ORIGINAL RESEARCH

Evaluation of a Multitenant SaaS Using Monolithic and Microservice Architectures

Poonam Mangwani[1,2,3] · **Niti Mangwani**[1,2,3] · **Sachin Motwani**[1,2,3]

Received: 3 November 2022 / Accepted: 17 December 2022 / Published online: 31 January 2023

Abstract
In the contemporary era, cloud computing has emerged as an eminent technology that offers on-demand services anytime and anywhere over the internet. Cloud environment allows organizations to scale their application based on demand. Traditionally, monolithic approach of application development has begun to face various bottlenecks and challenges. This has promoted a shift to a new paradigm, micro-service architecture for the development of cloud-based applications, which is gaining popularity due to decoupled independent services. Micro-service architecture contemplates overcoming the limited scalability of monolithic architecture. In this paper, a multitenant booking application is designed and developed using both monolithic and micro-service architecture as a case study. The application is deployed as Docker container images on Google cloud platform. The comparison of various factors, such as performance, scalability, load balancing, reliability, resource utilization, and infrastructure cost is performed. JMeter is used as a load generation and performance testing tool. Performance analysis in terms of response time is done for the multitenant booking application. Results indicate that independent scaling of micro-services leads to effective utilization of resources unlike the monolithic approach.

Keywords Cloud Computing · Docker Container · Microservices · Monolithic · Multitenant

Introduction

Cloud technology is flourishing as computing resources are available efficiently on demand, also known as the pay-per-use model. Organizations can deploy their applications using infrastructure as a service (IaaS), Platform as a service (PaaS), or they can outsource an application using the Software as a service (SaaS) model [1]. An extended model of SaaS, namely multitenant SaaS is gaining prominence where a single version of the software is shared among multiple tenants resulting in maximum utilization of resources. A tenant is any organization consisting of a group of users who share a common view of an application [2]. Moreover, it helps tenants to minimize maintenance costs due to centralized version control. Many tech giants and software companies are adopting multitenant architectural styles to develop SaaS applications due to the advantages.

To cater to the variable load, a mechanism is required that can dynamically allocate and deallocate resources. With the traditional way of developing an application using the monolithic approach, there is a lot of room for the resources to be underutilized. Applications developed using monolithic architectural style consist of multiple modules that perform specific tasks embedded in a single codebase which makes it tightly coupled. Scaling in this architectural style becomes a challenging task as scaling only a particular module which is in demand is impossible and the only way out is scaling the entire application. This results in allocating resources to the whole application and ultimately to other unnecessary modules as well, thus resulting in wastage of resources.

Moreover, adding or updating a module requires the whole application to stop and be restarted, which results in downtime making it unreliable. Also, the monolithic application has a single point of failure [3]. If any module of the application fails, it will break the whole application

This article is part of the topical collection "Advances in Computational Intelligence, Paradigms and Applications" guest edited by Young Lee and S. Meenakshi Sundaram.

✉ Poonam Mangwani
poonam.mangwani@iips.edu.in

1 Devi Ahilya University, Indore, Madhya Pradesh 452001, India

2 Groww, Bangalore, Karnataka 560103, India

3 Deutsche Bank, Pune, Maharashtra 411028, India

representing a single point of failure. The above limitations have led many organizations to move to a new paradigm known as micro-service architecture [4].

Micro-service architecture is an architectural style to develop software applications as a suite of small lightweight services that are loosely coupled and isolated. Micro-service architectural style is advantageous due to its multiple characteristics. Each micro-service can be developed and deployed independently due to which they provide better fault tolerance as all the microservices are independent and any fault in one of the micro-services won't ideally affect the other. Moreover, continuous updates and integration are possible which result in faster release cycles. Notably, efficient scaling of resources can be achieved as it consists of decoupled independent services.

To further understand the issues and challenges of micro-service and monolithic architecture, a multitenant booking application is designed and developed using both architectural styles. Docker containers are used as container technology. Google Kubernetes engine is used as an environment for deploying, managing, and scaling containerized applications [5]. Performance in terms of response time and scalability analysis on both the architectures is performed.

The rest of the paper is organized into the following sections. In "Related work", Related Work is stated. "A case study" provides a motivational example of a multitenant booking application and discusses characteristics of both the architectural styles. Tests and results are presented in "Test and results". Finally, "Conclusion and future work" concludes the paper.

Related work

With the advent of technology and the rapid increase in the number of users, tech giants are facing high loads. Hence, companies like Amazon, Netflix, and Uber are migrating from the traditional monolithic approach to the micro-service-based approach. These organizations must develop and integrate emerging features into their applications continuously, which is supposedly a challenge in the monolithic approach. The benefits that these companies avail compensate for the heavy costs that are incurred in the migration process from monolithic applications to microservices [6].

[3] describes the main features of microservices and highlights how these features improve scalability. Basic Principles like Bounded Context, size, and independence are discussed. One of the problems is the integration of microservices with the Internet of Things.

Singh et al. in [4], analyzed challenges in deployment and continuous integration of micro-services. To get around these challenges, an automated system is proposed and designed which is deployed on the docker container using a social networking application. Comparison of various performance parameters is done using monolithic and microservice architecture. Results show that applications developed and deployed using the microservice approach required less time and effort than the monolithic approach.

In [7], authors presented a program analysis-based method to migrate monolith legacy applications to microservices architecture. The key factors affecting the migration of legacy systems to microservice architecture are determined using empirical research. The experimental results show that the proposed method can solve the problem of migration of legacy applications efficiently with high accuracy and low performance cost.

Authors in [8] proposed a performance analytical model and experimented to study the performance indicators by capturing the details of provisioning at both container and VM layers. A micro-service platform is designed and developed on Amazon EC2 cloud using Docker technology family to identify important elements contributing to the performance of micro-service platforms.

Another study has been carried out in [9] to compare the performance of implementations of an application using native, Docker, and KVM implementation. In general, Docker equals or exceeds KVM performance in every case. Results showed that both Docker and KVM introduce negligible overhead for CPU and memory. Also, KVM is less suitable for workloads that are latency sensitive.

Authors in [10], have proposed a dynamic scaling model based on queueing networks to scale containers virtual resources to guarantee QoS parameters and save costs. JMT Simulator is used to validate the proposed model. The results show that depending on the workload conditions, the model can predict how many containers should be de-provisioned or provisioned.

Yadav et al. in [11], have explained a hybrid algorithm based on workload in real time that uses a threshold-based reactive model and time series forecasting-based proactive model for the provisioning and de-provisioning of computing resources. The experimental results show that the proposed hybrid approach is better than time series forecasting models in terms of horizontal scaling, average CPU utilization and response time.

[12], contributes to support software architects in rational decision-making whether migrating monolith systems to micro-services is feasible, or an entire application should be developed using this architectural style from scratch. For this, various experiments are performed in three different deployment environments. Some of the findings are monolith performs better than micro-services on a single machine, application performance degrades when scaled beyond a certain limit and the implementation technology does not have a distinguishable impact on the scalability performance.

Authors in [13], aim to check the scalability performance of cloud-based applications using different cloud environments, different configuration settings, and demand scenarios. It demonstrates the use of two technical scalability metrics for cloud-based software services for the comparison of software services running on the same as well as on different cloud platforms. The integration of technical scalability metrics into a previously proposed utility-oriented metric is also mentioned and discussed how to use these metrics for measuring and testing the scalability of cloud-based software services.

A case study

To evaluate the implication of using both architectural styles, a multitenant booking application is developed that imitates a real-world scenario. It offers various services for performing online bookings for flights, stays, and taxis. Various organizations can register themselves and subscribe to provide these services. These organizations behave as tenants. This application supports multiple tenants, and each tenant can have hundreds and thousands of users accessing the application concurrently. End users can register themselves online under different tenants and perform various booking operations. The application is developed using both monolithic as well as micro-service architecture.

Monolithic Architecture

Implementation

To develop this multitenant application using the monolithic architecture, all the service modules like flight, hotel, conveyance, and user are encapsulated into a single codebase and share a common database. Different organizations register themselves as tenants and configure the services according to their requirements. End users perform booking transactions after registering under these tenants. The application is developed using MVC framework where MVC denotes Model-View-Controller. It contains three layers: user interface as view, data as model, and business logic as controller. The architecture of the monolithic application is illustrated in Fig. 1. The application is developed using Flask, a python framework. User interface is developed using HTML5, CSS and JavaScript while the business logic is implemented using Python. For databases, MySQL is used which is a relational database. The application follows the REST framework and exposes APIs for facilitating user requests.

Deployment and Scaling

The monolithic booking application is deployed in a cloud environment using the Google Kubernetes Engine (GKE). Docker Engine is used to containerize the application into a docker image. This container image is then sourced and deployed inside a GKE cluster and is then finally exposed to the internet. A cluster contains nodes where each node consists of several pods. Each pod contains an instance of

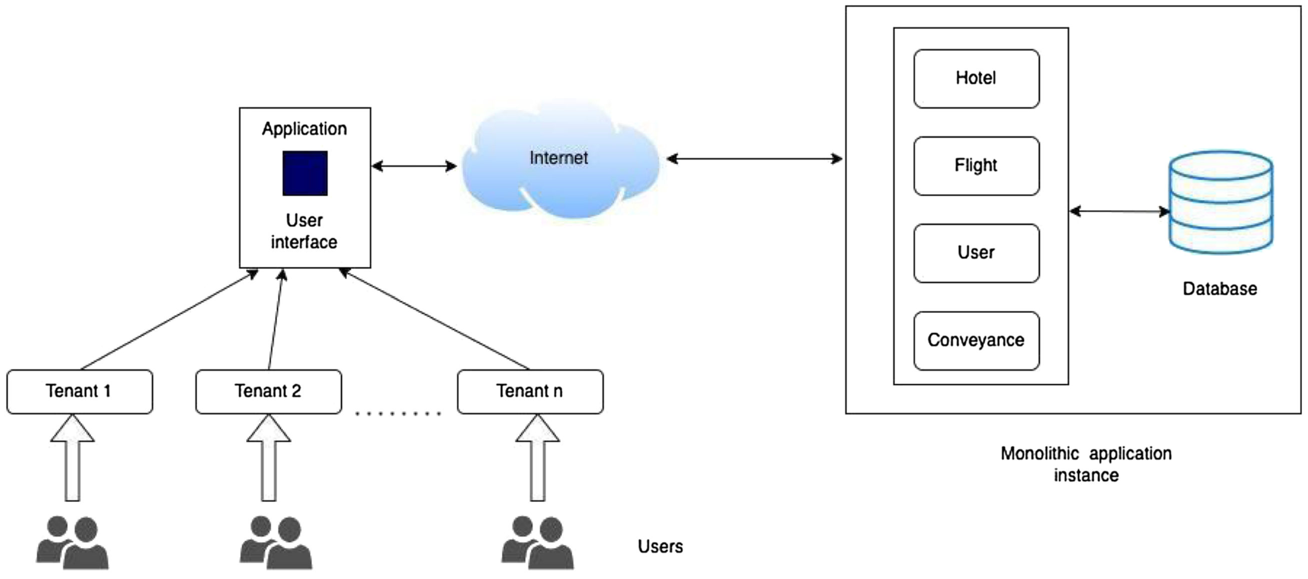

Fig. 1 Development model of the application using monolithic architecture

the application. Number of pods can be varied as per the load. Load balancer is used to scale the application when the load increases and distribute the load equally among the replicas i.e., pods. The deployment model is illustrated in Fig. 2 as follows:

Initially, with few user requests, the application works fine with a single instance since the load is low. With the increase in the number of user requests, a single instance is unable to handle the increased load. To cater to this, the application needs to be scaled. The demand for some functionalities of application might be higher than the others. For example, during a vacation period, there is a high probability of flight service getting more requests than the conveyance service.

Unfortunately, due to tight coupling of all the functionalities in a single application, scaling only the flight service will be a bottleneck and ultimately the whole application needs to be scaled. This means that every time the load increases for a particular module or service, the whole application needs to be scaled.

Cost and Resource Utilization

Resource utilization is the measure of the extent to which the available resources are being used. Resource utilization is a vital factor in determining the productivity of the application and the costs involved.

The resource type taken into consideration here is memory. Since the application offers four different services: Flight, Hotel, User, and Conveyance, each requiring mi memory for processing perfectly, where mi = m, the total memory allocation for one instance of this multitenant monolithic application would be:

$$m_{flight} + m_{flight} + m_{user} + m_{conveyance} = 4m \tag{1}$$

With the increasing load on flight service, the whole application needs to be scaled which would result in two instances of this application. Hence, the overall memory allocation would be a total of 8 m, as scaling of only flight service was required, but the whole application was scaled twice. This would result in underutilization of resources and therefore would increase the infrastructure cost.

Microservice Architecture

Implementation

To overcome the challenges in the monolithic application, the multitenant monolithic application was divided into different individual microservices with each microservice performing a specific functionality. A monolithic application can be classified into various microservices based on their differences and similarities, such as the technology used, security concerns, functionalities, etc. [6]. Here, the monolithic application is transformed into four different microservices based on the services they provide to the users.

Each service in the multitenant application, such as Flight, Hotel, Conveyance and User, acts as an independent and isolated service and connects to its own separate database. Due to the isolation between microservices, they can even be developed using different technologies and deployed

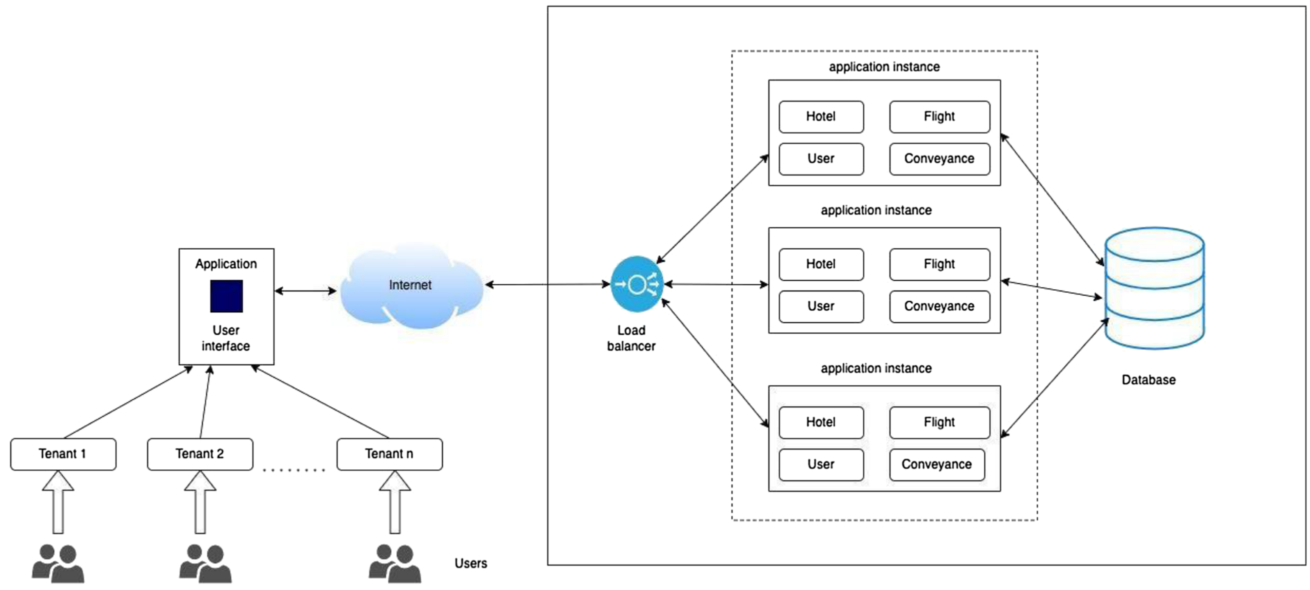

Fig. 2 Deployment model of the application using monolithic architecture

independently. However, the application has adopted a uniform technology. The architecture of the microservice application is illustrated in Fig. 3. Each microservice is developed using Flask, a Python framework. User interface is developed using HTML5, CSS and JavaScript while the business logic is implemented using Python. For databases, MySQL is used which is a relational database. The application follows the REST framework and exposes APIs for facilitating user requests.

Deployment and scaling

Each micro-service viz. Flight, Conveyance, Hotel and User is containerized into a docker container image using the Docker Engine which requires the source code of the microservice application and a docker file that contains commands to assemble the image. Containerizing is a method of virtualization that bundles all the components of and required by an application into a single image. This container image is first tested locally and is then pushed to a source repository on Google Kubernetes Engine (GKE). Further, a GKE cluster is created which allows to deploy and run the docker image. The image is then deployed, and ultimately exposed to the internet. Each micro-service contains REST APIs which are used for communication with micro-services, and between them. An API Gateway is configured in place to route the requests to these micro-services. A load balancer is used to distribute the load uniformly among the replicas of each micro-service. The deployment is illustrated in Fig. 4 as follows:

Initially during the low load condition, each microservice runs a single instance. Since now flight microservice is independent of other microservices, it can be scaled independently during high load, rather than scaling the complete application. Thus, microservices help in effective scaling and utilization of resources.

Along with the effective scaling, the updating and deployment cycle are also shorter for the microservice-based architecture as the specific microservice that needs to be updated will only be deployed and the remaining micro-services will remain untouched.

The multitenant microservice booking application is deployed on cloud environment and configured it with the auto-scaler to efficiently scale based on load under specific set of defined resource conditions. The necessary steps to deploy the application using microservices are:

1. Create a GKE cluster to deploy microservices.
2. Create a Docker file for each microservice and build it to generate the image and finally push it to the Google Container Registry.
3. Write manifest files—deployment.yml, service.yml and autoscaler.yml for each microservice to deploy on Kubernetes.

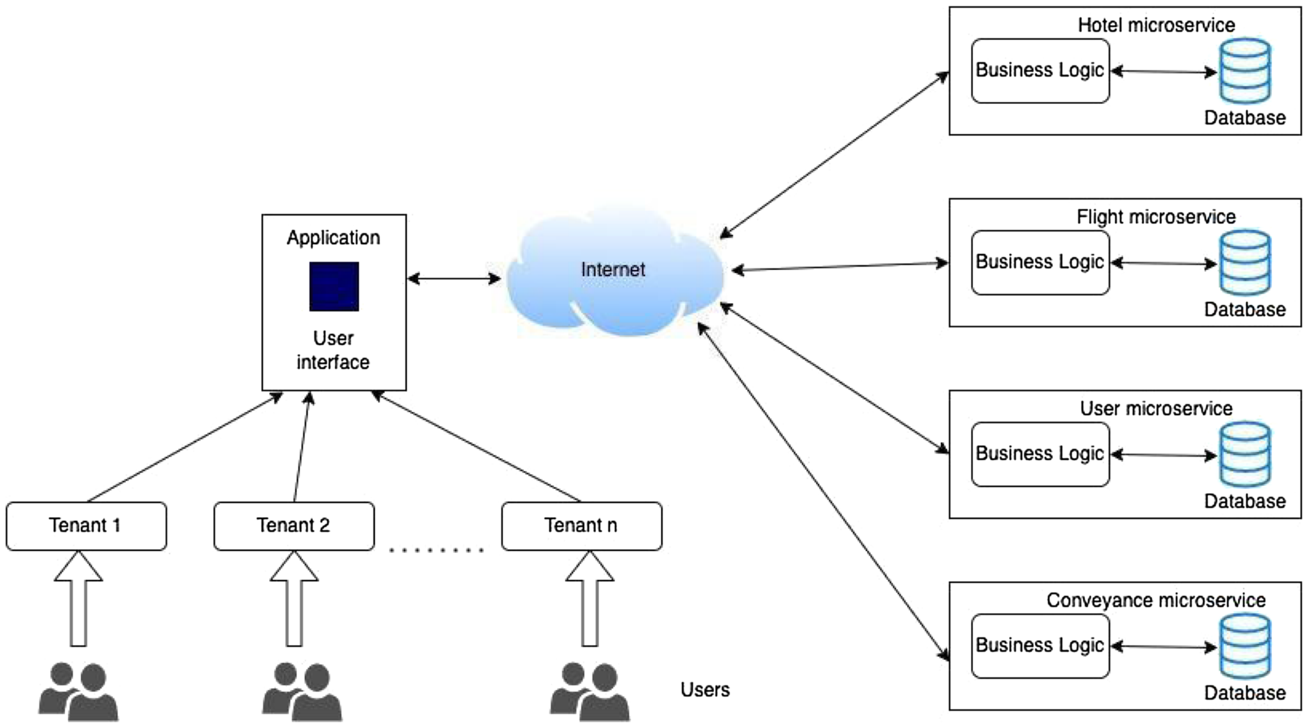

Fig. 3 Development model of the application using microservice architecture

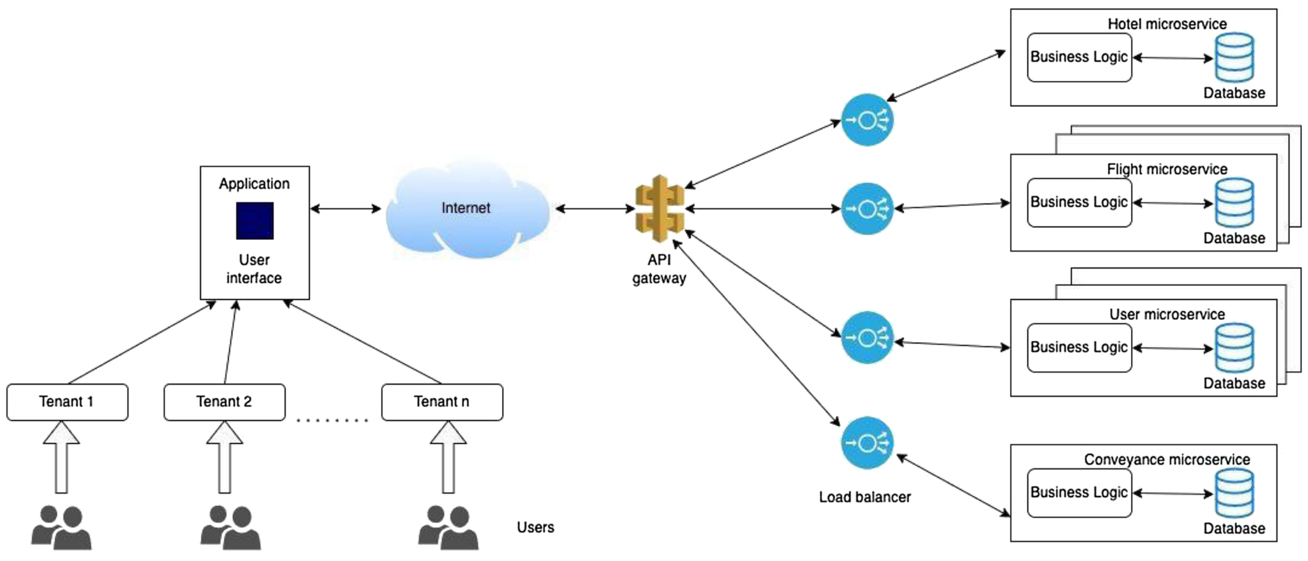

Fig. 4 Deployment model of the application using microservice architecture

- Deployment.yml: The file will contain the details about the Pod such as the image name, the port at which the microservice will run and the labels.
- Service.yml: It exposes an application running on a set of Pods as a load balancer.
- Autoscaler.yml: It specifies the minimum and maximum number of replicas that each microservice will have based on the resource's usage like CPU and memory.

4. Apply the manifest files to generate the Pods for each microservice and their corresponding service on Kubernetes.
5. Microservices can communicate with each other using the service discovery provided by Kubernetes.

Availability

In a monolithic application, a single error can cause the entire application to fail whereas this behavior can be avoided in a microservice architecture since it contains decoupled deployable units, which won't affect the entire system. For example, a fault in the hotel booking service might cause the whole hotel service to go down, but other services would function intact such as flight booking service will still allow users to book their flights and so on. Hence, microservice architectural style prevents a single point of failure unlike the monolithic architectural style.

Test and Results

After developing the multitenant booking application using both monolithic and microservice architectural approaches, different areas and factors are compared, such as development techniques, deployment and maintenance, resource utilization, performance, and scalability. The results are discussed in this section. To analyze and compare the performance in terms of response time of both the architectures, JMeter tool is configured to generate variable load by creating threads to simulate concurrent user requests. Apache JMeter is an open-source java-based software to perform load testing and performance testing of web applications [14]. Similar experimental settings were applied for testing both the approaches. Initially, JMeter is configured to create 100 user requests with a hold time of two minutes. The number of user requests increased in the increments of 100 to 500 users. The incoming user requests are created visualizing end users which may belong to different tenants. Multiple tests are performed at random times and performance results are collected through various reports. Figure 5 shows a comparison of average response times for both the approaches under variable load conditions. As seen from the graph, initially with few user requests, the response time for monolithic and microservice approach is almost similar. But as a greater number of requests are coming, the response time of monolithic starts increasing as compared to microservice approach. This illustrates that monolithic applications

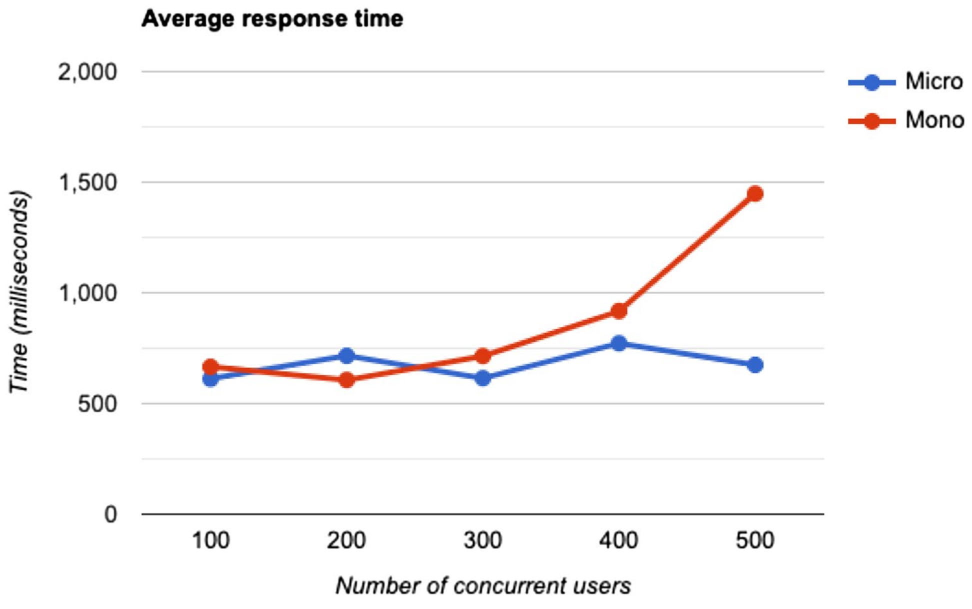

Fig. 5 Comparison of monolithic and microservice architectural styles in terms of average response time in milliseconds

perform better at low loads and microservices-based applications work better at higher loads. This is because in the case of microservices, the user requests are distributed among different independent microservices, whereas in the monolithic approach, all the user requests are handled by a single application containing all the modules. As the load increases, new replicas are being created for the microservices which help in sharing the load and this may happen only for the microservice that is receiving a high number of user requests. Figure 6 shows the 90-percentile response time graph for both the approaches for 500 users. It is a statistical measure that represents a value below which 90% of the total samples fall.

Conclusion and future work

Based on the development and deployment of the Multitenant booking application using both architectural styles, it could be understood and pinpointed the bottlenecks and benefits of the monolithic and microservice approach. One of the benefits of microservices is the capability to develop a large application as a suite of microservices that are independently developed and deployed, loosely coupled, and can be efficiently scaled. This study shows that the multitenant booking application with microservice design performs better than monolithic under high loads i.e.,

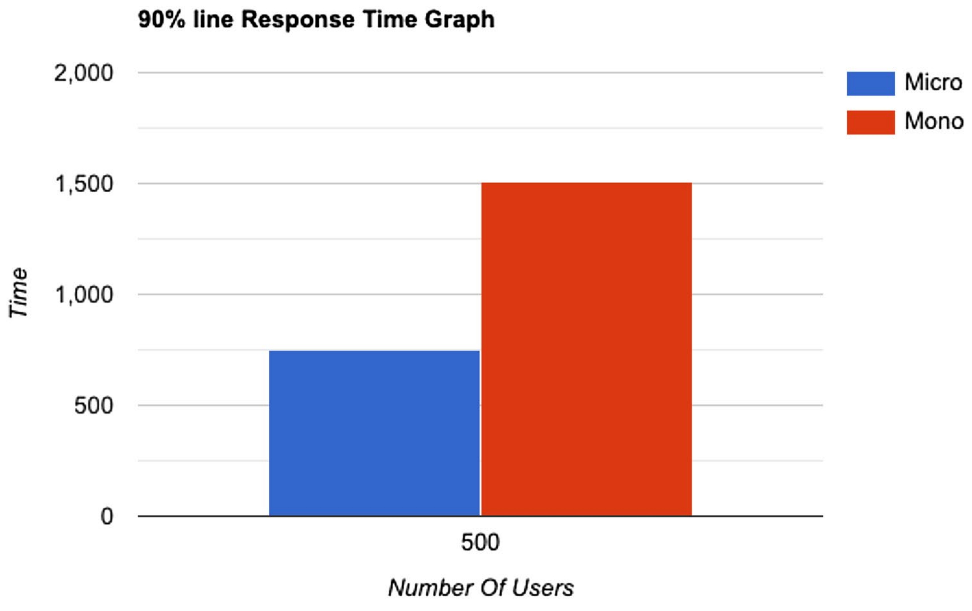

Fig. 6 90 percentile response time graph for monolithic and microservice architectural styles

greater number of user requests. A comparison between deployment methods and resource scaling of the application shows how microservices achieve efficient scaling by scaling only the microservice that needs to entertain more user requests unlike monolithic where the whole application needs to be scaled. This indicates a reduction in infrastructure costs.

As future work, the focus will be on enhancing the multi-tenant booking application by developing additional microservices using similar or different technology stack and integrating them. More performance tests will be performed to evaluate other performance and scaling metrics.

Funding No funding received for this research.

Data availablity The Multitenant application was deployed on Google cloud for performing analysis. The results were collected using Jmeter tool.

Declarations

Conflict of interest No conflict of interest.

References

1. Villamizar M, et al. Evaluating the Monolithic and the Microservice architecture pattern to deploy web applications in the cloud evaluando el patrón de arquitectura monolítica y de micro servicios para desplegar aplicaciones en la nube. 10th Comput Colomb Conf. 2015;11:583–90.
2. Adewojo AA, Bass JM. Evaluating the effect of multi-tenancy patterns in containerized cloud-hosted content management system. Proc Euromicro Int Conf. 2018. https://doi.org/10.1109/PDP2018.2018.00047.
3. Dragoni N, et al. Microservices: how to make your application scale. Lect Notes Comput Sci. 2018;10742:95–104.
4. Singh, V. & Peddoju, S. K. (2017) Container-based microservice architecture for cloud applications. Proceeding-IEEE Int Conf Comput Commun Autom. 17: 847–852
5. https://cloud.google.com. https://cloud.google.com, last Accessed 01 Nov 2021
6. Leitner P, Cito J, Stöckli E. Modelling and managing deployment costs of microservice-based cloud applications. Int Conf Util Cloud Comput. 2016;20:165–74.
7. Ren Z, et al. Migrating web applications from monolithic structure to microservices architecture. ACM Int Conf Proceeding Ser. 2018. https://doi.org/10.1145/3275219.3275230.
8. Khazaei H, Barna C, Beigi-Mohammadi N, Litoiu M. Efficiency analysis of provisioning microservices. Proc Int Conf Cloud Comput Technol Sci. 2016;1:261–8.
9. Felter W, Ferreira A, Rajamony R, Rubio J. An updated performance comparison of virtual machines and linux containers. TX: IBM Res. Austin; 2015. p. 171–2.
10. El Kafhali S, El Mir I, Salah K, Hanini M. Dynamic Scalability Model for Containerized Cloud Services. Arab J Sci Eng. 2020;45:10693–708.
11. Yadav MP, Raj G, Akarte HA, Yadav DK. Horizontal scaling for containerized application using hybrid approach. Ing des Syst d'Information. 2020;25:709–18.
12. Blinowski G, Ojdowska A, Przybylek A. Monolithic vs. microservice architecture: a performance and scalability evaluation. IEEE Access. 2022;10:20357–74.
13. Al-Said Ahmad A, Andras P. Scalability analysis comparisons of cloud-based software services. J Cloud Comput. 2019;8:5543.
14. https:jmeter.apache.org. https:jmeter.apache.org, last. Accessed 01 Nov 2022