ECE547/CSC547 Cloud Architecture

Project Report

Fall 2023

Indranil Banerjee, ibanerj, 200472217
Devanshi Savla, dksavla, 200472640

February 11, 2024

"We, the team members, understand that copying & pasting material from any source in our project is an allowed practice; we understand that not properly quoting the source constitutes plagiarism. All team members attest that we have properly quoted the sources in every sentence/paragraph we have copy & pasted in our report. We further attest that we did not change words to make copy & pasted material appear as our work."

Contents

1 Introduction **4**
1.1 Motivation . . . 4
1.2 Executive Summary . . . 4

2 Problem Statement **5**
2.1 The Problem . . . 5
2.2 Business Requirements . . . 5
2.3 Technical Requirements . . . 6
2.3.1 Conflicting TR's: . . . 9
2.4 Trade offs . . . 9

3 Provider Selection **10**
3.1 Criteria for choosing a provider . . . 10
3.2 Provider Comparison . . . 11
3.3 The final selection . . . 12
3.3.1 The list of services offered by the winner . . . 12

4 The first design draft **15**
4.1 The basic building blocks of the design . . . 15
4.1.1 Compute Services: . . . 15
4.1.2 Network & Security Services: . . . 15
4.1.3 Storage Services: . . . 15
4.1.4 Backup & Recovery Services: . . . 15
4.1.5 Cost Services: . . . 15
4.1.6 Monitoring Services: . . . 15
4.2 Top-level, informal validation of the design . . . 16
4.3 Action items and rough timeline (Skipped) . . . 19

5 The second design **20**
5.1 Use of the Well-Architected framework . . . 20
5.2 Discussion of pillars . . . 23
5.2.1 Security Pillar: . . . 23
5.2.2 Cost Optimization Pillar: . . . 23
5.3 Use of Cloud formation diagrams . . . 25
5.4 Validation of the design . . . 27
5.5 Design principles and best practices used . . . 30
5.5.1 Operational Excellence: . . . 30
5.5.2 Security: . . . 30
5.5.3 Reliability: . . . 30
5.5.4 Performance Efficiency: . . . 31
5.5.5 Cost Optimization: . . . 31
5.5.6 Sustainability: . . . 31
5.6 Trade offs revisited . . . 32
5.7 Discussion of an alternate design (Skipped) . . . 32

6 Kubernetes experimentation **33**
6.1 Experiment Design . . . 33
6.1.1 Description . . . 33
6.1.2 TR's worked on during this experiment . . . 33
6.1.3 Design . . . 33
6.1.4 Prerequisites . . . 34
6.1.5 Steps followed for the EKS cluster setup and image deployment . . . 34
6.2 Workload generation with Locust . . . 39
6.2.1 What is locust? . . . 39
6.3 Analysis of the results . . . 40

7 Ansible playbooks (Skipped) 44
7.1 Description of management tasks (Skipped) 44
7.2 Playbook Design (Skipped) 44
7.3 Experiment runs (Skipped) 44

8 Demonstration (Skipped) 45

9 Comparisons (Skipped) 46

10 Conclusion 47
10.1 The lessons learned 47
10.2 Possible continuation of the project (Skipped) 47

1 Introduction

1.1 Motivation

It is essential to understand the different aspects of deploying an application on the cloud. We aim to learn about comparing different cloud providers based on their services, pricing, and other factors to choose the best provider for our needs. We would also learn more about different cloud architecture practices and design patterns generally taken into consideration while deploying any application, in our case an application with sensitive data such as healthcare.

1.2 Executive Summary

The primary objective of this project is to implement a comprehensive healthcare application by leveraging cloud technology. This initiative focuses on addressing key challenges within the healthcare sector, primarily centered around the efficient management of patient medical records across a network of hospitals. The application aims to streamline the healthcare data ecosystem, providing a centralized platform for the storage, retrieval, and transfer of crucial medical information. The project will focus on the following key areas:

1. The core focus of this project is the deployment of a robust healthcare application tailored for use in a chain of hospitals. The cloud-based infrastructure will serve as the foundation for ensuring scalability, accessibility, and security in managing patient data.

2. The healthcare application will encompass the comprehensive medical history of each patient. Additionally, it will facilitate doctors in writing and managing prescriptions seamlessly.

3. An overarching goal of the project is to enhance the interoperability of patient records. The application is designed to facilitate the effortless transfer of medical records between different doctors and hospitals.

4. Patients will have the ability to transfer their medical records securely and efficiently to different healthcare providers, promoting continuity of care and reducing administrative hurdles.

In summary, the deployment of this healthcare application on the cloud seeks to revolutionize the way patient data is managed and shared across healthcare institutions. The project's emphasis on interoperability, comprehensive medical records, and improved patient experience aligns with the broader goals of advancing healthcare services through innovative and secure technology solutions.

2 Problem Statement

2.1 The Problem

We are aiming to develop and deploy a secure and scalable healthcare application on cloud architecture that allows patients to share their medical history with different doctors across a chain of hospitals, and for doctors to write prescriptions on the application.

The key challenges to address are:

1 ***Security:*** The application must protect the confidentiality and integrity of patient medical records. This can be achieved by using a variety of cloud security features [21], such as encryption, IAM roles and policies, and VPCs.

2 ***Scalability:*** The application must be able to handle a large number of concurrent users and transactions. This can be achieved by using various cloud scalable services, such as Amazon's Elastic Compute Cloud (EC2), Elastic Container Service (ECS), DynamoDB etc. [22]

3 ***Integration:*** The application must be able to integrate with existing hospital systems, such as electronic health records (EHR) systems and prescription fulfillment systems. This can be achieved using cloud integration services, such as Simple Notification Service (SNS) and Simple Queue Service (SQS).

2.2 Business Requirements

Our business requirements for deploying an e-Medicare portal on the cloud are:

BR-1 **Scalability and Elasticity:** The application should be able to vertically and horizontally scale up or scale down according to the load or traffic.[33] For example, a flu season might experience emergency spikes in application usage or a scenario similar to Covid-19.

BR-2 **High Availability:** The application should be available to the user 24x7. The application should be highly fault tolerant minimizing downtime. [34]

BR-3 **Security and Compliance:** Since the application would be dealing with sensitive data of patients as well as doctors, it should be highly secure. Also, this includes protecting patient data and meeting HIPAA compliance requirements. [35]

BR-4 **Interoperability and No Vendor Lock-in:** The healthcare application should be able to interoperate with other healthcare systems. This is important for enabling seamless care coordination and data sharing between different healthcare providers. The application should not be limited to just one cloud provider, it should be extendable to other cloud providers as well. Interoperability can be a requirement from the perspective of a cloud architect and a software architect but we are considering it from the point of view of a cloud architect. [36]

BR-5 **Cost Effectiveness:** Resource and service monitoring tools should be employed to understand the usage requirements and regulate the provisioning of such services, eventually minimizing the cloud operations cost. [37]

BR-6 **Disaster Recovery and Backup:** Since the application will have sensitive patient records and data, disaster recovery and backup is an important aspect. Deploying the application across multiple regions and creating disaster recovery sites are essential.[38]

BR-7 **Identity and Access Control Management:** To guarantee that only individuals with the proper authorization can access patient data and healthcare systems, put in place strong access control and identity management systems.[39]

BR-8 **Audit Logging and Monitoring:** The application should be monitored and system-level logs should be saved. This could be used for future debugging and alert generation.[60]

BR-9 **Tenant Identification:** The system must have a robust tenant identification mechanism to ensure that data and resources are securely segregated and accessible only to authorized tenants. The user base may include different hospital organizations, doctors, patients, etc.

BR-10 **Performance Efficiency:** The system must demonstrate efficient performance to ensure responsive and reliable access to resources and data, regardless of the scale and complexity of usage.

BR-11 **Reliability:** The system architecture must be highly fault-tolerant, ensuring a highly reliable application uptime. This also depends on the cloud provider's SLA.[40]

2.3 Technical Requirements

Business Requirement	***Technical Requirement***	***Justification***
BR-1 Scalability & Elasticity	**TR-1.1** Auto-Scaling Mechanism	Implement an auto-scaling mechanism to dynamically adjust resources based on application load, ensuring the system can handle spikes in usage during flu seasons or similar events.[41]
	TR-1.2 Application Performance Monitoring	Implement robust application performance monitoring tools such as Dynatrace [61], a combination of Prometheus and Grafana to continuously assess the system's performance. Monitoring would help in identifying bottlenecks, resource constraints, and performance issues in real time, enabling proactive adjustments for optimal application performance and also aiding in auto-scaling.
	TR-1.3 Distributed and Scalable Architecture	Implement a distributed and scalable architecture by accommodating fluctuating workloads, and handling increasing user demands by introducing load balancing in the architecture. This approach will enable the system to dynamically allocate resources and distribute the load accordingly.
BR-2 High Availability	**TR-2.1** Multiple Availability zones	Implement multiple availability zones in a region to handle any data center failure and make the system always available. The application deployed in the secondary region will act as a backup in case of a complete primary region failure.
	TR-2.2 Multi-region deployment architecture	Deploy the application across various global regions to handle a complete region failure and at the same time implement complete data replication across the regions in real-time, subsequently increasing architecture flexibility. [62]
	TR-2.3 Monitoring and observability	Continuous monitoring of the availability zones and taking required actions on the basis of certain trigger events. Also, generate push notifications on every triggered event.
BR-3 Security and Compliance	**TR-3.1** Data Encryption and Protection	Implement robust encryption protocols both during data transmission and ensure the security of the sensitive patient information being stored in databases. Use a secure vault technology for storage and encryption of sensitive data.
	TR-3.2 HIPAA Compliance	Ensure compliance with the Health Insurance Portability and Accountability Act (HIPAA)[63] by implementing specific safeguards related to the protection of health information.

Business Requirement	***Technical Requirement***	***Justification***
	TR-3.3 Web Application Firewall & Throttling for APIs	A WAF [64]filters and inspects API traffic, blocking malicious requests and providing an additional layer of security for API endpoints. Implement API throttling mechanisms to limit the rate of incoming requests from a single source. Throttling prevents abusive usage, Distributed Denial of Service (DDoS) attacks, and ensures that APIs are available for legitimate users while maintaining system stability.
BR-4 Interoperability & No Vendor Lock-in	**TR-4.1** Automate the deployment	Automate the deployment process independent of the cloud provider by implementing IaaC like Terraform or Ansible. This will enable consistent and repeatable deployments across different cloud providers and facilitate the system's adaptability to multi-cloud environments.
	TR-4.2 Multi-Cloud Strategy	Distributing the application workload across multiple cloud providers will ensure resilient and robust architecture. This will also mitigate risks associated with a single point of failure, and prevent vendor lock-in by diversifying the system's reliance on cloud providers.
	TR-4.3 Use of standard APIs	Implement the application using standard REST APIs for easier migration to different cloud providers. This will improve the ability of different systems to work together by making sure that the integration points are compatible with many different platforms, reducing the challenges of integration and making it easier to create solutions that work with different vendors. Also, it will help in handling stateless requests.
BR-5 Cost Effectiveness	**TR-5.1** Cost Optimization Tools	Deploy monitoring tools for resources and services to streamline cloud resource allocation, minimize operational expenses, and optimize cost-efficiency. These tools will aim to identify and eliminate underutilized resources, right-size instances, and redundant resources.
	TR-5.2 Resource budgeting	Setting up and managing resources as per the allocated budget and keeping track of the predefined financial plan. This proactive approach helps control costs, prevent over-provisioning, and align resource provisioning with anticipated demand.
BR-6 Disaster Recovery and Backup	**TR-6.1** Regular Data Backup	Establish regular data backup practices to safeguard and maintain up-to-date duplicates of essential information. Frequent backups guarantee that in the event of data loss or corruption, data can be restored to a recent point in time, thereby minimizing data loss and system downtime.
	TR-6.2 Data Recovery	In case of a failure, the data should be recovered to the last good state that was backed up. Establish clear and detailed data recovery procedures and protocols to ensure systematic restoration of data and services in the event of data loss or system failures.
	TR-6.3 Failover mechanism	Create a dedicated disaster recovery site [65] to handle a complete site failure. Implement a failover mechanism that provides seamless continuity of services in case of system failures. This mechanism automatically redirects traffic and operations to redundant systems or locations, reducing downtime and ensuring service availability.

Business Requirement	***Technical Requirement***	***Justification***
BR-7 Identity Access and control management	**TR-7.1** Implement RBAC	Implementing Role Based Access Control system for the application to ensure that only authorized individuals can access sensitive information.
	TR-7.2 Secure API Gateways	Enhance security by employing secure API gateways to handle access control and authentication for external systems and users, ensuring a robust level of protection.
BR-8 Audit Logging and Monitoring	**TR-8.1** Observability pipeline	Create a continuous observability pipeline to gather real-time system logs, enabling future debugging, compliance reporting, and visualize them over a centralized dashboard.
	TR-8.2 Comprehensive Monitoring and Alerts	Create a monitoring system that sends regular alerts on certain thresholds and triggers. The alerting system will continuously track system performance, security, and operational metrics.
	TR-8.3 Generate actionable insights	Automate responses from the visualization to troubleshoot or make operational changes. The visualization pipeline will generate key performance metrics that lead to the determination of the application's SLA.
BR-9 Tenant Identification	**TR-9.1** Tenant Identification & Isolation	By identifying and isolating the resources and data of different tenants, the system should ensure that one tenant's data cannot be accessed by another. The isolation will also ascertain that none of the tenant's data will interfere with the other, ensuring a more reliable solution.
BR-10 Performance Efficiency	**TR-10.1** Container Orchestration	Implement a container orchestration platform, such as Kubernetes, to efficiently manage and deploy application containers. Container orchestration enhances performance efficiency by automating tasks like container scaling, load balancing, and resource allocation, ensuring seamless application deployment and scaling as needed.
	TR-10.2 Caching Mechanism	Integrate a robust caching mechanism like Redis to store frequently accessed data, reducing the load on the backend and enhancing system performance. This will minimize subsequent database queries and network requests, resulting in enhanced performance metrics.
	TR 10.3 Minimal response time approx. 800ms	The web application shall achieve a maximum response time of 800 mili seconds for 95% of user interactions under normal operating conditions. User interactions include but are not limited to, page loading, form submissions, data encryption, decryption, and data retrieval.
BR-11 Reliability	**TR-11.1** Primary-Secondary Failover Mechanism	Deploy a redundant secondary application architecture that mirrors the primary. In case of a failure in the primary architecture, the system should redirect requests to the secondary with minimal downtime as per the documented SLA.
	TR-11.2 Routine Data storage & Backup	Since the application will have highly sensitive data, it is important to regularly back up the existing data in case of any storage failure. This will be a highly reliable solution in case of an outage.

Table 1: Mapping of TR's to their respective BR's

2.3.1 Conflicting TR's:

1. **High Availability vs Cost [TR 2.2 and TR 5.1]:**
   TR 2.2 explains the need for Multi-region deployment architecture. To ensure that the application will be highly available, it needs to have a backup architecture or secondary architecture ready in case of any complete region failure or data failure. In this case, we would have to build a redundant architecture which will in turn increase the cost that is explained in TR 5.1 Cost Optimization. This conflict would be resolved by choosing High Availability as the application is for the healthcare domain and it has to be available almost 24x7.

2. **Security vs Performance Efficiency [TR 3.1 and 10.3]:**
   TR 3.1 explains the Data Encryption and protection scheme for sensitive data or patients' health records. Encrypting sensitive information is essential. This requirement mandates the implementation of encryption at both HTTP and REST levels. By enforcing HTTPS for all client-server interactions, data in transit remains confidential and shielded from unauthorized access. Simultaneously, RESTful API communications employ industry-standard encryption algorithms, fortifying data integrity and thwarting potential breaches. But this approach will have a greater response time in some cases where we have a huge client load, which will degrade the performance time or response time already stated in TR 10.3.

2.4 Trade offs

1 High Availability vs Cost: [Selected High Availability](BR-2 and BR-5)
For a system to be highly available, it has to be deployed over multiple availability zones to overcome any complete zonal failure. Subsequently, to achieve high availability, the architecture needs to be fault-tolerant. To make a highly fault-tolerant architecture, the solution needs to be deployed over multiple geographic regions to overcome any complete regional failure. To achieve all these aspects, we need multiple servers, data centers, or cloud regions, for which we need to maintain redundant architecture. This redundant architecture needs will increase the cost of maintaining the complete architecture.
So, here high availability and cost have a trade-off. If we want a highly available solution, we have to pay more and vice versa. But for a solution that will be used by a lot of patients and will be used in emergency situations, high availability is the obvious choice.

2 Security vs Performance Efficiency: [Selected Security] (BR-3 and BR-10)
Implementing comprehensive security measures can introduce performance overhead, such as encryption/decryption, authentication checks, and firewall processing. One example is the encryption and decryption at HTTP and REST levels, which are responsible for the security and transmission of sensitive patient data. Although, Data encryption provides higher security to an application, the performance overhead increases, which makes it a trade-off to choose between security and performance efficiency. Security takes precedence over any other aspect, so we will be choosing security in this case over performance overhead and try to better the performance.

3 Provider Selection

3.1 Criteria for choosing a provider

1. **Availability**
   Availability as a criterion for choosing a provider is important because of two main reasons:
   - Having an option of multiple availability zones spread across a region and in turn, multiple regions will provide us with an exhaustive list of options to select from while deploying the application.[66]
   - Positioning data zones in close proximity to clients is beneficial because that not only optimizes time and cost but also significantly reduces the risk of data loss. By locating data zones near clients, the number of hops required for data transmission between the client and the provider is minimized.[66]
2. **Scalability and Flexibility**
   - It's vital to be able to adjust the amount of resources being used in the architecture according to the needs. Choosing a cloud provider that gives the service to easily scale up or down, letting the applications expand or contract smoothly is essential.
   - Features such as auto-scaling, and container orchestration technologies allow resources to adapt in real-time to changes in the amount of work, making sure the system performs well and stays cost-effective.[67]
3. **Security and Compliance**
   - Security is of prime importance, especially for sensitive data such as healthcare information. Choosing a cloud provider with robust security measures/services, including data encryption, identity and access management, and compliance certifications (e.g., HIPAA for healthcare data) is essential. [63]
   - Also, ensuring that the provider aligns with industry regulations and standards relevant to the application is vital.
4. **Data Management and Storage**
   - We need to evaluate the providers on the basis of the database services. How many client database services would they be offering would also be a point to be factored in while choosing a provider.
   - Cloud storage providers offer a variety of storage tiers, such as standard, archival, and high-performance storage, to accommodate the diverse access patterns and requirements of different types of data. The cloud provider with the most cost-effective and suitable storage options would be chosen.[69]
   - The speed and responsiveness of data retrieval will impact the performance of the application. This will be another factor in choosing a cloud provider that provides high-performance storage solutions and ensures quick and efficient access to stored information.[69]
5. **Service Offerings**
   - An essential part of selecting a cloud provider is to assess the variety of solutions/offerings it provides to build its architecture.
   - A provider with a wide range of services would give us multiple options to satisfy industry-level technical and functional requirements.
6. **Cost and Pricing Models**
   - To ensure high availability of the application, it requires redundant architecture which in turn increases the cost of building & maintaining the architecture, so choosing a cloud provider that has better pricing models will help us manage the allocated budget.[68]
   - A cloud provider with more reasonable and effective data storage and management pricing would be preferred for the architecture.

3.2 Provider Comparison

Sr. No.	***Criteria/Providers***	***Azure***	***GCP***	***AWS***	***Winner***
1	High Availability	Azure has **60+ regions** globally. [43]Each region consists of multiple availability zones to ensure high availability and fault tolerance.	GCP has **39 regions and 108 availability zones** globally. [44]	AWS has **32 regions and 102 availability zones** around the world. [45]	**Azure**
2	Scalability and Flexibility	• Azure provides only **L4 load balancing** at the network layer. [48] • Azure also provides the auto-scaling feature but does not provide target tracking. [56]	• GCP provides **L4 and L7 load balancing**. GCP does not provide spot instances. [47] • GCP provides auto-scaling but does not allow a mixture of spot instances and on-demand VM's in the auto-scaling cluster. [56]	• AWS provides elastic **L4, L7, and gateway load balancing** which makes it capable of scaling requests at the application layer also. [49] • AWS allows spot instances to be used in a mixture with EC2's in auto-scaling clusters. AWS also allows target tracking during auto-scaling. [56]	**AWS**
3	Security and Compliance	All three cloud providers offer similar security services like IAM, SSO, RBAC, etc. [57]	All three cloud providers offer similar security services like IAM, SSO, RBAC, etc. [57]	All three cloud providers offer similar security services like IAM, SSO, RBAC, etc. AWS additionally provides multifactor authentication and is more robust and serverless. [57]	**AWS**

Sr. No.	***Criteria/Providers***	***Azure***	***GCP***	***AWS***	***Winner***
4	Data Management and Storage	All three cloud providers offer similar data storage options with appropriate storage classes and archival processes. • **$0.021 GB/month in Azure storage.**[42]	All three cloud providers offer similar data storage options with appropriate storage classes and archival processes. • **$0.023 GB/month in GCP cloud storage.**[42]	All three cloud providers offer similar data storage options with appropriate storage classes and archival processes. • **$0.023 GB/month in Amazon S3.**[42]	**Azure**
5	Service Offerings	Over 200 services [59]	Over 100 services [58]	Over 200 services [58]	**AWS**
6	Cost and Pricing Models (This is considering compute instances)	Azure prices a per-minute model for VM's.[42]	GCP provides a per-second pricing model with a minimum of one minute.[42]	AWS also provides a per-second pricing model with a minimum of 60 seconds.[42]	**AWS**

Table 2: Provider Comparison table

3.3 The final selection

Taking into consideration the above factors, we choose **AWS as the winner** cloud provider because the majority of criteria like scalability, security, service offerings, cost, and pricing model are satisfied by it (4/6). Below are the services provided by AWS for the criteria.

Sr. No.	***Criteria***	***Services***
1	High Availability	AWS EC2, AWS Elastic Diaster Recovery
2	Scalability and Flexibility	AWS ELB, AWS EKS
3	Security and Compliance	AWS WAF, AWS IAM, AWS KMS
4	Data Management and Storage	AWS S3, AWS Elastic Cache
5	Cost and Pricing Models	AWS Cost Calculator, AWS Cost and Usage Reporting

Table 3: Services selected based on criteria

3.3.1 The list of services offered by the winner

1. **Amazon EC2** [1]
   - Amazon Elastic Compute Cloud is a web service that allows users to create secure, resizable compute instances of the cloud.
   - It is used to allocate and access application infrastructure and architecture. It is scalable and reliable.
2. **AWS Elastic Load Balancer** [2]
   - The AWS elastic load balancer (ELB) helps create a distributed architecture and divide the network load according to the traffic for an application.
   - It is able to scale up and down according to the network traffic. It helps with keeping high availability and application scalability.
   - The AWS ELB provides three types of load balancers: Application load balancer (L7 layer), Network load balancer (L4 layer) and Gateway load balancer.
3. **AWS Auto-scaling** [3]
   - AWS Auto Scaling intelligently manages your applications' resource utilization, ensuring consistent and predictable performance while minimizing costs.

- AWS Auto Scaling simplifies scaling decisions by providing tailored recommendations that enable you to optimize performance, costs, or a balance between them.

4. **Amazon Managed Service for Prometheus** [4]
   - Amazon Managed Service for Prometheus seamlessly integrates with Prometheus to monitor and alert on containerized applications and infrastructure at scale.
   - Prometheus is a popular open-source monitoring and alerting solution optimized for container environments.
5. **AWS Managed Grafana** [5]
   - AWS Grafana is a data visualization service that helps you monitor and visualize a dashboard view for the operational metrics, system logs, and traces.
   - AWS Grafana simplifies workspace management by automatically provisioning, configuring, scaling, and maintaining your workspaces.
6. **AWS Web Application Firewall** [6]
   - AWS Web Application Firewall is a web application firewall service that helps protect web applications from common web exploits.
   - It integrates seamlessly with other AWS services, providing a layer of security for applications hosted on Amazon CloudFront, Application Load Balancers, and API Gateway.
7. **AWS CloudTrail** [7]
   - It provides a comprehensive history of changes to resources, helping with security analysis, resource tracking, and compliance auditing.
   - CloudTrail enables you to monitor and respond to AWS resource changes, user activity, and potential security threats.
8. **AWS Key Management Service** [8]
   - AWS Key Management Service (KMS) is a managed service that makes it easy to create and control cryptographic keys, allowing to encrypt data at rest and in transit.
   - AWS KMS simplifies key management tasks, including key rotation, and provides a secure foundation for your data protection strategy.
9. **AWS Cost Calculator** [9]
   - It is a tool that allows users to estimate the cost of using AWS services.
   - The Cost Calculator provides insights into resource costs, allowing users to optimize and budget for their AWS usage.
10. **AWS Cost and Usage Reporting** [10]
    - It provides comprehensive reports with hourly or daily granularity, facilitating cost management and optimization.
    - By analyzing cost and usage reports, users can identify trends, allocate expenses, and make informed decisions to optimize their AWS spending.
11. **AWS Backup** [11]
    - AWS Backup simplifies the backup process, making it easy to configure and manage backups across multiple AWS resources.
    - It centralizes and automates the backup of your data across AWS services.
12. **AWS S3** [12]
    - Amazon Simple Storage Service (S3) is a scalable object storage service designed to store and retrieve any amount of data.

- S3 is widely used for data storage and backup, and as a foundation for building scalable and secure applications.

13. **AWS Elastic Disaster Recovery** [13]
    - It leverages AWS services to ensure data resilience and availability in the event of a disaster or disruption.
    - It helps in creating redundant architecture for any case of data sync failure, there is a secondary architecture ready to replace the primary, and the incoming traffic is redirected to it.
14. **AWS Identity and Access Management** [14]
    - It allows you to manage users, groups, and permissions, defining who can access specific resources and what actions they can perform.
    - It also helps in tenant identification and manages the access control according to the same.
15. **AWS API Gateway** [15]
    - It enables secure and efficient communication between applications by handling tasks such as authorization, access control, and rate limiting.
    - API Gateway facilitates the development of robust and scalable API architectures.
16. **AWS CloudWatch** [16]
    - AWS CloudWatch is a monitoring and observability service that provides data and actionable insights for your applications and infrastructure.
    - It collects and tracks metrics, monitors log files, sets alarms, and reacts to changes in AWS resources.
17. **AWS Elastic Kubernetes Service** [17]
    - AWS Elastic Kubernetes Service (EKS) is a fully managed Kubernetes service for containerized applications. It simplifies the deployment, management, and scaling of containerized applications using Kubernetes.
18. **AWS Elastic Cache** [18]
    - AWS Elastic Cache is a fully managed, in-memory caching service compatible with popular caching engines. It helps improve the performance of applications by reducing latency and enhancing data retrieval.

4 The first design draft

4.1 The basic building blocks of the design

The services provided by AWS will be the building blocks of the design. There are some essential services for this application deployment which are mentioned below:

4.1.1 Compute Services:

1. **Amazon EC2**: Will be used to create secure, resizable compute instances of cloud.[1]
2. **AWS Elastic Kubernetes Service**: It is used to autonomously handle the availability and scalability of Kubernetes control plane nodes, which are responsible for tasks such as scheduling containers, ensuring application availability, storing cluster data, and other critical functions. [17]
3. **AWS Auto-scaling**: AWS Auto Scaling automatically adjusts the number of compute resources in response to changes in demand, ensuring optimal performance and cost efficiency for applications.[3]

4.1.2 Network & Security Services:

1. **AWS Web Application Firewall**: Used to protect web applications from common web exploits and malicious attacks by allowing users to create rules that filter and monitor incoming web traffic. Since security is a major concern in healthcare applications.[6]
2. **AWS Elastic Load Balancer**: It is used to distribute or redirect the traffic of the network to different compute instances, and scale up and down resources as needed.[2]
3. **AWS Key Management Service**: It is utilized to securely create and control encryption keys, allowing users to easily encrypt and decrypt data within various AWS services and create rules based on IP address. [8]
4. **AWS Identity and Access Management**: It is used for managing access rights according to the tier and control.[14]

4.1.3 Storage Services:

1. **AWS S3**: S3 is used for scalable and secure cloud storage and retrieving any amount of data at any time.[12]

4.1.4 Backup & Recovery Services:

1. **AWS Elastic Disaster Recovery**: Used to ensure business continuity by providing a scalable and automated solution for the backup, replication, and recovery of critical applications and data in the event of a disaster or unexpected outage.[13]

4.1.5 Cost Services:

1. **AWS Cost Calculator**: The AWS Cost Calculator provides users with estimates of their monthly AWS costs based on anticipated usage and configurations, helping businesses plan and manage their budget effectively.[9]
2. **AWS Cost and Usage Reporting**: AWS Cost and Usage Reporting delivers detailed insights into an organization's AWS usage patterns and associated costs, facilitating informed decision-making and cost optimization strategies.[10]

4.1.6 Monitoring Services:

1. **AWS CloudWatch**: It will be used to monitor and verify the system logs, operational metrics, and performance of the application.[16]

4.2 Top-level, informal validation of the design

Sr. No.	***AWS Service & Description***	***Mapping TRs***	***Justification***
1	**AWS EC2** (Amazon Elastic Compute Cloud): provides secure, resizable compute capacity in the cloud. [1]	Scalability & Elasticity **TR 1.1**	AWS EC2, the Elastic Compute Cloud, stands out as an optimal solution for achieving scalability and elasticity, primarily attributable to its Auto Scaling functionality. It enables the dynamic adjustment of EC2 instances in response to demand, ensuring an automatic and seamless scaling up or down to handle workload fluctuations effectively.
2	**AWS ELB** (Amazon Elastic Load Balancer): distributes incoming traffic across multiple targets, such as EC2 instances, containers, and IP addresses.[2]	Scalability & Elasticity **TR 1.3**, Reliability **TR 11.1**	ELB's dynamic scaling capabilities adapt to changes in demand, automatically adjusting the load distribution, and enabling seamless scalability to meet the needs of a distributed system. ELB's failover mechanism enhances reliability by automatically rerouting traffic to healthy compute instances.
3	**AWS Auto-scaling**: scales Amazon EC2 resources up or down based on predefined metrics, helping maintain the desired performance and cost-efficiency. [3]	Scalability & Elasticity **TR 1.1**, Availability **TR 2.1**	The auto-scaling feature for EC2 will help maintain the optimum number of compute instances depending on the varying workload and seamlessly spinning up and down instances with the dynamic workload ensuring scalability and high availability.
4	**Amazon Managed Service for Prometheus**: provides a serverless, Prometheus-compatible monitoring service for container metrics that enables secure monitoring of container environments at scale. [4]	Scalability & Elasticity **TR 1.2**, Audit Logging & Monitoring **TR 8.1**, **TR 8.3**	It facilitates the development of a robust observation pipeline. By collecting, processing, and visualizing metrics, it allows for quick identification of performance anomalies or issues. It can also be used for determining the scaling needs for the compute or storage resources.
5	**AWS Grafana**: provides data visualization service that helps you monitor and visualize a dashboard view for the operational metrics, system logs, and traces. [5]	Audit Logging & Monitoring **TR 8.2**	On the basis of the Prometheus service provided by AWS, a centralized Grafana dashboard will help us view and manage the comprehensive observability pipeline and create alerts and actionable insights from the same.
6	**AWS Web Application Firewall**: protects web applications from common web exploits and malicious attacks.[6]	Security & Compliance **TR 3.3**	By inspecting web traffic, WAF will help us in protecting the web application from various online threats, including SQL injection, cross-site scripting (XSS), and other common exploits. AWS WAF enables throttling of web traffic, allowing you to control the rate of incoming requests. WAF will also help us protect against DDOS attacks.

Sr. No.	***AWS Service & Description***	***Mapping TRs***	***Justification***
7	**AWS CloudTrail**: provides a comprehensive log of AWS API calls and related events, allowing users to monitor and audit activities, troubleshoot operational issues, and enhance security and compliance. [7]	Security & Compliance **TR 3.2**	AWS CloudTrail is essential for achieving and maintaining HIPAA compliance which is important in healthcare applications due to its robust capabilities in providing detailed auditing and monitoring of AWS resources. It records all API calls and related events, offering transparency into user activity, resource changes, and system events.
8	**AWS Key Management Service**: a service to securely create and manage encryption keys.[8]	Security & Compliance **TR 3.1**	AWS KMS will help us in creating a centralized and secure key management system. KMS offers encryption of data at rest and in transit, ensuring a standardized approach to data protection across all AWS resources.
9	**AWS Cost Calculator**: This service helps calculate and get an estimate of the AWS resources being used in the architecture helping in resource allocation and budgeting.[9]	Cost Effectiveness **TR 5.2**	The cost calculator will help us to optimize resource usage, prevent cost overruns, and make informed decisions to achieve a balance between performance and cost efficiency in their AWS deployments.
10	**AWS Cost & Usage Reporting**: It helps create a comprehensive set of cost and usage data available and generate billing reports. [10]	Cost Effectiveness **TR 5.1**	AWS Cost and Usage Reporting tool will provide insights into resource usage at a granular level with optimization possibility. It will also enable in creation of alerts as per the budget threshold and prevent unexpected overruns. It will help us identify underutilized resources and get customized reports.
11	**AWS Backup**: It is a solution for disaster recovery and it provides centralized management for data backup across all AWS services. [11]	Disaster Recovery & Backup **TR 6.1**, **TR 6.2**	In the event of a disaster or data loss, AWS Backup facilitates efficient and granular recovery, allowing users to restore specific files, volumes, or entire resources. AWS backup could be integrated with storage services like S3 providing a secure and resilient repository for backup data.
12	**AWS S3**: It is an object storage service offering high scalability, data availability, security, and performance.[12]	Disaster Recovery & Backup **TR 6.2**, Reliability **TR 11.2**	AWS S3 provides 99.999999999% (11 9's) durability, making it highly reliable for routine data storage and backup. We would use S3 to store volumes of sensitive data backup which could be used for data recovery without compromising performance.
13	**AWS Elastic Disaster Recovery**: This service replicates on-premises or cloud-based workloads to AWS, enabling fast and reliable recovery from outages and disasters.[13]	High Availability **TR 2.2**, Disaster Recovery & Backup **TR 6.1**, Reliability **TR 11.1**	It offers automated recovery points at regular intervals and data backup which helps restore their systems to a predefined state, maintaining data resiliency, minimizing data loss and downtime. It also ensures high availability by seamless transition between primary and secondary environments.

Sr. No.	***AWS Service & Description***	***Mapping TRs***	***Justification***
14	**AWS Identity and Access Management**: It helps create and manage users, groups, and roles, and we can grant them permissions to AWS resources. It helps in securing managed access to AWS resources. [14]	Identity Access & control management **TR 7.1**, Tenant Identification **TR 9.1**	By implementing RBAC, IAM ensures that users, roles, and groups have precisely the permissions they need and nothing more, reducing the risk of unauthorized access or actions. It also provides the ability to create isolated environments for different tenants, ensuring that each tenant's resources are segregated and secure. IAM allows tailored policies for different tenants which makes the maintenance of multiple tenants much easier and scalable.
15	**AWS API Gateway**: It is a fully managed service that makes it easy for developers to create, publish, maintain, monitor, and secure APIs at any scale. It acts as a front door for applications to access data, business logic, or functionality from your backend services. [15]	Identity Access & control management **TR 7.2**, Interoperability & No vendor Lock-in **4.3**	API Gateway supports HTTPS, encrypting data in transit and ensuring secure communication between clients and APIs, which is crucial for sensitive data protection. It also allows to control the access to the APIs with robust authentication and authorization mechanisms.
16	**AWS CloudWatch**: It is a monitoring and observable service that provides comprehensive data and actionable insights to monitor and manage AWS, on-premises, hybrid, and other cloud applications and infrastructure resources.[16]	High Availability **TR 2.3**, Audit Logging & Monitoring **TR 8.1**, **TR 8.2**, **TR 8.3**	AWS CloudWatch provides an observability pipeline by consolidating logs, metrics, and traces in a unified platform. It provides monitoring capabilities like tracking performance metrics, setting alert thresholds, etc. It also facilitates the creation of actionable insights based on the visualization trends and any anomalies above predefined thresholds.
17	**AWS Elastic Kubernetes Service**: It is a fully managed Kubernetes service that makes it easy to deploy, manage, and scale containerized applications on AWS.[17]	Performance Efficiency **TR 10.1**	It eliminates the need to install, operate, and maintain your own Kubernetes control plane, and it integrates with a wide range of AWS services, such as Amazon Elastic Container Registry (Amazon ECR) to provide a seamless experience for deploying and managing containerized applications on AWS.
18	**AWS Elastic Cache**: It is a fully managed, Redis- and Memcached-compatible service delivering real-time, cost-optimized performance for applications.[18]	Performance Efficiency **TR 10.2**	AWS Elastic Cache provides the application, the ability for in-memory caching, enabling faster access to frequently requested data. This improves the response time of the application and in turn the performance overall.

Table 4: Mapping of the services provided by AWS to the respective technical requirements

4.3 Action items and rough timeline (Skipped)

5 The second design

5.1 Use of the Well-Architected framework

The Well-Architected Framework, developed by AWS, provides a set of best practices and guidelines for designing and maintaining secure, high-performing, resilient, and efficient infrastructure or architecture.[46] The above framework is organized into six pillars, each representing an important key aspect of a well-designed architecture. So now for each pillar, there are specific steps or considerations suggested to guide the design process.[46] Below mentioned are some of the specific steps suggested by the Well-Architected Framework:

1. ***Operational Excellence:***

   - **Organization**: Establishing clear and well-defined processes is crucial for ensuring operational efficiency and consistency. This involves identifying and documenting the steps involved in each core business function, such as order fulfillment, customer service, and product development.[50]
   - **Preparation:** Operational readiness reviews are proactive measures that help organizations prepare for potential disruptions or events that could impact their ability to deliver products or services. [50]
   - **Procedure or Operate:** Perform operations using consistent procedures and playbooks. Procedures provide a step-by-step guide for performing tasks, while playbooks outline the specific actions to be taken in response to specific scenarios.[50]
   - **Documentation:** Accurate and up-to-date documentation is critical for supporting operational excellence. This includes maintaining documentation for processes, procedures, policies, and systems.[50]
   - **Monitor & Evolve:** Continuous monitoring of key operational metrics provides valuable insights into the health and performance of an organization's operations. Dedicate work cycles to making nearly continuous incremental improvements. This can be justified by our Audit logging and monitoring service like AWS CloudWatch.[50]

2. ***Security:***

   - **Security Foundations:** Establish and document security policies and procedures. Define and enforce security standards across the organization. Conduct regular security training and awareness programs for personnel.[51]
   - **Identity and Access Management (IAM):** Implement strong authentication mechanisms. Enforce the principle of least privilege for access to resources. Regularly audit and review user access rights. We have implemented this step using the AWS IAM service.[51]
   - **Infrastructure Protection:** Design and implement security measures at the infrastructure level. Utilize firewalls, security groups, and network ACLs to control traffic. This is done using AWS WAF in our application.[51]
   - **Detection:** Implement mechanisms for detecting and responding to security events. Utilize automated tools for continuous monitoring of security metrics.[51]
   - **Data protection:** Classify data based on sensitivity and implement appropriate controls. Encrypt data at rest and in transit using strong cryptographic protocols. Implement data masking or tokenization where applicable.[51]
   - **Incident Response:** Develop and test an incident response plan. Establish communication channels for reporting and responding to incidents.[51]
   - **Application security:** Integrate security into the software development life cycle. Perform regular security assessments, including code reviews and penetration testing. Implement security controls for application-level vulnerabilities.[51]

3. ***Reliability:***

   - **Foundations:** Implement redundant components to enhance fault tolerance. Leverage multiple Availability Zones to distribute resources for increased resilience. Employ well-established networking and security practices to fortify the foundation. We have designed our application to be deployed in multiple availability zones and regions to make it fault-tolerant.[52]

- **Workload Architecture:** Design the workload to scale horizontally, enabling it to handle increased demand by adding resources dynamically. Utilize load balancing to distribute traffic evenly across instances for optimal performance. Provide service contracts per API. This is implemented in our project using AWS EC2 and AWS ELB which distributes the load according to the incoming traffic.[52]
- **Change Management:** Configure the workload to monitor logs and metrics and send notifications when thresholds are crossed or significant events occur. Implement automation for deploying changes to reduce the risk of human error. Conduct thorough testing before deploying changes to identify and address potential issues proactively. We have used AWS CloudWatch and AWS Grafana for it.[52]
- **Failure Management:** Backing up data regularly and protection of the workload should be done using the fault isolation method. Anticipating and designing for component failures is critical to ensure the overall resilience of the workload. Regularly conduct disaster recovery drills to ensure the effectiveness of the recovery plan. This is designed in our project using AWS Elastic Disaster Recovery.[52]

4. ***Performance Efficiency:***

- **Architecture Selection:** Choosing the right architecture is very crucial, it involves factors like cost, available services, and resources, also evaluate how our trade-off affects the customers as well as architects. Adopt a modular and scalable architecture that allows for efficient resource allocation and expansion as the application evolves.[53]
- **Compute and hardware:** Choose appropriate instance types and sizes to match the workload characteristics, ensuring optimal utilization of compute resources. Leverage the cloud's elasticity to dynamically scale your compute resources in response to workload fluctuations, preventing both over-provisioning and under-provisioning of capacity. Enhance efficiency by incorporating hardware accelerators for specific tasks, enabling more effective performance compared to alternatives relying solely on CPU-based processing.[53]
- **Data management:** Optimize database performance through indexing, partitioning, and appropriate storage choices to ensure efficient data retrieval and processing. Deploy strategies aimed at optimizing data and enhancing data query processes to achieve greater scalability and improved performance efficiency for your workload. Implement caching mechanisms to reduce database load and enhance data retrieval speed. This is done using AWS S3 and AWS Backup in our application.[53]
- **Networking and Content Delivery:** Utilize CDNs to cache and deliver content closer to end-users, reducing latency and improving overall application performance. Optimize network configurations to minimize latency, enhance data transfer speeds, and improve overall responsiveness.[53]
- **Process and Culture:** Recognize Key Performance Indicators (KPIs) that provide both quantitative and qualitative metrics for assessing workload performance. Integrate continuous performance testing into the development process to identify and address performance issues early in the application lifecycle.[53]

5. ***Cost Optimization:***

- **Practice Cloud Financial Management:** Regularly track and analyze spending, identifying areas for potential optimization. Set up monitoring services that send alerts when any particular threshold is reached. Establish budgets for the cloud and prepare forecast studies. AWS Cost and Usage Reporting is implemented to continuously monitor and track [54]
- **Expenditure and Usage Awareness:** Implement real-time monitoring of expenditure and resource usage. Set up alerts and notifications to proactively manage and control unexpected costs.[54]
- **Cost-Effective Resources:** Leverage Reserved Instances for steady-state workloads to achieve significant cost savings. Optimize resource allocation by selecting the right instance types and sizes. AWS Cost calculator is used to determine the cost and budget of each resource used, this strategy is used in our project.[54]

- **Manage Demand and Supply Resources:** Introduce throttling mechanisms to manage client retries effectively. Employ buffering techniques to store incoming requests, allowing for deferred processing at a later time. Supply the resources dynamically. Strategically use on-demand resources for variable workloads to avoid unnecessary expenses.[54]
- **Optimize Over Time:** Regularly review and refine your architecture for cost efficiency. Implement automation for continuous optimization, adjusting resources based on demand. This is also done by AWS Cost and Usage Reporting.[54]

6. ***Sustainability:***

- **Region selection:** The selection of a cloud region for workload substantially impacts its key performance indicators (KPIs), including performance, cost, and environmental impact. To optimize these KPIs, align the cloud region decisions with both business requirements and sustainability objectives.[55]
- **Alignment to demand:** Scale the workload architecture dynamically, review and optimize workload SLAs to minimize resources and increase sustainability. Implement throttling and buffering. AWS EKS has node groups and they have a minimum number of EC2, on top of that we have auto-scaling policies that can be spun up when there is demand, which increases sustainability.[55]
- **Software and architecture:** Leverage efficient software and architecture patterns like queue-driven architectures to ensure sustained high utilization of deployed resources. Opt for software patterns and architectures that optimize data access and storage, minimizing the overall demand on compute, networking, and storage resources needed to support the workload.[55]
- **Data management:** Utilize technologies that facilitate data access and storage patterns. Implement policies to effectively manage the lifecycle of your datasets. Leverage elasticity and automation to seamlessly expand block storage or file systems. Regularly eliminate unnecessary or redundant data to optimize resource utilization.[55]
- **Hardware and services:** Minimize the hardware footprint to meet your requirements efficiently. Optimize the utilization of hardware-based compute accelerators. Use the managed services provided by the cloud provider.[55]

5.2 Discussion of pillars

5.2.1 Security Pillar:

Security is a paramount consideration in cloud applications, encompassing a comprehensive set of practices and measures designed to safeguard data, systems, and infrastructure. The Security pillar is foundational, reflecting our commitment to ensuring confidentiality, integrity, and availability of resources.

As organizations transition to the cloud, ensuring the security of digital assets becomes a foundational imperative. The Security pillar addresses the multifaceted challenges associated with protecting data, applications, and infrastructure in a cloud environment. It goes beyond traditional security measures, acknowledging the shared responsibility model and emphasizing proactive, automated, and integrated security practices.

There are various design principles[46] for Security pillar like:

1. Establish a robust identity and access management framework to ensure that only authorized entities have access to resources. Implement strong authentication and authorization mechanisms to build a secure foundation.[46]
2. Implement logging and monitoring mechanisms to enable traceability. [46]
3. Implement a defense-in-depth strategy by applying security measures at every layer of the architecture.[46]
4. Integrate security into the development and deployment pipeline through automation. Implement continuous security practices, including automated security testing, to identify and remediate vulnerabilities early in the development lifecycle.[46]
5. Implement encryption for data both in transit and at rest.[46]
6. Minimize human access to sensitive data by implementing the principle of least privilege.[46]
7. Define processes for detecting, responding to, and recovering from security incidents. Regularly conduct security drills to test and improve incident response capabilities.[46]

For our project, the Security pillar is very important because our application deals with patients and hospital data which is very sensitive. From the above design principles, we have incorporated IAM, logging and monitoring, encryption for data at both ends: HTTP and at REST, and regular data backup and recovery process.

5.2.2 Cost Optimization Pillar:

Cost optimization is a strategic approach that aims to maximize the value of resources while minimizing unnecessary expenses.[46] In our cloud project, the Cost Optimization pillar plays a crucial role in ensuring that we achieve operational efficiency without compromising performance or scalability.

The Cost Optimization pillar is designed to ensure that organizations maximize the value of their cloud investments, balancing efficiency, performance, and financial stewardship.[46] This pillar addresses not only the immediate financial considerations but also the long-term sustainability of cloud operations.

There are various design principles[46] for Cost Optimization pillar like:

1. Develop and implement effective financial management practices to gain visibility into cloud spending. This includes budgeting, tracking, and optimizing costs to ensure alignment with organizational goals.[46]
2. Embrace a pay-as-you-go model, allowing resources to be provisioned and scaled based on actual consumption. This ensures that costs align directly with usage, promoting efficiency and avoiding overprovisioning.[46]
3. Continuously assess the efficiency of cloud resources and workloads. Regularly review performance metrics, identify areas for improvement, and optimize configurations to achieve maximum efficiency.[46]

4. Focus on high-value activities by offloading routine operational tasks to managed services. This allows teams to concentrate on innovation and strategic initiatives, reducing the effort and cost associated with undifferentiated heavy lifting.[46]

5. Implement tools and processes for analyzing and attributing expenditure. This involves tracking costs back to specific teams or projects, enabling accountability, and facilitating informed decision-making.[46]

For our project application, we are considering some of the design principles stated above like implementing financial management practices, implementing tools and services for analyzing and tracking cost and usage of resources, and reviewing the performance metrics.

5.3 Use of Cloud formation diagrams

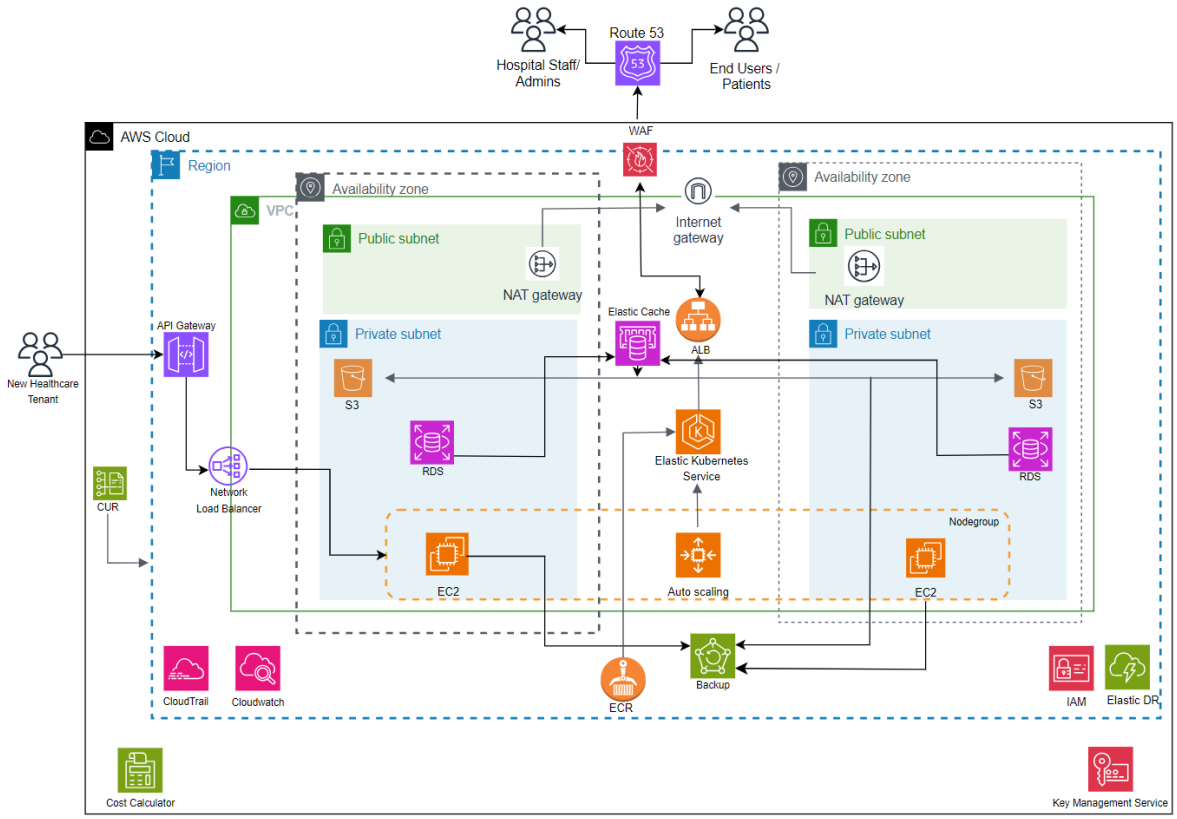

Figure 1: Cloud architecture diagram

The above diagram shows the cloud architecture of how the healthcare application would be deployed over the AWS cloud. All the important and trivial services have been listed in section **3.3**, and justification has also been provided for the choice of the same. Following is the complete explanation of the architecture and its components.
In the proposed healthcare application architecture, we intend to have 3 types of users/tenants. The following are the types :

1. End users/patients
2. Hospital staff/administrator
3. New Healthcare tenant

Any user/tenant will access the application via an IAM role associated with him/her. Even when considering a new healthcare tenant trying to onboard to the application or accessing it via APIs, the API gateway will authenticate using Identity and Access Management **(IAM)** roles for secure and fast authentication. The entire architecture in this case is shown to be deployed over 2 **availability zones**, with a Virtual Private CLoud **(VPC)** spanning over all the AZs.

The end user or a hospital staff will access the application via a DNS which will be provided by the AWS **Route53** service. Route53 spans over multiple AWS regions and Availability zones and routes the traffic based on the user's geographical location and the health of the available resources using the Anycast routing algorithm. We have Web application firewall **(WAF)** attached to Route53 to protect the web application from common web exploits and attacks. WAF contains a component called Web ACL (Access Control List) which is a set of rules and conditions that define how WAF should inspect and filter incoming web traffic. These rules and conditions enforce the security policies that protect the application. The WAF upon successful traffic filtering, passes the request to the Application Load Balancer **(ALB)**, responsible for Layer 7

(Application level) load balancing. ALB does content-based routing, allowing to route requests to different backend services based on the content of the request. The ALB forwards the request to the Elastic Kubernetes Service **(EKS)** where the actual workload is running. EKS has certain nodegroups attached to it which contain multiple Elastic cloud compute **(EC2)** instances. These EC2 instances are provisioned dynamically as per the varying workload. Depending on a threshold of CPU utilization, network throughput, etc, the autoscaling policy determines how many EC2 nodes are to be provisioned and we can also define the scale-down policies in the same way. In the first place, EKS gets the application image from Elastic Container Repository **(ECR)**, which contains all the successfully built images of the healthcare application. The application uses AWS Relational Database Service **(S3)** for storing relational data. the architecture also uses S3 to store any kind of object storage needs, like, flat files, binaries, data dumps, etc. All the data across the RDS and S3 storage is cached in AWS **Elasticache**, via managed Redis servers. Redundant read replicas of Redis servers are maintained to fetch data at high speeds, ensuring high performance and low latency. Elasticache provides multi-availability zone replication ensuring high availability and also scales automatically as per the varying workload requirements.

In the case of a new Healthcare provider who wants to use the already existing APIs before on-boarding on the application, would require IAM roles to be granted to access the APIs. This flow would be a little bit different from the general one. This flow would first go through an **API Gateway** checking the security and authentication and granting access to the authorized user to be forwarded to the Network Load Balancer **(NLB)**. The request would then be forwarded to the workload running on the EKS cluster.

From the point of security, we have included IAM for Role-based Access Control to the application and the APIs. We also have included **AWS Cloudtrail**, which enables governance, compliance, operational auditing, and risk auditing of your AWS account, especially enforcing HIPAA compliance. Since this application will have an enormous amount of sensitive patient data, it's very important to have air-tight security for the application and the data involved. The architecture includes Key Management System **(KMS)**, to create and manage cryptographic keys for use with AWS services. It supports both symmetric and asymmetric keys. AWS KMS supports automatic key rotation for customer-managed keys. This helps improve security by regularly replacing old keys with new ones.

We have used AWS Elastic Disaster Recovery **(EDR)** service for replicating the architecture into a region of interest where we want to have the application up and running when it fails in the primary region. It saves costs by removing idle recovery site resources and paying for a full disaster recovery site only when needed. For the observability and monitoring stack, we have implemented AWS **Cloudwatch**. This helps us generate metrics for every resource that we use check how the resources are performing and visualize the trends of the resource performance. Also, we set alerts and triggers on the basis of a certain event.

For cost calculation and budgeting, we have used **AWS Cost Calculator**. This provides us with an insight into how much the architecture created will cost us on average and that will help us in budgeting our resources accordingly. AWS Cost and Usage Reporting tool **(CUR)**, will help us to generate detailed reports about AWS spending and usage. These reports will provide insights into how resources are being used, which services are contributing to costs, and how changes to your environment impact spending.

5.4 Validation of the design

Sr. No.	*Technical Requirement*	*Corresponding service from design & Justification*
1	**TR-1.1** Auto-Scaling Mechanism	We have designed this technical requirement using **AWS Cluster Auto Scaling**, it dynamically scales up and down resources depending upon the load.
2	**TR-1.2** Application Performance Monitoring	We have designed this requirement using the **AWS Cloudwatch**, it helps in tracking the system logs which can later be utilized for performance enhancement.
3	**TR-1.3** Distributed and Scalable Architecture	**AWS Elastic Load Balancer** is designed to provide a distributed architecture that can automatically distribute the incoming network traffic and redirect to the different resources or instances.
4	**TR-2.1** Multiple Availability zones	To implement high availability as a requirement, we have included instance autoscaling on demand as per the varying workload which would ensure multiple availability zones. The service used for this is **AWS Auto-scaling** and **AWS EC2**.
5	**TR-2.2** Multi-region deployment architecture	To design this requirement, we have implemented **Elastic Disaster Recovery** which would ensure redundant architecture design in multiple regions.
6	**TR-2.3** Monitoring and observability	We have used **AWS CloudWatch** to monitor the system logs, and operational metrics and keep track of the observability pipeline.
7	**TR-3.1** Data Encryption and protection	The WAF pillar Security is heavily designed using the data encryption principle and for implementing the same, we have made use of **AWS Key Management Service** which encrypts and decrypts the data at not only HTTP level but also at REST level for ensure highly secure architecture.
8	**TR-3.2** HIPAA Compliance	**AWS CloudTrail** helps us maintain the rules and regulations of HIPAA compliance which is essential for our healthcare application.
9	**TR-3.3** Web Application Firewall & Throttling for APIs	**AWS WAF** makes use of several firewall techniques to prevent attacks like DDOS, and also ensure the strategy of throttling and buffering for APIs.
10	**TR-4.3** Use of standard APIs	To support encryption of data at REST and HTTP, the service **AWS APIGateway** helps us to have a secure communication channel between client and APIs. It also allows secure access only via RBAC protocols.
11	**TR-5.1** Cost Optimization Tools	**AWS Cost and Usage Reporting** tool helps us to track the cost of resource usage and helps us monitor and optimize the budget using comprehensive reports.
12	**TR-5.2** Resource budgeting	For optimal use of budgets and the cloud resources, we have used **AWS Cost Calculator** to simulate the needs for the architecture and the related cost. This helped us in prior budgeting of the required services and planning ahead.

Sr. No.	***Technical Requirement***	***Corresponding service from design & Justification***
13	**TR-6.1** Regular Data Backup	To ensure regular data backup, we have implemented **AWS EDR** and **AWS Backup**. AWS backup helps us in centralized and automated data protection across all AWS services. We can create multiple and relevant backup policies for every service.
14	**TR-6.2** Data Recovery	We have implemented the TR using **AWS Elastic Disaster Recovery** service which provides different redundant architecture creation for being fault tolerant.
15	**TR-6.3** Failover mechanism	To ensure there is a passive instance always available when the primary architecture goes down, we implemented **AWS Disaster Recovery(EDR)** service. This will ascertain that the application is always available and requests will be redirected to the passive instances with very minimal downtime, as stated in the SLA.
16	**TR-7.1** Implement RBAC	We have implemented role-based access and control using the **Identity and Access Management (IAM) service**.This ensures secure access to applications and APIs based on the roles set for every individual with no extra or redundant accesses.
17	**TR-7.2** Secure API Gateways	To implement secure API access, we included **AWS API Gateway** service in our architecture. This ensures secure access to APIs as it implements RBAC for access.
18	**TR-8.1** Observability pipeline	We have included **AWS Cloudwatch** as the service for implementing the observability pipeline. It includes Prometheus which uses the Cloudwatch agent to send the metrics data and logs and that in turn could be used for further visualization and alerting.
19	**TR-8.2** Comprehensive Monitoring and Alerts	We have incorporated **AWS CloudWatch** as the primary service for establishing our observability pipeline. Utilizing the CloudWatch agent to transmit metrics data and logs. This integration enables us to leverage the collected data for advanced visualization and alerting purposes.
20	**TR-8.3** Generate actionable insights	By implementing **AWS Cloudwatch**, we can monitor applications and their related metrics which gives an insight into its performance. We can also create actionable insights and generate alerts based on the events triggered.
21	**TR-9.1** Tenant Identification & Isolation	**The AWS IAM (Identity and Access Management)** service justifies the tenant identification technical requirement by providing a robust framework for securely managing and controlling access to AWS resources, allowing for the precise definition of roles and permissions specific to each tenant.
21	**TR-10.1** Container Orchestration	**AWS Elastic Kubernetes Service** is helpful in making it easier to deploy and manage the containerized applications on AWS. It is a managed Kubernetes service.

Sr. No.	***Technical Requirement***	***Corresponding service from design & Justification***
22	**TR-10.2** Caching Mechanism	**AWS Elasticache** justifies the Performance Efficiency design principle through its caching mechanism by providing a highly scalable and low-latency in-memory data store. By caching frequently accessed data, Elasticache reduces the need for repeated database queries, accelerating data retrieval and enhancing overall application performance.
23	**TR-11.1** Primary-Secondary Failover Mechanism	We have implemented the TR using **AWS Elastic Disaster Recovery** service which provides different redundant architecture creation for being fault tolerant. So in any data failure scenario, the secondary architecture can act as the primary architecture and all the traffic can be redirected to it.
24	**TR-11.2** Routine Data storage & Backup	This technical requirement is fulfilled by using **AWS S3** and **AWS Backup** in which the data is regularly stored and backed up at regular intervals.

Table 5: Validation of the design and justification for each TR

5.5 Design principles and best practices used

5.5.1 Operational Excellence:

The design principles used for the operational excellence pillar are as mentioned below:

1. **Perform operations as code:** This principle emphasizes treating infrastructure and operations procedures as code, to implement this we have used a version control system and automated testing so we can track the code changes and represent them as code.[46]

2. **Learn from all operational failures:** This principle emphasizes the importance of continuous learning and evolving to prevent similar issues in the future. We have dealt with minor failures in our deployment and testing, through which we happen to learn a lot.[46]

Here are some of the best practices that we have used in our design of the application deployment:

1. **Collecting Metrics for Measurement:** We have gathered metrics to quantify and measure the achievement of desired outcomes. Metrics can include performance indicators, error rates, response times, and other relevant data points. [46]

5.5.2 Security:

The design principles used for the security pillar are as mentioned below:

1. **Protect data in transit and at rest:** We have incorporated this design principle by implementing encryption at both levels: HTTP and at REST. This makes the system highly secure.[46]

2. **Implement a strong identity foundation:** This principle is implemented by establishing a robust identity and access management system. Users, systems, and processes should have distinct identities with appropriate permissions. This helps in controlling and auditing access to resources, reducing the risk of unauthorized access and potential security breaches.[46]

3. **Enable traceability:** Traceability involves the ability to monitor and track activities in your cloud environment. This includes logging and auditing actions taken by users and systems. We have designed this by AWS Cloudwatch to monitor the logs.[46]

Here are some of the best practices that we have used in our design of the application deployment:

1. **Identify Security Incidents:** It's important to have mechanisms in place to identify and detect security incidents promptly. This includes implementing monitoring, logging, and alerting systems that can provide insights into activities within the cloud environment.[46]

2. **AWS Shared Responsibility Model:** The AWS Shared Responsibility Model clarifies the division of responsibilities between AWS and the customer. AWS is responsible for the security "of" the cloud, including the physical infrastructure, while customers are responsible for the security "of" the cloud, such as configuring and securing their own data, applications, and identity/access management. Understanding this model has helped us leverage AWS services while ensuring their specific security responsibilities are met.[46]

5.5.3 Reliability:

The design principles used for the reliability pillar are as mentioned below:

1. **Automatically recover from failure:** Design your systems to automatically recover from failures without requiring manual intervention. This involves implementing mechanisms such as auto-scaling, load balancing, and fault tolerance to ensure that, in the event of a failure, the system can quickly and automatically recover to a healthy state. This is satisfied by AWS Auto-scaling, AWS ELB, and Disaster Recovery Site.[46]

2. **Scale horizontally to increase aggregate workload availability:** Instead of relying solely on vertical scaling (adding more resources to a single server), design your systems to scale horizontally by adding more instances or nodes. Horizontal scaling is done by the feature AWS Auto-scaling implemented in our design.[46]

3. **Stop guessing capacity:** Avoid making capacity planning decisions based on guesses or predictions. Instead, monitor and analyze the actual usage patterns of your system, and use that data to make informed decisions about capacity requirements. We have used this practice to adjust the capacity dynamically depending on the load.[46]

Here are some of the best practices that we have used in our design of the application deployment:

1. **Resilience in the Cloud:** While low-level hardware component failures are a common concern in on-premises data centers, cloud providers like AWS often abstract away these concerns. However, there is still the potential for failures to impact workloads in the cloud. To address this, it's essential to implement resiliency in the workload. This is implemented through maintaining a failover mechanism and having a disaster recovery strategy in place.[46]

5.5.4 Performance Efficiency:

The design principles used for the performance efficiency pillar are as mentioned below:

1. **Going global in minutes:** Leverage cloud services to quickly expand your application's reach to a global audience. With cloud providers like AWS, we have deployed resources and services in multiple regions around the world, also they can be automated deployed because we have implemented using IaaC.[46]

Here are some of the best practices that we have used in our design of the application deployment:

1. **Make trade-offs for performance improvement:** Understanding the trade-offs and making intentional decisions based on performance goals are key aspects of optimizing an architecture. We have implemented a caching mechanism to justify this design.[46]

5.5.5 Cost Optimization:

The design principles used for the cost optimization pillar are as mentioned below:

1. **Implement cloud financial management:** We have implemented this using the AWS Cost and Usage Reporting service. This includes creating budgets, tracking expenses, and implementing cost-allocation strategies.[46]

5.5.6 Sustainability:

The design principles used for the cost optimization pillar are as mentioned below:

1. **Maximize utilization:** We have set the threshold utilization of the CPU in case of auto-scaling of cluster nodes to the maximum rate possible where we don't find any application throttling but at the same time don't waste any resources. The new nodes get provisioned only when it hits the maximum utilization limit and it actually requires the new node for a better application performance, thus ensuring future resource sustainability.[46]

5.6 Trade offs revisited

We had discussed the trade-offs in the architecture design on the basis of the Business and technical requirements in section 2.4. Here, in this section, we revisit the same trade-offs and justify them on the basis of the services chosen and the final architecture constructed.

1. **High Availability vs Cost:** [**Selected High Availability**]
   In our case, we have deployed the application over multiple availability zones. Also, to make the application highly fault-tolerant, we have deployed the redundant architecture across another region using the Disaster Recovery Site strategy. This creates a need for multiple maintenance of redundant resources like EC2 instances, ECR repositories, API gateways WAFs, etc, which adds to the cost of maintaining the architecture. Although there is a trade-off here between High Availability and cost, we chose high availability as this will be used by a lot of patients and will be used in emergency situations and high availability is the obvious choice in that scenario.

2. **Security vs Performance Efficiency:** [**Selected Security**]
   We are using EKS for distributed workload execution and container orchestration techniques. Although EKs is very efficient in managing loads, it does have a security key store, to store encrypted keys and secrets. Accessing these secrets will require a considerable amount of time when there are a million concurrent users, resulting in a performance overhead. Since security takes precedence over any other aspect, we will be choosing security in this case over performance overhead and try to better the performance in another way like caching certain data that are used pretty often. Also, being an application that deals with very sensitive data of patients, it's imperative to choose security over a minor performance overhead.

5.7 Discussion of an alternate design (Skipped)

6 Kubernetes experimentation

6.1 Experiment Design

6.1.1 Description

This experiment is intended to design, deploy, and test an application on the AWS cloud, with the application deployed over AWS Elastic Kubernetes Service**(EKS)**. The application image is taken from the docker hub official images website [19]. The image used is linuxserver/nextcloud. With **Nextcloud** [20] you pick a server of your choice, at home, in a data center or at a provider. And that is where your files will be. Nextcloud runs on that server, protecting your data and giving you access from your desktop or mobile devices. Following is the image of how the UI looks like once the application is successfully deployed on AWS EKS :

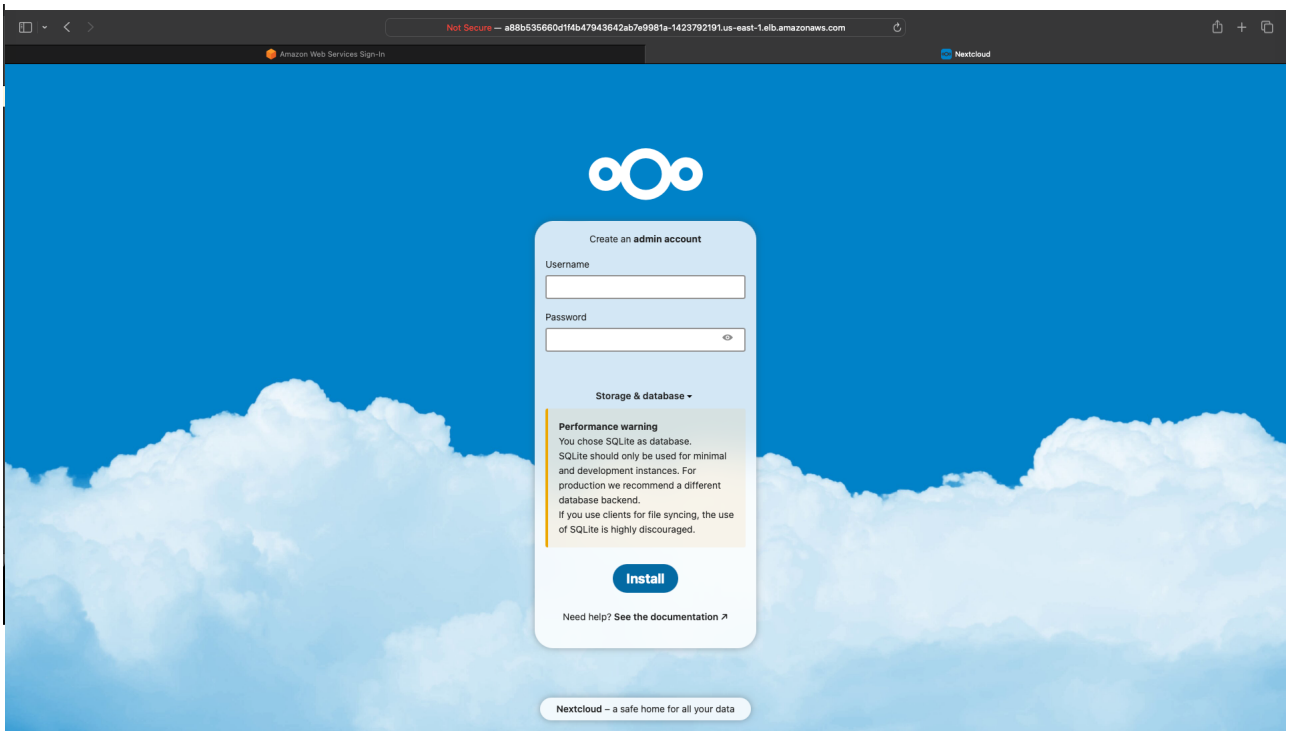

Figure 2: Deployed application UI

6.1.2 TR's worked on during this experiment

1. **TR-1.1** Auto-Scaling Mechanism
2. **TR-1.3** Distributed and Scalable architecture
3. **TR-2.1** Multiple availability zones
4. **TR-10.1** Container Orchestration

6.1.3 Design

The image used is first pushed to AWS Elastic container registry**(ECR)**. An EKS cluster is then created with an associated nodegroup containing Elastic Compute Cloud **(EC2)** instances. For this experiment, we deployed the solution in 6 different availability zones, namely :

1. us-east-1a
2. us-east-1b
3. us-east-1c

4. us-east-1d
5. us-east-1e
6. us-east-1f

The deployment of the EC2s in multiple availability zones justifies **TR-2.1**.
Two levels of autoscaling have been implemented in this design :

1. **Cluster Autoscaling** : This is configured at the node level in the nodegroup. Whenever there is a spike in CPU usage of the application above 40%, the policy attached to the nodegroup will automatically spin up an EC2 instance to accommodate the varying workload.
2. **Horizontal Pod Autoscaling (HPA)** : the concept of HPA has been implemented on the pods deployed on the EKS cluster. Whenever the CPU usage threshold for a particular pod increases than what is mentioned $\geq$**70**, a new pod is created automatically by the Kubernetes cluster. We have defined a minimum of 1 replica and a maximum of 10 replicas when the load is high enough.

For deployment of the application on the EKS cluster, two manifests have been created. A deployment manifest takes care of the pod and its resource requirement details and a service manifest takes care of keeping the application alive at a certain port for the load test to be carried out via locust.
Locust has been installed initially and the locust UI has been used to generate multiple load profiles and check how the EKS cluster and the node group behave based on the load.

6.1.4 Prerequisites

The system should have the following pre-requisites installed, before starting with the experimentation :

1. Docker [23]
2. AWS CLI [24]
3. kubectl [25]
4. Locust [26]

6.1.5 Steps followed for the EKS cluster setup and image deployment

Most of the AWS resource setup is done using the AWS portal dashboard, which we felt was easier to use than eksctl commands to set up the EKS cluster. Following are the broad steps followed for the setup :

1. Configure the IAM user on the local terminal using AWS cli.
2. Create an ECR repository using the AWS console and push the image intended to be deployed over the EKS cluster.

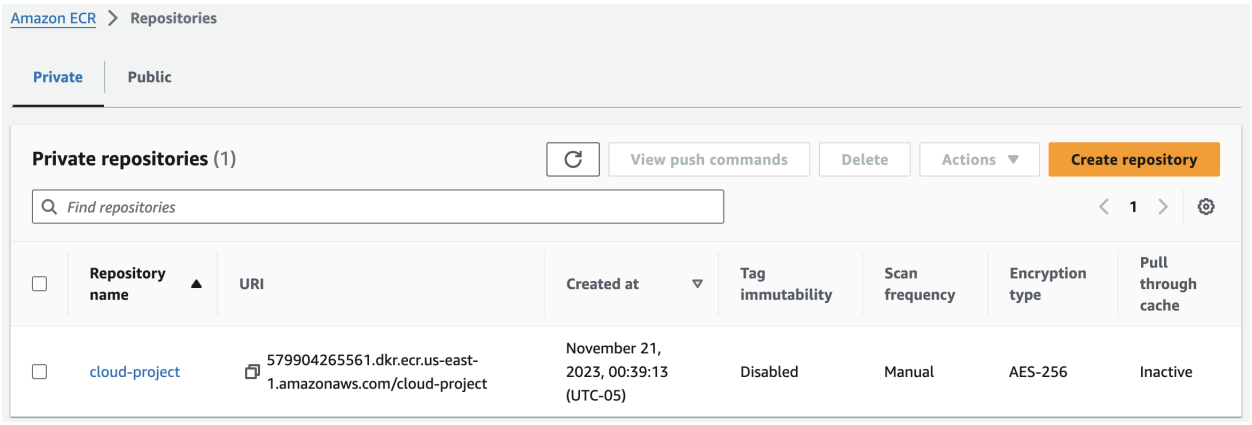

Figure 3: ECR Repository containing our application image

3. Create an EKS[27]cluster using the AWS console. This justifies **TR-10.1**.

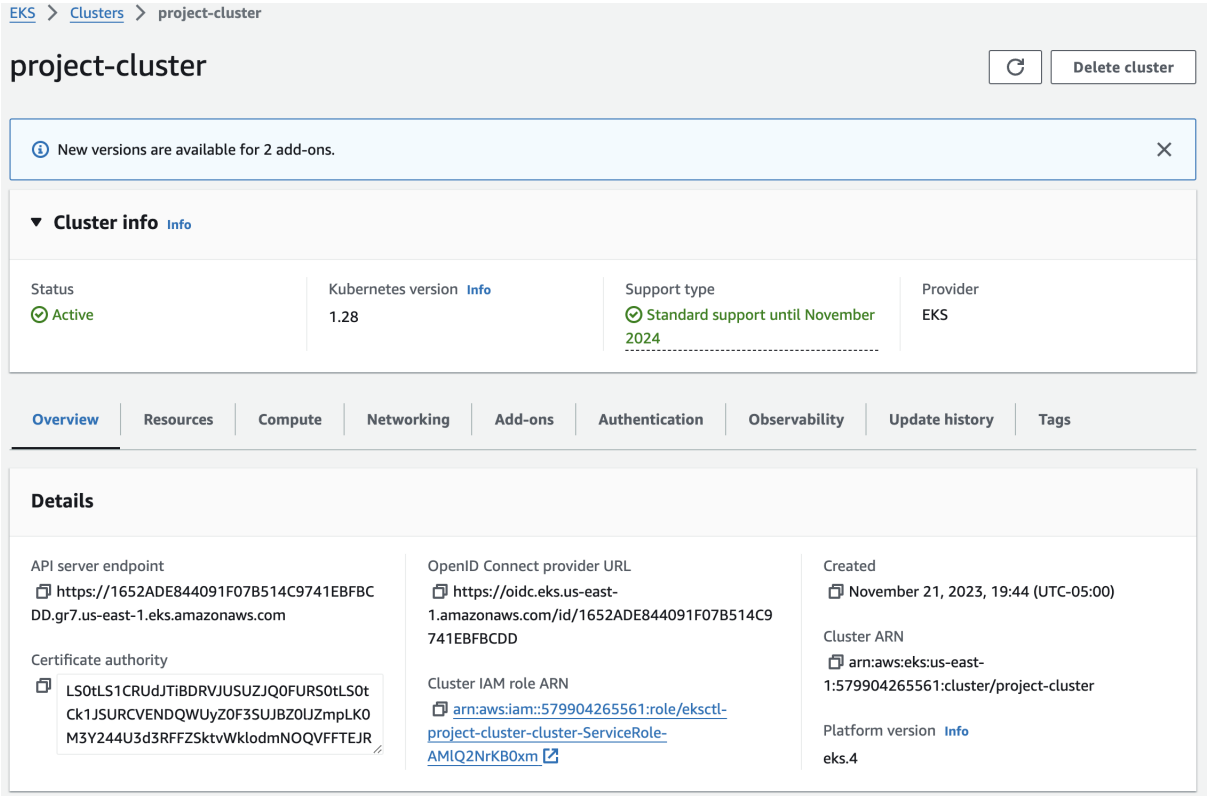

Figure 4: EKS Cluster created for the experiment

4. Create a nodegroup[28] and associate it with the EKS cluster. We have chosen two different types of VMs for the experiment :
   (a) t2.micro
   (b) t3.small

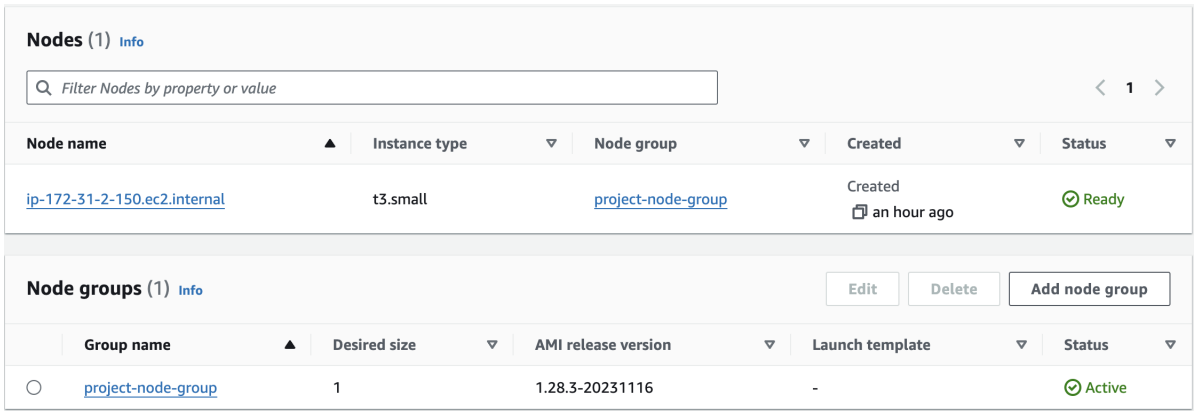

Figure 5: Node group created for the EKS cluster

5. The EC2 resources in the nodegroup automatically get associated with an auto-scaling group when it meets the dynamic policy requirement set initially. This justifies the **TR-1.1**.

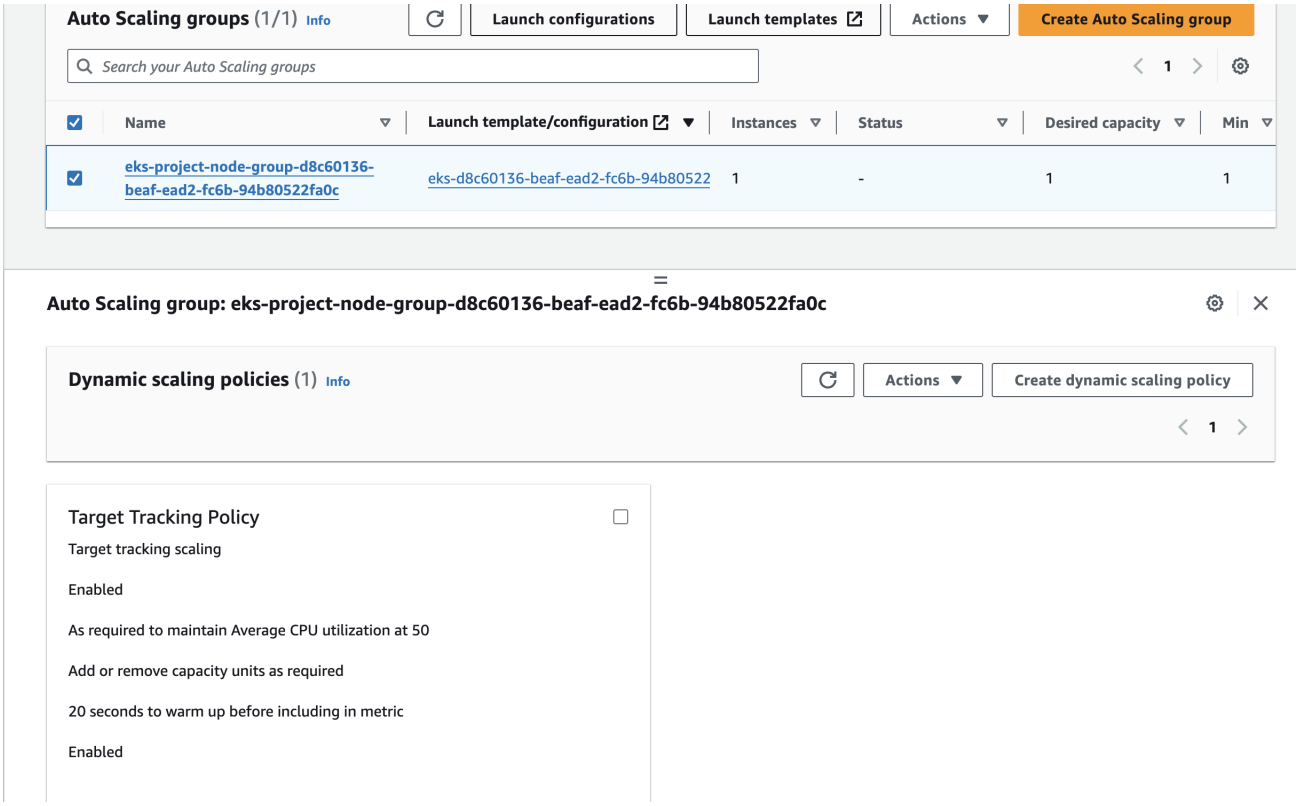

Figure 6: Autoscaling policy set for the cluster nodes

6. Once the nodegroup is created, we can use the AWS CLI command to in the terminal first to set the kubeconfig [29]context for the EKS created and then use any kubectl command on the cluster. The commands used are as follows :

   (a) **aws eks –region us-east-1 update-kubeconfig –name project-cluster**

7. Deploy the application image using the Deployment manifest and create a load balancer service for the same using a separate manifest. [27]

   (a) Deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapp-nextcloud
spec:
  replicas: 1
  selector:
    matchLabels:
      app: webapp-nextcloud
  template:
    metadata:
      labels:
        app: webapp-nextcloud
    spec:
      containers:
      - name: webapp-nextcloud
        image: linuxserver/nextcloud
        imagePullPolicy: Always
        ports:
          - containerPort: 80
```

```
        - containerPort: 443
      resources:
        requests:
          cpu: "250m"
        limits:
          cpu: "500m"
```

(b) Service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: nextcloud-service
spec:
  type: LoadBalancer
  selector:
    app: webapp-nextcloud
  ports:
    - protocol: TCP
      name: http
      port: 80
      targetPort: 80
    - protocol: TCP
      name: https
      port: 443
      targetPort: 443
```

8. Enable kubernetes metrics server [30] for the metrics of the pods to be shown when HPA is applied on the cluster. The command for enabling the metrics server is as follows :
   (a) kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml
9. Apply Kubernetes Horizontal Pod Autoscaler (HPA) [31] with a threshold CPU utilization of 80%. The command used for the same is as follows :
   (a) kubectl autoscale deployment webapp-nextcloud –cpu-percent=80 –min=1 –max=10
10. An elastic load balancer setup that directs the requests internally to the EKS cluster. The same DNS mentioned in the screenshot has been used for locust load generation. This justifies **TR-1.3**.

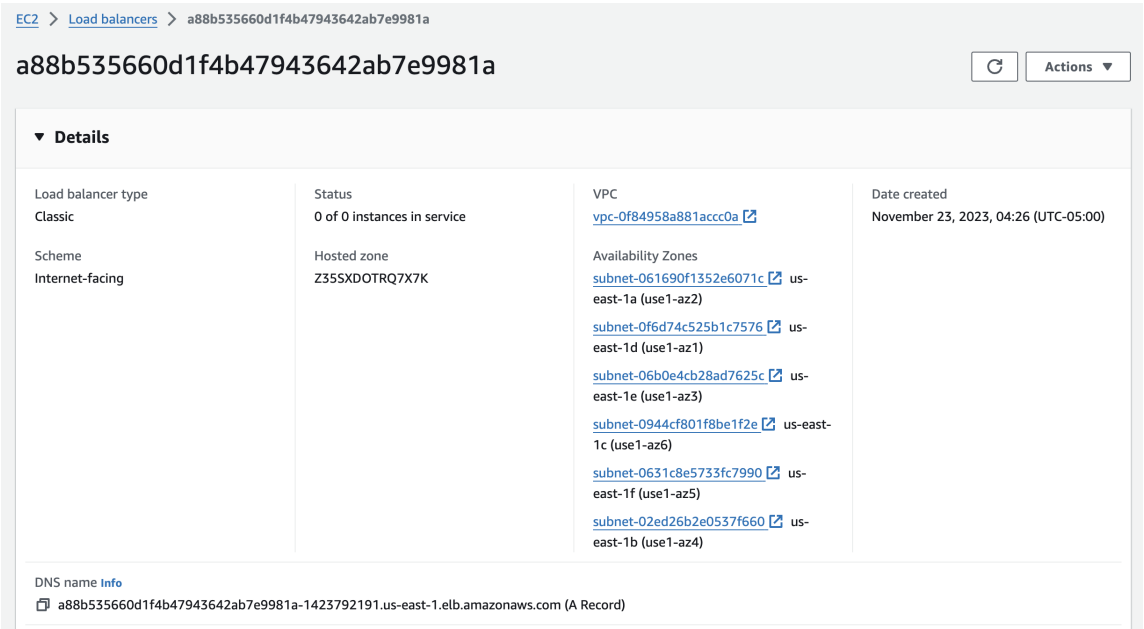

Figure 7: Elastic load balancer for EC2s in the EKS nodegroup

11. Install locust in the local system using pip.

6.2 Workload generation with Locust

6.2.1 What is locust?

Locust is a powerful open-source tool designed for evaluating web application performance through load testing. Its main goal is to simulate multiple users interacting with a system simultaneously, helping to uncover scalability issues and potential performance bottlenecks. Built in Python, Locust allows users to script user behavior using Python code, providing a high level of flexibility and customization. The tool includes real-time monitoring features, offering insights into crucial performance metrics such as response times and failure rates. This makes Locust an invaluable resource for optimizing and ensuring the reliability of web applications under various load conditions. [32]
We used the following details in locust to generate the load on the application on EKS :

1. **URL** : http://a88b535660d1f4b47943642ab7e9981a-1423792191.us-east-1.elb.amazonaws.com
2. **Method** : GET
3. Load profiles tested :
   (a) t2.micro EC2 instance with linear user load ramp up

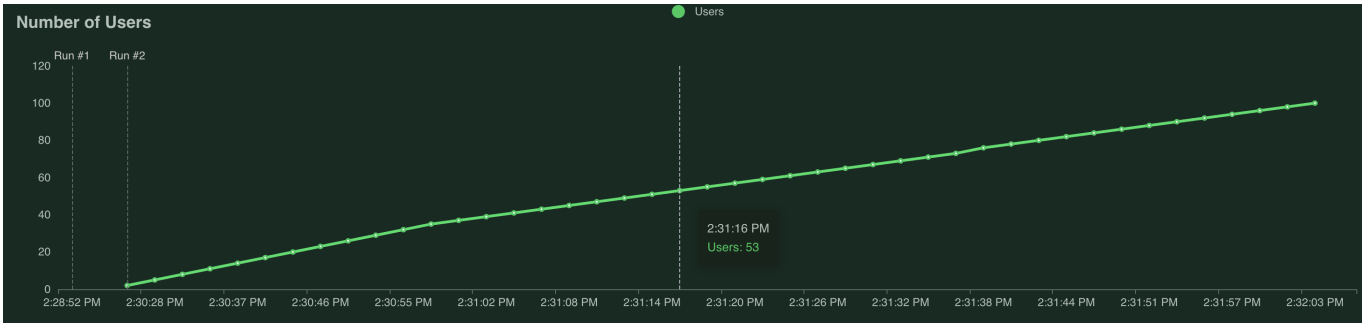

Figure 8: Linear User load

   i. 1000 users with no HPA,
   ii. 10000 users with no HPA, 1 pod (Node CPU throttling)
   iii. 5000 users with HPA

   (b) t3.small EC2 instance with exponential user load ramp-up

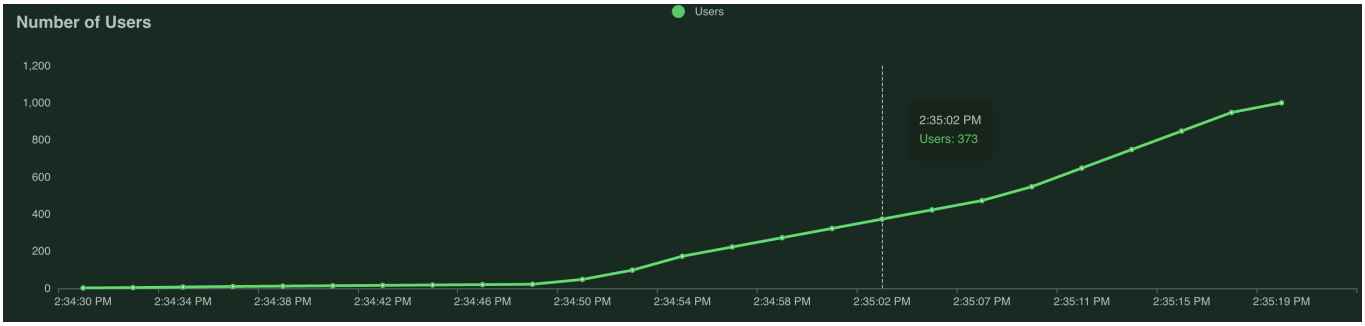

Figure 9: Exponential user load

   i. 1000 users with no HPA
   ii. 25000 users with HPA and cluster autoscaling

6.3 Analysis of the results

1. **Load test with 1000 users, without HPA and a single replica**
   In the case of both linear and exponential user load increases, the response times increased, but the trend is more uniform and gradual in the case of exponential load.

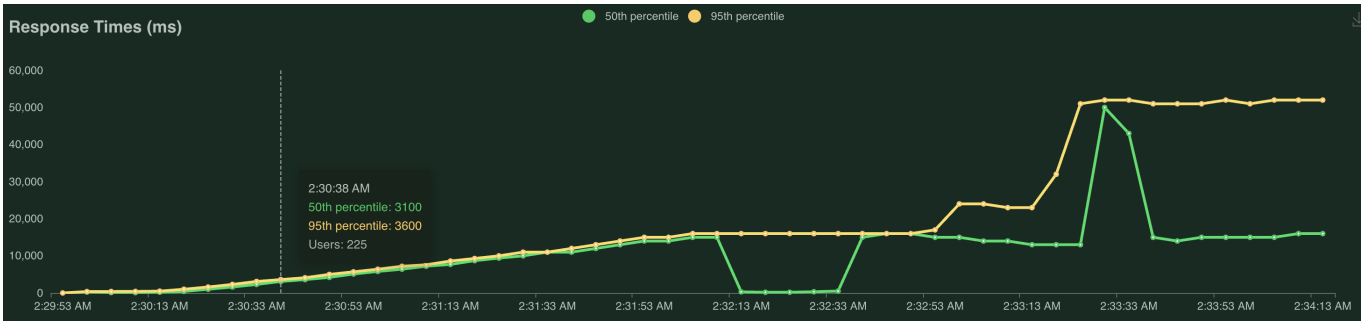

Figure 10: Linear user load response time

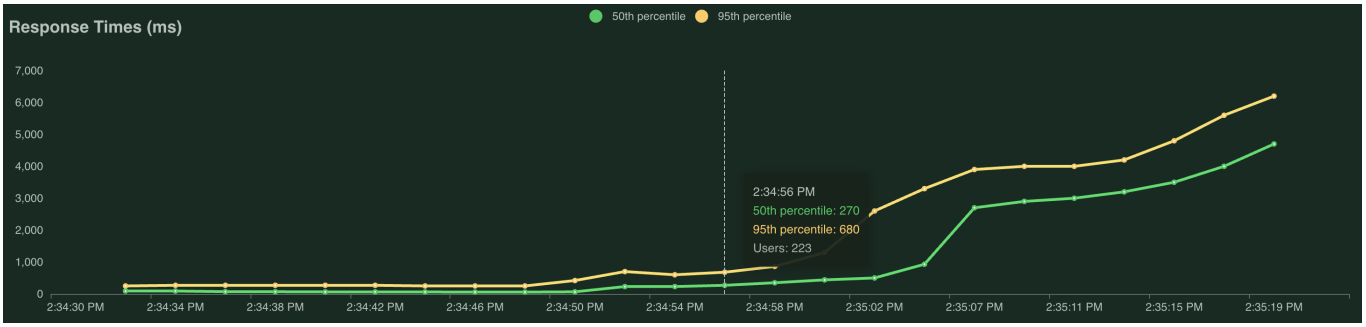

Figure 11: Exponential user load response time

2. **Load test with 10000 users and linear load with no HPA or Cluster autoscaling**
   The test with 10000 users and linear load took around 35 minutes to reach the intended number of users.

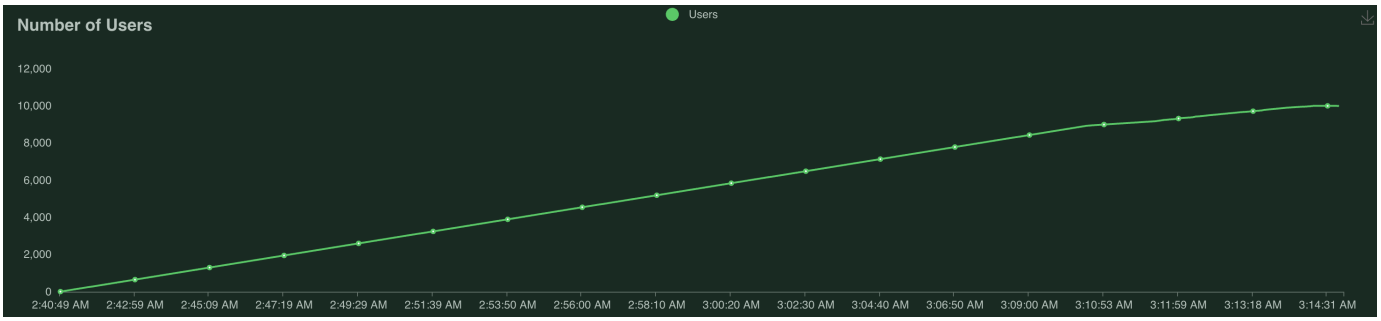

Figure 12: Linear user load for 10000 users

With the increase in the users and the concurrent load, the response time also increases. This can be validated from the below graph :

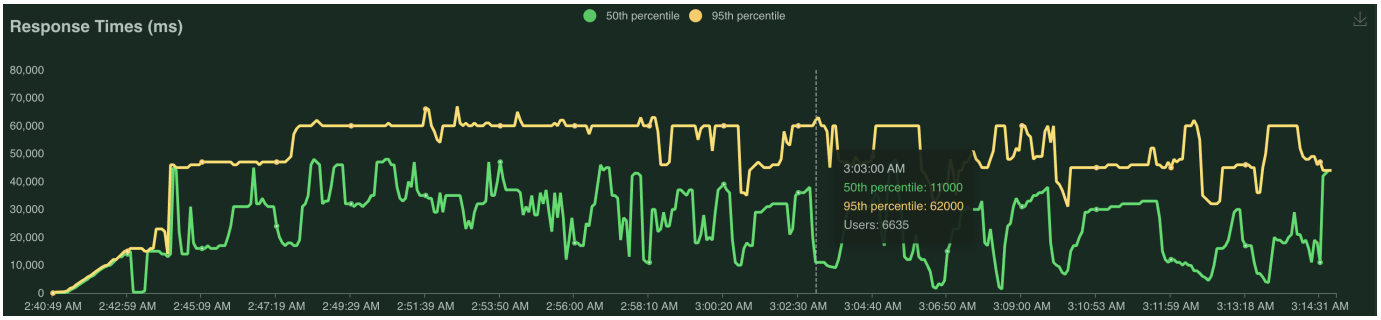

Figure 13: Response time increases as user load increases

As there was no new node to be provisioned, the complete load was on a single node. The EC2 monitoring stack also shows the increase in CPU load for this particular load test:

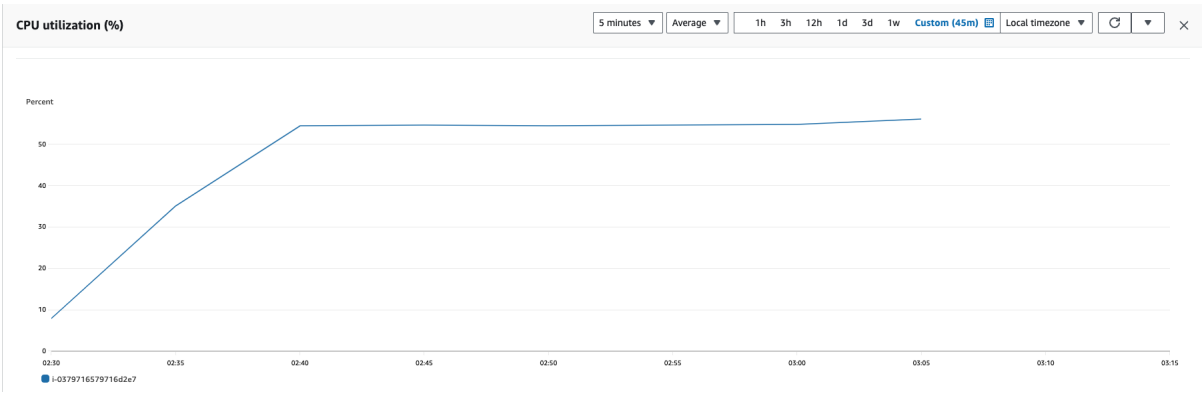

Figure 14: EC2 CPU load increases for a single node architecture

Although, with this huge user load, there were quite a few types of error requests. the following were the errors encountered during the test :

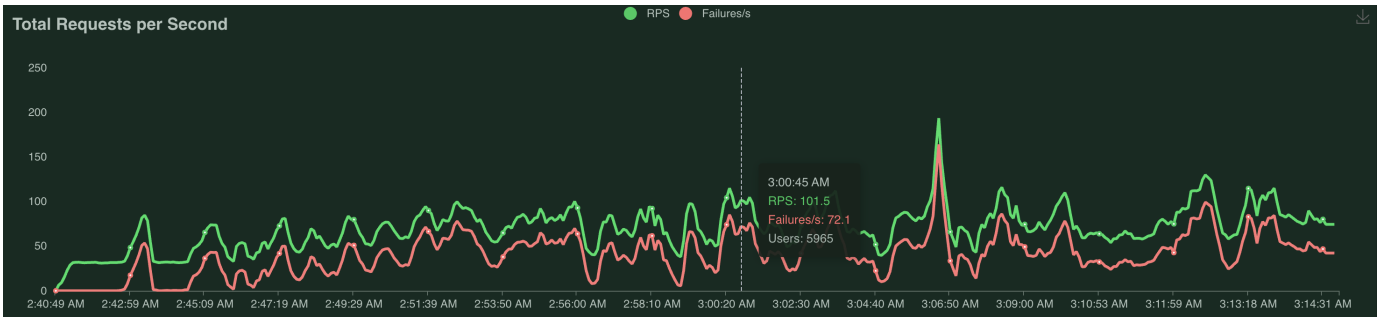

Figure 15: Error requests

# fails	Method	Name	Type
68653	GET	/:443/	RemoteDisconnected('Remote end closed connection without response')
30	GET	/:443/	gaierror(8, 'nodename nor servname provided, or not known')
13459	GET	/:443/	ConnectTimeoutError(<urllib3.connection.HTTPConnection object at 0x....>, 'Connection to aff5a5c1ff8dc44019fb5620123212fb-200671065.us-east-1.elb.amazonaws.com timed out. (connect timeout=None)')
205	GET	/:443/	ConnectionResetError(54, 'Connection reset by peer')
16	GET	/:443/	ReadTimeout(ReadTimeoutError("HTTPConnectionPool(host='aff5a5c1ff8dc44019fb5620123212fb-200671065.us-east-1.elb.amazonaws.com', port=80): Read timed out. (read timeout=None)"))
34	GET	/:443/	HTTPError('500 Server Error: Internal Server Error for url: /:443/')

Figure 16: Types of errors encountered

3. **Load test profile with Horizontal Pod Autoscaler but no Cluster Autoscaler**
   The user load in this test scenario was kept as linear which soared up to 5000. But in this case, HPA was applied on the EKS cluster with CPU utilization threshold set at 50%. The minimum number of replicas set was 1 and a maximum of 10. During the test, as per the load incremented, 4 replicas were spun off to accommodate the varying load.

```
indranil@IndranilsLaptop Project % kubectl get hpa webapp-nextcloud --watch
NAME              REFERENCE                    TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
webapp-nextcloud  Deployment/webapp-nextcloud  0%/50%    1         10        1          22s
webapp-nextcloud  Deployment/webapp-nextcloud  30%/50%   1         10        1          2m
webapp-nextcloud  Deployment/webapp-nextcloud  200%/50%  1         10        1          2m15s
webapp-nextcloud  Deployment/webapp-nextcloud  200%/50%  1         10        4          2m30s
webapp-nextcloud  Deployment/webapp-nextcloud  200%/50%  1         10        4          2m45s
webapp-nextcloud  Deployment/webapp-nextcloud  200%/50%  1         10        4          3m
webapp-nextcloud  Deployment/webapp-nextcloud  199%/50%  1         10        4          3m15s
webapp-nextcloud  Deployment/webapp-nextcloud  200%/50%  1         10        4          3m45s
webapp-nextcloud  Deployment/webapp-nextcloud  200%/50%  1         10        4          4m
```

Figure 17: HPA for 5000 linear user load

4. **Load test with exponential user load, HPA, and cluster autoscaling**
   This load profile included 25000 users with an exponential user concurrency. An autoscaling policy was set to provision a machine when the EC2 CPU utilization reached a threshold of 50%. The HPA setting was kept intact as of test3, with a maximum replica of 8. The HPA statistics are as follows for this test :

```
indranil@IndranilsLaptop Project % kubectl get hpa webapp-nextcloud --watch
NAME              REFERENCE                    TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
webapp-nextcloud  Deployment/webapp-nextcloud  0%/80%    1         8         1          32s
webapp-nextcloud  Deployment/webapp-nextcloud  1%/80%    1         8         1          75s
webapp-nextcloud  Deployment/webapp-nextcloud  23%/80%   1         8         1          90s
webapp-nextcloud  Deployment/webapp-nextcloud  55%/80%   1         8         1          105s
webapp-nextcloud  Deployment/webapp-nextcloud  94%/80%   1         8         1          2m
webapp-nextcloud  Deployment/webapp-nextcloud  200%/80%  1         8         2          2m15s
webapp-nextcloud  Deployment/webapp-nextcloud  147%/80%  1         8         3          2m30s
webapp-nextcloud  Deployment/webapp-nextcloud  147%/80%  1         8         4          2m45s
webapp-nextcloud  Deployment/webapp-nextcloud  129%/80%  1         8         4          3m1s
webapp-nextcloud  Deployment/webapp-nextcloud  131%/80%  1         8         7          3m16s
webapp-nextcloud  Deployment/webapp-nextcloud  160%/80%  1         8         7          3m31s
webapp-nextcloud  Deployment/webapp-nextcloud  132%/80%  1         8         8          3m46s
webapp-nextcloud  Deployment/webapp-nextcloud  146%/80%  1         8         8          4m1s
```

Figure 18: HPA for 25000 exponential user load test

Following are the autoscaling policy set and the state when a new EC2 was provisioned with the policy in effect :

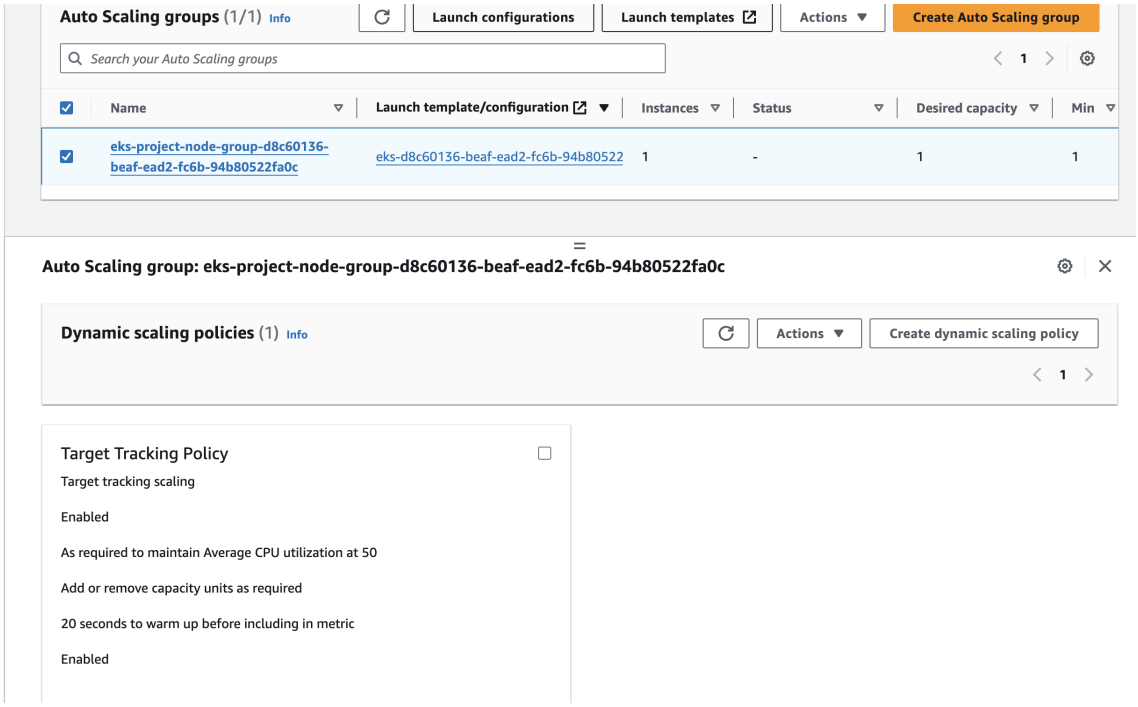

Figure 19: Autoscaling policy set for the nodegroup

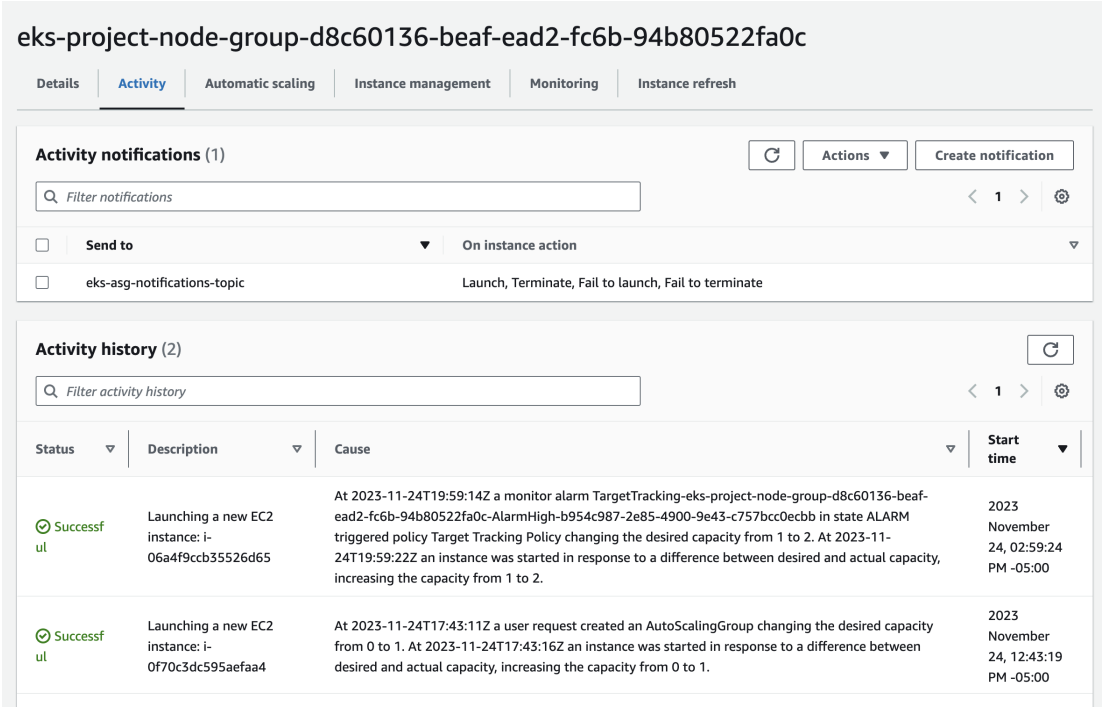

Figure 20: EC2 autoscaled from 1 to 2 resources when CPU utilization is higher than threshold

7 Ansible playbooks (Skipped)

7.1 Description of management tasks (Skipped)

7.2 Playbook Design (Skipped)

7.3 Experiment runs (Skipped)

8 Demonstration (Skipped)

9 Comparisons (Skipped)

10 Conclusion

10.1 The lessons learned

1. Importance of clearly defining and understanding the problem statement before initiating any project.
2. Emphasis on gathering both business and technical requirements to ensure an understanding of the project needs.
3. Importance of establishing detailed criteria for selecting a service provider, considering factors such as functionality, cost, and compatibility.
4. Adherence to design principles and best practices provided by AWS WAF pillars like operational excellence, security, reliability, performance efficiency, cost optimization, and sustainability.
5. Understanding the benefits of tools like Locust for workload generation and its role in performance testing.
6. Understanding the trade-offs involved in technical decisions and ensuring that these are carefully evaluated and documented.
7. Use of cloud formation diagrams to visually represent the architecture, aiding in better communication and understanding among team members.
8. Lessons learned from experimenting with Kubernetes, including the importance of well-defined experiment designs and prerequisites.
9. Acknowledging the importance of adaptability in project planning.

10.2 Possible continuation of the project (Skipped)

References

[1] Amazon EC2

[2] AWS ELB

[3] AWS Auto-scaling

[4] Amazon Managed Service for Prometheus

[5] AWS Grafana

[6] AWS WAF

[7] AWS CloudTrail

[8] AWS KMS

[9] AWS Cost Calculator

[10] AWS Cost and Usage Reporting

[11] AWS Backup

[12] AWS S3

[13] AWS Elastic Disaster Recovery

[14] AWS IAM

[15] AWS API Gateway

[16] AWS CloudWatch

[17] AWS Elastic Kubernetes Service

[18] AWS Elastic Cache

[19] DockerHub Official image store

[20] What is Nextcloud?

[21] Best Practices for HIPAA security

[22] Reliable Scalability architecting

[23] Docker Installation steps

[24] AWS CLI Installation steps

[25] Kubectl Installation steps

[26] Locust Installation steps

[27] Kubernetes Application Deployment with AWS EKS and ECR

[28] Managed node groups

[29] Kubeconfig File Explained With Practical Examples

[30] Installing the Kubernetes Metrics Server

[31] Official Kubernetes HorizontalPodAutoscaler Walkthrough

[32] Official Locust Webpage

[33] Horizontal Scaling

[34] Availability as a Business requirement.

[35] Guidance on HIPAA & Cloud Computing

[36] Interoperability: Improving the healthcare consumer experience

[37] Monitoring Cloud Resources and Costs

[38] DRS

[39] HIPAA Compliance Requirements for Access Control and Authentication

[40] Reliability and high availability in cloud computing environments: a reference roadmap

[41] Autoscaling

[42] Cloud Pricing Comparison: AWS vs. Azure vs. Google Cloud Platform in 2023

[43] GCP Cloud locations

[44] Azure Global Infrastructure

[45] AWS Global Infrastructure

[46] The 6 Pillars of the AWS Well-Architected Framework

[47] GCP Load Balancers Overview

[48] Azure Load Balancers Overview

[49] AWS Load Balancers Overview

[50] Operational Excellence Pillar

[51] Security Pillar

[52] Reliability Pillar

[53] Performance Efficiency Pillar

[54] Cost Optimization Pillar

[55] Sustainability Pillar

[56] Autoscaling compared: Azure vs GCP vs AWS

[57] Cloud security comparison: AWS vs. Azure vs. GCP

[58] AWS vs Azure vs GCP: Comparing The Big 3 Cloud Platforms

[59] Services provided by Azure

[60] Audit logging and monitoring overview

[61] Application performance monitoring using Dynatrace

[62] 5 reasons to build multi-region application architecture

[63] What is HIPAA?

[64] What is a Web Application Firewall?

[65] Understanding Disaster Recovery in the Cloud

[66] Availability

[67] What is Cloud Scalability?

[68] Cloud-cost monitoring

[69] Understanding Cloud storage