Evaluating the Monolithic and the Microservice Architecture Pattern to Deploy Web Applications in the Cloud

Evaluando el Patrón de Arquitectura Monolítica y de Micro Servicios Para Desplegar Aplicaciones en la Nube

Mario Villamizar, Oscar Garcés, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas
Systems and Computing Engineering Department,
Universidad de los Andes,
Bogotá, Colombia
{mj.villamizar24, ok.garces10, hcastro, m.verano239, l.salamanca10, rcasalla}@uniandes.edu.co

Santiago Gil
Project Management Department
Mapeo
Bogotá D.C., Colombia
sgil@heinsohn.com.co

***Abstract*—Cloud computing provides new opportunities to deploy scalable application in an efficient way, allowing enterprise applications to dynamically adjust their computing resources on demand. In this paper we analyze and test the microservice architecture pattern, used during the last years by large Internet companies like Amazon, Netflix and LinkedIn to deploy large applications in the cloud as a set of small services that can be developed, tested, deployed, scaled, operated and upgraded independently, allowing these companies to gain agility, reduce complexity and scale their applications in the cloud in a more efficient way. We present a case study where an enterprise application was developed and deployed in the cloud using a monolithic approach and a microservice architecture using the Play web framework. We show the results of performance tests executed on both applications, and we describe the benefits and challenges that existing enterprises can get and face when they implement microservices in their applications.**

***Abstract*—La computación en nube ofrece nuevas oportunidades para desplegar aplicaciones escalables de forma eficiente, permitiendo que las aplicaciones empresariales puedan ajustar dinámicamente sus recursos de cómputo bajo demanda. En este artículo analizamos y probamos el patrón de arquitectura de micro servicios, utilizado durante los últimos años por grandes empresas de Internet como Amazon, Netflix y LinkedIn para desplegar grandes aplicaciones en la nube como un conjunto de pequeños servicios que se pueden desarrollar, probar, desplegar, escalar, operar y actualizar de forma independiente, permitiendo que estas empresas logren ganar agilidad, reducir complejidad y escalar sus aplicaciones en la nube de forma más eficiente. Se presenta un caso de estudio en el cuál una aplicación empresarial fue desarrollada y desplegada en la nube utilizando un enfoque monolítico y una arquitectura de micro servicios utilizando el framework de desarrollo web Play. Se muestran los resultados de las pruebas de rendimiento realizadas a ambas aplicaciones, y se describen los beneficios y retos que las empresas existentes deben enfrentar para implementar arquitecturas basadas en micro servicios en sus aplicaciones.**

Keywords—cloud computing; microservices; service oriented architectures; SOA; scalable applications; infrastructure as a services; platform as a service; PaaS; IaaS; continuous delivery; software engineering; software architecture; microservice architecture;

I. Introduction

Cloud computing [1] is a model that allows companies to deploy enterprise applications that (if they have been well designed) can scale their computing resources on demand. Companies can deploy their own applications on Infrastructure as a Service (IaaS) [2] or Platform as a Service (PaaS) [3] solutions or they can buy ready to use applications under the Software as a Services (SaaS) [4] model. When companies deploy their own applications on IaaS or PaaS solutions they face different challenges in order to take advantage of the cloud computing capabilities such as auto scaling, continuous delivery, hot deployments, high availability, dynamic monitoring, among others. Most companies that move an application to an IaaS/PaaS solution typically tried to move a monolithic application. In this context *a monolithic application means an application with a single large codebase/repository that offer tens or hundreds of services using different interfaces such as HTML pages, Web services or/and REST services*.

One of the main reasons companies must move applications to IaaS/PaaS solutions is to gain efficiency in their operations trying to be able to scale those applications on-demand and supporting peak periods. The typical approach is to deploy in the cloud three (presentation, business and persistence) tiers web applications developed with different technological stacks such as JEE, .NET or PHP. Nevertheless, this approach faces different challenges [5] because many technological stacks and applications servers were not designed keeping into account the ability to add/remove servers on demand [6].

Scaling monolithic applications is a challenge because they commonly offer a lot of services, some of them more popular than others. If popular services need to be scaled because they are highly demanded, the whole set of services also will be scaled at the same time, which generate that non popular

This paper is the result of a research project sponsored by Colciencias and Mapeo, a Colombian software company.

services consume large amount of server resources even when they are not going to be used.

From the development methodology most companies deploying applications to the cloud also need to be able to innovate as fast as possible due to their competition may hurt their business if they do not innovate with better product experiences and new features. This is the reason why the term *continuous delivery* [7] has been very popular during the last years mainly in startups, large Internet companies and SaaS providers. The continuous delivery methodology allows companies change and update their applications in production continuously using agile development methodologies and cycles.

In monolithic applications all services are developed on a single codebase shared among multiple developers, when these developers want to add or change services they must guarantee that all other services continue working. The complexity increases as more services are added limiting the ability of companies to innovate with new versions and features. In addition, when new application versions are deployed to production the complete set of services are restarted providing bad experiences to users using them. A monolithic deployment also represents a single point of failure; if the application fails the whole set of services goes down.

The above limitations and problems have been faced by large Internet companies, which have used different mechanisms, strategies and technologies that can be labeled as the microservice architecture pattern [8]. With this pattern these companies have been able to innovate quickly, reduce complexity, scale computing resources efficiently and grow development teams in a controlled way. To understand the efforts and challenges of using this pattern, we developed and tested a case study where we could experience the benefits and challenges of using this pattern in an application while the application is developed, tested, deployed, scaled, operated and upgraded.

The remainder of this paper is organized as follows: Section 2 presents different efforts, strategies and methodologies used in applications developed with the Service Oriented Architecture (SOA) approach and the microservices architecture pattern. The description of the case study developed and tested is presented in Section 3. The implementation of the application using monolithic and microservice architectures is described in Section 4. Section 5 details the application deployments on Amazon Web Services (AWS) infrastructure. Section 6 presents the results of performance tests executed to the monolithic and the microservice application, and presents concerns that must be keep into account to implement microservice architectures. Section 7 concludes and presents several research lines that can be addressed in the near future by industry practitioners, researchers and Open Source communities.

II. Related Work

Applications that need to scale to thousands or millions of users are very common today due to many factors such as the amount of users with Internet access, the use of mobile app stores, the creation of large SaaS products, the creation of massive products by startups, the change of many business models from B2B to B2C, the execution of different government initiatives to provide more online services to citizens, among others. Many of these applications are deployed on IaaS/PaaS solutions to support their rapid growth and unpredictable peak periods.

At enterprise level, an application A starts using a monolithic approach as a single codebase offering a set of services S (S_1, S_2, S_x). The codebase is shared among a set of developers D (D_1, D_2, D_y) and the production environment is maintained and operated by other set of operators O (O_1, O_2, O_z). When the application begins to increase their popularity, more services or developers will be added increasing the complexity and the time required to launch new features or improvements.

The problem of complexity of large business applications have been addressed using different SOA [9] approaches, where an application is divided as a set of business applications A (A_1, A_2, A_x) offering services to others through different protocols (mainly SOAP). Some routing mechanisms/systems such an Enterprise Service Bus (ESB) [10] are used to route/send messages among applications. An SOA strategy allows that each application can be developed by a set of developers teams T (T_1, T_2, T_y) (regularly grouped by business functions), and operated by a team of operators O.

Although SOA implementations can be a solution for the requirements of some companies, these implementations are expensive, time consuming and complex. The challenges of implementing SOA strategies have been widely studied by businesses and academia [11] [12]. Additionally, ESB products were designed to support the workloads of enterprise applications with hundreds or thousands of users, but when ESBs are used with Internet scale applications that have hundreds of thousands or millions of users, they become a bottleneck, generating high latencies and providing a single point of failure. ESBs were not designed with the cloud in mind so it is difficult to add or remove servers to them on demand. In regards to agility, the addition of new requirements for end users in SOA implementations requires a lot of complex configuration in the ESB, which consumes a lot of time.

To avoid the problems of monolithic applications and take advantage of some of the SOA architecture benefits, the microservice architecture pattern has emerged as a lightweight subset of the SOA architecture pattern, that it is being used by companies like Amazon [13], Netflix [14], Gilt [15], LinkedIn [16] and SoundCloud [17] to support and scale their applications and products. There have been a lot of discussions [18] [19] [20] between industrial practitioners and researchers if the microservice architecture pattern is a new software architecture style or if it is the same reference architecture proposed by SOA. Those discussions converge in that microservices adopt SOA concepts that have been used during the last decade, but it is an architecture style more focused in achieving agility [21] and simplicity at business and technical level, avoiding the complexity of centralized ESBs, and allowing development teams can quickly scale and deploy

applications continuously to millions of users on cloud computing solutions.

The microservice pattern [22] proposes to divide an application A as a set of small services μS (μS_1, μS_2, μS_n), each one offering a subset of the services S (S_1, S_2, S_x) provided by the application A. Every microservice is developed and tested independently by a development team μT_i using the technological stack (including the presentation, business and persistence layers) more appropriated to the services offered by the microservice. Each microservice is developed using independent codebases and the team μT_i is also in charge of deploying, scaling, operating, and upgrading the microservice on a cloud computing IaaS/PaaS solution. In the presentation layer the services are published using the software architecture style REST (Representational State Transfer) [23] due to their simplicity and adoption by large Internet companies.

API Gateway from here

In front of the microservices there is a set of gateways applications G (G_1, G_2, G_m) each one offering services to specific types of end users such as web users, iOS users, Android users, public API users, among others. Each gateway exposes their services using different interfaces and protocols such as webpages/HTTP, SOAP/HTTP and REST/HTTP. Gateways receive requests from end-users, consume one or several microservices, and send the results to end users. Each gateway is also developed, tested, deployed, scaled, operated and upgraded independently by a team GT_j. Microservices commonly expose their services to gateways (not to end-users) and gateways typically do not have persistence layers.

The fact that microservices and gateways are developed and maintained by independent teams as self-managed applications, facilitates to increase the number of developers in a more scalable way due to a large and complex monolithic application can be seen as a set of small and simple applications. Each microservice/gateway is developed using different types of programing languages (Java, .NET, PHP, Ruby, Phyton, Scala, etc.) and persistent technologies (SQL, No-SQL, etc.). In the cloud, each microservice/gateway can scale independently using the server types (high CPU, high Memory, high I/O, etc.) and auto scaling rules more appropriated. The microservice pattern also avoid the single point of failure and allow the use of continuous delivery strategies due to that each new deployment only affect the microservice/gateway being updated and other microservices/gateways can continue their operation without disruption.

Although the microservice architecture has been successfully implemented by some companies in the industry, allowing companies to scale their applications in the cloud in an efficient way, to reduce the complexity, to grow development teams easily, and to achieve agility; there is still a lot of challenges that must be keep into account when new companies want to use this pattern in their applications. In this paper we implemented a small enterprise application using the monolithic approach and the microservice pattern to identify areas in the software development process and in the operations and scalability of applications that are affected when this pattern is used. We also compare the performance of using microservices architectures versus using a monolithic approach.

III. Microservice Case Study

To evaluate in a real scenario the implications of using microservice architecture, we worked with a software company to develop an application using the monolithic approach and the microservice pattern. The application is designed to support the business process of generating and querying the payment plans for loans of money delivered by an institution to their customers. The application will be offered to different tenants (institutions) each one consuming the application under the SaaS model. Each tenant will have an admin user that will manage the tenant account and provide access to tenant employees to allow that they can generate and query payment plans when tenant customers visit their offices.

The application will offer a lot of services, however, to provide a simple case study the application was designed to support mainly the two most popular services. The first service, called S_1, is in charge of generating a payment plan with the set of payments (from 1 to 180 months) that must be paid by new tenant customers interested in getting loans of money. The second service, called S_2, is responsible of returning an existing payment plan with its respective set of payments.

The software company, in this case the SaaS provider, is expecting to add a lot of customers during the first year and they plan to deploy this application on an IaaS solution that allow them to scale their infrastructure according to the demand. The service S_1 implements CPU intensive algorithms to generate every payment plan and their typical response time is around 3000 milliseconds. The service S_1 does not store information; it receives some parameters entered by end users (principal (amount), number of payments, credit type and interest rate) and returns the payment plan based on those parameters.

The service S_2 have the responsibility of receive the unique id of a payment plan already stored in a relational database, and return the complete information of the payment plan. There is another service in charge of storing new payment plans in the database, that service was not included in the case study. The typical response time of service S_2 is around 300 milliseconds and this service consumes mainly the database. The number of request per minute that must be initially supported by the application and the response times that will be included in the SLAs of the SaaS solution are summarized in Table I.

Below we describe the two architectures defined to develop the application using a monolithic and microservice architecture:

A. Monolithic architecture

To develop this application using a typical monolithic approach, the application would have a single codebase and it would be developed using an MVC web application framework such as JEE, .NET, Symfony, Rails, Grails, Play, among many others. These frameworks allow to develop three tiers applications providing different tools and libraries to develop the presentation, business and persistence layers. The architecture of a monolithic application is illustrated on Fig 1.

TABLE I. BUSINESS REQUIREMENTS FOR BOTH SERVICES

Service	Requests per Minute	Average Response Time (ms)	Maximum Response Time (ms)
S_1 – Generate a payment plan	30	3,000	6,000
S_2 – Get a payment plan	1,100	300	1,500

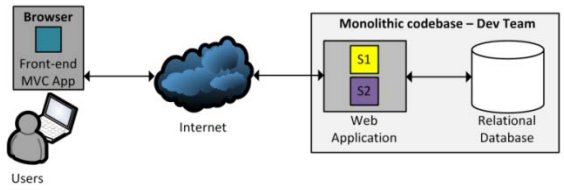

Fig. 1. Monolothic architecture.

At the presentation tier the application may send static assets (HTML, CSS and JavaScript) and dynamic data required by every request to the browsers directly from the web servers; or the application may implement a front-end MVC framework in the browser such as Angular.js [24] or Backbone.js, where most static assets are downloaded by browsers in the first request, and subsequent requests to web server are made invoking REST services using the lightweight data-interchange format JSON (JavaScript Object Notation). Due to the last approach remove load out of web servers, because most processing related to manipulation of HTML, CSS and JavaScript files is executed inside the browser, we decided to use that approach in the monolithic application.

In this architecture the web application publish the two services as REST services over Internet, which are consumed by the front-end MVC application executed in the browser. The monolithic architecture would be deployed on a Cloud solution using the deployment illustrated in Fig. 2. To scale the application, the architecture requires the use of a load balancer, several web servers where the application codebase is deployed, and a relational database to store information.

B. *Microservice architecture*

To use a micro service architecture the first task is to decide the number of microservices that will be used, by simplicity in this case study we decided to select two micro services (μS_1 and μS_2), one for each service of the monolithic application. Each microservice may be developed using different technological stacks as three tiers applications. The architecture of the microservice application is illustrated on Fig 3. At the presentation layer both microservices expose its main service (S1 and S2) using REST over a private network. The services exposed by each microservice are consumed by the gateway. Due to μS_1 only generate new payment plans without storing information, it does not have the persistence layer. In contrast, μS_2 returns the complete information of a payment plan already saved in the relational database requiring persistence.

The gateway is developed as a light web application that receives request from end-users (browsers), gets the result consuming one or more micro services, and returns the results. At the presentation layer, the gateway exposes to Internet two services to end-users (browsers). Hence, it does executes a complex business logic implemented by microservices; it must consumes the services offered by the microservices (μS_1 and μS_2) through REST, gets the result from microservices, and returns the results to end-users. In this architecture the gateway publishes the two services as REST services over Internet which are consumed by the front-end MVC application executed in the browser. The interchange message protocol used between browsers and the gateway, and the gateway and each microservice is JSON. The gateway does not store any information so it does not need a persistence layer.

The microservice architecture would be deployed on a Cloud solution using the deployment illustrated in Fig. 4. To scale this architecture the gateway and each microservice may be scaled independently. The microservice μS_1 is deployed using a load balancer and several web servers. The microservice μS_2 is also deployed using a load balancer and several web servers, but it also uses a relational database to store information. The gateway is deployed using a load balancer and several web servers.

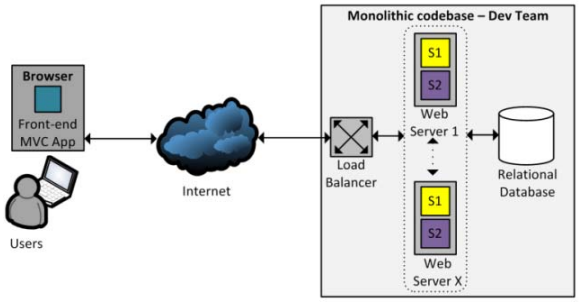

Fig. 2. Deployment of the monolithic architecture.

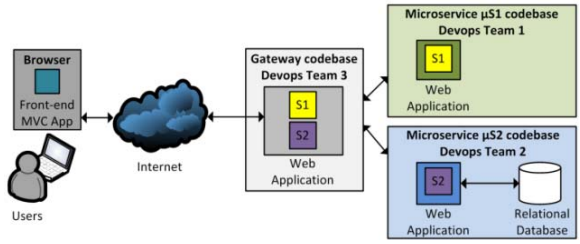

Fig. 3. Microservice architecture.

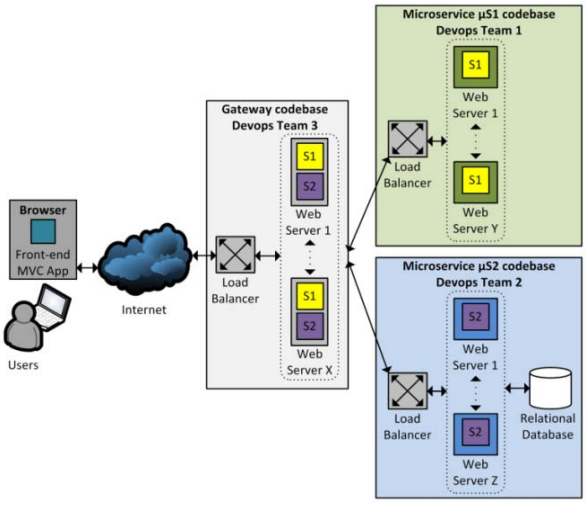

Fig. 4. Deployment of the microservice architecture.

Both architectures were developed by the same development team. In a real scenario the monolithic application would be developed by two teams, a team developing the web application and other for the front-end application. The microservices application would be developed by four small teams, a team developing the gateway, other the microservice μS1, other the microservice μS2, and another the front-end.

IV. Implementation

To implement both architectures we use the Play web framework [25] with Java (Play applications can be developed using Java or Scala). We selected Play because it provides a lightweight, stateless and asynchronous, and cloud friendly architecture. Play applications are executed on the embedded server Netty, which is designed to start its execution in seconds and to consume low computing resources.

In applications that require a relational database, Ebeans was used as ORM because it is the default ORM for Play/Java applications. For the application executed in the browser we decided to develop the front-end MVC application for both architectures with Angular.js due to it is support by Google.

It is important to highlight that both architectures may be implemented using other back-end frameworks such as JEE, .NET, Symfony, Rails or Grails, or front-end frameworks such as Backbone.js or Ember.js; however the goal of this work is to compare how is the process of developing an application using a monolithic architecture versus using the microservice architecture, so we select the same technological stack for both architectures. The implementation of microservice architectures using other frameworks may change in some technical details; however, one of the benefits of using microservices is the ability of using multiple technological stacks, so the architectures, deployments and results generated in this paper may be used as a reference to microservices or gateways implemented in other frameworks.

A. Monolithic architecture development

The monolithic architecture was implemented as two independent applications:

- **Web application**. This application was developed using Play 2.2.2, Scala 2.10.2 and Java 1.7.0. The relational database used was PostgreSQL 9.3.6.
- **Front-end application**. This application was developed using Angular.js 1.3.14 (HTML5, CSS, and jQuery).

B. Microservice architecture development

The microservice architecture was implemented as four independent applications:

- **Microservice μS_1 application**. This application was developed using Play 2.2.2, Scala 2.10.2 and Java 1.7.0.
- **Microservice μS_2 application**. This application was developed using Play 2.2.2, Scala 2.10.2 and Java 1.7.0. The relational database used was PostgreSQL 9.3.6.
- **Gateway application**. This application was developed using Play 2.2.2, Scala 2.10.2 and Java 1.7.0.
- **Front-end application**. This application was developed using Angular.js 1.3.14 (HTML5, CSS, and jQuery).

V. Deployment on a Cloud Computing Infrastructure

To know which are the concerns and challenges to deploy, scale, operate and upgrade microservice architectures in the cloud, both architectures were deployed on Amazon Web Services (AWS).

A. Monolithic architecture deployment

The monolithic architecture was deployed as it is show in Fig. 5. The two applications were deployed as follow:

- **Web application**. This Play web application was deployed on an Elastic Compute Cloud (EC2) instance type *c4.2xlarge (8 vCPUs, 31 ECUs and 15GB RAM)* using the Netty web server 3.7.0. For the PostgreSQL database we use the Relational Database Service (RDS) offered by AWS with an instance type *db.m3.medium (1 vCPU and 3,75GB RAM)*. The services of the web application were exposed to Internet.
- **Front-end application**. The static files of the Angular.js application (views, models and controllers) were stored on the Play web server. When a user enters to the web application, in the first request the assets of Angular.js are downloaded to the browser from the web server; then the REST services exposed by the web server are consumed from the Angular.js application (executed in the browser) using JSON.

B. Microservice architecture deployment

The microservice architecture was deployed as it is illustrated in Fig. 6. The four applications were deployed as follow:

- **Microservice μS_1 application**. This Play web application was deployed on an EC2 instance type *c4.xlarge (4 vCPUs, 16 ECUs and 7,5GB RAM)* using the Netty web server 3.7.0. The REST services exposed by the Microservice μS_1 were configured to be accessible only by the gateway.
- **Microservice μS_2 application**. This Play web application was deployed on an EC2 instance type *m3.medium (1 vCPUs, 3 ECUs and 3,75GB RAM)* using the Netty web server 3.7.0. For the PostgreSQL database we use the Relational Database Service (RDS) offered by AWS with an instance type *db.m3.medium (1 vCPU and 3,75GB RAM)*. The REST service exposed by the Microservice μS_2 were also configured to be accessible only by the gateway.

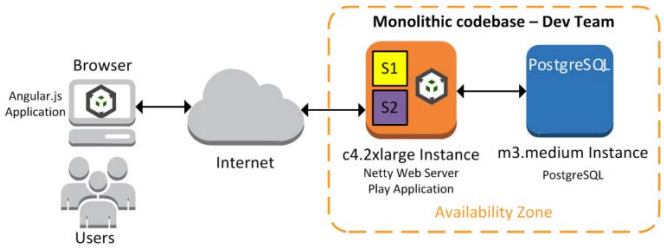

Fig. 5. Deployment of the monolithic architecture on Amazon Web Services

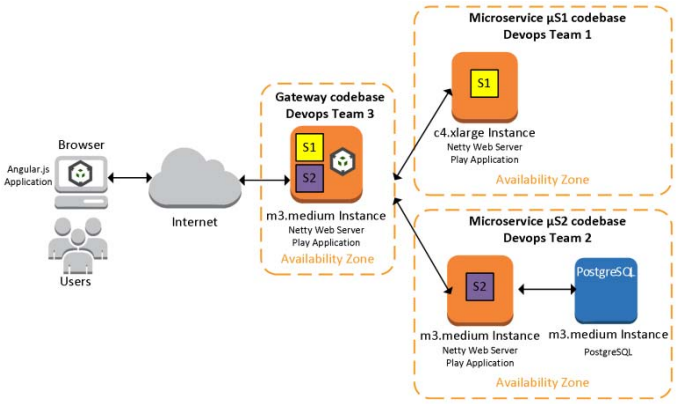

Fig. 6. Deployment of the microservice architecture on Amazon Web Services

- **Gateway application**. This Play web application was deployed on an EC2 instance type *m3.medium (1 vCPUs, 3 ECUs and 3,75GB RAM)* using the Netty web server 3.7.0. The REST services of the gateway were exposed to Internet.
- **Front-end application**. Similar to the monolithic application, the static files of the Angular.js application were stored on the Gateway. When a user enters to the web application, in the first request the assets of Angular.js are downloaded to the browser from the gateway; then the REST services exposed by the gateway are consumed from the Angular.js application (executed in the browser) using JSON.

The instance types used in both architectures were selected based on performance tests executed (described in Section IV). For each architecture we selected the smaller instance that supported the business requirements (t2 instances type family was not keep into account due to its CPU volatility). The costs of running the monolithic architecture and the microservice architecture are summarized in Table II and Table III. Costs associated to bandwidth, storage and backups were not kept into account as they will be the same in both cases. The costs were calculated in June 2015 using the pricing list available to the US East (N. Virginia) Region.

In the deployment of the monolithic architecture a single instance was used for the web server, and in the deployment of the microservice architecture three instances were used: one for each microservice and one for the gateway; as it is shown in Fig. 2 and Fig. 4, a load balancer with several instances may be used to scale the gateway and microservice capacities; however those deployments and tests are proposed as future work.

VI. Test and Results

With the development of the case study different areas of the monolithic vs microservice architecture were evaluated at different levels such as performance, development methodology, deployment and operation, and the adoption process by businesses. The experiences and results are described in this section.

TABLE II. Infrastructure Costs of the Monolithic Deployment

Service	Cost per Hour (USD)	Quantity per Month	Cost per Month (USD)
Web application. 1 EC2 instance c4.2xlarge.	$0.464	720	$334.08
Web application. 1 RDS instance db.m3.medium with Single-AZ.	$0.090	720	$64.80
Total cost per month			**$398.88**

TABLE III. Infrastructure Costs of the Microservice Deployment

Service	Cost per Hour (USD)	Quantity per Month	Cost per Month (USD)
Microservice μS_1. 1 EC2 instance c4.xlarge.	$0.232	720	$167.04
Microservice μS_1. 1 RDS instance db.m3.medium with Single-AZ.	$0.090	720	$64.80
Microservice μS_2. 1 RDS instance m3.medium.	$0.067	720	$48.24
Gateway. 1 Instance m3.medium.	$0.067	720	$48.24
Total cost per month			**$328.32**

A. Performance

To analyze how it is the performance of both architectures to support the business requirements listed in Table I, we configure a JMeter [26] 2.13 instance on AWS to compare how is the average response time of the monolithic versus the microservice architecture. In the test, JMeter was configured to execute 30 requests per minute to the service S1 and 1,100 requests per minute to the service S2 during 10 minutes, to simulate a constant workload; in the monolithic architecture the requests were sent to the REST services published by the web application (see Fig. 5), and in the microservice architecture the requests were sent to the REST services published by the gateway (see Fig. 6).

The execution of performance tests with JMeter allow us to identify the more suitable instance type used for each architecture. For the deployment of the monolithic Play application the performance tests were executed on c4 family instance types (optimized for CPU) such as c4.large, c4.xlarge and c4.2xlarge; the c4.2xlarge was the instance type that support the business requirements. In the microservice deployment a similar process was followed to identify the more appropriate instance type. For the three Play applications, microservice μS_1, microservice μS_2, and gateway, the instance types identified were c4.xlarge, m3.medium and m3.medium respectively.

The average response time and the 90% line response time (the value below which 90% of the requests fall) we get with both architectures during the performance tests with JMeter are shown in Fig. 7. and Fig 8. Those figures show that both architectures supported the business requirements (described in Table I), and validate that microservices does not impact considerably the latency of the responses due to the use of more hosts (in this case the gateway).

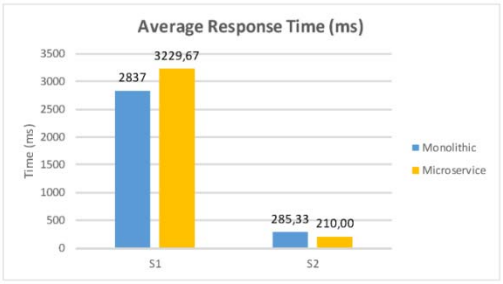

Fig. 7. Average response time with the monolithic and microservice architecture

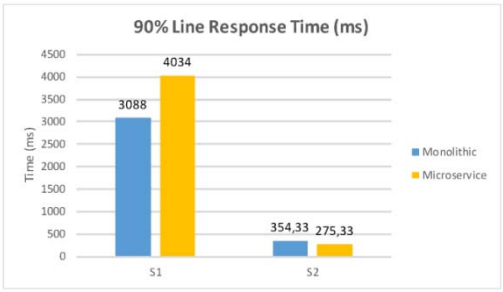

Fig. 8. 90% Line response time with the monolithic and microservice architecture

They also validate that microservice architectures allow to select instance types at more granular level (per microservice and gateway) which help to reduce costs. In this case study with the microservice architecture the provider may reduce infrastructure costs by 17%, (see Table II and Table III).

B. *Development methodology and efforts*

The development of applications using microservice architectures requires a change in the way that most companies and software vendors have traditionally been developing monolithic applications using single codebases. Microservice architectures allow small teams to work on small applications (microservices) without worrying about how other microservices or teams work. Every team can use different technologies to implement microservices/gateways according to the business and technical requirements; to avoid the use of a lot of technologies that can be difficult to be managed some guidelines should be offered to all teams so they can decide which set of technologies to use in each microservice/gateway. The design, documentation and publication of REST API is very important to allow that services published by microservices can be easily consumed by gateways, and services published by gateways can be easily consumed by end-user applications (browsers, mobile apps, APIs, etc.).

During the development process we validated that microservices introduce many problems of distributed systems (failures, timeouts, distributed transactions, data federation, responsibility assignments, etc.) which make the development process more complex in some areas. Many problems that in monolithic applications are typically delegated to application containers (JBOSS, Glassfish, WebLogic, IIS) or in SOA implementation are managed by ESB, in microservices must be managed at application level. To deploy and scale microservices and gateways on cloud environments, it is recommended to use lightweight/embedded servers, such as Netty or Jetty which offer basic functionalities, can be started in seconds, do not share state, and can be added or removed from clusters and load balancers at any time.

C. *Deployment, scaling, and continuous delivery*

The deployment of the microservice architecture on AWS required the deployment of several independent applications (microservices and gateways). The deployment of each microservice or gateway required specific configurations on the application and on services offered by AWS; when new versions of the gateway or a microservice were published it was very easy to break external dependent services; so in microservices architectures it is very important to maintain service versioning. When new services are added or existing services are deprecated, it is very important that different teams (publishers and consumers) work and plan together the upgrade process to avoid breaking problems.

The use of continuous delivery strategies in microservices can be a time consuming task, due to repetitive and manual tasks must be executed in each deployment; the use of automations tools are mandatory to save time and gain agility. Microservice implementations also require the use of Devops (Development + Operation) strategies where teams that develop applications also deploy, operate and monitor them in the cloud. In the deployment on AWS we could monitor each application (gateway/microservices) easily using New Relic; however one of the problems faced during the deployment is how to trace the flow of a request made by an end-user application through gateways and microservices.

Although we do not execute test to scale horizontally each microservice/gateway to support more requests during peak periods; in the deployment shown in Fig. 6, the gateway and each microservice could be scaled vertically to use the number of instances more appropriated. That is one of the benefits of using microservices because each microservice/gateway can scale independently using different policies. At business level this may represent large saving in IT infrastructure costs, and a more efficient way to take advantage of the pay per use and on demand benefits of the cloud model.

D. *Adoption, business culture and guidelines*

Based on the case study, microservices architectures should be employed in businesses that need to scale their applications to hundreds of thousands or millions of users, because its implementation requires additional abilities that are not present in many companies. The adoption of microservice architectures require a new culture of development and innovation that must be complemented by a set of guidelines and good practices at company/application level. Those practices should be shared by all of the teams developing microservices/gateways. The adoption of microservices should be implemented as a long term business strategy and it should not be seen as a project, because their adoption requires efforts and abilities that must be developed incrementally.

For applications with a small number of users (hundreds or thousands of users), the monolithic approach may be a more practical and faster way to start. In practical cases that were reviewed [13] [14] [15] [16] [17], most applications using

microservice architectures started as monolithic applications and due to scaling problems at infrastructure and team management, they were incrementally modified to implement microservices. Large microservice implementation can be composed of tens or event hundreds of microservices that are developed, tested, deployed, scaled, operated and upgraded independently by small teams, allowing to those companies to implement new features and improvements quickly.

VII. Conclusions and Future Work

Based on the development of a case study we could experience and identify some of the benefits and challenges that microservice architectures provide to businesses that want to offer applications to thousands or millions of users. One of the benefits of using microservices is the ability to publish a large application as a set of small applications (microservices) that can be developed, deployed, scaled, operated and monitored independently. Microservices allow companies to manage large codebase applications using a more practical methodology where incremental improvements are executed by small teams on independent codebases and deployments. The agility, cost reduction and granular scalability, brings some challenges of distributed systems and team development management practices that must be considered.

As future work we will execute more performance tests to evaluate more deeply the benefits of scaling each microservice/gateway independently and the cost reductions that companies can gain using more granular scaling policies. The process of how to define the number of microservices and gateways is another area that will be considered. The process to automate the deployment of microservices and gateways using different strategies, tools and/or managed cloud services (such as Docker, Amazon EC2 Container Service, and AWS Lambda) will be evaluated. Tests to evaluate other concerns of microservices such as failures tolerance, distributed transactions, heterogeneous data distribution, service versioning and microservice granularity, also will be an interest area. An analysis of how to adopt solutions already implemented in SOA/ESB implementations to microservice environments will also be done.

References

[1] Rajkumar Buyya, "Cloud computing: The next revolution in information technology," in *1st International Conference on Parallel Distributed and Grid Computing (PDGC)*, Solan, 2010, pp. 2-3.

[2] Peter Vosshall, "Web Scale Computing: The Power of Infrastructure as a Service," in *Service-Oriented Computing – ICSOC 2008*, Athman Bouguettaya, Ingolf Krueger, and Tiziana Margaria, Eds. Berlin, Germany: Springer, 2008, pp. 1-1.

[3] Daniel Beimborn, Thomas Miletzki, and Stefan Wenzel, "Platform as a Service (PaaS)," *Business & Information Systems Engineering*, vol. 3, no. 6, pp. 381-384, December 2011.

[4] Sebastian Walter Schütz, Thomas Kude, and Karl Michael Popp, "The Impact of Software-as-a-Service on Software Ecosystems," in *Software Business. From Physical Products to Software Services and Solutions*, Georg Herzwurm and Tiziana Margaria, Eds. Berlin, Germany: Springer, 2013, pp. 130-140.

[5] Giuseppina Cretella and Beniamino Di Martino, "An Overview of Approaches for the Migration of Applications to the Cloud," in *Lecture Notes in Information Systems and Organisation*, Leonardo Caporarello, Beniamino Di Martino, and Marcello Martinez, Eds.: Springer, 2014, pp. 67-75.

[6] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A. Lozano, "A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments," *Journal of Grid Computing*, vol. 12, no. 4, pp. 559-592, December 2014.

[7] Joachim Rossberg and Mathias Olausson, "Continuous Delivery," in *Pro Application Lifecycle Management with Visual Studio 2012*.: Apress, 2012, ch. 4, pp. 425-432.

[8] James Lewis and Martin Fowler. (2014, March) Microservices. [Online]. http://martinfowler.com/articles/microservices.html

[9] James McGovern, Oliver Sims, Ashish Jain, and Mark Little, *Enterprise Service Oriented Architectures*.: Springer, 2006.

[10] Hyun Jung La, Jeong Seop Bae, Soo Ho Chang, and Soo Dong Kim, "Practical Methods for Adapting Services Using Enterprise Service Bus," in *Web Engineering*, Luciano Baresi, Piero Fraternali, and Geert-Jan Houben, Eds. Berlin, Germany: Springer, 2007, pp. 53-58.

[11] M.P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-Oriented Computing: State of the Art and Research Challenges," *Computer*, vol. 40, no. 11, pp. 38-45, November 2007.

[12] J. Hutchinson et al., "Evolving Existing Systems to Service-Oriented Architectures: Perspective and Challenges," in *ICWS 2007. IEEE International Conference on Web Services*, 2007, pp. 896-903.

[13] Staci Kramer. (2011, October) GIGAOM | The Biggest Thing Amazon Got Right: The Platform. [Online]. https://gigaom.com/2011/10/12/419-the-biggest-thing-amazon-got-right-the-platform/

[14] Tony Mauro. (2015, February) Nginx | Adopting Microservices at Netflix: Lessons for Architectural Design. [Online]. http://nginx.com/blog/microservices-at-netflix-architectural-best-practices/

[15] Yoni Goldberg. (2014, October) InfoQ | Scaling Gilt: from Monolithic Ruby Application to Distributed Scala Micro-Services Architecture. [Online]. http://www.infoq.com/presentations/scale-gilt

[16] Steven Ihde. (2015, March) InfoQ | From a Monolith to Microservices + REST: the Evolution of LinkedIn's Service Architecture. [Online]. http://www.infoq.com/presentations/linkedin-microservices-urn

[17] Phil Calçado. (2014, June) SoundCloud | Building Products at SoundCloud - Part I: Dealing with the Monolith. [Online]. https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-1-dealing-with-the-monolith

[18] Mark Little. (2014, March) InfoQ | Microservices and SOA. [Online]. http://www.infoq.com/news/2014/03/microservices-soa

[19] Johannes Thones, "Microservices," *IEEE Software*, vol. 32, no. 1, pp. 116-116, 2015.

[20] Bob Rhubart. (2015, March) Oracle | Microservices and SOA. [Online]. http://www.oracle.com/technetwork/issue-archive/2015/15-mar/o25architect-2458702.html

[21] George Lawton. (2015, January) TechTarget | How microservices bring agility to SOA. [Online]. http://searchcloudapplications.techtarget.com/feature/How-microservices-bring-agility-to-SOA

[22] Chris Richardson. (2014, March) Microservices | Pattern: Microservices Architecture. [Online]. http://microservices.io/patterns/microservices.html

[23] S. Vinoski, "REST Eye for the SOA Guy," *IEEE Internet Computing*, vol. 11, no. 1, pp. 82-84, January 2007.

[24] V. Balasubramanee, C. Wimalasena, R. Singh, and M. Pierce, "Twitter bootstrap and AngularJS: Frontend frameworks to expedite science gateway development," in *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, Indianapolis, 2013, p. 1.

[25] John Hunt, "Play Framework," in *A Beginner's Guide to Scala, Object Orientation and Functional Programming*.: Springer, 2014, pp. 413-428.

[26] Dan Rahmel, "Testing a Site with ApacheBench, JMeter, and Selenium," in *Advanced Joomla!*: Apress, 2013, pp. 211-247.