SOCA (2017) 11:233–247
DOI 10.1007/s11761-017-0208-y

ORIGINAL RESEARCH PAPER

Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures

Mario Villamizar[1] · Oscar Garcés[1] · Lina Ochoa[1] · Harold Castro[1] · Lorena Salamanca[2] · Mauricio Verano[2] · Rubby Casallas[2] · Santiago Gil[3] · Carlos Valencia[3] · Angee Zambrano[3] · Mery Lang[3]

Received: 13 June 2016 / Revised: 10 April 2017 / Accepted: 12 April 2017 / Published online: 27 April 2017

Abstract Large Internet companies like Amazon, Netflix, and LinkedIn are using the microservice architecture pattern to deploy large applications in the cloud as a set of small services that can be independently developed, tested, deployed, scaled, operated, and upgraded. However, aside from gaining agility, independent development, and scalability, how microservices affect the infrastructure costs is a major evaluation topic for companies adopting this pattern. This paper presents a cost comparison of a web application developed and deployed using the same scalable scenarios with three different approaches: 1) a monolithic architecture, 2) a microservice architecture operated by the cloud customer, and 3) a microservice architecture operated by the cloud provider. Test results show that microservices can help reduce infrastructure costs in comparison with standard monolithic architectures. Moreover, the use of services specifically designed to deploy and scale microservices, such as AWS Lambda, reduces infrastructure costs by 70% or more, and unlike microservices operated by cloud customers, these specialized services help to guarantee the same performance and response times as the number of users increases. Lastly, we also describe the challenges we faced while implementing and deploying microservice applications, and include a discussion on how to replicate the results on other cloud providers.

Keywords Cloud computing · Microservices · Service-oriented architectures · Scalable applications · Software engineering · Software architecture · Microservice architecture · Serverless architectures · AWS Lambda · Amazon Web Services

This work is the result of a research project partially funded by Colciencias under contract 0569-2013 “Construcción de una línea de productos de aplicaciones de crédito/cartera que serán ofrecidas a través de un Marketplace en forma SaaS para el sector no bancarizado (PYMES) de Colombia” COD. 1204-562-37152.

✉ Mario Villamizar
mj.villamizar24@uniandes.edu.co

Oscar Garcés
ok.garces10@uniandes.edu.co

Lina Ochoa
lm.ochoa750@uniandes.edu.co

Harold Castro
hcastro@uniandes.edu.co

Lorena Salamanca
l.salamanca10@uniandes.edu.co

Mauricio Verano
m.verano239@uniandes.edu.co

Rubby Casallas
rcasalla@uniandes.edu.co

Santiago Gil
sgil@heinsohn.com.co

Carlos Valencia
cvalencia@heinsohn.com.co

Angee Zambrano
azambrano@heinsohn.com.co

Mery Lang
mlang@heinsohn.com.co

[1] COMIT Research Group, Systems and Computing Engineering Department, Universidad de los Andes, Bogotá D.C., Colombia

[2] TICSw Research Group, Systems and Computing Engineering Department, Universidad de los Andes, Bogotá D.C., Colombia

[3] Project Management Department, Mapeo, Bogotá D.C., Colombia

1 Introduction

Cloud computing [1] is a model that allows companies to deploy enterprise applications that, if properly designed, can scale their computing resources on demand. Companies can either 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 that use the Software as a Service (SaaS) [4] model. When companies deploy their own applications on IaaS or PaaS solutions in order to take advantage of cloud computing capabilities such as configuring auto scaling, ensuring continuous delivery and hot deployments, and achieving high availability and dynamic monitoring, among others, they face different time and cost-consuming challenges. Furthermore, while trying to avoid the hassle and the development costs of migrating to the cloud, most companies start by deploying traditional and monolithic applications that use IaaS/PaaS solutions.

In this context, we take the definition made by Martin Fowler in [5] about what a *monolithic architecture* means. A monolithic architecture makes reference to an application with a single codebase/repository that expose tens or hundreds of different services to external systems or consumers using different interfaces such as HTML pages, Web services, and/or REST services. The application is developed by a development group and changes made by any developer can affect the whole set of services because all changes are made on the same codebase. The codebase can be deployed on single-server or multi-server (behind a load balancer) environments.

Scaling a monolithic application is a challenge because of the irregular consumption behaviour of the different services deployed on the same application. Thus, when the demand for highly consumed services increases, additional infrastructure is required for the whole application, regardless of the consumption pattern of the services. Therefore, given that infrastructure is shared among services, server resources are wasted in the execution of unused services, thus increasing related costs.

Microservice architectures propose a solution to efficiently scale computing resources and help solve many other issues that arise in monolithic architectures [5]. The allocated infrastructure can be better tailored to the microservices' needs due to the independent scaling of each one of them, which eventually diminishes the infrastructure costs needed to run the applications.

However, microservice architectures face additional challenges such as the effort required to deploy each microservice, and to scale and operate them in cloud infrastructures. To address these concerns, services like AWS Lambda [6] have been launched by leading providers like Amazon Web Services (AWS). AWS Lambda allows implementing microservice architectures without the need to manage servers. Thus, it facilitates the creation of functions (i.e. microservices) that can be easily deployed and automatically scaled, and it also helps reduce infrastructure and operation costs.

To understand how microservice architectures and AWS Lambda affect the infrastructure costs of an application, we developed and tested a case study with a real application implemented and deployed in the cloud using three different approaches: a monolithic architecture, a microservice architecture managed and scaled by the cloud consumer, and a microservice architecture automatically managed and scaled by the cloud provider (i.e. AWS Lambda). In this case study, we identified and compared the infrastructure costs required for each approach and also detected some of the efforts and challenges of using each approach while the application was 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 that was developed and tested is presented in Sect. 3. The implementation of the different architectures is described in Sect. 4. Section 5 describes the application deployments on the AWS infrastructure. Section 6 shows the results of performance tests executed for each architecture in order to compare their infrastructure cost, and presents the concerns and tradeoffs that must be considered by companies when trying to implement microservices. Section 7 presents how the results may be used to test other serverless services such as those launched recently by Google, Microsoft, and IBM. Section 8 concludes and presents several research lines that can be addressed.

2 Background

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

Suppose that, at an enterprise level, an application A starts using a monolithic approach as a single codebase and offers a set of services S ($S_1, S_2, \ldots, S_x$). The codebase is shared among a set of developers D ($D_1, D_2, \ldots, D_y$), and the production environment is operated by a set of operators O ($O_1, O_2, \ldots, O_z$). When the application begins to increase its demand, more services or developers will be added, thus increasing the complexity and the time required to launch new features or improvements. The problem of the complexity of large business applications has been addressed using different SOA [7] approaches, where an application is divided as a set of business applications A ($A_1, A_2, \ldots, A_x$) and each one offers services to the others through different protocols (mainly SOAP). Some routing mechanisms/systems, such as the Enterprise Service Bus (ESB) [8], are used to route/send messages among applications. A SOA strategy allows each application to be developed by a set of developer teams T ($T_1, T_2, \ldots, T_y$) (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, such implementations are expensive, time-consuming, and complex [9]. Therefore, the challenges of implementing SOA strategies have been widely studied by businesses and academia [10]. 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 for cloud environments, because they make it difficult to add or remove servers on demand. Regarding agility, the addition of new requirements for end-users in SOA implementations requires a lot of complex configurations in the ESB, which is a time-consuming task.

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. This pattern is being used by companies like Amazon [11], Netflix [12], Gilt [13], LinkedIn [14], and SoundCloud [15] to support and scale their applications and products. There has been a lot of discussion [16–18] between industrial practitioners and researchers about whether the microservice architecture pattern is a new software architecture style or the same reference architecture proposed by SOA. That discussion converges on the idea that microservices adopted SOA concepts used during the last decade, but are still an architecture style focused on achieving agility [19] and simplicity at business and technical levels, while avoiding the complexity of centralized ESBs and allowing development teams to quickly and continuously scale and deploy applications to millions of users.

The microservice pattern [20] proposes to divide an application A into a set of small business services μS ($\mu S_1, \mu S_2, \ldots, \mu S_n$), each one of them offering a subset of the services S ($S_1, S_2, \ldots, S_x$) provided by such application A. Every microservice is developed independently by a development team μT_i by using the technological stack —including the presentation, business, and persistence layers—that is more appropriate for 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, and operating the microservice in a cloud computing IaaS/PaaS solution. In the presentation layer, the services are published using the REST (Representational State Transfer) [21] architecture style due to its simplicity and its adoption by large Internet companies.

Facing the defined microservices, there is a set of application gateways G ($G_1, G_2, \ldots, G_m$), 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 its services using different interfaces and protocols like webpages/HTTP, SOAP/HTTP, and REST/HTTP. Gateways receive requests from end-users, consume one or several microservices, and send the results to the requesters. Each gateway is also independently developed, tested, deployed, scaled, operated, and upgraded by a team GT_j. Microservices commonly expose their services to gateways instead of end-users, and they typically do not have persistence layers.

The fact that microservices and gateways are developed and maintained as self-managed applications by independent teams, allows increasing the number of developers in a more scalable way. Moreover, a large and complex monolithic application can be regarded as a set of small and simple applications. Each microservice/gateway is developed using different types of programing languages (e.g. Java, .NET, PHP, Ruby, Phyton, Scala) and persistent technologies (e.g. SQL, No-SQL). In the cloud, each microservice/gateway can be scaled independently, using the server types (e.g. high CPU, high Memory, high I/O) and auto scaling rules that are more appropriate. The microservice pattern also avoids single points of failure and allows the use of continuous delivery strategies, because each new deployment affects only the microservice/gateway being updated while other microservices/gateways continue their operation without disruption.

One of the concerns of implementing microservices is related to the efforts required to deploy and scale each microservice/gateway in the cloud. Although companies implementing microservices can use different DevOps [22] automation tools such as Docker, Chef, Puppet, Auto Scaling (Amazon), among others, the implementation of such tools consumes time and resources. To address this concern, cloud providers like AWS have recently launched services such as AWS Lambda, which allows the deployment of microservices without the need to manage servers. This service is designed to offer a per request cost structure, which results in developers only worrying about writing individual functions to implement each microservice/gateway and then deploying them on AWS Lambda. Once deployed, those functions can be scaled automatically; AWS charges for each function execution, while hiding the deployment, operation, and monitoring of load balancers or web servers. This per request model helps reduce infrastructure costs because each function can be executed in computing environments adjusted to its requirements, and the customer pays only for each function execution, thus avoiding infrastructure payment when there is no microservice/gateway consumption (base infrastructure is not required).

Microservice architectures are being implemented by large companies to scale their applications in the cloud in an efficient way, to reduce complexity, to easily expand development teams and to achieve agility. However, when companies want to start adopting microservices while developing new applications, they generally have some of the following doubts: 1) how can microservices help to reduce infrastructure costs, 2) how are the development process and business culture changed when microservices are implemented, and 3) how can emerging cloud services such as AWS Lambda—designed to allow the automatic deployment and scalability of microservices— help reduce infrastructure costs.

To answer the questions above, we developed a small enterprise application using the monolithic architecture, the microservice architecture operated by the cloud customer, and the microservice architecture operated by the cloud provider. We deployed the three applications on AWS in order to get and compare the infrastructure costs required to execute them, and we identified areas in the software development process, and in the operations and scalability of the applications, that are affected when microservice architectures are used.

3 Case study

In order to evaluate the implications of using microservices operated by the cloud customer and the cloud provider in a real scenario, versus using a monolithic architecture, we worked together with a software company in the development of an application that uses the three aforementioned architectures. The application was designed to support the business process of generating and querying payment plans for loans of money delivered by an institution to its customers. The application was offered to different tenants (institutions), each one using the application within the SaaS model. Each tenant had an admin user that managed the tenant account. The admin user provided access to tenant employees, allowing them to generate and query payment plans when tenant customers visited their offices. In the database layer, the application used the shared database multi-tenant model, where the information of all tenants is stored in the same database schema. At an application level, all tenants were supported using the same set of applications/web servers.

The company (i.e. the SaaS provider) expected to add a lot of tenants with thousands of users during the first years of operation. In addition, they planned to deploy this application in an IaaS solution that would enable them to scale their infrastructure on demand. In order to provide a simple case study, we considered only two services from the complete set of services offered by the original application.

The first service, called S_1, was in charge of generating a payment plan that included the set of payments (from 1 to 180 months). This service implemented CPU intensive algorithms to generate payment plans because it uses different variables related to the customer acquiring the credit (ages, salary, location, expenses, etc.), the type of credit (mortgage, vehicles, etc.), among other topics defined by the SaaS customers. It did not store any information in the database, and its typical response time was around 3000 milliseconds. Moreover, it received some parameters (principal, number of payments, credit type, interest rate, etc.) and returned the payment plan based on those parameters.

The second service, called S_2, was responsible for returning an existing payment plan and its corresponding set of payments. This service received the unique ID of a payment plan stored in a relational database, and returned the complete information of the payment plan and its set of payments (the service used to store payment plans was not considered). The typical response time of S_2 was around 300 milliseconds and it had a high consumption of database queries.

Below, we describe the three architectures that were defined to develop the application:

3.1 Monolithic architecture

In order to develop the application using a typical monolithic approach, we kept in mind that it should have a single codebase and that it should be developed using an MVC web application framework such as JEE, .NET, Symfony, Rails, Grails, Play, among others. Such frameworks enable the development of three-tier applications, and provide different tools and libraries to develop the presentation, business and persistence layers. The architecture of a monolithic application is illustrated in Fig. 1.

At the presentation tier, the application servers generally send the required static assets (HTML, CSS, and JavaScript)

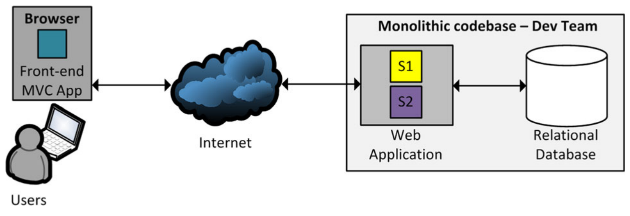

Fig. 1 Monolithic architecture

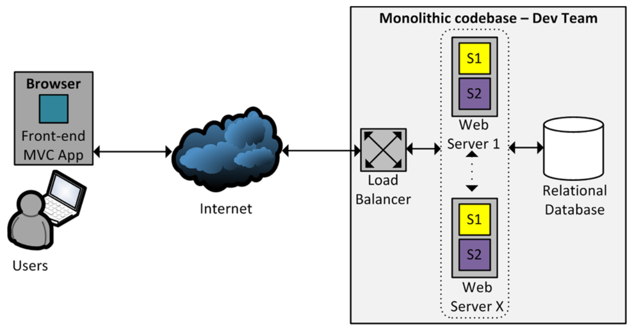

Fig. 2 Deployment of the monolithic architecture

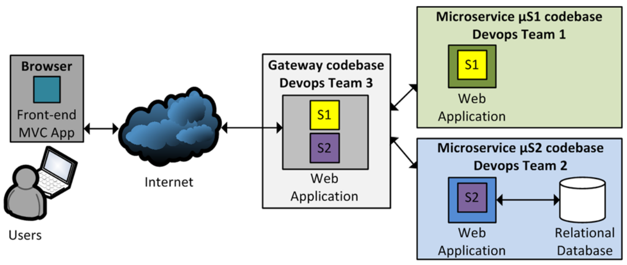
Browser
Front-end MVC App
Users
Internet
Gateway codebase Devops Team 3
S1
S2
Web Application
Microservice µS1 codebase Devops Team 1
Web Application
Microservice µS2 codebase Devops Team 2
Web Application
Relational Database

Fig. 3 Microservice architecture

and dynamic data to the browsers; however, the application may implement a front-end MVC framework in the browser, such as Angular.js [24] or Backbone.js. In this case, most static assets are downloaded by browsers in the first request, and subsequent requests to web servers are performed by invoking REST services using JavaScript Object Notation (JSON)—a lightweight data-interchange format. Given that the last approach removes load from web servers, due to the execution of HTML, CSS, and JavaScript inside the browser, we decided to use that approach in the monolithic application.

In this architecture, the web application publishes the two services as REST services over the Internet, and they are consumed by the MVC front-end application executed in the browser. This approach can be deployed in a single-server environment, where the scalability is limited to the computing resources of one server, or it can be deployed in a multi-server environment. As proposed by Bass et al. [23], in multi-server environments, a load balancer is required to distribute the load among multiple applications servers. Additionally, several web servers are deployed with the application codebase and a relational database is used to store information. The deployment of the monolithic architecture in a multi-server environment is shown in Fig. 2, where the scalability is limited to the computing resources of the server cluster, where several instances of the same type are used, allowing to scale the application in a linear way (1X, 2X, 3X, etc.).

Strategies to scale monolithic applications at more granular levels (1.1X, 1.2X, 1.3X, etc.) could be achieved using different instance types; however, these strategies are difficult to manage and scale, due to, for example, AWS autoscaling group only allows automatically adding or removing cluster instances of the same type. As described above, these scaling strategies were not considered in the paper.

3.2 Microservice architecture operated by the cloud customer

In order to use a microservice architecture, the first task relies on deciding the number of microservices that will be implemented. For simplicity, in this case study we selected two micro services (μS_1 and μS_2), one for each service of the monolithic application. Each microservice may be developed as an independent three-tier application using different technological stacks. The microservice architecture proposed for the application is illustrated in Fig. 3. At the presentation layer, both microservices expose their main service (S_1 and S_2) using REST over a private network. The service exposed by each microservice is consumed by the gateway. Given that μS_1 only generates new payment plans without storing information, it does not need the persistence layer. In contrast, μS_2 returns the complete information of a payment plan saved in the relational database, therefore, it requires persistence.

The gateway was developed as a light web application that receives requests from end-users (browsers) through the Internet, consumes the private services offered by the microservices (μS_1 and μS_2) through REST, gets the results from the microservices, and returns the results to end-users. In this architecture, the gateway publishes the two services as REST services over the Internet, which are consumed by the MVC front-end application executed in the browser. The message interchange 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 can be deployed in a cloud solution using the deployment illustrated in Fig. 4. In this architecture, the gateway and each microservice can be scaled independently. Microservice μS_1 is deployed using a load balancer and several web servers. Microservice μS_2 is also deployed using a load balancer and several web servers, and it also uses a relational database. The gateway is deployed using a load balancer and several web servers.

3.3 Microservice architecture operated by AWS Lambda

For the microservice architecture operated by the cloud provider, we have selected AWS and its AWS Lambda service, because it was the first cloud service specifically designed to run microservices. As mentioned in Sect. 7, other cloud providers have started to offer similar services.

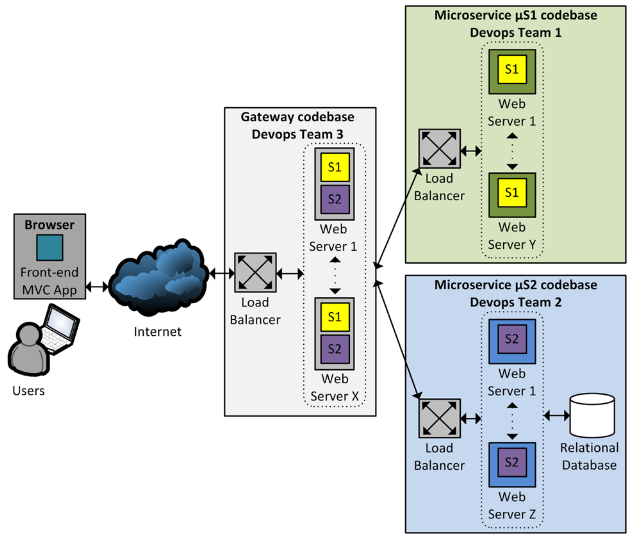

Fig. 4 Deployment of the microservice architecture

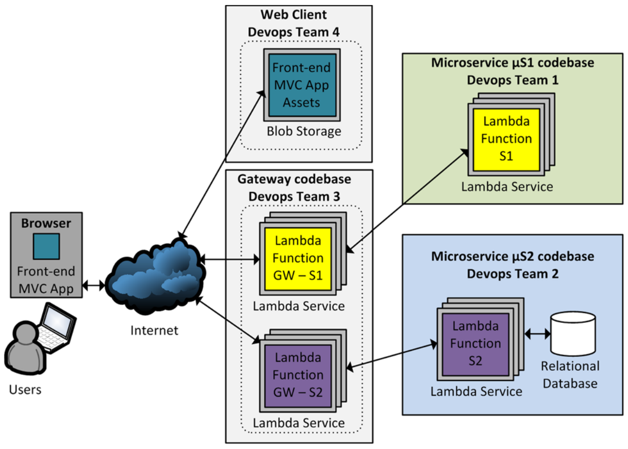

Fig. 5 Deployment of the AWS Lambda architecture

Functions are the unit of execution/development in AWS Lambda, and they can be developed in Java 8, Python, or Node.js. In the context of a web application, a function is a REST/JSON service. Based on the latter, each microservice (μS_1 and μS_2) can be implemented as a function that exposes a REST service, which is consumed by gateway functions.

In contrast with the microservice architecture, where both gateway services were implemented in the same web application, in AWS Lambda, the two gateway services must be implemented as two independent functions that receive requests from end-users (browsers) through the Internet consume microservice functions through REST, get the results from microservice functions, and return the results to end-users. Each gateway function uses the Internet to publish a REST service, which is consumed by the MVC front-end application. Although we separate the functions according to their role to facilitate the comparison with the microservice architecture operated by the cloud customer, in gateway and microservice functions, in AWS Lambda every function is a microservice so Gateway functions must also be considered as microservices.

The AWS Lambda architecture is shown in Fig. 5. The message interchange protocol used between browsers and gateway functions, and between gateway functions and microservice functions, is JSON. Gateway functions and the μS_1 function do not store any information, so they do not need a persistence layer, while the μS_2 function does require persistence. Given that the MVC front-end application must be retrieved by browsers when users access the application, this application is stored in a blob storage system with capacity to respond to HTTP/HTTPS requests. When a user accesses the application, the static assets (HTML, CSS, and JavaScript) are downloaded to the web browser from the blob storage system in the first request, and subsequent requests are sent to the gateway functions through REST. The blob storage system was not required by the monolithic architecture and the microservice architecture operated by the cloud customer, because in those architectures, the MVC front-end application is retrieved from web servers and the gateway, respectively.

The three 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 another for the front-end application. The microservice architecture operated by the cloud customer and the microservice architecture operated by AWS Lambda were developed by four small teams: a team developing the gateway, another the μS_1 microservice, another the μS_2 microservice, and a last one the front-end application.

4 Implementation

We implemented the monolithic architecture with two technological stacks, in order to compare and get a baseline performance. The selected stacks were Play web framework [24] with Java (Play applications can be developed using Java or Scala) and Jax-RS [25]. These frameworks were chosen because they provide a lightweight, stateless, and cloud-friendly architecture. Play and Jax-RS applications are executed in embedded servers, Netty and Jetty, respectively, which are designed to start their execution within seconds and to consume low computing resources.

It is important to highlight that the monolithic and microservice 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; nonetheless, the goal of this work is to compare how the infrastructure costs are affected with the development of each architecture. The implementation of microservice architectures using other frameworks may change some technical details; however, one of the benefits of using microservices is the ability of using multiple technological stacks, so that the architectures, deployments, and results generated in this paper can be used as a reference to implement microservices or gateways in other frameworks.

Given that the gateways in microservice architectures are mainly sending REST requests to microservices, it is important that they implement non-blocking I/O REST libraries [26], which means that their threads do not block their execution while waiting for a response from the microservices. The Play web framework was developed implementing non-blocking mechanisms in its service stack (unlike Jax-RS); this is the reason why the microservice architecture operated by the cloud customer was implemented using Play. The AWS Lambda architecture was implemented developing functions with Node.js [27], one of the programming languages supported by AWS Lambda.

4.1 Monolithic architecture development

The monolithic architecture in Play/Java 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 that was used is PostgreSQL 9.3.6. and the Object Relational Mapping (ORM) employed was Ebeans.
- **Front-end application**. This application was developed using Angular.js 1.3.14 (HTML5, CSS, and jQuery) due to its support by Google.

The monolithic architecture in Jax-RS/Java was implemented as two independent applications:

- **Web application**. This application was developed using JAX-RS (Jersey 1.8) and Java 1.7.0. The relational database that was used is PostgreSQL 9.3.6. and the ORM employed was JPA 2.0/Hibernate.
- **Front-end application**. This application was developed using Angular.js 1.3.14 (HTML5, CSS, and jQuery).

4.2 Microservice architecture development

The microservice architecture operated by the cloud customer 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 used relational database was PostgreSQL 9.3.6 and the employed ORM was Ebeans.
- **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).

4.3 AWS Lambda architecture development

The microservice architecture operated by AWS Lambda was implemented as four independent functions of type microservice-http-endpoint and an MVC front-end application:

- **Microservice μS_1 function**. This function was developed using Node.js 0.10.25.
- **Microservice μS_2 function**. This function was developed using Node.js 0.10.25. The relational database that was used is PostgreSQL 9.3.6, and the access to the database was implemented using the pg npm module.
- **Gateway S_1 function**. This function was developed using Node.js 0.10.25.
- **Gateway S_2 function**. This function was developed using Node.js 0.10.25.
- **Front-end application**. This application was developed using Angular.js 1.3.14 (HTML5, CSS, and jQuery).

5 Deployment in a cloud computing infrastructure

In order to compare the infrastructure costs of running each architecture, the three architecture implementations were deployed in AWS. We started the comparison by defining a baseline architecture for the monolithic architecture. Then, we executed the performance tests described in Sect. 6, in order to calculate the number of requests per minute that the monolithic architecture supported, in both stacks (Play and Jax-RS). Based on that performance, we defined a similar infrastructure for the deployment of the microservice architecture, with the goal of supporting a similar number of requests per minute. Finally, we executed individual tests to identify the best configuration to execute each function in AWS Lambda. We describe the deployment and infrastructure services used for each architecture below.

5.1 Monolithic architecture deployment

The monolithic architecture in Play and Jax-RS was deployed as shown in Fig. 6. The two applications—in Play and Jax-RS—were deployed as follows:

- **Web application**. Each web application, in Play and Jax-RS, was deployed in four Elastic Compute Cloud (EC2) instances of c4.large type (2 vCPUs, 8 ECUs, and 3,75GB RAM), and used the Netty web server 3.7.0 and the

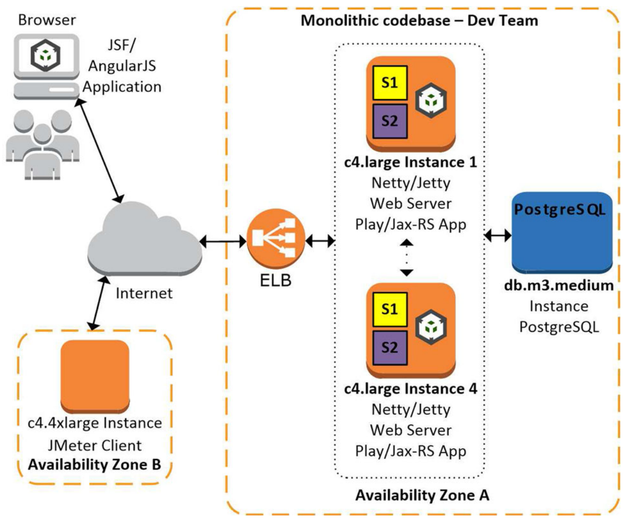

Fig. 6 Deployment of the monolithic architecture on Amazon Web Services

Jetty web server 7.6.0, respectively. For the PostgreSQL database, we used the AWS Relational Database Service (RDS) with Single-AZ in a db.m3.medium instance type (1 vCPU and 3,75GB RAM). Load balancing among the multiple web servers was configured with the Elastic Load Balancer (ELB), a service provided by AWS. Other instance types may have been used; however, performance comparisons with different instance types are beyond the scope of this paper. The services of the web application were exposed through the Internet.

- **Front-end application**. The static files of the Angular.js application (views, models, and controllers) were stored in the Netty and Jetty web servers. When a user accesses the web application, the assets of Angular.js are downloaded to the browser from the web server in the first request; then, the REST services exposed by the web server are consumed from the Angular.js application.

5.2 Microservice architecture deployment

The microservice architecture operated by the cloud customer was deployed as illustrated in Fig. 7. The four applications were deployed as follows:

- **Microservice μS_1 application**. This Play web application was deployed in three EC2 instances of c4.large type (2 vCPUs, 8 ECUs, and 3,75GB RAM) and using the Netty web server 3.7.0. An ELB was used to balance the load among the multiple web servers. The REST services exposed by the μS_1 microservice were configured to be accessible only by the gateway.

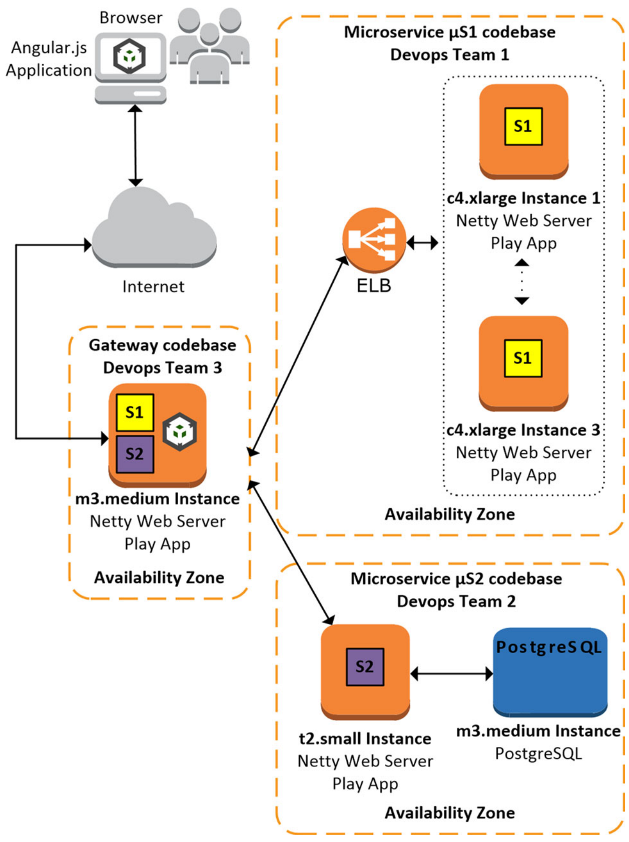

Fig. 7 Deployment of the microservice architecture on Amazon Web Services

- **Microservice μS_2 application**. This Play web application was deployed in one EC2 instance of t2.small type (1 vCPUs, Variable ECUs, and 2,0GB RAM) and using the Netty web server 3.7.0. The PostgreSQL database was deployed in RDS with a db.m3.medium instance type (1 vCPU and 3,75GB RAM). The REST service exposed by the μS_2 Microservice was also configured to be accessible only by the gateway.
- **Gateway application**. This Play web application was deployed in an EC2 instance of m3.medium type (1 vCPUs, 3 ECUs, and 3,75GB RAM) and using the Netty web server 3.7.0. The REST services of the gateway were exposed through the Internet.
- **Front-end application**. Similar to the monolithic application, the static files of the Angular.js application were stored in the gateway. When a user accesses the web application, the assets of Angular.js are downloaded to the browser from the gateway in the first request; then, the REST services exposed by the gateway are consumed from the Angular.js application.

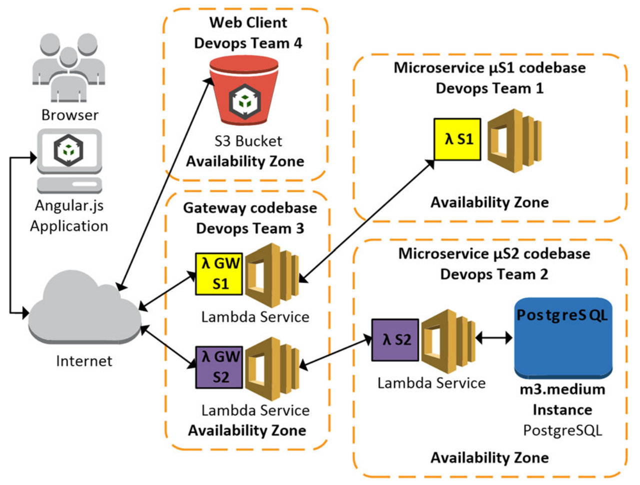

Fig. 8 Deployment of the Amazon Web Services Lambda architecture

5.3 Deployment of the AWS Lambda architecture

The microservice architecture operated by AWS Lambda was deployed as shown in Fig. 8. The four independent microservice-http-endpoint type functions and the MVC front-end application were deployed as follows:

- **Microservice μS_1 function**. This function was deployed in a configuration of 512MB RAM and it was only accessible from gateway functions.
- **Microservice μS_2 function**. This function was deployed in a configuration of 320MB RAM, and it was only accessible from gateway functions. The PostgreSQL database was deployed in RDS with a db.m3.medium instance type (1 vCPU and 3,75GB RAM).
- **Gateway S_1 function**. This function was deployed in a configuration of 512MB RAM, and it was exposed through the Internet.
- **Gateway S_2 function**. This function was deployed in a configuration of 320MB RAM, and it was exposed through the Internet.
- **Front-end application**. The static files of the Angular.js application were stored in a Simple Storage Service (S3) bucket provided by AWS. When a user accesses the web application, the assets of Angular.js are downloaded to the browser from S3 in the first request; then, the REST services exposed by the gateway functions are consumed from the Angular.js application.

6 Test and results

The development of the case study allowed us to compare the infrastructure costs and performance of each architecture by executing different stress tests. The corresponding results are described in this section.

6.1 Performance tests

In order to test and compare the performance and infrastructure costs of the three architectures, we defined three business scenarios. The first scenario determined that 20% of the requests made to the web application consumed service S_1 and 80% consumed service S_2. We called this the 20/80 scenario. In the second scenario, each service received 50% of the requests; therefore, it is called the 50/50 scenario. In the third scenario, called 80/20, 80% of the requests consumed service S_1 and 20% consumed service S_2.

To execute the stress tests of each architecture, we configured JMeter [28] 2.13 in an AWS c4.large EC2 instance, and we defined the maximum response time of S_1 and S_2 in 20.000 and 3.000 ms, respectively (their typical response times are 3.000 and 300 ms, respectively). In addition, JMeter was configured to execute a predefined number of requests per minute, with services S_1 and S_2 simulating a constant workload, depending on the scenario tested throughout 10 min.

During the performance tests to the monolithic and the microservice architecture operated by the cloud customer (Figs. 6, 7), the applications servers (Netty/Jetty) were configured with a pool of several threads to attend several requests concurrently according to the computing capabilities available to the instances. In the performance test to AWS Lambda (Fig. 8), this was not necessary because each request was executed in an isolated environment.

In order to test the performance of the monolithic architecture, we defined the base infrastructure shown in Fig. 6, whose monthly infrastructure costs of \$403.20 USD are shown in Table 1. Costs associated with bandwidth, storage, and backups were not taken into account, as they were the same in the three architectures. All costs were calculated using the pricing list available in September 2015 for the AWS US East (N. Virginia) Region.

The stress tests were executed with the goal of identifying the maximum number of requests supported by the monolithic architecture in Play and Jax-RS. This number was calculated by increasing the number of requests for each scenario until the application began to generate errors or the response time defined for S_1 and S_2 was not met. The results of the performance tests executed for the monolithic architecture are shown in Fig. 9. These results show that, when considering the monolithic architecture, Jax-RS provides a better performance than Play.

Based on the number of requests per minute supported by the monolithic architecture, we executed performance tests for the microservice architecture operated by the cloud customer, in order to identify the minimum infrastructure needed

Table 1 Infrastructure costs of the monolithic architecture on AWS

Service	Cost per hour (USD)	Quantity per month	Cost per month (USD)
Web application. EC2 Instance c4.large	0.110	720*4	316.80
Web application. RDS Instance db.m3.medium with Single AZ	0.095	720*1	68.40
Web application. ELB Instance	0.025	720*1	18.00
Monthly infrastructure costs			403.20

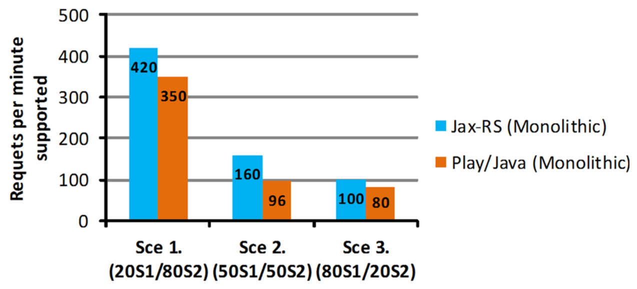

Fig. 9 Performance results of the monolithic architecture in Play and Jax-RS on AWS

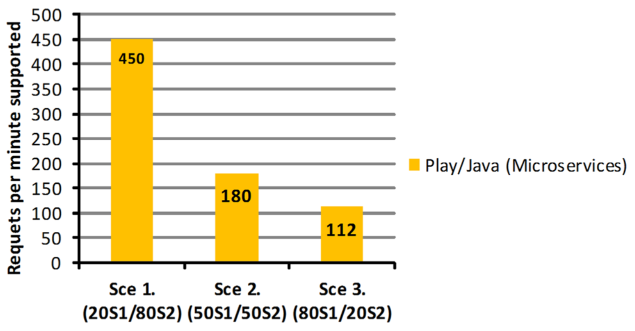

Fig. 10 Performance results of the microservice architecture in Play on AWS

for supporting the same number of requests. The execution of such tests allowed us to identify the most suitable instance type used for the μS_1 microservice, the μS_2 microservice, and the gateway. The identified infrastructure is illustrated in Fig. 7, and its monthly cost of $390.96 USD is given in Table 2.

The results of the performance tests for the microservice architecture are illustrated in Fig. 10. These results show that, in our case study, microservices can provide an even better performance than Jax-RS does in the monolithic architecture, at a lower cost. The monthly cost of the microservice architecture was $390.96 USD in contrast to the $403.20 USD of the monolithic architecture; furthermore, the former supported more requests per minute in the three defined scenarios.

Similarly, the costs in AWS Lambda are based on the number of requests and their configuration per executed microservice/gateway function. Therefore, the performance tests and cost estimation performed for the deployment shown in Fig. 8 were defined to support the same number of requests per minute as the microservice architecture. The costs involved during the performance tests in AWS Lambda to support the same number of requests per minute as the microservice architecture are summarized in Table 3. These results show that AWS Lambda supports the same number of requests per minute as the microservice architecture, at a lower cost in the three defined scenarios, and involves monthly costs of $193.81, $173.26, and $168.08,

Table 2 Infrastructure costs of the microservice architecture on AWS

Service	Cost per hour (USD)	Quantity per month	Cost per month (USD)
Microservice μS_1. EC2 Instance c4.large	0.110	720*3	237.60
Microservice μS_1. ELB Instance	0.025	720*1	18.00
Microservice μS_2. EC2 Instance t2.small	0.026	720*1	18.72
Microservice μS_2. RDS Instance db.m3.medium with Single AZ	0.095	720*1	68.40
Gateway. EC2 Instance m3.medium	0.067	720*1	48.24
Monthly infrastructure costs			$390.96

Table 3 Infrastructure costs of the AWS Lambda architecture

Description	Scenario 20/80	Scenario 50/50	Scenario 80/20
Total number of requests	450	180	112
μS_1 and Gateway S_1 functions. Number of requests per minute	90	90	90
μS_1 and Gateway S_1 functions. Memory configuration (MB)	512	512	512
μS_1 and Gateway S_1 functions. Requests per month	3,888,000	3,888,000	3,888,000
μS_1 and Gateway S_1 functions. Response time per request (ms)	3000	3000	3000
μS_1 and Gateway S_1 functions. Total computing seconds	11,664,000	11,664,000	11,664,000
μS_1 and Gateway S_1 functions. Total computing (GB/s)	5,832,000	5,832,000	5,832,000
μS_2 and Gateway S_2 functions. Number of requests per minute	360	90	22
μS_2 and Gateway S_2 functions. Memory configuration (MB)	320	320	320
μS_2 and Gateway S_2 functions. Requests per month	15,552,000	3,888,000	950,400
μS_2 and Gateway S_2 functions. Response time per request (ms)	300	300	300
μS_2 and Gateway S_2 functions. Total computing seconds	4,665,600	1,166,400	285,120
μS_2 and Gateway S_2 functions. Total computing (GB/s)	145,800	364,500	89,100
μS_1 and S_1 Gateway functions. Cost per number of requests	$0.78	$0.78	$0.78
μS_1 and S_1 Gateway functions. Cost per computing duration	$97.22	$97.22	$97.22
μS_2 and S_2 Gateway functions. Cost per number of requests	$3.11	$0.78	$0.19
μS_2 and S_2 Gateway functions. Cost per computing duration	$24.30	$6.08	$1.49
μS_2 Microservice. RDS Instance db.m3.medium with Single AZ	$68.40	$68.40	$68.40
Monthly costs (USD)	**$193.81**	**$173.26**	**$168.08**

respectively, while the monthly cost in microservices is $390.96.

6.2 Cost comparison

Given that each architecture was deployed in different infrastructures, we defined and calculated the metric *Cost per Million of Requests (CMR)* for each architecture in the three scenarios, in order to easily compare their execution costs. For each scenario and architecture, this metric was calculated by dividing the monthly infrastructure costs by the number of requests supported per month, which is calculated by multiplying the number of requests supported per minute by 43,200—the number of minutes per month (60*24*30). We assumed a constant throughput per minute during a month. The CMR metric for each architecture is shown in Fig. 11.

These results show that the microservice architecture implemented with Play can help reduce up to 9.50, 13.81, and 13.42% of the costs per scenario in contrast to the monolithic architecture implemented with Jax-RS. However, the use of services exclusively designed to implement microservices, such as the third architecture implemented in AWS Lambda, can help reduce up to 50.43, 55.69, and 57.01% of the costs per scenario in contrast to the microservice architecture; up to 55.14, 61.81, and 62.78% of the costs per scenario in contrast to the monolithic architecture implemented in Jax-RS; and up to 62.61, 77.08, and 70.23% of the costs per scenario in contrast to the Play monolithic architecture.

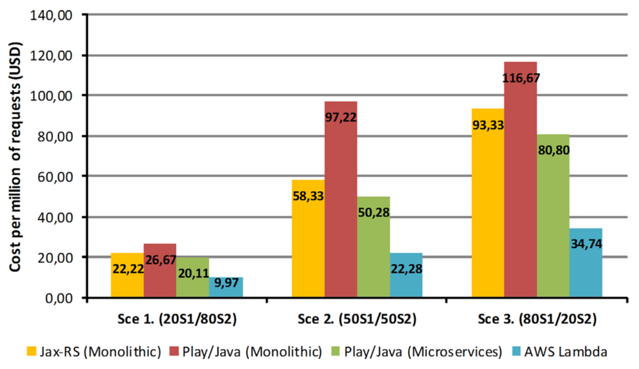

Fig. 11 Cost comparison of the three architectures per million of requests

6.3 Response time

To identify how each architecture affects the response time to requests during peak periods, we measured the average response time (ART) during the performance tests. The ART for S1 and S2 are shown in Figs. 12 and 13, respectively. The ART for S_1 and S_2 in the monolithic architectures with Play and Jax-RS are similar; however, when considering the microservice architecture, the ART for both services increases because each request must pass between the gateway and each microservice. For the microservice architecture operated by AWS Lambda, the ART remains very similar for both services in the three scenarios, even though it var-

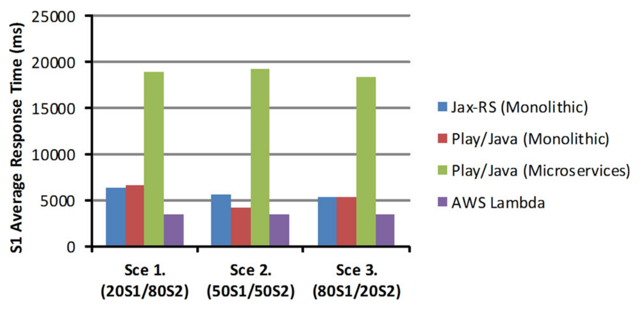

Fig. 12 Average response time for S_1 during peak periods

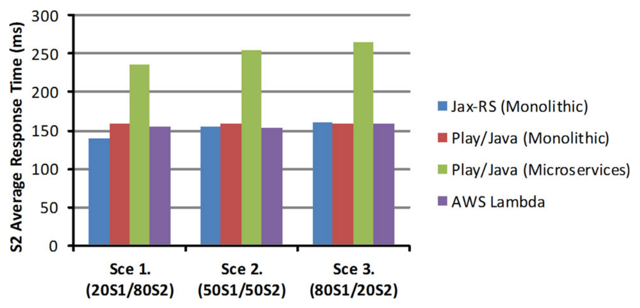

Fig. 13 Average response time for S_2 during peak periods

ied between scenarios for the other architectures. This result was obtained because each function/request in AWS Lambda is executed in an isolated computing environment dedicated to a single request, while each web server executes several concurrent requests in the other architectures.

6.4 Development methodology and efforts

With the development of the case study, we could validate that microservice architectures (including AWS Lambda) require a change in the way that most companies and software vendors have traditionally been developing monolithic applications. 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 business and technical requirements, which makes mandatory the definition of company guidelines to avoid the use of technologies that can be difficult to manage. Therefore, the documentation of REST services exposed by microservices/gateways gains importance in order to enable services to be used by multiple teams.

During the development process, we also confirmed that when microservices are implemented, the development and deployment of several independent applications make the development and testing processes more complex; Developer and operation teams need to identify and resolve problems related to distributed systems that can affect several independent and decentralized services (e.g. failures, timeouts, distributed transactions, data federation, responsibility assignments, service versioning). Problems that are solved by application containers (e.g. JBOSS, Glassfish, WebLogic, IIS) in monolithic applications or by ESBs in SOA implementations must be managed at application level when considering microservices.

One of the problems identified during the use of AWS Lambda is that each function must be independently developed and deployed, which makes the per-function development and deployment processes difficult to manage. Projects such as Jaws [29] are trying to create tools to facilitate these processes. Additionally, the use of services such as AWS Lambda are inherent to a particular vendor by nature, so companies must evaluate the lock-in problems that they may face when using them in the short and long term.

6.5 Deployment, scaling, and continuous delivery

The deployment of a microservice architecture in AWS required the deployment of several independent applications (microservices and gateways), each one requiring specific configurations in cloud infrastructures. When new gateway or microservice versions are published, it is easy to break external services, which highlights the importance of maintaining service versioning and plan among multiple teams, and of defining the upgrade process to avoid breaking problems.

The use of ready-to-use services specifically designed for microservice deployment, such as AWS Lambda, helps save time and money in infrastructure management tasks, because a subset of the implementation of continuous delivery and DevOps (Development + Operation) strategies are assumed by the cloud provider. However, if microservices are implemented without using this type of services, the use of those strategies can be a time-consuming task, due to the repetitive execution of manual tasks in each deployment. In those cases, the use of automation tools is mandatory in order to save time and gain agility. To deploy and scale microservices and gateways, the use of lightweight/embedded servers—such as Netty or Jetty, which can be started in seconds—is recommended. These servers do not share states and can be added or removed from clusters and load balancers at any time.

One of the concerns identified during performance tests of microservice architectures is the inability to monitor the flow of a request made by an end-user through gateways and multiple microservices, this continues to be a challenge. However, the greatest benefit of using microservices is the ability to scale each microservice/gateway independently by using different policies. At a business level, this may represent large savings in IT infrastructure costs and a more efficient way to take advantage of the pay per use benefits of the cloud model.

6.6 Adoption, business culture, and guidelines

Based on the case study, microservice architectures should be employed in businesses that need to scale their applications to hundreds of thousands or millions of users. However, their implementation requires additional abilities that are not present in many companies. The adoption of microservice architectures requires a new culture of development and innovation that must be complemented by a set of guidelines and good practices at a company level. Accordingly, 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 incrementally developed.

7 Discussion

This paper focuses on comparing the cost of developing and deploying the same application using three different architectures and deployment models with the goal of identifying how different architectures and deployments can affect the infrastructure costs of running and scaling an application in the cloud. In order to get valid results, we had to implement three versions of the application and deploy each one using the appropriate services on AWS.

After the process of developing, deploying, and testing the different architectures, we could determine and analyse the different technical and culture challenges required to use microservice architectures. To estimate the costs of running each architecture, we experimented with different performance tests. We also defined the Cost per Million of Requests (CMR) metric to be able to compare the cost of the different architectures. During the performance tests, the ART metric allowed us to conclude that cloud services specifically designed to deploy microservices, such as AWS Lambda, can maintain the same response time even when the number of users increases.

7.1 Tests on other cloud providers

Although the efforts of the paper were focused on comparing the costs on AWS, the process, architecture, and deployment models used can be replicated on other cloud providers. The architectures, experiments, metrics, and challenges could be used as a guide to test the same architectures on other cloud providers. For example, the three architectures presented in this paper, and their corresponding performance tests and cost comparison, could be implemented on other providers such as Windows Azure, Google Cloud, or IBM. The results of such a test could facilitate the cost comparison of running each architecture across different cloud providers as well as help to identify the pros and cons that must be taken into account to deploy and scale each architecture on other cloud providers.

7.2 Emerging microservice/serverless cloud services

Although AWS Lambda was the first cloud service specifically designed to run microservices, the popularity of such services roots from their ability to scale applications without cloud customers having to face the challenges of running and scaling application servers, which has resulted in other cloud providers launching similar services.

Google Cloud launched the Alpha release of the Google Cloud Functions service [30], which is defined as a lightweight compute solution for developers to create single-purpose, stand-alone functions that respond to Cloud events without the need to manage a server or runtime environment. Windows Azure launched the Preview release of the Azure Functions service [31], which is defined as a Serverless event driven experience that extends the existing Azure App Service platform. These nano-services can scale based on demand and you pay only for the resources you consume. And finally, IBM launched the IBM Bluemix OpenWhisk service [32], which they describe serves to Execute code on demand in a highly scalable environment without servers.

A comparison regarding the cost of running applications on those services might be quite interesting. Such a comparison requires the third application presented in this paper to be adapted according to the requirements, development framework, and restrictions of each service.

7.3 Implementing microservices using Serverless Frameworks

It is very important to consider that applications developed and deployed on cloud services such as AWS Lambda are highly coupled to the solution of each provider; therefore, the lock-in when these solutions are used is very high. Some efforts such as the Serverless Framework [33] are directed towards creating frameworks that facilitate the creation of microservices or serverless functions (another name given to this type of services), which can be easily deployed on services such as AWS Lambda, Google Cloud Functions, Azure Functions, or IBM Bluemix OpenWhisk.

The third application developed in this paper, specifically for AWS Lambda, could be redeveloped using the Serverless Framework to facilitate its deployment on serverless services of other cloud providers. The same performance tests, cost comparison, metrics, and results could be executed, identified, and analysed for each cloud provider. Tests on other cloud providers would help the research community and companies to easily compare the costs of running microservices applications on different cloud providers, and to identify other interest topics that must be considered to deploy scalable applications on emerging serverless services.

The test on other cloud providers and also the use of the Serverless Framework are proposed as future work.

8 Conclusion and future work

Based on the performance tests executed for a monolithic architecture, a microservice architecture operated by the cloud customer, and a microservice architecture operated by AWS Lambda, we can conclude that the use of emerging cloud services such as AWS Lambda, exclusively designed to deploy microservices at a more granular level (per HTTP request/function), allows companies to reduce their infrastructure costs in up to 77.08%. Microservices also enable large applications to be developed as a set of small applications that can be independently implemented and operated, thus managing large codebases by using a more practical methodology, where incremental improvements are executed by small teams on independent codebases. Agility, cost reduction, and granular scalability must be balanced with the development efforts, technical challenges, and costs incurred by companies resulting from microservices requiring the adoption of new practices, processes, and methodologies.

Furthermore, the case study allows us to conclude that 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 the reviewed practical cases, most applications using microservice architectures started as monolithic applications and were incrementally modified to implement microservices due to scaling problems at infrastructure and team management levels.

As future work, we will evaluate the architectures with a greater number of services with the goal of testing the impact of implementing other type of services such as memory, I/O, and network-intensive services. We will also evaluate the costs of running microservices on serverless services from providers such as Google, Windows Azure, and IBM, as well as the costs incurred by companies during the process of implementing microservices.

The process of defining the number of microservices to implement, and the tools required to automate the deployment of microservices in general purpose IaaS solutions and services such as AWS Lambda, will also be evaluated. Tests to evaluate other technical concerns regarding microservices, such as performance analyses across different components involved in microservices, failure tolerance, distributed transactions, data distribution, service versioning, and microservice granularity, are also interesting research areas.

The cost and performance comparison to migrate legacy monolithic stateful application to stateless cloud services such as AWS Lambda is another future work area, where different topics such as the efforts and challenges to redesign, rearchitect, reimplement, and redeploy the applications are very interesting to analyse.

References

1. Buyya R (2010) Cloud computing: the next revolution in information technology. In: 2010 1st international conference on parallel distributed and grid computing (PDGC), pp 2–3
2. Vosshall P (2008) Web scale computing: the power of infrastructure as a service. In: Bouguettaya A, Krueger I, Margaria T (eds) Service-oriented computing ICSOC 2008. Lecture notes in computer science, vol 5364. Springer, Heidelberg, pp 1–1
3. Beimborn D, Miletzki T, Wenzel S (2011) Platform as a service (PaaS). Bus Inf Syst Eng 3(6):381–384
4. Schtz S, Kude T, Popp K (2013) The impact of software-as-a-service on software ecosystems. In: Herzwurm G, Margaria T (eds) Software business. From physical products to software services and solutions. Lecture notes in business information processing, vol 150. Springer, Berlin, pp 130–140
5. Lewis J, Fowler M (2014) Microservices. http://martinfowler.com/articles/microservices.html. Accessed 23 Apr 2017
6. Amazon Web Services (2015) AWS Lambda. https://aws.amazon.com/lambda/. Accessed 23 Apr 2017
7. McGovern J, Sims O, Jain A, Little M (2006) Understanding service-oriented architecture. In: Enterprise service oriented architectures. Springer Netherlands, pp 1–48. doi:10.1007/1-4020-3705-8_1
8. La H, Bae J, Chang S, Kim S (2007) Practical methods for adapting services using enterprise service bus. In: Baresi L, Fraternali P, Houben G-J (eds) Web engineering. Lecture notes in computer science, vol 4607. Springer, Berlin, pp 53–58
9. Papazoglou M, Traverso P, Dustdar S, Leymann F (2007) Service-oriented computing: state of the art and research challenges. Computer 40:38–45
10. Hutchinson J, Kotonya G, Walkerdine J, Sawyer P, Dobson G, Onditi V (2007) Evolving existing systems to service-oriented architectures: perspective and challenges. In: IEEE international conference on web services 2007, ICWS 2007, pp 896–903
11. GIGAOM (2011) The biggest thing Amazon got right: the platform. https://gigaom.com/2011/10/12/419-the-biggest-thing-amazon-got-right-the-platform/. Accessed 23 Apr 2017
12. Nginx (2015) Adopting microservices at Netflix: lessons for architectural design. http://nginx.com/blog/microservices-at-netflix-architectural-best-practices/. Accessed 23 Apr 2017
13. InfoQ (2014) Scaling Gilt: from monolithic ruby application to distributed scala micro-services architecture. http://www.infoq.com/presentations/scale-gilt. Accessed 23 Apr 2017
14. InfoQ (2015) From a monolith to microservices + REST: the evolution of LinkedIn's service architecture. http://www.infoq.com/presentations/linkedin-microservices-urn. Accessed 23 Apr 2017
15. SoundCloud (2014) Building products at SoundCloud—Part I: Dealing with the monolith. https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-1-dealing-with-the-monolith. Accessed 23 Apr 2017
16. Thones J (2015) Microservices. IEEE Softw 32:116
17. InfoQ (2014) Microservices and SOA. http://www.infoq.com/news/2014/03/microservices-soa. Accessed 23 Apr 2017
18. Oracle (2015) Microservices and SOA. http://www.oracle.com/technetwork/issue-archive/2015/15-mar/o25architect-2458702.html. Accessed 23 Apr 2017

19. TechTarget (2015) How microservices bring agility to SOA. http://searchcloudapplications.techtarget.com/feature/How-microservices-bring-agility-to-SOA. Accessed 23 Apr 2017
20. Microservices (2014) Pattern: microservices architecture. http://microservices.io/patterns/microservices.html. Accessed 23 Apr 2017
21. Vinoski S (2007) REST eye for the SOA guy. IEEE Internet Computing 11:82–84
22. Nemeth F, Steinert R, Kreuger P, Skoldstrom P (2015) Roles of DevOps tools in an automated, dynamic service creation architecture. In: 2015 IFIP/IEEE international symposium on integrated network management (IM), pp 1153–1154
23. Bass L, Clements P, Kazman R (2012) Software Architecture in Practice, 3rd edn. Addison-Wesley Professional, Boston
24. Hunt J (ed) (2014) Play framework. In: A beginner's guide to scala, object orientation and functional programming. Springer, Berlin, pp 413–428
25. Juneau J (ed) (2013) Building RESTful web services. In: Introducing Java EE 7. Apress, New York, pp 113–130
26. Venkatesan V, Chaarawi M, Gabriel E, Hoefler T (2011) Design and evaluation of nonblocking collective I/O operations. In: Cotronis Y, Danalis A, Nikolopoulos D, Dongarra J (eds) Recent advances in the message passing interface. Lecture notes in computer science, vol 6960. Springer, Berlin, pp 90–98
27. Doglio F (ed) (2015) Node.js and REST. In: Pro REST API development with Node.js. Apress, New York, pp 47–63
28. Rahmel D (ed) (2013) Testing a site with ApacheBench, JMeter, and Selenium. In: Advanced Joomla!. Apress, New York, pp 211–247
29. InfoWorld (2015) Jaws takes a bite out of AWS Lambda app deployment. http://www.infoworld.com/article/2990795/cloud-computing/jaws-takes-a-bite-out-of-aws-lambda-app-deployment.html. Accessed 23 Apr 2017
30. Google (2016) Google Cloud Functions. https://cloud.google.com/functions/. Accessed 23 Apr 2017
31. Microsoft (2016) Azure Functions. https://azure.microsoft.com/en-us/services/functions/. Accessed 23 Apr 2017
32. IBM (2016) IBM Bluemix OpenWhisk. http://www.ibm.com/cloud-computing/bluemix/openwhisk. Accessed 23 Apr 2017
33. Serverless (2016) Serverless Framework. https://serverless.com/. Accessed 23 Apr 2017