Microservices Architecture for Gaming Industry Companies

Sofia Estrela
Instituto Superior Técnico
Universidade de Lisboa
sofia.estrela@tecnico.ulisboa.pt

***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, leaderboards 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.**

***Index Terms*—video gaming industry; system architecture; microservices; scaling; hybrid cloud**

I. Introduction

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].

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 the user base grows, the system has to scale accordingly, increasing the resources available to tasks that 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 [2].
- High availability – Users expect to be able to play at any time, with minimal delay. When latency and jitter are an issue, users rapidly lose interest in the game [3].

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, that like many game development companies, started to have performance issues, with the growth of the offered games and rise in the number of players, due to the current structure being 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.

In this study, Microservices Architecture for Gaming Industry Companies (MAGIC) is presented and its implementation exemplified. It offers a 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 the company deal with peaks in its load: offloading the exceeding requests to the cloud has become a possibility.

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.

This report is organised as follows: In Sec. II are another research works done in related areas. In Sec. III the context, objectives and requirements of Chilltime are presented. In Sec. IV is shown the architecture of MAGIC and in Sec. V its implementation. Finally in Sec. VI an evaluations of the requirements is done based on the architecture and implementation choices, with Sec. VII concluding this report.

II. State of the Art and Related Work

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 [4]–[6]. 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 done about the integration of game engines as microservices in the mobile cloud [7] as well as back-end architectures for scalability gains and for Massive Multiplayer Online Role-Playing Games (MMORPG) [8], [9].

While the last research topic referred is significantly related to the research done in this report, 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.

III. Context, Objectives and Requirements

A. Context

As it was previously stated, this research work was done in collaboration with Chilltime. 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 (Real Time Strategy (RTS) on browser), Marble Adventures (mobile puzzle) and Soccer Avatars (mobile quiz).

World War Online has the biggest user base and suffers the most from the architecture's limitation. Players battle against each other with the goal of building the biggest empire. 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 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.

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.

Chilltime also interact with their players via email and app notifications to both mobile phones and desktops.

The original architecture in the organisation 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 running LAMP (Linux, Apache, MySQL and PHP) servers. 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, making the games slower for users and harder to update or extend for developers. Scaling this system was a complex affair, 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 also led to security vulnerabilities. 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 more challenging.

B. Objectives

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. Based on this information, the main objectives for the new architectur eare 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.

C. Requirements

A set of requirements is being defined for the system, based on the previous discussion. As such, the requirements are:

- The system should guarantee isolation;
- The components must auto-recover from failure;
- The system should not have internal polling requests;
- The components should be deployable off-site;
- The battle system should auto-scale off-site;
- The battle system should scale up to 500 simultaneous requests without degrading its performance
- The system should host multiple games simultaneously
- The system must have monitoring and logging services
- The system must increase testability and observability
- The system must be able to send messages to players via the adequate channel
- The communication system must be able to incorporate new channels

IV. Architecture

MAGIC is based on Microservices, so this section starts with a brief introduction of this concept and its most relevant patterns. A microservice is an autonomously developed and independently deployable component that implements useful functionality and has a bounded context with a very clear usage API and supports interoperability through message-based communication [10], [11]. Microservices ensures two key concepts: Loose coupling and high cohesion [12]. The first ensures a degree of isolation between services and the second ensures that everything similar or related is positioned together, the rest being in a separate environment. The benefits of microservices include:

- An effective way of scaling, as the separation of concerns enables scaling only what is needed.
- Improved resilience, as the loose coupling of services limits errors' area of impact. [13].
- Autonomous teams, by minimising the cross collaboration, as services are very independent. This enhances the teams' efficiency [10].
- Technology heterogeneity and optimisation for replaceability, as changes involves less risk and effort, facilitating the exploration new software and language and eliminating technology lock-in [12].
- Composability by ensuring the reduction of development time through increased opportunities for reusability [10].
- Ease of deployment as there is no need to test the whole system for every modification made, only the service that has been changed [12].
- Increased agility as it is easier for companies to adapt to changes and try new paths and features [10].
- Less complexity in the code as services become more independent and smaller in size, helping developers to work quicker [10].

To implement microservices there are some pattern of utilisation that should be explored and adopted where beneficial. The first is related to communication which can be synchronous or asynchronous. In the first, the client waits for the server to give a response, blocking until received, being Representational State Transfer (REST) the main implementation pattern. The asynchronous communication does not expect an answer, the main example being messaging. Synchronous communication is easier to implement but asynchronous communication decreases the runtime coupling between services, enhancing the benefits of microservices.

Regarding data, both read and write operations can be analysed. Making queries can be done by either composing API requests or using an event sourcing approach, which consists in having services publishing their changes to the network and having services subscribing to these streams. Command Query Responsibility Segregation (CQRS) can be used on top, to have a "rolling snapshot" of the state of a value, based on the event source stream received, which requires a more complex implementation that API composition but ensures decoupling.

Regarding data transaction, as they are distributed, the most common pattern in microservices are sagas [11] which define a sequence of local transaction that guarantee eventual consistency. Sagas can be either choreographed or orchestrated, the first having each node connecting to the next, while in the second there is an orchestrator agent that manages the transaction [12], requiring a more complex (although centralised) implementation.

MAGIC aims to create a company-wide structure that complies with the objectives and requirements while having in consideration the business functionalities required. 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 [14].

A. *First Layer – Context*

This layer of the C4 model defines the context of the application. The diagram for the first layer is presented in Fig. 1.

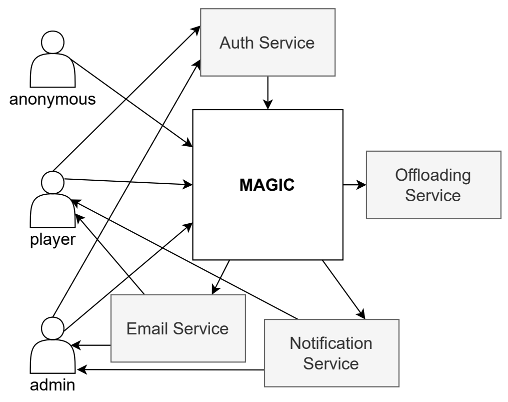

Fig. 1. Context Abstraction Layer

Starting by the leftmost components – the users – there exists three main types: anonymous that do not have an account in the company, players that can play the games and administrators that have unrestricted access to the services.

Referring again to Fig. 1, these users communicate with the central component – MAGIC. This component represents the companies' infrastructure, including the game engines, the web-pages, Application Program Interface (API) and back-office support services.

The registered users 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, which 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.

B. *Second Layer – Containers*

This layer will focus on MAGIC main components. The diagram for the second layer is presented in Fig. 2.

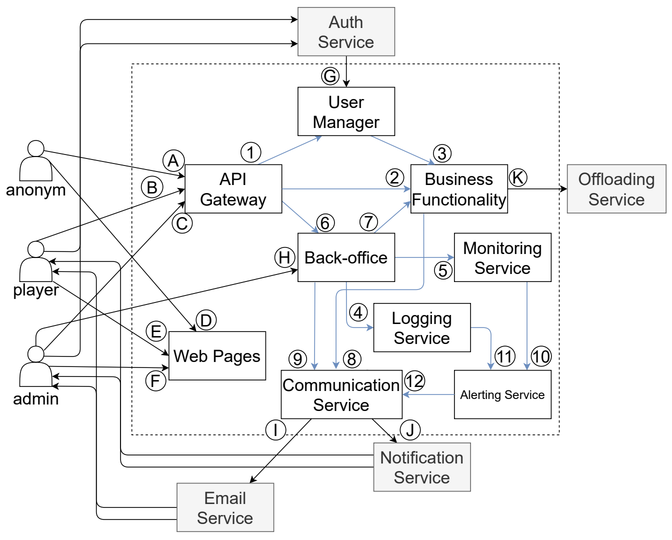

Fig. 2. Container Abstraction Layer

Regarding the internal components, they're as follows:

- API Gateway – Conjugated with the Auth Service, it provides the first barrier of any API request to the system, also having a service discovery mechanism to route the requests to the correct service before sending the answer to the client.
- Web Pages – It contains all web services which are accessible by the users.
- Communication Services – It contains an interface that trigger the sending of email/notifications to the correct users.
- User Manager – It connects the user's information, with the access tokens and identifier from the Auth Service.
- Business Functionality – It contains all services regarding the specific functionality of the company which, in this case, is related to gaming.
- Back-office – It represents a set of web pages and respective backend functions with the authorisation to perform special queries and perform actions related to the administration and management of the products.
- Logging – Mechanisms to retrieve logs from the services and storing them in a way that is easy to filter and analyse.
- Monitoring – Mechanisms that verify if the services are online and healthy, by collecting metrics and performing health checks.
- Alerting – It analyses the the logging and monitoring output and triggers alarms when certain errors happen or thresholds are surpassed.

All these components relate to each other and to external services. In Fig. 2, internal relationships are marked with a number and external interactions with capital letters.

The API Gateway component receives requests from the users (A, B and C), which are then identified/validated in the User Manager (1) and sent to the service with the answer inside the Business Functionality (2). The users also connect with the Web Pages (D, E and F).

The User Manager service receives identification and validation requests by the API Gateway and information from the external Auth Service. This user identification is attributed to the whole company, independently of the game they are connecting to. The relationship between users and games is only analysed inside the Business Functionality component, hence the connection between these services (3).

The admin user 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 accessing to the Business Functionality (7). If these requests involve communicating with the client, the service can solicit that request to the Communication Service (8).

The Business Functionality receives requests from the API Gateway (2), User Manager (3) and Back-Office (7). These requests are directly related to the data and action from games. 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. Some might need to be forwarded to the Communication Service (9), to ensure the user is notified of a certain event.

The Communication Service sends the formatted messages to the specific communication channels chosen – in this architecture either email or notification (I and J, respectively). The Business Functionality component might also request services from a cloud provider, if it decides that the existing local resources are not enough to ensure timely execution of the tasks, being sent to the Offloading Service (K).

The Alerting Service analyses the data in the Monitoring and Logging services (10 and 11, respectively) and, if deemed necessary, an email can be sent to the appropriate person, by sending a request to the Communication Service (12).

On a last note, although it is not represented in Fig. 2 as to produce a more understandable image, the Logging and Monitoring services are connected to every service represented in the diagram to retrieve the logs and metrics.

C. *Third Layer – Components*

The third layer focuses on the components that allow the containers presented in the second layer to perform as desired. Regarding the Business Functionality components, the layer three diagram of this container is presented in Fig. 3.

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

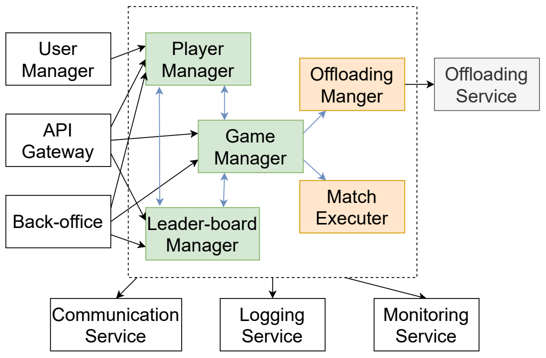

Fig. 3. Components Abstraction Layer – Business Functionality

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 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.

V. Implementation

Changing the infrastructure of an existing company is a complex project. 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 [10]. 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.

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 creation of new features also generates great opportunities to continue evolving the system.

Regarding Chilltime's implementation process, it was no different. Not all elements have been migrated to the new infrastructure, due to the constraints referred to previously.

A. Technology Stack

1) Docker: Docker [15] offers the first mature implementation of container management promising significant productivity gains in the DevOps area due to the Infrastructure as Code (IaC) approach, and an easy management and adaptability of the deployment environment, which was not possible with Virtual Machines (VM).

Docker works by having a daemon on the server that manages 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 [16], 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 facilitate the scaling process of certain workers, as the number of instances can be easily changed with the `docker-compose` command. Finally, Docker Compose has restart policy options – it can be set to `no`, `always`, `on-failure` and `unleass-stopped` – helping with fault-tolerance.

2) 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 [17]. 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.

3) RabbitMQ: As for message brokers, the chosen was RabbitMQ – an open-source, lightweight yet extremely powerful and versatile message broker [18]. 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 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. 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.

4) 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 [19]. This edge router is capable of two types of routing: Layer 4 (TCP – based on the 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.

5) 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 [20], "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.

6) Node.js: Based on JavaScript, Node.js is an event-driven language that can produce highly scalable servers using an event loop software architecture [21]. 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.

7) Jenkins: Jenkins is a tool that can construct deployment pipelines that is able to build, test and deploy the company's software [22]. It can easily be integrated with Kubernetes to automatically deploy the new versions on the system, being a very interesting option to automate the deployment cycle for the company.

B. Environment Setup

The starting point 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, all services installed where deployed in Docker containers. In an initial approach, it was not used any container orchestrator, due to the system being only in this single physical server.

Additionally, a message broker – RabbitMQ – for asynchronous communication was installed, due to asynchronous messaging being generally a better option to ensure less coupling between services and enabling one-to-many communication. Thus, in cases where there are clear benefits from using messaging, this should be the chosen option.

C. API Gateway

The API Gateway was implemented right after due to representing the entry point of the system. For this, Traefik was installed, adding services such service discovery, routing and load balancing to the system.

D. Communication Services

The communication service was adopted when Chilltime changed their email sender provider to one more suitable for their needs: Mailjet [23]. The necessity to create this new feature provided the perfect opportunity to migrate. The app notification service immediately followed using Firebase Cloud Messaging as its external service provider [24]. In Fig. 4 the implementation of the communication services are presented.

When the service sends an HTTP request to the correct hostname, Traefik will direct the message to the correct endpoint based on its hostname and path, forwarding it to the correct communication container. The first containers reached are the relay services that simply change the communication from synchronous to asynchronous. There is relay service one per communication channel, both written using Node.js.

These relay services also provide an extra layer of security using Hash-based Message Authentication Code (HMAC) as 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.

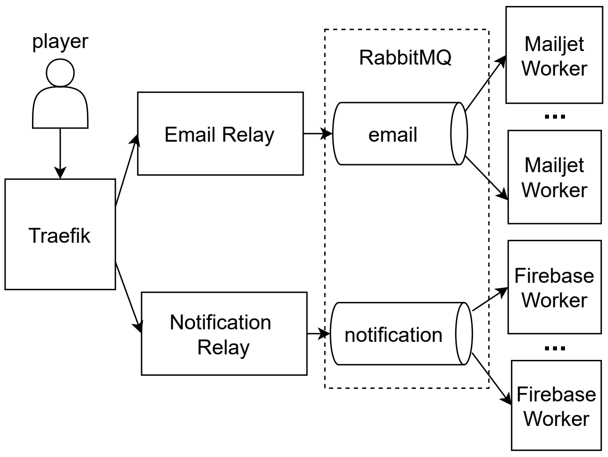

Fig. 4. Implementation of the Communication Services

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.

E. Logging, Monitoring, Alerting and Back-office

Observability is an essential trait in microservices, therefore, Elastic Stack was implemented to ensure the observability components of MAGIC – the Back-Office, Monitoring, Logging and Alerting Services. With this, it was possible to better understand how the containers were performing and to more easily debug issues. In Fig. 5, the implementation is presented.

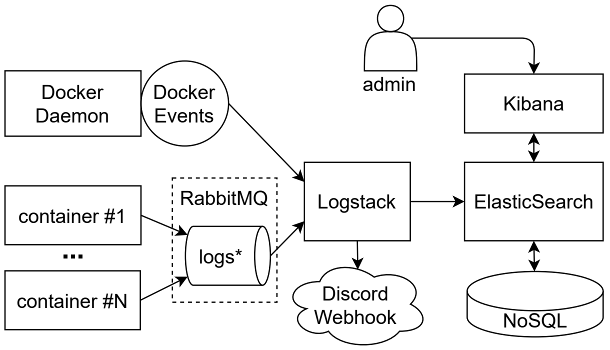

Fig. 5. Back-office, Logging, Monitoring and Alerting – Implementation

The most important logs to collect are about the requests received and the 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 container 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. These logs are 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 implement distributed tracking of requests, to further help debugging, as logs are given a request ID that makes it feasible to track requests over containers.

Regarding health checks, it is possible to achieve through Docker deamon, who does these requests to the containers and monitoring the responses, emitting an event in case a service starts, stops or becomes unhealthy and redirecting them to Logstash, it becomes possible to retrieve the aforementioned metrics. This way, it is possible to monitor the containers' behaviour. As for the alerting capabilities, they are implemented by having the Logstash sending a message to a Discord webhook – the internal messaging system in the team – in case docker event warns about a stopped or an unhealthy container.

F. Game Manager, Match Executor, Offloading Manager – World War Online

These services were implemented for World War Online – the most played game at Chilltime – since the battle engine had to be updated. 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 Fig. 6, where the system for the battles is presented.

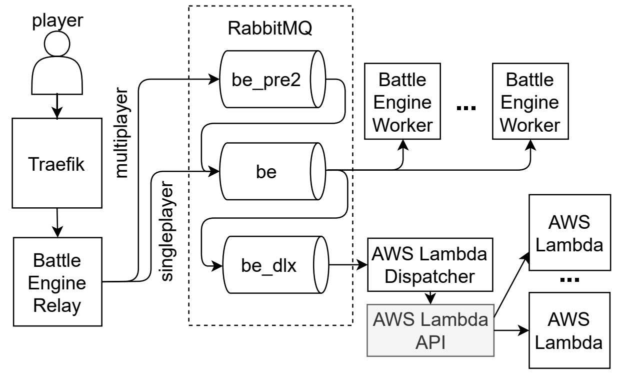

Fig. 6. World War Online Battle Engine – Implementation

When choosing the cloud solution, a serverless solution was preferred due to not having to deal with any management of the underlying infrastructure as opposed to Infrastrcture as a Service (IaaS), Container as a Service (CaaS) or Platform as a Service (PaaS) solutions. this lead to choosing AWS Lambda (and S3 for uploading purposes) for the job [25].

As for RabbitMQ, Dead Letter Exchange (DLX) and messages Time-To-Live (TTL) were added to the configuration. 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, which causes the redirection of the message to the DLX for another service to consume. By using message ackowleges and adjusting the prefect consumer value – the number of simultaneous requests a consumer can handle –, it is possible for the message to be redirected to the DLX instantaneously if not consumer is available – with prefetch as one and the message TTL as zero.

When the player triggers a battle, it is then directed to Traefik which routes the request to the relay. In case of a singleplayer match, it is processed directly after the user triggers the battle. Focusing on singleplayer, messages are sent with TTL as zero, to the `be` queue. If they can be consumed immediately, they run locally. Otherwise, they are sent to the DLX, which places 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. Since multiplayer waits for two minutes before processing the battle, it has an 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, being redirected to the DLX queue – the `be` queue for singleplayer battles.

Regarding the containers themselves, all services use Node.js, the relay service having HMAC protection on the exposed API and the AWS Lambda dispatcher having the AWS client library. Within the Lamba functions, there is a simple python code that battle processing files.

G. 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. 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 decomposition of the monolith, in a first stage, the database can be shared among services. This should be temporary as it highly increases the coupling between services. Data transactions are crucial and for that goal, 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. Furthermore, 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.

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.

VI. Evaluation

A. Chilltime's Objectives Evaluation

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.

Regarding delay minimisation, with the correct implementation plan microservices can be helpful. It is important that 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.

AS for ensuring 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 concern but having Docker Compose, the reset options offered may 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 the microservices isonaltion of components. This modularisation helps to easily modify or replace components withou impacting the adjacent ones. This characteristic also eases code maintenance and extension, since similar features are centralised.

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.

B. Requirements Evaluation

1) The system should guarantee isolation: 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.

C. The components must auto-recover from 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. 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.

1) 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 the usage of asynchronous messages that is typical from microservices, enables the creation of an alternative. 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.

2) 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 Container as a Service (CaaS) options and orchestration services supplied by cloud providers.

3) The battle system should auto-scale off-site: The match processing units are by nature one of the most resource-intensive tasks. In order to achieve this, the architecture has an offloading manager 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 easily adjusted, ensuring that the system can have an efficient use of the local resources before sending requests to the cloud.

4) 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).With Kubernetes, there are auto-scalling options available that can act by analysing the Central Processing Unit (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 states that up to 1000 instances of Lambda functions can run concurrently [26] – enough to ensure the 500 requested.

5) 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.

6) 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. The Elastic Stack has enabled the implementation of all these services in a very efficient way, as well as extra features to facilitate the filtering and visualisation of the data.

7) The system must increase testability and observability: Testability becomes an easier 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 every time. 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 even more 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.

8) 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 change it when needed. In the implementation, Chilltime chose to use Mailjet and Firebase Cloud Messaging.

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

VII. Conclusion

As it can be seen in this report, MAGIC provides a solid foundation for the implementation of a system that is capable of meeting the given requirements. MAGIC is able to scale without performance reduction, moving to the cloud when necessary, while being able to ensure all functional requirements as well and facilitating the work of employees. Thus, the main goal of this project is considered to be achieved.

There is still much work left to do for the company to have the complete implementation of the architecture. 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 reflected 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.

A. Future Work

Regarding the future work, 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. There are various data, security, monitoring and machine learning API that can provide further functionality.

Finally, there are some game types with unique characteristics that might need special treatment, such as 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. On the other hand, First Person Shooter games's network management – such as the number of messages sent or the number of hops between microservices – can certainly be 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.

References

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

[3] 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.

[4] 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.

[5] 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.

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

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

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

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

[10] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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