



**Gabriel Adorno Marcondes**

Bachelor in Computer Science

## **Developing A Multi Application Real-Time Platform Using Cloud Serverless Technologies**

Dissertation submitted in partial fulfillment  
of the requirements for the degree of

Master of Science in  
**Computer Science and Engineering**

Co-advisers: Bernardo Toninho, Assistant Professor, Faculdade de  
Ciências e Tecnologia - Universidade Nova de Lisboa  
Mário Franco, CTO, Magycal Interactive

Examination Committee

Chair: Prof. Dr. Nuno Manuel Ribeiro Preguiça  
Rapporteur: Prof. Dra. Carmen Pires Morgado



FACULDADE DE  
CIÊNCIAS E TECNOLOGIA  
UNIVERSIDADE NOVA DE LISBOA

April, 2021



## **Developing A Multi Application Real-Time Platform Using Cloud Serverless Technologies**

Copyright © Gabriel Adorno Marcondes, Faculty of Sciences and Technology, NOVA University Lisbon.

The Faculty of Sciences and Technology and the NOVA University Lisbon have the right, perpetual and without geographical boundaries, to file and publish this dissertation through printed copies reproduced on paper or on digital form, or by any other means known or that may be invented, and to disseminate through scientific repositories and admit its copying and distribution for non-commercial, educational or research purposes, as long as credit is given to the author and editor.



## ACKNOWLEDGEMENTS

Thanks NOVA School of Science and Technology | FCT NOVA for sheltering me during this many years of learning, challenges and accomplishments. You will always be a home for me.

Thank you Department of Informatics for refining me into the gem I am today with all your pressure and motivation.

Thank you Bernardo Toninho for such dedicated work and help during this trial that put me to the test many times.

Thank you Magycal for providing me with the incentive and opportunity to complete an objective that was in stand by for a few years.

Thanks friends for being there for me whenever I needed. Catarina, Cordeiro, Micah, Ritas and Silva for our lunches. Silvia for all the love and care. Nelson for the fun and laughing. And many others that I carry with me wherever I go. It meant a lot to have you along the way so I didn't feel so lonely.

Thank you father for all the support and unconditional love. Thank you sister for helping even from a distance with all your craziness. Thank you mother, wherever you are, for being a paragon of faith and consistency that I try to follow.

Thank you Joana for making me choose to be the best version of myself every day.



## ABSTRACT

---

Magycal Interactive is a software company that has produced a significant impact in the Portuguese television sector. Magycal is Magycal Interactive's cloud based server-side framework that was developed to standardize common services (chats, polls, authentication) provided by applications such as Viva Ronaldo, Secret Story e SPORT TV Digital Hub.

As popularity and success of each application increases, Magycal becomes more technically outdated. Its monolithic architecture, which previously allowed for easy development is becoming a development bottleneck. Scaling the server is increasing in cost as the platform grows, and developing updates and new features is more difficult since services are becoming more tightly coupled with each release.

In this work, we propose an architectural shift for Magycal where we decouple services for better scalability, development and deployment. After a study of existing architectural options, we have concluded that the most suitable candidate architecture that meets the demands of Magycal is the microservices architecture. To test our hypothesis and determine the feasibility of the architectural change, we have selected a service of Magycal that was implemented following a microservice-oriented design.

Our implementation was validated via API calls to ensure the modifications maintained correct behavior of the framework. The new service had its implementation benchmarked and compared to the corresponded Magycal existing service. We concluded that the changes to Magycal yield a more robust framework with reduced costs of maintaining, development and deployment.

**Keywords:** cloud, monolithic applications, software architectures, microservices

---





## RESUMO

---

A Magycal Interactive é uma empresa de software que produz um impacto significativo no setor televisivo português. Magycal é a plataforma servidor da empresa na *cloud* desenvolvida para padronizar serviços comuns (canais de conversa, votações, autenticação) fornecidos por aplicações como Viva Ronaldo, Secret Story e SPORT TV Digital Hub.

À medida que a popularidade e o sucesso de cada aplicação aumenta, o Magycal torna-se tecnicamente mais desatualizado. A sua arquitetura monolítica, que anteriormente permitia desenvolvimento fácil, torna-se um problema. O custo de escalabilidade do servidor está a aumentar à medida que a plataforma cresce, e o desenvolvimento de atualizações e novos recursos é mais difícil, pois os serviços tornam-se mais fortemente acoplados a cada nova versão.

Neste trabalho, propomos uma mudança de arquitetura para o Magycal, onde dissociamos os serviços para melhor escalabilidade, desenvolvimento e *deployment*. Após um estudo das opções arquiteturais existentes, concluímos que a arquitetura candidata mais adequada às necessidades do Magycal é a arquitetura de microserviços. Para testar nossa hipótese e determinar a viabilidade da mudança arquitetural, selecionamos um serviço do Magycal que foi implementados seguindo um design orientado a microserviços.

A nossa implementação foi validada com chamadas API para garantir que as modificações mantiveram o comportamento correto da estrutura. O novo serviço teve a sua implementação medida e comparadas ao serviço existente no Magycal. Foi concluído que as mudanças no Magycal produzem uma estrutura mais robusta, com custos reduzidos de manutenção, desenvolvimento e implementação.

**Palavras-chave:** *cloud*, aplicações monolíticas, arquiteturas de software, microserviços

---



# CONTENTS

<b>List of Figures</b>	<b>xiii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Context . . . . .	1
1.1.1 Business Context . . . . .	1
1.1.2 Magycal . . . . .	2
1.2 Accomplished Work . . . . .	5
1.3 Document Layout . . . . .	5
<b>2 State of the art</b>	<b>7</b>
2.1 The Cloud . . . . .	7
2.1.1 Cloud Services . . . . .	7
2.1.2 Deployments of The Cloud . . . . .	8
2.1.3 Where does Magycal fit? . . . . .	9
2.2 Software Architecture . . . . .	9
2.2.1 Layered Architecture . . . . .	9
2.2.2 Event Driven Architecture . . . . .	10
2.2.3 Microkernel Architecture . . . . .	12
2.2.4 Microservices Architecture . . . . .	13
2.2.5 Space-based Architecture . . . . .	15
2.3 Microservices . . . . .	17
2.3.1 Advantages over Monolith . . . . .	17
2.3.2 Drawbacks of the Architecture . . . . .	17
2.3.3 From Monolith into Microservice . . . . .	18
<b>3 Solution</b>	<b>21</b>
3.1 Context . . . . .	21
3.2 Magycal . . . . .	21
3.2.1 Amazon Resources . . . . .	22
3.2.2 Chat service . . . . .	23
3.2.3 Challenges and Remarks . . . . .	24
3.3 Microservice Chat . . . . .	25
3.3.1 Comparison of Architectures . . . . .	25

## CONTENTS

---

3.3.2	API Validation . . . . .	26
3.3.3	Load Balancer . . . . .	27
<b>4</b>	<b>Benchmark</b>	<b>29</b>
4.1	Set up . . . . .	29
4.2	Methodology . . . . .	29
4.3	Comparison . . . . .	30
4.3.1	Results . . . . .	31
4.3.2	Conclusions . . . . .	33
<b>5</b>	<b>Conclusion</b>	<b>35</b>
5.1	Summary . . . . .	35
5.2	Upgrading Magycal . . . . .	36
5.3	Future Work . . . . .	36
	<b>Bibliography</b>	<b>39</b>

## LIST OF FIGURES

1.1	Set of Magycal Modules . . . . .	3
1.2	Example of Magycal Flow . . . . .	3
1.3	Magycal Scale . . . . .	4
2.1	Pictures of Service Models . . . . .	8
2.2	Types of cloud based on deployment models . . . . .	8
2.3	Layered Architecture Example . . . . .	10
2.4	Mediator Topology . . . . .	11
2.5	Broker Topology . . . . .	12
2.6	Microkernel . . . . .	13
2.7	Microservices . . . . .	13
2.8	Space Based . . . . .	15
2.9	Pattern Analysis Summary . . . . .	16
3.1	Validation of Magycal Chat API . . . . .	24
3.2	Magycal Changes . . . . .	26
3.3	Magycal Scaling Changes . . . . .	26
3.4	Validation of Microservice Chat API - Console Terminal Requests . . . . .	27
4.1	Magycal Monolith API Calls - GETs Requests . . . . .	30
4.2	Magycal Monolith API Calls - POSTs Requests . . . . .	31
4.3	Chat Microservice API Calls - GETs Requests . . . . .	31
4.4	Chat Microservice API Calls - POSTs Requests . . . . .	32
4.5	Chat Microservice API Calls - GETs Requests . . . . .	32
4.6	Chat Microservice API Calls - POSTs Requests . . . . .	33



## INTRODUCTION

*The journey of a thousand miles begins with one step. (Lao Tzu)*

The thesis is a result of the cooperation between *Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa* and the company Magycal Interactive.

This chapter introduces the context in which Magycal Interactive exists in the Portuguese software industry and motivates the problem this thesis aim to solve as a result of limitations in its server-side framework.

### 1.1 Context

Magycal Interactive is a growing company in the software industry. The company's main focus is the development of application aiming to produce significant impact in the Portuguese television sector. The company has established multiple successful partnerships to build applications such as Viva Ronaldo, Secret Story and SPORT TV Digital Hub.[44].

#### 1.1.1 Business Context

“We are revolutionising the way people connect and interact with television shows and live events.”

Magycal Interactive identified that television has evolved, and that broadcasts can and should be customized to each spectator. They envisioned the replacement of the “same-content-for-everyone” broadcast paradigm of old with new ways of watching, interacting and social sharing. The first screen, term coined for identifying the broadcasting device, can be augmented with a *second screen*. This second screen, usually an application in a mobile device, allows users to interact in real-time with broadcasts such as a soccer game or a TV show.

Viva Ronaldo [29] is an example of a second screen application. It was built around the celebrity Cristiano Ronaldo providing a unique entertainment, gaming and social following experience for his fans. It had a News Feed that integrated all of Ronaldo's social media activity and top fans' posts, a Play section where fans could enter their bets for the next match, and thematic events so fans could share photos and videos supporting Ronaldo. Viva Ronaldo featured a second screen experience during Cristiano's matches, allowing fans to answer real-time polls, trivia questions, key moments about what would happen next, and even predict that a goal was about to happen, all together with thousands of other fans.

### 1.1.2 Magycal

“Thanks to Magycal, our world-class, fully scalable platform, we are creating a new paradigm of fan engagement to take your TV Show or Live Event to the next level and generate extra revenue streams along the way. We craft and deliver extraordinary second screen and social networking gamified applications to emotionally connect properties with massive target audiences of fans and followers, powered by Magycal.”

Magycal (a portmanteau of the word magical + Generation Y, also known as the millennial generation) is the cloud-based, server-side platform used by all Magycal Interactive's developed applications. It came to be during the development of the Viva Ronaldo application (2012-2014). The idea was to group several services so that they could be reused in multiple applications. This way, each new development could be more focused on specific content and interaction. Also, it would reduce costs and produce added value for the involved parties.

Before Magycal, each application required its own set of services to be developed from ground up. Some of these services (e.g. authentication, social chats) would be duplicated from other applications. Although feasible, this model increased costs, time for delivery and chances of errors. A common set of services was identified and bundled together to create the first version of Magycal. As new applications were being developed, Magycal continued to be updated and upgraded.

Magycal is currently deployed on the cloud and subdivided into *areas* and *modules*. The areas are called Analytics, User Experience+ (UX+) and User Interaction (UI). The Analytics area contains services related to gathering data and personalizing user content for a customized experience. The UX+ area provides extra functionality to engage users and keep them connected as long as possible. The UI area is responsible for adding real-time interaction between all users. Some of the modules that are relevant for the work are depicted in the Figure 1.1. Others are omitted for confidentiality reasons.

The modules of the platform are pieced together into a single code base that provides multiple features through a monolithic architecture. Figure 1.2 depicts a simplified example of how Magycal interacts with some well known actors:



Figure 1.1: Set of Magycal Modules



**Stream** Broadcasts a transmission to Tv and Magycal actors such as a soccer match or a concert;

**Tv** Presents the Viewer with unaltered data from Stream provider;

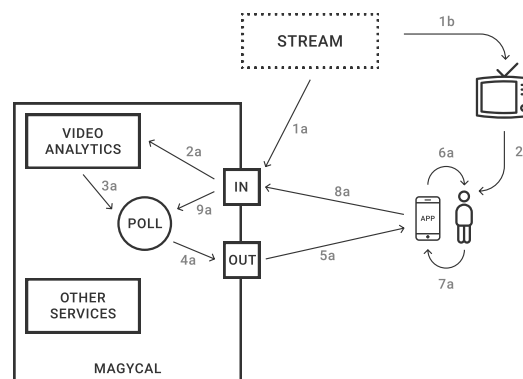
**Viewer** Interacts with both the broadcast from the Tv and content from the App;

**App** Presents enhanced content based on the Tv's current broadcast and Magycal's processing of the current broadcast;

**Magycal** Process Stream and send second screen content such as quizzes or a call to action to the App.

The Stream actor starts the flow by broadcasting some content to Magycal(1a) and TV(1b) actors. The Tv presents the content to the Viewer as it is ending one of the alternatives flow(2b). Magycal distributes the stream(2a) so it can be processed by the internal services independently to generate events(3a). Depending on the broadcast content, some services can be more important than others. For example, during a soccer match, several quizzes can request the Poll service while the Chat service is in standby mode. All events generated by Magycal are funneled to the App(4a,5a) which is the medium that allows Magycal to interact with the Viewer(6a,7a). Interactions can be both ways since Magycal also feeds on data input by Viewers(8a,9a) during a broadcast to create more events such as answers to quizzes and in-app messages.

Figure 1.2: Example of Magycal Flow

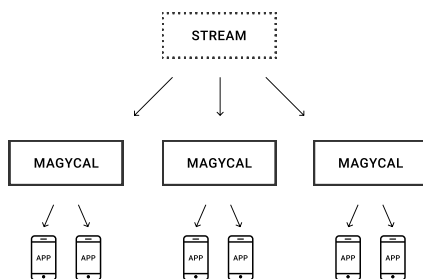


Magycal is a scalable platform with core services that can be used by new applications. Magycal has two axis for scaling: vertical and horizontal. Figure 1.3 is a bird's

eye view example of the application flow under heavy load based on the flow depicted in figure 1.2. The vertical scaling axis is not represented in the figure but is a limited resource. The represented horizontal scaling axis is used to extend even further Magycal's reach whenever the vertical scaling is not sufficient for the current application load.

Figure 1.2 shows how services are trapped under a single structure while figure 1.3 shows the costly horizontal scaling of the architecture.

Figure 1.3: Magycal Scale



While the cloud has the potential to dynamize software development, deployment and execution, Magycal's monolithic architecture acts as a bottleneck. The key limitations of Magycal are:

**Wasteful Scaling** Current scaling Magycal requires that all of its source code to be replicated. All services are copied whenever a popular service X requires more resources. As the platform grows, this can lead to waste and increased costs.

**Single Point of Failure** As services gain more and more popularity, chances for failure increase. In a monolithic architecture such as that of Magycal when an failure occurs it can disrupt the whole platform.

**Limited Modularity / Extensibility** Upgrading existing services and extending with new features can be quite challenging in a monolithic application. Components can be coupled together which makes difficult for testing and debugging. This increases the time of development, chances of errors, and therefore costs.

**Rigid Deployability** Magycal is comprised of many modules and services. Regular updates are common to most of them. Deployment of those updates requires the entire platform to be halted for a period of time generating unstable quality of service.

These issues are becoming more relevant as new partnerships are established and projects are created, relying on the Magycal platform. Mitigating these problems would have a significant positive effect in the companies' growth.

## 1.2 Accomplished Work

Magycal has undoubtedly improved the quality of products delivered by Magycal Interactive. However, as discussed in section 1.1.2, we have identified paths for improvement that need to be addressed. From the possible improvements, reducing the scaling cost of the framework was decided to be the target of the research. We claim it to be possible based on similar success cases such as Netflix, Amazon and Uber [34].

Magycal is a fully fledged platform deployed to assist Magycal Interactive's applications. New concepts and implementation should not interfere with any working deployment so we port a version of Magycal for testing purposes only. This copy of the framework had some of its components migrated to a new machine on Amazon. Among the components, we identify the long term database, the cache database, server software and the source code itself.

We have analyzed other architectural designs in chapter 2 that take more advantage from the cloud's capabilities[50] to reorganize the framework current monolithic architecture that is hindering Magycal Interactive's evolution. This new design focus on decoupling services. Independent services that distribute the single-point of failure problem creating a more resilient framework and minimizing overall instability from errors to users. Services can scale independently based on popularity, development is more efficient because there is less interference from outside services, and deployments are divided for easier and faster upgrades.

Since full development of independent services for the entire framework during this thesis was not a realistic objective given the size of the platform, instead, we focused on a proof of concept to demonstrate those described advantages. With this in mind, a chat service was created using the microservice architecture design. This chat service has the same components of Magycal so it can mimic the work flow for a more accurate comparison.

Benchmark was performed using the Vegeta tool[48] to compare both implementations. Results of the benchmark are discussed in chapters 3 and 5 to support the hypothesis stated before that it was possible to reduce cost of the Magycal framework via implementation of microservices.

## 1.3 Document Layout

This chapter introduced Magycal Interactive's Magycal framework and its opportunities for growth. Chapter 2 presents relevant technologies and concepts for this research such as the cloud and software architectures. Chapter 3 depicts a solution, an evaluation method for this proposed solution, and a comparison of results between the new implementation and the original Magycal platform. Chapter 5 summarizes the project and depicts future paths for researching.



## STATE OF THE ART

*Keep going. Everything you need will come to you at the perfect time. [35]*

This chapter presents the technology relevant for this work. It encompasses the cloud, cloud providers and software architectures. We focus the study on different architectures to find out which ones best suit Magycal.

### 2.1 The Cloud

The cloud is an abstraction for enabling universal, on-demand and convenient network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services), It can be rapidly provisioned and released with minimal management effort or service provider interaction[32], which allows companies to deploy enterprise applications that (if they have been well designed) can scale their computing resources on demand[14].

#### 2.1.1 Cloud Services

The main abstractions of cloud computing are Infrastructure as a Service (IaaS), Platform as a Services (PaaS), and Software as a Service (SaaS)m illustrated in Figure 2.1. In IaaS, a cloud provider offers infrastructure such as servers, computing resources and storage. Resources scale as necessary, companies only pay what they consume and the infrastructure cost is outsourced to a provider. In PaaS, a platform is built on top of raw components from infrastructure to provide consumers a way to tweak cloud components more easily. In SaaS, applications are provided for the use of consumers.

These abstractions can also be viewed as a layered architecture where services of a higher layer can be composed from services of the underlying layer.

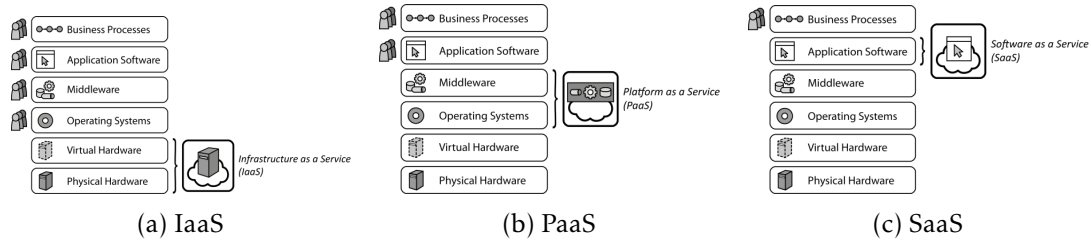


Figure 2.1: Pictures of Service Models

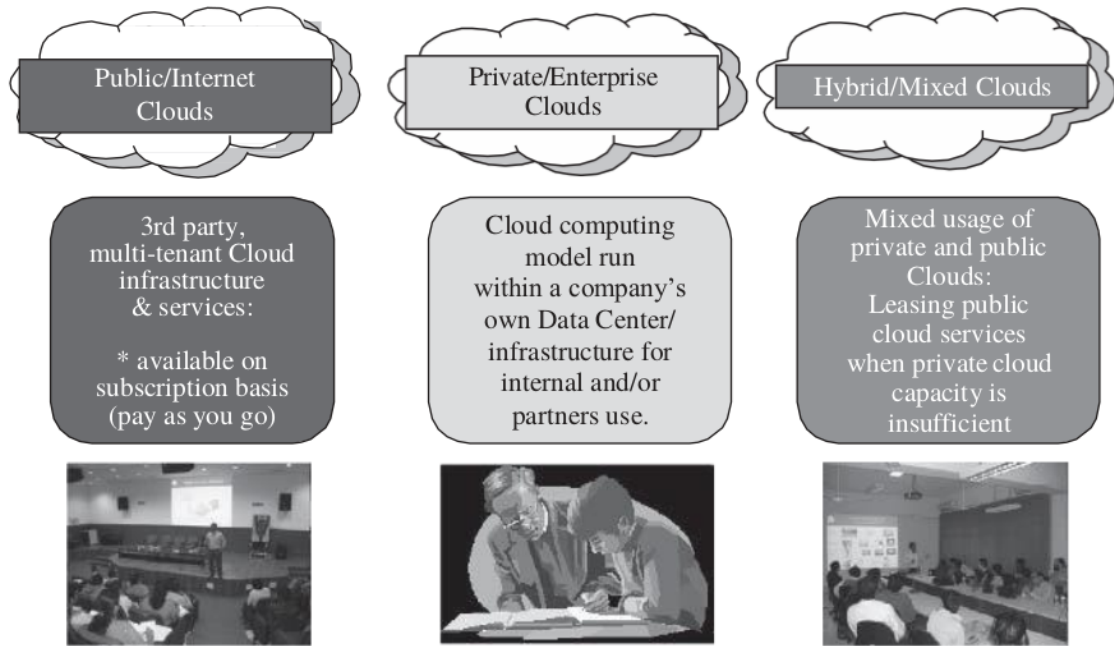


Figure 2.2: Types of cloud based on deployment models

### 2.1.2 Deployments of The Cloud

In addition to the abstractions referenced above, the cloud can be also divided among types of deployment. A summary of types of cloud based of deployment models can be viewed in Figure 2.2.

**Public Cloud Deployment.** Public clouds offer several key benefits to service providers, including no initial capital investment on infrastructure and shifting of risks to infrastructure providers. It is a cloud made available in a pay-as-you-go manner to the general public. Public clouds are location independent, reliable and highly scalable, but less secure and not customizable[49].

**Private Cloud Deployment.** Private clouds are setup and run for a particular enterprise. It has privacy and security. Also, it has limited scalability and are restricted to an area.

**Community Cloud Deployment.** Community clouds compose infrastructure shared among

different organizations with similar activities. Typical examples are universities using it for learning and research.

**Hybrid Cloud Deployment.** Hybrid clouds are a combination of public and private cloud models that tries to address the limitations of each approach. Core activities are hosted on a private cloud, while less essential services are outsourced to a public cloud.

As this research is not focused in the analysis of cloud or any of it's features we just noted a few concepts that are useful for understanding the basics of the cloud.

### 2.1.3 Where does Magycal fit?

Magycal is deployed to the public cloud AWS EC2 machine[33] as stated before. It can be classified as PaaS since it provides the middleware services for the applications developed by Magycal Interactive.

## 2.2 Software Architecture

Software Architecture (SA) appeared to tackle a real problem for software development. As complexity grew, design and representation was needed for large-scale structures of software systems [30, 43]. The normal use of SA is as a blueprint of the system during development, for maintenance and reuse. SA can also be used to analyze and validate architectural choices complementing traditional code-level analysis techniques. Model-driven techniques and architectural programming languages were already introduced to guide the design and coding process from an architectural artifact [22, 24]. In summary, SA specifications are used for many purposes [11, 12, 36]: as a documentation artifact, for analysis, and to guide the design and coding process. There is not a single correct architecture for all since each one has pros and cons that need to be considered.

Mark Richards describes in his book [42] the five architectures commonly used for developing software systems. He scores them against some concepts discussed in Section 2.2.5.1 so new architects can choose the best architecture that fits their needs.

### 2.2.1 Layered Architecture

The layered architecture is the most commonly implemented architecture because companies usually divide development into layers (presentation, business, persistence, database). An example can be observed in Figure 2.3 with the four usual number of layers. Each layer has a specific role to perform in the application flow and provides the next higher layer with services. The presentation layer is the front end and handles all user interaction. The business layer contains all operations from application logic to be performed on top of user requests. The persistence layer retrieve and provide data to the underling database layer to store.

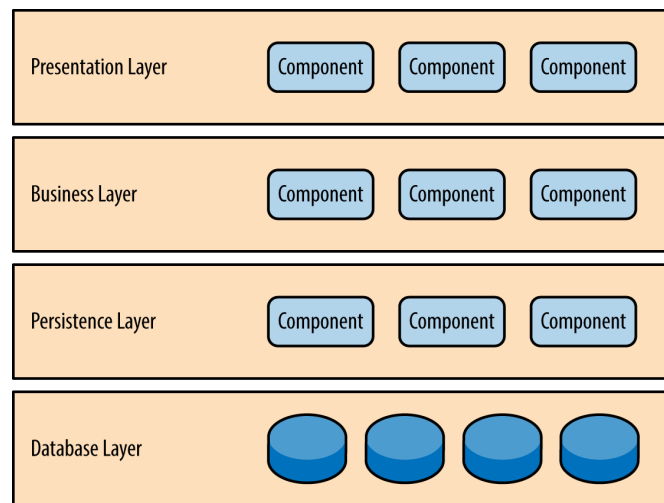


Figure 2.3: Layered Architecture Example

The application flow is a specific one with this pattern so layer isolation can be maintained. For example, a request originating from the presentation layer must first go through the business layer and then to the persistence layer before finally reaching the database layer. This leads to an architecture sinkhole anti-pattern where requests must go through all layers without performing any real operations. A workaround is introducing an open layer concept where the open layer may be skipped from the normal flow of application requests. This option increases layer coupling because changes into this open layer requires modification of all layers related to it breaking the isolation rule. This pattern usually leads to monolithic applications where one alteration implies the redeployment of the whole application.

The layered architecture pattern is a widely know pattern that reflects organization division is and not difficult to implement. Testing layers is a very easy task because other layers of the system can be mocked reducing tests dependencies. To maintain isolation, the application flow must be enforced. This leads to great inefficiency so high performance applications should not be developed under this pattern. Components are coupled within each layer which requires scheduled deployments during offline hours to minimize issues. This reflects a slow release of updates.

Magycal is mostly implemented with this layered architecture design. Our goals align with mitigating the downfalls of this pattern as described.

### 2.2.2 Event Driven Architecture

Event driven architecture is an asynchronous distributed systems pattern used for highly scalable applications. It is composed of decoupled, single-process components that receive and process events. This architecture can be subdivided into two classes based on the orchestration needs.



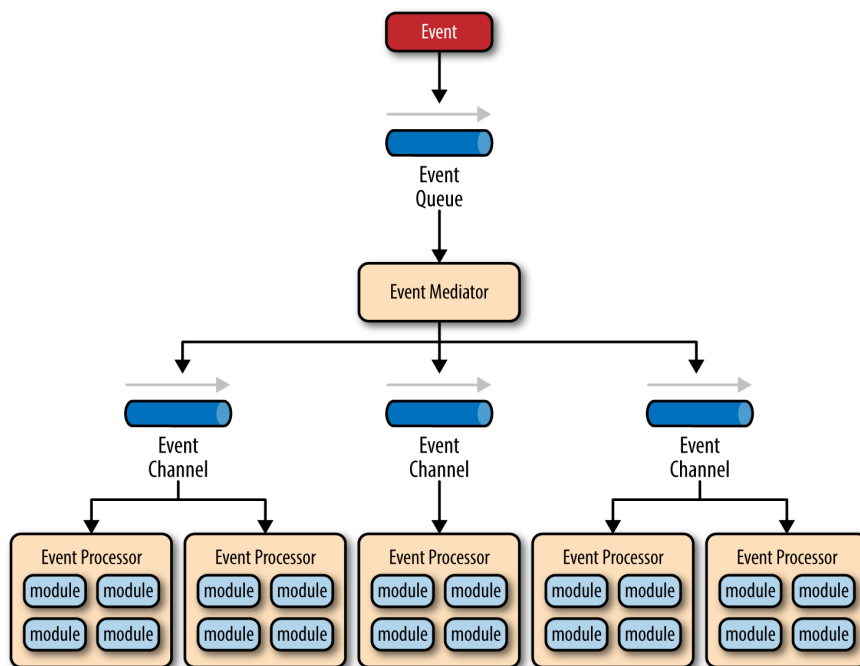


Figure 2.4: Mediator Topology

### 2.2.2.1 Mediator Topology

The mediator topology, as shown in Figure 2.4, is composed of event queues, an event mediator, event channels, and event processors. The event queue stores an event for the mediator to process asynchronously. Then, the mediator redirects to the correct channel the event without executing any business logic. Each event channel is listened by multiple event processors that are self contained, independent and decoupled to perform specific tasks. When an event arrives, a processor processes it, executes the appropriate logic and returns the result to the mediator. If processing is necessary, the mediator launches the received event again to the appropriate channel. If some steps can be performed asynchronously, the mediator inserts events into multiple event channels at the same time.

### 2.2.2.2 Broker Topology

The broker topology is simpler than the mediator topology presented before as can be observed in Figure 2.5. The components are only event channels and event processors. The event first arrives at a customer specific processor which handles some business logic and, if nothing else is required, returns the requested data to the client. If more processing is necessary, the customer processor generates an event and places it into the right event channel for the next processor to pick up. The orchestration is replaced this way by a series of chained events. Similar to the mediator topology, an event can trigger multiple processors at the same time in a parallel asynchronous way.

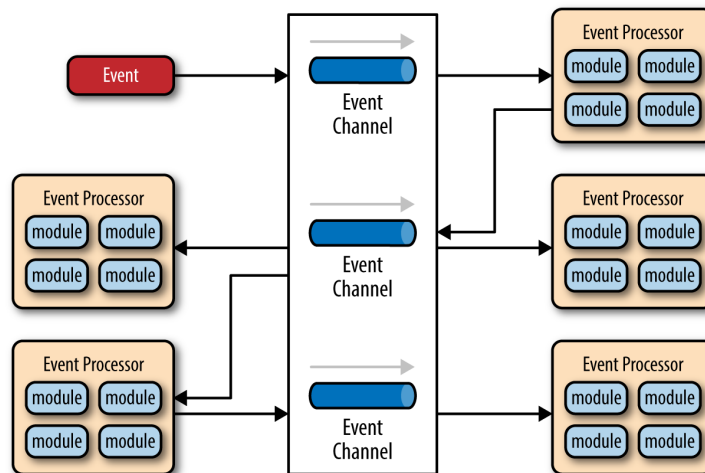


Figure 2.5: Broker Topology

The event driven architecture is complex to implement and maintain due to its asynchronous distributed nature. It is required of the developer to address various distributed systems issues on component level such as component failure and lack of responsiveness. Also, there is no concept of atomic transactions so the granularity of each process must be defined accordingly to each operation. Although unit testing is feasible with some component that generate events, asynchronous behavior difficults the process. The most difficult aspect is maintaining event-processor contracts so it is recommended to settle on a standard data format and establish a contract versioning policy from the start.

All this overhead comes with advantages: components are decoupled enough that new versions of each processor can be rapidly deployed not requiring a strict schedule. Each component can scale independently as necessary. Queuing and dequeuing can have its cost, but asynchronous behavior obtained from the design outweighs this cost and supports high performance application.

### 2.2.3 Microkernel Architecture

This architecture is also called plug-in architecture pattern because it is composed of a core system for basic functions and plug-ins to add on top of that with new features. Plug-ins register into the core system as illustrated in Figure 2.6 creating a single piece of software. Business logic is divided between all components where the core system provides more basic functions and redirect previously registered advanced functions to the correct plug-in to execute. It is possible for third parties to develop and integrate software in this pattern if a contract, adapters and versioning between the core system and plug-ins is established.

The microkernel architecture is extensible and flexible because of its plug-ins independence (loosely coupled) allowing for changes to be integrated into the system easily. Deployment can be performed at run-time because implementation normally increments

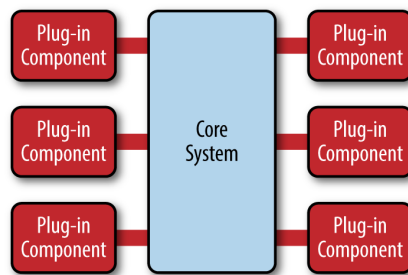


Figure 2.6: Microkernel

existing features minimizing downtime. Testing is easy because components functions can be mocked to the core system to demonstrate a prototype of new feature. High performance can be achieved if extra unused plug-ins are deactivated. The architecture was not developed with scalability in mind. It is possible to scale each plug-in, although not very efficiently, but the core system will always remain the same. Development requires many design options such as contracts at early stages when few plug-ins exist which makes hard to predict what might be needed and prepare a core system for that.

#### 2.2.4 Microservices Architecture

This pattern was created to solve problems present in monolithic applications using the layered pattern and distributed applications developed through a service-oriented architecture pattern. Creating a pipeline for development and deployment in a monolithic application can be tricky because all modules are coupled together. Microservices decouple some of the modules to create service components with different granularities. Distributed applications can be daunting and over complex so microservices remove orchestration and simplify connectivity and access to services. The result is depicted in Figure 2.7. There are three topologies for microservices: API REST-based, REST-based and centralized messaging.

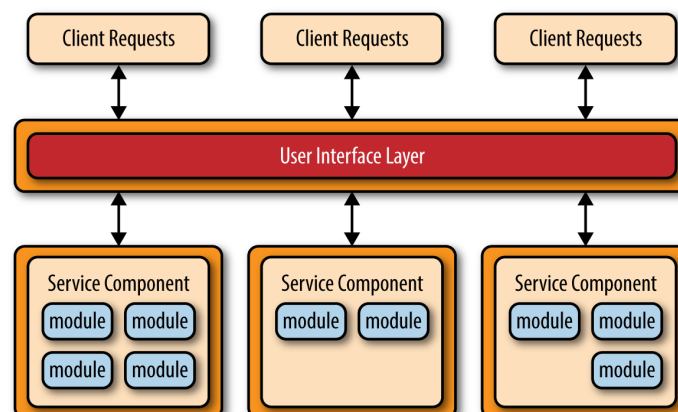


Figure 2.7: Microservices

#### 2.2.4.1 API REST-based Topology

The API REST-based topology is named after its REST-based interface implemented with separately deployed web-based API layer for client requests. Services components are very fine-grained (one or two modules) for specific tasks, independent from the rest of services.

#### 2.2.4.2 REST-based Topology

The REST-based topology is slightly different from the API REST-based topology. Instead of a web-based API, this topology has an interface layer that returns rendered layouts. Also, components tend to be larger, more coarse-grained, and represent a small portion of the overall business application rather than fine-grained, single-action services.

#### 2.2.4.3 Centralized Messaging Topology

The centralized messaging topology is similar to REST-based topology because it has a layer for returning rendered requests to clients. The difference lies in using a lightweight centralized message broker instead REST-based calls to access services. This broker does not perform any orchestration, transformation or processing. The design allows for advanced queuing mechanisms, asynchronous messaging, monitoring, error handling, and better overall load balancing and scalability. The single point of failure and architectural bottleneck issues usually associated with a centralized broker are addressed through broker clustering and broker federation (splitting a single broker instance into multiple broker instances to divide the message throughput load based on functional areas of the system).

The biggest challenge presented by this pattern is defining the correct granularity of components. If the granularity is too coarse-grained, the pattern does not bring advantages since it becomes too similar to a monolith, on the other hand, with an overly fine-grained leads to orchestration requirements that turn microservices pattern into a soa pattern. If done correctly, it solves many of the common issues found in both monolithic applications as well as service-oriented architectures and produces a more robust application, provides better scalability, and can more easily support continuous integration and delivery [16].

The microservices architecture have highly independent decoupled services which allows hot deployments. Problems that occur from hot deployment are usually small and localized not interfering with the overall experience. Testing can be performed easily because there are few services dependency. Development can be sped up by dividing the application into domains and assigning each one to a different team. However, high performance applications are not suited for this distributed pattern because of the communication overhead.

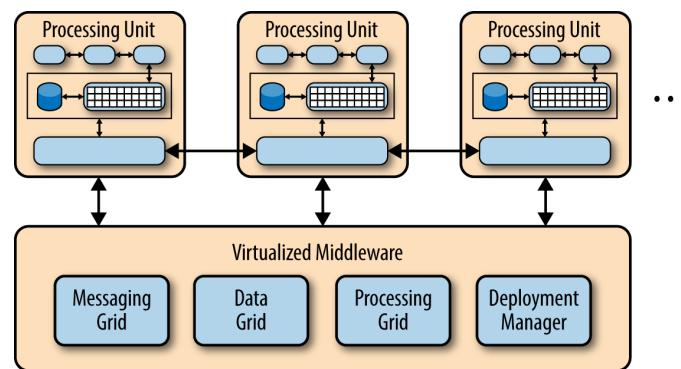


Figure 2.8: Space Based

### 2.2.5 Space-based Architecture

The space-based architecture, also referred as cloud architecture pattern, was developed to address and solve scalability and concurrency issues. This is accomplished by replacing a central database restriction for replicated in-memory data grids. Each grid is located inside a processing unit that can scale endlessly. These are connected by the virtualized middleware as shown in Figure 2.8.

Processing units usually contain application modules, the already mentioned in-memory data grid, an optional asynchronous persistent store for failover, and a replication engine used by the virtualized middleware to replicate data changes made by one processing unit to other active processing units.

The middleware is responsible for handling user interaction, orchestration, synchronization and scalability. It has a maximum of four components: a message grid to allocate incoming request into units, a data grid (most important piece of the pattern) responsible for duplication in-memory data between active processing units in microseconds, process grid that may or may not exist depending on application division, and deployment manager that keeps track of load conditions to auto scale the number of processing units.

The space-based architecture is a complex and expensive pattern not recommended for large scale relational databases with large amount of data. Components do not have to reside on cloud environments, but they were built with this in mind and have implementations that easily support cloud deployment. It is highly scalable because there is no central database to bottleneck and agile because each processing unit can be brought and down quickly. It is not well suited for unit testing or benchmarking because replicating high loads can be expensive. Development using in-memory and caching mechanisms is complex most likely because of the lack of familiarity with required tools for that purpose and the special care needed so new code does not impact performance and scalability.

#### 2.2.5.1 Comparison

Mark Richards defines in his book [42] the vectors for comparing those described architectures.

	Layered	Event-driven	Microkernel	Microservices	Space-based
Overall Agility	↓	↑	↑	↑	↑
Deployment	↓	↑	↑	↑	↑
Testability	↑	↓	↑	↑	↓
Performance	↓	↑	↑	↓	↑
Scalability	↓	↑	↓	↑	↑
Development	↑	↓	↓	↑	↓

Figure 2.9: Pattern Analysis Summary

**Overall Agility** is how a pattern supports rapid modifications of the environment.

**Deployment** is how easy it is for any changes to be incorporated into the running application.

**Testability** is how easy it is to build and run tests for that architecture.

**Performance** is how the pattern supports high performance applications.

**Scalability** is how does a component of the architecture scale, if possible, and how expensive that is.

**Development** is how complex is it to maintain, debug and build new features for that architecture.

We have graded each architecture with these vectors according to the previous discussion, but here we present a side by side comparison depicted in the Figure 2.9. It is easy to observe from the figure that Magycal benefits most from implementing the microservice architecture. The layered architecture is excluded because we are trying to move away from it. The event driven architecture is the extreme opposite of what Magycal is right now and it would mean taking Magycal all apart and reconstructing from ground up which we do not think it is possible because of some coupled modules. The microkernel architecture has limited potential to scale up so it must be discarded. The space-based architecture can not be adopted because we have a large relational database that would be very costly to maintain using this pattern. The microservices architecture pattern is a compromise between layered and event-driven architectures, a common ground that could work for Magycal. The evaluation above also points into that direction. Observing the microservices column, our objectives defined in the introduction (better development/deployment and scalability) is possible. In fact, the only negative side of the architecture

does not interfere with Magycal at all because we are not building high performance applications.

## 2.3 Microservices

The microservice architecture has been roughly described in the previous section, but here is presented some further comparison to the monolith architecture. Below are the advantages and constraints the microservice architecture has and how it can affect Magycal. Furthermore, there is also some remarks about the migration process required from the monolith architecture into the microservice architecture.

### 2.3.1 Advantages over Monolith

Applications with a monolithic structure need to be scaled horizontally and are limited to the implementation language used initially [17], resources are wasted unnecessarily due to the overall replication of the application and any complexity in the existing codebase is also replicated. On the other hand, microservices can be individually scaled up and down within their architecture [45] and a further point that underlines the technical independence of microservices is the fact that the programming language to be used can also be selected for each service in accordance to individual requirements [9].

The lack of agility of monolithic architectures can potentially slow down the entire further development of existing business models or the introduction of new ones. One cause is the ability to implement changed or new requirements, since these activities are increasingly complex to manage and development capacities are therefore minimized by the excessive time for implementation required [26].

In monolithic structures, the work of various teams on the same codebase results in large dependencies [17]. Conversely, microservices are characterized by the possibility of clearly defining responsibilities with regard to the entire software development cycle, although microservices may lead to more communication overhead if multiple services use a specific library or call a generic service.

Due to the strong interdependencies, the failure of part of a monolithic application can lead to a cascade and thus to the total failure of a system. In microservice architectures, targeted safety mechanisms can be used to avoid such complete failures [26].

### 2.3.2 Drawbacks of the Architecture

Microservices also present drawbacks that must be accounted for. Microsoft pointed out some of them [18].

Distributed application adds complexity for developers when they are designing and building the services. Developers must implement inter-service communication

which adds complexity for testing and exception handling. It also adds latency to the system.

Deployment complexity from dozens of microservices that need high scalability. An orchestrator or scheduler can mitigate that complexity, otherwise that additional complexity can require far more development efforts than the business application itself.

Atomic transactions between multiple microservices usually are not possible. Eventual consistency between multiple microservices must be embraced.

Increased global resource needs when you replace a monolithic application with a microservices approach. The amount of initial global resources needed by the new microservice-based application will be larger than the infrastructure needs of the original monolithic application. This approach is because the higher degree of granularity and distributed services requires more global resources. However, given the low cost of resources in general and the benefit of being able to scale out certain areas of the application compared to long-term costs when evolving monolithic applications, the increased use of resources is usually a good tradeoff for large, long-term applications.

Partitioning the microservices is another challenge because it is necessary to decide how to partition an end-to-end application into multiple microservices. One approach is to identify areas of the application that are decoupled from the other areas and that have a low number of hard dependencies. In many cases, this approach is aligned to partitioning services by use case.

### 2.3.3 From Monolith into Microservice

In the past, replacing a monolithic application was often not considered due to the effort involved and the technical capabilities at hand [13, 21]. The introduction of microservices changed this perspective, allowing monolith application to be tackled in many little steps by splitting up all its functions and complexities into microservices with fast results [20, 27, 47]. Agile methods and new company organization were developed to reflect this change in architectural design. Utilizing the advantage of being individually deployable, companies can meet the requirements of speed and results in the Digital Transformation with Microservices easier [13, 38].

However, many challenges are faced when moving from a monolith architecture into a microservice architecture. In a monolithic application, for example, the graphical user interface and the business logic are typically interdependent, something which must be considered while designing and implementing the microservice architecture. A common approach to deal with this is refactoring supported by tests [13, 26, 46].



Migration from monolith applications into microservices can either be a one step migration, also called big bang migration, where all components from the monolith application are replaced by microservices equivalents in a single step or it can be a multiple step migration where microservices components gradually replace monolith business logic [28].

When done right, this approach can yield a resilient, scalable and maintainable distributed application while eliminating many of the disadvantages of monolith applications. Due to the polyglotism characteristic of microservices, experienced and long-established software developers in an company can still be integrated in the shift towards a modern architecture [10, 27], without the need to learn new programming languages.



## SOLUTION

*Everything you can imagine is real. (Pablo Picasso)*

In this chapter we describe the porting of Magycal platform into a testing environment on Amazon [1], we introduce and validate the chat service used a case study for the project, and detail the implementation of a chat service also uploaded to Amazon.

### 3.1 Context

It is important to reiterate that the main goal of this work is to reduce costs of the Magycal framework. More specifically, a microservice architecture design is attempted to reduce scaling costs so each individual service can scale independently, based on client demand.

The major drawback from the microservice pattern is an increased complexity of development because of its distributed nature. The case study selected for testing is an independent service that does not suffer from this communication overhead, but in future tests for different services the communication overhead must be accounted for.

All implementation developed during this research considered only free / trial tools provided by Amazon. Choosing Amazon tools over other options was a no-brainer because Magycal is currently deployed on Amazon servers. This decision reduced the learning curve since the working tools were familiar. Also, regarding the necessary technologies, other providers should have similar tools available with different names and management only.

### 3.2 Magycal

We explain in this section why the migration process was necessary, how the migration process was executed and a validation of the migration process. Section 3.2.1 describes

the migration process for Magycal's macro components. Section 3.2.2 describes the chat service of the migrated Magycal framework. A few challenges hindered some progress of the process and they are explained in Section 3.2.3.

### 3.2.1 Amazon Resources

A version of the current platform was migrated to Amazon servers so changes and testing would not interfere with live projects. Also, so testing could be performed under a controlled environment. This migration process had three major steps: resource allocation on Amazon, configuration of components, and validation of each component's configuration. Sections 3.2.1.1 and 3.2.1.2 depicts these steps for each necessary resource on Amazon.

#### 3.2.1.1 Database

The database set up for testing was the free tier database tool from Amazon [7]. Magycal has a centralized relational database to support long term storage so only one database instance was required.

Configuration of the database entailed creating a user with read and write permissions, and uploading staging data to emulate a working environment.

Validation was performed by manually opening a connection to the database and placing queries to produce results.

#### 3.2.1.2 EC2

Magycal was uploaded to an EC2 machine on Amazon[3]. This machine runs a Ubuntu 18.04 LTS version with the apt package manager used to install all the requirements of the framework described below.

**Nginx** is the server engine responsible for controlling network traffic[37]. Nginx configuration files from the working platform were imported and reused for setting up the new environment. Adjustments to fields such as port and server name configuration were necessary.

**Php** is the server side language for processing requests[39]. Php dependencies of Magycal are managed with composer[15]. Default configurations were used for php and its dependencies.

**Redis**[40] is the memory database used as cache. Default configuration files were adjusted to work with the Magycal.

Validation of these components was necessary to ensure Magycal was installed properly. The Php and Nginx components were easily validated by a default strategy used when configuring such environments. This strategy required the creation of an information php file on the root of the project and then to access it via a browser. Php information

retrieved from the server indicated that both Php and Nginx were properly configured. In addition, this file also described Php extra libraries installed for further validation. The Redis component required to send an API call to a service of Magycal such as the chat described in Section 3.2.2. Then, it was necessary to connect to the Redis component, retrieve saved information and validate.

### 3.2.2 Chat service

After porting Magycal to Amazon, an array of services become available. Some of these services are ready for use as soon as the Magycal is migrated and configured as described in Section 3.2.1 because of their simple logic and/or lack of further configuration. In fact, for the chat service, all that was required was an existing community that already exists thanks to all staging data uploaded to the database from Section 3.2.1.1.

All interaction to the chat service is performed via API calls and, since the chat service is the case of study for the project, its endpoints are exposed below for validation and benchmarking.

**GET /message/chat/id** requests all messages from a chat identified by *id*.

**POST /message/chat/id** writes a message into a chat identified by *id* after having an authenticated user.

The chat service has messages cached with Redis. This feature was used for validation of the Redis component during migration explained in Section 3.2.1.2. This feature was also considered for the microservice chat version described in Section 3.3 so testing could have less interference.

From this description, one might assume that the chat service is independent from other services of Magycal. Nonetheless, the chat service shares resources with all other configured services even though they are not actively being called and that is evident in Chapter 4. Additionally, during this work, simultaneous service execution was not simulated because of the limited scaling resources used on Amazon.

#### 3.2.2.1 API Validation

After Magycal was ported to Amazon, an API validation was required before performing any benchmark.

Figure 3.1a shows two test calls to Magycal's *POST* and *GET* chat API described in 3.2.2. The *POST* request writes a message to a community that can be retrieved via the *GET* endpoint. Figure 3.1b shows the status of the database before the *POST* is executed. The last message of the chat had the "*hello, everyone! I'm here!*" body. Once the *POST* call is successful and a *GET* is performed, it can be verified that a new message with the body "*I'm still here.*" was placed on top of the previous message from figure 3.1c. *GET* data

is retrieved in *JSON*(figure 3.1a) and the visualized with an online *JSON* parser tool[25] (figures 3.1b and 3.1c).

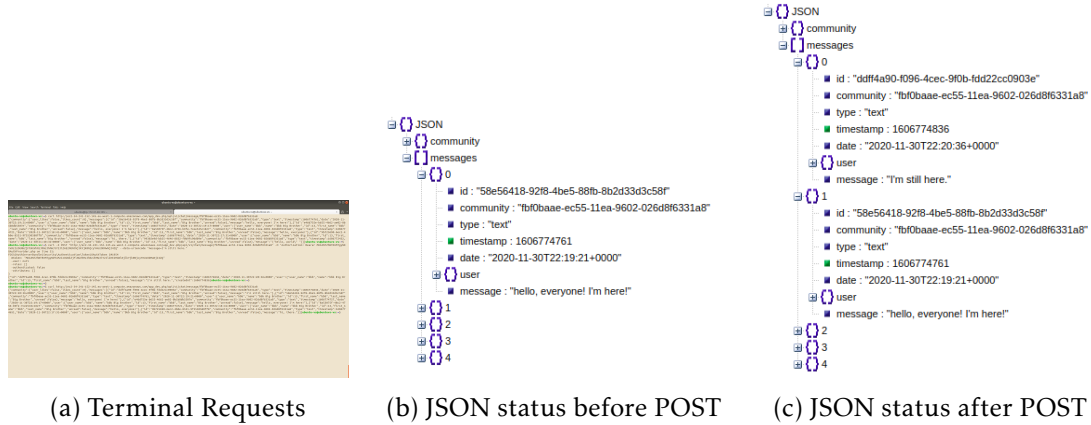


Figure 3.1: Validation of Magycal Chat API

### 3.2.3 Challenges and Remarks

We now summarize the problems and challenges faced during the porting of Magycal to Amazon. Section 3.2.3.1 describes difficulties faced during porting and Section 3.2.3.2 describes an alternative postponed for future development.

#### 3.2.3.1 Porting Challenges

Problems and challenges were faced during porting and configuration of Magycal. Some of them were from our own making (see Section 3.2.3.2), but most of them had as its root cause obfuscated and coupled code and configurations that required large amounts of time to debug and set properly. Although the case of study was the chat service, requirements from other services had to be installed and configured as well for Magycal to work.

This challenge is one of the reasons this project is in place. Removing obfuscated and coupled code could lead to better development and maintenance of Magycal.

#### 3.2.3.2 Docker Attempt

It was planned to port Magycal using docker containers [19]. Docker images would have all requisites installed and configured for future replication and testing. Also, it could be used for scaling with Amazon as well. Unfortunately, one component of the framework could not communicate over the intra-network created by the Docker system and an alternative option had to be considered. The solution to the problem at hand was disclosed at [41], but the project had already moved to the testing phase by that time.

### 3.3 Microservice Chat

The chat service from Magycal was chosen as a case of study for the project meaning that it was the selected service to be extracted from the framework and implemented into the microservice architecture for testing. Note that the chat service from Magycal is still there, but it will not be called upon during testing.

The new chat service is a simple service that is self contained and is only available to others via it's own API. The service only performs a few modular operations, truly isolating itself from any external dependencies. The chat microservice was created with Go Language [23] and the application mimicked the API from Magycal exposed below.

**GET /chatroom** requests all messages from chatroom.

**POST /message** writes a message into a chatroom.

Since tests performed in Chapter 4 only need one chat room, the API was simplified so it only supports one chat room. This approach was picked over the real code from Magycal because it would be necessary to adapt existing monolith coupled code logic into microservice independent detached code logic. Although an interesting exercise that will be performed in the future for production migration, it is estimated that additional logic would not be much different from the real implementation. The chat service only handles chat related tasks so it is assumed that any requests to the service was already correctly authenticated(e.g. another microservice). Similar to Magycal, a redis server was configured for caching the messages in memory.

No persistent storage was configured for the service since asynchronous dumps to a database could be performed without impacting testing in Chapter 4. Persistence is certainly an important topic to be handled before having the microservice working in a production environment and it will be further investigated in the future considering that another common bottleneck of monolith applications is a centralized database. After decoupling the service logic into a microservice, it is only expected to detach the database for improved performance. The most simple solution would be to create a single database that all chat microservices have access to. More complex solutions could also be devised for better performance such as a master slave system [31].

#### 3.3.1 Comparison of Architectures

Before moving ahead with the API validation for the chat component, we compare in figure 3.2 the architecture design shift of introducing a new microservice into the system. Figure 3.2a is the original design of Magycal and figure 3.2b is the new design introduced with the new chat microservice.

From figure 3.2b is possible to notice that the chat service was moved from the original Magycal architecture to a new hybrid approach that presents some services in the monolith architecture and a new chat service with the microservice architecture.

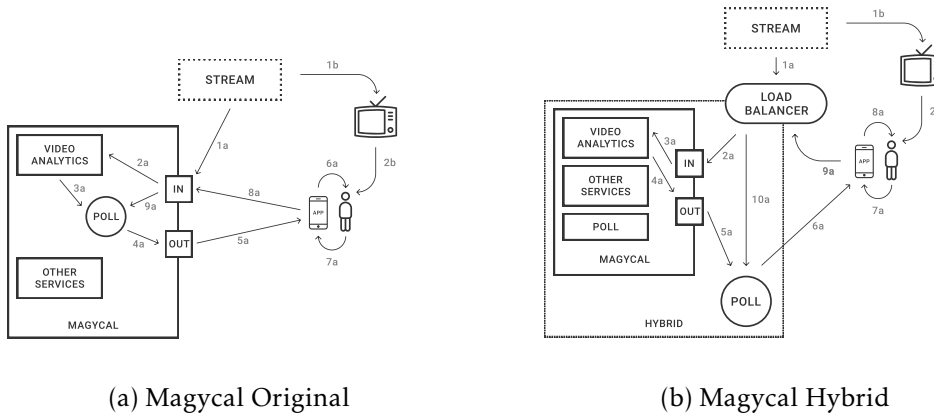


Figure 3.2: Magycal Changes

Since the new approach is meant to optimize scaling components, we compare the scaling of both approaches in figure 3.3. Figure 3.3a shows the original scaling of Magycal and figure 3.3b shows the scaling of the hybrid approach. The chat service now scales independently of the remaining services. The end goal, not aimed in the project, is to reduce as much as possible the Magycal box and replace it with new microservices (Figure 3.2b).

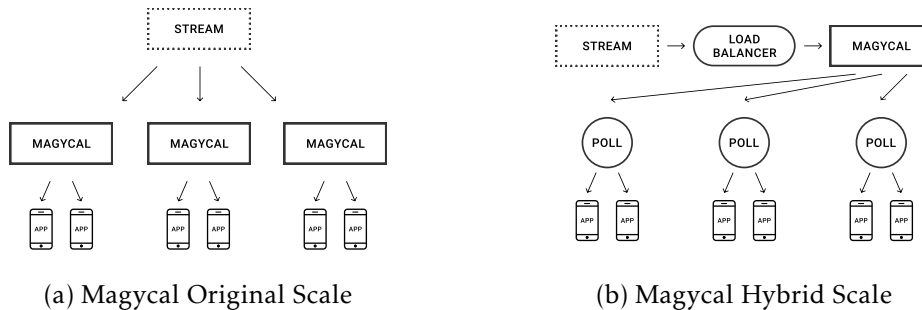


Figure 3.3: Magycal Scaling Changes

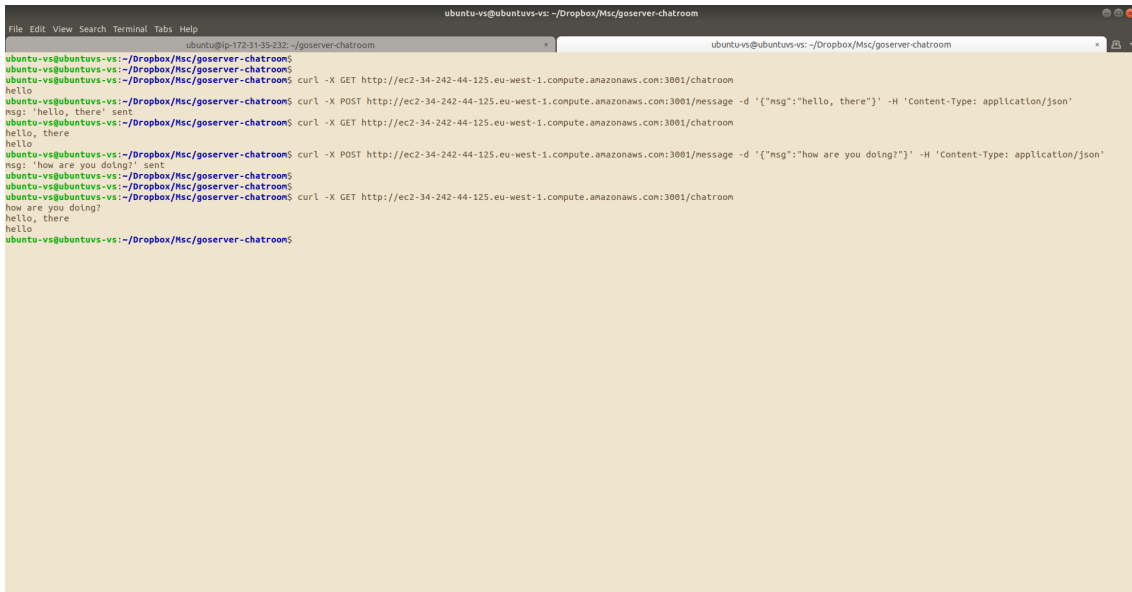
A clear drawback from the microservice architecture is duplication of some base resources. The hybrid example from figure 3.2b requires at least one machine for the original Magycal and one machine for the chat microservice. Nonetheless, it is expected that this extra initial cost will be mitigated in the future by transferring traffic from the original Magycal platform which incurs substantially more expensive scaling cost to other microservices with significantly reduced scaling cost.

### 3.3.2 API Validation

After creating the new chat service, it is possible to validate it via API calls. Figure 3.4 shows *POST* and *GET* calls to the microservice server API. The output can be visualized without any parsing due to the simplification of the service.



Validation begins with a *GET* that retrieves the "hello" message previously sent to chatroom. Then, the messages "hello, there" and "how are you doing?" are sent to emulate a replies. It can be verified from the third *GET* showed in the console.



```

ubuntu-vs@ubuntuvs-vs: ~/Dropbox/Msc/goserver-chatroom
ubuntu-vs@ubuntuvs-vs:~/Dropbox/Msc/goserver-chatroom$ curl -X GET http://ec2-34-242-44-125.eu-west-1.compute.amazonaws.com:3001/chatroom
hello
ubuntu-vs@ubuntuvs-vs:~/Dropbox/Msc/goserver-chatroom$ curl -X POST http://ec2-34-242-44-125.eu-west-1.compute.amazonaws.com:3001/message -d '{"msg":"hello, there"}' -H 'Content-Type: application/json'
msg: 'hello, there' sent
ubuntu-vs@ubuntuvs-vs:~/Dropbox/Msc/goserver-chatroom$ curl -X GET http://ec2-34-242-44-125.eu-west-1.compute.amazonaws.com:3001/chatroom
hello, there
ubuntu-vs@ubuntuvs-vs:~/Dropbox/Msc/goserver-chatroom$ curl -X POST http://ec2-34-242-44-125.eu-west-1.compute.amazonaws.com:3001/message -d '{"msg":"how are you doing?'}' -H 'Content-Type: application/json'
msg: 'how are you doing?' sent
ubuntu-vs@ubuntuvs-vs:~/Dropbox/Msc/goserver-chatroom$ curl -X GET http://ec2-34-242-44-125.eu-west-1.compute.amazonaws.com:3001/chatroom
how are you doing?
hello, there
ubuntu-vs@ubuntuvs-vs:~/Dropbox/Msc/goserver-chatroom$

```

Figure 3.4: Validation of Microservice Chat API - Console Terminal Requests

### 3.3.3 Load Balancer

A load balancer from Amazon [6] was considered to resemble a real application flow illustrated in Figure 3.2b. But, prior to configuration, it was discovered that Amazon charges for each request processed instead of the usual free / trial plans used for other components of the project. The alternative adopted was to place all calls to API that directed to the each respective machine. Since all parties are handicapped by the same problem, it does not interfere with the results presented in Chapter 4.

The load balancer will be used in the future to configure production machines where a hybrid version of the platform will be used. That will enable applications to keep working despite of the changes performed into the server.



## BENCHMARK

*Be tolerant with others and strict with yourself. (Marcus Aurelius)*

This chapter describes the testing set up, the testing methodology adopted for comparing the chat service from Magycal migrated into Amazon against the new chat developed under a microservice architecture described in Chapter 3 and a comparison of the tests results obtained from the benchmark.

### 4.1 Set up

As mentioned before, only free / trial technology from Amazon was used during the thesis which limited the amount and type of machines available for testing. It was possible to set up two free tier EC2 machines from Amazon[4, 5] within given limitations, one for Magycal and the other for the microservice chat service.

Magycal also required an external database set up with the free RDS database from Amazon[7, 8]. The only available database for the baseline is inferior to the current deployment of Magycal ,but it should not interfere with the benchmarking process since the traffic to Magycal is also reduced to testing calls.

### 4.2 Methodology

Testing was performed by placing API calls to Magycal chat and the microservice chat as described in Chapter 3. This is an unrealistic traffic scenario for Magycal, but the alternative would require a fully operational platform not be possible in the scope of this project because of the infrastructure cost.

Tests were performed in sets of runs. Each run had a 30 second length, a rate of requests per second (1, 5, 10) and type of requests performed (*GET* and/or *POST*). The

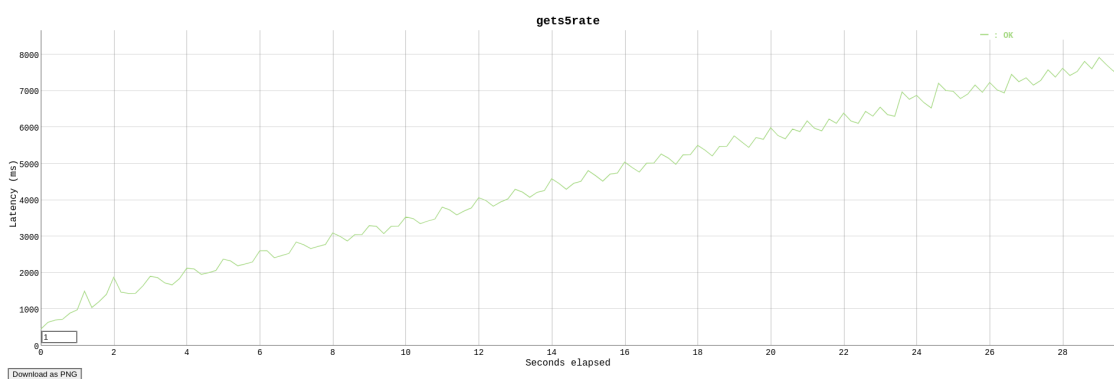


Figure 4.1: Magycal Monolith API Calls - GETs Requests

length of a run and rate per second values were gathered from a few calibration tests where we tested the limits of the environment. From each set of tests it was selected the run with the lowest mean response time for comparison in an attempt to remove communication spikes and other possible problems of distributed system nature. Tests had a warm up where database and memory were filled prior to benchmark calls. All benchmark results were extracted from reports of the Vegeta HTTP load testing tool[48].

### 4.3 Comparison

This section compares results obtained from benchmarking Magycal chat service and the microservice chat service. Both services are the only ones actively called during tests.

Magycal chat service is a downgrade from the production case because it does not have the same scaling capabilities. To offset this, none of the live traffic is present which means that most of the EC2 machine and database is dedicated to the chat service.

The microservice chat is as close to production code as possible. Magycal code was not used for implementation because it is coupled with other services such as database synchronization. In its place, we developed a simplified approach where multiple chat rooms and persistence should be implemented in the future. Persistence could be performed asynchronously during reduced traffic of the service. Multiple chat rooms logic would require a slightly different Redis data structure (a hash map instead of a list) and an upgrade of the API. Therefore, these simplifications do not have much impact in testing.

The results obtained from the benchmark tests are detailed in 4.3.1. With such results, it was possible to arrive at some conclusions in 4.3.2.

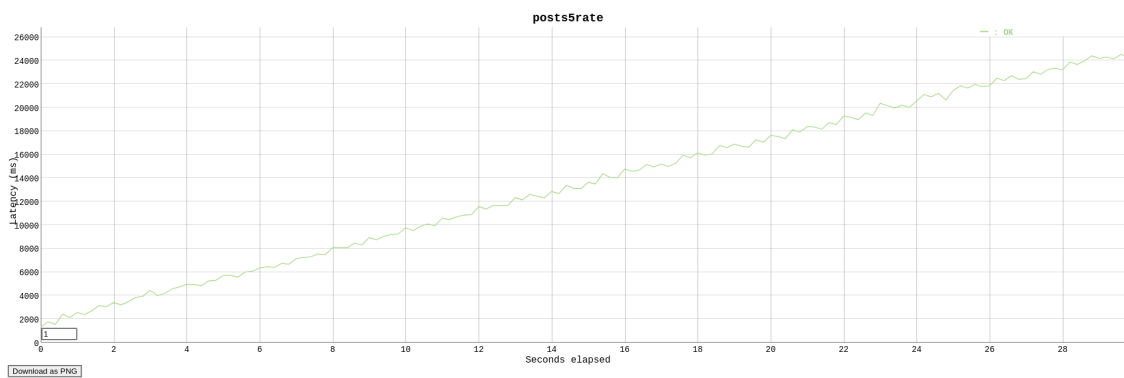


Figure 4.2: Magycal Monolith API Calls - POSTs Requests

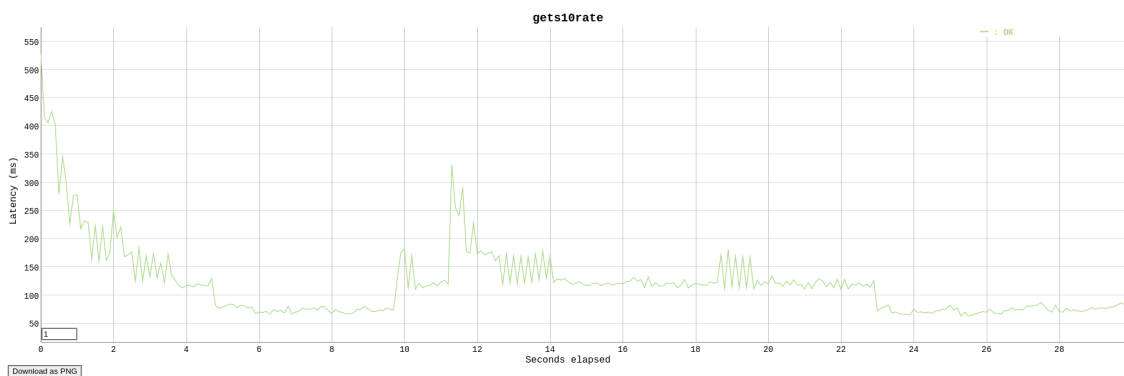


Figure 4.3: Chat Microservice API Calls - GETs Requests

### 4.3.1 Results

Figure 4.1 shows *GET* calls invoked to Magycal monolith. Five calls were placed per second during a thirty second length period. It can be observed that the latency increases over time. The average time of response is 4.37 seconds and the maximum time of response is 7.95 seconds.

Figure 4.2 shows *POST* calls invoked to Magycal monolith. Five calls were placed per second during a thirty second length period. It can be observed that the latency increases over time. The average time of response is 13.30 seconds and the maximum time of response is 24.48 seconds.

Figure 4.3 shows *GET* calls invoked to chat microservice. Ten calls were placed per second during a thirty second length period. It can be observed that the latency is constant over time. The average time of response is 0.12 seconds and the maximum time of

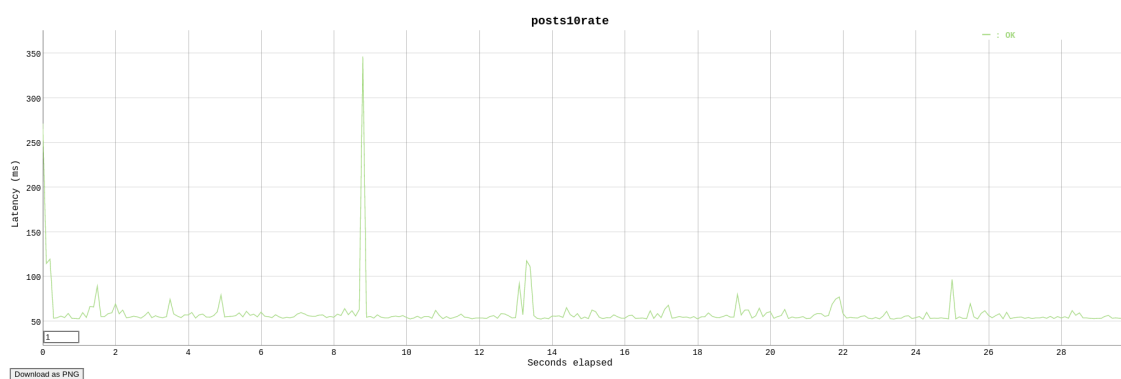


Figure 4.4: Chat Microservice API Calls - POSTs Requests

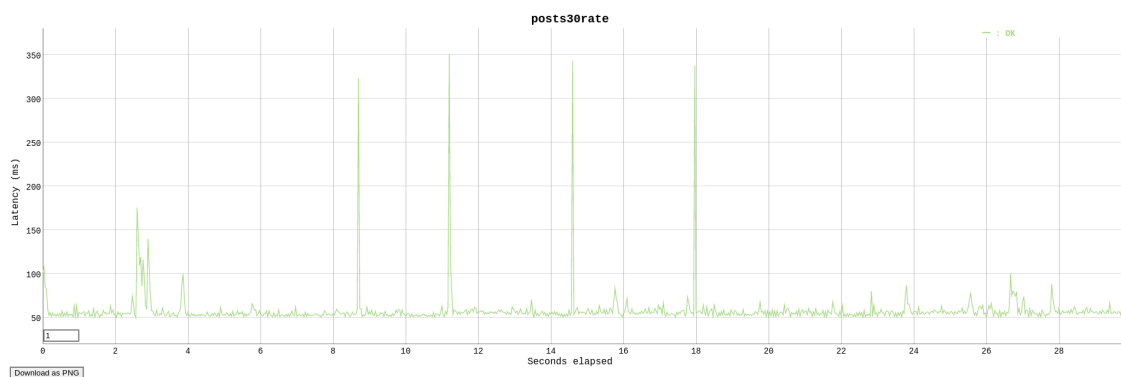


Figure 4.5: Chat Microservice API Calls - GETs Requests

response is 0.53 seconds.

Figure 4.4 shows *POST* calls invoked to chat microservice. Ten calls were placed per second during a thirty second length period. It can be observed that the latency is constant over time. The average time of response is 0.06 seconds and the maximum time of response is 0.38 seconds.

Figure 4.5 shows *GET* calls invoked to the microservice chat. Thirty calls were placed per second during a thirty second length period. It can be observed that the latency is constant over time. The average time of response is 0.31 seconds and the maximum time of response is 3.84 seconds.

Figure 4.6 shows *POST* calls invoked to chat microservice. Ten calls were placed per second during a thirty second length period. It can be observed that the latency is constant over time. The average time of response is 0.06 seconds and the maximum time

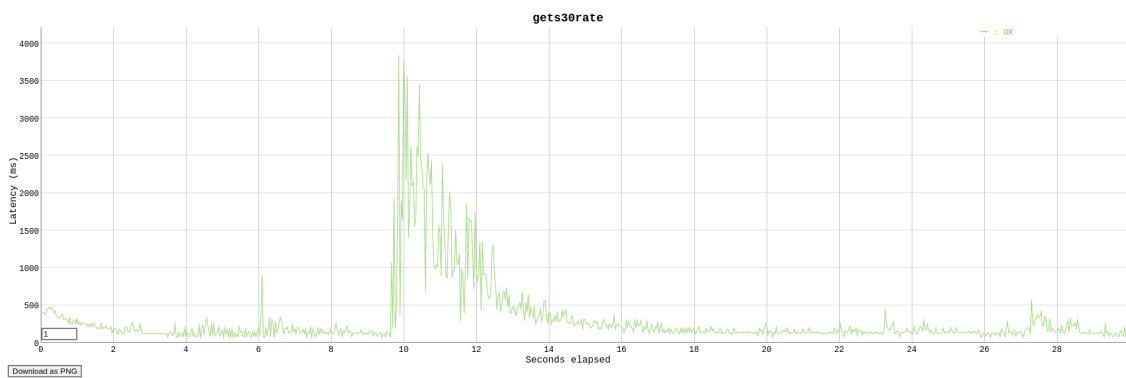


Figure 4.6: Chat Microservice API Calls - POSTs Requests

of response is 0.35 seconds.

### 4.3.2 Conclusions

Figures 4.1 and 4.2 show that Magycal consumed increasingly more resources under frequent stress. Tests performed in those charts considered 5 requests per second. If this number is increased to 10 requests per second, Magycal return timeouts errors with the configuration used for this project.

On the other hand, 4.3 and 4.4 show that chat microservice is able to handle 10 request per second, the same amount that timed out Magycal, graciously. In fact, the chat microservice was able to respond up to 30 requests per second before exhausting the same configuration used for Magycal. This can be observed in figures 4.5 and 4.6.

It is possible to conclude that it is advantageous to move the chat from Magycal monolith into a microservice architectural design from the data and comparison above. The chat service would become more reliable because it could handle more requests at the same time and it would also become more cost effective because it could handle those new requests with a lower resources.





C H A P T E R



# 5

## CONCLUSION

*The greatest glory in living lies not in never falling, but in rising every time we fall. (Nelson Mandela)*

In this chapter we summarize the work accomplished, how Magycal Interactive could implement it, and enumerates new paths for research.

### 5.1 Summary

Magycal Interactive is a growing company in the software industry and Magycal is the cloud-based, server-side platform used by all Magycal Interactive's developed applications. It has been identified in Chapter 1 that Magycal can be upgraded for reducing current costs of scaling, development and maintenance.

This work depicts the porting of Magycal into a testing environment on Amazon, the creation of a chat service in the microservice architecture design and a comparison between those deployments.

As demonstrated in Chapter 4, the chat microservice implementation can outperform a ported Magycal chat service based on production code. The microservice could handle more requests before scaling therefore reducing infrastructure costs. It is also cheaper to scale the microservice chat than Magycal because a single service has far less resource requirements than a monolith comprised of multiple services.

A clear downside of dividing services into multiples machines is the higher initial base cost compared to a single machine for a monolith. Nonetheless, considering the size of Magycal and demands of Magycal Interactive's current applications, the downside presented is expected to be outweighed by all the scaling already required by the monolith implementation.

Magycal current have the following limitations presented in Chapter 1: *Wasteful Scaling*, *Single Point of Failure*, *Limited Modularity / Extensibility*, and *Rigid Deployability*. Moving from the monolith architecture into the microservice architecture would target all of those hindrances. *Wasteful Scaling* would be reduced now that each service can scale based on popularity. *Single Point of Failure* would be distributed along the multiple microservices. Since code is divided, *Limited Modularity / Extensibility* and *Rigid Deployability* are not an issue anymore.

This project brings to Magycal Interactive's consideration a new approach to upgrade its framework even further and make Magycal more robust and reliable.

## 5.2 Upgrading Magycal

With this work, Magycal Interactive has a starting point for shifting the Magycal monolith architecture into a microservice architecture. The microservice architecture is not free of charge as it has been stated and so it would be wise to start with just one service under production to gather real data and evaluate possible future migrations. Not every service may have the best performance under the microservice architecture so a hybrid approach such as the one presented in this work may be a viable solution as well.

Considerations aside, Magycal Interactive can now implement a hybrid approach. To do so, it would be required to follow the next steps.

**Update chat code** with desired behavior for the production service. As stated, persistence and multiple chat rooms are contemplated here.

**A new scaling cluster** to house the microservice chat implementation. Clusters are responsible for managing the number of machines active and can be configured to consider the number of service requests, therefore, horizontally scaling the number of servers for the chat service.

**Create a load balancer** to redirect all chat calls to the new cluster and other requests to the original Magycal.

Bear in mind that these operations have an associated cost. Also, this is but one way to compose the infrastructure. It is suggested this way because no further work than these steps would be required to have a production ready upgrade. The company has already plans to upgrade the platform and this project is going to be used as a starting point.

## 5.3 Future Work

Magycal is a complex platform with many improvements not possible to realize under this work such as:

**New Microservices** Magycal is a monolith with several coupled services. It is necessary to identify and to divide those service into groups and develop new microservices based on them.

**CI/CD Methodology** The present method of development and deployment from Magycal is not fit for the new architectural design. A new pipeline must be created to replace the old one.

**Distributed Databases** A common bottleneck for applications is the connection to a centralized database. Microservices allow for dividing this centralized database into smaller groups, but as the number of the same services grow so does the number of simultaneous connections to the database.

**Software Development Kits (SDK)** Decoupling services make it possible to realize some of the API into different Software Development Kits (SDK) for multiple types of front-user applications. This SDK could be distributed as Software as a Service (SaaS), explained in Chapter 2 so other applications could make use of Magycal.

**Docker** Docker is increasing in popularity nowadays because it allows users to build whole systems detached from the host machine specifications. This feature could be very useful for Magycal because it would smooth the scaling process. Additionally, Amazon already provides support for Docker images installs[2].



## BIBLIOGRAPHY

- [1] AWS. URL: <https://aws.amazon.com/>.
- [2] AWS Docker. URL: <https://aws.amazon.com/docker/>.
- [3] AWS EC2. URL: <https://aws.amazon.com/ec2/>.
- [4] AWS EC2 Free. URL: <https://aws.amazon.com/ec2/instance-types/t2/>.
- [5] AWS EC2 Free Features. URL: <https://aws.amazon.com/ec2/instance-types/>.
- [6] AWS Load Balancing. URL: <https://aws.amazon.com/elasticloadbalancing>.
- [7] AWS RDS. URL: <https://aws.amazon.com/rds/>.
- [8] AWS RDS Features. URL: <https://aws.amazon.com/rds/features/>.
- [9] A. Balalaie, A. Heydarnoori, and P. Jamshidi. “Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture.” In: *IEEE Software* 33.3 (2016), pp. 42–52. DOI: 10.1109/MS.2016.64.
- [10] A. Balalaie, A. Heydarnoori, P. Jamshidi, D. A. Tamburri, and T. Lynn. “Microservices migration patterns.” In: *Software: Practice and Experience* 48.11 (2018), pp. 2019–2042.
- [11] M Bernardo and P Inverardi. “Formal Methods for Software Architectures, Tutorial book on Software Architectures and Formal Methods.” In: *SFM-03: SA Lectures, LNCS 2804* (2003), pp. 62–64.
- [12] R. J. Bril, R. L. Krikhaar, and A. Postma. “Architectural support in industry: a reflection using C-POSH.” In: *Journal of software maintenance and evolution: research and practice* 17.1 (2005), pp. 3–25.
- [13] A. Bucchiarone, N. Dragoni, S. Dustdar, S. T. Larsen, and M. Mazzara. “From monolithic to microservices: An experience report from the banking domain.” In: *Ieee Software* 35.3 (2018), pp. 50–55.
- [14] R. Buyya. “Cloud computing: The next revolution in information technology.” In: *2010 First International Conference On Parallel, Distributed and Grid Computing (PDGC 2010)*. 2010, pp. 2–3. DOI: 10.1109/PDGC.2010.5679963.
- [15] Composer. URL: <https://getcomposer.org/>.
- [16] Continuous Delivery and Integration. URL: <https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment>.

- [17] S. De Santis, L. Florez, D. V. Nguyen, E. Rosa, et al. *Evolve the Monolith to Microservices with Java and Node*. IBM Redbooks, 2016.
- [18] *Design a microservice-oriented application*. URL: <https://docs.microsoft.com/en-us/dotnet/architecture/microservices/multi-container-microservice-net-applications/microservice-application-design>.
- [19] *Docker*. URL: <https://www.docker.com/>.
- [20] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. “Microservices: yesterday, today, and tomorrow.” In: *Present and ulterior software engineering (2017)*, pp. 195–216.
- [21] D. Escobar, D. Cárdenas, R. Amarillo, E. Castro, K. Garcés, C. Parra, and R. Casallas. “Towards the understanding and evolution of monolithic applications as microservices.” In: *2016 XLII Latin American Computing Conference (CLEI)*. IEEE. 2016, pp. 1–11.
- [22] *Fujaba RT Project*. URL: <https://web.cs.upb.de/archive/fujaba/projects/real-time.html>.
- [23] *Go Language*. URL: <https://golang.org/>.
- [24] F. Hacklinger. “Java/A-Taking Components into Java.” In: *IASSE 4 (2004)*, pp. 163–168.
- [25] *JSON Viewer*. URL: <http://jsonviewer.stack.hu/>.
- [26] H. Knoche and W. Hasselbring. “Using microservices for legacy software modernization.” In: *IEEE Software* 35.3 (2018), pp. 44–49.
- [27] J. Lewis and M. Fowler. “Microservices: a definition of this new architectural term.” In: *MartinFowler.com* 25 (2014), pp. 14–26.
- [28] N. Lu, G. Glatz, and D. Peuser. “Moving mountains—practical approaches for moving monolithic applications to Microservices.” In: *International Conference on Microservices (Microservices 2019)*. 2019.
- [29] *Magycal Interactive*. URL: <https://magycal.com/>.
- [30] S. Mary and G. David. “Software architecture: perspectives on an emerging discipline.” In: *Prentice-Hall* (1996).
- [31] *Master-Slave Database Replication*. URL: <https://www.toptal.com/mysql/mysql-master-slave-replication-tutorial>.
- [32] P. M. Mell and T. Grance. *SP 800-145. The NIST Definition of Cloud Computing*. Tech. rep. Gaithersburg, MD, United States, 2011.
- [33] *Microservice Architecture Pattern*. URL: <https://microservices.io/patterns/microservices.html>.
- [34] *Microservices Examples*. URL: <https://blog.dreamfactory.com/microservices-examples/>.

- 
- [35] *Motivation*. URL: <https://motivationping.com>.
- [36] G. Mustapic, A. Wall, C. Norstrom, I. Crnkovic, K. Sandstrom, J. Froberg, and J. Andersson. "Real world influences on software architecture-interviews with industrial system experts." In: *Proceedings. Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA 2004)*. IEEE, 2004, pp. 101–111.
- [37] *NGIX*. URL: <https://www.nginx.com/>.
- [38] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. Josuttis. "Microservices in practice, part 1: Reality check and service design." In: *IEEE Annals of the History of Computing* 34.01 (2017), pp. 91–98.
- [39] *PHP*. URL: <https://www.php.net/>.
- [40] *Redis*. URL: <https://redis.io/>.
- [41] *Redis Issue*. URL: <https://github.com/luin/ioredis/issues/763>.
- [42] M. Richards. *Software architecture patterns*. Vol. 4. O'Reilly Media, Incorporated 1005 Gravenstein Highway North, Sebastopol, 2015.
- [43] M. Shaw and P. Clements. "The golden age of software architecture." In: *IEEE software* 23.2 (2006), pp. 31–39.
- [44] *Showcase*. URL: <https://magycal.com/showcase/>.
- [45] J. Soldani, D. A. Tamburri, and W.-J. Van Den Heuvel. "The pains and gains of microservices: A Systematic grey literature review." In: *Journal of Systems and Software* 146 (2018), pp. 215–232.
- [46] D. Taibi, V. Lenarduzzi, and C. Pahl. "Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation." In: *IEEE Cloud Computing* 4.5 (2017), pp. 22–32.
- [47] J. Thönes. "Microservices." In: *IEEE software* 32.1 (2015), pp. 116–116.
- [48] *Vegeta*. URL: <https://github.com/tsenart/vegeta>.
- [49] A. Verma and S. Kaushal. "Cloud Computing Security Issues and Challenges: A Survey." In: July 2011, pp. 445–454. DOI: 10.1007/978-3-642-22726-4\_46.
- [50] M. Villamizar, O. Garcés, H. Castro, M. Verano Merino, L. Salamanca, R. Casallas, and S. Gil. "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud." In: Oct. 2015. DOI: 10.1109/ColumbianCC.2015.7333476.

