

PERFORMANCE ANALYSIS OF MICROSERVICES BEHAVIOR IN CLOUD VS CONTAINERIZED DOMAIN BASED ON CPU UTILIZATION

Sushant . Jhingran*

Sharda University, Computer Science, Greater Noida, ,201305, India

E-mail: sushantjhingran@gmail.com

ORCID iD: <https://orcid.org/0000-0002-6933-4282>

Nitin . Rakesh

Sharda University, Computer Science, Greater Noida, ,201305, India

E-mail: nitin.rakesh@gmail.com

ORCID iD: <https://orcid.org/0000-0002-1343-5244>

Abstract: - Enterprise application development is rapidly moving towards a microservices-based approach. Microservices development makes application deployment more reliable and responsive based on their architecture and the way of deployment. Still, the performance of microservices is different in all environments based on resources provided by the respective cloud and services provided in the backend such as auto-scaling, load balancer, and multiple monitoring parameters. So, it is strenuous to identify Scaling and monitoring of microservice-based applications are quick as compared to monolithic applications [1]. In this paper, we deployed microservice applications in cloud and containerized environments to analyze their CPU utilization over multiple network input requests. Monolithic applications are tightly coupled while microservices applications are loosely coupled which help the API gateway to easily interact with each service module. With reference to monitoring parameters, CPU utilization is 23 percent in cloud environment. Additionally, we deployed the equivalent microservice in a containerized environment with extended resources to minimize CPU utilization to 17 percent. Furthermore, we have shown the performance of the application with “Network IN” and “Network Out” requests.

Keywords— *Application Deployment; cloud; Docker; Micro service; virtualized;*

1. Introduction

Application deployment contains various phases such as the development of applications based on design patterns and the deployment of applications in the appropriate servers. Deployment of microservices can be done in multiple environments. Application deployment can be done in multiple hosting environments. Every hosting environment has different parameters to check the performance and behavior of an application. Microservices are mostly used in service industries because of their lightweight features. Microservices can be deployed on the cloud and measure their performance on different metrics parameters. Cloud gives us a different type of environment in terms of serverless computing. Serverless structures include EC2, ECR, and CodeStar other services like services of different platforms [2]. Microservices performance can be measured on metrics provided by a serverless cloud environment.

In cloud computing, Serverless approach is used to deploy applications. Auto Scaling features are provided by various cloud environments to avoid the temporary stopping of any application. Serverless computing is also an approach to deploying microservice-based applications. An environment can be created for the deployment of microservice-based applications on cloud-like Elastic container registry and docker [3][4]. During the deployment an image created and a receptive container will be initialized to execute the application. This combination makes our application lightweight. Monitoring of any application requires some metrics like Network utilization and CPU utilization. These parameters define the performance of any application. Cloud Vendors provide different types of serverless approaches like S3(Simple storage services), RDS (Relation Database services), and ECS (Elastic container services). In the cloud domain, monitoring of applications can be done from different availability zones[5]. These zones are provided by cloud vendors like us-east 1a. Multiple users can be created and they all can access and monitor the data with cloud vendors. Traditionally applications are hosted on servers provided by many providers [6]. To improve performance. Applications are used to deploy in package format. These packages are created by developers in the format of Jar or War files. Some of the developers have developed these files in which the combination of these hosted servers contains various facilities. Applications are used to deploy on various hosting environments like Linux and cloud-based hosting. Bundle files are deployed on a cloud-based environment with docker and Kubernetes to provide application scalability [7]. Cloud environment also provides elastic commuting and beanstalk services for deployment with monitoring and notifications of services. Applications are used to deploy on various hosting environments like Linux and cloud-based hosting. Bundle files are deployed on a cloud-based environment with docker and Kubernetes to provide application scalability [7]. Cloud environment also provides elastic commuting and beanstalk services for deployment with monitoring and notifications of services.

2. LITERATURE REVIEW

The performance of any application depends upon the structure of its development. Earlier applications were developed on a modular approach i.e. small modules were developed and linked together. Modular approach makes code reusable, which can be used later. Web applications are deployed on servers so they require very less space in memory. A web server known as a deployment environment, on that application can be deployed easily. Applications can be easily accessible with a browser from a remote machine. The actual application is deployed on server from where response will be fetched of the specific requests made by a user. Developers deploy their applications on a server instead of multiple machines and users can access web portals on remote machines. Applications are usually installed in one server and instantiated by multiple machines[8]. The earlier monolithic architecture was used to develop applications [9]. Controllers are responsible to handle the request which comes from the front end. The controller forwards respective requests to the service layer and after that, if required then the service layer communicates with the Dao layer. All layers or modules access a single database or we can say that all modules are inside a single container i.e. single code base with multiple modules. All modules will be bundled in a single jar and after that jar file can be deployed on a server for client access. A single instance of one application can be run

on multiple applications but it will become monolithic architecture and these instances cannot communicate with each other. In a monolithic type of architecture components of an application are tightly coupled into a single module-based call. The different parts of the application, such as the user interface, business logic, and data storage, are all tightly integrated together and built as a single unit. Microservices and monolithic architecture are two different approaches to building and organizing software systems. Monolithic architecture is a traditional approach, where all components of an application are combined into a single codebase and deployed as a single unit. The application is tightly coupled, which means that changes to one part of the system can affect the entire application. Microservices architecture is a software design pattern in which an application is broken down into small, independent, and loosely coupled services. Each service is responsible for a specific task and can be developed, deployed, and scaled independently of the other services. The services communicate with each other over a network, typically using lightweight protocols such as HTTP or gRPC. Microservices architecture is easier to test as each service can be tested separately, leading to a more efficient testing process, also it's more secure as security bugs in one service do not affect the whole application. Scaling is a complex task in monolithic-based applications, and an extension of the project becomes a stumbling block [10]. Adaptations of change in API is very difficult for monolithic applications. Integration of different technologies is troublesome in a monolithic application. Any error in any specific module can break down the complete structure of the application. Network latency is comparatively good in monolithic applications because of a single communication channel. Microservices are independent modules that work simultaneously. Each microservice can communicate on a service layer with the other with some lightweight protocols [11]. Decoupled modules call independent databases according to service and all services can communicate with each other with REST or JSON. All services are loosely coupled because of an independent code base. Services can be updated independently without any scalability issue i.e. if anyone service X is updated for a time being then other services run smoothly and once updating will be done X services up automatically without reflecting the complete project. Even if different domain is used to develop any microservice then they can easily communicate with each other.

Architecture of Micro services

The architecture of a microservices system generally consists of several different components that work together to provide the overall functionality of the system. Some of the key components of a microservices architecture include:

Services: These are the individual components of the system that perform specific tasks. Each service is a self-contained unit of functionality that can be developed, deployed, and scaled independently.

Service Registry: This is a central repository that keeps track of all the services in the system and their location. Services use the service registry to discover other services and to register themselves when they come online.

API Gateway: This is a component that acts as a single-entry point for all incoming requests to the system. The API gateway is responsible for routing requests to the appropriate service,

handling authentication and authorization, and performing other tasks such as rate-limiting and caching.

Message Bus: This is a messaging system that allows services to communicate with each other asynchronously. Services use the message bus to send messages to one another, rather than calling each other directly.

Load Balancer: This is a component that distributes incoming requests across multiple instances of a service. The load balancer can help ensure that the system remains available and responsive, even under heavy load.

Database: Microservices are designed to be loosely coupled, so each service has its own database, each service is responsible for its own data, this means that the data stored in one service is not directly accessible by other services.

Monitoring and Logging: Microservices architecture requires extensive monitoring and logging in order to track the health and performance of individual services, as well as to troubleshoot issues.

These components work together to provide a flexible, scalable, and resilient architecture that is well suited for building large, complex systems. However, it should be noted that the specific implementation of a microservices architecture can vary depending on the needs of the project and the technologies being used.

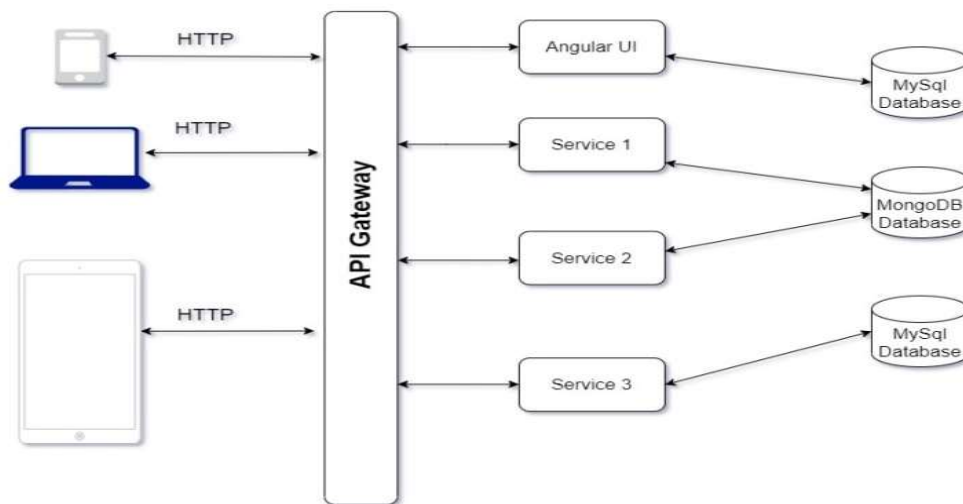


Fig. 1. Architecture of microservices

Architecture of microservice contains multiple microservices which are being called by each other with different ports. API gateway is a single end point who is responsible to communicate between microservices and client, if API gateway will not present then client call each microservice independently which behaves like different projects [12]. Client call single URL of API gateway; rest module will be taken care by independent microservices to access all backend services. Fault tolerance will be managed with hystrix library. This library can manage

in case of any service is down. This architecture follows a Eureka service discovery as a microservice where other microservices can be register and discover once required. In micro service, multiple functionalities can be communicated either with HTTP or JSON format. To dockerize microservice application, docker image and containers are created to dockerize application [13]. Microservice application can be deploy on Docker container once Docker file is created. IDEs or initializer are used to create microservice based application. While creating microservice only REST-API is required. Once Application will setup than configure all the respective path and initiate docker console for deployment. Each microservice is owned and maintained by a small, cross-functional team that is responsible for its development, deployment, and operation. Microservices are designed to be loosely coupled, meaning that they should be able to operate independently of one another and should not have strong dependencies on each other. Microservices should be designed to be deployed automatically and should be easy to deploy and scale in a continuous delivery environment. Microservices should be designed to emit metrics, traces and logs so that engineers can have a better view of the current state of the systems. To organize a microservices architecture is to use a service registry, where each service registers itself when it starts up and deregisters itself when it shuts down. Clients of the services can then use the service registry to discover the location of a service at runtime. The service registry can also be used to store metadata about the services, such as their current version, status, and health.

3. COMPARATIVE ANALYSIS OF THE VARIOUS HOSTING ENVIRONMENT

Microservice applications deployed in different environment such as cPanel, Virtual private server and multiple cloud providers. Multiple parameters can be used to monitor application performance. Monitoring parameters are used to analyses the behavior of application in respective cloud domains. Below some parameters are shown with different environment.

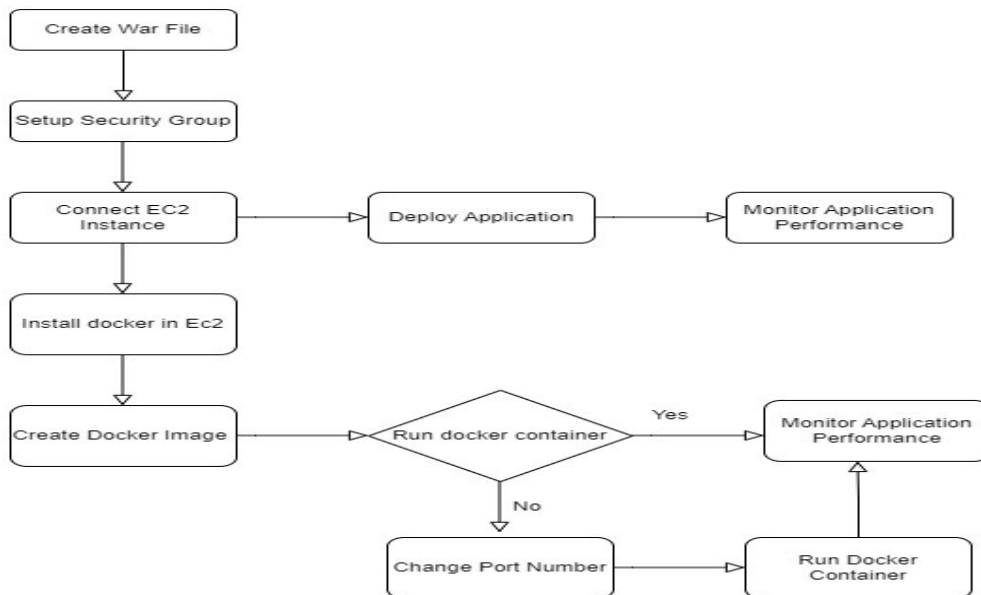


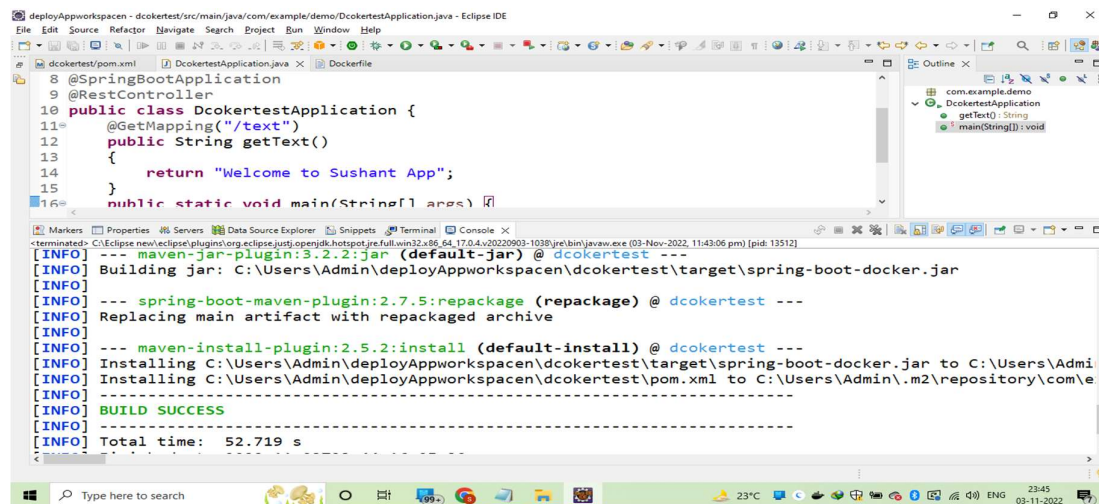
Fig. 2. Process Flow chart

PERFORMANCE ANALYSIS OF MICROSERVICES BEHAVIOR IN CLOUD VS CONTAINERIZED DOMAIN BASED ON CPU UTILIZATION

In Fig.2, Process of implementation flow is showing with multiple services are used in application such as Product Service, Order Service, Notification Service, Inventory Service and Notification Service. These services will be going to communicate with each other in synchronous and asynchronous communication. Stateless services are also used which are not going to communicate with any database. Product Service use lombok to reduce boilerplate code. NoSql based service been setup to communicate between different databases.

a) Setup application in Docker Environment.

Docker is very less dependent on OS. Traffic can be dynamically handled with container to improve the performance. Rest APIs are used to setup micro service application in Docker. Web dependency added features of REST in microservices. Rest Controller can provide its access to WWW. Maven architecture can create a JAR file. Docker containers are portable and can run on any system that has Docker installed, making it easy to deploy any microservices on a wide range of platforms. Docker can be used as part of a continuous delivery pipeline to package and deploy microservices automatically, which can speed up release process.



```
8 @SpringBootApplication
9 @RestController
10 public class DcokertestApplication {
11     @GetMapping("/text")
12     public String getText()
13     {
14         return "Welcome to Sushant App";
15     }
16     public static void main(String[] args) {
17         // ...
18     }
19 }

[INFO] --- maven-jar-plugin:3.2.2:jar (default-jar) @ dcokertest ---
[INFO] Building jar: C:\Users\Admin\deployAppworkspacen\dcokertest\target\spring-boot-docker.jar
[INFO] --- spring-boot-maven-plugin:2.7.5:repackage (repackage) @ dcokertest ---
[INFO] Replacing main artifact with repackaged archive
[INFO] --- maven-install-plugin:2.5.2:install (default-install) @ dcokertest ---
[INFO] Installing C:\Users\Admin\deployAppworkspacen\dcokertest\target\spring-boot-docker.jar to C:\Users\Admin\m2\repository\com\example\demo\dcokertest-application\0.0.1-SNAPSHOT\spring-boot-docker.jar
[INFO] Installing C:\Users\Admin\deployAppworkspacen\dcokertest\pom.xml to C:\Users\Admin\m2\repository\com\example\demo\dcokertest\pom.xml
[INFO] BUILD SUCCESS
[INFO] Total time: 52.719 s
```

Fig. 3. Shows creation of Jar archive file is created in target folder to setup application in docker environment with maven install

PERFORMANCE ANALYSIS OF MICROSERVICES BEHAVIOR IN CLOUD VS CONTAINERIZED DOMAIN BASED ON CPU UTILIZATION

```

C:\Windows\system32\cmd.exe - docker build -t spring-boot-docker.jar
C:\Users\Admin\deploy\appworkspaces\dcokertest>docker build -t spring-boot-docker.jar
[+] Building 286.9s (8/8) FINISHED
=> [internal] load build definition from Dockerfile
=> transferring dockerfile: 178B
=> [internal] load .dockerignore
=> transferring context: 2B
=> [internal] load metadata for docker.io/library/openjdk:8
[auth] library/openjdk:pull token for registry-1.docker.io
=> [internal] load build context
=> transferring context: 17.63MB
=> [1/2] FROM docker.io/library/openjdk:8@sha256:86e863cc57215cfb181bd319736d0baf625fe8f150577f9eb58bd937f5452cb8
252.5s
=> resolve docker.io/library/openjdk:8@sha256:86e863cc57215cfb181bd319736d0baf625fe8f150577f9eb58bd937f5452cb8
1.0s
=> sha256:86e863cc57215cfb181bd319736d0baf625fe8f150577f9eb58bd937f5452cb8 1.04kB / 1.04kB
0.0s
=> sha256:3ef2ac94130765b73fc8f1b42ff04f77996ed8210c297fcfa28ca880ff0a217 1.79kB / 1.79kB
0.0s
=> sha256:b273004037cc3af245d8e08c4bfa672b93ee7dcb289736c82d0b58936fb71702 7.81kB / 7.81kB
0.0s
=> sha256:001c52e26ad57e3b25b439ee0052f6692e5c0f2d5d982a00a8819ace5e521452 55.00MB / 55.00MB
43.7s
=> sha256:d9d4b9b6e964657da49910b495173d6c4f0d9bc47b3b44273cf82fd3273d165 5.16MB / 5.16MB
10.9s
=> sha256:2068746827ec1b043b571e4788693eab7e9b2a95301176512791f8c317a2816a 10.88MB / 10.88MB
15.2s
=> sha256:9dae329d35993868ef75ac8b7c6eb407fa53abbc3a264c218c2ec7bca716e6 54.58MB / 54.58MB
76.5s
=> sha256:d85151f15b6683b98f21c3827ac545188b1849efb14a1049710ebc4692de3dd5 5.42MB / 5.42MB
27.5s
=> sha256:52a8c426d30b691c4f7e8c4b438901ddeb82ff80d4540d5bbd49986376d85cc9 210B / 210B
29.3s
=> sha256:8754a66e005839a091c5ad0319f055be393c7123717b1f6fee8647c338ff3ceb 105.92MB / 105.92MB
181.9s
=> extracting sha256:001c52e26ad57e3b25b439ee0052f6692e5c0f2d5d982a00a8819ace5e521452
56.3s
=> extracting sha256:d9d4b9b6e964657da49910b495173d6c4f0d9bc47b3b44273cf82fd3273d165
2.7s
=> extracting sha256:2068746827ec1b043b571e4788693eab7e9b2a95301176512791f8c317a2816a
3.3s
=> extracting sha256:9dae329d35993868ef75ac8b7c6eb407fa53abbc3a264c218c2ec7bca716e6
34.2s
=> extracting sha256:d85151f15b6683b98f21c3827ac545188b1849efb14a1049710ebc4692de3dd5
3.4s
=> extracting sha256:52a8c426d30b691c4f7e8c4b438901ddeb82ff80d4540d5bbd49986376d85cc9
4.0s
=> extracting sha256:8754a66e005839a091c5ad0319f055be393c7123717b1f6fee8647c338ff3ceb
40.3s
[2/2] ADD target/spring-boot-docker.jar spring-boot-docker.jar
=> exporting to image
=> exporting layers
=> writing image sha256:2f39cde9520f2310a943fc39f4db0d09b73469882573a204a22a00e0075bbf78
0.7s
=> naming to docker.io/library/spring-boot-docker.jar
0.5s
    
```

Fig. 4. Creating Docker image of Micro service

```

C:\Windows\system32\cmd.exe - docker run -p 9090:8090 spring-boot-docker.jar
=> extracting sha256:52a8c426d30b691c4f7e8c4b438901ddeb82ff80d4540d5bbd49986376d85cc9
0.0s
=> extracting sha256:8754a66e005839a091c5ad0319f055be393c7123717b1f6fee8647c338ff3ceb
40.3s
[2/2] ADD target/spring-boot-docker.jar spring-boot-docker.jar
=> exporting to image
=> exporting layers
=> writing image sha256:2f39cde9520f2310a943fc39f4db0d09b73469882573a204a22a00e0075bbf78
4.8s
=> naming to docker.io/library/spring-boot-docker.jar
0.5s

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them

C:\Users\Admin\deploy\appworkspaces\dcokertest>docker images
REPOSITORY          TAG          IMAGE ID          CREATED          SIZE
spring-boot-docker.jar  latest      2f39cde9520f     5 minutes ago   544MB
tomcat               latest      6dd589e60602     7 days ago     474MB

C:\Users\Admin\deploy\appworkspaces\dcokertest>docker run -p 9090:8090 spring-boot-docker.jar
    
```

Fig. 5. Shows installed Docker file

b) Setup application in Cloud Environment

Elastic container service provides us an environment to deploy microservice. It is a fully managed container orchestration service. In ECS multiple Docker container can be run without any additional configuration. Cloud provides multiple environments to deploy application that can easily monitor and scaled. This application is set up with the following software which all is open source.

Table 1. Shows Version Configuration of Microservices

Environment Name	Version
------------------	---------

PERFORMANCE ANALYSIS OF MICROSERVICES BEHAVIOR IN CLOUD VS CONTAINERIZED DOMAIN
BASED ON CPU UTILIZATION

Java	8+
Linux	Kernel 5.10 AMI
Docker	21H2
Spring Boot	2.7.7(SNAPSHOT)
API	REST

An Elastic computing cloud was set up with the following hardware and network to access requests from the server.

Table 2: Shows Instance Configuration on cloud for microservice.

Storage	8 GB
RAM	8 GB
Private IP	Yes
Public IP	Yes
Protocols	SSH and TCP(Anywhere)

PERFORMANCE ANALYSIS OF MICROSERVICES BEHAVIOR IN CLOUD VS CONTAINERIZED DOMAIN BASED ON CPU UTILIZATION

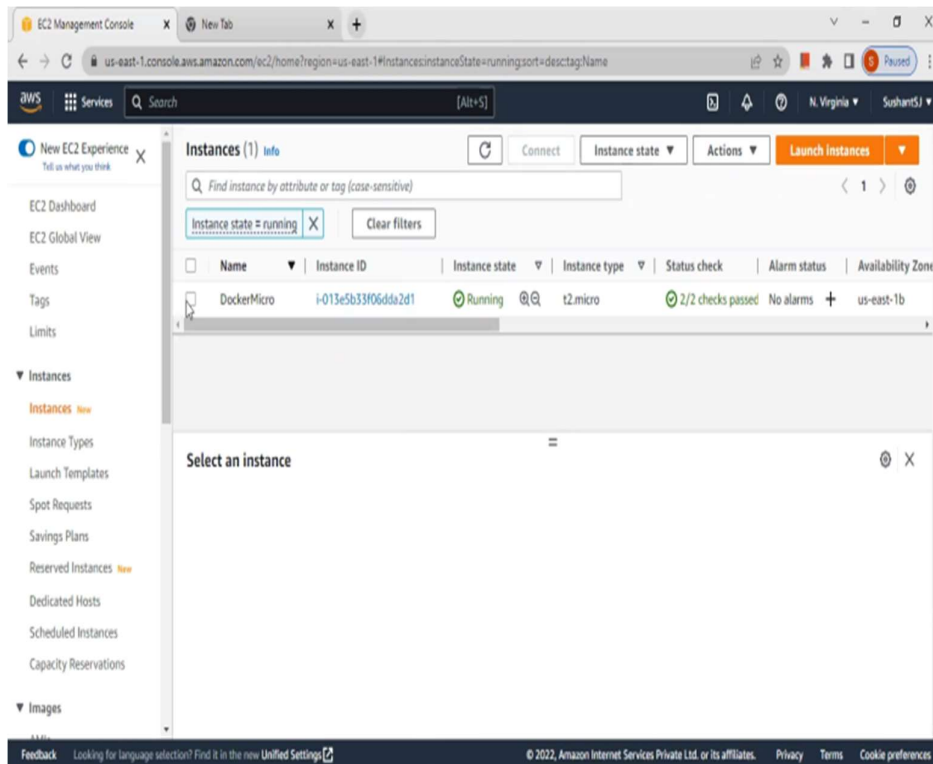


Fig 6. Shows status of instance running on elastic computing cloud.

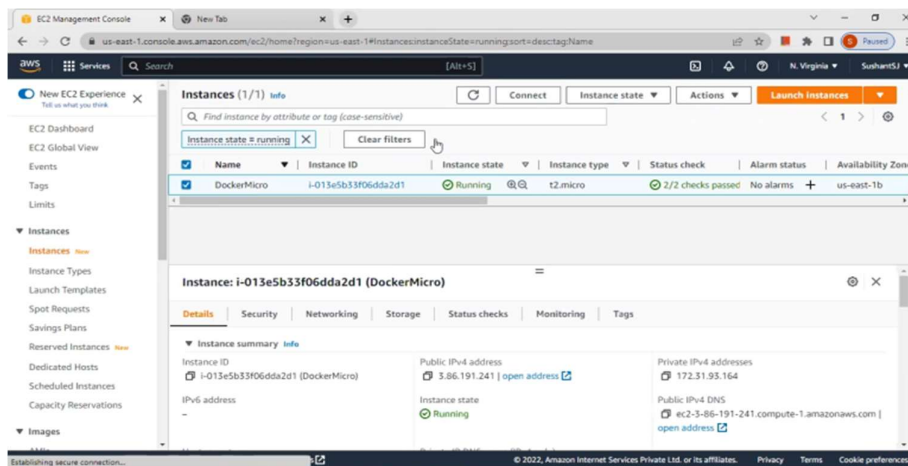


Fig. 7. Status of instance with public IP, private IP, Availability zone and state of instance.

In above Fig. 7, Setup of instance had shown. An elastic compute environment has been launched and running to monitor the behavior of microservice in cloud environment. Inbound rule is setup in security protocol to allow incoming network request from different network. Based on incoming request, the behavior of microservice is analyzed. CPU Utilization shows the behavior of application in multiple network input request.

PERFORMANCE ANALYSIS OF MICROSERVICES BEHAVIOR IN CLOUD VS CONTAINERIZED DOMAIN BASED ON CPU UTILIZATION

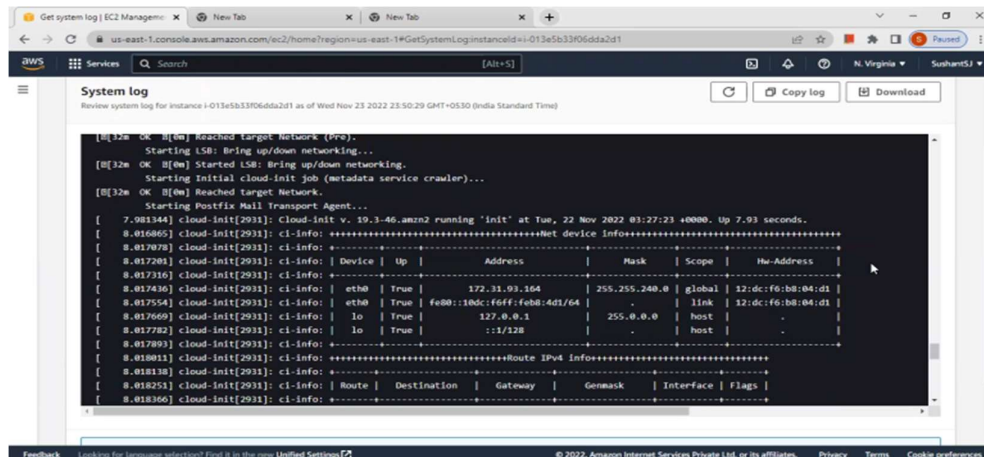


Fig. 8. Shows logs based on given private IP in console provide by elastic computing cloud.

System logs are generated to monitor the performance over IP. and display the analytics which can use further if required as shown in Fig. 8. Microservice is being analyzed in multiple intervals via Cloud Watch monitoring such as 12 hours, 24 hours and 72 hours on scaling parameters. Three parameters are using in this paper to monitor the performance.

1. CPU Utilization: The proportion of the instance's assigned EC2 compute units that are currently in use This measure shows how much processing power is needed to run a particular application on a chosen instance [14].

Unit: Percent

CLI Command to test utilization:

```
aws cloudwatch get-metric-statistics --namespace AWS/EC2 --metric-name CPUUtilization \
--dimensions Name=InstanceId,Value=i-013e5b33f06dda2d1 --statistics Maximum \
--start-time 2022-11-21T16:00:00 --end-time 2022-11-23T18:00:00 --period 360
```

Above formula have been used to calculate CPU utilization in aws CLI.

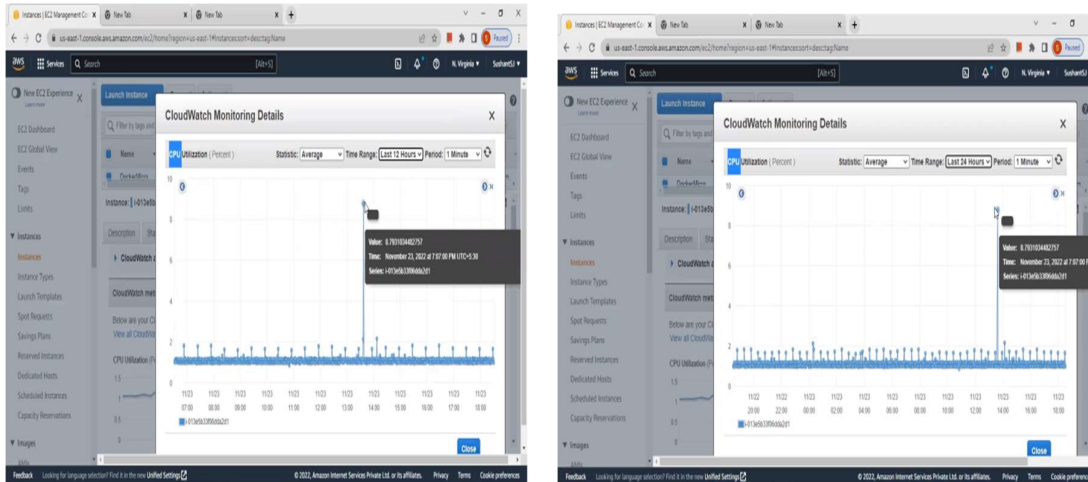
2. Network in: -The total amount of data that the instance received across all network interfaces. This measure shows how much network traffic is coming into a single instance[14].

Unit: Bytes

3. Network Out: -The total amount of bytes delivered across all network interfaces by the instance. This measure shows how much network traffic leaves a single instance [14].

Unit: Bytes

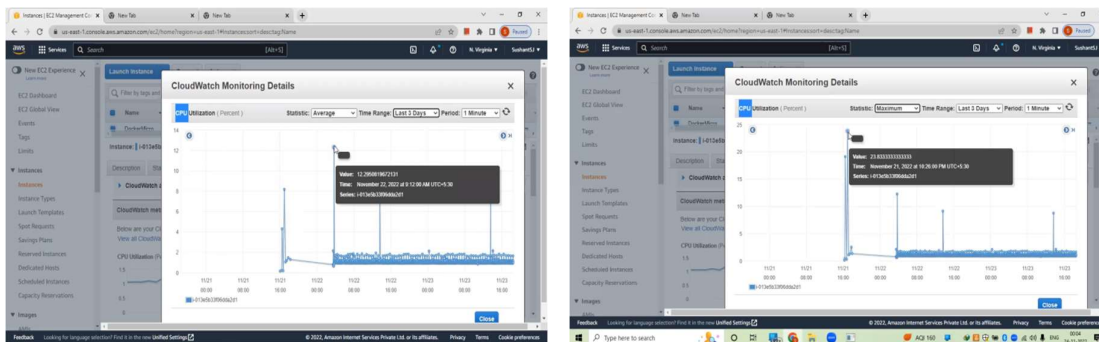
PERFORMANCE ANALYSIS OF MICROSERVICES BEHAVIOR IN CLOUD VS CONTAINERIZED DOMAIN BASED ON CPU UTILIZATION



(a)

(b)

Fig. 9. Result CPU utilization in different interval based on application performance (a) shows performance of application in 12 hours of interval (b) shows performance of application in 24 hours of interval.



(a)

(b)

Fig. 10. Result CPU utilization in different interval based on application performance (a) shows performance of application in 72 hours of interval (b) shows performance of application for maximum time of interval.

Monitoring of application had done with cloud watch, shows in Fig. 9 and Fig. 10. Analysis of application had done in different time zone over different amount of request. To analyze load, request came from external sources, which are added in Network-In simultaneously response is generated in terms of Network-Out.

Table 3: Shows Configuration of Microservices on Docker Container.

Storage	10 GB
RAM	8 GB
Private IP	Yes

PERFORMANCE ANALYSIS OF MICROSERVICES BEHAVIOR IN CLOUD VS CONTAINERIZED DOMAIN
BASED ON CPU UTILIZATION

Public IP	Yes
Protocols	SSH and TCP

Docker is a containerization platform that allows packaging an application and its dependencies in a container, which can then be run on any machine with a Docker runtime [15]. This can be especially useful when deploying microservices, as it allows packaging each service in its own container and running them independently on a single machine or across a fleet of servers. Docker helps to ensure that the environment in which a service is running remains consistent regardless of where it is deployed. This makes application easier to develop and test, and also makes it easier to deploy them to production, as can be sure that process will run in the same way regardless of the underlying infrastructure [13]. Docker had also scaled microservices horizontally by simply adding more containers into infrastructure. This can be done manually or through the use of a container orchestration platform such as Kubernetes.

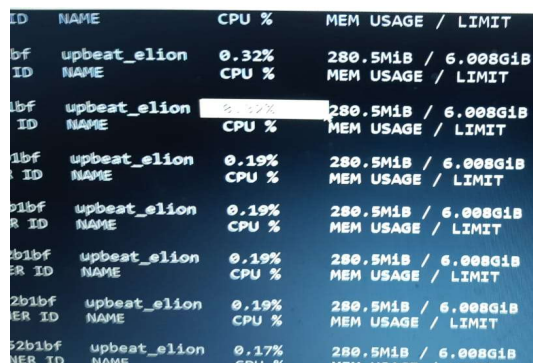


Fig.11. Shows deployment of microservices in Docker environment and display CPU utilization.

4. Result and discussion

4.1 Performance of CPU in elastic cloud computing

In this section, Performance of microservices has been evaluated based on CPU utilization. In elastic cloud computing environment, microservice application shows maximum CPU utilization of 23.33% on 106734510 network input and 355398 network output. CPU performance had calculated on multiple request which comes as Network In and response had shown in Network Out. Based on these request performance of CPU is displayed in below table.

Table 4: Shows performance of Micro services on AWS EC2

Duration	CPU Utilization (Percent)	Network In (Bytes)	Network Out (Bytes)
12 hours	8.7931034482757	193618	88.25
24 hours	8.7931034482757	193618	92871

PERFORMANCE ANALYSIS OF MICROSERVICES BEHAVIOR IN CLOUD VS CONTAINERIZED DOMAIN
BASED ON CPU UTILIZATION

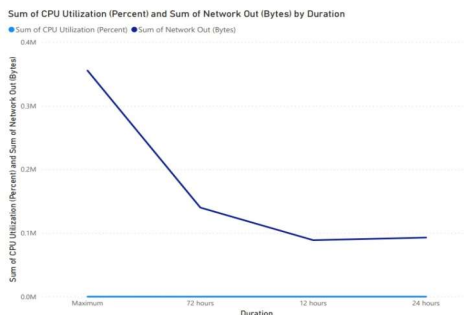
72 hours	12.2950819672131	2506742	139993
Maximum	23.8333333333333	10673410	355398

Performance of application was analyzed on different time interval and test CPU utilization. In second phase of implementation same microservices deployed on container and following result were generated.

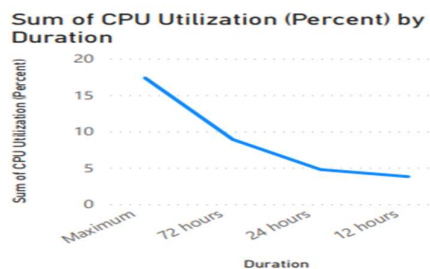
Table 5: Shows performance of Microservices on Docker Container

Duration	CPU Utilization (Percent)	Network In (KB)	Network Out (Bytes)
12 hours	3.795467	184.734	113.825
24 hours	4.765785	201.872	174.379
72 hours	8.90236	1953.258	1453.776
Maximum	17.3578	9564.102	7535.472

From Table 4 and Table 5, it can be verified that CPU utilization of application is less in containerized environment. This container was setup on Docker with fixed memory, because of lightweight nature of Docker, microservice shows very less utilization of CPU. Following are the graphical representation of CPU utilization. based on above result flowing result were generated.



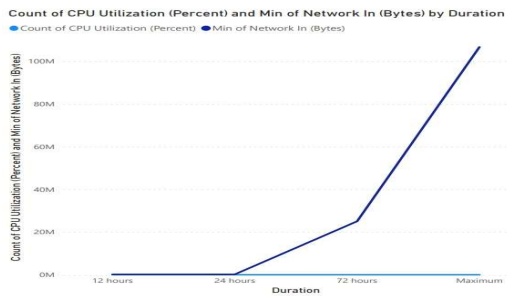
(a)



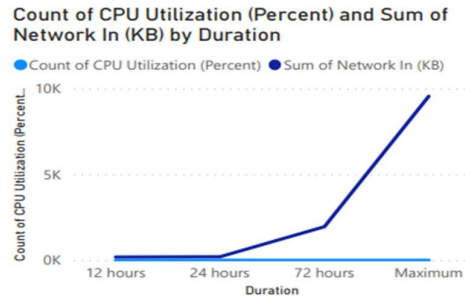
(b)

Fig. 12. Shows duration of CPU Utilization in different zone in elastic computing cloud (a) Shows CPU utilization duration and sum of network out in bytes (b) shows count of CPU utilization duration based on network count in bytes.

PERFORMANCE ANALYSIS OF MICROSERVICES BEHAVIOR IN CLOUD VS CONTAINERIZED DOMAIN
BASED ON CPU UTILIZATION



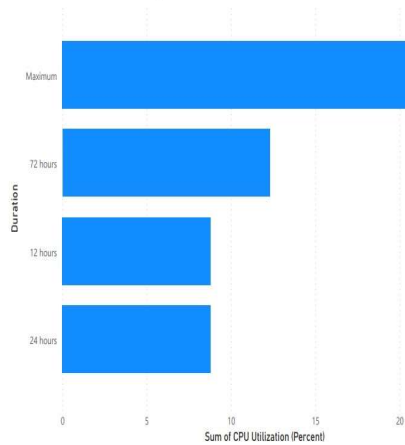
(a)



(b)

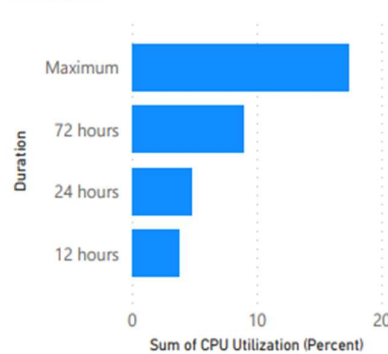
Fig. 13. Shows duration of CPU Utilization with respect to Network In and CPU utilization in docker domain (a) Shows CPU utilization duration based on network out in bytes (b) shows count of CPU utilization duration based on network count in kilobytes

Sum of CPU Utilization (Percent) by Duration



(a)

Sum of CPU Utilization (Percent) by Duration



(b)

Fig. 14. Shows sum of CPU Utilization in elastic computing cloud and docker environment (a) Shows sum of CPU utilization in regular time interval in cloud environment (b) shows sum of CPU utilization in regular time interval in containerized environment.

In above figures, performance of CPU utilization was observed in different time interval with different request over Network-In and Network-Out in cloud and containerized environment which can easily define that performance of application is better in containerized environment. In Docker request came in network is 9564.102(KB) and out is 7535.472 with CPU utilization is 17% while in cloud environment request came in network is 10673410 and out is 355398 with CPU utilization is 23% . All IP requests comes under network Input and accordingly network output was generated. In this paper, we majorly focused on CPU Utilization and network utilization at different time intervals to continuously monitor the performance of applications.

5. Future Scope

In this paper, the performance analysis microservice was done with cloud and Docker environments. In the cloud domain, Elastic computing environments were used while the same implementation will also be done in Kubernetes and AWS Fargate services. Serverless platforms can automatically scale microservices up or down based on demand, without the need for manual intervention. Serverless platforms offer a high level of security, with built-in measures such as network isolation and secure access controls. This can help to protect microservices from potential threats and vulnerabilities. A pipeline can be created for continuous integration and continuous deployment to scale up application in kubertnate and fargate environment for better result.

CONFLICT OF INTEREST

The authors declare no conflict of interest.

6. Reference

- [1] Al-Doghman, F., Moustafa, N., Khalil, I., Tari, Z. and Zomaya, A., 2022. AI-enabled Secure Microservices in Edge Computing: Opportunities and Challenges. *IEEE Transactions on Services Computing*.
- [2] S. N. Srirama, M. Adhikari, and S. Paul, "Application deployment using containers with auto-scaling for microservices in cloud environment," *J. Netw. Comput. Appl.*, vol. 160, no. February, p. 102629, 2020, doi: 10.1016/j.jnca.2020.102629.
- [3] S. Chhabra and A. K. Singh, "A Probabilistic Model for Finding an Optimal Host Framework and Load Distribution in Cloud Environment," *Procedia Comput. Sci.*, vol. 125, pp. 683–690, 2018, doi: 10.1016/j.procs.2017.12.088.
- [4] Kithulwatta, W.M.C.J.T., Jayasena, K.P.N., Kumara, B.T. and Rathnayaka, R.M.K.T., 2022. Integration With Docker Container Technologies for Distributed and Microservices Applications: A State-of-the-Art Review. *International Journal of Systems and Service-Oriented Engineering (IJSSOE)*, 12(1), pp.1-22.
- [5] H. Xu and B. Li, "Dynamic Cloud Pricing for Revenue Maximization," *IEEE Trans. Cloud Comput.*, vol. 1, no. 2, pp. 158–171, 2013, doi: 10.1109/TCC.2013.15.
- [6] P. Jain, Y. Munjal, J. Gera, and P. Gupta, "Performance Analysis of Various Server Hosting Techniques," *Procedia Comput. Sci.*, vol. 173, no. 2019, pp. 70–77, 2020, doi: 10.1016/j.procs.2020.06.010.
- [7] Telang, T., 2023. Containerizing Microservices Using Kubernetes. In *Beginning Cloud Native Development with MicroProfile, Jakarta EE, and Kubernetes* (pp. 213-230). Apress, Berkeley, CA.1.
- [8] Bao, L., Wu, C., Bu, X., Ren, N. and Shen, M., 2019. Performance modeling and workflow scheduling of microservice-based applications in clouds. *IEEE Transactions on Parallel and Distributed Systems*, 30(9), pp.2114-2129.
- [9] Saman, B., 2017. Monitoring and analysis of microservices performance. *Journal of Computer Science and Control Systems*, 10(1), p.19..

- [10] Coulson, N.C., Sotiriadis, S. and Bessis, N., 2020. Adaptive microservice scaling for elastic applications. *IEEE Internet of Things Journal*, 7(5), pp.4195-4202..
- [11] Cerny, T., Donahoo, M.J. and Trnka, M., 2018. *Contextual understanding of microservice architecture: current and future directions. ACM SIGAPP Applied Computing Review*, 17(4), pp.29-45.
- [12] Montesi, F. and Weber, J., 2016. Circuit breakers, discovery, and API gateways in microservices. arXiv preprint arXiv:1609.05830.
- [13] Wan, X., Guan, X., Wang, T., Bai, G. and Choi, B.Y., 2018. Application deployment using Microservice and Docker containers: Framework and optimization. *Journal of Network and Computer Applications*, 119, pp.97-109.
- [14] https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/viewing_metrics_with_cloudwatch.html
- [15] Manu, A.R., Patel, J.K., Akhtar, S., Agrawal, V.K. and Murthy, K.B.S., 2016, March. A study, analysis and deep dive on cloud PAAS security in terms of Docker container security. In *2016 international conference on circuit, power and computing technologies (ICCPCT)* (pp. 1-13). IEEE.
- [16] Jaramillo, D., Nguyen, D.V. and Smart, R., 2016, March. Leveraging microservices architecture by using Docker technology. In *SoutheastCon 2016* (pp. 1-5). IEEE..
- [17] Stubbs, J., Moreira, W. and Dooley, R., 2015, June. Distributed systems of microservices using docker and serfnode. In *2015 7th International Workshop on Science Gateways* (pp. 34-39). IEEE.
- [18] Alam, M., Rufino, J., Ferreira, J., Ahmed, S.H., Shah, N. and Chen, Y., 2018. Orchestration of microservices for iot using docker and edge computing. *IEEE Communications Magazine*, 56(9), pp.118-123.
- [19] Singh, S. and Singh, N., 2016, July. Containers & Docker: Emerging roles & future of Cloud technology. In *2016 2nd international conference on applied and theoretical computing and communication technology (iCATccT)* (pp. 804-807). IEEE.
- [20] Baresi, L., Quattrocchi, G. and Tamburri, D.A., 2022. Microservice Architecture Practices and Experience: a Focused Look on Docker Configuration Files. arXiv preprint arXiv:2212.03107.
- [21] Al-Debagy, O. and Martinek, P., 2018, November. A comparative review of microservices and monolithic architectures. In *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)* (pp. 000149-000154). IEEE.
- [22] Jhingran, S. and Rakesh, N., 2021, July. Performance factor impacting behavior of microservices in various hosting domains. In *2021 Fourth International Conference on Computational Intelligence and Communication Technologies (CCICT)* (pp. 160-164). IEEE.