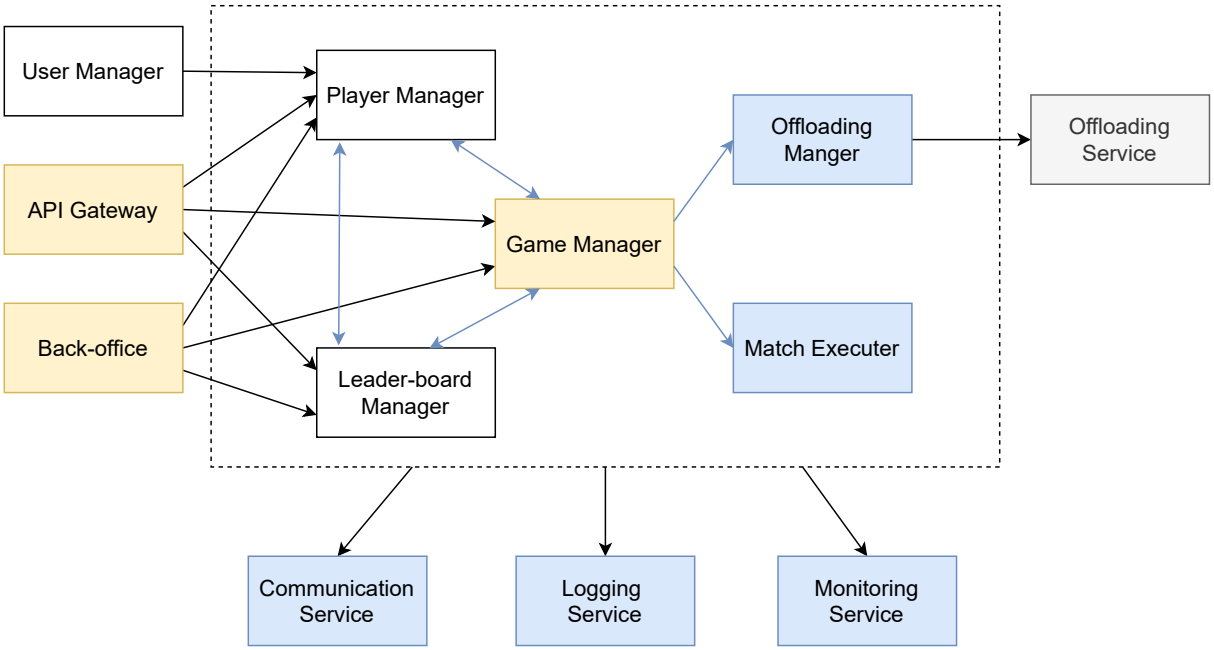

Figure 5.2: Implementation State of the Business Functionality Components Layer (Third Layer)

5.2.1 Environment Setup

The starting point of the Chilltime's implementation of this architecture, was doing a software update on one of the company's servers. The update caused the server to be more secure, therefore being able to install the most recent versions of any software desired. This server was cleared, and new services that run inside were all installed on top of Docker containers. In an initial approach, it was not used any container orchestrator, due to the system being only in this single physical server.

After configuring the server and installing Docker, there is one last step before starting the implementation of the services: having a message broker for asynchronous communication. The reason behind having this type of communication is that, as it has been explored in Chapter 2, asynchronous messaging is generally a better option to ensure less coupling between services while making one-to-many communication possible. Thus, in cases where there are clear benefits from using messaging, this should be the chosen option. As explored in Section 5.1, the software adopted was RabbitMQ.

5.2.2 API Gateway

The first implementation step was the creation of the API Gateway, as it represents the company's interface with external users and the entry point of the system. This service can be observed in Figure 5.1. To create this service, the Traefik project was installed. API Gateways, however, can provide more features. Service discovery, routing and load balancing were considered the most necessary ones for the company, hence its expeditious implementation.

5.2.3 Communication Services

Following the API Gateway, the next step was the creation of the communication services. This service was adopted when Chilltime changed their email sender provider to one more suitable for their needs: Mailjet [72]. The necessity to create this new feature provided the perfect opportunity to focus on developing the first set of functional microservices. Due to their similarity, the app notification service immediately followed using Firebase Cloud Messaging as its external service provider [73].

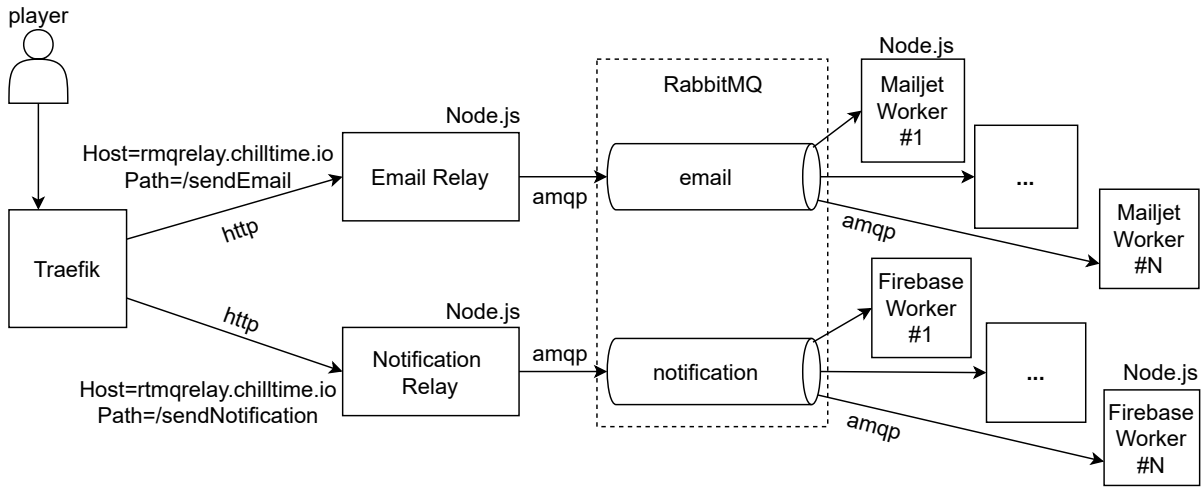

Figure 5.3: Implementation of the Communication Services

When the service sends an HTTP request to the correct hostname, Traefik will direct the message to the correct endpoint based on its path, forwarding it to the correct communication service.

The first containers reached are the relay services that simply translate HTTP requests to AMQP messages, changing the communication from synchronous to asynchronous. There are two relay services, one per communication channel, written using Node.js.

These relay services also provide an extra layer of security using Hash-based Message Authentication Code (HMAC), an encryption mechanism for message authentication using cryptographic hash functions. It provides a way to verify the integrity of the information transmitted based on a secret key shared between server and client. This mechanism proves to be appropriate since the only ones authorised to use this communication services are other internal servers. It may be argued that this intermediate relay is not necessary, as servers can put requests directly on RabbitMQ, however some services in the company do not offer the possibility of installing RabbitMQ client libraries – yet – due to language or software being outdated.

The message broker is configured to have two exchanges – email and notifications – directly bound to queues with the same name. When a message is received by the queue, it is later consumed by one of the active workers – also written in Node.js with the email/notification client libraries installed. So

far, only one worker has been used per communication channel, as it is sufficient for the load. But the system is prepared to have more instances of these workers, consuming from the same queues, if the service needs scaling.

5.2.4 Logging, Monitoring, Alerting and Back-office

As discussed in Chapter 2, observability is an essential trait in microservices. Therefore, when designing other services for the company, the Elastic Stack was implemented to ensure the observability components of MAGIC – the Back-Office, Monitoring, Logging and Alerting Services from Figure 5.1. With this, it was possible to better understand how the containers were performing and to more easily debug issues. In Figure 5.4, the implementation is presented.

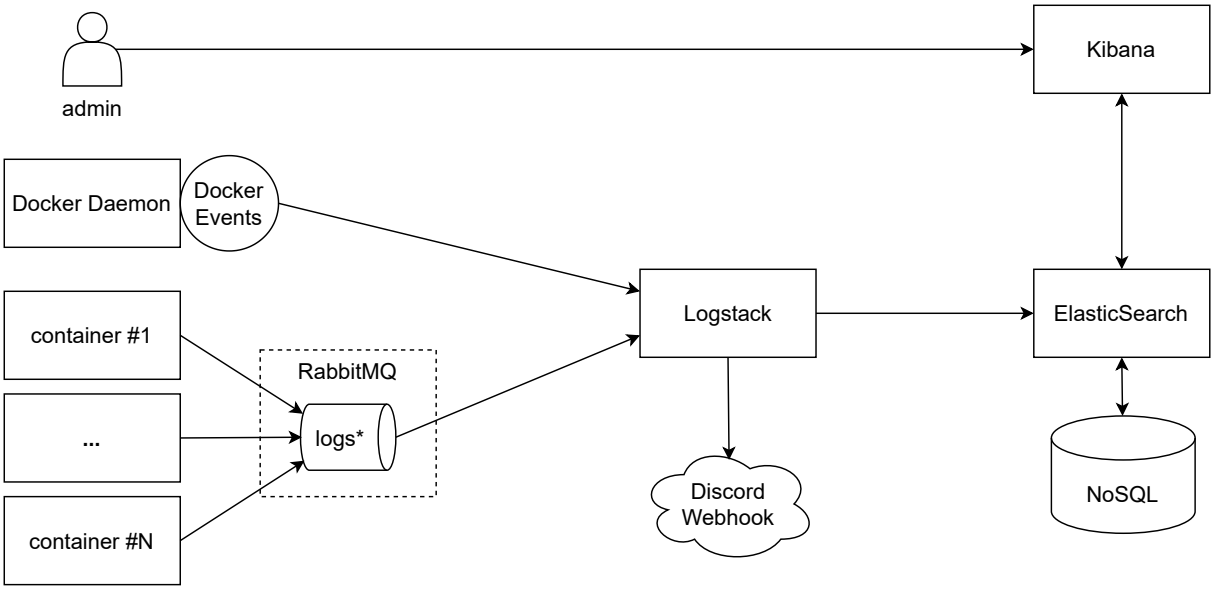

Figure 5.4: Back-office, Logging, Monitoring and Alerting – Implementation

The most important logs to collect are about the requests and processing information on each container, which can be very useful for debugging and to derive patterns of utilisation. To do this, the concept of event sourcing was applied – the microservices publish events to the network, and the interested parties are able to subscribe to access the information. In this case, by using RabbitMQ queues for services to publish logs, it was possible to have the Logstash to subscribe to those queues and retrieve that information. These logs are then slightly modified (to aid with filtering) and sent to Elasticseach. From there on, admin users have access to Kibana to visualise, filter and analyse them.

These logs have also been modified to have distributed tracking of requests, to further help debugging. Log messages were given a request ID that makes it feasible to track requests over containers.

As discussed in Chapter 2, another interesting metric to observe is the health of the services, and

this is possible to achieve through health check API calls to the microservices. Docker deamon does the monitoring of these requests according to the configuration setup by the images, emitting an event in case a service becomes unhealthy. By requesting the docker events from Docker daemon and then directing the output to the Logstash, it becomes possible to retrieve the aforementioned metrics. This way, it is possible to monitor the containers' behaviour – such as starting, stopping and heath state.

Lastly, there are the alerting capabilities of the system. These were also implemented using the Elastic Stack, by having the Logstash send a message to a Discord webhook – the internal messaging system in the team. These alerts are only sent in case the events received from docker event warning about a stopped or an unhealthy container.

5.2.5 Game Manager, Match Executor, Offloading Manager – World War Online

In Figure 5.2, it is possible to analyse the inside of the Business Functionality. Some of these services were implemented for World War Online – the most played game at Chilltime – since it was being modified to start running on Unity instead of PHP, as previously explained in Chapter 3.

The match processing unit is usually a resource intensive task. The former version of the battle mechanism had performance issues in peak times, with players using the delays to gain an advantage over other users. Therefore, one of the main goals with the new architecture was to ensure more potential scaling for this service. This can be achieved by having a hybrid cloud system, as it can be seen on Figure 5.5, where the system for the battles is presented.

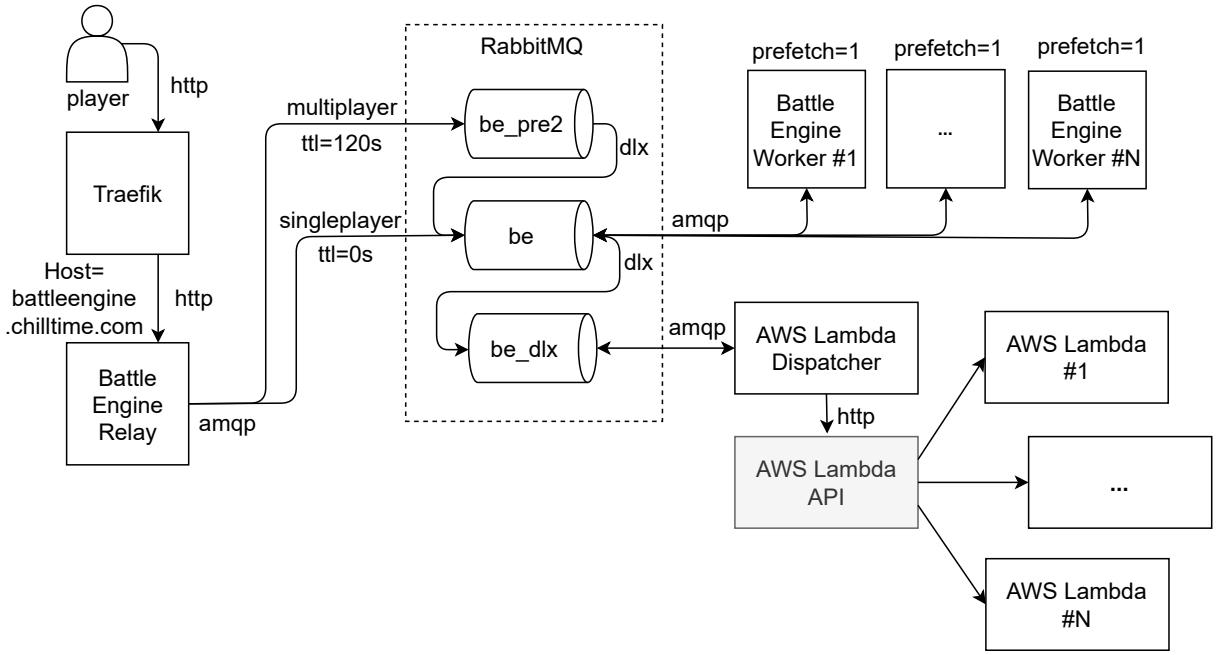

Figure 5.5: World War Online Battle Engine – Implementation

As it has been explained in Chapter 3, this battle engine is prepared for two types of matches: singleplayer and multiplayer. The singleplayer is processed directly after the user triggers the battle, while the multiplayer waits for two minutes before processing, so the attacked player can react and change its defence.

The first step of implementation involved choosing the correct cloud solution, which amounted to either be CaaS or serverless. To make the system easier to manage, the focus was on finding a simple serverless solution, in which it was possible to run the Unity application while being easy to upgrade as changes were made. After comparing the various systems available, the services used were the Lambda and S3 (for storing the Unity binaries) [74].

The next step was the configuration of RabbitMQ. In this case, the approach was modified by adding Dead Letter Exchange (DLX) and messages Time-to-Live (TTL). The DLX is an attribute of a queue that ensures rejected messages are routed to another exchange. One of the reasons for a message to be rejected is by surpassing its TTL, meaning that it has expired. This causes the redirection of a message that has waited too long in a queue to another queue, where another service can consume it. By using message Acknowledgess (ACKs) and adjusting the prefect consumer value – the number of simultaneous requests that a consumer can handle –, it is possible for the message to be redirected to the DLX instantaneously – using consumers with prefetch value as one and the message TTL to zero.

As it can be seen in this image, when the player triggers a battle, it is then directed to Traefik which routes the request to the relay. Focusing on the singleplayer, messages are sent with TTL as zero, to the `be` queue. If they can be consumed immediately by one of the registered consumers (which have the prefetch parameter set as one), they run locally. Otherwise, they are sent to the DLX, which will place the message in the `be_dlx` queue. This queue has a consumer service that connects to the AWS's API to launch a Lambda function. The battle engine is then processed in the cloud.

In the case of multiplayer, there is one extra step. The relay service will place the message in the `be_pre2` queue with the TTL of two minutes. As the queue has no consumers, the messages expire two minutes later, being consequently directed to the DLX queue – the `be` queue for singleplayer battles.

Regarding the containers themselves, the relay service is very similar to the other relay explained before: using Node.js with HMAC protection on the exposed API. The local workers are also implemented with Node.js, that run the binary file. The AWS Lambda dispatcher is also written in Node.js, with the AWS package for easy access to the AWS functionality. Within the Lamba functions, there is a simple python code that runs the binary file, which is added as a layer to the function. To upload a new version of the Battle Engine binary file, a `.zip` folder is uploaded to an S3 bucket. Then, the layer can be updated to a new version by uploading the file from the S3.

5.3 Future Implementations Plan

The several microservices implemented are part of a work in progress that will include all other functionalities, products and services provided by the company. A great part of the migration to the new architecture involves refactoring the existing functions of the monolithic application into smaller microservices, and that process alone may take years. In this section, the next steps of the implementation for Chilltime are explained in more detail, to ensure a better understanding of the final picture and how this architecture can really impact the company in the following years.

One of the first steps is to update the other three servers in order to ensure their security and Docker support. After, an orchestrator like Kubernetes can be used to manage containers across different hosts.

Regarding the Business Functionality from 5.2, the decomposition of the monolith is the next step. First, the API endpoints need to be adapted to the specific game terminology – for example, in World War Online, friendships are comparable to the squad concept in the game. In a first stage, the database can be shared among services. This should be temporary (or only done in a very small set of microservices) as it highly increases the coupling between service. Distribution of databases can be a daunting problem, that needs careful planning in order to created a flexible and efficient distributed database.

Data transactions are crucial for this infrastructure. An example is when a user is deleted – all Business Functionality services need to be updated to remove the player from their databases. As explored in Chapter 2, sagas are an option, existing as choreographed and orchestrated. Choreographed sagas are the most appropriate approach regarding the size of Chilltime. Moreover, the orchestrator service would need to be designed and maintained by the company.

As for making queries, the API composition approach should be used when it is sufficient, as long as having to deal with runtime coupling is not a problem for the users or developers. Moreover, as Traefik ensures the integration with Kubernetes and has a service discovery system, it can forward the requests in the infrastructure, regardless of the efficiency and fault tolerance mechanisms implemented.

In some cases, event sourcing and CQRS are worth considering. However, as it requires a more complex implementation, it should only be applied if there is a clear need or the implementation is simplified due to the nature of the task, like log collection. A possible implementation of a system using CQRS is the Leader-board Service, that subscribes to the updates on the score parameters of the games and calculate a new rolling snapshot of the current state, when necessary.

Finally, regarding DevOps, developing building pipelines can also be beneficial to accelerate the deployment process and facilitate the employees work: using the already existing Jenkins server in Chilltime office.

6

Evaluation

Contents

In this chapter, the analysis of the results from this architecture and implementation are derived. This chapter is written based on the objectives and requirements defined in Chapter 3.

6.1 Chilltime's Objectives Evaluation

In Chapter 3, Chilltime's objectives with this architecture were stated and explained. In this section, MAGIC's improvements towards these goals are analysed.

Microservices play an essential role to ensure the company moves in the right direction regarding the pre-established objectives. In respect of obtaining high availability, their modularity allows the organisation to modify each service without worrying about impacting others, which results in easier updates with minimal downtime. Docker and Docker Compose simplify the process of starting and stopping these containers, and Kubernetes promises to automatically update any component without downtime. The isolation between components guaranteed by microservices and delivered through Docker also ensures that the adjacent services are not affected in case of failure.

Regarding delay minimisation, microservices might seem counter-intuitive due to the added communication overhead, but it all boils down to a correct implementation plan. However, the migration to microservices should only be as complex as necessary to ensure the services can surpass the monolithic design's limitations, without introducing new constraints to the system. It is essential to keep this balance in mind for the next steps of implementation in Chilltime.

Another objective was related to having a smooth performance in peak times. This was achieved by integrating cloud services in Chilltime's workflow and implementing a hybrid approach on the most resource-intensive tasks.

Fault recovery was another major concern for the company. In this implementation, Docker Compose offers reset options that help containers recover from errors. However, it is Kubernetes that ensures the most impactful mechanisms to deal with faults in the servers.

Additionally, Chilltime wanted to simplify software and server maintenance, which is enhanced by microservices as they ensure isolation between components due to their loose coupling. This modularisation helps to easily modify components in the system when updating them to a more recent version or replacing them with more adequate software. This characteristic also eases code maintenance and extension, since similar features are centralised (as opposed to being scattered around the monolithic application) and different features are isolated from each other, which makes them easier to change without worrying about impacting the others.

As for cost management, it depends on the software chosen. In this implementation, all tools and frameworks used are open-source or free. The only exception is the cloud services, but those are paid per usage. The alternative would be buying more physical servers, which also implies high upfront costs.

Finally, the company does not have a system administrator and, therefore, the new tools and processes should be maintainable by developers. Docker and Docker Compose help ensure this, and, since the chosen cloud services are serverless, no administration is required for the underlying structure.

6.2 Requirements Evaluation

In chapter 3, the company's requirements were determined based on the objectives set. In this section, an evaluation of how MAGIC and its implementation in the company have impacted each requirement is done, ensuring they are met. In relevant cases, simulations were done for demonstration purposes.

The system should guarantee isolation between components

The first requirement is about isolation. This means that when a component has a problem, it should not affect other adjacent components. In terms of architecture, the microservices approach can be an enormous advantage due to the system's loose coupling. By diminishing the point of contact between components, and have them well defined, it is much easier to code around the interaction with interfaces in order to prevent errors from having a bigger impact.

By using containers and Docker in the implementation, the concept of isolation goes even further – by providing a virtualised environment for each service –, thus, an error in a component is even prevented from affecting other environments deployed in the same machine.

As an example, if the email relay service becomes unavailable, the company will not be able to send emails until the service becomes healthy again. In this case, if a service requests for an email, it will receive an answer saying that it is not possible to perform that request. The service requesting will then be able to take the most fitting approach to this error based on the context, without becoming blocked.

The system components must auto recover in case of failure

To have auto recovery abilities, the system should be able to detect it is down and recover. To this end, the architecture contains a metrics service that receives notification of possible problems in the infrastructure. With this information, the system can effortlessly implement the necessary mechanisms in order to rectify the problem, including alerting the appropriate person using the Alerting service.

As explained in Chapter 5, in Chilltime's implementation it was used Docker. With the Docker Compose feature, it is possible to specify the behaviour of the system in case the container stops by, for example, configuring it to automatically restart the container every time it is down. Moreover, there is also a health check API request to verify if the component is working well, and the metrics system also receives a message if the component becomes unhealthy. The current action of a stopped/unhealthy container is to both restart unless stopped and to notify the developer team through the internal messaging system each time it happens. Once Kubernetes is installed, more auto-recovery options become

available. As an example, services can be rolled back to a previous stable version and in case the hardware itself becomes unavailable, it is possible for the services to be deployed in another host.

As a case in point, the firebase worker was used to test this requirement. In this test, the Node.js code was slightly changed to run `process.exit(1);` upon receiving a message from RabbitMQ. This caused the application and docker to crash, as it can be seen in Figure 6.1. As the service stopped working, a message was sent to the appropriate team via Discord, as it can be seen in Figure 6.2. Then, due to the restart option on the docker-compose file that was set to `unless-stopped`, the container was immediately restarted as it can be seen from Figure 6.3.

```
Starting firebase-worker-dev_backend_1 ... done
Attaching to firebase-worker-dev_backend_1
backend_1  |
backend_1  | > firebase-worker@1.0.0 start /home/node/app
backend_1  | > node src/server.js
backend_1  |
backend_1  | npm ERR! code ELIFECYCLE
backend_1  | npm ERR! errno 1
backend_1  | npm ERR! firebase-worker@1.0.0 start: `node src/server.js`
backend_1  | npm ERR! Exit status 1
```

Figure 6.1: Application exiting with an error

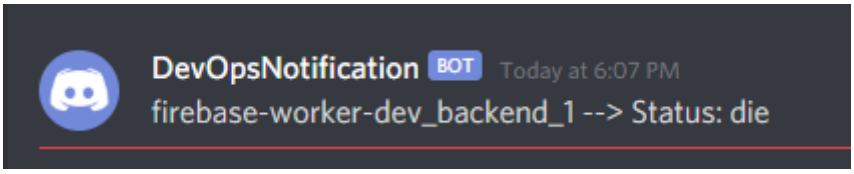

Figure 6.2: Alerting message on internal channel

Time	Type	status	name
Dec 16, 2020 @ 18:07:47.123	container	health_status: healthy	firebase-worker-dev_backend_1
Dec 16, 2020 @ 18:07:31.720	container	start	firebase-worker-dev_backend_1
Dec 16, 2020 @ 18:07:30.527	network	-	svcnet-prod
Dec 16, 2020 @ 18:07:29.371	network	-	svcnet-prod
Dec 16, 2020 @ 18:07:28.958	container	die	firebase-worker-dev_backend_1
Dec 16, 2020 @ 18:07:21.686	container	health_status: healthy	firebase-worker-dev_backend_1
Dec 16, 2020 @ 18:07:06.269	container	start	firebase-worker-dev_backend_1

Figure 6.3: Docker event logs

The system should not have internal polling requests

For every task that is scheduled, there are polling systems based on cronjobs, that periodically check the database for tasks to process. Polling creates unnecessary load on databases and one of the requirements was to remove this load. The usage of asynchronous messages that is typical from microservices, enables the creation of alternatives to deal with polling.

In the implementation made, there is one example of a polling system that was removed – the World War Online battle engine. As the processing of multiplayer requests is only done after a couple of minutes, the usage of RabbitMQ messages TTL and DLX has enabled the configuration of a way to start the processing on time, without having to query the database every 15 seconds, as it was being done before. Additional systems that use scheduling and polling can use the same approach to enhance performance.

The components should be deployable off-site

A microservice can be complex but its scope of action should be very well defined. Due to their smaller size and very defined interface, the migration process to the cloud is much easier to accomplish.

Regarding the implementation on Chilltime, the usage of container and Docker is a major advantage in the process of migrating a service to the cloud, due to the CaaS and orchestration services supplied by cloud providers. This means that, to migrate a service, the images and volumes need to be updated in the cloud and then, with little configuration, they run without having to change anything inside these containers.

The battle system should be able to auto-scaling off-site

The match processing units are by nature one of the most resource-intensive tasks. This requirement ensures that the functionality of the game is not affected by peaks in the load. In order to achieve this, the architecture has an offloading manager in Figure 4.3 that is able to send requests to the cloud.

Chilltime implemented a hybrid cloud system with AWS. Using RabbitMQ, it was possible to distribute the requests between the various local workers before sending them to the cloud if none were available. The number of workers is fairly easily adjusted, ensuring that the system can have an efficient use of the local resources before sending requests to the cloud.

By way of example, it was developed a script that sent eight battle requests with the same board to process. As the local implementation had five active workers for this game, three requests were sent to the cloud. In Figure 6.4 it is possible to observe the logs taken from our logging system, showing that eight requests were received in the relay, then five were received in the workers and three in the dispatcher (offloading manager). In Figure 6.5 it is displayed the monitor information from AWS Lambda. As it can be seen, the first graph describes the number of invocations counting three requests, the

seconds has the average duration of requests and the last has the error count in the requests, showing 100% of success rate.

Time	uuid	service	status_code
Dec 17, 2020 @ 16:15:13.223	94a2aecc-cf67-43aa-af2d-347f10313893	be-lambda	200
Dec 17, 2020 @ 16:15:12.661	61bc3374-ddd2-4a04-9400-660a06d72935	be-lambda	200
Dec 17, 2020 @ 16:15:11.233	00c648d8-9b7f-407e-9ff1-d84b4e8bb143	be-lambda	200
Dec 17, 2020 @ 16:15:09.227	76dc8195-f581-409a-85f3-aa62a5943acf	be-worker	200
Dec 17, 2020 @ 16:15:07.577	4f1cb3c7-82b0-4c03-99aa-83902ed3040f	be-worker	200
Dec 17, 2020 @ 16:15:07.372	cbd1768b-616a-45a9-bffa-ef4193c6509f	be-worker	200
Dec 17, 2020 @ 16:15:06.747	134b541c-8475-4141-8970-ae9b53ca671f	be-worker	200
Dec 17, 2020 @ 16:15:06.460	fc18f655-65e0-4ca2-a4f9-80347232ff83	be-worker	200
Dec 17, 2020 @ 16:14:46.498	94a2aecc-cf67-43aa-af2d-347f10313893	be-relay	200
Dec 17, 2020 @ 16:14:46.338	00c648d8-9b7f-407e-9ff1-d84b4e8bb143	be-relay	200
Dec 17, 2020 @ 16:14:46.161	61bc3374-ddd2-4a04-9400-660a06d72935	be-relay	200
Dec 17, 2020 @ 16:14:46.013	fc18f655-65e0-4ca2-a4f9-80347232ff83	be-relay	200
Dec 17, 2020 @ 16:14:45.765	4f1cb3c7-82b0-4c03-99aa-83902ed3040f	be-relay	200
Dec 17, 2020 @ 16:14:45.606	76dc8195-f581-409a-85f3-aa62a5943acf	be-relay	200
Dec 17, 2020 @ 16:14:45.445	134b541c-8475-4141-8970-ae9b53ca671f	be-relay	200
Dec 17, 2020 @ 16:14:45.298	cbd1768b-616a-45a9-bffa-ef4193c6509f	be-relay	200

Figure 6.4: Logs from test made with eight simultaneous requests

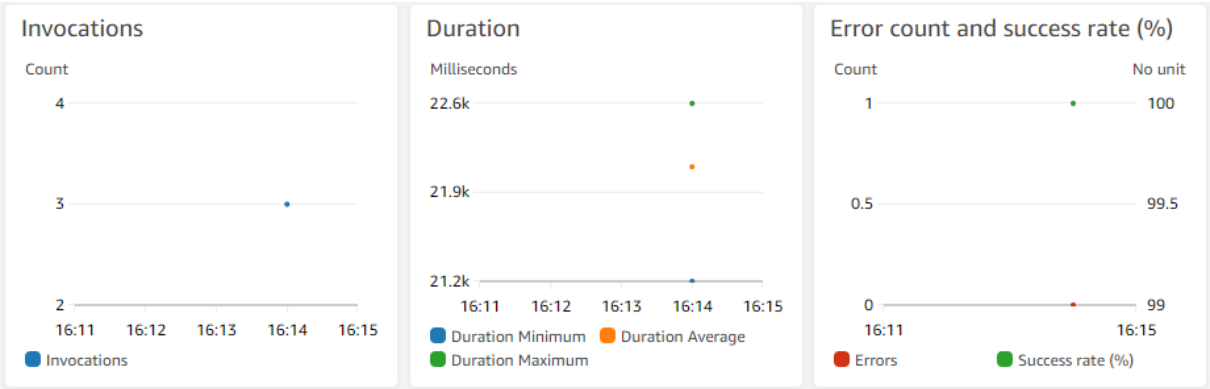

Figure 6.5: AWS Lambda monitoring console for eight simultaneous requests

The battle system should scale up to 500 simultaneous requests without degrading its performance

One of the biggest benefits in having microservices is the possibility of having a more refined scalability. In this case, to scale up to 500 simultaneous games, the system only needs to scale the number of local workers (up to the resources available in the system), and possibly the offloading managers and game manager (although these require a very small processing time when compared to the workers).

In terms of the current implementation, when using Docker and Traefik it is extremely simple to increase the number of any component (relay, workers and AWS Lambda dispatcher). In the future, as the company adopts Kubernetes, there are auto-scalling options available that can act by analysing the CPU consumption, for example. Regarding the cloud itself, as AWS Lambda is a serverless service, it is not possible to monitor or manage the underlying infrastructure, however the AWS defines that up to 1000 instances of Lambda functions can run concurrently [75] – enough to ensure the 500 requested.

To demonstrate the capabilities of the system, some tests were executed to understand the changes in performance with the growth on the number of concurrent requests. To do this, the same battle setup was selected and used throughout this experience. The number of local workers was set to five instances, meaning that the first five requests are dealt locally and the following are sent to the lambda dispatcher to be sent to the cloud. In order to monitor the performance of the requests, timestamps were retrieved at three moments: when the request arrived to the system, before the battle processing starts and right after the processing finishes. With that information it was possible to retrieve the time that it took for a request to start processing (network time) and the time taken to process the request (processing time). This experiment starts with a burst of five requests (that only run locally, as there were five local workers). Then, it gradually increases until it reaches a 500 burst of requests (5 running locally and 495 running in the cloud). On each burst, the median of the network time and the median of the processing time was calculated, the results being presented in Table 6.1.

Table 6.1: Results from burst experiment

Nº Requests	Nº Cloud Requests	Network Time (ms)	Processing Time (s)
5	0	9.4	43.15
100	95	17.8	46.13
200	195	155.6	45.08
300	295	31.5	40.75
400	395	181.9	41.17
500	495	29.0	36.62

From Table 6.1 it is possible to see that the network time changes a great deal between request numbers, yet these changes are not significant as they are in miliseconds (when compared to the processing time, in seconds). To better visualise the changes in the processing time, these are presented in Figure 6.6.

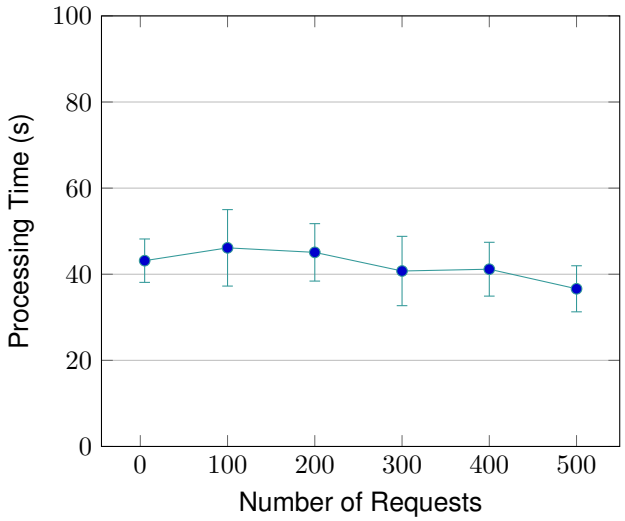

Figure 6.6: Processing time variation in the burst experiment

The variation of the processing time is also not significant, decreasing even from the 200 burst forward. It can be concluded that the processing time (and therefore the performance) is not influenced by the number of requests, and the deviation is probably related to AWS underlying infrastructure optimisation that is done. It is also inferred that the battle system does scale up to 500 simultaneous requests without degrading its performance, successfully meeting the requirement.

The system should host multiple games simultaneously

Gaming companies have various games. Instead of having their architecture separating each game as a different infrastructure, in this architecture, there is the possibility to have services taking care of similar services between different games. This enables the creation of infrastructures that host multiple games efficiently, grouping similar tasks so they only need to be coded and maintained once and in one place. Examples of similar tasks that were made centralised services are: log ins, user management, friendship management, and leader-board management.

The system is designed to have several match processing units available, having different functions inside, in order to provide all the necessary games. At the moment, Chilltime has only migrated the World War Online battle engine to the new infrastructure, but more will follow. When adding a new processing function, there are some steps that should be followed. First, a new relay service is required for players to be able to trigger battles in the platforms where they are playing. This relay should then publish a message in an exchange created for this purpose. The configuration should be similar: having a DLX, sending messages with TTL, configuring local workers with prefetch as one. The local consumers should then be created, wrapping around the battle processing code (or executable) in an application that connects to the queue to accept battles. Regarding the offloading manager, it should read from the DLX end queue and send a message to trigger the correct AWS Lambda function.

The system must have monitoring and logging services

Logging and monitoring are two crucial services in microservices that help developers perform efficiently. The architecture designed had both monitoring and logging services connected to every other service, as well as an alerting service and a back-office system for an easier access to the information.

In Chilltime's implementation, the Elastic Stack has enabled the creation of all these services in a very efficient way, as well as extra features to facilitate the filtering and visualisation of the data, making it easier to understand what is going on inside the system. As an example, in Figure 6.3 it is possible to see a number of logs retrieved from the system and in Figure 6.7 it can be seen a dashboard with some graphics rendered to more easily fetch relevant information from the logs and metrics retrieved about World War Online emails and battle engine usage.

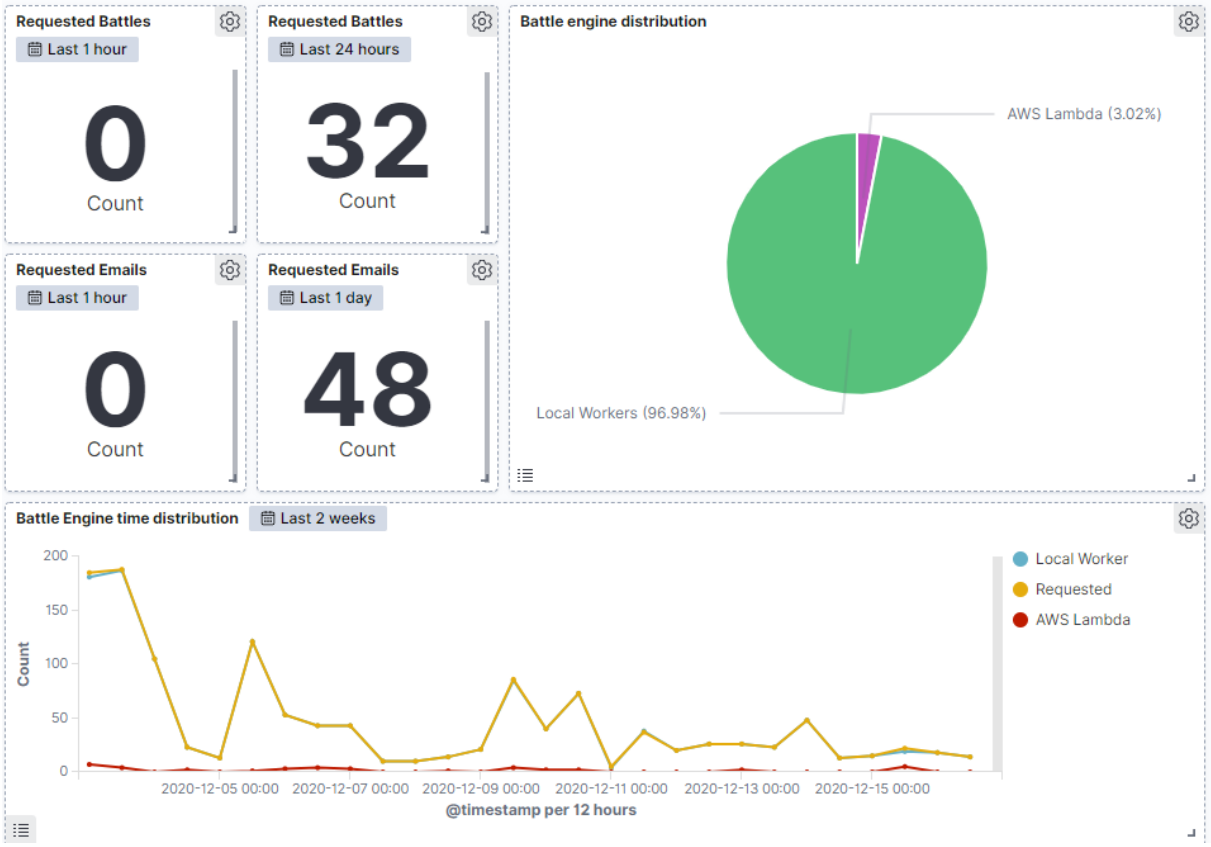

Figure 6.7: Dashboard for World War Online battle engine

The system must increase testability and observability

Testability becomes an easier and more controlled process using a microservices approach: as the systems are loosely coupled and isolated, it is much easier to test each feature separately without having to test the whole system. Testing can be done in an incrementally bigger scope and in an automatic way.

Observability is powered not only by having the logging and monitoring systems but also by the alerting and a back-office services that can make special requests to better understand the services' state.

Regarding the testing implementation, it was possible to use the combination of small scripts to trigger each feature implemented by simply making HTTP requests to the appropriate API endpoint (with the correct authentication and information to perform the task), as well as looking at the output in the logs to understand if the system was performing as expected. This can be further automated, once the company explores Jenkins and its possibilities for building pipelines. Moreover, in terms of observability, a distributed tracking system was also implemented, that can be explored by administrators in Kibana's interface.

Examples of monitoring, logging and alerting have been shown in Figure 6.2, Figure 6.3 and Figure 6.7. An example of distributed tracking in the email system can be seen in Figure 6.8, where the same `uuid` is used in the same request and it can be seen that they are received in the email-relay and later in email-worker for processing, as expected.

Time	uuid	status_code	status	service
Dec 16, 2020 @ 17:17:06.724	b939789a-3e02-41d8-9880-6f9005a9605f	200	INFO	email-worker
Dec 16, 2020 @ 17:17:06.600	b939789a-3e02-41d8-9880-6f9005a9605f	200	INFO	email-relay
Dec 16, 2020 @ 17:16:57.683	78ce7e91-42a2-4ff5-9700-fd6866010769	200	INFO	email-worker
Dec 16, 2020 @ 17:16:57.555	78ce7e91-42a2-4ff5-9700-fd6866010769	200	INFO	email-relay
Dec 16, 2020 @ 17:16:51.988	9ac00456-0cf2-4108-89ed-21791057d80a	200	INFO	email-worker
Dec 16, 2020 @ 17:16:51.608	9ac00456-0cf2-4108-89ed-21791057d80a	200	INFO	email-relay

Figure 6.8: Logs for emails with an unique ID (`uuid` column)

The system must be able to send messages to players via the adequate channel

In this architecture, it was added the email and the app notification system. These services do not depend on a specific email provider or notification provider, enabling the company to choose the more appropriate one and alter it in the future without problem.

In the implementation, Chilltime chose to use Mailjet and Firebase Cloud Messaging so it is now possible for these emails and notifications to be sent. In Figure 6.9 and in Figure 6.10 there is an example of an email and a notification for the user.

The communication system must be able to incorporate new channels

Even though the company uses the email and app notification only, it is expected that the notification channels evolve over time. Therefore, the architecture needed to be able to expand its functionality. In

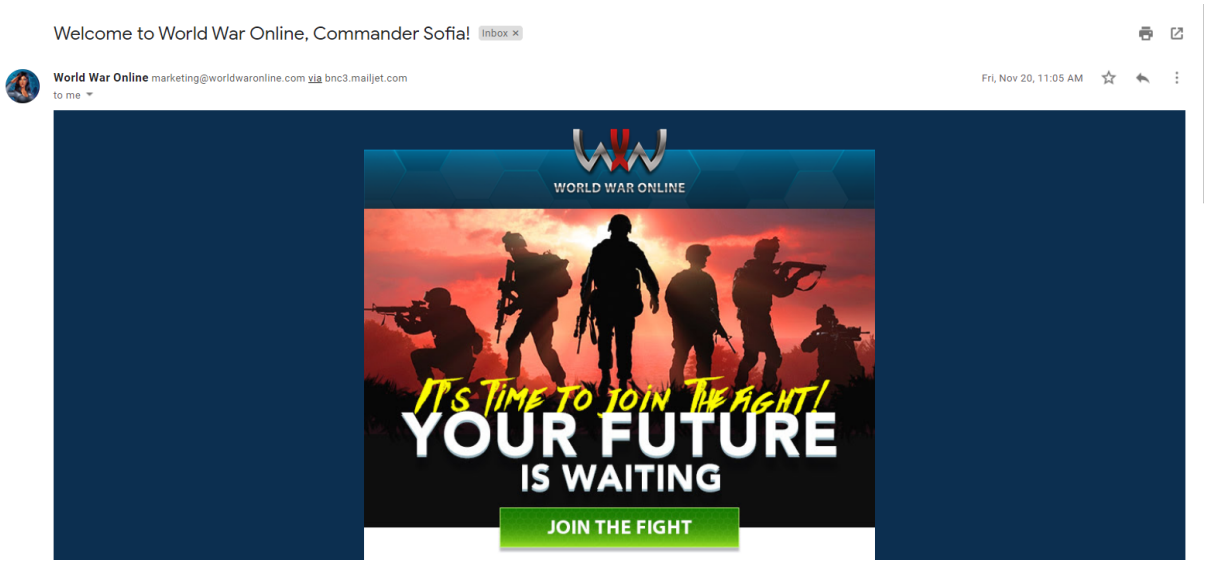

Figure 6.9: Email example with the email system

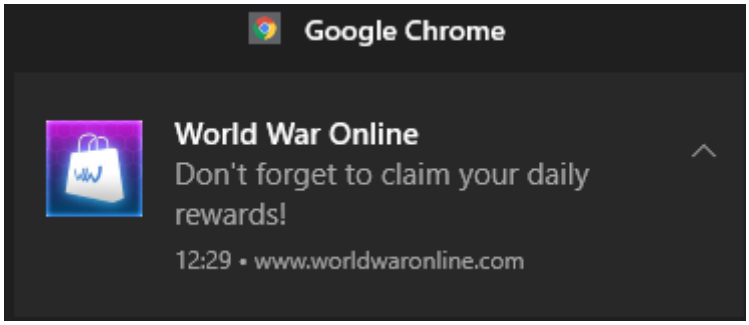

Figure 6.10: Notification example with the notification system

that regard, a microservice architecture can really help with extensibility, by creating a new service with the new functionality and exposing the API for other services to connect with.

Regarding the system's implementation, the steps would be to create a relay microservice (it should be very similar to the other communication services' relay) and connect it to a new exchange and queue on RabbitMQ, created specifically for the new channel. A second microservice should also be created, in a language that supports both RabbitMQ client libraries and connection to the new channels' API. Both containers should also send logs (via the `log` RabbitMQ queue) of the relevant information, with unique identifiers for distributed tracking.

7

Conclusion

Contents

This chapter presents the conclusions that can be derived from this research work as well as suggestions for future work based on this report's findings.

7.1 Conclusion

This report describes the work developed for the Master Thesis in Electrical and Computer Engineering. It starts with an overview of microservices – a relevant architectural approach designed for scalability and agility. It is followed by an analysis of the video games industry's evolution and research on the various challenges in its implementation for this business field. Moreover, the current cloud options available are also explored. Next, since the work conducted was accomplished in collaboration with Chilltime, their objectives and requirements for the new infrastructure are defined, followed by an explanation of MAGIC – the proposed architecture to provide a solid foundation for the implementation of a system that is capable of meeting those requirements. As it can be seen from Chapter 6, MAGIC is able to scale without performance reduction, moving to the cloud when necessary, while being able to ensure all functional requirements as well. Thus, the main goal of this project can be considered to have been achieved.

There is still much work left to do in Chilltime for the company to have the complete implementation of the architecture designed. However, it is not possible to accelerate an architecture migration process, due to the need of involving every team in the company, therefore requiring careful planning of each change made. For the elements that Chilltime needs to enforce later, an implementation plan was designed with tool suggestions and an analysis of the critical factors to help with the decisions.

While this architecture was implemented for the Portuguese business, it is important to state that this work was done with other gaming companies in mind as well, since the problems faced by Chilltime – performance issues due to scaling and difficulty in being more agile – also concern other companies in the same field. As such, this work is perfectly suitable for others to use and adapt the implementation to their specific needs. However, the size of the company can be particularly important when determining if this architecture is a good fit: a smaller company, with fewer users or games, might not benefit from a large number of microservices, as they might be bringing complexity and delays that do not solve any real issue. Similarly, bigger companies with millions of users should probably divide even more the resources and services needed in their infrastructure, for example, considering sharding the databases based on the locations of their users.

7.2 Future Work

Regarding the future work that can be done on top of this research and architecture, there are many interesting projects that can be tackled. The first would be to explore the possible changes in the implementation for a much bigger user base with millions of active players.

Another interesting topic is the exploration of other cloud services that a gaming company can benefit from, as in this work they are only being used for extending the computing power available for the company. There are various data, security, monitoring and machine learning APIs that can provide further functionality for these companies.

Finally, there are some game types with unique characteristics that might need special treatment. One example are the MMORPG that have huge persistent open worlds which can be explored by users and interact in real-time. These types of games require an efficient use of databases to manage the state of the world. Another type are the FPS games where the network management – such as the number of messages sent or the number of hops between microservices needed to perform the requests – can be certainly impactful due to these games being real-time, with a high rate of updates. As future work, it would be interesting to understand if these game specifications can be efficiently performed based on this architecture or how it would need to change in order to adapt to these games.

Bibliography

[1] J. Jörnmark, A.-S. Axelsson, and M. Ernkvist, "Wherever Hardware, There'll be Games: The Evolution of Hardware and Shifting Industrial Leadership in the Gaming Industry," in *Proceedings of the Digital Games Research Association International Conference (DiGRA)*, 2005, p. 13.

[2] D. Sanchez-Crespo, *Core Techniques and Algorithms in Game Programing*. New Riders Publishing, 2003.

[3] G. Armitage, M. Claypool, and P. Branch, *Networking and Online Games: Understanding and Engineering Multiplayer Internet Games*. John Wiley & Sons Ltd, 2006.

[4] K. Karnes. (2020) Hyper-Casual Games: Mobile Gaming's Greatest Genre — CleverTap. Accessed: 2020-08-03. [Online]. Available: https://clevertap.com/blog/hyper-casual-games/

[5] S. Ferretti and G. D'Angelo, "Online Gaming Scalability," *Encyclopedia of Computer Graphics and Games*, pp. 1–3, 2018.

[6] K. T. Chen, P. Huang, and C. L. Lei, "How sensitive are online gamers to network quality?" *Communications of the ACM*, vol. 49, no. 11, pp. 34–38, 2006.

[7] T. Barron and A. LaMothe, *Multiplayer Game Programming*, S. Doell, E. Smith, C. Snyder, and K. Simmons, Eds. Prima Publishing, 2001.

[8] D. Arsenault, "Video Game Genre, Evolution and Innovation," *Journal for Computer Games Culture*, vol. 3, no. 2, pp. 149–176, 2009.

[9] R. I. Clarke, J. H. Lee, and N. Clark, "Why Video Game Genres Fail: A Classificatory Analysis," *Games and Culture*, vol. 12, no. 5, pp. 445–465, 2017.

[10] J. Huang, *Encyclopedia of Computer Graphics and Games*. Springer International Publishing, 2020.

[11] A. R. Bharambe, S. Rao, and S. Seshan, "Mercury: A scalable publish-subscribe system for internet games," *NetGames 2002 - Proceedings of the 1st Workshop on Network and System Support for Games*, pp. 3–9, 2002.

[12] A. Bharambe, J. Pang, and S. Seshan, “Colyseus : A Distributed Architecture for Online Multiplayer Games,” *USENIX Association - NSDI ’06: 3rd Symposium on Networked Systems Design & Implementation 155*, pp. 155–168, 2006.

[13] L. Chan, J. Yong, J. Bai, B. Leong, and R. Tan, “Hydra: A massively-multiplayer peer-to-peer architecture for the game developer,” *Proceedings of the 6th ACM SIGCOMM Workshop on Network and System Support for Games, NetGames ’07*, pp. 37–42, 2007.

[14] J. Jardine and D. Zappala, “A hybrid architecture for massively multiplayer online games,” *Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games, NetGames’08*, pp. 60–65, 2008.

[15] M. Doherty, “A Software Architecture for Games,” *University of the Pacific Department of Computer Science Research and Project Journal (RAPJ)*, vol. 1, no. 1, 2003.

[16] S. Caltagirone, B. Schlief, M. Keys, and M. J. Willshire, “Architecture for a Massively Multiplayer Online Role Playing Game Engine,” *Journal of Computing Sciences in Colleges*, vol. 18, no. 2, pp. 105–116, 2002.

[17] S. Bogojevic, S. Bogojevic, M. K. September, and M. K. September, “The Architecture of Massive Multiplayer Online Games,” *Computer*, 2003.

[18] Q. Liu, “Integrating Game Engines into the Mobile Cloud as Micro-Services,” Ph.D. dissertation, University of Saskatchewan, 2018.

[19] C. Cardin, “Design of a horizontally scalable backend application for online games,” Ph.D. dissertation, Aalto University, 2016.

[20] M. Vähä, “Applying microservice architecture pattern to a design of an MMORPG backend,” Ph.D. dissertation, University of Oulu, 2017.

[21] K. Wiegers and J. Beatty, *Software Requirements*, 3rd ed. Redmond: Microsoft Press, 2013.

[22] C. Richardson, *Microservices Patterns*, M. Michaels, C. Mennerich, and L. Weidert, Eds. Shelter Island: Manning Publications Co., 2019.

[23] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns Elements of Reusable Object-Oriented Software*, 37th ed. Westford: Addison-Wesley, 2009.

[24] I. Nadareshivili, R. Mitra, M. McLarty, and M. Amundsen, *Microservice Architecture: Aligning Principles, Practices, and Culture*, 1st ed., B. MacDonals and H. Bauer, Eds. Sebastopol: O’Reilly Media, 2016.

[25] R. Chen, S. Li, and Z. Li, “From Monolith to Microservices: A Dataflow-Driven Approach,” *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, vol. 2017-Decem, pp. 466–475, 2018.

[26] D. Namiot and M. Sneps-Sneppe, “On Micro-services Architecture,” *International Journal of Open Information Technologies*, vol. 2, no. 9, pp. 24–27, 2014.

[27] S. Newman, *Building Microservices Designing Fine-Grained Systems*, 1st ed., B. MacDonald and M. Loukides, Eds. O’Reilly Media, 2015.

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

[29] J. Nemer. (2019) Advantages and Disadvantages of Microservices Architecture. Accessed: 2020-11-20. [Online]. Available: https://cloudacademy.com/blog/microservices-architecture-challenge-advantage-drawback/

[30] R. C. Martin, *Designing object-oriented C++ applications using the Booch method*. Prentice Hall, 1995.

[31] K. Bakshi, “Microservices-Based Software Architecture and Approaches,” *IEEE Aerospace Conference Proceedings*, 2017.

[32] M. Mark, *REST API Design Rulebook*. O’Reilly, 2013, vol. 53, no. 9.

[33] S. Daya, N. Van Duy, K. Eati, C. M. Ferreira, D. Glozic, V. Gucer, M. Gupta, S. Joshi, V. Lampkin, M. Martins, S. Narain, and R. Vennam, *Microservices from Theory to Practice: Creating Applications in IBM Bluemix Using the Microservices Approach*, 1st ed. IBM Redbooks, 2015.

[34] Pattern: Messaging. Accessed: 2020-12-19. [Online]. Available: https://microservices.io/patterns/communication-style/messaging.html

[35] G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building and Deploying Messaging Solutions*, 15th ed. Add, 2005.

[36] S. Gilbert and N. Lynch, “Perspectives on the CAP Theorem,” *Computer*, vol. 45, no. 2, pp. 30–36, 2012.

[37] J. Bacon and T. Harris, *Operating Systems: Concurrent and Distributed Software Design*. Addison Wesley, 2003.

[38] Pattern: Sagas. Accessed: 2020-12-19. [Online]. Available: https://microservices.io/patterns/data/saga.html

[39] Pattern: Event Sourcing. Accessed: 2020-12-19. [Online]. Available: https://microservices.io/patterns/data/event-sourcing.html

[40] Pattern: API Composition. Accessed: 2020-12-19. [Online]. Available: https://microservices.io/patterns/data/api-composition.html

[41] Pattern: CQRS. Accessed: 2020-12-19. [Online]. Available: https://microservices.io/patterns/data/cqrs.html

[42] Pattern: Multiple service instances per host. Accessed: 2020-12-19. [Online]. Available: https://microservices.io/patterns/deployment/multiple-services-per-host.html

[43] Pattern: Service Instance per VM. Accessed: 2020-12-19. [Online]. Available: https://microservices.io/patterns/deployment/service-per-vm.html

[44] Pattern: Service instance per container. Accessed: 2020-12-19. [Online]. Available: https://microservices.io/patterns/deployment/service-per-container.html

[45] B. Burns, J. Beda, and K. Hightower, *Kubernetes Up & Running*, 2nd ed. O'Reilly Media, Inc., 2019.

[46] Pattern: Serverless deployment. Accessed: 2020-12-19. [Online]. Available: https://microservices.io/patterns/deployment/serverless-deployment.html

[47] Pattern: API Gateway / Backends for Frontends. Accessed: 2020-12-19. [Online]. Available: https://microservices.io/patterns/apigateway.html

[48] Pattern: Access token. Accessed: 2020-12-19. [Online]. Available: https://microservices.io/patterns/security/access-token.html

[49] Pattern: Log aggregation. Accessed: 2020-12-19. [Online]. Available: https://microservices.io/patterns/observability/application-logging.html

[50] Pattern: Health Check API. Accessed: 2020-12-19. [Online]. Available: https://microservices.io/patterns/observability/health-check-api.html

[51] Pattern: Application metrics. Accessed: 2020-12-19. [Online]. Available: https://microservices.io/patterns/observability/application-metrics.html

[52] Pattern: Distributed tracing. Accessed: 2020-12-19. [Online]. Available: https://microservices.io/patterns/observability/distributed-tracing.html

[53] J. Hurwitz, R. Bloor, M. Kaufman, and F. Halper, *Cloud Computing for Dummies*. Wiley Publishing, Inc., 2010.

[54] Openstack. Accessed: 2020-12-24. [Online]. Available: https://www.openstack.org/

[55] M. J. Kavis, *Architecting the Cloud*. Wiley, 2014.

[56] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, "Cloud Container Technologies: a State-of-the-Art Review," *IEEE Transactions on Cloud Computing*, vol. 7161, no. c, pp. 1–14, 2017.

[57] M. K. Hussein, M. H. Mousa, and M. A. Alqarni, "A placement architecture for a container as a service (CaaS) in a cloud environment," *Journal of Cloud Computing*, vol. 8, no. 1, pp. 1–15, 2019.

[58] M. Stigler, *Beginning Serverless Computing*, 1st ed. Apress Media, 2018.

[59] AWS. (2017) Introducing AWS Fargate. Accessed: 2020-12-10. [Online]. Available: https://aws.amazon.com/about-aws/whats-new/2017/11/introducing-aws-fargate-a-technology-to-run-containers-without-managing-infrastructure/

[60] H. Joseph, *Unity in Action*, 2nd ed. Manning Publications Co., 2018.

[61] D. F. Galletta, R. Henry, and S. Mccoy, "Web Site Delays : How Tolerant are Users?" *Jais*, vol. 5, no. December 2002, pp. 1–28, 2003.

[62] M. Consalvo, *Cheating: gaining advantage in videogames*. Cambridge: The MIT Press, 2007.

[63] S. Brown. (2019) C4 Model. Accessed: 2020-12-06. [Online]. Available: https://c4model.com

[64] A. Vazquez-Ingelmo, A. Garcia-Holgado, and F. J. Garcia-Penalvo, "C4 model in a software engineering subject to ease the comprehension of UML and the software," *IEEE Global Engineering Education Conference, EDUCON*, vol. 2020-April, no. April, pp. 919–924, 2020.

[65] K. Matthias and S. P. Kane, *Docker Up & Running*, 2nd ed. O'Reilly Media, Inc., 2018.

[66] Overview of Docker Compose. Accessed: 2020-12-25. [Online]. Available: https://docs.docker.com/compose/

[67] G. M. Roy, *RabbitMQ in depth*. Manning Publications Co., 2018.

[68] Traefik Documentation. Accessed: 2020-12-10. [Online]. Available: https://doc.traefik.io/traefik/

[69] P. Shukla and S. Kumar M N, *Learning Elastic Stack 6.0*. Packt Publishing Ltd., 2017, vol. 53, no. 9.

[70] T. Hughes-Croucher and M. Wilson, *Node Up and Running*, 1st ed. O'Reilly Media, Inc., 2012, vol. 53, no. 9.

[71] B. Laster, *Jenkins 2 Up and Running*. O'Reilly Media, Inc., 2018.

[72] Mailjet Main Page. Accessed: 2020-12-10. [Online]. Available: https://www.mailjet.com/

[73] Firebase Cloud Messaging. Accessed: 2020-12-10. [Online]. Available: https://firebase.google.com/products/cloud-messaging

[74] P. Sbarski, *Serverless architectures on AWS*. Manning Publications Co., 2017.

[75] AWS Lambda Quotas. Accessed: 2020-12-17. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html