International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 6s
DOI: https://doi.org/10.17762/ijritcc.v11i6s.6959
Article Received: 30 March 2023 Revised: 16 May 2023 Accepted: 31 May 2023

Performance Analysis of Microservices Behavior in Cloud vs Containerized Domain based on CPU Utilization

Sushant Jhingran[1], Nitin Rakesh[2]
[1]Department of Computer Science & Engineering
Sharda University
Greater Noida, India
sushantjhingran@gmail.com
[2]Department of Computer Science & Engineering
Sharda University
Greater Noida, India
Nitin.rakesh@gmail.com

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;

I. INTRODUCTION

The process of application deployment consists of several stages, including developing applications using design patterns and deploying them on appropriate servers. Microservices can be deployed in various environments, and application deployment can be carried out in multiple hosting environments. Each hosting environment has specific parameters for assessing application performance and behavior. Microservices are commonly utilized in service industries due to their lightweight characteristics. They can be deployed on the cloud, where their performance can be measured using various metrics. The cloud provides a distinct serverless computing environment, which includes services such as EC2, ECR, Codestar, and other platform-specific services [2]. The performance of microservices can be evaluated using metrics available in a serverless cloud environment. Cloud computing employs a serverless approach for application deployment, ensuring that applications do not experience temporary interruptions by utilizing Auto Scaling features offered by different cloud environments. Serverless computing is also utilized for deploying applications based on microservices. To facilitate the deployment of microservice-based applications on the cloud, environments like Elastic Container Registry and Docker can be created [3][4]. In the deployment process, an image is generated, and a container is initiated to run the application, which results in a lightweight application. Monitoring an application involves tracking metrics such as network utilization and CPU utilization, which determine its performance. Cloud vendors offer various serverless approaches, including S3 (Simple Storage Services), RDS (Relational Database Services), and ECS (Elastic Container Services). In the cloud realm, application monitoring can be conducted from different availability zones [5]. Cloud vendors offer availability zones, such as us-east 1a, which enable users to create multiple accounts and access data while monitoring it. In contrast, traditional hosting of applications involves utilizing servers provided by various service providers [6]. To enhance performance, applications are deployed in the form of packaged files. These files, such as Jar or War files, are created by developers and contain a combination of hosted servers with different capabilities. Applications can be deployed on various hosting environments, including Linux and cloud-based

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 6s
DOI: https://doi.org/10.17762/ijritcc.v11i6s.6959
***Article Received**: 30 March 2023 **Revised**: 16 May 2023 **Accepted**: 31 May 2023*

hosting. Bundle files are deployed on cloud-based environments using technologies like Docker and Kubernetes, which ensure application scalability [7]. The cloud environment offers elastic computing and Beanstalk services, which facilitate deployment along with monitoring and service notifications. Applications can be deployed on different hosting environments, including Linux and cloud-based hosting. Bundle files are deployed on cloud-based environments using technologies like Docker and Kubernetes, enabling application scalability [7]. The cloud environment offers elastic computing and Beanstalk services for deployment, along with monitoring and service notifications.

II. Literature Review

The performance of an application is influenced by its development structure. In the past, applications were developed using a modular approach, where smaller modules were created and interconnected. This modular approach allows for code reusability, which can be beneficial in future development. Web applications, being deployed on servers, occupy minimal memory space. A deployment environment, referred to as a web server, is used to easily deploy web applications. These applications can be accessed remotely through a browser on a different machine. When a user makes specific requests, the actual application is deployed on the server, and the response is fetched accordingly. Instead of deploying applications on multiple machines, developers typically choose to deploy them on a single server, enabling users to access web portals from remote machines. Consequently, applications are typically installed on one server and instantiated by multiple machines [8]. Applications were traditionally developed using a monolithic architecture in the past [9]. Controllers handle incoming requests from the front end in an application. These requests are then forwarded to the service layer, and if necessary, the service layer communicates with the Dao layer. All the layers or modules access a single database, meaning they are contained within a single container or code base that consists of multiple modules. These modules are bundled into a single JAR file, which can be deployed on a server for client access.

Running multiple instances of a single application creates a monolithic architecture, where the instances cannot communicate with each other. In a monolithic architecture, the components of an application are tightly coupled within a single module. The user interface, business logic, and data storage are tightly integrated and built as a single unit. Changes to one part of the system can have an impact on the entire application.

On the other hand, microservices architecture is a different approach to building and organizing software systems. It involves breaking down an application into small, independent, and loosely coupled services. Each service has its own specific task and can be developed, deployed, and scaled independently of other services. Services communicate with each other over a network using lightweight protocols such as HTTP or gRPC.

Microservices architecture offers benefits such as easier testing, as each service can be tested separately, leading to a more efficient testing process. It also provides enhanced security, as security vulnerabilities in one service do not affect the entire application. Scaling can be more easily managed in a microservices-based architecture compared to monolithic applications. Additionally, extending the project in a monolithic architecture can become challenging.

In summary, microservices architecture provides increased flexibility and scalability in contrast to the conventional monolithic approach [10]. Making changes to APIs in monolithic applications can be challenging. Integrating different technologies can also pose difficulties in monolithic applications. A single error in a specific module can have a cascading effect, potentially causing the entire application to fail. However, monolithic applications benefit from lower network latency due to their single communication channel.

In contrast, microservices are independent modules that operate concurrently. Each microservice can communicate with others through a service layer using lightweight protocols [11]. The system employs decoupled modules that utilize separate databases based on their respective services. These services can communicate with one another using REST or JSON. The services are loosely coupled due to their independent code bases. This allows for individual updates to be made to services without causing scalability problems. For example, if service X is being updated, the other services can continue running smoothly, and once the update is complete, service X will automatically integrate without impacting the entire project. Furthermore, even if microservices are developed using different domains, they can easily communicate with each other.

A. Architecture of Micro services

A microservices architecture typically comprises various interconnected components that collaborate to deliver the overall functionality of the system. Several essential components of such an architecture include:

Services: These are distinct modules within the system that perform specific tasks. Each service operates as an independent unit of functionality, capable of being developed, deployed, and scaled autonomously.

Service Registry: This serves as a centralized repository responsible for tracking all services within the system and their respective locations. Services utilize the service registry to discover other services and register themselves when they become available.

API Gateway: This component acts as a single point of entry for all incoming requests to the system. The API gateway

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 6s
DOI: https://doi.org/10.17762/ijritcc.v11i6s.6959
***Article Received**: 30 March 2023 **Revised**: 16 May 2023 **Accepted**: 31 May 2023*

manages request routing to the appropriate services, handles tasks like authentication, authorization, rate-limiting, caching, and performs other necessary functions.

Message Bus: This is a messaging system enabling asynchronous communication among services. Rather than directly calling one another, services utilize the message bus to exchange messages.

Load Balancer: The load balancer evenly distributes incoming requests across multiple instances of a service. It helps ensure system availability and responsiveness, especially during periods of high load.

Database: Microservices are designed with loose coupling in mind. Consequently, each service possesses its own dedicated database, responsible for managing its own data. This means that data stored within one service is not directly accessible by other services.

Monitoring and Logging: Extensive monitoring and logging are vital in a microservices architecture. These practices enable the tracking of individual service health, performance, and facilitate troubleshooting when issues arise.

These components collaborate to create a flexible, scalable, and resilient architecture suitable for constructing large and complex systems. However, it's important to note that the implementation of a microservices architecture can vary based on project requirements and the technologies employed.

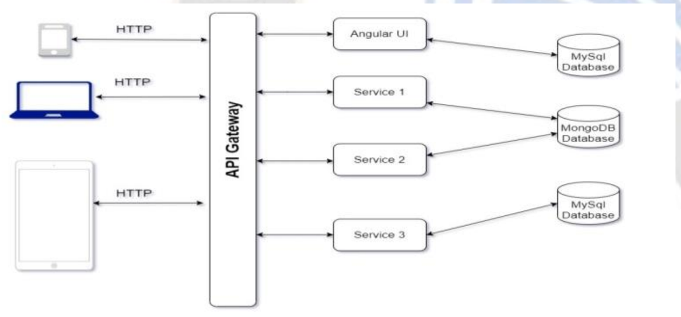


Figure 1. Architecture of microservices

The microservices architecture consists of multiple microservices that interact with each other through different ports. An API gateway serves as a central entry point that facilitates communication between the microservices and the client. Without the API gateway, the client would need to call each microservice independently, which would essentially treat them as separate projects [12]. The client interacts with a single URL, which represents the API gateway. The API gateway takes care of routing the requests to the appropriate microservices that access the backend services. In order to handle fault tolerance, the system utilizes the Hystrix library, which can manage situations when a service is unavailable.

This architecture incorporates the Eureka service discovery pattern, where microservices can register themselves and be discovered when needed. Communication between microservices can occur through HTTP or by exchanging data in the JSON format. To containerize the microservices application, the process involves creating Docker images and containers. This allows for the application to be packaged and run in a containerized environment [13]. To deploy a microservice application, it can be packaged into a Docker container using a Dockerfile. IDEs or initializers are often used to create microservice-based application. During the creation of the microservice, only a REST API is typically required. Once the application is set up, the respective paths are configured, and the Docker console is initiated for deployment. Each microservice is owned and maintained by a small, cross-functional team, which is responsible for its development, deployment, and operation. Microservices are designed with loose coupling in mind, meaning they can function independently without strong dependencies on other services. They should be designed to be automatically deployable and easily scalable in a continuous delivery environment. Microservices should emit metrics, traces, and logs to provide engineers with a comprehensive view of the system's current state. Organizing a microservices architecture often involves using a service registry. Services register themselves when they start up and deregister when they shut down. Clients of the services can then utilize the service registry to discover the location of a service at runtime. Additionally, the service registry can store metadata about the services, such as their version, status, and health.

III. Comparative Analysis of Various Hosting Enviornemnt

Microservice applications are implemented in a variety of environments, including cPanel, virtual private servers, and multiple cloud providers. Several parameters can be employed to monitor the performance of these applications. Monitoring parameters are utilized to assess and analyze the behavior of the application within their specific cloud domains. Below some parameters are shown with different environments.

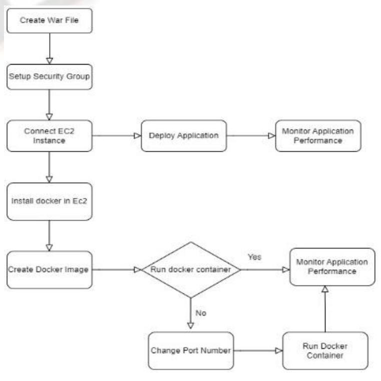


Figure 2. Process Flow chart

ISSN: 2321-8169 Volume: 11 Issue: 6s
DOI: https://doi.org/10.17762/ijritcc.v11i6s.6959
***Article Received**: 30 March 2023 **Revised**: 16 May 2023 **Accepted**: 31 May 2023*

Figure 2 illustrates the implementation flow, showcasing the utilization of multiple services in the application, including Product Service, Order Service, Notification Service, and Inventory Service. These services communicate with each other through both synchronous and asynchronous communication methods. Additionally, there are stateless services that do not interact with any databases. The Product Service leverages the use of Lombok to minimize the amount of boilerplate code. Furthermore, a NoSQL-based service has been established to facilitate communication between different databases.

A. Setup application in Docker Environment.

Docker demonstrates minimal reliance on the underlying operating system, offering flexibility and improved performance by dynamically managing network traffic through containers. Microservice applications in Docker utilize REST APIs for setup, while the web-dependent features of REST enhance microservices functionality. The Rest Controller enables access to the World Wide Web (WWW). Maven architecture facilitates the creation of JAR files. Docker containers are highly portable and capable of running on any system with Docker installed. This portability simplifies the deployment of microservices across a wide range of platforms. Leveraging Docker as part of a continuous delivery pipeline allows for automated packaging and deployment of microservices, thereby accelerating the release process.

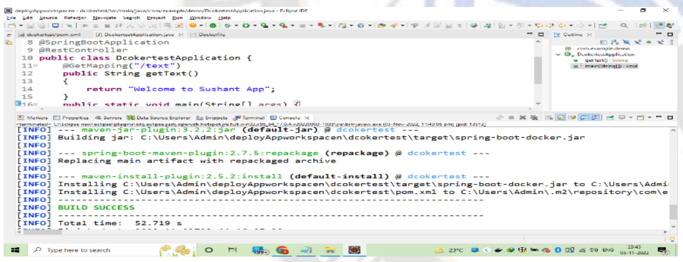

```
@SpringBootApplication
@RestController
public class DcokertestApplication {
    @GetMapping("/text")
    public String getText()
    {
        return "Welcome to Sushant App";
    }
    public static void main(String[] args) {
```
```
[INFO] --- maven-jar-plugin:3.2.2:jar (default-jar) @ dcokertest ---
[INFO] Building jar: C:\Users\Admin\deployAppworkspacen\dcokertest\target\spring-boot-docker.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:2.7.5:repackage (repackage) @ dcokertest ---
[INFO] Replacing main artifact with repackaged archive
[INFO]
[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\Admi
[INFO] Installing C:\Users\Admin\deployAppworkspacen\dcokertest\pom.xml to C:\Users\Admin\.m2\repository\com\e
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  52.719 s
```

Figure 3. Jar file creation in target location to deploy in docker with maven

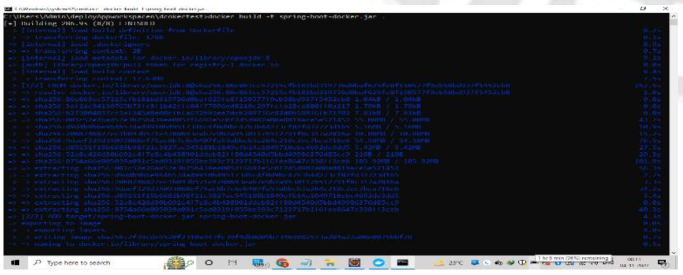
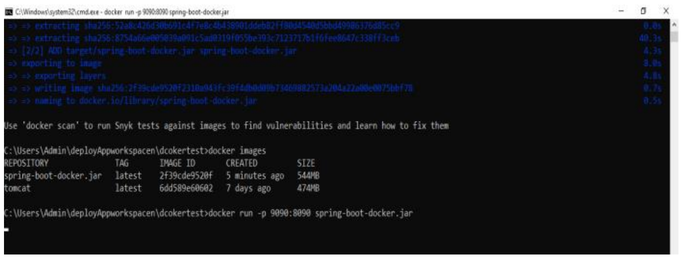

```
C:\Users\Admin\deployAppworkspacen\dcokertest>docker build -t spring-boot-docker.jar .
[+] Building 286.9s (8/8) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 178B
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load metadata for docker.io/library/openjdk:8
 => [auth] library/openjdk:pull token for registry-1.docker.io
 => [internal] load build context
[illegible]
 => [2/2] ADD target/spring-boot-docker.jar spring-boot-docker.jar
 => exporting to image
 => => exporting layers
[illegible]
 => => naming to docker.io/library/spring-boot-docker.jar
```

Figure 4. Creation of Docker image for Microservice


```
 => => extracting sha256:52a8c426d30b691c4f7e8c4b438901ddeb82ff80d4540d5bbd49986376d85cc9     0.0s
 => => extracting sha256:8754a66e005039a091c5ad0319f055be393c7123717b1f6fee8647c338ff3ceb    40.3s
 => [2/2] ADD target/spring-boot-docker.jar spring-boot-docker.jar                             4.3s
 => exporting to image                                                                         8.0s
 => => exporting layers                                                                        4.8s
 => => writing image sha256:2f39cde9520f2310a943fc39f4db0d0b7346988257a204a22a00e0075bbf78      0.7s
 => => 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\deployAppworkspacen\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\deployAppworkspacen\dcokertest>docker run -p 9090:8090 spring-boot-docker.jar
```

Figure 5. Docker setup in CLI

B. Setup application in Cloud Environment

Elastic container service provides us an environment to deploy microservice. This container orchestration service is fully managed. Without requiring any additional settings, ECS allows the running of many Docker containers. The cloud offers a variety of environments where applications can be deployed and quickly scaled. The software used to set up this application is entirely open source and is listed below.

Table 1. Shows Version Configuration of Microservices

Environment Name	**Version**
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	TCP(Anywhere)

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 6s
DOI: https://doi.org/10.17762/ijritcc.v11i6s.6959
***Article Received**: 30 March 2023 **Revised**: 16 May 2023 **Accepted**: 31 May 2023*

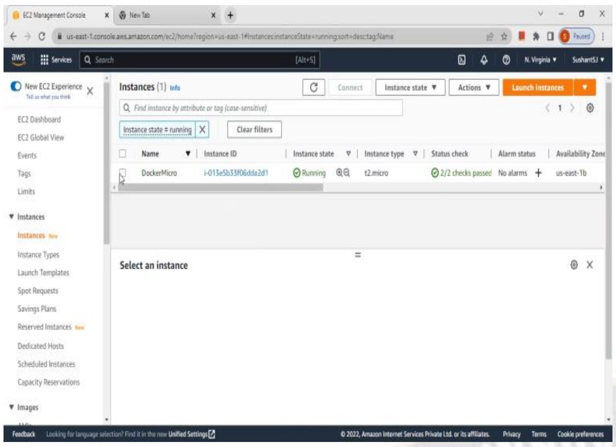


Figure 6. Shows status of instance running on elastic computing cloud

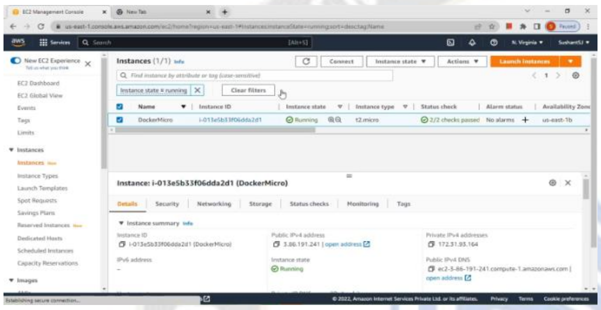
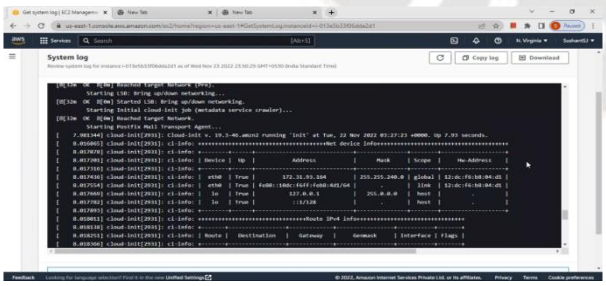


Figure 7. Status of instance with public IP, private IP, Availability zone and state of instance

In above Figure 7, Setup of instance had shown. An elastic computer environment has been launched and running to monitor the behavior of microservice in cloud environments. Security protocols' inbound rules are configured to accept incoming network requests from other networks. The behavior of the microservice is examined in response to the incoming request. CPU use demonstrates how application responds to various network input requests.



```
        Starting LSB: Bring up/down networking...
[0[32m  OK  0[0m] Started LSB: Bring up/down networking.
        Starting Initial cloud-init job (metadata service crawler)...
[0[32m  OK  0[0m] Reached target Network.
        Starting Postfix Mail Transport Agent...
[    7.981344] cloud-init[2931]: Cloud-init v. 19.3-46.amzn2 running 'init' at Tue, 22 Nov 2022 03:27:23 +0000. Up 7.93 seconds.
[    8.016065] cloud-init[2931]: ci-info: +++++++++++++++++++++++++++++++++++++++Net device info++++++++++++++++++++++++++++++++++++++++
[    8.017078] cloud-init[2931]: ci-info: +--------+------+-----------------------------+---------------+--------+-------------------+
[    8.017201] cloud-init[2931]: ci-info: | Device |  Up  |           Address           |      Mask     | Scope  |     Hw-Address    |
[    8.017316] cloud-init[2931]: ci-info: +--------+------+-----------------------------+---------------+--------+-------------------+
[    8.017436] cloud-init[2931]: ci-info: |  eth0  | True |        172.31.93.164        | 255.255.240.0 | global | 12:dc:f6:b8:04:d1 |
[    8.017554] cloud-init[2931]: ci-info: |  eth0  | True | fe80::10dc:f6ff:feb8:4d1/64 |       .       |  link  | 12:dc:f6:b8:04:d1 |
[    8.017669] cloud-init[2931]: ci-info: |   lo   | True |          127.0.0.1          |   255.0.0.0   |  host  |         .         |
[    8.017782] cloud-init[2931]: ci-info: |   lo   | True |           ::1/128           |       .       |  host  |         .         |
[    8.017893] cloud-init[2931]: ci-info: +--------+------+-----------------------------+---------------+--------+-------------------+
[    8.018011] cloud-init[2931]: ci-info: +++++++++++++++++++++++++++++++++Route IPv4 info++++++++++++++++++++++++++++++++++
[    8.018138] cloud-init[2931]: ci-info: +-------+-------------+-----------+---------------+-----------+-------+
[    8.018251] cloud-init[2931]: ci-info: | Route | Destination |  Gateway  |    Genmask    | Interface | Flags |
[    8.018366] cloud-init[2931]: ci-info: +-------+-------------+-----------+---------------+-----------+-------+
```

Figure 8. Shows logs based on a given private IP in console provided by elastic computing cloud.

System logs are generated to monitor the performance over IP. and display the analytics which can be used further if required as shown in Figure 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 used 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

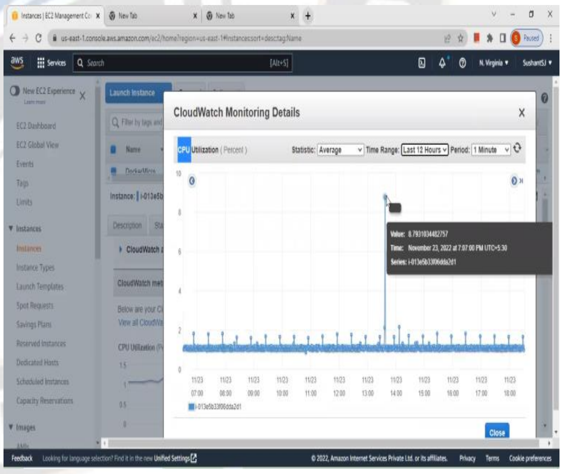


Figure 9. CPU utilization in 12 hours interval

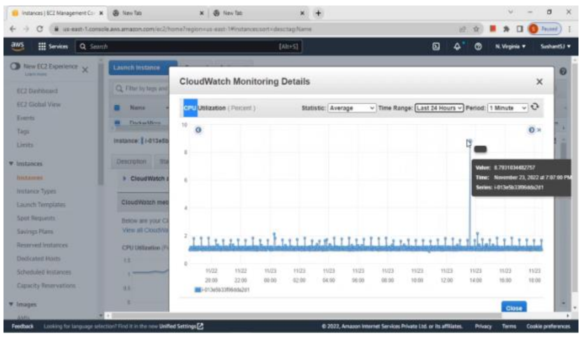
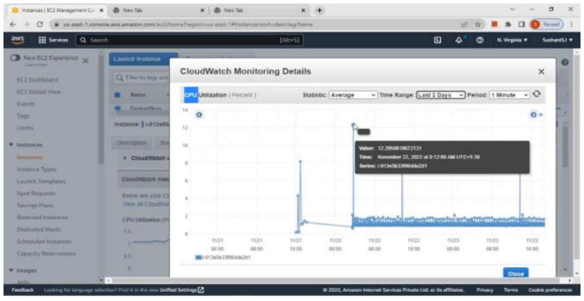


Figure 10. CPU utilization in 24 hours interval

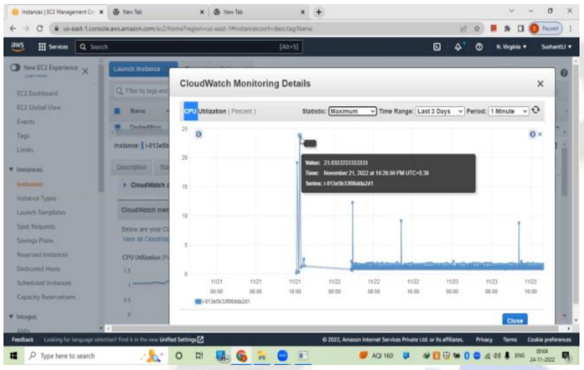
International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 6s
DOI: https://doi.org/10.17762/ijritcc.v11i6s.6959
***Article Received**: 30 March 2023 **Revised**: 16 May 2023 **Accepted**: 31 May 2023*


CloudWatch Monitoring Details

CPU Utilization (Percent)   Statistic: Average   Time Range: Last 3 Days   Period: 1 Minute

Value: 12.2950819672131
Time: November 22, 2022 at 9:12:00 AM UTC+5:30
Series: i-013e5b33f96dda2d1

Figure 11. Result CPU utilization in 72 hours of interval


CloudWatch Monitoring Details

CPU Utilization (Percent)   Statistic: Maximum   Time Range: Last 3 Days   Period: 1 Minute

Value: 23.8333333333333
Time: November 21, 2022 at 10:26:00 PM UTC+5:30
Series: i-013e5b33f96dda2d1

Figure 12. Result CPU utilization in maximum of interval

Figures 9, 10, 11, and 12 depict the application's monitoring, which was done with Cloud Watch. varied time zones and varied numbers of requests were used for the analysis of the application. Requests from outside sources are inserted in Network to analyse load, Response is generated simultaneously in terms of Network-Out.

Table 3: Shows Configuration of Microservices on Docker Container.

Storage	10 GB
RAM	8 GB
Private IP	Yes
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 enables the independent operation of each service on a single computer or across a fleet of servers by packaging each service in its own container. Regardless of where a service is deployed, Docker helps to ensure that the environment in which it is running remains consistent. Due to the certainty that the process will function uniformly regardless of the underlying infrastructure, this makes applications easier to create, test, and deploy to production [13]. Additionally, Docker had horizontally expanded microservices by merely adding more containers to the system. Both manual and a container orchestration tool like Kubernetes can be used for this.

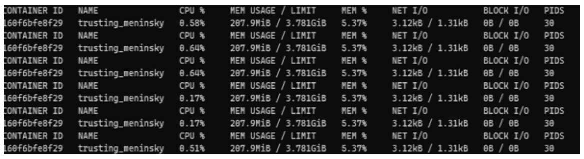

```
CONTAINER ID   NAME                CPU %     MEM USAGE / LIMIT     MEM %     NET I/O             BLOCK I/O   PIDS
160f6bfe8f29   trusting_meninsky   0.58%     207.9MiB / 3.781GiB   5.37%     3.12kB / 1.31kB     0B / 0B     30
CONTAINER ID   NAME                CPU %     MEM USAGE / LIMIT     MEM %     NET I/O             BLOCK I/O   PIDS
160f6bfe8f29   trusting_meninsky   0.64%     207.9MiB / 3.781GiB   5.37%     3.12kB / 1.31kB     0B / 0B     30
CONTAINER ID   NAME                CPU %     MEM USAGE / LIMIT     MEM %     NET I/O             BLOCK I/O   PIDS
160f6bfe8f29   trusting_meninsky   0.64%     207.9MiB / 3.781GiB   5.37%     3.12kB / 1.31kB     0B / 0B     30
CONTAINER ID   NAME                CPU %     MEM USAGE / LIMIT     MEM %     NET I/O             BLOCK I/O   PIDS
160f6bfe8f29   trusting_meninsky   0.17%     207.9MiB / 3.781GiB   5.37%     3.12kB / 1.31kB     0B / 0B     30
CONTAINER ID   NAME                CPU %     MEM USAGE / LIMIT     MEM %     NET I/O             BLOCK I/O   PIDS
160f6bfe8f29   trusting_meninsky   0.17%     207.9MiB / 3.781GiB   5.37%     3.12kB / 1.31kB     0B / 0B     30
CONTAINER ID   NAME                CPU %     MEM USAGE / LIMIT     MEM %     NET I/O             BLOCK I/O   PIDS
160f6bfe8f29   trusting_meninsky   0.51%     207.9MiB / 3.781GiB   5.37%     3.12kB / 1.31kB     0B / 0B     30
```

Figure 13. Shows deployment of microservices in Docker environment

and display CPU utilization.

IV. Result and Discussion

A. *Performance of CPU in elastic cloud computing*

This section examined the performance of a microservices-based application in terms of CPU usage. Microservice applications in an elastic cloud computing environment exhibit a maximum CPU utilization of 23.33% on 106734510 network input and 355398 network output. Multiple requests that came across the network were used to calculate CPU performance, and the results were displayed in network output. Based on this request, the performance of the CPU is shown in the table below.

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
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 the second phase of implementation the 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

Tables 4 and 5 show that the CPU utilization of the application is lower in a containerized environment. Due to Docker's lightweight design, this container was set up with fixed RAM and exhibits very low CPU utilization. The graphical representation of CPU utilization is shown below. Based on the outcome, flowing results were produced.

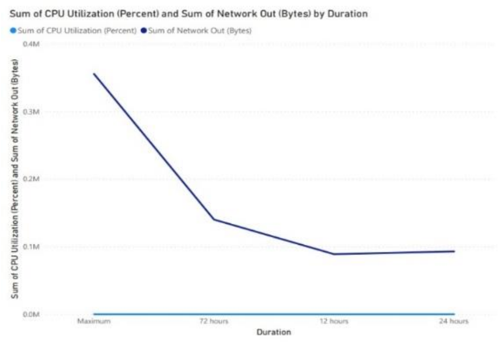


Figure 14. Shows CPU utilization duration and sum of network out in bytes.

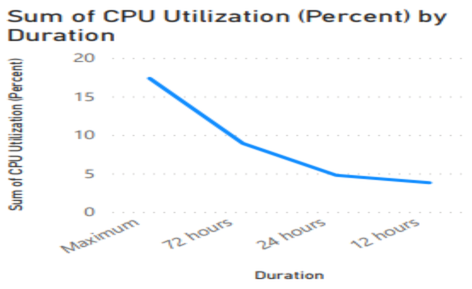


Figure 15 Shows count of CPU utilization duration based on network count in bytes.

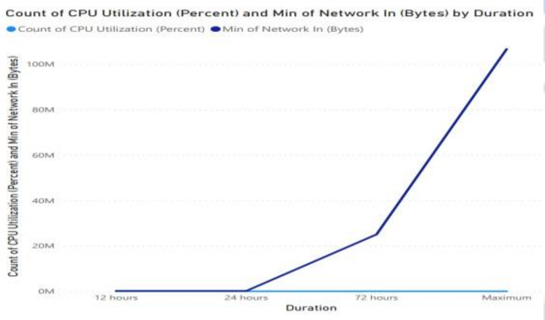


Figure 16 Shows CPU utilization duration based on network out in bytes.

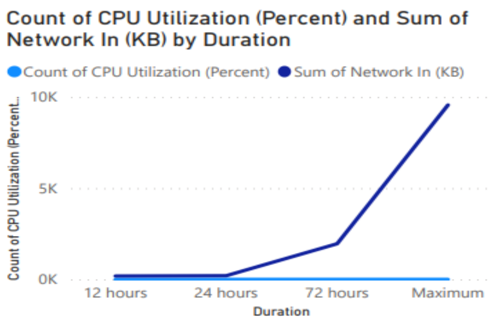


Figure 17 shows the count of CPU utilization duration based on network count in kilobytes.

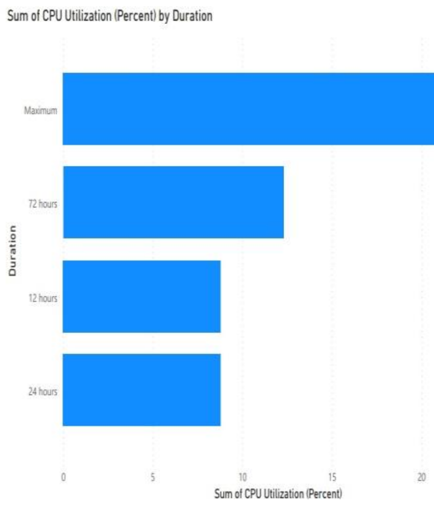


Figure 18 Shows sum of CPU utilization in regular time interval in cloud environment

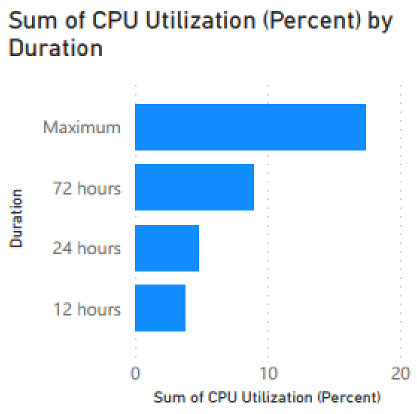


Figure 19 Shows sum of CPU utilization in regular time interval in containerized environment.

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 come 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.

V. FUTURE SCOPE

In this paper, the performance analysis microservice was done with cloud and Docker environments. Elastic computing environments were used in the cloud, and the same

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 6s
DOI: https://doi.org/10.17762/ijritcc.v11i6s.6959
***Article Received**: 30 March 2023 **Revised**: 16 May 2023 **Accepted**: 31 May 2023*

implementation will be used in Kubernetes and AWS Fargate services as well. Without the need for manual intervention, serverless platforms may automatically scale microservices up or down in response to demand. A high level of security is provided by serverless platforms to internal safeguards including network isolation and protected access controls. This can aid in defending microservices against potential dangers and weaknesses. To scale up an application in a Kubernetes and Fargate environment for better results, a pipeline for continuous integration and deployment can be built.

REFERENCES

[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] Adibatti, S. ., Sudhindra, K. R. ., & Manisha S., J. . (2023). 3 Phase Atrous Net with DCO-3DSPMRINET Model for Scoliosis Prediction. International Journal of Intelligent Systems and Applications in Engineering, 11(1), 79–91. Retrieved from https://ijisae.org/index.php/IJISAE/article/view/2446.

[13] Montesi, F. and Weber, J., 2016. Circuit breakers, discovery, and API gateways in microservices. arXiv preprint arXiv:1609.05830.

[14] 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.

[15] https://docs.aws.amazon.com/AWSEC2/latest UserGuide/viewing_metrics_with_cloudwatch.html

[16] 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.

[17] Jaramillo, D., Nguyen, D.V. and Smart, R., 2016, March. Leveraging microservices architecture by using Docker technology. In SoutheastCon 2016 (pp. 1-5). IEEE..

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.