Seminar Report

On

Monolithic v/s Microservices Architecture

By

Kshitish Deshpande

1032170436

Under the guidance of

Dr. Shilpa S. Paygude

MIT-World Peace University (MIT-WPU)

Faculty of Engineering

School of Computer Engineering & Technology

*** 2020 - 2021 ***

MIT-World Peace University (MIT-WPU)

Faculty of Engineering

School of Computer Engineering & Technology

CERTIFICATE

This is to certify that Mr. Kshitish Deshpande of B.Tech., School of Computer Engineering & Technology, Trimester – IX PRN. No. 1032170436, has successfully completed seminar on

Monolithic v/s Microservices Architecture

To my satisfaction and submitted the same during the academic year 2020 - 2021 towards the partial fulfillment of degree of Bachelor of Technology in School of Computer Engineering & Technology under Dr. Vishwanath Karad MIT- World Peace University, Pune.

Dr. Shilpa S. Paygude

Seminar Guide

Prof. Dr. M.V.Bedekar

Head
School of Computer Engineering & Technology

LIST OF FIGURES

Figures	Page
1. Deployment of the Monolithic Architecture	12
2. Deployment of Monolithic Architecture on Amazon Web Services	13
3. Infrastructure costs of the monolithic architecture on AWS	14
4. Deployment of Microservices Architecture	15
5. Infrastructure costs of the microservice architecture on AWS	16
6. Deployment of Microservices using Amazon Web Services	17
7. Performance results of the monolithic architecture in Java on AWS	18
8. Performance results of the monolithic architecture in Java on AWS	19
9. Average response time with the monolithic and microservice architecture	19
10. 90%-line response time with the monolithic and microservice architecture	20
11. Basic Structure of the Monolithic application in case study	22
12. Basic Structure of the Microservices based application in case study	23
13. Comparison of latency of Microservices and Monolithic architectures	25
14. Comparison of successful throughput of Monolithic and Microservices	26
15. Comparison table of the error rates of Monolithic and Microservices	27

ABBREVIATIONS

Sr No.	Abbreviation	Explanation
1.	IaaS	Infrastructure as a Service
2.	PaaS	Platform as a Service
3.	SaaS	System as a Service
4.	HTML	Hyper Text Markup Language
5.	REST	Representational state transfer
6.	API	Application Programming Interface
7.	AWS	Amazon Web Services
8.	ec2	Elastic Cloud Compute
9.	SOA	Service-oriented architecture
10.	ESB	Enterprise Service Bus

ACKNOWLEDGEMENT

With immense pleasure I, Mr. Kshitish Deshpande of B.Tech., School of Computer Engineering & Technology, Trimester – IX PRN. No. 1032170436 express my sincere thanks to Prof. Dr. M V. Bedekar, Head of School of Computer Engineering and Technology, for providing all the necessary resources required for the completion of my seminar.

I express my profound thanks to my seminar guide, Dr. Shilpa S. Paygude for her valuable suggestions and guidance in the preparation of seminar report.

I am also grateful to all those who have indirectly guided me and helped in the preparation of seminar.

Kshitish Deshpande
PB31
Panel-2

Index

ABSTRACT 7

1. Introduction 8

2. Literature Survey 10

3. Monolithic and Microservices Architecture on Amazon Web Services Infrastructure (IaaS) 12

3.1 Monolithic Architecture 12

3.2 Microservices Architecture 14

3.3 Tests and results 17

4. Monolithic v/s Microservices Architecture on bare metal servers 21

4.1 Application Specifications 21

4.2 Hardware used in the case study 23

4.3 Test Methodology 23

4.4 Test Results 25

5. Inferences and Conclusion 28

REFERENCES 30

PLAGIARISM CHECK 31

ABSTRACT

Having come into light just a few years ago, microservices are an accelerating trend these days. Indeed, microservices approach offers tangible benefits including an increase in scalability, flexibility, agility, and other significant advantages. Netflix, Google, Amazon and other technology giants have moved effectively from monolithic architecture to microservices. In the meantime, several businesses consider following this example to be the most productive way to develop business. The monolithic approach is on the contrary a default paradigm for designing a software application. Even so, the trend is decreasing because developing a monolithic framework presents many challenges associated with managing a large code base, introducing a modern technology, scaling, launching, incorporating new changes and more. The monolithic solution, then, is obsolete and should be left in the past? And is the entire program worth moving from a monolith to microservices? Is creating an application for microservices going to help you achieve your business goals? This paper aims to examine the benefits and inconveniences of an architecture of microservices over a monolithic architecture.

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

1. Introduction

For many engineering frameworks, managing a developing degree of vulnerability brings about an expansion in framework intricacy and a large group of new difficulties in planning and architecture of such frameworks. These frameworks need to react to a lot of changes in the market, innovation, administrative scene, and spending accessibility. Changes in these components are obscure to the frameworks' planner during the plan stage, yet in addition during prior stages, for example, idea improvement and prerequisites investigation, of the framework's life cycle. The capacity to manage a significant level of vulnerability converts into higher design adaptability in building frameworks, which empowers the framework to react to varieties even more quickly, with less expense, or less effect on the framework adequacy.

When used with the cloud computing model, Microservices architecture allows organizations to deploy software solutions as different separate microservices which (when well-engineered) can scale their computation resources exclusively on demand. Organizations may either send their own applications to Infrastructure as a Service (IaaS) or Platform as a Service (PaaS) arrangements or buy applications ready to use under the Software as a Services (SaaS) model. When organizations deploy their own applications on IaaS or PaaS frameworks, they experience different difficulties in utilizing the distributed computing services, like auto scaling, persistent conveyance, hot organizations, high accessibility, dynamic screening, etc. Most organizations moving an application into an IaaS / PaaS system usually tried moving a monolithic application. A monolithic application in this study means an application with a single massive codebase / store providing tens or hundreds of services using different interfaces, such as HTML pages, Web services or / and REST services.

Scaling monolithic applications is a challenge as they typically provide multiple services, some of which are more common than others. If common services need to be scaled because they are highly requested, the entire range of services would also be simultaneously scaled, ensuring that unpopular services consume vast quantities of server resources even though they are not going to be used.

From the development approach, most businesses deploying applications to the cloud also need to be able to innovate as quickly as possible due to their competition, if they do not innovate with improved product offerings and new functionality will harm their company. That is why the word "continuous delivery" has been popular over the past few years mainly in startups, large Internet companies and SaaS providers. The continuous delivery methodology allows companies to continuously change and update their applications in production using agile methodologies and cycles of development.

All services are built in monolithic systems on a common codebase shared by multiple developers; when these developers choose to implement or modify services, they have to guarantee that all other services will continue to function. Complexity increases as more services are introduced which restrict companies' ability to innovate with new versions and features. Furthermore, as new program versions are introduced for production, the entire collection of services is restarted, creating poor experiences for consumers who use them. Even a monolithic implementation reflects a single failure point; if the application fails the entire number of programs is going down.

Despite the move towards microservice architecture in the industry as a whole, studies on system-level driving forces and cost / benefit analysis of moving from monolithic to microservice architecture remained scarce. These studies are critical for decision models determining the net utility of migration to distributed systems. The underlying structural driving forces and trade-offs of moving from monolithic to microservice architecture are similar in nature to those for moving from centralized to modular architecture.

This paper tries to explore the advantages and disadvantages of moving from a monolithic to microservices architecture by comparing their performance, infrastructural costs of deployment and difficulties in development in multiple case studies from existing literature with the usage of Amazon Web Services as both an IaaS provider (ec2 VM instances) and a PaaS provider (Amazon Lambda).

2. Literature Survey

Fowler discusses the variations between a monolithic architecture and an architecture of microservices [6]. It is important to consider the distinctions between such architectures to consider what our research is about. In the article by Fowler he discusses the architecture of microservice as a suite of small services. Each of these services run its own process and interact with the other services using a lightweight interface such as an HTTP API. Fowler proceeds to describe the monolithic architecture as a single entity, where business applications are typically divided into three parts: a client user application, a server-side application, and a database. A monolithic system consists of a single logical executable, the server-side application. He continues to clarify the differences between a microservice architecture and a monolithic architecture in his paper, with the most significant differences being:

- There are multiple processes in a microservices architecture while a monolithic architecture has just one process.
- A microservice can be built independently of other microservices whereas a single framework with classes, functions, and namespaces is built as a monolithic architecture.
- A microservice will scale independently of other microservices while the entire framework must scale in the monolithic architecture.
- A microservice can be modified and implemented without the use of other microservices, while the whole framework needs to be implemented in the monolithic architecture.

Research is also exploring the issues and impacts of splitting into many microservices a monolithic program. Knoche writes that breaking up a monolithic structure involves careful preparation and could have a significant effect on performance [7]. Knoche offers a gradual eight-step strategy to maintain output while breaking down a monolithic structure into microservices. The approach involves simulating performance to see what the actual performance is by using the microservice-based architecture. Knoche introduces numerous ways of applying the new architecture based on microservices.

Additionally, Balalaie et al. present insights and lessons learned while transitioning incrementally from a database to a microservice architecture [8]. They write about how they switched from a monolithic pipeline to a pipeline of microservices that is separate for each application, so that they can be deployed independently. Changing from monolithic to microservices offers a multiple cross-functional team choice that makes DevOps.

We find findings from our literature analysis that indicate containers have no or remarkably little performance deficit compared to running on physical hardware. We find out in all studies relating to microservices and virtual machines that microservices are better than a conventional virtual system. We conclude that this means that there is no clear image of the real effect microservices have on results relative to monolithic structures. We have seen the performance rely on a variety of different factors such as programming language, environment, database technology, container technology and system architecture. In the next two sections two of the most informative case studies are highlighted.

3. Monolithic and Microservices Architecture on Amazon Web Services Infrastructure (IaaS)

From the case study of Villamizar, M., Garcés, O., Ochoa, L., Castro, H., Salamanca, L., Verano, M., ... Lang, M. [3] and Mario Villamizar, Oscar Garcés, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas [5]

3.1 Monolithic Architecture

The traditional monolithic architecture of Play / Java and Jax-RS / Java was deployed on bare metal servers.

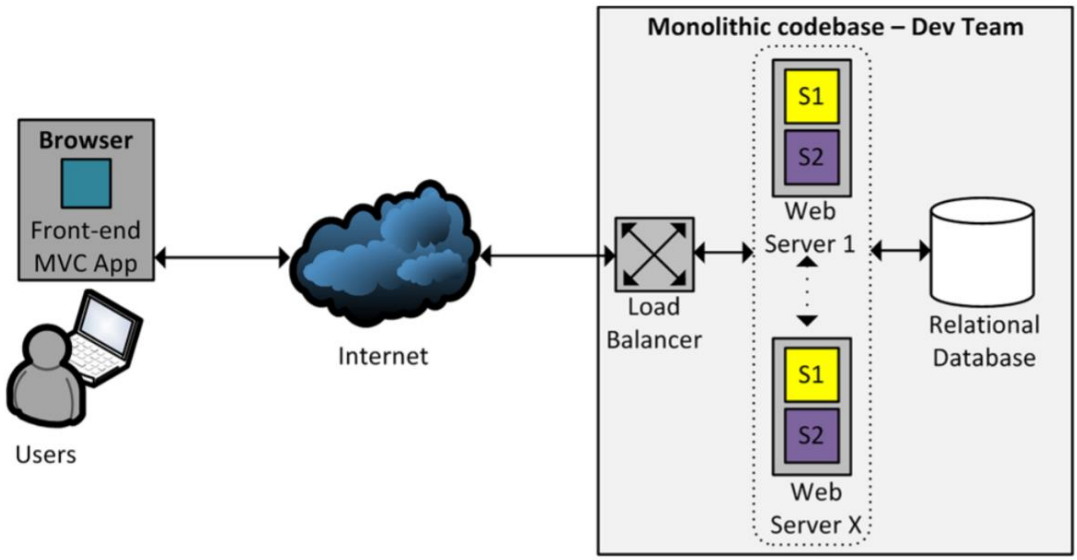


Fig. 1 Deployment of the Monolithic Architecture

Both the Monolithic architecture in Play / Java and Jax-RS / Java were implemented as two separate applications – a web server with 2 services using PostgreSQL 9.3.6 relational database and a frontend framework developed using Angular.js 1.3.14 (HTML5, CSS, and jQuery). The Play / Java web application was built using Play 2.2.2, Scala 2.10.2 and Java 1.7.0, and the Object Relational Mapping (ORM) used was Ebeans, while the Jax-RS / Java web application was built using JAX-RS (Jersey 1.8) and Java 1.7.0, and the ORM used was JPA 2.0 / Hibernate. 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 they 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. Load balancing among the multiple web servers was configured with the Elastic Load Balancer (ELB), a service provided by AWS.
- 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.

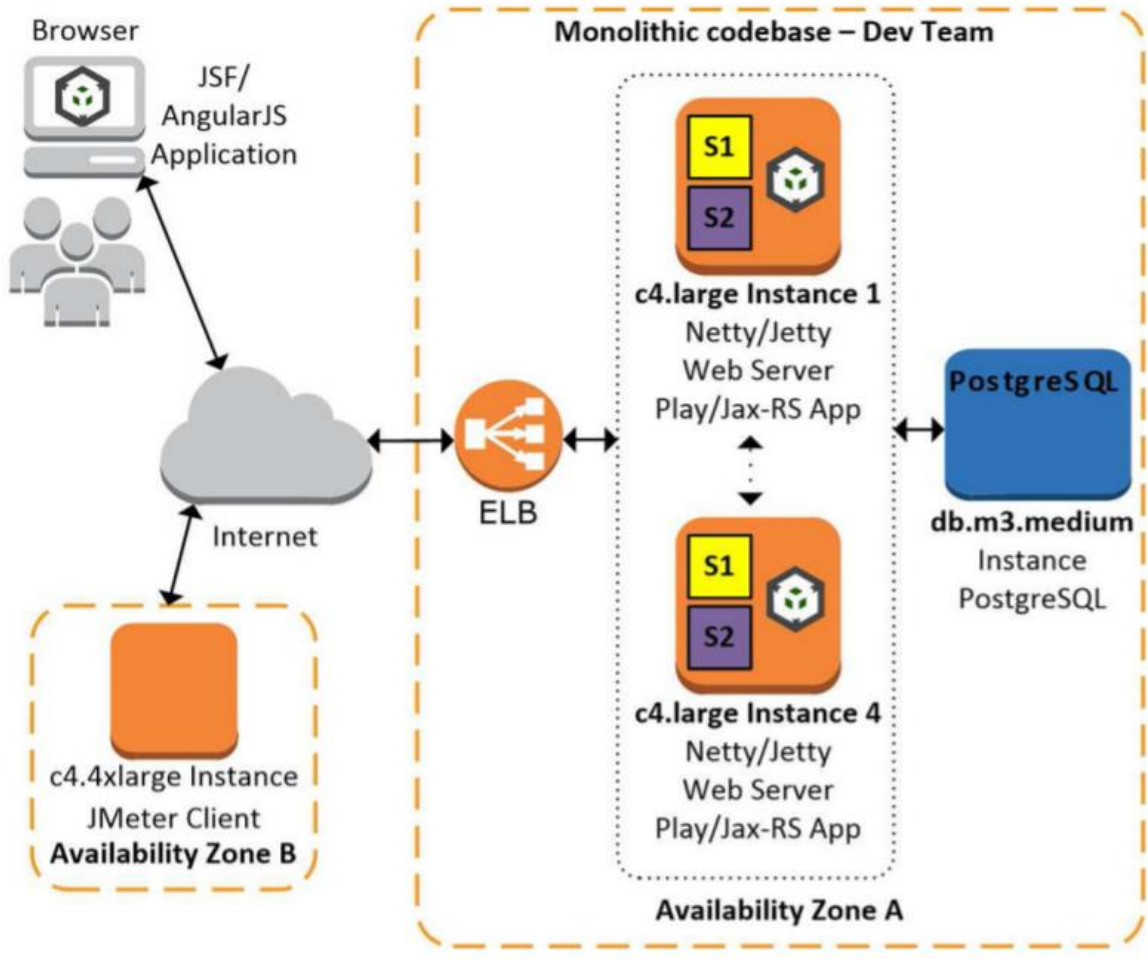


Fig. 2 Deployment of Monolithic Architecture on Amazon Web Services

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. 3 Infrastructure costs of the monolithic architecture on AWS

3.2 Microservices Architecture

The microservices architecture using Play/Java was deployed onto bare metal servers. Since the monolithic application consisted of a web application with 2 services, in this case study two microservices (each servicing one of either S1 or S2, the same microservices as those present in the monolithic application) were used along with PostgreSQL relational database. In the case study, since only S2 service required a persistence layer, no database instance was attached to microservice servicing S1. Both microservices expose their main service (S1 and S2) using REST over a private network. The service exposed by each microservice is consumed by the gateway. 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 same frontend application as the one in the monolithic architecture was reused here to let users interact with the microservices.

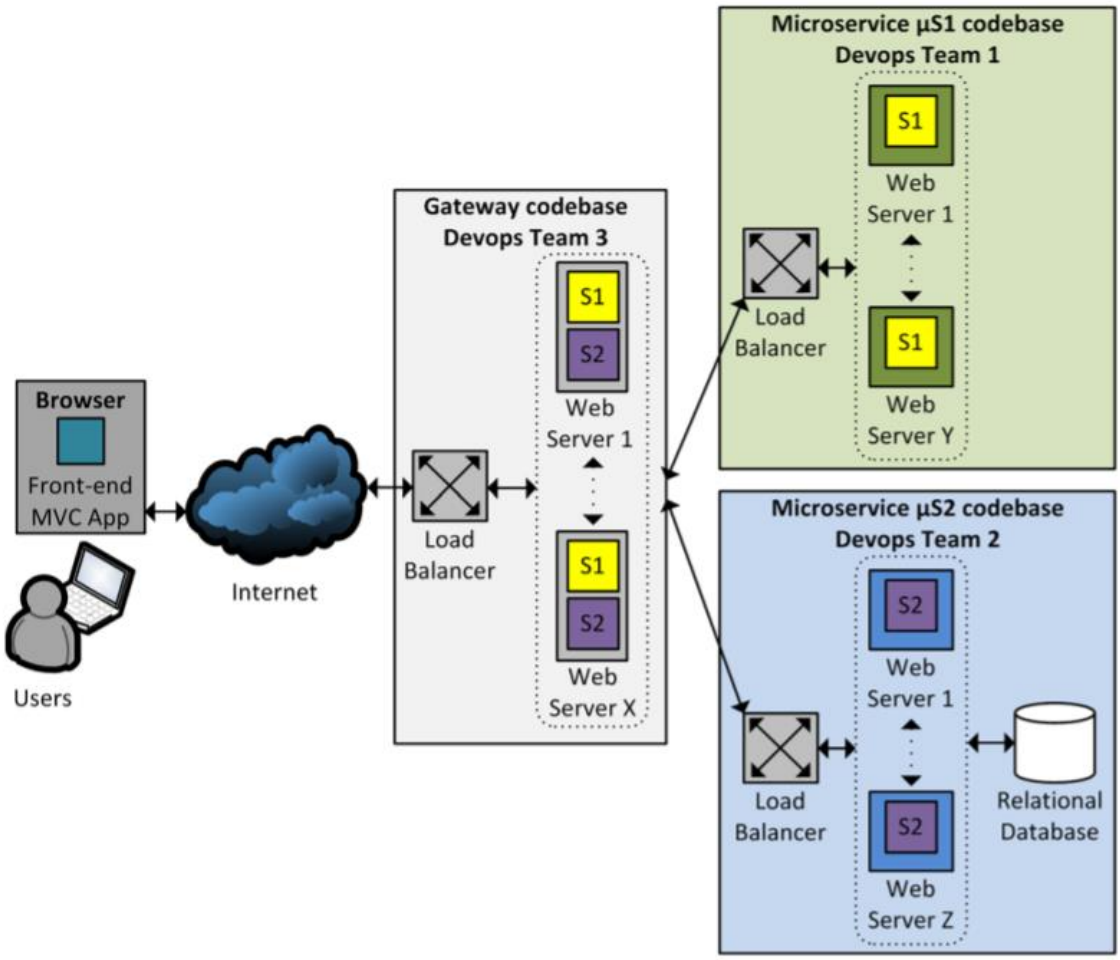


Fig. 4 Deployment of Microservices Architecture

The details of the four independent applications are as follows

- Microservice µS1 application : Crafted using Play 2.2.2, Scala 2.10.2, and Java 1.7.0. It was deployed in three c4.large-type EC2 instances (2 vCPUs, 8 ECUs, and 3,75 GB RAM) and using the 3.7.0 Netty web server. An ELB was used to manage the load between multiple webservers. The REST resources exposed by the µS1 microservice were configured to be available only by the gateway.
- Microservice µS2 application : This application was developed using Play 2.2.2, Scala 2.10.2, and Java 1.7.0, and the employed ORM was Ebeans. This application was deployed in a t2.small style EC2 instance (1 vCPUs, Variable ECUs, and 2.0 GB RAM), using the 3.7.0 Netty web server. The PostgreSQL database was deployed in RDS with an instance type db.m3.medium (1vCPU and

3,75 GB RAM). Also the REST service exposed to the µS2 Microservice was designed to be accessible only via the gateway.

- Gateway application : This application was developed using Play 2.2.2, Scala 2.10.2, and Java 1.7.0. This application was deployed in a m3.medium type EC2 instance (1vCPUs, 3 ECUs, and 3.75 GB RAM), using the 3.7.0 Netty web server. The gateway 's REST services had been revealed via the Internet.
- Front-end application : This application was developed using Angular.js 1.3.14 (HTML5, CSS, and jQuery). 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.

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

Fig. 5 Infrastructure costs of the microservice architecture on AWS

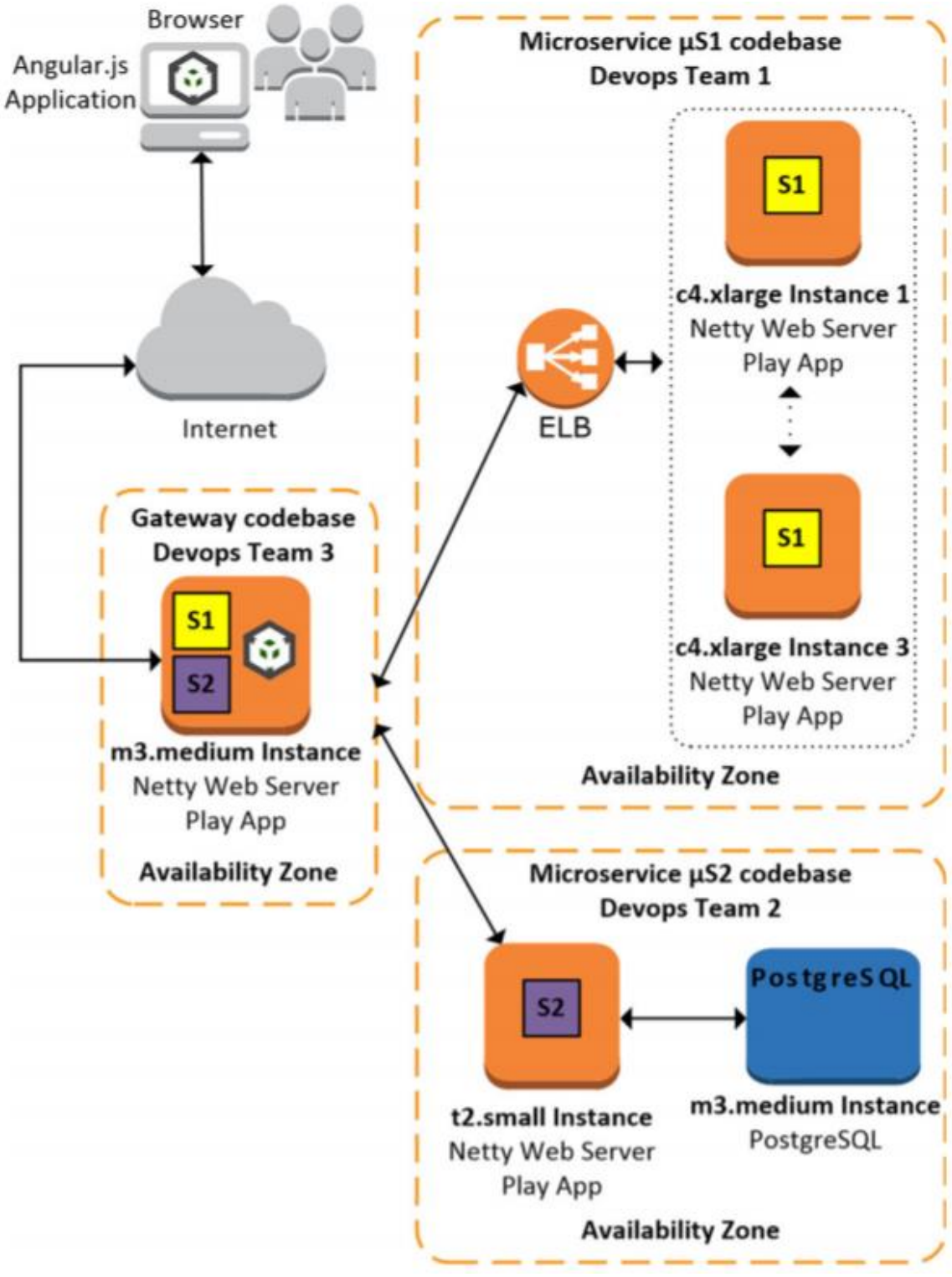


Fig. 6 Deployment of Microservices using Amazon Web Services

3.3 Tests and results

Tests were then performed in three defined scenarios using JMeter 2.13 in an AWS c4.large EC2 instance.

- The first scenario determined that 20% of the requests made to the web application consumed service S1 and 80% consumed service S2.
- In the second scenario, each service received 50% of the requests.
- In the third scenario, called 80/20, 80% of the requests consumed service S1 and 20% consumed service S2.

The total requests per minute supported by the applications (calculated by increasing the number of requests for each scenario until the application began to generate errors or the response time for S1 was over 6,000ms or for S2 was over 1,500ms) were calculated for the above three scenarios for both monolithic and microservices applications.

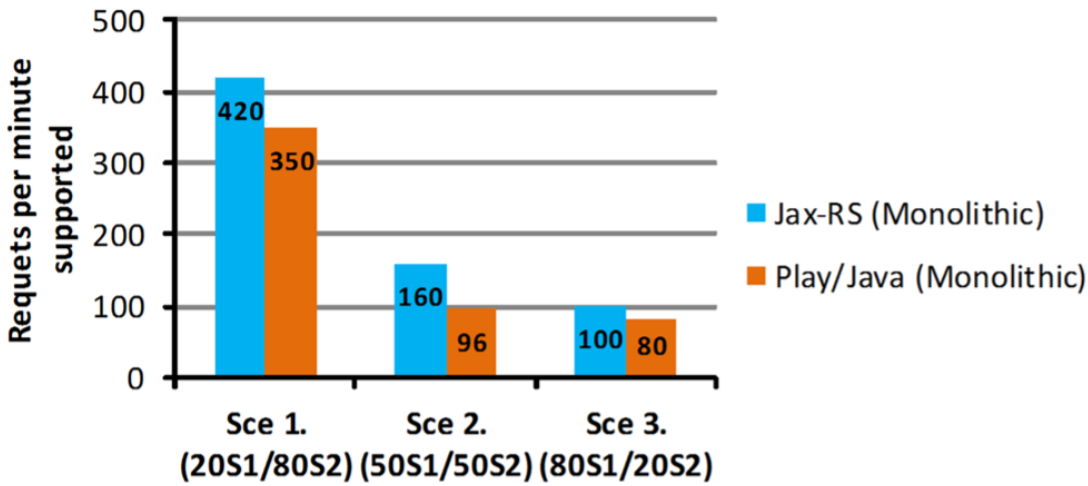


Fig. 7 Performance results of the monolithic architecture in Play and Jax-RS on AWS

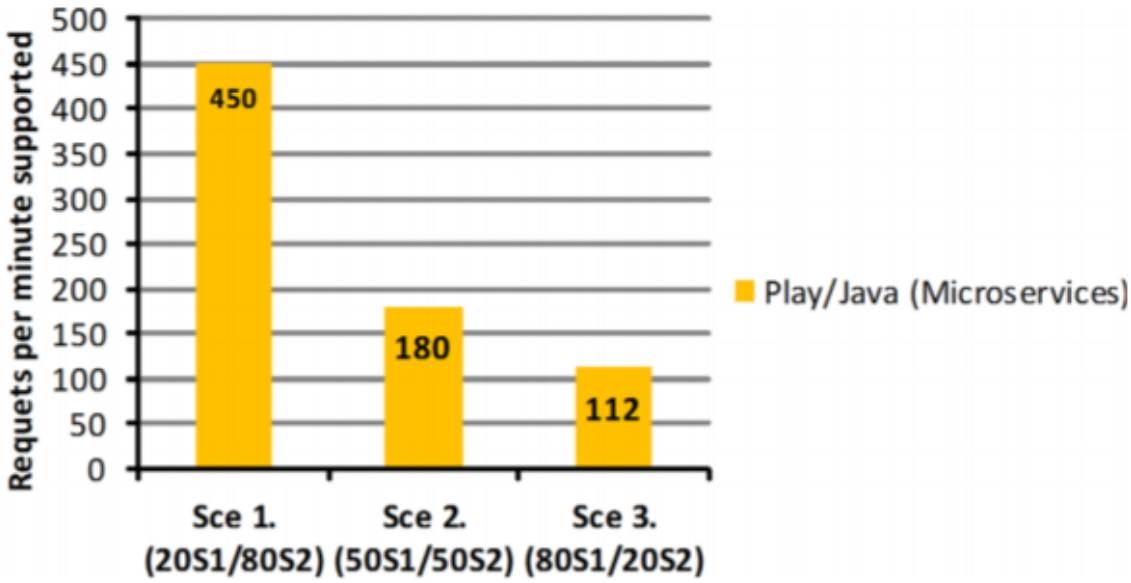


Fig. 8 Performance results of the microservice architecture in Play on AWS

The Average Response time as well as the 90% Line Response time (the value below which 90% of the requests fall) were also calculated for both monolithic and microservices architectures. In case of monolithic, the average time of response of across both the Play/Java and Jax-RS/Java applications was calculated.

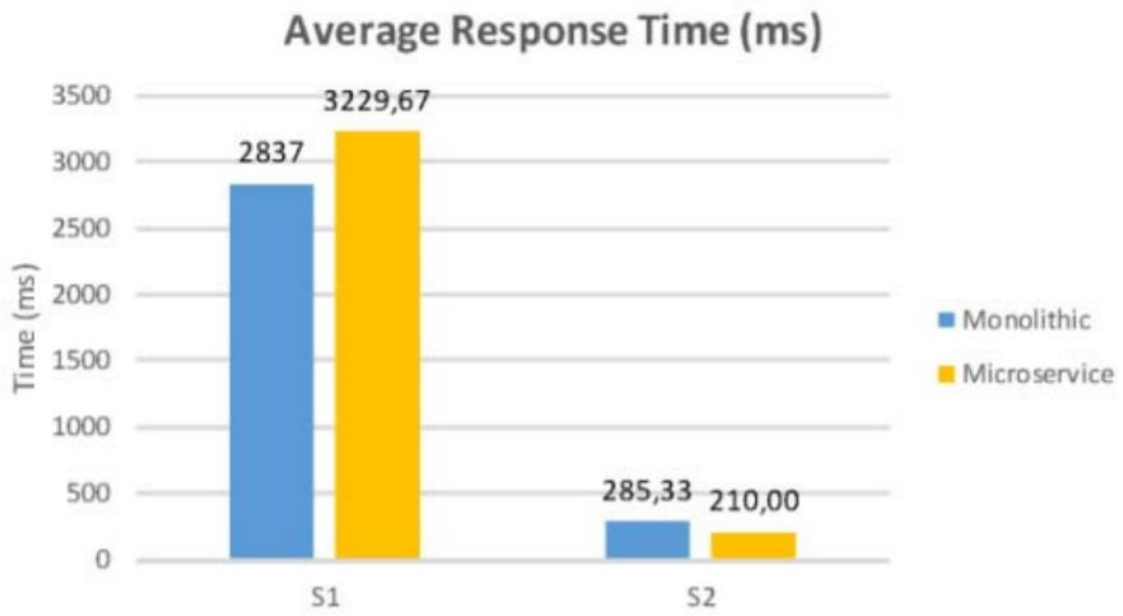


Fig. 9 Average response time with the monolithic and microservice architecture

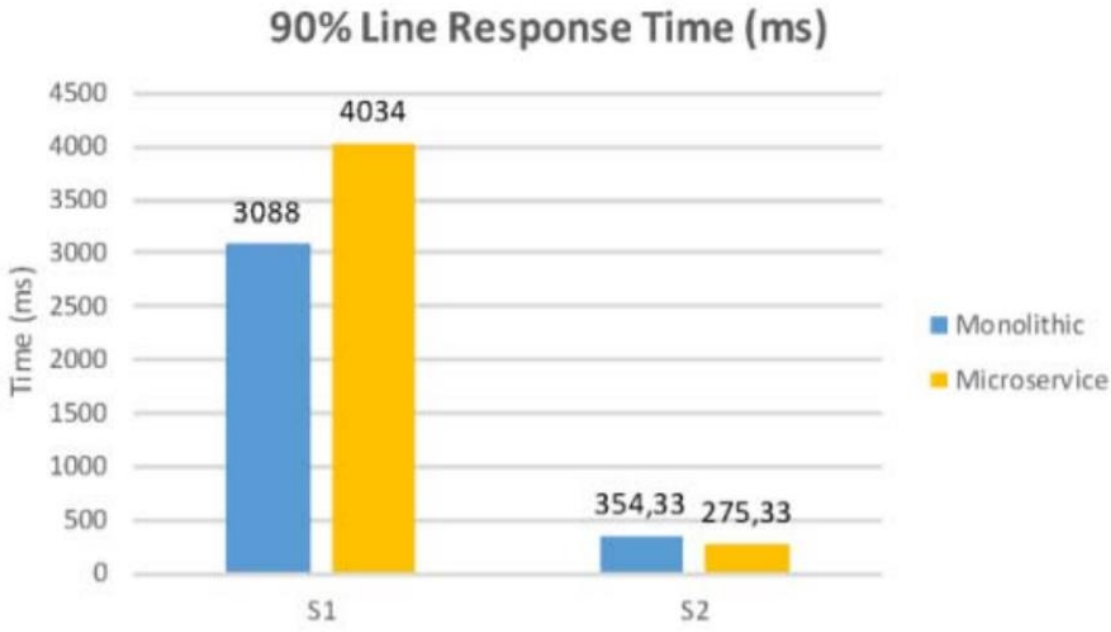


Fig. 10 90% Line response time with the monolithic and microservice architecture

4. Monolithic v/s Microservices Architecture on bare metal servers

From the case study of R. Flygare and A. Holmqvist [4]

4.1 Application Specifications

A complete system was simulated in the environment which included the user interface, REST API, business logic, and database. The microservice-based architecture was divided into four microservices, each running in its own Docker container, while the monolithic framework was divided into three parts (UI, REST API and Server, Database). The user interface was developed using HTML5, CSS3 and jQuery 3.2.0 while the REST system was developed using Scala 2.11.8, Java 8 with JDK 1.8.0, SBT 0.13.15 and Akka HTTP 10.0.5. Cassandra was used for database support, using Spark Cassandra Connector 2.0 and Spark 2.1 to integrate the database with the REST system.

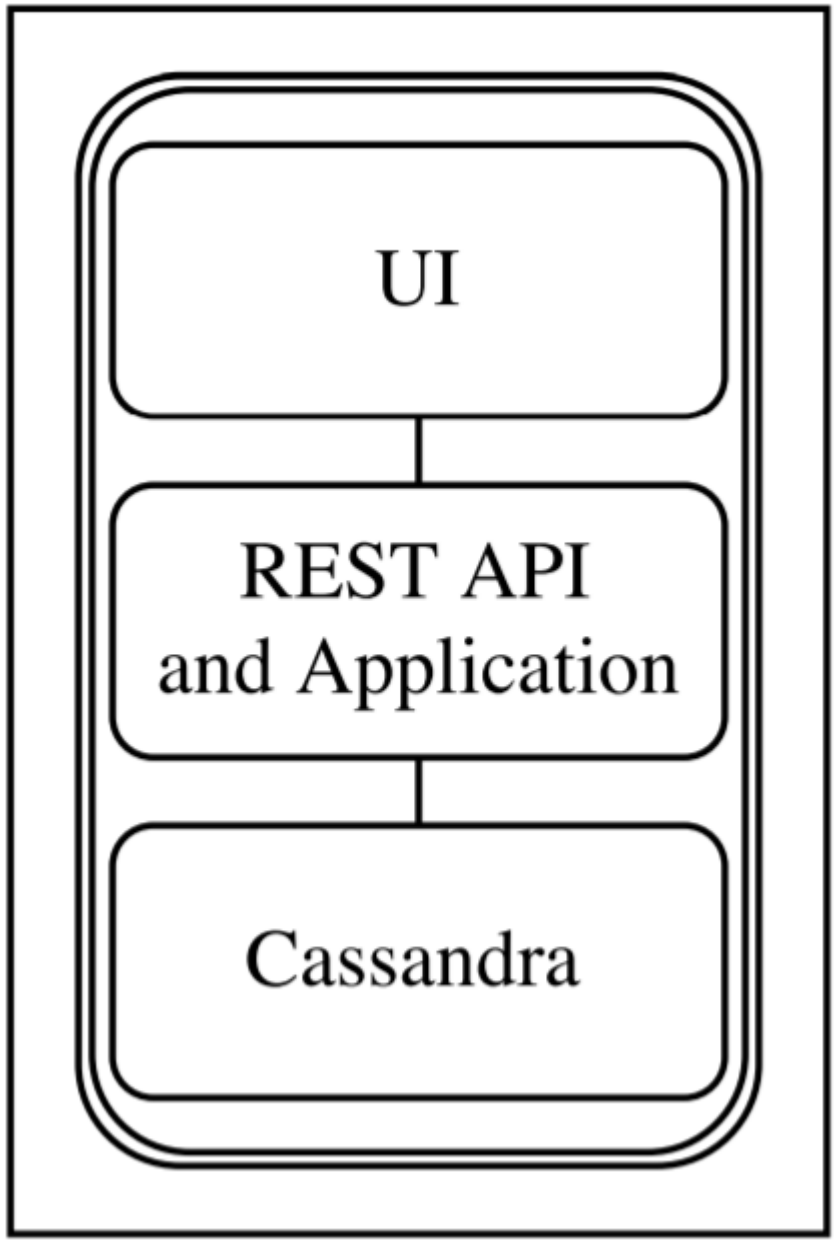


Fig. 11 Basic Structure of the Monolithic application used in the case study

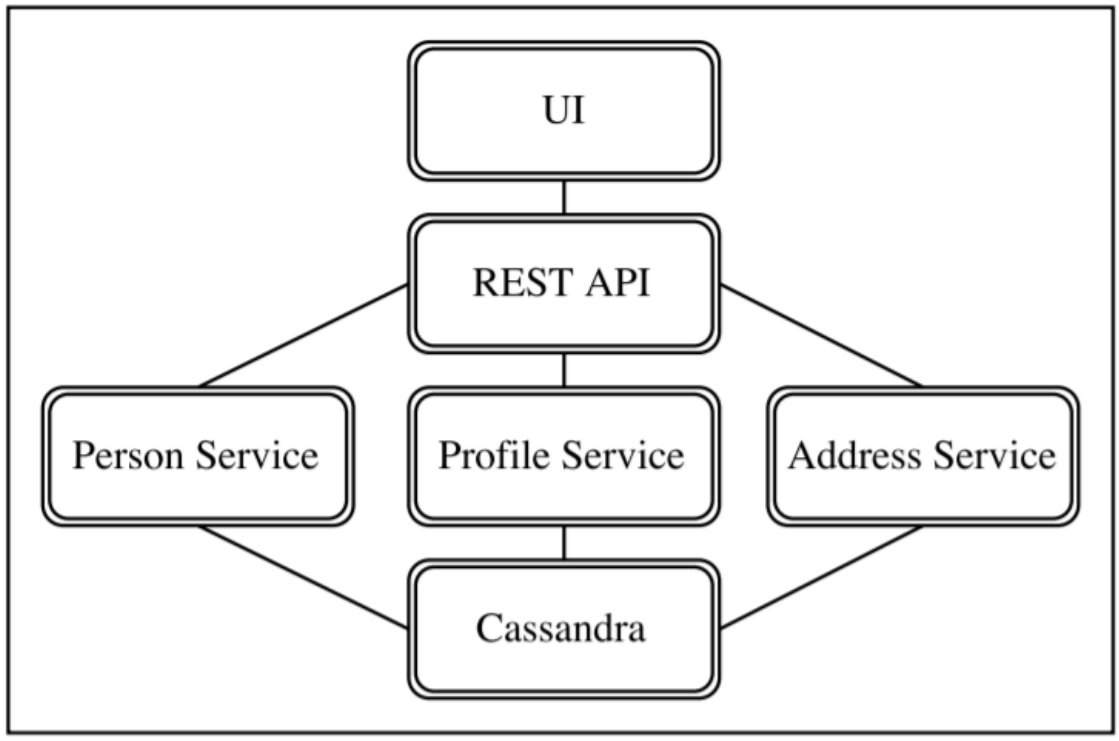


Fig. 12 Basic Structure of the Microservices based application used in the case study

4.2 Hardware used in the case study

Following hardware was used for the experiment

- 2x Intel NUC5i7RYH with 16 GB RAM, Intel Core i7-5557U Processor and Ubuntu Server 16.04 as host operating system. This bare metal server was used to deploy the microservices as well as the monolithic application.
- JMeter executed from a computer on the same local network as the other machines, running Windows 10 pro with an Intel Core i54690K processor and 16GB RAM.

4.3 Test Methodology

For the measurements in CPU and RAM usage, Datadog agent was on both Intel NUC computers, where the agent sent the data to Datadog, where the data was later analyzed. JMeter 3.2 was used for simulating usage and measuring response time, error rate and successful throughput. All the measurements are done in a test plan for JMeter. In the test

plan, a ramp up period of 60 seconds was used to run each test case for two minutes following with a delay of two minutes before starting the next. In the test cases JMeter acts as users interacting with the user interface and makes REST request.

Four different scenarios were described as test cases to be carried out. All these situations related to the creation and retrieval of multiple assets entities. For example, a property may be a first name, a last name, or a phone number. These entities were to be stored on the development layer of persistence and retrieved from it on request. The four scenarios were as follows

- **TC1** Create a Person and fetch a Person (2 properties)
    - 1 simultaneous user
    - 10 simultaneous users
    - 100 simultaneous users
    - 1000 simultaneous users
- **TC2** Create an Address and fetch an Address (5 properties)
    - 1 simultaneous user
    - 10 simultaneous users
    - 100 simultaneous users
    - 1000 simultaneous users
- **TC3** Create a Profile and fetch a Profile (10 properties)
    - 1 simultaneous user
    - 10 simultaneous users
    - 100 simultaneous users
    - 1000 simultaneous users
- **TC4** All of the above test cases concurrently
    - 1 simultaneous user
    - 10 simultaneous users
    - 100 simultaneous users
    - 1000 simultaneous users

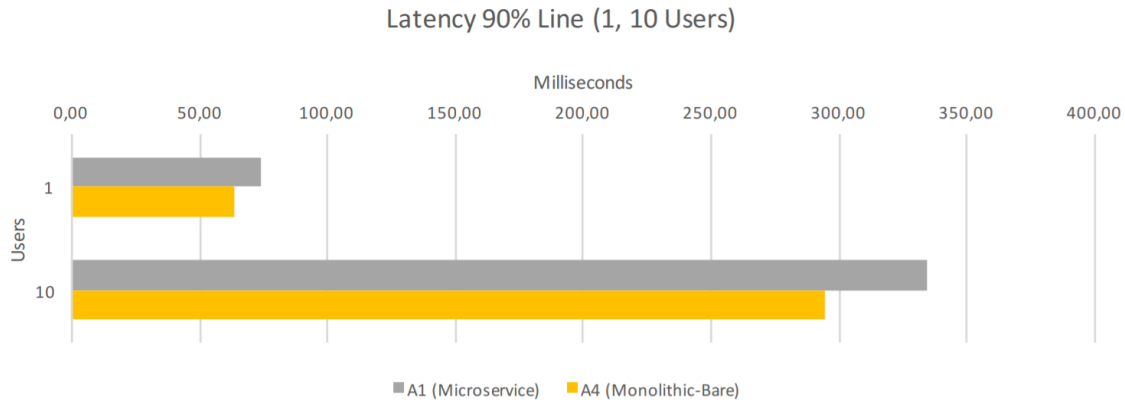
With JMeter 's support, all data was saved to CSV files that were imported into Excel where the data was analyzed and compiled into diagrams and tables. All data was

obtained using JMeter built-in tools. JMeter 's performance aspect readings were average response time, 90%-line response time, error rate, and throughput.

4.4 Test Results

The average CPU use was 46.65 per cent while running the tests on the microservice architecture and the overall memory consumption was 13.42 GiB. On the other hand, the average Processor use was 49.12 per cent while running the tests on the monolithic architecture and the overall memory consumption was 8.07 GiB. This means that the monolithic architecture used 5.36 GiB less RAM compared to the microservices architecture with an improvement of 2.47 percentage points in CPU use.

The data collected revealed that in all test cases, the monolithic application has lower latency than microservices except when there are 1000 users. We see for one user that the microservices application is 14 percent slower, for 10 users we can see that the microservices application is 12 percent slower, for 100 users the microservices application is 31 percent slower, and lastly for 1000 users the microservices application is 130 percent faster.



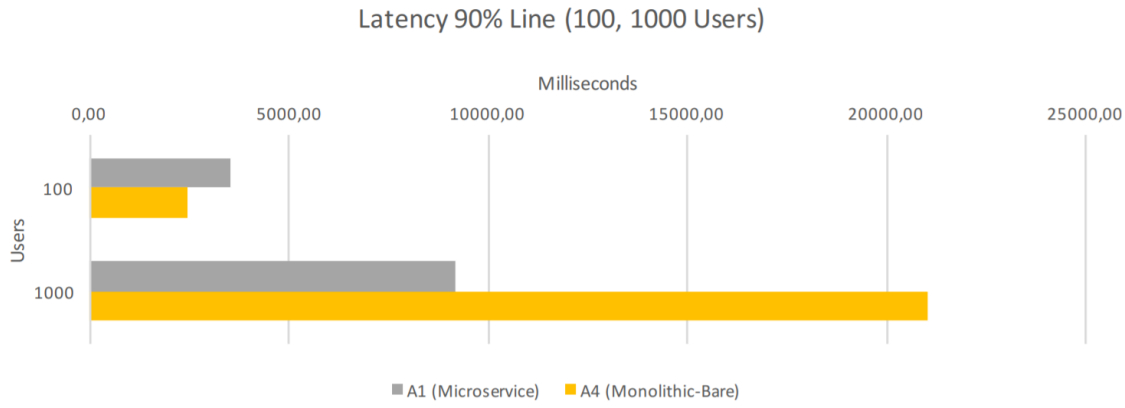
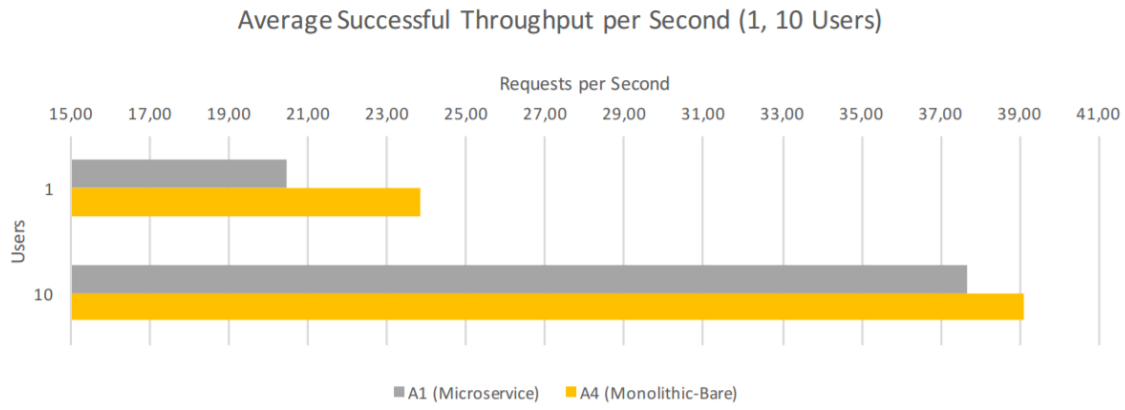


Fig. 13 Comparison of latency of Microservices and Monolithic architectures

In addition, the monolithic framework has a higher throughput rate in all test cases compared to microservices. For one user, the architecture of microservices has a 14 per cent lower success rate, for 10 users a 4 per cent lower, for 100 users a 5 per cent lower and for 1,000 users a 19 per cent lower success rate.



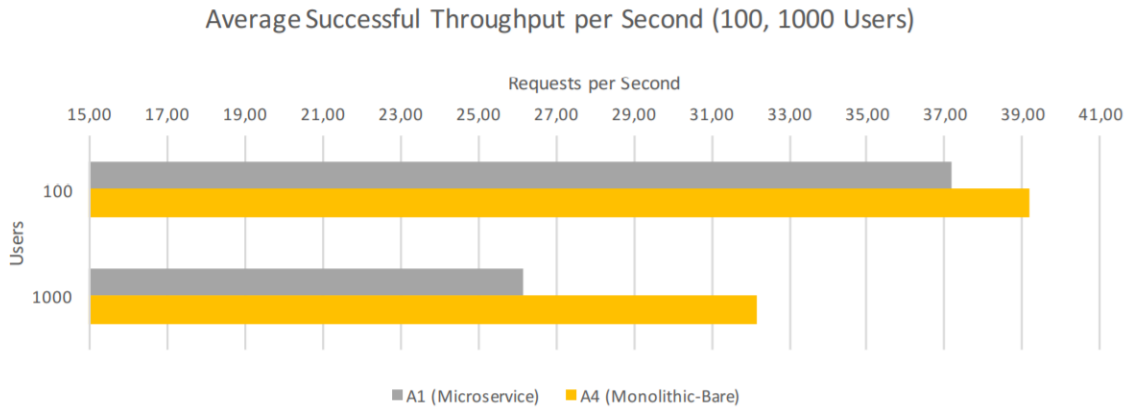


Fig. 14 Comparison of successful throughput of Monolithic and Microservices Architecture

The average error rates were lesser for microservices in all test cases except when there were 1000 simultaneous users. This high error rate in the test case with 1000 simultaneous users was explained as the result of the system not being able to handle the number of parallel users and hence denying all requests immediately. This also explains the difference in results of latency and average successful throughput.

Average Error Rate (%)	1	10	100	1000
A1 (Microservice)	1%	7%	12%	70%
A4 (Monolithic-Bare)	2%	10%	13%	41%

Fig. 15 Comparison table of the error rates of Monolithic and Microservices Architecture

5. Inferences and Conclusion

From the literature reviewed and the two case studies discussed in the previous, we can draw some conclusions as to the differences between Monolithic and Microservices architectures. However, there is a significant amount of research gap that may invalidate our inferences since these are nothing more than our understandings of the literature and case studies reviewed.

Developing applications using microservice architectures needs a shift in the way that most organizations and software vendors have historically designed single-code applications. Microservice architectures allow small teams to work on small applications (microservices) without thinking about how other microservices or teams function. Each team should use various technologies to incorporate microservices / gateways in compliance with business and technological requirements; to prevent the use of a lot of technologies that could be difficult to manage, some guidance should be given to all teams so that they can determine which collection 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.).

Microservices introduces several problems with distributed systems (failures, timeouts, distributed transactions, data federation, assignments of responsibility, etc.) which in some areas make the development process more complex. Many problems that are normally allocated to application containers (JBOSS, Glassfish, WebLogic, IIS) in monolithic applications or addressed by ESB in implementation of SOA must be addressed in microservices at the application level. To deploy and scale microservices and gateways into cloud environments, it is recommended that lightweight / embedded servers such as Netty or Jetty, which provide simple functionality, be started in seconds, do not share state, and can be added or removed from clusters and load balancers at any time.

The bare-metal Monolithic architecture is more resource friendly with higher hardware efficiency compared to the microservice architecture and is faster to some extent but

depends deeply on the number of concurrent users. However, cost comparisons between the two on IaaS infrastructure (Amazon ec2 VM instances) alongside container networks showed that the performance between the two can be kept comparable with the microservices supporting more requests per minute than monolithic applications and also having minimally lesser throughput and response times with the microservices application costing lesser to deploy than the monolithic application. Usage of PaaS providers like Amazon Lambda aggressively reduce the costs of deploying microservices, with the result being that companies can reduce their infrastructure costs by up to 77.08% [3]. The better standalone performance of monolithic applications is far outweighed by the cost-effectiveness and scalability of microservices.

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. However, as the number of users increases (tens of thousands and above), the cost effectiveness and ease of scalability of the microservices approach justifies the labor intensive process of setting up a microservices architecture / converting a monolithic application to serve microservices.

REFERENCES

[1] M. Mosleh, K. Dalili and B. Heydari, "Distributed or Monolithic? A Computational Architecture Decision Framework," in IEEE Systems Journal, vol. 12, no. 1, pp. 125-136, March 2018, doi: 10.1109/JSYST.2016.2594290.

[2] R. Flygare and A. Holmqvist, 'Performance characteristics between monolithic and microservice-based systems', Dissertation, 2017.

[3] Villamizar, M., Garcés, O., Ochoa, L. et al. Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures. SOCA 11, 233–247 (2017). https://doi.org/10.1007/s11761-017-0208-y

[4] J. Gouigoux and D. Tamzalit, "From Monolith to Microservices: Lessons Learned on an Industrial Migration to a Web Oriented Architecture," 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), Gothenburg, 2017, pp. 62-65, doi: 10.1109/ICSAW.2017.35.

[5] M. Villamizar et al., "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," 2015 10th Computing Colombian Conference (10CCC), Bogota, 2015, pp. 583-590, doi: 10.1109/ColumbianCC.2015.7333476.

[6] M. Fowler, J. Lewis. Microservices - a definition of this new architectural term, March 2014. https://martinfowler.com/articles/microservices.html [Online; posted 25-March-2014].

[7] Holger Knoche. Sustaining runtime performance while incrementally modernizing transactional monolithic software towards microservices. In Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering, ICPE '16, pages 121–124, New York, NY, USA, 2016. ACM.

[8] A. Balalaie, A. Heydarnoori, and P. Jamshidi. Microservices architecture enables devops: Migration to a cloud-native architecture, May 2016.

PLAGIARISM CHECK