

Microservices vs Monolithic Architecture: Load Testing in AWS on ReactJS Web Application for Performance

MSc Research Project
Programme Name

Parag Sharad Salunkhe
Student ID: 20224699

School of Computing
National College of Ireland

Supervisor: Jitendra Sharma

National College of Ireland
Project Submission Sheet
School of Computing



Student Name:	Parag Sharad Salunkhe
Student ID:	20224699
Programme:	Programme Name
Year:	2022
Module:	MSc Research Project
Supervisor:	Jitendra Sharma
Submission Due Date:	15/12/2022
Project Title:	Microservices vs Monolithic Architecture: Load Testing in AWS on ReactJS Web Application for Performance
Word Count:	5595
Page Count:	20

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	
Date:	15th December 2022

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	<input type="checkbox"/>
Attach a Moodle submission receipt of the online project submission , to each project (including multiple copies).	<input type="checkbox"/>
You must ensure that you retain a HARD COPY of the project , both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	<input type="checkbox"/>

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Microservices vs Monolithic Architecture: Load Testing in AWS on ReactJS Web Application for Performance

Parag Sharad Salunkhe
20224699

Abstract

Microservice and Monolithic cloud native architectures are in very demand right now. When many giants are migrating to microservice architecture researchers say monolithic architecture still use full for many reasons, We deployed a NodeJS monolithic application to EC2 and micro service architecture to the EKS cluster. An analysis of performance was made after putting the architectures under different test using the Locust load testing tool. Evaluation say that under less time the monolithic provided a very good response time whereas for high load microservice was able to stay stable and kept the response time much less compared to monolithic architecture.

1 Introduction

The use of cloud computing has increased significantly over the past few years, and that many businesses are moving to microservice cloud-based native architecture in order to take advantage of the benefits offered by the cloud. However, some businesses are only partially migrating to the microservice architecture, and are continuing to use older, monolithic architectures for some of their applications. This is because moving to the microservice architecture from monolithic can be a complex and challenging process, and many businesses are still experimenting with different architectures and deployment strategies in order to find the most efficient and effective way to use the cloud. While some businesses are moving to microservice cloud-native architecture, others are adopting a hybrid approach, in which some applications are run in the cloud and others are run on-premises. This allows businesses to take advantage of the benefits of the cloud while still using their existing infrastructure and systems. Ultimately, the best architecture and deployment strategy for a particular business will depend on its specific needs and requirements.

A monolithic application is a software application that consists of a single, large code-base that contains all the functionality of the application. This type of architecture is often seen in traditional enterprise applications, where all the different services and components of the application are tightly coupled and interdependent. While this approach has some advantages, such as simplicity and ease of deployment, it can also make it difficult to scale and maintain the application over time.. This means that all components of the system share the same resources, such as memory, databases, and files. Because of this

tight coupling, the components of a monolithic system are not independently executable. Monolithic architectures can be contrasted with microservice architectures, in which the system is composed of smaller, independently deployable components that communicate with each other through well-defined interfaces. This allows for greater flexibility and scalability, as individual components can be developed and updated independently of the rest of the system. However, monolithic architectures can still be effective for certain types of applications, particularly those that are relatively simple and do not require a high degree of flexibility or scalability.

Although microservice provides more benefits it is faced with several challenges as well. Microservice architecture can be complex to design, implement, and maintain. It requires a deep understanding of the system and its dependencies, as well as careful planning and coordination across teams. In a microservice architecture, individual services need to communicate and share data with each other. This can be challenging to implement and manage, especially at scale. In a monolithic system, dependencies are typically managed at the application level. In microservice architecture, dependencies need to be managed at the service level, which can be complex and time-consuming. In a microservice architecture, individual services can be developed and deployed independently. This can make it more challenging to test and deploy the system as a whole, as the services need to be tested and deployed together to ensure that they work correctly. In a microservice architecture, individual services can be scaled independently. This can be challenging to manage and optimize, as it requires careful planning and monitoring to ensure that the system is meeting the performance and scalability needs of the business.

Monolithic also faces several challenges like the entire application is deployed as a single unit. This means that the entire application must be scaled together, which can be challenging and may not provide the flexibility and scalability needed to meet the changing needs of the business. In a monolithic system, changes to the application must be made at the application level, which can be difficult and time-consuming. This can make it challenging to iterate and experiment with new features and functionality, which can limit the flexibility and adaptability of the system. Also the entire application must be tested and deployed together. This can be time-consuming and may require the application to be taken offline, which can be disruptive to the business. Monolithic systems can be complex to design, implement, and maintain. This can make it difficult for teams to work with and understand the system, which can impact productivity and efficiency. In a monolithic system, dependencies are typically managed at the application level. This can be challenging and may not provide the flexibility and control needed to manage dependencies effectively.

In contrast, the microservice architecture is a more modular and flexible approach to building applications. In this architecture, the application is divided into smaller, independent services that communicate with each other through well-defined interfaces. This makes it easier to scale and maintain the application, as each service can be scaled independently and changes can be made to individual services without impacting the entire system.

Many companies are starting to migrate their traditional, monolithic applications to the cloud, where they can take advantage of the scalability and flexibility of cloud computing. This typically involves deploying the application using infrastructure-as-a-service (IaaS) or platform-as-a-service (PaaS) solutions, which provide a flexible and scalable platform for deploying and running applications. In some cases, companies may choose to redesign their monolithic applications as microservices in order to take full advantage

of the benefits of cloud computing. Microservice architectures offer several benefits over monolithic architectures, particularly when it comes to scalability and flexibility. By decomposing a large, monolithic application into smaller, independent services, it becomes easier to scale and maintain the overall system. Each service can be scaled independently, allowing for better resource utilization and more efficient use of computing resources. Additionally, because the services are loosely coupled, changes can be made to individual services without impacting the rest of the system, making it easier to manage and maintain the overall application. However, implementing a microservice architecture can be challenging, as it requires significant effort to deploy and manage each individual service.

A web application is created in both a monolithic and a microservice architecture as part of the proposed study. The apps make use of the NodeJS Framework. Both should maintain to use the same infrastructure in order to produce reliable outcomes. For creating the application images, Docker Hub is bring used. AWS EKS is mostly used by us to create our cluster and get the image from Docker Hub. AWS CloudWatch is used to collect different metrics. The Locust tool is used to vary the load on the web application during testing of the API. We'll deploy monolithic applications using AWS EC2.

The following sections constitute the next part of the research. The findings of examined research publications that were relevant to our study were described in Section 2. the author's contribution, including the techniques used and the individual authors' discoveries of their varied results. The highlights of the authors work and evaluation are thoroughly covered. This section's subtopics talk about the technologies that made the project successful. The approach of the proposed project is covered in Section 3. The approach includes the architecture that was utilized and the steps we took to provide a valid evaluation. Section 4 goes into great detail on both architectures, how they interact with one another, and which AWS services they each utilize. Section 5, implementation of the methods discussed is being conducted. Different test cases are being implemented to get results. Metrics are collected for further evaluation. Use of the tools for conducting tests will be discussed. In Section 6, assessment is done using the metrics that were gathered from various test scenarios. After examining all of the test cases, an extensive discussion regarding the evaluation is made. In Section 7, we provide a summary of the research results and the objective we were able to accomplish. highlighting the main conclusions reached during the review and the limitations taken into consideration when creating the project. Finally, we address potential directions for future work and strategies to further our program objectives or strengthen the validity of the study's findings.

1.1 Research Question

Will the performance and latency be improved in microservice/cloud native architecture than the monolithic architecture by using NodeJS app in AWS?

2 Related Work

In this section, previous research related to topic of microservice and monolithic architecture will be discussed. Previous valuable findings and their critical evaluations and comparing it with each other.

2.1 Architecture migration, challenges faced and reason for migration related papers

According to Newman (2021), 70 percent of companies have migrated from a monolithic architecture to a microservice cloud-native architecture. Microservice architecture allows for independent scaling of individual services, which can improve overall system performance and availability. This is particularly useful in cloud environments, where resources can be easily added or removed to meet changing demand. Brunner et al. (2015) explains that with microservice architecture, individual services can be developed and deployed independently, allowing for greater flexibility in the development process. This can make it easier to iterate and experiment with new features and functionality. In contrast, a failure in a monolithic system can affect the entire system, whereas in a microservice architecture, failures are isolated to individual services, making the system more resilient and reducing the impact of failures. Kalske et al. (2017) studied the difficulties of migrating from a monolithic architecture to a microservice architecture and found that if the legacy architecture was monolithic, it was difficult to migrate, but it was still possible. The author suggested that businesses should only consider the transition if the benefits outweigh the costs. Bucchiarone et al. (2018) explains why Danske Bank migrated to a microservice architecture for a number of reasons, including simple deployment, centralized logging and monitoring, and the ability to act proactively on suspicious or faulty behavior. However, the adoption of microservices also comes with challenges, such as the time and effort needed to design and expand each microservice to the cloud. The author Mendonça et al. (2021) Istio, a company that was a collaboration between Google and IBM, initially used a microservice architecture for the control plane but later switched to a monolithic architecture based on feedback from users. This simplification improved installation, configuration, and debugging, as well as scalability and reduced start, upgrade, and removal times. The team believes the change has improved the lives of Istio users Shahid (2019) and Pianini and Neri (2021) discuss how DevOps, Kubernetes, Docker, and automatic scaling can be used to save time and resources, but may also require additional migration, development, and integration. Haugeland et al. (2021) suggests migrating monolithic applications to cloud-native microservice-based SaaS that is customizable, with a focus on enabling tenant-specific customization while ensuring data isolation and avoiding competition for resources. This approach allows for customization outside of the main application's execution context. Brunner et al. (2015) conducted a case study on the migration of a monolithic application to a microservice architecture in the cloud, and found that the benefits of the transition outweighed the costs and challenges. The author Bajaj et al. (2020) proposed a hybrid solution as an alternative to traditional monolithic applications. This approach allowed businesses to reap the benefits of both architectures simultaneously by only migrating selected sections of the application.

2.2 Architecture Comparison and Evaluation related papers

The goal of this study is to evaluate and compare the performance and cost-effectiveness of monolithic and microservice architectures in the context of cloud computing. Previous research conducted by authors Villamizar et al. (2016) and Al-Debagy and Martinek (2018), has provided valuable insights and motivation for this study. The author Villamizar et al. (2016), evaluated the performance, cost analysis, and response time of a web application were examined under three different scalability scenarios in order to compare

the expenses associated with its development and implementation. The first phase of the research consisted of conducting tests on all three architectures, analyzing the results, and comparing those findings with the costs of the underlying infrastructure for each of the three use cases. In the end, a third study was conducted to estimate the average response time while the architecture was being put through performance testing. This study showed how the architectures impacted response times for requests made during peak hours. The findings of the tests showed that the use of microservices led to a decrease in the amount of money spent on infrastructure compared to traditional monolithic architectures. Similarly, the author Al-Debagy and Martinek (2018), the performance of monolithic and microservice architectures was examined in more detail. The authors conducted simulations using a web application hosted on Azure, using the Apache Jmeter tool to simulate excessive traffic. The simulations involved a varying number of users, and the evaluation lasted for three minutes. It was observed that as the number of users increased, the CPU consumption and response time for the microservice architecture grew. This is because microservices require data from multiple services, which takes more time to process. The authors concluded that while microservices are useful for handling large numbers of users, they come at a higher cost due to the increased CPU and RAM usage. Overall, this study aims to build on the previous research by conducting a more in-depth analysis of the performance and cost-effectiveness of monolithic and microservice architectures in the cloud. The results of this study will provide valuable insights for companies looking to adopt either of these architectures for their web applications.

The author Akbulut and Perros (2019) conducted performance analysis on microservice architecture and came to a conclusion that adopting a microservice architecture can have a significant impact on the performance and scalability of a web application. To fully realize the benefits of this architecture, practitioners should follow a series of steps. The first step is to conduct unit testing to verify that the services work as intended. This can be done using automated testing frameworks such as NUnit or JUnit. Load testing should also be performed periodically to assess the application's topology and identify bottlenecks. It is important to use service-level agreements (SLAs) and have knowledge of user behavior to better characterize traffic patterns and perform more realistic tests. Architects can use simulators such as Hoverfly or Vagrant to evaluate different configurations without disrupting the ecosystem's traffic. Finally, testing the resiliency of the microservices can reveal potential infrastructure failures. Monitoring the system will provide valuable insights into its performance and scalability.

The author Tapia et al. (2020), the performance of both monolithic and microservice architectures was evaluated through stress testing, using equivalent levels of hardware resources for each architecture. The same web application was deployed using a monolithic architecture on a KVM virtual server, and a microservice-based architecture on containers. The evaluation included computational metrics such as CPU performance, disk reading and writing speed, memory, and network reception and transmission. The results showed that using a microservice architecture can be useful in cases where massive scaling is needed, as the services can be automated to improve response time.

The author Astyrakakis et al. (2019) evaluated a framework for automating the deployment of Kubernetes clusters on the OpenStack platform and validating cloud-native applications. The framework uses OpenStack as a hypervisor for building virtual machines and Firebase Database for storing settings and information. It also includes a tool for building Kubernetes clusters using custom automated scripts and the cloud-init application on custom Linux images, as well as a unique tool for validating microservice

cloud-native applications. the proposed framework revealed that its functionalities are completed in relatively low average times, compared to manual or semi-manual solutions. In specific experiments, the average completion time for the cluster deployment functionality was 5.7 minutes, which is significantly lower than what is typically required for such a task. Similarly, the average completion time for the application validation functionality was 11.4 minutes. These results suggest that the framework is effective in automating the deployment of Kubernetes clusters and applications. Similarly, the author Gupta et al. (2021) in his proposed method in this paper involves setting up a Kubernetes cluster and connecting multiple nodes to the internal application running on them. This manual approach to implementation allows for better organization and the ability to perform custom configurations that are not possible with preconfigured clusters offered by cloud platforms. The proposed work will also deploy an architecture that includes a website and an externally connected database. This will allow for scalability of resources across multiple nodes, as well as improved design and security. The database will be abstracted at the backend and only accessible through the website, protecting it from direct access by users or clients. Overall, this setup will provide better control and flexibility in managing the application and its associated resources. Similarly author Carroll et al. (2021) presents a solution for setting up and maintaining a scalable preproduction environment for integration testing. The proposed approach involves routing test traffic through preproduction deploys within the production cloud before they are released to production traffic. These preproduction deploys are isolated from the production system using staging databases that are copied daily from the production system. This provides sufficient isolation without the need for a complete staging environment. The technical foundations of this solution are the separation of the deploy step from the release step, and the extension of the microservice architecture to support multiple concurrent versions with independent traffic routing. This forms a cloud-native development (CND) architecture that can be used for regression testing immediately before production release. The testing is scalable and of high quality, as it is performed on the same system that will be released after a successful test run.

The author of Barczak et al. (n.d.) conducted research on the performance of monolithic and microservice architectures in a web application hosted on Azure. The Apache Jmeter tool was used to simulate excessive traffic, with simulations involving 50 users lasting for three minutes. The results showed that as the number of users increased, the CPU consumption and response time for the microservice architecture grew. This is because microservices require data from multiple services, which takes more time to process. The author concluded that while microservices are useful for handling large numbers of users, they come at a higher cost due to the increased CPU and RAM usage.

In contrast, the author of Balalaie et al. (2015) found that microservices can introduce new complexities to a system and may not always be the best solution for all cases. The decision to hire additional highly-trained software engineers and test professionals also contributed to the overall cost. However, microservices can also increase productivity and reduce costs in the long run due to their lower energy consumption. In general, no single microservice design was found to be superior in terms of effectiveness or efficiency, and the authors concluded that microservices have not yet reached their full potential.

3 Methodology

Node.js which JavaScript runtime is used to build web applications using a monolithic or microservice architecture. In a monolithic architecture, a single Node.js application is built and deployed as a single unit. This approach is suitable for smaller applications that do not require complex interactions between different components. In a Node.js monolithic architecture, the entire application is written in JavaScript and run on the Node.js runtime. This allows for a single codebase and a consistent development experience, but can make it difficult to scale and maintain the application as it grows in complexity. In a microservice architecture, on the other hand, the application is divided into smaller, independent services that communicate with each other through APIs. This approach is suitable for larger, more complex applications that require more flexibility and scalability. In a Node.js microservice architecture, each service can be written in Node.js and run on the Node.js runtime, allowing for a consistent development experience across the services. However, managing and deploying multiple services can be more complex than managing a single monolithic application. Overall, Node.js can be a powerful framework for building web applications using either a monolithic or microservice architecture. Its popularity and extensive ecosystem of libraries and tools make it a good choice for developers who want to build scalable and maintainable web applications. For both the applications we have 5 services, a) userManagement, b) dataManagement, c) atandanceManagement, d) salaryManagement and e) APIgateway.

In a monolithic architecture, a single application is built and deployed as a single unit. This means that the entire application, including the front-end, back-end, and database, is contained within a single instance. However, it is possible to use multiple instances with a monolithic architecture, but this is not typically necessary unless the application has a very large number of users or requires a high level of availability. Whereas, Microservice architecture application will have different pods for each service in the EKS, and with the use of ingress controller the traffic will be rerouted to any given service accessed by the user.

The microservice application is deployed on EKS, with the help of ingress controller we can reroute the traffic to services needed by the user. Each service will have its own pod in the cluster. For the monolith application, it is deployed in the EC2 cluster. After all the setup is ready on the AWS, Load testing tool called Locust can be setup on the local machine. Locust is an open-source load testing tool that allows to define user behavior in Python code and simulate a large number of users accessing your application simultaneously. It is designed to be easy to use and flexible, so it tests the performance of your application under various load conditions. To use Locust, will need to install it on your local computer and create a Python script that defines the user behavior you want to simulate. This script can include user actions such as logging in, accessing different pages, and submitting forms. Once your script is ready, you can run Locust and specify the number of users you want to simulate and the rate at which they should perform actions. Locust will then simulate the specified number of users and track their performance, such as the response time of your application and any errors that may occur. You can use the results of your load test to identify bottlenecks and other issues that may affect the performance of your application under heavy load. You can also use Locust's web interface to monitor the progress of your load test in real-time and view detailed statistics and graphs.

Amazon CloudWatch is a monitoring service provided by Amazon Web Services that

allows developers to monitor their Amazon EC2 instances and other AWS resources in real-time. With CloudWatch, developers can collect and track metrics, set alarms, and automatically react to changes in their AWS resources. This is important because it allows developers to ensure that their applications are running smoothly and efficiently. One of the key metrics that CloudWatch can help developers monitor is CPU utilization. CPU utilization is a measure of how much of a computer's processing capacity is being used at any given time. This is important because it can help developers understand how much of their resources are being used, and whether their applications are running optimally. Another important metric that CloudWatch can help developers monitor is CPU credit usage and CPU credit balance. CPU credits are a measure of the amount of processing power that an Amazon EC2 instance can use over a certain amount of time. CPU credit usage is a measure of how many CPU credits an instance has used, while CPU credit balance is a measure of how many CPU credits an instance has remaining. Monitoring these metrics can help developers understand the performance of their applications, and identify potential bottlenecks or other issues.

By monitoring key metrics like CPU utilization, CPU credit usage and CPU credit balance with the help of AWS CloudWatch, and using load testing tool Locust to simulate real-world usage scenarios, developers can gain valuable insights into the performance of their applications. This can help them identify potential bottlenecks and optimize the performance of their applications.

To make complications less before deploying applications to the cloud. Postman tool was used. Postman is a tool that allows developers to send and receive HTTP requests and responses. It was used for testing APIs to ensure that they are working as expected. Open the Postman application and create a new request. In the request, enter the URL for the application that you want to test. Select the HTTP method e.g like our application had GET, POST, PUT that were tested. The application will respond with a response, which will be displayed in the Postman application. This can help developer quickly and easily test multiple requests and scenarios.

4 Design Specification

4.1 Monolith Architecture

Monolithic architecture is supposed to be deployed to ec2. To deploy monolithic architecture to ec2 follow the steps:

- Create ec2 instance, we choose free tier instance, t2micro, which allots 1cpu only.
- install application on ec2 instance, connect to ec2 using ssh, we use yum package manger to install dependencies. Run 'Server.js" application script.
- Create and configure security group to control inbound and outbound traffic. The ports open for inbound traffic are 80,82,443. Outbound all ports.
- Ip will be assigned to the instance

After the instance is successfully launch we can further configure the instance for autoscaling group if needed. In this case there is no attachment of any auto scaling group. Select the instance created to check what cloudwatch is monitoring and by default cloudWatch monitors some statistics. By default every 5 minutes instance metrics are

gathered, if needed detailed statistics can be achieved by configuring detailed monitoring settings.

In the figure 5 we see the monoliths application architecture based on the deployment on AWS Cloud.

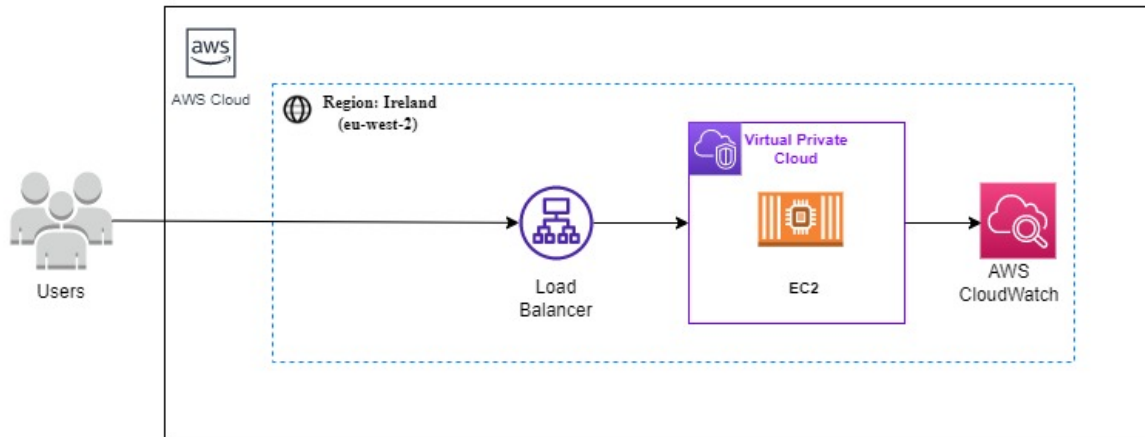


Figure 1: Monolithic Application Architecture on AWS Cloud

4.2 Microservice Architecture

To deploy Microservices application in EKS cluster we need to create k8's deployment files, Push the images from docker hub to EKS and deploy the EKS cluster. The images of the application are uploaded to docker hub with each service having its own container, in our case 5 containers. Following are the steps used for further deployment:

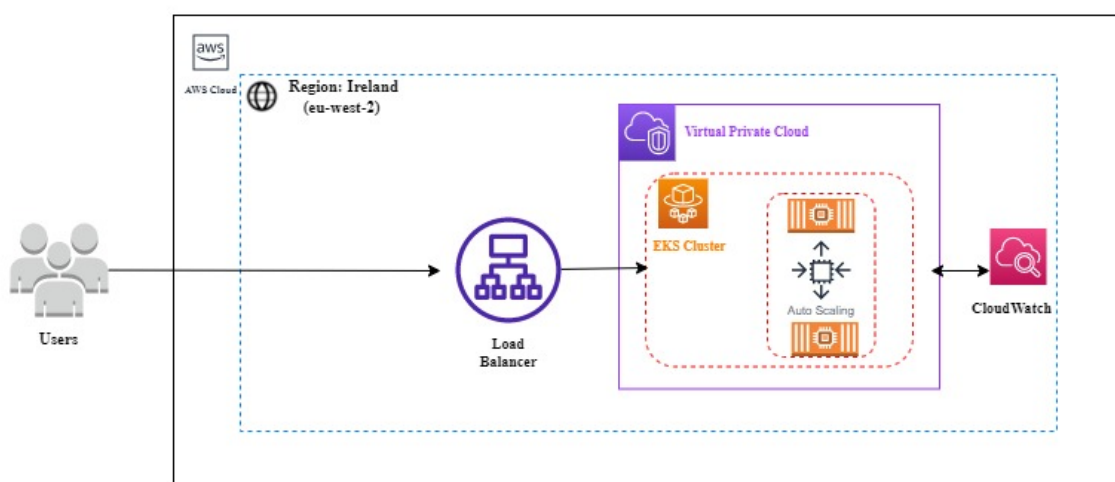


Figure 2: Microservice Application Architecture on AWS Cloud

- Created new user giving programmatic access
- 2 roles were created, one was EKS role. On role page , choose EKS service and select EKS cluster
- Role 2 was for Node group role , we assign 3 permissions to this role, these permissions help EC2 create nodes, pull images from dockerhub -

```
AmazonEKSEKSWorkerNodePolicy
AmazonEC2ContainerRegistryReadOnly
AmazonEKS_CNI_Policy
```

- Got to EKS from AWS dashboard, name the cluster , choose latest kubernetes version, select the role 1 created before
- configure network, subnets are automatically created , keep default security group,
- chose t3.medium, as resources were low for the applications pods below these configurations for ec2 instances .
- Create Node Group assigning it role 2 we made before, set minimum and maximum node size of 1 and 3, desired size 2.

The screenshot shows the AWS EKS console interface. The top section is titled 'Nodes (2) Info' and contains a search bar and a table with two rows of node information. The bottom section is titled 'Node groups (1) Info' and contains buttons for 'Edit', 'Delete', and 'Add node group', along with a table showing one node group.

Node name	Instance type	Node group	Created	Status
ip-172-31-45-49.eu-west-1.compute.internal	t3.medium	FirstNodeGroup	Created December 8, 2022, 17:54 (UTC+00:00)	Ready
ip-172-31-6-12.eu-west-1.compute.internal	t3.medium	FirstNodeGroup	Created December 8, 2022, 17:54 (UTC+00:00)	Ready

Group name	Desired size	AMI release version	Launch template	Status
FirstNodeGroup	2	1.23.13-20221112	-	Active

Figure 3: Nodes Created

- cluster is created and 2 instances of t3.micro are launched.
- the following command should respond on aws-cli app

```
aws sts get-caller-identity
```

- to connect kubectl with AWS use the following command where 'Research' is my clusters name. To get nodes write 2nd command below

```
aws eks --region eu-west-1 update-kubeconfig --Research
kubectl get nodes -o wide
```

- create deployment files . The yaml files of individual services would look like the following:

```
      apiVersion: apps/v1
kind: Deployment
metadata:
  name: usermanagement
spec:
  replicas: 1
  selector:
    matchLabels:
      app: usermanagement
  template:
    metadata:
      labels:
        app: usermanagement
    spec:
      containers:
        - name: user-backend
          image: parag2608/usermanagement:latest
          imagePullPolicy: IfNotPresent
---
apiVersion: v1
kind: Service
metadata:
  name: user-srv-cluster
spec:
  type: ClusterIP
  selector:
    app: usermanagement
  ports:
    - name: user-backend-cluster-ip
      protocol: TCP
      port: 3000
      targetPort: 3000
---
apiVersion: v1
kind: Service
metadata:
  name: user-lb-srv
spec:
  ports:
    - protocol: TCP
      port: 80
```

```

targetPort: 3000
selector:
  app: usermanagement
type: LoadBalancer

```

image url is taken from dockerhub .There are three objects in the yaml file, a) deployment b)service and a type 'LoadBalancer' which on EKS cluster will be rerouting all the traffic from port 80 to port 3000

- deploying services to EKS is made by next command and the 2nd command is used for accessing load balancer

```

kubectl apply -f <path-to-deployment-file>
kubectl get services

```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
apigateway-lb-srv	LoadBalancer	10.100.10.232	a8dd07f36ef094211bbb0370b6c8e514-1599625824.eu-west-1.elb.amazonaws.com	80:31116/TCP
apigateway-srv-cluster	ClusterIP	10.100.241.8	<none>	3000/TCP
attendance-lb-srv	LoadBalancer	10.100.73.172	a13c02eececf54c96a097542190de3fb-1765433170.eu-west-1.elb.amazonaws.com	80:31361/TCP
attendance-srv-cluster	ClusterIP	10.100.17.9	<none>	3000/TCP
backend-lb-srv	LoadBalancer	10.100.143.11	a53339cb183a143488688b7db4cbac43-1287242740.eu-west-1.elb.amazonaws.com	80:31648/TCP
backend-srv-cluster	ClusterIP	10.100.155.119	<none>	3000/TCP
data-lb-srv	LoadBalancer	10.100.96.58	a7dcfed7353f9441bb9ac380a7571848-748156671.eu-west-1.elb.amazonaws.com	80:31885/TCP
datamanagment-srv-cluster	ClusterIP	10.100.29.232	<none>	3000/TCP
kubernetes	ClusterIP	10.100.0.1	<none>	443/TCP
salarymanagement-lb-srv	LoadBalancer	10.100.19.105	a378402d3354842798f856bccacef00b-1909250752.eu-west-1.elb.amazonaws.com	80:30238/TCP
salarymanagement-srv-cluster	ClusterIP	10.100.117.186	<none>	3000/TCP
user-lb-srv	LoadBalancer	10.100.84.213	aee9247869e8a4a84827851118a0355a-1014631463.eu-west-1.elb.amazonaws.com	80:30080/TCP
user-srv-cluster	ClusterIP	10.100.233.161	<none>	3000/TCP

Figure 4: Images pushed to EKS cluster

Workloads: Pods (5) View details

Pod is the smallest and simplest Kubernetes object. A Pod represents a set of running containers on your cluster.

[Learn more](#)

default < 1 >

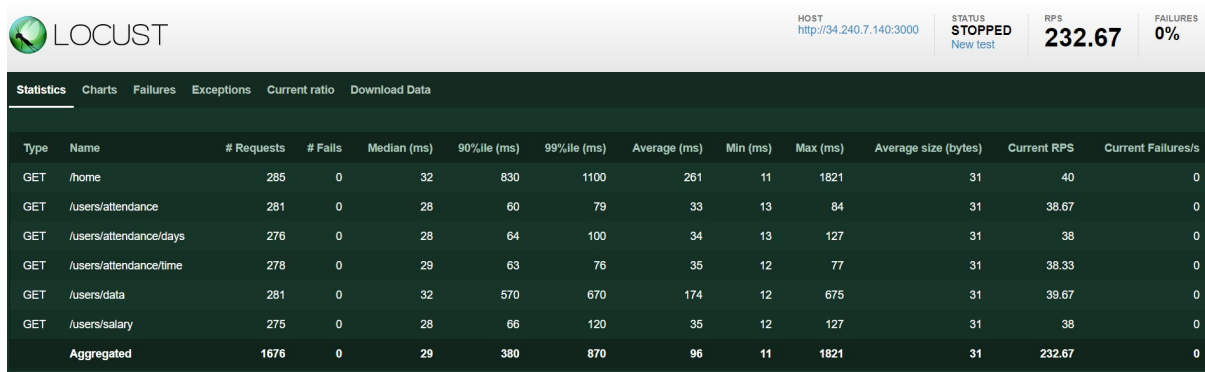
	Name	Age
<input type="radio"/>	apigateway-58d8fcc74-stcs5	Created 9 minutes ago
<input type="radio"/>	attendance-bd5bfc465-6c6tm	Created 9 minutes ago
<input type="radio"/>	datamanagment-b7945b4df-hsqm4	Created 9 minutes ago
<input type="radio"/>	salarymanagement-7bfcf9496c-fd924	Created 2 minutes ago
<input type="radio"/>	usermanagement-68c9fdcdb7-2j6zh	Created 9 minutes ago

Figure 5: Each service assigned a pod in the cluster

- For all the containers in dockerhub we deploy each service to EKS.

5 Implementation

Three test case scenarios implemented and evaluated. Locust will be used for swarming users at a given rate by giving the public IP that we get from aws. With the help of AWS cloud watch we can get metrics for the instances. From fig 6 we can see a preview of how after testing we can see number of requests and response time also for our test scenarios we will see the graph like in fig 7 which shoes graph of total requests, response time and number of users and see how with time the servers of both architecture respond. And from fig 8 and 9 we can see cpu credit usage spike during tests and cpu utilization.



The screenshot shows the Locust web interface. At the top right, it displays 'HOST: http://34.240.7.140:3000', 'STATUS: STOPPED (New test)', 'RPS: 232.67', and 'FAILURES: 0%'. Below this is a navigation menu with 'Statistics' selected. The main content is a table with the following data:

Type	Name	# Requests	# Fails	Median (ms)	90%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	/home	285	0	32	830	1100	261	11	1821	31	40	0
GET	/users/attendance	281	0	28	60	79	33	13	84	31	38.67	0
GET	/users/attendance/days	276	0	28	64	100	34	13	127	31	38	0
GET	/users/attendance/time	278	0	29	63	76	35	12	77	31	38.33	0
GET	/users/data	281	0	32	570	670	174	12	675	31	39.67	0
GET	/users/salary	275	0	28	66	120	35	12	127	31	38	0
Aggregated		1676	0	29	380	870	96	11	1821	31	232.67	0

Figure 6: Example of locust on testing

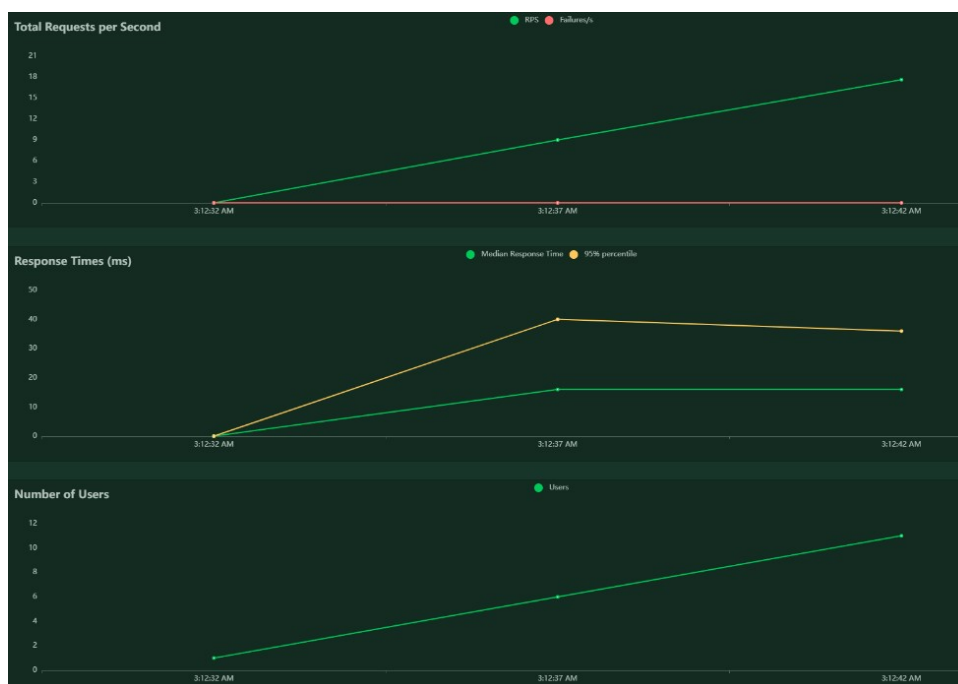


Figure 7: Example of locust Graph on testing

In test case 1 we will set number of users as 100 and spawn rate 10, test goes on for 10 seconds.

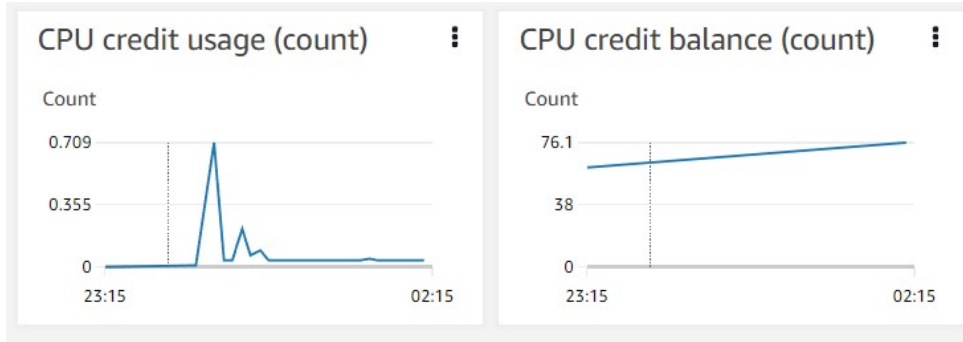


Figure 8: CPU credit usage

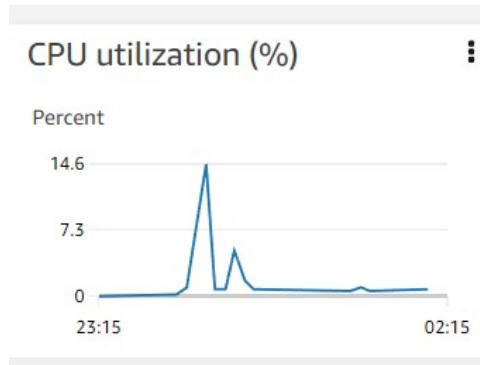


Figure 9: CPU utilization

In test case 2 we will set number of user 1000 and spawn rate 100, test goes on for 10 seconds.

In test 3 we will set number of users at 10000 and spawn rate 1000, test goes on for 10 seconds.

6 Evaluation

In this section we will review all 3 tests and its analysis.

6.1 Experiment / Case Study 1

In test 1 when the number of users were kept 1– and spawn rate at 10 for 10 seconds we observe that there is low response time for the monolithic architecture whereas microservice architecture has high response time. Monolithic delivers a smooth constant response overall, whereas microservices has spikes up and comes down although delivering a high response time. From the figure 10 and 11 we can observe the overall results of test 1. from figure 12 and 13 we can compare both the graphs for response rate of both the architectures.

6.2 Experiment / Case Study 2

In the test scenario 2 we had kept number of users at 1000 and spawn rate of 100 for 10 seconds. We can observe that the microservice architecture rises a bit and becomes stable and declines a little giving a good response time than the monolithic. We see response



Figure 10: Locust UI on test 1 on Monolithic Architecture

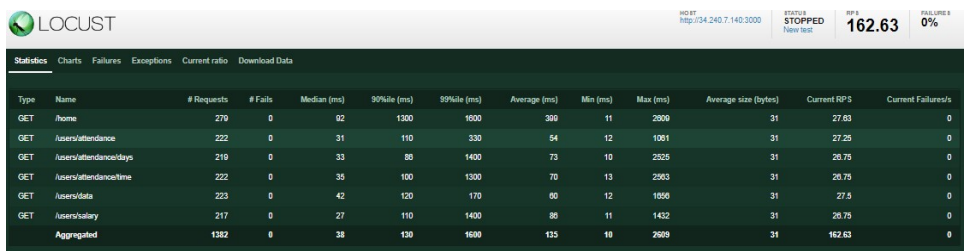


Figure 11: Locust UI on test 1 on Microservice architecture

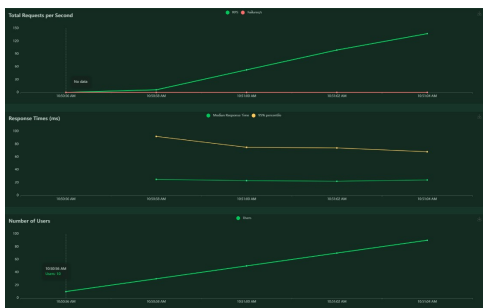


Figure 12: Graph of test 1 on Monolithic architecture

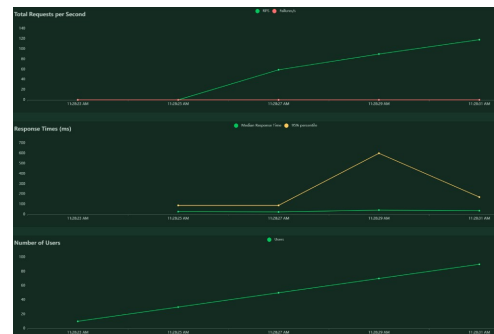


Figure 13: Graph of test 1 on Microservice architecture

time increasing towards the end in the monolithic. So by increasing a stress we observe microservice remains stable and monolithic gives a little high response rate. From the figure 14 and 15 we can observe the overall results of test 1. from figure 16 and 17 we can compare both the graphs for response rate of both the architectures.

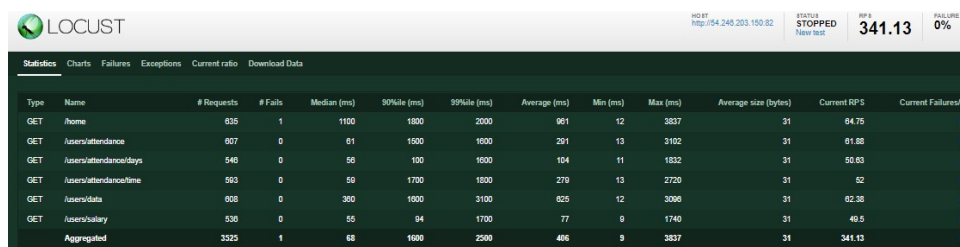


Figure 14: Locust UI on test 2 on Monolithic Architecture

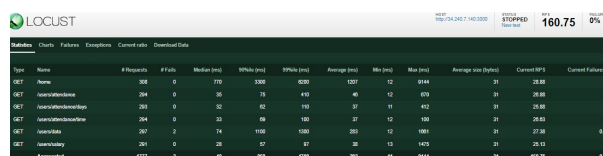


Figure 15: Locust UI on test 2 on Microservice Architecture

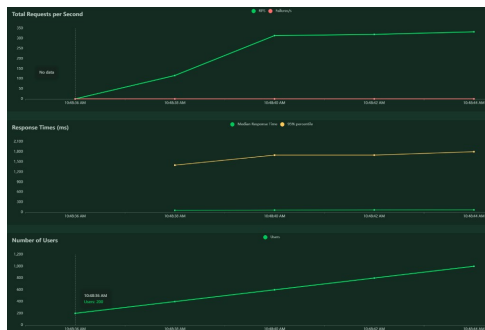


Figure 16: Graph of test 2 on Monolithic architecture

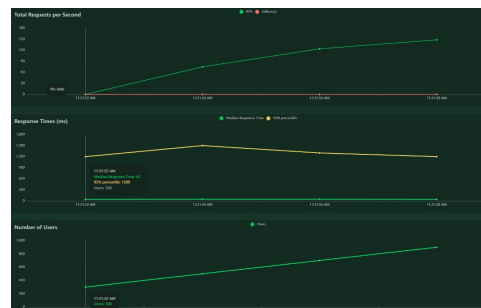


Figure 17: Graph of test 2 on Monolithic architecture

6.3 Experiment / Case Study 3

In the test case 3 we expose both the architectures to a very high stress, number of users at 10 thousand and a spawn rate of 1000 for 10 seconds. We observe that the api's in monolithic architecture apis have failed to respond due to instant burst of load. whereas we do see a rise in response time of microservice architecture but the response time keeps getting stable. Response time for the apis are monolithic are relatively high compared to those of microservice. We can say that at higher traffic on both the architectures the EKS cluster stays stable and delivers a good response time. From the figure 18 and 19 we can observe the overall results of test 1. from figure 20 and 21 we can compare both the graphs for response rate of both the architectures.

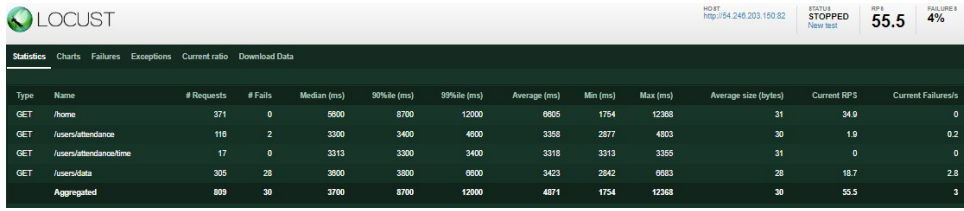


Figure 18: Locust UI on test 3 on Monolithic Architecture

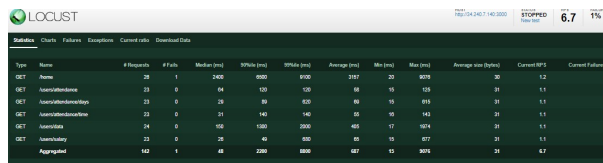


Figure 19: Locust UI on test 3 on Microservice Architecture



Figure 20: Graph of test 3 on Monolithic architecture

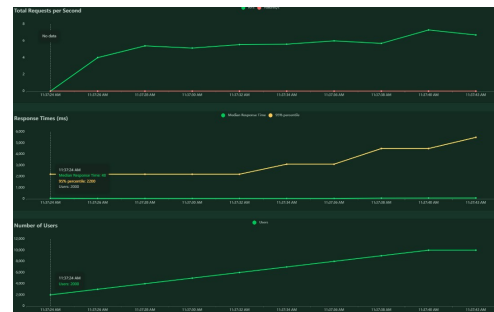


Figure 21: Graph of test 3 on Monolithic architecture

6.4 Discussion

From all the test cases that we have analyzed and after comparing with the evaluations of authors Gos and Zabierowski (2020) and Blinowski et al. (2022) who tested their application on azure cloud on simple instance and both having tested java applications, we can say that a different framework is not making any significant difference in the response time. Whereas we can say that kubernetes does help maintaining stability for high traffic load. Also cpu consumption rate can stay healthy even after burst of traffic. From figure 22 and 23 we can see that for the same tests the cost of utilizing these services is very high in the Microservice architecture on EKS. We can say that when there are less number of users for the application and the team is small they can opt for monolithic as they would be on a budget. Also if its a big company they can choose to opt for microservice on eks cluster as it provides immense stability with its auto scaling feature. Also the instance are automatically created if any of the instance fails in the cluster. The limitations to this research is the related to the application. We have deployed a very simple application but to test the real dense and heavy applications can give more reliable outputs.

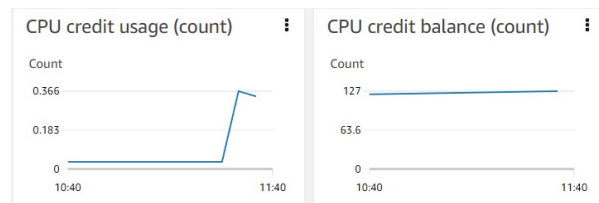


Figure 22: Microservice architecture Credit usage

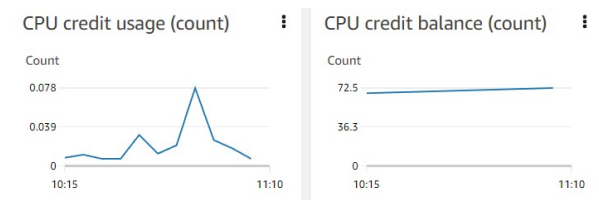


Figure 23: Monolithic architecture credit usage

7 Conclusion and Future Work

The objective of the research was to check if there will be any performance difference due to nodeJS framework and using of kubernetes. After an evaluation we can say that there is no significant enhancement or reduction in performance with the type of framework or language used for building the application. Kubernetes microservice adoption does show a huge difference in performance compare to monolithic application but we have to outweigh the budget for it before making decision of deploying there. Both the architectures have different performance to different loads hence we can conclude that a small company may choose to adopt monolithic architecture as the response time is good for low traffic and ec2 handle the cpu utilization very well keeping the costs very low compared to the other.

Analysis say the adoption of microservice architecture is going to be high in future. Hence a research can be done regarding Disaster Recovery and how much rto and rpo they provide. Disaster Recovery is a huge topic to dig in as many availability zone are also included and many regions are in sync with each other.

References

- Akbulut, A. and Perros, H. G. (2019). Performance analysis of microservice design patterns, *IEEE Internet Computing* **23**(6): 19–27. JCR Impact Factor 2019: 2.68, Cited by=36.
- Al-Debagy, O. and Martinek, P. (2018). A comparative review of microservices and monolithic architectures, *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, IEEE, pp. 000149–000154. Cited by=56.
- Astyrakakis, N., Nikoloudakis, Y., Kefaloukos, I., Skianis, C., Pallis, E. and Markakis, E. K. (2019). Cloud-native application validation stress testing through a framework for auto-cluster deployment, *2019 IEEE 24th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*, pp. 1–5.
- Bajaj, D., Bharti, U., Goel, A. and Gupta, S. (2020). Partial migration for re-architecting a cloud native monolithic application into microservices and faas, *International Conference on Information, Communication and Computing Technology*, Springer, pp. 111–124.
- Balalaie, A., Heydarnoori, A. and Jamshidi, P. (2015). Migrating to cloud-native architectures using microservices: an experience report, *European Conference on Service-Oriented and Cloud Computing*, Springer, pp. 201–215.
- Barczak, A., Barczak, P. M. and Toledo, M. (n.d.). Performance comparison of monolith and microservices based applications.
- Blinowski, G., Ojdowska, A. and Przybyłek, A. (2022). Monolithic vs. microservice architecture: A performance and scalability evaluation, *IEEE Access* **10**: 20357–20374. JCR Impact Factor : 3.476.
- Brunner, S., Blöchlinger, M., Toffetti, G., Spillner, J. and Bohnert, T. M. (2015). Experimental evaluation of the cloud-native application design, *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*, IEEE, pp. 488–493.
- Bucchiarone, A., Dragoni, N., Dustdar, S., Larsen, S. T. and Mazzara, M. (2018). From monolithic to microservices: An experience report from the banking domain, *IEEE Software* **35**(3): 50–55.
- Carroll, J. J., Anand, P. and Guo, D. (2021). Preproduction deploys: Cloud-native integration testing, *2021 IEEE Cloud Summit (Cloud Summit)*, IEEE, pp. 41–48.
- Gos, K. and Zabierowski, W. (2020). The comparison of microservice and monolithic architecture, *2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*, IEEE, pp. 150–153. Cited by=26.

- Gupta, M., Sanjana, K., Akhilesh, K. and Chowdary, M. N. (2021). Deployment of multi-tier application on cloud and continuous monitoring using kubernetes, *2021 5th International Conference on Electrical, Electronics, Communication, Computer Technologies and Optimization Techniques (ICEECCOT)*, pp. 602–607.
- Haugeland, S. G., Nguyen, P. H., Song, H. and Chauvel, F. (2021). Migrating monoliths to microservices-based customizable multi-tenant cloud-native apps, *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 170–177.
- Kalske, M., Mäkitalo, N. and Mikkonen, T. (2017). Challenges when moving from monolith to microservice architecture, *International Conference on Web Engineering*, Springer, pp. 32–47. Cited by=84.
- Mendonça, N. C., Box, C., Manolache, C. and Ryan, L. (2021). The monolith strikes back: Why istio migrated from microservices to a monolithic architecture, *IEEE Software* **38**(5): 17–22.
- Newman, S. (2021). *Building microservices*, ” O’Reilly Media, Inc.”. Cited by=2320.
- Pianini, D. and Neri, A. (2021). Breaking down monoliths with microservices and devops: an industrial experience report, *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, pp. 505–514.
- Shahid, H. (2019). *Refactoring monolithic application into cloud-native architecture*, Master’s thesis, University of Stavanger, Norway.
- Tapia, F., Mora, M. Á., Fuertes, W., Aules, H., Flores, E. and Toulkeridis, T. (2020). From monolithic systems to microservices: A comparative study of performance, *Applied sciences* **10**(17): 5797. Cited by=18.
- Villamizar, M., Garces, O., Ochoa, L., Castro, H., Salamanca, L., Verano, M., Casallas, R., Gil, S., Valencia, C., Zambrano, A. et al. (2016). Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures, *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, IEEE, pp. 179–182. Cited by=207.