ECE547/CSC547 - Cloud Architecture Project - Fall 2023

Movie Information System in Cloud

Rohan Kausik Nandula (rnandul)

Sahil Bhalchandra Purohit (spurohi2)

November 24, 2023

We, Rohan Kausik Nandula and Sahil Bhalchandra Purohit, 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.

Index

1 Introduction

1.1 Motivation

Engaging in a cloud architecture project offers a unique opportunity to explore diverse cloud computing platforms like Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP). Through hands-on experience, we can understand the process of designing and deploying cloud-based applications, unraveling the complexities of this technology. This practical knowledge not only hones our technical skills but also fulfills the high demand for cloud computing expertise in the IT industry. Moreover, this project serves as a catalyst for sharpening our problem-solving and critical thinking abilities, encouraging creative architectural solutions that are both effective and economical.

1.2 Executive summary

In the ever-evolving landscape of Movie Information Systems, embracing cloud technology becomes more than a choice—it's a game-changer. Imagine a movie database that effortlessly adjusts to the pulse of user interest, seamlessly accommodating the buzz of blockbuster releases and the excitement of awards seasons. This adaptability is at the heart of cloud computing, offering a level of scalability that ensures the system effortlessly keeps pace with user demands. Not just a one-size-fits-all solution, cloud technology is remarkably flexible, allowing businesses to sculpt their information systems precisely as they envision.

Cost considerations, often a concern in technological ventures, find a reassuring answer in cloud computing. Gone are the days of hefty upfront investments; instead, a pay-as-you-go model means businesses only pay for the resources they use, offering substantial savings. Security, paramount in handling sensitive movie data, is robustly addressed. Advanced encryption, stringent access controls, and vigilant intrusion detection systems form an impenetrable shield around your information. To bring this technology to life, we plan to harness an array of AWS services. These services help in hosting the MIS Application and also allows for seamless Networking services, ensuring every component of the system functions in harmony. Deep insights into user behavior, essential for informed decisions, are unlocked through powerful analytics services. In essence, cloud computing isn't just a tool; it's the foundation upon which innovative, reliable, and secure movie information systems can thrive.

2 Problem Description

2.1 The problem

The overarching challenge in designing a movie information system is to create an efficient and user-friendly platform that, upon receiving the title of a movie as input, retrieves and displays essential details about that movie. These details typically include the movie's title, release date, director, genre, a list of actors, and the movie's IMDB rating.

To address this problem effectively, the system must be capable of interacting with a database or utilizing an external API to fetch the relevant information. This process involves ensuring that the user's input is processed accurately and efficiently. It should handle a variety of inputs or movie titles, ensuring it can retrieve the correct movie details, while minimizing repetition or redundancies in the data presentation.

In essence, the design challenge revolves around creating a smooth and intuitive experience for users seeking movie information based on a title, making sure they receive accurate and comprehensive details without complications.

2.2 Business Requirements

BR.1: Ensure 99% uptime, providing uninterrupted access to users.
(Well-Architected Pillars: Operational Excellence, Reliability)
BR.2: Accommodate time-varying workloads and maintain performance stability during unexpected traffic surges.
(Well-Architected Pillars: Scalability, Performance Efficiency)
BR.3: Identify and authenticate tenants accurately for billing and content access purposes.
(Well-Architected Pillars: Security, Operational Excellence)
BR.4: Adhere to copyright regulations and protect personal and proprietary data as per industry compliance standards.
(Well-Architected Pillars: Security, Compliance)
BR.5: Optimize costs in cloud resource utilization without compromising system performance.
(Well-Architected Pillars: Cost Optimization, Operational Excellence)
BR.6: Design the architecture to avoid vendor lock-in and maintain business agility.
(Well-Architected Pillars: Reliability, Operational Excellence)

BR.7: Ensure system architecture is flexible enough to integrate with external services and capable of migrating to different cloud providers without excessive overhead.
(Well-Architected Pillars: Scalability, Performance Efficiency)
BR.8: Guarantee data availability and integrity for user requests and compliance purposes.
(Well-Architected Pillars: Reliability, Security)
BR.9: Implement monitoring tools to track system performance for data-driven optimization decisions.
(Well-Architected Pillars: Operational Excellence, Performance Efficiency)
BR.10: Achieve minimal latency in data retrieval to enhance user experience.
(Well-Architected Pillars: Operational Excellence, Performance Efficiency)
BR.11: Optimize cloud resource utilization.
(Well-Architected Pillars: Operational Excellence, Performance Efficiency)
BR.12: Identify and thwart insider threats.
(Well-Architected Pillars: Security)

2.3 Technical Requirements

TR1.1: Design for high availability to support the system's 99% uptime requirement.
TR1.2: Continuously monitor and automate failure detection and recovery to minimize downtime and enhance fault tolerance.
Justification: To achieve this uptime, the system must be resilient to failures and capable of automatic recovery without human intervention (TR1.1, TR1.2). High availability designs and automated processes are critical to achieving and maintaining the desired uptime.

TR2.1: Implement auto scaling of computing resources based on time-varying workloads.
TR2.2: Use traffic analysis tools to predict and prepare for traffic spikes.
Justification: Dynamic scaling (TR2.1) and predictive traffic management (TR2.2) ensure the system can adapt to workload changes quickly, maintaining stability even during unanticipated high-traffic events. This is essential to meeting user expectations and service levels.

TR3.1: Incorporate a tenant management system that supports secure user authentication.
TR3.2: Employ a billing system that accurately reflects tenant usage and access.
TR3.3: Isolate the tenants properly.
Justification: These technical requirements align with the BR3 by ensuring that a robust tenant identification system is in place, essential for authentication and access to subscribed content. It also ensures that tenants information is tracked accurately and therefore, accurate billing for users can be provided which is one of the important business goals. Also, proper tenant isolation ensures that each tenant's data privacy, security, and operational independence within a shared environment.

TR4.1: Encrypt data at rest and in transit in accordance with industry standards.
TR4.2: Implement access controls and compliance checks for handling personal and copyrighted information.
Justification: Encryption and access controls (TR4.1, TR4.2) are the technical mechanisms by which the system will protect sensitive data in compliance with legal and regulatory standards. These mechanisms are essential for mitigating risks of data breaches and intellectual property infringement.

TR5.1: Continuously monitor cloud resource usage to identify optimization opportunities.
TR5.2: Utilize cost management tools to track and reduce expenditures.
Justification: Resource usage monitoring and cost management (TR5.1, TR5.2) provide the data and controls needed to balance cost with performance. By actively managing these aspects, the system can avoid unnecessary expenses while maintaining the performance standards required by users.

TR6.1: Use cloud-agnostic infrastructure as code tools for system deployment.
TR6.2: Design loosely coupled system components to facilitate interoperability and migration.
Justification: Cloud-agnostic tooling and design principles (TR6.1, TR6.2) are the technical underpinnings that allow the business to avoid vendor lock-in and maintain the flexibility to switch cloud providers or integrate multiple services as needed for strategic agility.

TR7.1: Implement APIs for easy integration with external services.
TR7.2: Design modular components that can be migrated or replaced without extensive system reconfiguration.
Justification: APIs and modular design (TR7.1, TR7.2) enable technical interoperability and flexibility, which are imperative for the business to integrate with various external services and to migrate between cloud environments with minimal disruption and cost.

TR8.1: Utilize a distributed database architecture for redundancy and quick data access.
TR8.2: Implement regular data backups and employ version control mechanisms.
Justification: A distributed database and data backup strategies (TR8.1, TR8.2) are technical solutions that directly support the availability and integrity of data, which is a business necessity for both operational functionality and regulatory compliance.

TR9.1: Deploy a monitoring system to gather performance metrics in real-time.
TR9.2: Analyze performance data to identify trends and areas for improvement.
Justification: Real-time monitoring and performance data analysis (TR9.1, TR9.2) provide the necessary insights for ongoing optimization, which is crucial for maintaining system performance at a level that meets or exceeds business and customer expectations.

TR10.1: Optimize content delivery networks to reduce data retrieval times.
TR10.2: Implement edge computing where feasible to minimize response times.
Justification: Optimizing content delivery networks and utilizing edge computing (TR10.1, TR10.2) are technical strategies aimed directly at reducing latency, which is essential for providing the fast and responsive user experience that the business demands.

TR11.1: Automate resource scaling based on certain triggers to efficient resource usage.
TR11.2: Right-size resources regularly to fit the workload requirements without over-provisioning.
Justification: Automated scaling and right-sizing (TR11.1, TR11.2) are technical methodologies that ensure resources are matched to need without over-provisioning, directly impacting cost efficiency and minimizing waste, which are key business objectives.

TR12.1: Integrate threat detection and analytics tools to monitor for unusual activities.
TR12.2: Enforce strict access controls and conduct regular security audits.
Justification: Threat detection and security audits (TR12.1, TR12.2) are technical safeguards essential for identifying potential insider threats, which is a critical aspect of maintaining the overall integrity and trustworthiness of the system from a business standpoint.

2.4 Tradeoffs

Trade-off 1: System Uptime vs. Maintenance Flexibility
BR.1: Ensure 99% uptime, providing uninterrupted access to users.
TR1.2: Continuously monitor and automate failure detection and recovery to minimize downtime.
While automated failure detection and recovery (TR1.2) support the high uptime (BR.1), they also imply that the system may go through automatic maintenance processes which could momentarily impact uptime. The trade-off is maintaining high availability versus the potential downtime caused by automated maintenance activities.

Trade-off 2: Cost Optimization vs. Performance Stability
BR.5: Optimize costs in cloud resource utilization without compromising system performance.
TR2.1: Implement auto-scaling to adjust resources in response to time-varying loads.
Auto-scaling (TR2.1) helps manage performance during varying workloads, which could incur additional costs, potentially conflicting with the goal of cost optimization (BR.5). The trade-off is between dynamically scaling resources to handle load (which can be costlier) and maintaining a lean operation to minimize costs.

Trade-off 3: Cost Optimization vs. Data Integrity and Availability
BR.5: Optimize costs in cloud resource utilization without compromising system performance.

BR.8: Guarantee data availability and integrity for user requests and compliance purposes.
TR8.1: Utilize a distributed database architecture for redundancy and quick data access.
Implementing a distributed database architecture (TR8.1) ensures data integrity and availability (BR.8) but could lead to higher costs, which might conflict with the objective of cost optimization (BR.5). The trade-off is managing the increased expense of a distributed database while still aiming to minimize overall costs.

Trade-off 4: Security vs. User Accessibility and Experience
BR.3: Identify and authenticate tenants accurately for billing and content access purposes.
TR12.1: Integrate threat detection and analytics tools to monitor for unusual activities.
Enhanced security measures (TR12.1) may introduce additional authentication steps or checks, potentially complicating the tenant authentication process (BR.3) and affecting user experience. The trade-off is between stringent security protocols and maintaining a user-friendly, accessible system.

3 Provider Selection

3.1 Criteria for choosing providers:

When selecting a cloud service provider (CSP) for moving our movie information system into their respective cloud environment, it's crucial to evaluate specific criteria that align with the system's technical and business requirements. The following are the criteria we used to choose our provider:

1. **High Availability and Fault Tolerance**: A movie information system must be operational and accessible at all times to satisfy user expectations and generate revenue. Thus, it's important to choose a CSP that offers services with a proven track record of high availability. The CSP should have multiple geographically dispersed data centers to ensure service continuity in the event of an outage or disaster.
2. **Scalability and Performance**: Movie information systems often experience variable workloads, with spikes during movie releases and award seasons. The chosen CSP must provide scalable resources to handle these variations without degradation in performance. Auto-scaling services and performance optimization tools are critical for managing these dynamic workloads efficiently.
3. **Security and Compliance**: As the system will handle user authentication and potentially store personal data, a CSP with robust security measures is paramount. The provider should offer compliance with industry standards to safeguard against breaches and ensure data privacy. Encryption, access controls, and compliance certifications are non-negotiable to protect users and adhere to legal regulations.
4. **Cost Optimization**: Since cloud costs can escalate with increased usage, it's necessary to select a provider that offers comprehensive cost management tools. These tools should help monitor and control resource utilization, allowing for an optimized balance between cost and performance.
5. **Flexibility and Portability**: The cloud environment should not lock the movie information system into a single provider's ecosystem. The CSP should support cloud-agnostic tools and services that enable the system to remain flexible and portable, making it easier to adapt to future changes in technology or business strategy.
6. **Performance Monitoring and Optimization**: The chosen provider should offer advanced monitoring tools that deliver insights into system performance. This data is essential for making informed decisions about optimizing the system, ensuring that the user experience remains high-quality and responsive.

7. **Latency Optimization**: Low latency is critical for a positive user experience, especially when delivering rich media content. The CSP must have a strong network infrastructure with edge locations close to the user base to reduce data transfer times and improve the speed of content delivery.
8. **Options**: It is always a good idea to have a variety of services available, which are currently unknown to us. This is because we may not be able to anticipate what additional services will be required during the process of building the architecture. Our focus is to have additional options to allow app interaction with 3rd party APIs.

3.2 Provider Comparison:

We have decided to rank the CSPs based on the compatibility of each criteria with our application.

Criteria	**AWS**	**Azure**	**GCP**	**Justification**
	Rank	**Rank**	**Rank**	
High Availability and Fault Tolerance	1	2	3	We chose AWS, Azure and GCP for comparison because of its proven track record in delivering high availability across more regions globally [4]. AWS's expansive infrastructure and services like Amazon S3 and EC2 offer a mature environment for achieving high uptime and fault tolerance. Azure follows closely with a strong set of similar features, but AWS leads in the breadth of options and proven use cases. GCP, while reliable, doesn't yet match the scale and global reach of AWS's offering.
Scalability and Performance	1	3	2	AWS takes the lead again with its Auto Scaling and Elastic Load Balancing services, which are more widely adopted than similar offerings from Azure (Azure Autoscale) and GCP (Google Compute Engine autoscaler). Azure lags slightly because its services are seen as less flexible by some users, and GCP, while offering competitive products, has a smaller user base and therefore less community-driven scalability

				solutions.[5][6]
Security and Compliance	1	2	3	AWS offers an extensive array of security features and compliance certifications that better Azure and GCP. AWS services such as Amazon Inspector for security assessments and AWS Shield for DDoS protection are considered industry leaders. Azure has robust security offerings as well, but AWS's experience and scale give it an edge. GCP is growing in this area but is often perceived as playing catch-up to the other two.[7]
Cost Optimization	1	2	1	Here, AWS and GCP share the 1st rank as both the CSPs offer per-second billing with a large variety of instances. Azure also offers per-second billing but this model isn't available for all the instances. [8]
Flexibility and Portability	2	3	1	GCP ranks highest for flexibility and portability mainly due to its deep investments in containerization services like Kubernetes and hybrid cloud platforms. AWS is also strong in this area with services like AWS Fargate and AWS Outposts, but GCP's open-source orientation and interoperability initiatives give it a slight edge. Azure is less flexible, especially in hybrid cloud scenarios, compared to GCP.[6]
Performance Monitoring and Optimization	1	2	3	AWS's extensive monitoring tools, such as AWS CloudTrail, Amazon CloudWatch, and AWS X-Ray, provide deeper insights and more advanced analytics compared to Azure Monitor and GCP's Stackdriver. AWS's services are also known for allowing more fine-tuned optimization, hence they are ranked the highest.[6]
Latency Optimization	1	3	2	AWS CloudFront, combined with AWS's more extensive global infrastructure, provides lower latency through its CDN compared to Azure's CDN and GCP's

				Cloud CDN. While all three offer edge computing solutions, AWS's established network of edge locations and partnerships is currently unrivaled. Azure and GCP are expanding their footprint but have not yet reached AWS's level of expanse in this field.
Options	1	3	2	When evaluating the services provided by the 'big three' for our app that requires interacting efficiently with third-party APIs, AWS ranks at the top. This is due to its mature and robust offerings, notably the well-integrated API Gateway and Lambda services, which are supported by a vast user base and supplemented by Step Functions for managing complex processes. Microsoft Azure comes in close with its API Management and Logic Apps, offering a potent mix for API integrations and a particularly seamless experience for enterprises already entrenched in Microsoft's ecosystem. Google Cloud Platform, while offering comparable services such as Cloud Endpoints and Cloud Functions, is perceived to be less mature in the enterprise arena compared to its competitors. [9]

3.3 The final selection:

Based on the criteria established and the rankings provided, Amazon Web Services (AWS) is the clear winner as the CSP for this project with majority of Rank 1s obtained. AWS's comprehensive suite of services aligns closely with our TRs, ensuring that we can meet our business and technical objectives.

3.3.1 The list of services offered by the winner

1. **Amazon EC2 & Auto Scaling**: Amazon EC2 provides scalable compute capacity which, when combined with AWS Auto Scaling, ensures resources adjust dynamically to meet the system's demand (TR2.1, TR11.1), aligning with our goals for efficiency and high availability (TR1.1). [10][11]
2. **AWS Elastic Load Balancing (ELB)**: ELB distributes incoming application traffic, which enhances fault tolerance and uptime (TR1.2), critical to achieving our system's 99% uptime requirement.[13]
3. **Amazon CloudWatch**: This monitoring service provides real-time insights into resource utilization (TR5.1), which is crucial for cost optimization (TR5.2) and identifying performance improvement opportunities (TR9.2).[12]
4. **Amazon RDS**: These database services offer distributed database architecture (TR8.1), enabling quick data access and assisting with regular data backups and version control (TR8.2).[14]
5. **AWS Identity and Access Management (IAM)**: IAM supports secure user authentication (TR3.1) and enforces strict access controls (TR12.2), which are foundational to maintaining system integrity and billing accuracy (TR3.2).[15]
6. **Amazon S3 & Versioning**: S3 provides secure and durable object storage, with versioning capabilities that contribute to data backup strategies (TR8.2) and support encryption for data at rest (TR4.1).[16]
7. **AWS CloudFormation**: It allows for infrastructure as code, facilitating deployment across cloud environments (TR6.1) and supporting modular and loosely coupled designs (TR6.2, TR7.2).[17]
8. **AWS Key Management Service (KMS)**: KMS simplifies encryption key management (TR4.1), which is vital for data security and compliance with industry standards.[18]
9. **Amazon GuardDuty**: GuardDuty offers threat detection to monitor for unauthorized and malicious activities (TR12.1), aligning with our robust security framework.[19]
10. **AWS API Gateway**: API Gateway facilitates integration with external services (TR7.1), supporting our modular system design and ensuring smooth third-party interactions.[20]
11. **Amazon VPC**: VPC enables us to provision a logically isolated section of AWS where we can launch resources, ensuring network security and facilitating edge computing implementations (TR10.2).[21]
12. **AWS Lambda**: Lambda allows running code without managing servers and autoscaling (TR11.1), which is essential for handling variable workloads and spikes in traffic (TR2.2).[22]
13. **Amazon CloudFront & Edge Locations**: This global content delivery network service accelerates delivery of data, videos, applications, and APIs to users worldwide (TR10.1), reducing retrieval times and improving overall user experience.[23]
14. **AWS Cost and Usage Report**: These reports aid in tracking cloud expenditure (TR5.2), allowing for effective cost management.[24]

15. **AWS Elastic Disaster Recovery**: Elastic Disaster Recovery minimizes downtime (TR1.2) and ensures business continuity, aligning with our high availability goals.[25]

4 The first design draft

In this section, we will provide the first draft of our proposed design for the movie information system based on the selected cloud service provider, Amazon Web Services (AWS). This initial draft will outline the basic building blocks of our design and provide informal validation of how the design will achieve the Technical Requirements (TRs) identified in Section 2.3.

4.1 The basic building blocks of the design:

Our design relies on a set of key services and components provided by Amazon Web Services (AWS) to meet the technical requirements (TRs) established. These building blocks form the core of our architecture and ensure the fulfillment of specified TRs:

1. ***Amazon IAM (Identity and Access Management):*** AWS IAM allows us to control access to AWS services and resources securely. It works by creating and managing user identities, groups, roles, and policies. IAM policies define what actions are allowed or denied for each user or resource. It enforces strict access controls by granting or denying permissions based on policies and addresses the "Security" Pillar of AWS WAF.[15]
2. ***Amazon CloudWatch:*** Amazon CloudWatch is a monitoring service that collects and tracks metrics and logs from AWS resources and applications. It works by capturing data and generating actionable insights through customizable alarms and dashboards. CloudWatch helps monitor performance of resources, detect anomalies, and trigger automated actions based on defined thresholds.[12]
3. ***AWS Elastic Load Balancing (ELB):*** To implement load balancer, we have two options ALB and ELB. ELB works on the Transport layer while ALB works on the Application layer. Since our application is monolithic and doesn't have different services as different processes, therefore we don't require ALB. ELB automatically distributes incoming traffic across multiple Amazon EC2 instances or other targets in different Availability Zones. It works by continuously monitoring the health of the registered targets and directing traffic to healthy instances. ELB enhances fault tolerance by distributing traffic and automatically routing it away from unhealthy instances. It addresses the "Reliability" Pillar of the AWS WAF.[13]
4. ***Amazon RDS (Relational Database Service):*** Amazon RDS simplifies database management by providing a managed relational database service. It works by setting up, operating, and scaling a relational database in the cloud. RDS supports various database engines and offers features like automated backups, software patching, and high availability through Multi-AZ deployments.[14]

5. ***Amazon S3 (Simple Storage Service):*** Amazon S3 is an object storage service that allows us to store and retrieve data. It works by organizing data into "buckets" and "objects" (files). S3 provides durable and scalable storage with built-in redundancy. It supports data encryption at rest and offers versioning to track changes to objects over time.[16]
6. ***AWS CloudFormation:*** AWS CloudFormation is a service for defining and provisioning infrastructure as code. It works by using templates (JSON or YAML) to describe AWS resources and their relationships. CloudFormation then automates the provisioning and management of those resources, ensuring consistency and reproducibility. It addresses the "Operational Excellence" Pillar of the AWS WAF.[17]
7. ***Amazon GuardDuty:*** Amazon GuardDuty is a threat detection service that continuously monitors for malicious activity in an AWS environment. It works by analyzing logs and network traffic to detect anomalies and known threats. GuardDuty provides alerts and findings to help us respond to security incidents. It addresses the "Security" Pillar of AWS WAF[19]
8. ***AWS API Gateway:*** AWS API Gateway is a fully managed service for creating, publishing, and securing APIs. It works by allowing us to define and deploy APIs, configure endpoints, and manage access control and authorization. API Gateway enables integration with backend services and supports API versioning.[20]
9. ***AWS Lambda:*** AWS Lambda is a serverless compute service that runs code in response to events. It works by executing functions in response to triggers, such as API Gateway requests or data changes in Amazon S3. Lambda scales automatically and charges you only for the compute time used.[22]
10. ***Amazon CloudFront:*** Amazon CloudFront is a content delivery network (CDN) service that accelerates the delivery of content to end-users. It works by caching and distributing content from edge locations located worldwide. CloudFront reduces latency by serving content from the nearest edge location to the user. It addresses the "Performance Efficiency" pillar of the AWS WAF.[23]
11. ***AWS Cost and Usage Report:*** AWS Cost and Usage Report provides detailed billing and usage data. It works by collecting information on resource usage and associated costs. The report helps organizations analyze and manage their AWS spending by providing insights into resource consumption. It addresses the "Cost Optimization" pillar of the AWS WAF.[24]
12. ***AWS Elastic Disaster Recovery:*** AWS Elastic Disaster Recovery enables automated disaster recovery solutions. It works by replicating data and resources to a secondary AWS region. In case of a disaster or outage, resources can be quickly activated in the secondary region to minimize downtime.[25]
13. **AWS EC2:** AWS EC2 provides capability to run applications with scalable infrastructure.[10]

4.2 Top-level, informal validation of the design:

Following TRs are established in Section 2.3 and Section 4.1 describes the design building blocks. Here we establish relationships between TRs and building blocks and provide reasons for how that service will satisfy one or more Technical Requirements (TRs):

Tenant Identification TR (TR3.1):

- **Service:** Amazon IAM (Identity and Access Management)
- **Reason:** Amazon IAM will serve as the central component for managing user authentication and access control. It enforces strict access controls and supports secure user authentication (TR3.1). IAM policies define permissions for user identities, ensuring that tenant-specific access is controlled and accurately directed.

Monitoring TRs (TR9.1, TR 5.1):

- **Service:** Amazon CloudWatch
- **Reason:** Amazon CloudWatch provides real-time monitoring of system performance and resource utilization. It captures and tracks metrics, which are essential for gathering performance metrics in real-time (TR9.1). CloudWatch's metrics and alarms ensure that the system remains observable and responsive.
- **Service**: AWS Cost and Usage Report
- **Reason**: This service addresses TR5.1 by monitoring cloud resource usage, identifying optimization opportunities, and providing insights for effective cost management.

High Availability and Fault Tolerance TRs (TR1.1, TR1.2):

- **Service:** AWS Elastic Load Balancing (ELB)
- **Reason:** ELB automatically distributes incoming application traffic across multiple instances in different Availability Zones (AZs). This enhances fault tolerance (TR1.2) by ensuring high availability and minimizing downtime. ELB continuously monitors the health of instances and routes traffic away from unhealthy ones, contributing to system reliability.

Data Encryption and Security TR (TR4.1, TR12.2):

- **Service:** AWS Relational Database Service (RDS)
- **Reason:** AWS RDS offers features like encryption at rest using AWS KMS-managed keys, which directly aligns with the data encryption requirement (TR4.1). It allows for encryption of data in transit with SSL/TLS, adding another layer of security as data moves between the database and the application. With the use of security groups and network access controls, AWS RDS enforces strict access controls, contributing to the protection against unauthorized access (TR12.2).

Scalability and Auto-Scaling TR (TR2.1, TR11.1):

Service: AWS AutoScaling , AWS Lambda

- **Reason:** AWS Lambda enables serverless computing for non-continuous tasks(TR11.1) and AWS AutoScaling enables auto-scaling of instances (TR2.1). Autoscaling continuously monitors time-variable workloads to scale resources up and down. Lambda functions, on the other hand, can be triggered in response to various non-continuous events, aligning with auto-scaling requirements to handle both continuous and non-continuous variations in load.

Performance Optimization TR (TR10.1):

- **Service:** Amazon CloudFront
- **Reason:** Amazon CloudFront is a content delivery network (CDN) service that optimizes content delivery by reducing data retrieval times (TR10.1). It caches and distributes content from edge locations worldwide, ensuring low latency and improved user experience for delivering media content.

Cost optimization TR (TR5.2):

- **Service:** AWS Cost and Usage Report
- **Reason:** AWS Cost and Usage Report provides insights into cloud expenditure (TR5.2), allowing for effective cost management and optimization. By analyzing resource usage and associated costs, organizations can control expenses and optimize their AWS spending.

Disaster Recovery and High Availability TR (TR1.2):

- **Service:** AWS Elastic Disaster Recovery
- **Reason:** AWS Elastic Disaster Recovery minimizes downtime (TR1.2) and ensures business continuity by replicating data and resources to a secondary AWS region. In case of a disaster or outage, resources can be quickly activated in the secondary region, enhancing high availability and fault tolerance.

API Integration TR (TR7.1):

- **Service:** AWS API Gateway
- **Reason:** AWS API Gateway facilitates easy integration with external services (TR7.1). It allows the design of APIs, configures endpoints, and manages access control and authorization, ensuring smooth third-party interactions and modular system components.

Data Backup and Version Control TR (TR8.2):

- **Service:** Amazon S3 (Simple Storage Service)

- **Reason:** Amazon S3 supports data backup strategies (TR8.2) by providing versioning capabilities. It allows tracking changes to objects over time, ensuring data integrity and backup management.

In our proposed system design, we see two important, and potentially conflicting Technical Requirements - TR1.1, TR1.2 (Design for high availability and fault tolerance) and TR5.2 (Utilize cost management tools to track and reduce expenditures). Our analysis as to how their implementations can be conflicted is as follows:

TR1.1, TR1.2: High Availability and Fault Tolerance

- **Implementation**: Services like AWS Elastic Load Balancing (ELB), Amazon RDS, and AWS Elastic Disaster Recovery are used.
- **Potential Conflict**: These services ensure high availability and fault tolerance through redundant resources, multi-AZ deployments, and disaster recovery strategies. However, "such redundancy and high-availability configurations often lead to higher operational costs due to the need for more resources and more complex management." (ChatGPT)

TR5.2: Cost Optimization

- **Implementation**: AWS Cost and Usage Report is utilized.
- **Potential Conflict**: This tool helps track and manage expenditures, encouraging cost savings and optimization. Implementing cost-saving measures might lead to reduced redundancy or downsizing of resources, which could negatively impact the system's availability and fault tolerance.

How are we planning to resolve this potential conflict:

- The challenge lies in finding a balance where the system remains highly available and fault-tolerant without incurring excessive costs. This may involve strategic decisions like choosing which services require high redundancy and which can be scaled down without significantly impacting system performance.
- Cost-Efficient High Availability Strategies: One approach could be using cost-effective high availability strategies such as spot instances or reserved instances for non-critical workloads, or selectively applying multi-AZ deployments where absolutely necessary.

4.3 Action items and rough timeline: SKIPPED

5 The second design

5.1 Use of the Well-Architected framework

In the design and implementation of our Movie Information System, we applied the AWS Well-Architected Framework to ensure that our architecture adheres to AWS's best practices. The framework consists of six pillars, which include Operational Excellence, Security, Reliability, Performance Efficiency, Cost Optimization, and Sustainability. Each pillar represents a set of core strategies and design principles that guide our approach to creating a robust, secure, efficient, and cost-effective system.[1]

Operational Excellence: This pillar focuses on running and monitoring systems to deliver business value and continually improving processes and procedures. Key practices include automation of manual tasks, responding to events to ensure business continuity, and defining standards to manage daily operations [26]. For our project, we have focused on creating a system that supports our operational goals and enables us to run workloads effectively. We've implemented monitoring and automation to gain operational insights and foster continuous improvement. By leveraging AWS services like CloudWatch and CloudFormation, we can automate deployments and monitor our systems to ensure they are functioning correctly and efficiently.

Security: Security encompasses the protection of information and systems. It involves confidentiality, integrity, and availability of data, risk assessment and mitigation strategies. This involves identity and access management, data encryption, and implementing security checks and balances [26]. Emphasizing the protection of data, systems, and assets, we incorporated security best practices throughout our design. This involves encrypting data at rest using AWS KMS, securing data in transit with SSL/TLS, and employing IAM for fine-grained access control. Amazon GuardDuty is also used to enhance our security posture by continuously monitoring for threats and unauthorized behavior.

Reliability: Reliability means the ability of a system to recover from infrastructure or service disruptions, dynamically acquire computing resources to meet demand, and mitigate disruptions such as misconfigurations or transient network issues. It includes setup for auto-recovery, proactive failure management, and graceful degradation [26]. To ensure our workload performs its intended function correctly and consistently, we have designed for reliability. This includes implementing AWS services like Elastic Load Balancing and Auto Scaling to handle changes in demand, and Amazon RDS with Multi-AZ deployment for high availability and data durability.

Performance Efficiency: This pillar focuses on using computing resources efficiently. It involves selecting the right resource types and sizes based on workload requirements, monitoring performance, and making informed decisions to maintain efficiency as business needs evolve. It employs advanced technologies, global deployment to minimize latency, and performance tuning [26]. Our design is built to maintain efficiency as demand changes and technologies evolve. We use Amazon CloudFront for efficient content delivery and AWS Lambda for its serverless execution model, which enables us to run code without provisioning or managing servers, thereby optimizing computing resources.

Cost Optimization: Cost Optimization is about avoiding unnecessary costs. It includes understanding and controlling where money is being spent, selecting the most appropriate and right number of resource types, analyzing spend over time, and scaling to meet business needs without overspending. It encompasses cost-effective resource selection, matching supply with demand, and optimizing over time [26]. We aim to deliver business value at the lowest price point. This is achieved by monitoring our usage with AWS Cost and Usage Report, selecting appropriate pricing models like Reserved Instances, and employing auto-scaling to ensure we're using resources efficiently.

Sustainability: Sustainability in the AWS Well-Architected Framework refers to the efficient use of resources to achieve the desired business outcomes with the least environmental impact. It involves optimizing the usage of resources to achieve higher performance, using serverless architectures to reduce resource wastage, and designing energy-efficient applications [26]. In recognition of the increasing importance of energy consumption, we are committed to improving the sustainability impacts of our workload. By optimizing resource utilization through serverless architectures and efficient data storage and transfer methods, we aim to reduce the energy footprint of our operations.

5.2 Discussion of pillars

For our Movie Information System, we have prioritized three AWS Well-Architected Framework pillars that are particularly aligned with the needs and goals of our project: Operational Excellence, Reliability, and Security [26]. Here’s a summary of each pillar based on the framework:

Operational Excellence Pillar:
Operational Excellence involves the ability to run and monitor systems in a way that delivers business value and continuously improves processes and procedures. Our project reflects this pillar through:

- Process Automation: Implementing automated deployments and management processes to ensure consistent and reliable operations.
- Monitoring and Logging: Actively monitoring system performance to quickly address issues and maintain operational health.
- Iterative Improvement: Applying lessons learned from operational experiences to make iterative improvements, enhancing the system's efficiency and effectiveness.

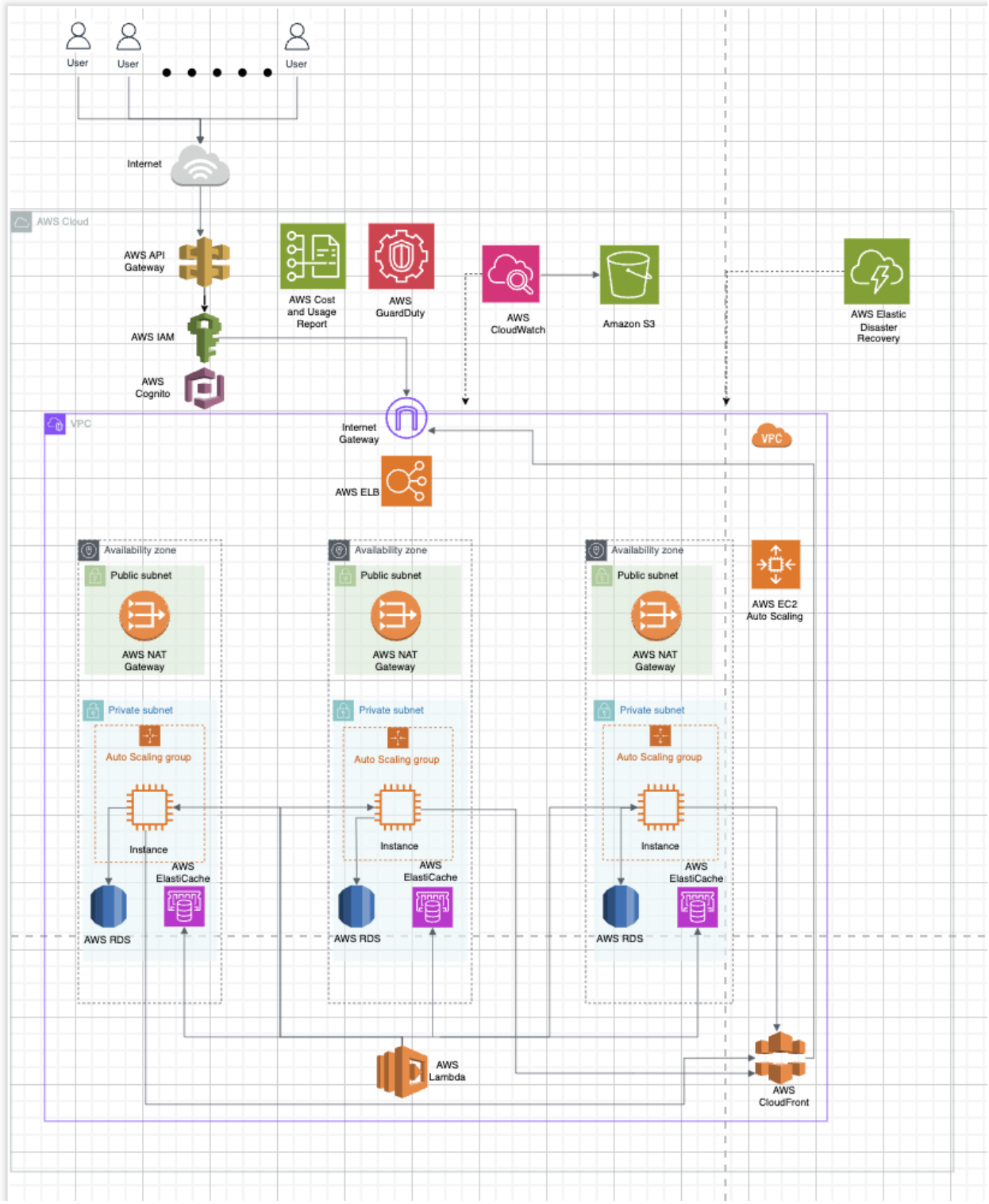
Reliability Pillar:
The Reliability pillar focuses on ensuring that a system can perform its intended functions correctly and consistently under a defined set of conditions. This is achieved in our project by:

- Automatic Recovery: Setting up systems to monitor for failures and automatically initiate recovery procedures to maintain service continuity.
- Regular Testing: Conducting regular tests to simulate failures and validate the effectiveness of recovery procedures.
- Scalability: Designing the system to scale horizontally, spreading loads across multiple resources to minimize the impact of any single failure.
- Capacity Planning: Monitoring system utilization to accurately scale resources in response to varying demands and avoid over-provisioning or under-provisioning.
- Change Management: Using automated processes to manage changes in the infrastructure to ensure they are systematically implemented and reviewed.

Security Pillar:
Security is about protecting data, systems, and assets while taking advantage of cloud technologies to strengthen the security posture. In our system, we incorporate this pillar through:

- Identity Foundation: Implementing strong identity and access management practices to control access to AWS resources.
- Traceability: Ensuring that all actions and changes within the environment are logged and monitored in real-time to enable swift action.
- Layered Approach: Applying multiple security controls across all layers of the system architecture.
- Automated Security Practices: Automating security best practices to quickly and cost-effectively scale security measures.
- Data Protection: Classifying and encrypting data at rest and in transit according to its sensitivity level and implementing appropriate access controls.
- Security Event Preparedness: Establishing and practicing incident response protocols to be well-prepared for potential security events.

5.3 Use of Cloud formation Diagrams



Components Involved:

1. User Request: Users interact with the system via the internet, initiating requests for service or content.
2. AWS API Gateway: Requests first encounter the API Gateway, which acts as a single entry point for API calls, providing an interface for various AWS services and performing tasks such as traffic management, authorization and access control, and monitoring.
3. AWS IAM and AWS Cognito: AWS Cognito helps with the authentication of the different types of tenants. The Identity and Access Management (IAM) system provides secure access control by validating user identities, ensuring proper permissions for resource access. This helps achieve proper tenant authentication and authorization.
4. VPC with Public and Private Subnets: Once the safe requests are filtered by the IAM, they enter the VPC environment.
    a. Public Subnet: Contains resources accessible from the internet - NAT Gateway.
    b. Private Subnet: Contains the core resources of the application that should not be directly accessible from the internet, like EC2 instances and RDS databases.
5. AWS ELB: The Elastic Load Balancer distributes incoming traffic across multiple targets - our EC2 instances - in different Availability Zones, which increases the fault tolerance of the application.
6. AWS EC2 Instances: Virtual servers in the Cloud where our Movie Information System application runs.
7. Auto Scaling Groups: Maintain application availability and allow the number of EC2 instances to scale up or down automatically according to conditions defined.
8. AWS RDS: The managed database service hosts the application data with automated backups, patch management, and encryption. Amazon RDS service helps store all the queries and results of the user requests. This replication of data across multiple availability zones allows for greater availability. We have utilized S3 services for retrieving the model version required for the computation of the response from the application.
9. AWS Elasticache: It helps with quick memory caching and retrieval to speed up computations for our application using fast, managed, in-memory caches.
10. AWS Lambda: We have implemented Lambda as a single, isolated service, which is used for specific, non-continuous tasks, such as handling S3 events or CloudFront requests. Our use of Lambda is to run functions in response to specific triggers rather than manage traffic, which ELB takes care of.
11. AWS CloudFront: A Content Delivery Network (CDN) that caches content at edge locations close to users to minimize latency to not more than 40-60ms[27].
12. Monitoring and Reporting Services:
    a. AWS Cost and Usage Report: Collects and reports on AWS usage for cost analysis and optimization, upholding our Monitoring TR.

b. AWS GuardDuty: Offers threat detection to protect AWS accounts and workloads, enabling our Security TR.
c. AWS CloudWatch: Monitors AWS resources and applications, collecting and tracking metrics and log files, which again uses S3 service to store the necessary metrics including response time, and downtime as a backup to help achieve our reliability and availability TRs.

13. Amazon S3: AWS Cloudwatch uses AWS S3 service to store the necessary metrics including response time, and downtime as a backup to help achieve our reliability and availability TRs. Although, S3 has other minor use cases depicted in some descriptions, this use case has higher priority.
14. AWS Elastic Disaster Recovery: Ensures business continuity by replicating data and applications across different AWS regions to minimize downtime during disruptions.
15. The infrastructure, managed via AWS CloudFormation, uses code for setup. It constantly monitors for changes and offers detailed feedback, effectively preventing conflicting alterations.

5.4 Validation of the design

Our movie information system is architected upon a cloud-based infrastructure that grants us the flexibility to deploy virtual machines effectively, via AWS. The responsibility for the upkeep of the underlying physical resources—servers, storage, and networking—lies with our cloud service provider, thereby allowing us to focus on managing and optimizing the layers of infrastructure we build atop these foundational components[28].

Central to our design is the principle of on-demand elasticity/scalability, which is facilitated through the rapid and automated provisioning of computing resources tailored to meet the fluctuating demands of our service. This elastic scaling capability ensures that we provision additional instances when user traffic spikes and similarly scale down when the demand subsides. This dynamic scalability is a pivotal feature of our system, not only promoting operational efficiency but also underpinning our commitment to cost optimization—one of our core business requirements.

In the sections that follow, we will delve into how this design enables us to meet most subset of TRs. We will also critically assess our architecture, identifying any potential gaps in fully meeting our technical requirements and discussing strategies to bridge these gaps for the movie information system.

Tenant Identification TR

1. Tenant Identification and Billing (TR 3.1, TR 3.2): In the context of a movie information system, there are typically 3 primary tenant groups that require access to the system with distinct permissions and capabilities:
   Viewers: The first set of tenants consists of viewers or users who access the system to browse, review, and interact with movie information. They need to be able to search for movies, read descriptions, watch trailers, and perhaps rate or review the films.
   Content Providers: The second set may be content contributors who could be studio representatives, critics, or authorized personnel responsible for uploading new movie information, updating existing entries, and ensuring that the content is accurate and up-to-date.
   Administrators: They oversee the entire operation, ensuring that the system is functioning optimally and securely.
   For the purpose of identification and management of these tenant groups, AWS Cognito is utilized to create user profiles that distinguish tenants based on their roles within the system. Once authenticated, each tenant's activity is tagged with Tenant Group IDs and Tenant Individual IDs, which can be used by the AWS IAM to enforce the permissions and roles that govern access to resources. IAM's policies are configured to prevent tenants from crossing into areas outside their purview. For instance, IAM roles for viewers would restrict write access to the database, limiting them to read-only actions. (ChatGPT)
   In our architecture, AWS IAM is strategically positioned before the VPC's entry point. This placement ensures that tenant identification and authorization are performed before any interaction with core resources, enhancing the system's security posture. When a request is funneled through the AWS API Gateway, it is intercepted by IAM and Cognito for authentication and authorization based on the established tenant roles.
   Using AWS IAM in tandem with AWS Cognito for tenant identification and management is the best approach because it allows for secure, scalable, and granular access control that is essential for a diverse and dynamic movie information system. This combination of services ensures that users can interact with the system in a manner that is consistent with their roles, improving both the user experience and the system's security. [This helps uphold the **Security** Pillar of the AWS WAF]

Monitoring TRs

2. For our movie information system hosted within AWS, monitoring goes beyond simple oversight—it is an intricate component that touches upon operational efficiency and cost optimization. AWS CloudWatch and AWS Cost and Usage Report are instrumental services that fulfill these monitoring necessities.
   Monitoring with AWS CloudWatch (TR5.1, TR9.1, TR9.2, TR 2.2):

AWS CloudWatch monitors the core elements of our infrastructure, including AWS S3 for storage, AWS RDS for database services, AWS EC2 for compute power, and AWS ELB for load balancing. Through CloudWatch, we track key metrics such as system downtime, response times, error rates, traffic loads, and CPU utilization. The real-time data garnered from these metrics is critical for performance analysis (TR9.1) and for identifying patterns indicative of system failures (TR9.2). The ability of CloudWatch to trigger alarms in the event of anomalies is vital for a swift corrective response. CloudWatch is configured to log detailed transactions and resource usage, enhancing the granularity of our monitoring efforts and ensuring the capture of actionable data. We establish threshold-based alarms that are fine-tuned to alert us of potential issues promptly while minimizing false alarms. All logs are securely stored in a dedicated S3 bucket situated outside the VPC, preserving data integrity and ensuring continuous availability.(ChatGPT)

Cost Monitoring with AWS Cost and Usage Report (TR5.2):

Financially, the AWS Cost and Usage Report service imparts a nuanced view into our cloud expenditure, mapping resource usage directly to costs (TR5.2). This insight allows us to ascertain the optimal use of resources and to identify areas where costs can be curtailed without sacrificing system performance. Configuring this service for detailed reporting on each service and resource, coupled with strategic tagging, facilitates precise tracking of expenses. Such a granular breakdown is pivotal for detecting inefficiencies and for honing in on cost optimization opportunities. [This also upholds the **Cost Optimization** Pillar of the AWS WAF]

Operational Excellence and Elasticity TRs

3. For our Movie Information System, the elasticity of the infrastructure is a key technical requirement to ensure that the system can handle variable workloads efficiently. AWS AutoScaling and AWS Lambda are services that address this requirement by providing on-demand resource allocation and event-driven computing power.

   Resource Scaling with AWS AutoScaling (TR11.1):

   AWS AutoScaling is crucial for adapting to the fluctuating demand inherent to a movie information system, which might experience surges in traffic during new movie releases or award seasons. This service automatically adjusts the number of Amazon EC2 instances in response to real-time traffic conditions, increasing instances during demand spikes to maintain performance, and scaling down during quieter periods to manage costs. The best configuration for AutoScaling in this scenario involves setting appropriate thresholds for scaling based on CPU utilization and network traffic metrics, ensuring that new instances are launched only when necessary, and terminated when demand wanes. These thresholds are determined by analyzing historical data and predicting traffic patterns related to movie industry events.

AWS Lambda (TR2.1):
Lambda plays a different but complementary role. It is employed for specific, non-continuous tasks that do not involve direct traffic management, such as processing S3 events when new content is uploaded or handling requests at the edge via CloudFront. This use of Lambda ensures that our system can respond immediately to event-driven needs without the overhead of managing server states. The optimal configuration for Lambda would set function-specific triggers and execution roles that grant necessary permissions to interact with other AWS services. For instance, a Lambda function triggered by S3 upload events would have execution roles that allow it to read from the S3 bucket and write to the RDS database if needed.

The combination of AWS AutoScaling and AWS Lambda satisfies the system's elasticity requirements by providing a dual approach to resource management. AutoScaling handles the macro-level adjustments needed to maintain service availability and performance, while Lambda addresses micro-level, event-driven tasks that contribute to the overall responsiveness and efficiency of the system. [Both these services also help uphold the **Performance Efficiency** Pillar of the AWS WAF]

Reliability TRs

4. In the architecture of our movie information system, ensuring reliability entails not only maintaining service availability but also safeguarding data integrity and continuity. AWS Elastic Disaster Recovery, AWS Elastic Load Balancer and Amazon S3 are two services that address these aspects of reliability.
   Downtime minimization with AWS Elastic Disaster Recovery (TR1.2):
   AWS Elastic Disaster Recovery plays a critical role in minimizing downtime and securing business continuity for the movie information system. This service is adept at replicating data and computational resources across primary and secondary AWS regions. In the event of a disaster or significant outage, such as a data center failure or regional service disruption, the system can swiftly failover to the secondary region, thus preserving high availability and enhancing fault tolerance. The configuration for Elastic Disaster Recovery would include regular, scheduled replication of all mission-critical data across multiple regions and automated failover processes. These configurations are vital as they ensure minimal Recovery Time Objectives (RTO) and Recovery Point Objectives (RPO), both of which are essential KPIs for disaster recovery and high availability. (ChatGPT) [Also upholds the **Performance Efficiency** Pillar of the AWS WAF]

   High availability and Fault Tolerance with AWS ELB (TR1.1, TR1.2):
   AWS ELB contributes to our reliability strategy by evenly distributing incoming traffic across multiple Amazon EC2 instances spread across different Availability Zones. ELB is

configured for automatic health checks, quickly detecting unresponsive or overwhelmed instances and rerouting traffic to healthy ones, thus ensuring that no single point of failure can compromise the system's uptime. This setup not only provides resilience against individual instance failures but also allows for seamless scaling during demand spikes, a common occurrence in the entertainment industry.

The ELB uses a dynamic, closed-loop algorithm to ensure that load balancing decisions adapt to changing conditions in real-time. The CPU that powers the ELB's decision-making process is inherently centralized within the ELB service. It manages the traffic distribution to various EC2 instances, making centralized computing resources both practical and efficient for our application. Given the nature of our movie information system, which likely experiences varying load with different user behaviors, a dynamic load balancer that supports a round-robin algorithm is suitable. This ensures that no single server bears too much burden at any time, which is critical for a system where user demand can be unpredictable. In the context of ELB, the round robin algorithm works by distributing incoming requests sequentially to the next server in line. It ensures that each EC2 instance within the Auto Scaling group receives an equal share of the workload over time.

For our system, load can be defined as the number of concurrent user requests or the CPU utilization of EC2 instances. These metrics provide a direct reflection of the demand on our system and are readily measurable through ELB and CloudWatch. Load should be measured at the EC2 instances to ensure that the computational workload is evenly distributed. Metrics including CPU utilization, request count per target, and network I/O are indicative of the load each instance is handling.

The goal for ELB in our movie information system is to maintain a balanced load across all instances to ensure high availability and fault tolerance. The criterion for load balancing is that no EC2 instance exceeds 70% CPU utilization to avoid performance degradation. When an instance approaches the set threshold, ELB's action would be to route new incoming requests to less burdened instances. In the case of an instance reaching its maximum capacity, ELB would stop routing traffic to it until it stabilizes.(ChatGPT)

ELB is an instance of the feedback loop where the health check data serves as input to determine load balancing actions. The continuous monitoring of instance health informs the dynamic routing decisions made by the ELB. Initially, we anticipated a tradeoff between scalability and maintaining a 99% uptime with AWS Elastic Load Balancing (ELB), as increasing the number of nodes typically complicates uptime assurance. However, considering the high cost of downtime in terms of revenue loss, we prioritized

uptime over cost. The assumption was that the expense incurred to ensure high availability would be less than the potential revenue lost due to system unavailability and latency. Consequently, we maintained our uptime commitment without compromise, leveraging ELB's scalability and high availability features to support our system's need for fast, continuous access. [This also upholds the **Operational Excellence** Pillar of the AWS WAF]

Data backups and version control with Amazon S3 (TR8.2):
Amazon S3 provides robust data backup capabilities, crucial for our movie information system's version control and data recovery strategies. By leveraging S3's versioning feature, every change made to an object is tracked and preserved over time, enabling rollback to previous versions if necessary and ensuring data integrity. For the movie information system, the best practice is to enable S3 versioning on all buckets containing valuable data such as movie metadata, user reviews, and system logs. Additionally, S3's lifecycle policies can be configured to automate the transition of older data to more cost-effective storage classes, such as S3 Glacier for long-term archiving, and to permanently delete outdated versions per compliance and data retention policies. (ChatGPT)

Performance BRs/TRs

5. To ensure our movie information system achieves optimal performance efficiency while also maintaining a high level of security, we have implemented a strategic combination of AWS services designed to minimize latency without straining our resource capabilities.

   Latency minimization with AWS CloudFront (TR10.1):
   CloudFront serves as a critical component in achieving our target latency of less than 50ms. By caching both static and dynamic content at edge locations, CloudFront ensures that user requests are served from the geographically nearest data point. This architecture reduces the number of network hops significantly, leading to much faster content delivery times. Key metrics for CloudFront that help us monitor and achieve our latency goals include Time-to-First-Byte (TTFB) and Cache Hit Ratio. These help us understand the effectiveness of our content caching and the speed at which content is being delivered to users. (ChatGPT)

   Amazon S3 Transfer Acceleration(BR10):
   By enabling Transfer Acceleration on S3, we leverage the CloudFront edge network for faster uploads to S3 buckets. This feature is essential for quickly updating content such as new movie listings or user-generated reviews. The First Byte Latency metric for S3 operations is crucial as it measures the time taken to begin retrieving a file, impacting the speed of content updates.

Frequently accessed data retrieval time reduction using AWS Elasticache (BR10):
Elasticache significantly reduces data retrieval times by storing frequently accessed information in memory. For our movie information system, the caching process helps accelerate application and database performance. It can assist our application in maintaining our request-response performance at 50 ms per request satisfying our TR. Amazon ElasticCache service will help us perform quick memory operations and thus should accompany all AWS Lambda functions that will ensure their quick operations. To achieve the same, we have placed an ElasticCache service alongside each AWS Lambda function in each region in our design. Thus, the design will ensure optimal performance in all available regions.[29]
[These services also uphold the **Operational Excellence** Pillar of the AWS WAF]

Cost Optimization TR:

6. In the design of our movie information system, cost optimization is a fundamental concern, addressed through strategic deployment of AWS services such as AWS Elastic Load Balancing (ELB) and AWS Cost and Usage Report.

   Continuous monitoring and resource optimization using AWS Cloud Watch and Elastic Load Balancing (TR5.1):
   AWS ELB is instrumental in optimizing compute resources across the system. By distributing application traffic across multiple EC2 instances in different Availability Zones within the VPC, ELB ensures that no single resource is overburdened. This is done with the help of AWS Cloud Watch. This distribution allows us to make optimal use of the available compute resources, as ELB dynamically responds to changes in traffic and system load. The architecture avoids flooding any one server with requests, which not only enhances performance but also aids in cost control — resources are scaled up or down based on demand, thus adhering to a pay-per-use model that is one of the cornerstones of cloud cost optimization.

   Cost Management using AWS Cost and Usage Report (TR5.2):
   Complementing AWS Cloudwatch and ELB, the AWS Cost and Usage Report service provides an in-depth look into our cloud spending, allowing us to analyze the relationship between resource usage and costs. This service helps us to identify and eliminate wasteful expenditure, as well as to capitalize on cost-saving opportunities such as reserved instances or sustained use discounts.

Security TR

7. Identifying and thwarting security threats (TR 12.1, TR 12.2):
   AWS GuardDuty is one of the services provided by AWS for securing the VPC. AWS Guard Duty continuously monitors AWS data sources such as, VPC flow logs, DNS logs etc to detect malicious activity. We will be using machine learning support and threat intelligence provided by GuardDuty to identify potential security threats. One of the main advantages of this service is that in future if required we can leverage its ability to integrate with threat intelligence provided by third-party service. This accommodates our security requirements and also allows for future improvements.

5.5 Design principles and the best practices used

The architecture of the movie information system leverages several design principles and best practices, ensuring that the system adheres to the tenets of operational excellence and is poised for robust and efficient performance. Each best practice is closely tied to the design principles, ensuring that the architecture of the movie information system not only adheres to the principles but also demonstrates their practical application in a cloud environment.

1. Operational Excellence[30]
   Design Principles:
   - **Perform operations as code**:
     - Organization and Preparation - Using AWS CloudFormation enables the movie information system to manage operations as code, allowing for consistent and repeatable deployment processes.
   - **Make frequent, small, reversible changes:**
     - Operate - The use of AWS Auto Scaling Groups (TR2.1) and Amazon CloudWatch (TR9.1) for monitoring operational performance allows for frequent, small, and reversible changes by automatically adjusting capacity to maintain steady performance.
   - **Anticipate failure:**
     - Evolve - By leveraging AWS Lambda (TR11.1) for serverless operations, including specific, non-continuous tasks, such as handling S3 events or CloudFront requests. Our use of Lambda is to run functions in response to specific triggers rather than manage traffic, which ELB takes care of. Hence, the system anticipates failure by reducing the server management burden and automatically scaling without requiring manual intervention.
2. Security[31]
   Design Principles:
   - **Implement a strong identity foundation:**

    - Identity and Access Management - AWS Cognito and IAM (TR3.1 and TR3.2) validate the principle of a strong identity foundation by managing user authentication and authorization, ensuring secure access.
- **Enable traceability:**
    - Detection - AWS GuardDuty (TR12.1 and TR12.2) support the principle of traceability by monitoring the environment for suspicious activity and logging API calls across the AWS infrastructure.
- **Protect data in transit and at rest:**
    - Use secure protocols - AWS RDS offers features like encryption at rest using AWS KMS-managed keys, which directly aligns with the data encryption requirement (TR4.1). It allows for encryption of data in transit with SSL/TLS, adding another layer of security as data moves between the database and the application.

3. Reliability[32]
   Design Principles:
    - **Test recovery procedures:**
        - Workload Architecture - Implementation of AWS S3 for data backups (TR8.2) and AWS RDS (TR8.1) for distributed database architecture validates testing recovery procedures by ensuring data is not lost and the system can recover from data-related incidents.
    - **Automatically recover from failure:**
        - Failure Management - AWS RDS's Multi-AZ deployments (TR4.1) enables the principle of automatic recovery by providing high availability and failover support for DB instances.
    - **Scale horizontally to increase aggregate system availability:**
        - Change Management - Utilizing AWS ELB (TR1.1 and TR1.2) validates the principle of horizontal scaling by distributing traffic across multiple instances, increasing availability.

4. Performance Efficiency[33]
   Design Principles:
    - **Democratize advanced technologies:**
        - Monitoring - AWS CloudWatch validates democratizing advanced technologies by providing advanced monitoring tools readily available for performance insights.(TR9.1, TR9.2)
    - **Go global in minutes:**
        - Tradeoffs - The use of Amazon CloudFront for content delivery demonstrates the principle of going global in minutes by allowing quick

deployment of content to global locations, balancing performance and cost. (TR10.1)

- **Use serverless architectures:**
    - Selection - AWS Lambda and Amazon S3 enable the movie information system to run code without provisioning servers and to store data globally.

5. Cost Optimization
   Design Principles:
    - **Adopt a consumption model:**
        - Cost-effective Resources: AWS Auto Scaling and choosing the right-sized EC2 instances validate the consumption model by ensuring resources match demand without over-provisioning.(TR2.1)
    - **Stop spending money on undifferentiated heavy lifting:**
        - Optimize Over Time - Employing managed services like Amazon RDS and AWS Lambda validates stopping spending on undifferentiated heavy lifting by outsourcing infrastructure management to AWS, reducing operational costs.
    - **Measure overall efficiency:**
        - Align costs with demand - AWS Cost and Usage report helps keep track of the cost of resource utilization which allows us to identify areas of unnecessary spending and mitigate those costs and realign the resource utilization.(TR5.2)

6. Sustainability[35]
   Design Principle:
    - **Understand the environmental impact of your workload:**
        - Region Selection - The use of AWS regions closest to the user base for deploying Amazon CloudFront and S3 validates understanding the environmental impact by reducing latency and the energy required to deliver content.
7. Design for Automation:
    a. **Infrastructure Automations:** Utilizing AWS CloudFormation enables the movie information system to define the entire stack as code, which ensures that infrastructure deployment is repeatable, consistent and version-controlled. This approach leads to improved traceability and auditable infrastructure changes.
    b. **Scale Up and Scale Down:** AWS Lambda's serverless model is central to the system's ability to handle variable workloads efficiently. It automatically scales computing resources in response to the number of incoming requests, ensuring the system remains responsive during peak times without incurring unnecessary costs during quieter periods.

c. **Monitoring and Automated Recovery:** Integrating AWS CloudWatch into the architecture provides comprehensive monitoring capabilities for all AWS resources. This service collects metrics and logs enabling the system to respond to operational changes proactively. CloudWatch Events are used to trigger automated responses, ensuring that potential issues are addressed with minimal manual intervention.

8. OTHER DESIGN PRINCIPLES:
    1. Adherence to **KISS** (Keep it Simple, Stupid) Principle:
        a. **Simplicity in Design:** The architecture avoids unnecessary complexity by using managed services like AWS RDS for database management, AWS Elastic Load Balancing for traffic distribution and Amazon S3 for storage. This simplicity reduces the cognitive load on developers and operators, enabling them to focus on the core functionality of the system rather than the intricacies of the underlying infrastructure.
        b. **Decoupling Components:** By decoupling the system components, the architecture ensures that each part can be developed, deployed and scaled independently. This not only simplifies management but also isolates faults, preventing a single point of failure from affecting the entire system.
        c. **Stateless Applications:** Where possible, the system components are designed to be stateless, particularly with AWS Lambda functions, which further simplifies scaling and recovery processes.
    2. Favour Managed Services
        a. **Operational Efficiency with Managed Services:** In our architecture, we integrate managed services like AWS Elastic Load Balancing, Amazon API Gateway, and Amazon S3, which significantly streamline operations. These services not only simplify the deployment and management of resources but also enhance the system's scalability and reliability. By leveraging these managed services, we reduce the complexity of maintenance and operational monitoring, leading to increased operational efficiency and cost savings. This approach allows us to focus our efforts on adding value to the movie information system rather than on the undifferentiated heavy lifting of infrastructure management. (ChatGPT)
        b. **Security and Compliance with Managed Services:** Beyond operational savings, we utilize specialized managed services that cater to specific needs of the movie information system. For instance, AWS Identity and Access Management (IAM) plays a pivotal role in our security posture. While it may not directly contribute to operational savings like open-source solutions might, AWS IAM is instrumental in fortifying our system's security. It provides fine-grained access control to AWS resources, ensuring that only authorized users and services can access sensitive

data and functionality. This focused use of managed services ensures that our system adheres to strict security protocols, thereby mitigating potential security risks. (ChatGPT)

5.6 Tradeoffs revisited

1. TR 1.1: High Availability to support 99% uptime vs TR 1.2: Continuous monitor and failure detection to minimize downtime

   The tradeoff is between the high availability (TR 1.1) and automatic maintenance process to avoid failure (TR 1.2) which could momentarily impact uptime. In cloud systems, balancing System Uptime and Maintenance Flexibility often involves trade-offs due to their interconnected and sometimes conflicting requirements.

   System Uptime refers to the time a system is operational and available to users. High Uptime is critical for customer-facing applications in which services are constantly accessed. High Uptime ensures reliability, and crucial for services where continuous availability is a must. But it comes with a cons that it requires a complex architecture with robust load balancing and failure mechanism increasing cost and complexity.

   Maintenance Flexibility refers to the ease with which system can be updated or repaired in case of any failure without any significant downtime. It allows the system to be updated frequently and also facilitates rapid response to emergencies and issues. Achieving this requires a system with loosely coupled components, which can add complexity and potential performance overhead.

Parameters	**High Availability**	**Maintenance Flexibility**
Scheduled Downtime Vs Continuous Availability	May limit window for maintenance, therefore challenging to schedule maintenance	System may face some downtime
Cost and Complexity	Can be expensive and complex to manage	Can be expensive and complex to manage
Performance	High performance	May introduce some latency
Scalability and Redundancy	Use highly scalable components, sometimes unnecessary usage	Unnecessary usage can be reduced by frequently updating based on logs
Risk Management	Due to complexity, risk of	Continuous maintenance

	security vulnerabilities	can reduce security risk

The architecture prioritizes System Uptime with multiple Availability Zones, Load Balancing, Auto Scaling and RDS. These services collectively ensure that the system can remain operational even if one or more components fail. However, Maintenance Flexibility is also addressed through the use of managed services like RDS, ElastiCache and Lambda, which can be updated or modified without impacting the overall system availability.

Our architecture prioritizes System Uptime of 99% based on following assumptions about business. Our system requires fast, continuous access to the system even if it costs more. We make an assumption that the cost required to maintain high availability is comparatively less than revenue lost due to downtime because of maintenance, implicating that latency should also be avoided. We have employed AWS GuardDuty, AWS IAM and AWS Cognito which can handle minor security risks which provides us an option to prioritize high availability. (ChatGPT)

2. TR 5.1 and 5.2: Cost Optimization vs TR 2.1: Performance Stability

   Cost Optimization often means running the minimum number of resources necessary to meet performance requirements, which can conflict with performance stability. Performance Stability is ensuring the system can handle varying loads with minimal latency or downtime.

Parameters	**Cost Optimization**	**Performance Stability**
Resource Utilization	Tries to minimize	Sometimes may over-provision resources
Latency	No promise about latency	Tries to minimize latency

As discussed in previous tradeoff, it is mentioned that the business priority is to provide fast service and assumption about the cost to maintain minimum performance is comparatively less than revenue missed due to latency and performance issues. Hence we decided to prioritize Performance Stability over Cost Optimization in our architecture.

For maintaining minimum performance, we have made use of availability zones, fault tolerance mechanism, auto scaling and load balancing services. Also caching and content delivery services are also used to provide fast responses.

However, we have not neglected cost optimization. The use of scalable and managed services suggests a balance, as these services are generally more cost effective than maintaining an equivalent on-premises infrastructure. The use of AWS Cost and Usage Report service also indicates attention to cost tracking and optimization. (ChatGPT)

3. TRs 5: Cost Optimization vs TRs 8: Data Integrity and Availability

   Cost Optimization focuses on reducing expenses by efficient resource utilization, choosing cost-effective service tiers, and avoiding over-provisioning. Whereas Data Integrity and Availability ensures that data from various sources are consolidated effectively and this data is available when needed. This often requires robust infrastructure and redundancy which increases costs.

Parameters	**Cost Optimization**	**Data Integrity and Availability**
Resource Utilization	Tries to minimize	Requires more resources to maintain data availability
Latency	No promise on latency	Ensuring low latency is key for data availability
Redundancy	Cannot provide	Requires multiple copies of data in case of failure
Data Transfer Cost	Data transfer is slow	Requires fast data transfer for quick data integrity
Scalability	Cost ineffective	If high workload, scalability is key

   As discussed in previous tradeoffs, we have decided to prioritize Data Integrity and Availability. Architecture aims to provide a resilient and highly available system without disregarding cost. It employs AWS services that offer scalability and data integration while including tools to monitor and manage expenses. Data Integrity and Availability is achieved through the uses of Multi-AZ RDS and Auto Scaling points towards a priority on availability. ElastiCache is included to reduce latency and database load, which is more aligned with performance and availability than cost. AWS CloudFront and API Gateway indicate a priority to availability and performance across geographical locations. While AWS CloudWatch and AWS Cost and Usage Report show that monitoring resource utilization and cost is also a priority ensuring that cost optimization is not overlooked.

4. TRs 3: Tenant Identification vs TRs 12: Security

   Tenant Identification refers to the ability to distinguish between different customers using the same cloud resources or services. Security aspect of architecture encompasses the measures and mechanisms put in place to protect data, applications, and integrity of computing resources from threats, unauthorized access and data breaches.

Parameters	Tenant Identification	Security
Resource Utilization	Goal is to use less resources	Requires more security mechanism, hence more resources
Latency	Little overhead	May be larger overhead if too many checks
User Experience	No extra overhead to user	User has to provide more identity information for access

In the discussion of Tradeoff 1, it is mentioned that to ensure fast our requirement is to provide minimal latency which security compliance would not guarantee. Also the architecture is widely distributed, in a sense that data is backed up in case of failure, highly scalable to handle workload changes and use of different availability zones to isolate one service point with another. These features provide resilience towards most of the attacks. If any one resource is attacked, other resources can still provide service without any issues. Because of these reasons, we decided to prioritize tenant identification over security. But we have employed AWS services in a manner that ensures tenants can be identified securely and efficiently, with security measures integrated into the process of tenant management to provide seamless and secure user experience. The balance between security and tenant identification is achieved by leveraging AWS's managed services, which offer both robust security features and tenant management capabilities.

5.7 Discussion of an alternate design [SKIPPED]

6 Kubernetes Experimentation

6.1 Experiment Design

As part of our experiment with k8s, we have selected following TRs to work with:

1. TR1.1: Design for high availability for 99% uptime requirement.
2. TR1.2: Fault tolerance to minimize downtime.
3. TR2.1: Autoscaling of computing resources based on time varying workloads.
4. TR5.1: Continuously monitor cloud resource usage to identify optimization opportunities.
5. TR3.3: Isolate tenants properly.

A brief description of Kubernetes:

- Kubernetes, often abbreviated as K8s, is an open-source platform designed to automate the deployment, scaling, and management of containerized applications. It organizes containers, which are the components of an application, into logical groups for simplified management and discovery. Kubernetes offers a range of building blocks that support the deployment, maintenance, and scaling of applications based on various metrics like CPU and memory usage. This system is modular and flexible, allowing it to accommodate diverse workloads.[36]
- Kubernetes operates through a loosely coupled architecture, relying on its API for interactions between its internal components, extensions, and containers. It manages computing and storage resources by treating them as Objects within the system.[36]
- As a sophisticated distributed system, Kubernetes effectively uses a cluster of machines as a unified resource pool. It functions like a distributed operating system, managing tasks such as scheduling resources, monitoring infrastructure health, and ensuring that the infrastructure and workloads maintain their desired states. Kubernetes is adept at running modern applications in various environments, including cloud services and private data centers. It consists of two primary layers: the head nodes and the worker nodes. The head nodes, forming the control plane, are responsible for workload scheduling and lifecycle management. Meanwhile, the worker nodes execute the applications. Together, these nodes form a cluster, enabling efficient and scalable application management.[37]

Design of experiments:

- **TR 1.1**: To ensure high availability for the 99% uptime requirement, the experiment involves configuring a Kubernetes cluster using Minikube with a single service replica, increasing the replica count based on CPU Utilization and implementing a load balancer for even traffic distribution. Deliberate pod termination is simulated to test the system's resilience. Then, we assess the difference between requests per second with the failed requests per second to

- **TR 1.2:** For fault tolerance to minimize downtime, the experiment sets up redundancy in the Minikube cluster with replicated pods distributed across different nodes. We introduced intentional pod termination to test the system's response. The time taken for the system to recover and return to normal operations is tracked using monitoring tools. The evaluation focuses on the system's ability to automatically redistribute the load and spawn new pods, minimizing downtime during fault scenarios.
- **TR 2.1:** To validate the autoscaling of computing resources based on time-varying workloads, the experiment deploys Horizontal Pod Autoscaler in Minikube to automatically adjust the number of pods in a deployment. Load conditions are created with locust tools to simulate varying traffic levels. Metrics related to the number of pods, CPU, and memory usage are continuously monitored during the tests. The evaluation confirms whether the system effectively scales up during periods of high load and scales down during low load without requiring manual intervention.
- **TR 5.1:** For monitoring purposes, inbuilt functionalities provided by Minikube like dashboard are used. Normal operations and stress conditions are executed to collect continuous metrics, including CPU usage, memory consumption, response times, and error rates. The evaluation includes monitoring the system performance to get metrics which can be used to identify patterns and help optimize resource utilization, improving the overall performance of the system.
- **TR 3.3:** For testing the isolation of tenants, we decided to experiment with the k8s network policy using AWS CLI. Here, we plan to create 3 namespaces, where the pods in 2 namespaces (our admin and user tenants occupy each namespace) are, by default, able to access the pods in the 3rd namespace (our database namespace with all the movie information). However, when the admin tenants want to access the database namespace for maintenance purposes, our user tenants must not be able to access the pods in the database namespace during this time period. For this reason, we intend to use the k8s network policy to not allow the pods in the user namespace to ping the pods in the database namespace and only allow the admin namespace pods to do so, adding an additional layer of security.

Configuration for experiments:

- Our **application** can perform two tasks. First, listing all the movie names available and second it takes a movie name as input and provides further information about the movie. The application is built using the Flask framework.
- The **Dockerfile** serves as a blueprint for building a Docker container image, tailored specifically for a Python application using Flask. It begins by basing the image on a slim variant of Python 3.9, optimizing for both size and security. The working directory is established within the container's file system, providing a dedicated space for the application's code. Dependencies, notably Flask, are installed using `pip`, ensuring the container will have all the necessary Python packages. The application code itself,

`app.py`, is then copied into the image, making it an integral part of the container. The Dockerfile also exposes port 5000, which is the default port for Flask applications, signaling that the container is intended to listen for web traffic on this port. Environmental variables are set to configure Flask, dictating the application to run and allowing it to accept traffic on all network interfaces. The final instruction specifies the command to start the Flask application when the container is run. Overall, this Dockerfile is pivotal for creating a consistent and self-sufficient runtime environment that is portable across any system that supports Docker, thereby simplifying deployments and development workflows. (ChatGPT)

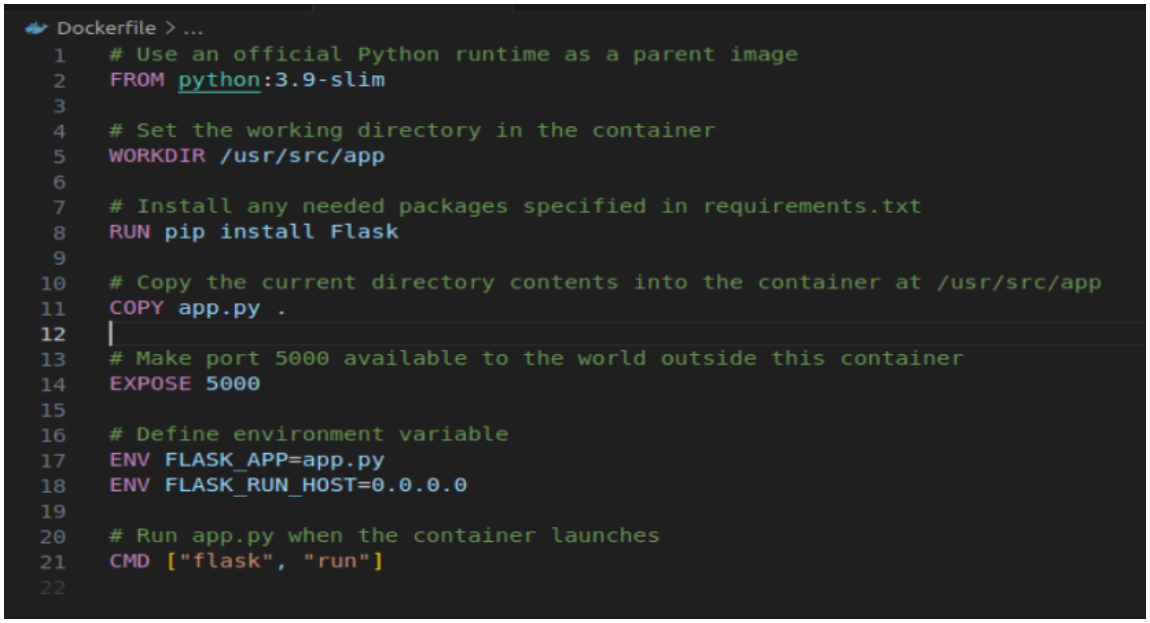


Figure: Dockerfile for our application

- The **Kubernetes deployment YAML** file is an orchestration specification that tells Kubernetes how to run and manage the application in a cluster environment. The YAML file is divided into two parts: one defining the service and the other describing the deployment. The service part ensures that the Flask application is accessible via a network, utilizing a LoadBalancer to distribute incoming traffic efficiently. This is vital for maintaining the application's accessibility and performance. The deployment section of the YAML file outlines the desired state of the application, including the number of replicas, which improves fault tolerance and availability by ensuring multiple instances are always running. It includes a reference to the container image built from the Dockerfile, marrying the containerized application with the Kubernetes ecosystem. (ChatGPT) The deployment also stipulates resource allocation, setting both requests and limits for CPU usage, which is crucial for maintaining performance and ensuring efficient resource utilization in the cluster. By managing these aspects, the deployment YAML is an essential component for automating the deployment, scaling, and operation of

containerized applications with Kubernetes, aligning with best practices for cloud-native development. (ChatGPT)

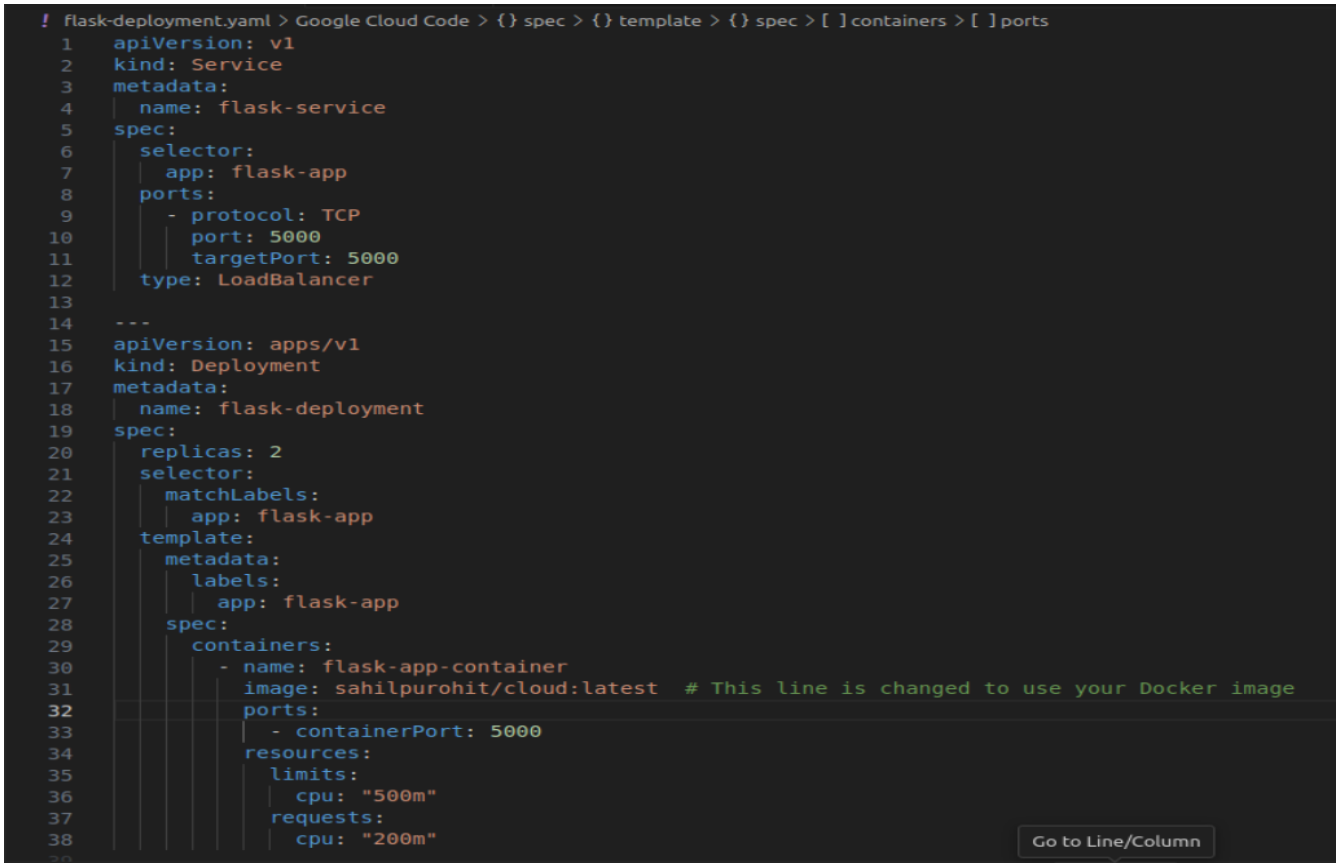


Figure: Kubernetes deployment YAML file

- **Network Policy YAML file**

  The YAML file defines a ‘NetworkPolicy’ in Kubernetes named ‘test-network-policy’ for the namespace "two", which dictates the allowed network connections to the pods. The policy specifically targets pods labeled with ‘environment: test’ within its namespace and permits inbound traffic exclusively from pods located in any namespace labeled with ‘myspace: one’. This selective ingress enforcement ensures that only pods from the specified namespace can communicate with the targeted pods, thereby isolating network traffic for security and organizational compliance.[40]

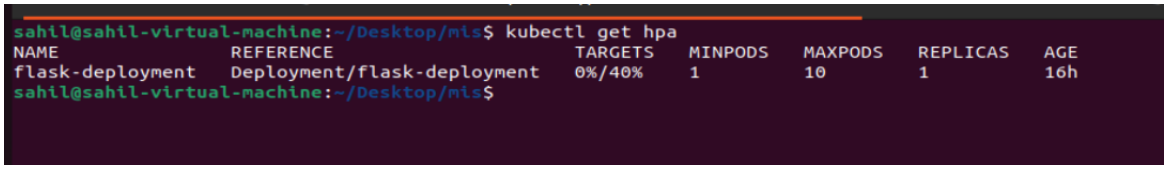
```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: two
spec:
  podSelector:
    matchLabels:
      environment: test
  policyTypes:
  - Ingress
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          myspace: one
```

Figure: Network Policy YAML File

Locust Experiment Setup:

- The **Horizontal Pod Autoscaler (HPA)** in Kubernetes is a crucial tool for applications requiring scalability, designed to automatically adjust the number of pods in a deployment based on the CPU utilization observed. It functions by leveraging metrics collected by the metrics server, which monitors resource usage and informs Kubernetes of the current demand. This data is pivotal in determining when to scale the number of pods up or down to meet the specified performance targets. (ChatGPT)
- To enable the HPA, we initiate it with a command specifying the deployment to scale, the target CPU utilization percentage, and the minimum and maximum number of pods allowed. For example, the command **'kubectl autoscale deployment flask-test-app --cpu-percent=40 --min=1 --max=10'** instructs Kubernetes to keep the CPU usage of the pods close to 40% [Note: We have chosen 40% to be the CPU utilization limit to showcase the scaling up and down of our cluster effectively]. If the usage goes higher, the HPA will increase the number of pods, up to a maximum of ten, to distribute the load more effectively. Conversely, if the CPU usage drops, it can reduce the number of pods to a minimum of one, optimizing resource usage. To monitor the status and effectiveness of

the HPA, the command **'kubectl get hpa**' lists all active Horizontal Pod Autoscalers, providing insights into their operations and the current scale of the associated deployments. This mechanism ensures that the application can handle varying workloads efficiently, maintaining performance and resource management dynamically. [38]



- The command **'kubectl get hpa flask-deployment --watch'** is used to monitor the status of a specific Horizontal Pod Autoscaler (HPA) in real-time within a Kubernetes cluster. In this case, it's targeting an HPA named **'flask-deployment'**. The **'get hpa'** portion of the command retrieves the current state of the autoscaler, including metrics like the current number of replicas, target CPU utilization, and current CPU utilization. [38]
- The **'--watch'** flag is important as it instructs **'kubectl'** to continuously monitor the HPA and output the results to the console. Instead of returning immediately, the command stays active and provides updates as they occur, reflecting any changes to the HPA in real-time. This shows how the HPA responds to changes in workload and ensures that the deployment scales as expected according to the configured parameters.[38]

Network Policy Experiment Setup:

- The aim of this experiment is to verify isolation of servers from different users. One scenario is during maintenance, database servers should only be accessible from admin namespace servers and users namespace servers shouldn't. In terms of our application in addition to maintenance, this experiment can also justify scenarios like a user should only have access to a database for read request and write request from user to database should not be accepted. This requires a policy driven network and our experiment verifies similar policy.[40]

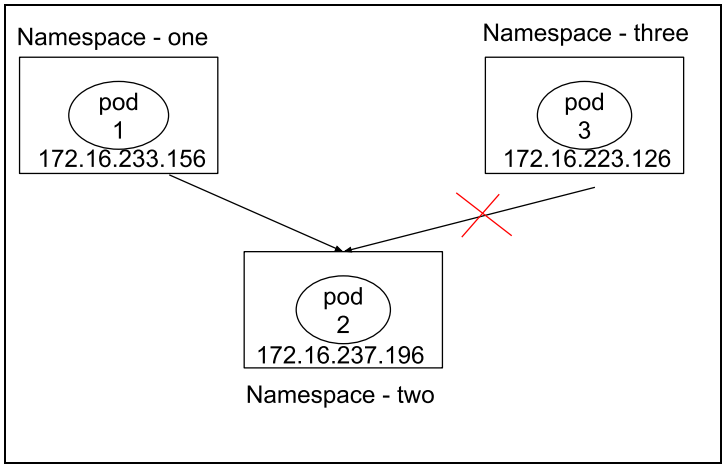


Figure: Network policy experiment setup

- **Above figure** shows the basic experiment setup: Namespace - one simulates Admin namespace, Namespace - two simulates Database namespace and Namespace - three simulates user namespace. The experiment is that Namespace two should allow request from Namespace one and not from Namespace three.
- We configure a Kubernetes cluster with Amazon EKS, emphasizing custom network configuration using Calico as the Container Network Interface (CNI). The experiment begins with creating an EKS cluster without default node groups, followed by the removal of the AWS-native CNI (**'aws-node'**) to pave the way for Calico. By installing the Tigera Operator and configuring Calico, the experiment explores an alternative networking setup, specifically opting out of using BGP for routing within the Calico network.[40]
- The experiment proceeds to expand the cluster by adding node groups with specific instance types and pod capacity. We then focus on namespace management and resource isolation by creating and editing multiple namespaces. Within these namespaces, we deployed Nginx servers as test workloads. Network policies are crafted and iteratively refined, where we need to edit and apply a network policy file. The final stages of the experiment involve listing detailed pod information across all namespaces and testing inter-pod connectivity using curl commands executed within specific pods. This comprehensive setup serves to evaluate the behavior and performance of the Kubernetes cluster under a Calico-managed network environment, with particular attention to cross-pod communication and namespace-specific network policies.[40] **The following images and code show the steps mentioned in this paragraph**:

```
eksctl create cluster --name mycalico-cluster --without-nodegroup
kubectl delete daemonset -n kube-system aws-node
kubectl create -f https://raw.githubusercontent.com/projectcalico/calico/v3.26.4/mani

# Creating Calico installation with bgp disabled
kubectl create -f - <<EOF
kind: Installation
apiVersion: operator.tigera.io/v1
metadata:
  name: default
spec:
  kubernetesProvider: EKS
  cni:
    type: Calico
  calicoNetwork:
    bgp: Disabled
EOF
```

```
eksctl create nodegroup --cluster mycalico-cluster --node-type
t2.micro --max-pods-per-node 100 --nodes 2
# Creating namespaces
kubectl create ns one
kubectl create ns two
kubectl create ns three

# Running Nginx deployments/pods in different namespaces
kubectl run nginx --image=nginx
kubectl -n one run pod_one --image=nginx
kubectl -n one run podone --image=nginx
kubectl get po -n one
kubectl -n two run podtwo --image=nginx
kubectl -n three run podthree --image=nginx

#Network policy applying
kubectl apply -f net
kubectl get po -A -o wide
```

6.2 Workload Generation with Locust

- Locust is an open-source tool used for load testing, which is a process to evaluate the resilience and behavior of a system when subjected to expected levels of demand. It specifically aims to assess the performance of websites by determining how many simultaneous users the system can support.[39]

- As a part of performance testing, load testing examines how the application behaves when multiple users access it at the same time. The focus of such tests is on assessing the application's speed and capacity under varying loads. Adjusting the load during testing helps in understanding the system's adaptability to different stress levels. Using Locust, one can emulate a specified number of users to generate traffic and send requests to the API server, thus mimicking real-world traffic patterns and studying how the application manages the stress.[39]
- For our application, we intent to use locust to generate load using two scenarios:
    - First scenario is a linearly increasing load of 1500 users for 300 seconds with a spawn rate of 5 users/second. (Ramp Load)
    - Second scenario is a staged based load generation on kubernetes cluster as following:
        - Duration: 0 - 60 sec Users: 120 Spawn Rate: 2
        - Duration: 60 - 90 sec Users: 30 Spawn Rate: 3
        - Duration: 90 - 180 sec Users: 720 Spawn Rate: 8
        - Duration: 180 - 220 sec Users: 520 Spawn Rate: 4
    - During the interval 0 - 60 seconds, we are increasing the load to 120 users by a spawning rate of 2 users per second. Time interval 60 - 90 seconds, we are decreasing the load to 30 users at a decreasing spawning rate of 3 users per second. During the interval 90 - 120 seconds, we are increasing the load to 720 users by a spawning rate of 5 users per second. Time interval 120 - 180 seconds, we are decreasing the load to 520 users at a decreasing spawning rate of 4 users per second.
- In these experiments, we aim to verify whether our k8s environment is able to satisfy the mentioned Technical Requirements.

6.3 Analysis of the results

1. **Experiment 1:** Testing our k8s cluster with a linearly increasing load of 1500 users for 300 seconds with a spawn rate of 5 users/second. (Ramp Load)

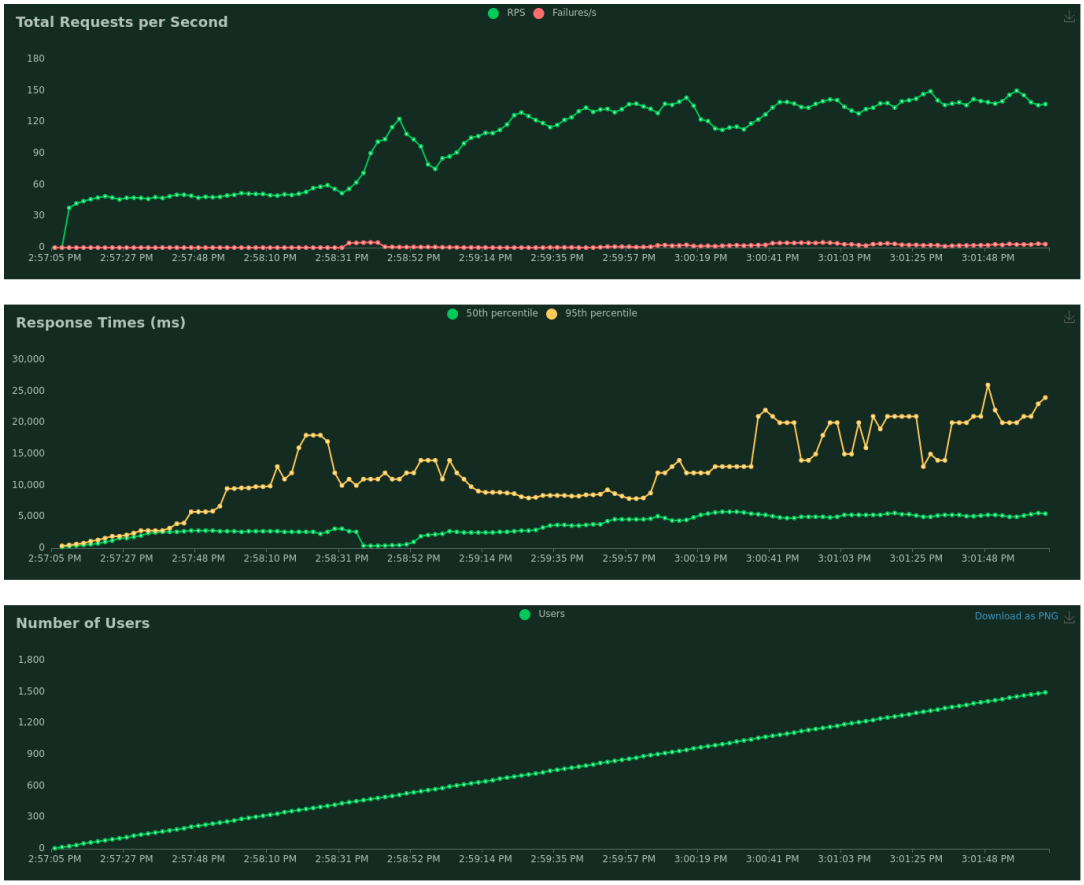


Figure 1: Locust: Ramp load applied to our application (The first graph shows the total requests sent to the cluster per second with the number of failures per second too as depicted by the green and red lines respectively) (The second graph shows the 50th and the 95th percentile of the response times of the application depicted by green and yellow lines respectively)(The third graph shows the load being provided to our application)

```
sahil@sahil-virtual-machine:~/Desktop/mis$ kubectl get hpa flask-deployment --watch
NAME               REFERENCE                     TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
flask-deployment   Deployment/flask-deployment   33%/40%    1         10        1          14h
flask-deployment   Deployment/flask-deployment   209%/40%   1         10        1          14h
flask-deployment   Deployment/flask-deployment   209%/40%   1         10        4          14h
flask-deployment   Deployment/flask-deployment   209%/40%   1         10        6          14h
flask-deployment   Deployment/flask-deployment   154%/40%   1         10        6          14h
flask-deployment   Deployment/flask-deployment   113%/40%   1         10        6          14h
flask-deployment   Deployment/flask-deployment   113%/40%   1         10        10         14h
flask-deployment   Deployment/flask-deployment   119%/40%   1         10        10         14h
flask-deployment   Deployment/flask-deployment   26%/40%    1         10        10         14h
flask-deployment   Deployment/flask-deployment   0%/40%     1         10        10         14h
flask-deployment   Deployment/flask-deployment   0%/40%     1         10        10         14h
flask-deployment   Deployment/flask-deployment   0%/40%     1         10        4          14h
flask-deployment   Deployment/flask-deployment   0%/40%     1         10        4          14h
flask-deployment   Deployment/flask-deployment   0%/40%     1         10        1          14h
^Csahil@sahil-virtual-machine:~/Desktop/mis$
```

Figure 2: Kubernetes: AutoScale monitoring for Ramp Load in MiniKube

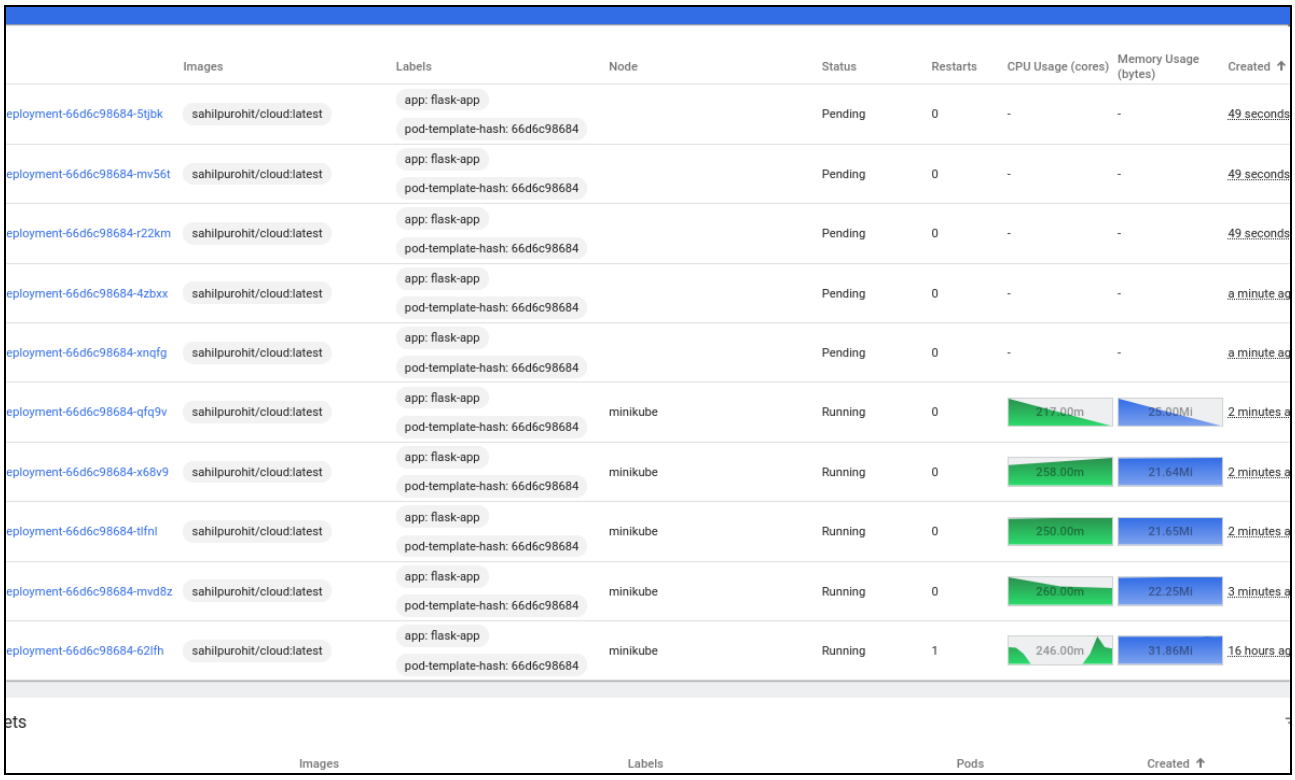


Figure 3: Kubernetes Dashboard: CPU Utilization and Performance Monitoring during Ramp Load application to our cluster.

The figures provided depict the outcomes of an experiment aimed at assessing technical requirements (TRs) for our application.

1. **High Availability (TR 1.1):** Figure 2, showing the "Total Requests per Second," reflects the system's ability to handle a steady stream of requests over time without significant drops, indicating high availability. If we go into detail, the high availability of the system is evidenced by the minimal number of failed requests, which, at an average of 1 to 5 failures per second amidst an average load of approximately 120 requests per second, indicates a robust system performance. Based on this data, we can deduce that the system's availability is within an impressive range of 96% to 99%. This shows that there are little to no service interruptions, demonstrating the system's ability to maintain operational status as it handles incoming requests amidst increasing workload, showcasing high availability and satisfying TR1.1.
2. **Autoscaling (TR 2.1):** The autoscaling feature of Kubernetes is demonstrated by the varying number of pod replicas in Figure 1. The HPA has adjusted the number of ‘flask-deployment’ pods from 1 to 10 based on the CPU utilization targets, allowing the system to respond to changes in workload dynamically. This ensures that the computing resources are scaled according to time-varying workloads, directly satisfying the autoscaling requirement. In Figure 2, as Target CPU

utilization increases from 33% to 209%, the number replicas also increases from 1 to 4 and 4 to 6. A note here is that 'watch' commands does not provide real time output, it generates output at different time intervals. When target utilization reached 209%, normally total replicas should be 6. But during this time 5 pods were in the process of creation out of which 3 pods started running quickly and remaining 2 were still in the pending state, similar results can be seen in Figure 3. Also, even with 6 replicas at 209% CPU utilization, the target of 40% was not achievable. So, the HPA spun up additional 4 pods to reach the maximum number(10) which was then able to reduce the target down to 113% (Still not under 40%). If the target CPU Utilization was more (say 70%), the target may have been reached by a lesser number of pods. Hence, we have showcased the auto scaling aspect of our cluster using kubernetes, satisfying TR2.1.

3. **Resource Usage Monitoring (TR 5.1):** Continuous monitoring is showcased in Figure 2, which displays metrics like total requests per second and response times and in Figure 1, with the HPA monitoring. In Figure 2, when CPU utilization is below 40%, even when the load is increasing in the initial stages, the response time is fast (between 2:57:05 pm - 2:57:48 pm) and there is no requirement for additional pods. However, in times where the load is greater than 40%, the HPA spins up the minimum possible pods to satisfy the demand. This shows that only the required amount of resources are provided and managed by the HPA, satisfying TR5.1 of continuous resource monitoring for optimization opportunities.

Together, these images represent a system that has been designed to meet the stringent requirements of high availability, autoscaling, and resource usage monitoring, thereby showing the effectiveness of using Kubernetes and tools like Locust and HPA for managing cloud-native applications.

2. **Experiment 2:** Testing our k8s cluster with a staged based load generation on kubernetes cluster as following:
    a. Duration: 0 - 60 sec Users: 120 Spawn Rate: 2
    b. Duration: 60 - 90 sec Users: 30 Spawn Rate: 3
    c. Duration: 90 - 180 sec Users: 720 Spawn Rate: 8
    d. Duration: 180 - 220 sec Users: 520 Spawn Rate: 4

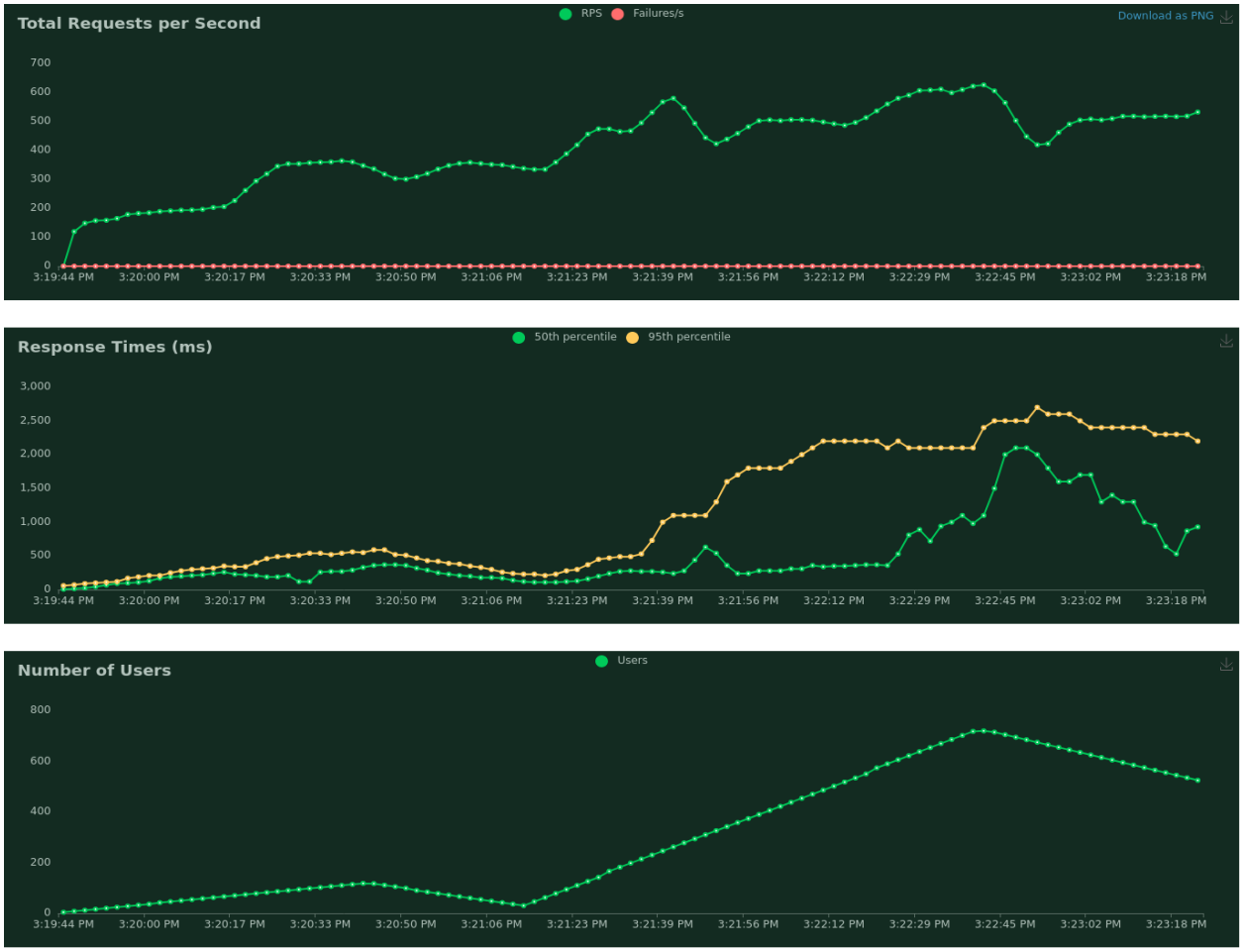


Figure 4: Locust: Staged load applied to our application (The first graph shows the total requests sent to the cluster per second with the number of failures per second too as depicted by the green and red lines respectively) (The second graph shows the 50th and the 95th percentile of the response times of the application depicted by green and yellow lines respectively)(The third graph shows the load being provided to our application)

```
sahil@sahil-virtual-machine:~/Desktop/mis$ kubectl get hpa flask-deployment --watch
NAME               REFERENCE                     TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
flask-deployment   Deployment/flask-deployment   0%/40%     1         10        1          15h
flask-deployment   Deployment/flask-deployment   52%/40%    1         10        1          15h
flask-deployment   Deployment/flask-deployment   52%/40%    1         10        2          15h
flask-deployment   Deployment/flask-deployment   236%/40%   1         10        2          15h
flask-deployment   Deployment/flask-deployment   236%/40%   1         10        4          15h
flask-deployment   Deployment/flask-deployment   236%/40%   1         10        6          15h
flask-deployment   Deployment/flask-deployment   155%/40%   1         10        6          15h
flask-deployment   Deployment/flask-deployment   155%/40%   1         10        8          15h
flask-deployment   Deployment/flask-deployment   118%/40%   1         10        8          15h
flask-deployment   Deployment/flask-deployment   118%/40%   1         10        10         15h
flask-deployment   Deployment/flask-deployment   57%/40%    1         10        10         15h
flask-deployment   Deployment/flask-deployment   0%/40%     1         10        10         15h
flask-deployment   Deployment/flask-deployment   0%/40%     1         10        10         15h
flask-deployment   Deployment/flask-deployment   0%/40%     1         10        10         15h
flask-deployment   Deployment/flask-deployment   0%/40%     1         10        1          15h
^Csahil@sahil-virtual-machine:~/Desktop/mis$
```

Figure 5: Kubernetes: AutoScale monitoring for Staged Load in MiniKube

```
sahil@sahil-virtual-machine:~/Desktop/mis$ kubectl get pods
NAME                                READY   STATUS    RESTARTS       AGE
flask-deployment-66d6c98684-62lfh   1/1     Running   1 (109m ago)   16h
sahil@sahil-virtual-machine:~/Desktop/mis$ kubectl get pods
NAME                                READY   STATUS              RESTARTS       AGE
flask-deployment-66d6c98684-62lfh   1/1     Running             1 (110m ago)   16h
flask-deployment-66d6c98684-mvd8z   1/1     Running             0              83s
flask-deployment-66d6c98684-qfq9v   0/1     Pending             0              7s
flask-deployment-66d6c98684-s7vvp   1/1     Running             0              22s
flask-deployment-66d6c98684-tlfnl   1/1     Running             0              22s
flask-deployment-66d6c98684-x68v9   0/1     ContainerCreating   0              7s
sahil@sahil-virtual-machine:~/Desktop/mis$ kubectl delete pod flask-deployment-66d6c98684-d7b67^C
sahil@sahil-virtual-machine:~/Desktop/mis$ kubectl delete pod flask-deployment-66d6c98684-s7vvp
pod "flask-deployment-66d6c98684-s7vvp" deleted
sahil@sahil-virtual-machine:~/Desktop/mis$ kubectl get pods
NAME                                READY   STATUS    RESTARTS       AGE
flask-deployment-66d6c98684-4zbxx   0/1     Pending   0              34s
flask-deployment-66d6c98684-62lfh   1/1     Running   1 (112m ago)   16h
flask-deployment-66d6c98684-mvd8z   1/1     Running   0              2m39s
flask-deployment-66d6c98684-qfq9v   1/1     Running   0              83s
flask-deployment-66d6c98684-tlfnl   1/1     Running   0              98s
flask-deployment-66d6c98684-x68v9   1/1     Running   0              83s
flask-deployment-66d6c98684-xnqfg   0/1     Pending   0              38s
```

Figure 6: Kubernetes: Pod crash and status after intentionally terminating a running pod

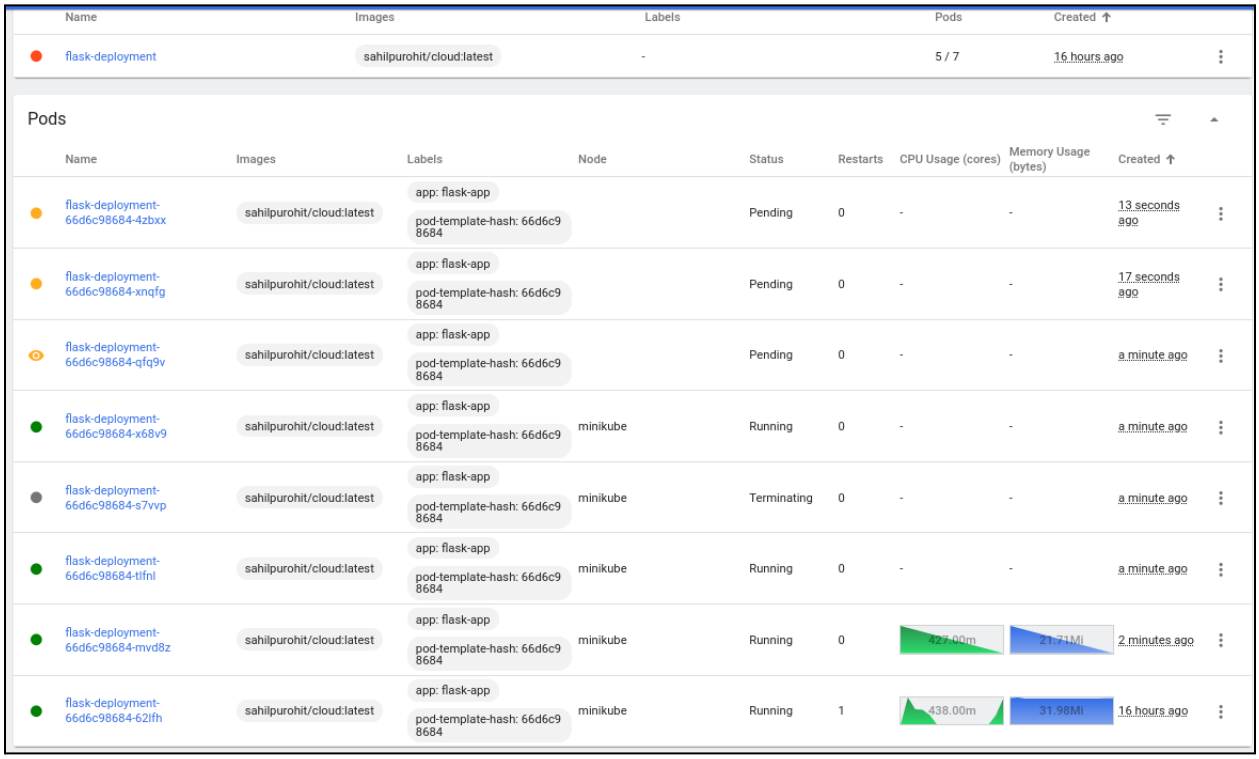


Figure 7: Kubernetes: Pod Crash from Kubernetes Dashboard after deliberated pod termination during staged load

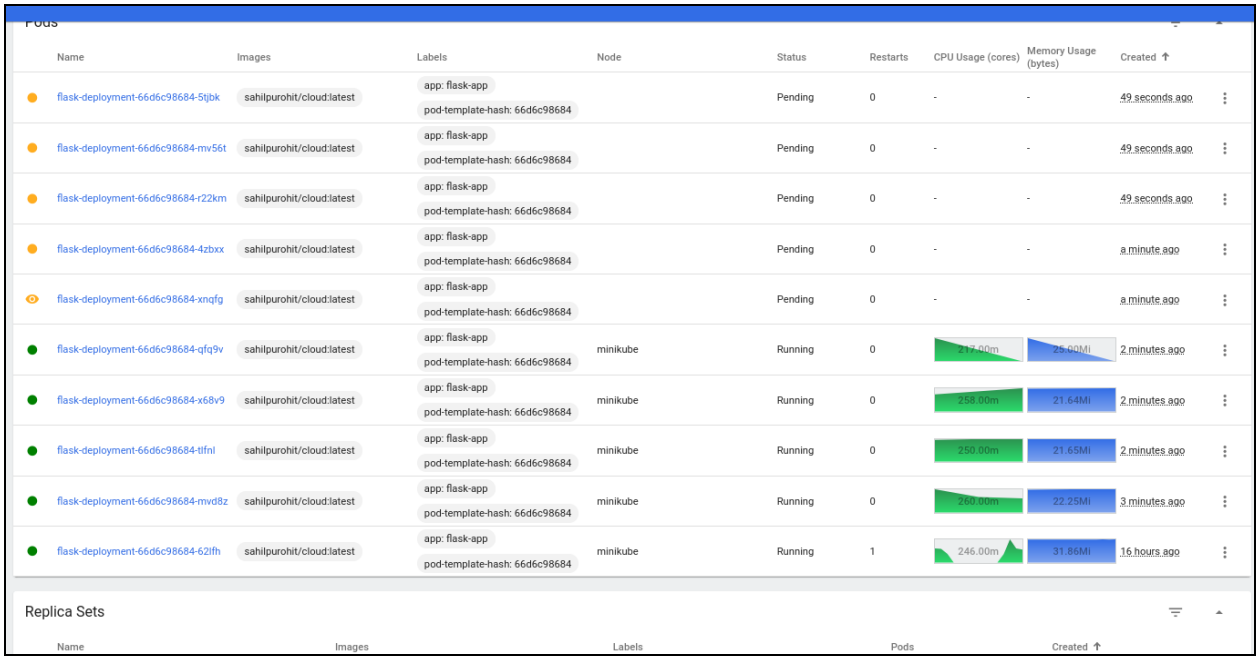


Figure 8: Kubernetes: Failure Recovery from a deliberated pod termination by recreating the killed pod

The **figures provided above** depict the outcomes of an experiment aimed at assessing technical requirements (TRs) for our application again.

1. **High Availability (TR 1.1):** Here we present another way of verifying the availability requirement. Availability can also be measured as a factor of failure time and normal behavior period. From Figure 6, we can conclude that after the failure of a pod, it took **5s** for a new pod to be created and total test time was **220s.** Therefore, availability = 100 - (5/220)*100 = 97.5%. Here, a point to note is that the pod creation time is independent of total time of normal behavior. Therefore, in case of failure, 34s failure time is unavoidable and hence we have less availability. But assuming the failure doesn't occur periodically and a long interval between failures, we can achieve almost 99% availability, satisfying TR1.1 requirement.
2. **Fault Tolerance (TR 1.2) :** With experiment 2, we also wanted to verify the fault tolerance of a Kubernetes cluster by intentionally deleting a pod and observing the system's response. Initially, all pods are displayed as running under a staged load (Figure 6). Upon deletion of a pod, the Kubernetes dashboard shows its status as **'Terminating'** (Figure 7), indicating that it's in the process of being shut down. Subsequently, Kubernetes initiated two new pods—one to replace the deleted pod and another due to the increased load, both marked **'Pending'**. This is despite the fact that existing pods were just starting up and hadn't yet received load, as indicated by their CPU and memory usage. Finally, the dashboard reflects the

creation of a new pod post-deletion (Figure 8), affirming the system's capacity to self-heal and scale, thus satisfying the fault tolerance requirement TR1.2.

3. **Autoscaling (TR 2.1):** The HPA has adjusted the number of 'flask-deployment' pods from 1 to 10 based on the CPU utilization targets, allowing the system to respond to changes in workload dynamically. This ensures that the computing resources are scaled according to time-varying workloads, directly satisfying the autoscaling requirement. Here, we want to mention that HPA has a field called '**scaleDownStabilizationWindow**'. It is the time for which k8s waits after decrease in workload to actually scale down the replicas. By default this value is 300s, therefore, we cannot see any immediate decrease in the number of replicas in Figure 6. Hence we can also verify that the application is following the scalability requirement of TR2.1 with constraints of '**Cool Down Period**' of k8s being followed.

3. **Load Balancing:** Following Figure 8 shows how kubernetes is managing load distribution. We can see in the figure below that either pods have the same CPU utilization or load is distributed such that all pods will eventually have the same CPU utilization. (Note, the load balancing operation is performed only with the running pods as new pods were being created due to excessive load and crossing the target CPU Utilization)

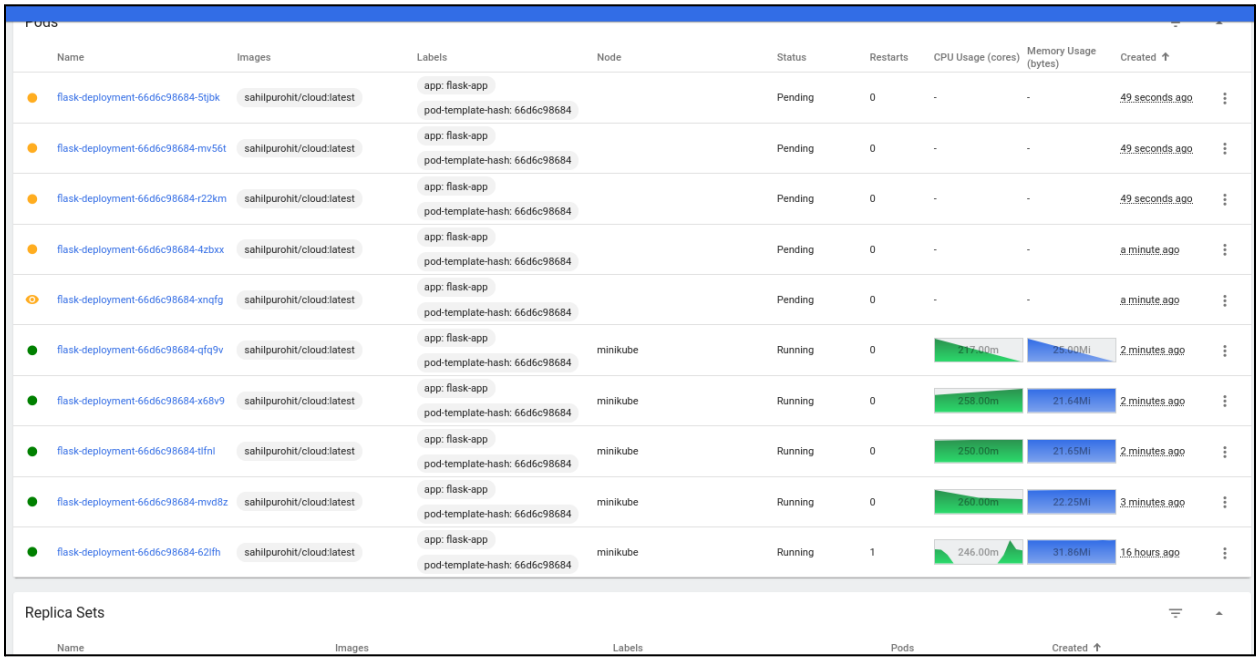


Figure 8: Kubernetes: Load Distribution during a staged load application

4. **Experiment 3 - Network Policy:**

In our experiment, we created 3 namespaces "one" (admin tenant's namespace), "two"

(database tenant namespace), “three” (user tenant’s namespace) with each of them having a single pod running.

```
kubectl -n one exec podone -- curl http://172.16.237.196
kubectl -n three exec podthree -- curl http://172.16.237.196
```

Upon trying to access the pod in the “two” namespace using the pod from the “one” namespace and “three” namespace as shown by the commands given above, the following steps take place:

a. **kubectl -n one exec podone -- curl http://172.16.237.196**: This command uses kubectl to execute a curl command from within a pod named ‘podone’ in the Kubernetes namespace ‘one’ to a pod in namespace ‘two’. The curl command makes an HTTP request to the address ‘http://172.16.237.196’. The ‘-n one’ specifies the namespace of the pod, and ‘exec podone’ tells Kubernetes to run the command inside the ‘podone container’.

b. **kubectl -n three exec podthree -- curl http://172.16.237.196**: Similarly, this command performs a curl request from within a pod named ‘podthree’ in the namespace ‘three’ to a pod in namespace ‘two’. It requests the same HTTP address as the first command.

These commands help test the network connectivity and accessibility of services from within different pods and namespaces within your Kubernetes cluster. Now, we see the results of the above commands.

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   615  100   615    0     0   780k      0 --:--:-- --:--:-- --:--:--  600k
```

Figure 9: Result of accessing pod in NS - 2 with pod from NS - 1

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:--  0:00:11 --:--:--     0^C
```

Figure 10: Result of accessing pod in NS - 2 with pod from NS - 3

Figure 9 shows the successful access of a pod in namespace 2 (NS-2) from a pod in namespace 1 (NS-1). The 100% under both % Total and % Received suggests that the requested data was completely received. The Average Speed Dload is 780k, and the Current Speed at the time the snapshot was taken was 600k. This indicates that the pods in NS 1 and NS 2 communicate successfully.

Figure 10 shows an attempted access to the same pod in NS-2, this time from a pod in namespace 3 (NS-3). The % Total, % Received, and % Xferd values are all 0, which typically means that no data was transferred. The Current Speed is also 0, and there's a small amount of time 00:11 shown under Time Spent, suggesting an attempt was made

but no data was successfully transferred. This could mean that the access was blocked or unsuccessful, due to the network policy that was set up.

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 Conclusions

10.1 The lessons learned

Throughout the course of this project, several key lessons have been learned that are instrumental for future endeavors in cloud-based system deployment and management:

1. Project Structure and Planning: The importance of meticulous planning in the early stages of project selection and requirement discussions became evident. A thorough understanding of Business Requirements (BR) and Technical Requirements (TR) laid a solid foundation for the project.
2. Cloud Service Selection: The process of selecting a cloud provider highlighted the criticality of aligning service offerings with project needs. The exploration of service catalogs to match the project's requirements was a vital step in ensuring the right fit.
3. Iterative Design Approach: Developing initial and subsequent drafts of the application and using cloud formation diagrams reinforced the value of an iterative approach. It allowed for continuous improvement and the ability to adapt the design to emerging insights and validation feedback.
4. Trade-offs and Design Principles: Discussions on trade-offs and adherence to design principles taught the team how to balance factors such as cost, performance, and scalability. These conversations were instrumental in making informed decisions that shaped the final design.
5. Kubernetes Experiments: The hands-on Kubernetes experiments to verify the TRs provided practical insights into the operational aspects of container orchestration and the dynamic nature of cloud resources. The experiments were invaluable in confirming the system's high availability, fault tolerance, autoscaling capabilities, and efficient resource usage monitoring.
6. Validation and Testing: The project underscored the significance of validation at each step. The use of tools like Locust for load testing and Kubernetes' HPA for autoscaling proved essential in testing the system's resilience and performance under simulated real-world conditions.
7. Adaptability and Learning: The project was a testament to the team's adaptability in learning and applying new technologies and concepts, which will be beneficial for tackling future technological challenges.

10.2 Possible continuation of the project

Moving forward, the project can be expanded and enhanced in several ways:

1. Advanced Load Testing: Further load testing can be conducted with increased complexity to simulate more varied traffic patterns and stress test the system's resilience.

2. Multi-Region Deployment: Expanding the deployment to multiple cloud regions could enhance disaster recovery capabilities and global availability.
3. Performance Optimization: Continuous monitoring can yield data that can be used to fine-tune performance, optimize resource usage, and reduce costs.
4. Security Enhancements: Implementing additional security measures and conducting thorough security audits would ensure the system's integrity and protect against emerging threats.

11 References

[1] YV and YP, “ECE547/CSC547 class notes”.
[2] “AWS S3 Wikipedia” (https://en.wikipedia.org/wiki/Amazon_S3)
[3] Seth Eliot and Lara Valverde, “AWS Well Architected Framework” (https://aws.amazon.com/blogs/apn/the-6-pillars-of-the-aws-well-architected-framework/)
[4] “AWS vs Azure vs Google Cloud: Availability Zones” (https://intellipaat.com/blog/aws-vs-azure-vs-google-cloud/#no2)
[5] “Compute compared: AWS vs Azure vs GCP” (https://acloudguru.com/blog/engineering/compute-compared-aws-vs-azure-vs-gcp)
[6] Veritis, “AWS Vs Azure Vs GCP – The Cloud Platform of Your Choice?” (https://www.veritis.com/blog/aws-vs-azure-vs-gcp-the-cloud-platform-of-your-choice/)
[7] “Cloud security comparison: AWS vs. Azure vs. GCP” (https://acloudguru.com/blog/engineering/cloud-security-comparison-aws-vs-azure-vs-gcp)
[8] “Cloud Pricing Comparison: AWS vs. Azure vs. Google Cloud Platform in 2022” (https://cast.ai/blog/cloud-pricing-comparison-aws-vs-azure-vs-google-cloud-platform/)
[9] “AWS Data Exchange for Data APIs” (https://aws.amazon.com/data-exchange/why-aws-data-exchange/apis/)
[10] “AWS EC2” (https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html)
[11] “AWS Autoscaling” (https://aws.amazon.com/autoscaling/)
[12] “AWS Cloudwatch” (https://aws.amazon.com/cloudwatch/)
[13] “Amazon ELB” (https://aws.amazon.com/elasticloadbalancing/)
[14] “AWS RDS” (https://aws.amazon.com/rds/)
[15] “Amazon IAM” (https://aws.amazon.com/iam/)
[16] “AWS S3 Wikipedia” (https://en.wikipedia.org/wiki/Amazon_S3)
[17] “AWS CloudFormation” (https://aws.amazon.com/cloudformation/)
[18] “AWS KMS” (https://aws.amazon.com/kms/)
[19] “Amazon GuardDuty” (https://aws.amazon.com/guardduty/)
[20] “AWS API Gateway” (https://aws.amazon.com/api-gateway/)
[21] “Amazon VPC” (https://docs.aws.amazon.com/vpc/latest/userguide/what-is-amazon-vpc.html)
[22] “AWS Lambda” (https://aws.amazon.com/lambda/)
[23] “AWS Cloudfront” (https://aws.amazon.com/cloudfront/)
[24] “AWS Cost and Usage Reports” (https://docs.aws.amazon.com/cur/latest/userguide/what-is-cur.html)
[25] “AWS Elastic Disaster Recovery” (https://aws.amazon.com/disaster-recovery/)
[26] “AWS Well Architected Framework” (https://aws.amazon.com/architecture/well-architected/)
[27] “Network Requirements for Latency - Critical Services in a Full Cloud Deployment” (https://www.researchgate.net/publication/309292016_Network_Requirements_for_Latency-

Critical_Services_in_a_Full_Cloud_Deployment)
[28] “Iaas vs Paas vs Saas” https://www.ibm.com/cloud/learn/iaas-paas-saas)
[29] “Text Summary Generator Project” by Palash Jhamb, Siddhant Gupta, Prateek Wadhwani
[30] “Design Principles for Operational Excellence” (https://docs.aws.amazon.com/wellarchitected/latest/framework/oe-design-principles.html)
[31]“Design Principles for Security” (https://docs.aws.amazon.com/wellarchitected/latest/framework/sec-design.html)
[32] “Design Principles for Reliability” (https://docs.aws.amazon.com/wellarchitected/latest/framework/rel-dp.html)
[33]“Design Principles for Performance Efficiency” (https://docs.aws.amazon.com/wellarchitected/latest/framework/perf-dp.html)
[34] “Design Principles for Cost Optimization” (https://docs.aws.amazon.com/wellarchitected/latest/framework/cost-dp.html)
[35] “Design Principles for Sustainability” (https://docs.aws.amazon.com/wellarchitected/latest/framework/sus-design-principles.html)
[36] “Kubernetes Wikipedia” (https://en.wikipedia.org/wiki/Kubernetes)
[37] Janakiram MSV, “Kubernetes Working” (https://thenewstack.io/how-does-kubernetes-work/)
[38]“MiniKube_Lab_Manual by Subhashini”
[39]“Locust Documentation” (https://docs.locust.io/en/stable/what-is-locust.html)
[40] “Kubernetes Network Policy Tutorial - yaml explained + Demo Calico” (https://www.youtube.com/watch?v=u1KUft3fsCk)