# Container-based Microservice Architecture for Cloud Applications

Vindeep Singh
Department of Computer Science & Engineering
Indian Institute of Technology Roorkee
Roorkee, India
vindeepsingh@gmail.com

Sateesh K Peddoju
Department of Computer Science & Engineering
Indian Institute of Technology Roorkee
Roorkee, India
drpskfec@iitr.ac.in

***Abstract*—Cloud Environment allows enterprises to scale their application on demand. Microservice design is a new paradigm for cloud application development which is gaining popularity due to its granular approach and loosely coupled services unlike monolithic design with single code base. Applications developed using microservice design results in better scaling and gives extended flexibility to the developers with minimum cost. In this paper, first, different challenges in deployment and continuous integration of microservices are analyzed. To overcome these challenges, later, an automated system is proposed and designed which helps in deployment and continuous integration of microservices. Containers are recently heavily used in deploying the applications as they are easy to manage and lightweight when compared to traditional Virtual Machines (VMs). We have deployed the proposed microservices architecture on the docker containers and tested using a social networking application as case study. Finally, the results are presented and the performance of monolithic and microservice approach is compared using various parameters such as response time, throughput, deployment time etc. Results show that application developed using microservice approach and deployed using the proposed design reduce the time and effort for deployment and continuous integration of the application. Results also shows that microservice based application outperform monolithic design because of its low response time and high throughput.**

***Keywords—Microservice Design; Monolithic Design; Cloud Computing; Virtualization; Containers; Application Scaling; Docker***

## I. INTRODUCTION

Cloud Computing is changing the way of development and deployment of application because of its unique requirements such as higher rate of scalability, reliability, continuous deployment and Integration, and minimum downtime etc. [1]. Companies require their enterprise applications to scale on demand, which can be achieved through different cloud computing models. Companies face different challenges while deploying their applications on cloud due to their unique requirements, but the major issue is to scale different functionalities of the applications according to their requirements [2].

Scaling becomes an issue for the applications that are following monolithic approach; as it bundles the whole application in a single code base [3]. A simple application is easy to develop using monolithic approach, but as the size of the application and its user base grows it becomes really complex and difficult to scale up [4]. Monolithic applications can have tens or hundreds of different services tightly coupled in a single code base, which makes it difficult for different development teams to coordinate with each other. This is the reason why most of the major companies are moving to a new paradigm, known as microservice architecture [2].

Applications in microservice approach consists of many decoupled services independent of each other. Each of these services perform a specific task that is developed and deployed independently [5]. These services expose their endpoints for transferring any data or information to other service. Since these services have a micro functionality, this new paradigm is termed as Microservice architecture and the services are called microservices. This model helps in coping with the challenges of Service oriented Architecture (SOA) [1] as different teams can work on different microservices independently and hence have a shorter release cycles.

The main contributions of this paper are:

- To design a deployment methodology, which can reduce the effort and cost for deployment of microservice based applications with minimum downtime.
- To implement the proposed design and analyze it using a social networking application based on microservice design as a case study.
- To compare the performance of microservice and monolithic design using the case study application.

The rest of the paper is divided into followings sections. In Section II we discuss about the Related work. Proposed Design is presented in section III. Section IV presents a case study social networking application. In section V, we present the Experimental Setup. Results are presented in section VI. Paper is concluded in section VII.

## II. RELATED WORK

Many major companies like Amazon, Netflix, Uber etc. are moving from conventional monolithic design to microservice design due to their large user base and high

ISBN: 978-1-5090-6471-7/17/$31.00 ©2017 IEEE 

requirement for scaling [6]. These companies continuously update and integrate new features to their existing model throughout the application life cycle. Despite of larger cost of migrating an application from monolithic design to a microservice design these companies invest a large sum on the migration because of many advantages of microservices [3].

*Lin et al* [6] discuss about how a web application can be migrated to a cloud using microservice approach. Another study done by a *Xiao et al.* [5] explain how SOA using microservices and API helps in modern enterprises to effectively use their resources and manage software components.

Virtualization of microservices has been the key to increase the performance of cloud applications [7]. Many researchers have worked on performance analysis of different virtualization techniques. Virtual Machines is one way of achieving virtualization. Container is another emerging technology for virtualization which is gaining popularity over VMs due to its high performance, lightweight and higher scalability. These performance analyses are done by *Barik et al.* [8] where they compare the performance of a docker container and virtual machine with various simulations. Containers differ from VM in their architecture, a major difference is that the containers share host operating system and the virtualization is done at kernel level. Hence container require guest processes to be compatible with host kernel. Compared to VMs where each VM has its own guest OS, Containers reduce the overhead of having different guest OS for different container process. This results in high performance, less memory requirement and reduced infrastructure cost.

A thorough analysis of provisioning microservices using Amazon cloud and docker container technology is done by the *Khazaei et al.* [7]. Docker[1] container is an emerging container technology which provides different functionalities to make provisioning of microservices cost effective and easy to manage. Since cloud applications have a distributed environment, docker containers deployed on different host needs a virtual docker system. In [9] *Naik et al.* discuss about docker swarm which helps in optimizing resources by selecting the most appropriate host to deploy the docker containers.

## III. PROPOSED DESIGN

In this section, we discuss how a microservice based applications can be deployed such that it supports Continuous Integration and Continuous delivery with minimum downtime. The following sub-sections will present the proposed methodology of deploying the applications with its working functionality and need for containers, approaches for scaling the applications, and methods of rolling the application updates in the proposed design.

Despite of various advantages of microservice approach, deployment becomes a challenge with increasing number of microservices [3]. Continuous Integration and Continuous Delivery are two major challenges in any enterprise application due to their short release cycle. Any update to a microservice or an addition of any new microservice to the application needs to be integrated with the application with minimum effort and minimum downtime for the application. Service Discovery is another major challenge in deployment of microservice application [2]. Fig. 1 shows the proposed design to address these two important challenges including continuous integration and continuous delivery, and service discovery. In this design API Proxy combined with the configuration storage helps in achieving Service Discovery. API proxy stores all the configuration information such as IP Address, Port Number, timeouts API endpoints etc. for all the microservices.

### A. New Service

Every newly created service should be able to register itself by storing its runtime configuration and updating the deployment information so that other microservices can communicate with it. The deployment server fetches the source code from the repository, compiles and tests the microservice code to create microservice binaries. Using these binaries, deployment server creates a container image which is stored in container repository. Once the image is created, these images are deployed to the deployment server. Deployment server adds the configuration information of the newly deployed service to the configuration storage. This configuration information is accessed by other microservices to communicate with other microservices. Also, this

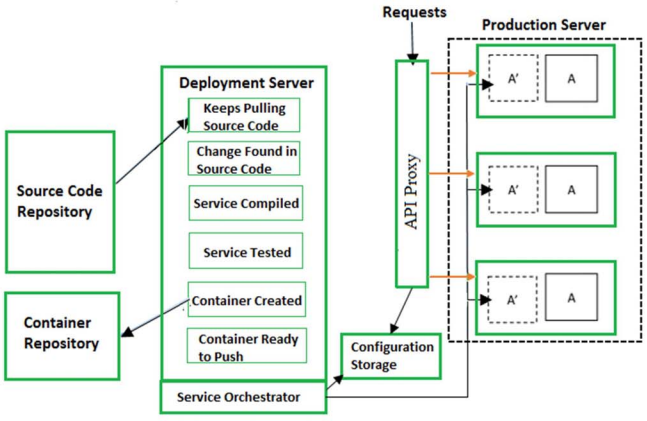


**Fig. 1.** Methodology of Deployment of Microservices on a Container using the Proposed Model.

[1] "https://www.docker.com", Accessed 2017

information is used by API Proxy for making any runtime configuration changes. Hence it provides the runtime configuration system, which can be updated and modified while the application is still running. This ensures minimum downtime when any microservice of application gets updated or return from failure. Deployment server uses container repository for starting more number of instances for an already running microservice. This ensures that deployment server do not need to build and test an already running microservice from scratch.

### *B. Service Update*

Microservices needs to be constantly updated because of many reasons such as bug fixes, addition of new features, change in logic etc. Since enterprise application must have zero downtime we need to reduce update time and ensure that the application can roll back to its previous state in case of any failure. This proposed model also serves the purpose of rolling the updates with minimum downtime. Microservice Application has a very short release cycle, so the system needs to ensure every upgrade to the application gets integrated to the application continuously with minimum downtime. We configured deployment server to deploy upgrades in case of any change in source code repository. Continuous integration can become very complex as the number of services increases, which require an automated system for deploying microservices continuously. Fig. 1 shows the flow of control while deploying an update in microservice design. Following Steps are taken by Deployment server while rolling the updates:

- Deployment Server Continuously pulls the code from source code repository.
- Whenever there is any modification (A') in any microservice (A), deployment server will compile, test and create a Container Image.
- This container image is pushed to the container repository with new tag. The deployment server sends commands to production server to deploy the updated microservice.
- In case of any failure during compilation, the deployment server will stop the updated container and system is restored back to previous state.
- When the newly updated container (A') is up and running, deployment server will change the configuration in configuration storage changing the IP Address for updated microservice to the newly created containers.
- Now all the requests for the microservice are routed to updated containers. Deployment server will then stop old container (A) and remove them from production server.

This process ensures minimum downtime as the application continues to work with the updated system only when the new containers are up and running.

## IV. CASE STUDY

This section discusses a case study on Social Networking application which has four major functionalities including Login, Register, Newsfeed, and Messaging. The application was developed and deployed for both microservice as well as monolithic approach using the deployment design.

### *A. MVC based Monolithic Approach*

First, we present how the application can be implemented using Monolithic approach and what are the disadvantages of this approach. The application follows the basic MVC structure with a back-end containing all the functionality to which user interacts using a webUI [10]. The back-end has all the functionality coupled together in a single code base, which store all the data in a single database shared among all the services.

The application starts with a small user base which can be handled by a single instance since the number of request are very less. But as the user starts to grow, the application needs to be scaled up to handle the increased load [11]. Some functionalities of application has more effect of increased users than the others due to the difference in popularity or usage. For Example, every user needs to register only one time and will not need registration service again, but a single user can use messaging service tens or hundreds of times in a single day. This usage difference creates a scaling problem in a monolithic application as the registration logic cannot be separated from messaging logic. So every time messaging service needs to scale up, the whole application with all the services needs to be scaled up as we can see in Fig. 2.

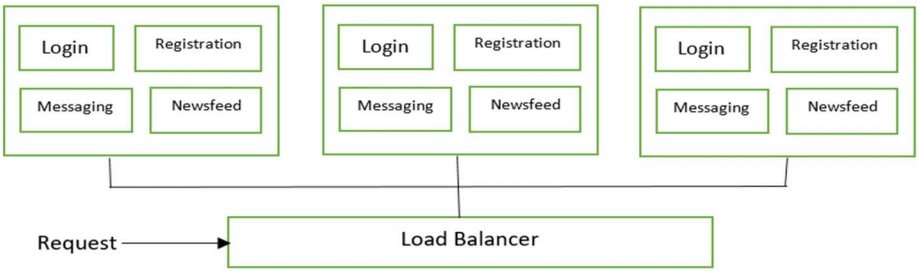


**Fig. 2.** Scaling in Monolithic Design

In Fig. 2 we can see how scaling in monolithic approach can lead to resource wastage as to scale up messaging service three times, whole application needs to be scaled up three times. This results in less utilization of resources and hence increase the infrastructure cost.

### B. Microservice Approach

To meet the challenges of monolithic approach, we divided the application into different microservices, with each microservice having a single function. There are many factors based on which a monolithic application can be divided into microservices such as functional dependency, security, function popularity etc. In the experimental studies on proposed work, we have divided the application such that there is one microservice for every function. Hence we have four different microservices for login, registration, newsfeed and messaging service. These microservice expose REST APIs which can be used by other microservice for any communication of data or information. There are other methods such as message passing by which microservices can communicate.

Since, now "Registration" microservice is independent of "Messaging" service, it can be scaled up independently without wasting any resources and can reduce the deployment cost. Microservice independency also helps to reduce complexity while application development. Since services are no longer coupled together, different development teams working on different microservices can take their own decisions such as coding language, communication protocol, database model etc. can be decided by individual teams for service development without any interruption from other teams. For example, in this application we can code "Registration" microservice in java while "Newsfeed" in python. Similarly, each functionality can have its own database model and communication protocol. For example, "Registration" microservice can use REST API for communication with other microservice, while "Newsfeed" can use kafka[2] messaging service to share feed information.

Fig. 3 shows how "Messaging" service scaled up independently without scaling other services and hence can reduce deployment cost and increase resource utilization.

## V. Experimental Setup

This subsection discuss different components required to automate the process of deployment, from building a microservice to deploying on the server.

TABLE I. Tools Used for Microservice Deployment Experiment

| *Design Component* | *Tool Used* |
|---|---|
| API Proxy | HAProxy |
| Configuration Storage | Consul |
| Build Server | Jenkins |
| Source Repository | GIT Repository |
| Container Repository | Docker Registry |

We have used specific tools for different components of proposed design as mentioned in Table I. In addition to this, our application needs a data storage to store application data for which we configured and installed a *Cassandra*[3] database. The microservices are developed using *Spring*[4] framework. All microservices are deployed in a *docker* container isolated from one another. *HAProxy*[5] also acts as a load balancer which we configured to use round robin algorithm to distribute the load on different instances of a microservice.

To automate the process of build, test and deploy all the microservices, we configured *Jenkins*[6] which pull up the code from *GIT* repository. Jenkins can be configured to pull the application code from any source similar to GIT. We installed scripts in Jenkins server to compile, test and package microservices in the form of docker image. These docker images are then deployed on the deployment server. We configured a local docker registry to store created docker images; this prevents Jenkins server from rebuilding an already created microservice.

Also docker registry helps to redeploy a microservice during any failure, which minimize the downtime. Jenkins server also updates configuration information of newly created microservice in consul[7], which is used for service discovery and request routing by HAProxy.

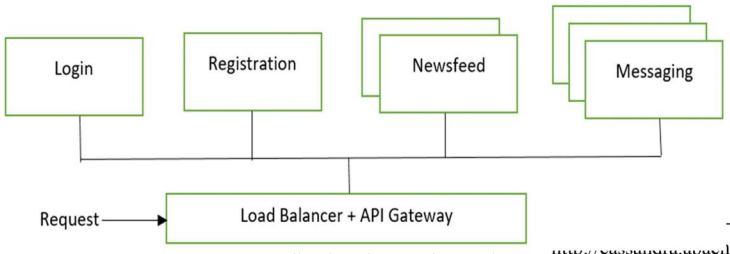


**Fig. 3.** Scaling in Microservice Design

[2] "https://kafka.apache.org", Accessed 2017

[3] "http://cassandra.apache.org", Accessed 2017

[4] "https://spring.io", Accessed 2017

[5] "http://www.haproxy.org", Accessed 2017

[6] "https://www.jenkins.io", Accessed 2017

[7] "https://www.consul.io", Accessed 2017

## VI. Results and Evaluation

Before implementing the proposed model, we have experimented to evaluate the impact of running a microservice on a container when compared to VM. The following subsection presents a comparative performance analysis of VM verses containers. Further, using the case study application, performance results were evaluated for monolithic design vs microservice design. With parameters such as response time, deployment time, throughput, scalability etc., the application was evaluated and achieved results are discussed.

### A. VM vs Container

Since microservices are independent of each other, sandboxing these microservices using a virtualized environment is possible which makes the system more secure and easy to manage. There are two main approach for virtualization Virtual Machines (VM) and Containers. Containers are easy to manage due to their lightweight; they outperform VMs in many aspects. To evaluate this, we applied both the approaches for our "Messaging" microservice and found many results which leads to selection of right approach for our application. Table II shows the comparative performance in three different categories. These results are analyzed below.

*Image Size:* Virtual Machines and containers differ in their structures. Virtual machines have their own guest OS which is independent from other VMs and thus every VMs contain whole OS in its image. On the other hand, containers running on same host shares host OS and libraries which results in reduced image size. We created a VM image and a container image for one of our microservice "Messaging". Results in Table II clearly shows a reduction of 66 % in container image size as compared to VM size.

*Deployment:* Enterprise applications have a very small release cycle and hence need to be updated and redeployed frequently in order to make changes to the application. Deployment time plays a major role to ensure minimum downtime for application. Hence we compared the deployment time for both VM and containers for messaging microservice and we can see in Table II that containers take significantly less time to deploy than the VM.

*Update:* Virtualization of any kind whether it is VM based or container based helps in achieving zero downtime and can reduce efforts to roll updates. Also ensure that for every update, system runs on previous microservice versions till the new version is up and running and in case of any failure system can roll back to its previous state. The difference in performance for updating a microservice in VM and containers is due to download and installation of binaries every time a new update needs to be rolled out. Since containers share most of the binaries of host OS, they take less time for image creation than a VM. In Table II we can see the difference in performance while updating "Messaging" microservice. These results shows that docker container outperforms VMs for our microservice application. Hence, we used docker containers to deploy microservices.

TABLE II. Virtual Machines vs Containers

| | ***VM*** | ***Container*** |
|---|---|---|
| Image Size | 1056 MB | 357 MB |
| Deployment Time | 10s | 2s |
| Update Time | 17s | 5s |

### B. Monlithic Vs. Microservices

For comparing the performance of both the approaches, we installed and used *Jmeter*[8] to send continuous requests to both the designs. For monolithic design we used web API for sending the request to the application, whereas for microservice design we used HAProxy to send requests to the target service.

Jmeter is configured to create 50 request threads with ramp up time (total time to send requests) of 10 sec and loop count of 1. Using this test, we evaluated the response time, throughput and 90% Line results (Response time for 90% of requests). Fig. 4 shows a comparison between response time for monolithic vs microservice design. We can see that initially with less number of threads, the difference between response time of the microservices and monolithic application is very less. But as more threads are created, response time in monolithic application gradually increases.

In the second test we recorded the maximum time as shown in Fig. 5(a), average time as shown in Fig. 5(b), and throughput as shown in Fig. 5(c) for both the applications. In this test we set number of threads to 2000 and ramp up time of 100 secs with 10 loop count. It is observed that microservice application outperform monolithic application with less response time for large number of requests. In case of microservices, requests are distributed between different independent microservices. On the other hand, in monolithic application all requests are handled by a single application which contains all the services in a single codebase. Microservice approach also results in higher throughput with more number of requests handled per second.

### C. Deployment and Scaling

Using the design architecture for deploying the microservices, we evaluated the time to deploy a microservice

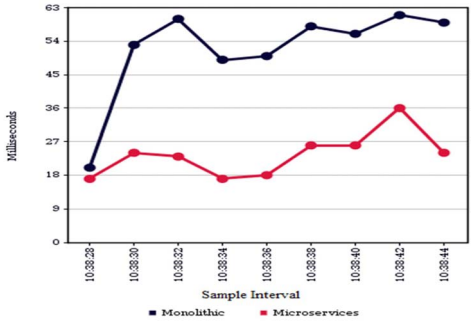


**Fig. 4.** Response Time Comparison for monolithic vs microservice approach

[8] "http://jmeter.apache.org", Accessed 2017

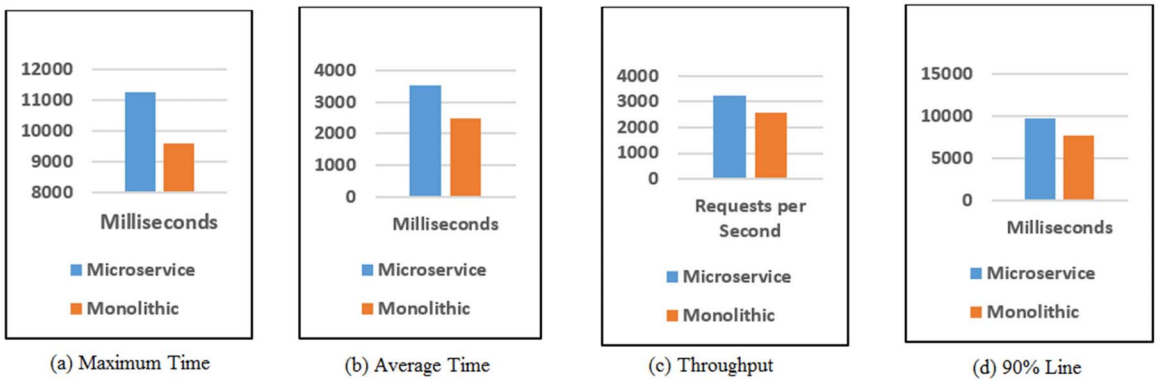


**Fig. 5.** Performance Comparison for Monolithic vs Microservice Approach

during initial deployment and while scaling the microservices. After the initial deployment, application needs to be scaled up or down according to the requirement of a particular service. Since each microservice is deployed independently, they take comparatively less time than a monolithic application. We tested and deployed "Registration" microservice which took about 16 secs to deploy and run on the server, whereas in case of monolithic application it took 40 secs to deploy the whole application. That's around 60% reduction in scale time.

Also, microservice approach allows application to be scaled at granular level and hence it can scale different microservices independently.

### *D. Rolling Updates*

Rolling updates allow microservice application to be continuously integrated whenever there are some updates for any microservice. We used our automated system to deploy an update for "login" service in both microservice as well as monolithic application. Since monolithic require whole application to be compiled and redeployed, it took 400 secs to deliver the updates. Whereas in case of microservices it took 160 secs to deliver the updates. Which shows how microservice application deployed using an autonomous system can reduce the down time of application and continuously deliver any updates with minimum efforts.

## VII. Conclusion

This study shows that microservice based application can outperform monolithic application by reducing the response time and increasing the throughput. Also, it describes how microservice based application helps in achieving higher scalability due to its granular design. Microservice based application allows different teams to work independently on different part of a large enterprise application and can easily integrate with each other. The results show how the automated deployment design for microservices can reduce time and effort of deployment, scaling and integrating updates to the running application with minimum down time. A Comparison between deployment of microservice verses deployment of monolithic design shows how microservice application can help in achieving higher scalability. Our experimental results also indicate that Containers are the right launchers for the microservice based applications when compared to VMs.

## References

[1] J. Wu and T. Wang, "Research and Application of SOA and Cloud Computing Model," 2014 Enterprise Systems Conference, Shanghai, 2014, pp. 294-299.

[2] YaleYu, H. Silveira and M.Sundaram,"A microservice based reference architecture model in the context of enterprise architecture," 2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC), Xi'an, 2016, pp. 18561860.

[3] P. Leitner, J. Cito and E. Stckli, "Modelling and Managing Deployment Costs of Microservice-Based Cloud Applications," 2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC), Shanghai, China, 2016, pp. 165-174.

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

[5] Z. Xiao, I. Wijegunaratne and X. Qiang, "Reflections on SOA and Microservices," 2016 4th International Conference on Enterprise Systems (ES), Melbourne, Australia, 2016, pp. 60-67.

[6] J. Lin, L. C. Lin and S. Huang, "Migrating web applications to clouds with microservice architectures," 2016 International Conference on Applied System Innovation (ICASI), Okinawa, 2016, pp. 1-4.

[7] H. Khazaei, C. Barna, N. Beigi-Mohammadi and M. Litoiu, "Efficiency Analysis of Provisioning Microservices," 2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Luxembourg City, 2016, pp. 261-268.

[8] R. K. Barik, R. K. Lenka, K. R. Rao and D. Ghose, "Performance analysis of virtual machines and containers in cloud computing," 2016 International Conference on Computing, Communication and Automation (ICCCA), Noida, 2016, pp. 1204-1210.

[9] N. Naik, "Building a virtual system of systems using docker swarm in multiple clouds," 2016 IEEE International Symposium on Systems Engineering (ISSE), Edinburgh, 2016, pp. 1-3.

[10] N. Alshuqayran, N. Ali and R. Evans, "A Systematic Mapping Study in Microservice Architecture,"2016 IEEE 9thInternational Conference on Service-Oriented Computing and Applications (SOCA), Macau, 2016, pp. 44-51.

[11] Toffetti, Giovanni and Brunner, Sandro and Bl¨ochlinger, Martin and Dudouet, Florian and Edmonds, Andrew, "An Architecture for Selfmanaging Microservices," AIMC '15, 2015.