Cloud Computing Technology
CSC/ECE 547 - Fall 2023

Integrated Content Catalog

Group Members:
Pranav Manbhekar (ppmanbhe; 200538542)
Shruti Dhond (sdhond; 200538334)
North Carolina State University

Plagiarism Declaration

"We, the team members, understand that copying & pasting material from any source in our project is an allowed practice; we understand that not properly quoting the source constitutes plagiarism.

All team members attest that we have properly quoted the sources in every sentence/paragraph we have copy & pasted in our report.

We further attest that we did not change words to make Copy & pasted material appear as our work."

1 Introduction

1.1 Motivation

Our primary motivation for undertaking this project is to leverage the knowledge and concepts learned from our Cloud Computing course into architecting a real-world cloud application. Our decision to choose this particular topic stems from the real challenges we encountered when navigating a vast volume of online streaming content dispersed across multiple platforms.

1.2 Executive summary

In today's digital era, an abundance of video content is accessible through various online streaming platforms such as Netflix, Amazon Video, Hulu, and many more. While the range of viewing options is vast, users often find themselves rummaging through multiple platforms to locate the specific content they desire, whether it's a movie or a web series. Furthermore, content is typically distributed across these platforms with minimal overlap, complicating the user's search for their preferred content.

To address this issue, we propose the development of an Integrated Digital Content Catalog. This catalog will aggregate video content from all major streaming platforms, presenting it to users as a unified repository. Users will benefit from a one-stop platform, where they can effortlessly explore a comprehensive selection of content and make their viewing choices. Once a selection is made, the platform will seamlessly redirect the user to the corresponding streaming service for playback. The platform will also provide unbiased content recommendations by analyzing the user’s past ratings and viewing behaviors.

2 Problem Description

2.1 The problem

To design an online streaming content catalog service that provides a complete catalog of all movies and TV shows from major OTT platforms. This application does not provide streaming services, but only presents the aggregated catalog and redirects users to the original platform if they wish to stream recommended content. Personalized recommendations are provided to users based on their watch history, search, and rating patterns.

2.2 Business Requirements

- BR 1: Highly adaptable to real-time changes across all content providers
- BR 2: Be available 24 x 7
- BR 3: Provide high performance
- BR 4: Accommodate time-varying workloads
- BR 5: Optimize costs
- BR 6: Secure operations in the application
- BR 7: Accurate recommendations of content
- BR 8: Tenant Identification
- BR 9: Avoid vendor lock-in
- BR 10: High scalability to accommodate ever-growing content catalog and user base
- BR 11: Provide region-wise content support with wide-area geographical coverage
- BR 12: Easy integration with third-party applications
- BR 13: Securely store user data and content data
- BR 14: Achieve operational excellence
- BR 15: Unbiased content featured in the catalog (Software Architect BR)
- BR 16: Organized, simplified, and curated catalog for each user (Software Architect BR)
- BR 17: Monitoring user actions, selections, and engagement (Software Architect BR)
- BR 18: Accurate & powerful search results for content (Software Architect BR)

2.3 Technical Requirements

We now present the technical requirements identified for each of the BRs.

- TR 1.1: Continuously sync catalog changes like new shows/movies added or deleted across all major content providers
  (BR1)

- TR 2.1: The system availability should be 99%
  (BR2)

- TR 2.2: Deploy the applications in multiple availability zones across different locations
  (BR2)

- TR 2.3: Ensure the system is fault tolerant and automate failure recovery using metrics and alarms
  (BR 2)

- TR 2.4: Implement periodic data backups and disaster recovery plans to prevent data loss
  (BR 2)

- TR 1.2, 3.1, 4.1: The resource capacity for CPU, memory, and storage should rapidly grow and shrink according to varying workloads
  (BR1, 3, 4)

- TR 3.2: Latency to load the catalog should not be more than 1 second
  (BR3)

- TR 1.3, 3.3, 5.1: Use efficient & cost-effective data extraction techniques
  (BR 1, 3, 5)

- TR 5.2: Implement robust monitoring and analytics tools to analyze resource usage, daily traffic, and associated costs, and configure resource usage limit.
  (BR 5)

- TR 6.1: Implement identity access management policies for all the resources
  (BR 6)

- TR 6.2: Protect user data from unauthorized access
  (BR 6)

- TR 7.1: Implement analysis tool for tenants' viewing patterns for a strong recommendation model
  (BR 7)

- TR 6.3, 8.1, 13.1: Identify the tenant and authorize access
  (BR 6, 8, 13)

- TR 10.1: The system should handle a large number of users and scale with the growing content data and number of users
- (BR 10)

- TR 7.2, 11.1: Deploy the application in various geographical locations for wide regional content support
  (BR 7, 11)

- TR 8.2: Serve multiple tenants at the same time
  (BR 8)

- TR 12.1: Provide the ability to connect to third-party applications by offering APIs for smooth integration, partnership with content providers for seamless content access
  (BR 12)

- TR 6.4, 13.2: Store user data, all the content data, metadata, and logs in appropriate secure storage
  (BR 6, 13)

- TR 1.2, 2.5, 4.2, 5.3 14.1, 9.1: Use managed container orchestration tools for automated deployment & management
  (BR 1, 2, 4, 5, 14, 9)

- TR 11.2: Ensure that content is accessed and displayed in compliance with licensing agreements and copyright laws for each content provider (BR 11)

- TR 2.6, 3.4, 4.3: Distribute load across multiple servers to increase performance and improve fault tolerance (BR 2, 3, 4)

BRs	TRs	Justification
BR 1	TR 1.1, 1.2, 1.3	● For the content catalog to be updated all the time, the design should regularly sync changes from content providers(1.1) ● Elasticity (1.2) & high performance (1.3) are required to achieve a regularly updated state of the catalog at all times
BR 2	TR 2.1, 2.2, 2.3, 2.4, 2.5, 2.6	● A metric is required to evaluate BR2, and it is stated in TR 2.1 ● Deploying applications in multiple locations and availability zones & making the application fault-tolerant will help to achieve the 24x7 availability goal. (TR 2.2, 2.3) ● Enabling automated failure recovery(2.3), data backups & disaster recovery(2.4) helps in quick issue identification and quick recovery from failures, and restoring the stable state after a disruptive event that maintains continuous availability ● Container orchestration tools can automate the distribution of containers, ensuring that applications remain available even in case of failures. (2.5) ● The distribution of load across multiple servers enhances fault tolerance by reducing downtime and contributes to high availability (2.6)
BR 3	TR 3.1, 3.2, 3.3, 3.4	● Elastic nature to scale up and down quickly is essential for high-performance at all times since there can be varying workloads (TR 3.1) ● TR 3.2 mentions metrics to evaluate the goal of measuring high-performance requirements

		• We need efficient data extraction methods like serverless computing to curate the catalog to ensure high performance at all times (TR 3.3) • Distribution of equal loads on servers results in optimal performance on each server leading to high performance & we can server large number of requests (TR 3.4)
BR 4	TR 4.1, 4.2, 4.3	• To accommodate time-varying workloads means the application should have elasticity, precisely the resources should rapidly scale up and down based on the traffic, so our system can perform as it is supposed to (TR 4.1) • Container orchestration tools enable automatic scaling of resources based on demand, ensuring that the infrastructure can handle increased loads during peak times and scale down during periods of lower activity (TR 4.2) • Distributing the load across multiple servers allows the system to scale horizontally during peak and effective resource utilization during non-peak times (TR 4.3)
BR 5	TR 5.1, 5.2, 5.3	• Using cost-effective data extraction methods including appropriate storage solutions and processing services like serverless computing(FaaS) can reduce operational costs(TR 5.1) • Analyzing daily traffic patterns to optimize scaling strategies can prevent over-provisioning and reduce costs. By monitoring resource usage patterns & identifying underutilized resources, we can set resource limits and optimize resource allocation, and reduce additional costs(TR 5.2) • Automated scaling ensures that resources are dynamically adjusted, which can result in cost savings (TR 5.3)
BR 6	TR 6.1, 6.2, 6.3, 6.4	• Implementing IAM policies can restrict resource access and unauthorized users cannot access any resource making operations secure(TR 6.1) • Protecting user data is essential from security perspective (TR 6.2) • We should be able to identify tenants (TR 6.3) & store each tenant's files securely & separately in appropriate storage (TR 6.4)
BR 7	TR 7.1, 7.2	• Analyzing each user's viewing pattern can improve the content recommendations over time (TR 7.1) • Recommendations should consider regional restrictions for accuracy (TR 7.2)
BR 8	TR 8.1	We should be able to identify tenants and authorize access to meet various other requirements of privacy, security, and resource management
BR 9	TR 9.1	Choosing container orchestration tools that support open standards and are not tied to a specific cloud provider provides flexibility and the ability to migrate workloads across different clouds if necessary

BR 10	TR 10.1	The number of users can increase over time and the content catalog is ever-increasing. The application should be able to scale smoothly over time
BR 11	TR 11.1, 11.2	• The application needs to be deployed in various geographical locations to provide region-wise support. (TR 11.1) • For accurate content support, we need to comply with license and copyright agreements with all providers (TR 11.2)
BR 12	TR 12.1	We need to expose APIs for smooth integration with third-party apps. Along with this, partnerships with content providers will enable better content exposure and agreements
BR 13	TR 13.1, 13.2	We must identify a tenant (TR 13.1) and store each tenant's data, metadata, and logs separately and securely in appropriate storage services (TR 13.2).
BR 14	TR 14.1	Automation provided by container orchestration tools reduces manual intervention in tasks such as deployment, scaling, and monitoring, leading to increased operational efficiency

2.4 Tradeoffs

- High content adaptability (BR 1) vs Cost (BR 5)

  Synchronizing the application's content catalog with all content providers continuously requires continuous pulling methods like hosting a cloud function which can incur heavy additional costs
- Accurate predictions (BR 7) vs Data privacy (BR 13)

  To provide better predictions using data mining or other recommendation systems, the user's data and additional metadata will be used which can raise privacy concerns
- Content Aggregation (BR 7) vs Compliance policies (BR 11)

  The content catalog should be comprehensive of all the listings from content providers, however due to regional restrictions, the catalog can be restricted or limited across different regions
- Scalability (BR 10) vs Cost (BR 5)

  To achieve high scalability, the infrastructure cost will be higher which compromises the cost optimization goal. Adequate architectural decisions can achieve the balance.
- Security (BR 6) vs Performance (BR 3)

  Additional security methods like secure data storage and IAM policies can increase the latency and response time, which could decrease the overall performance.
- Regional content support (BR 10) vs Cost (BR 5)

  To provide regional content support for all regions globally, the application must be deployed in multiple locations to comply with regional restrictions that would lead to redundancy in resources and increase overall infrastructure cost.

3 Provider Selection

3.1 Criteria for choosing a provider

1. Services offerings: The provider should have all the services in their service catalog to meet our TRs including specialized services like AI/ML-based recommendation models
2. Pricing: The pricing of services of our interest should be competitive, and monitoring cost metrics should be available
3. Geographical presence: It is essential that all service offerings from the provider should be available in all locations worldwide since our application needs to deal with regional restrictions
4. Data storage: The provider should offer data storage for different data models to avoid design restrictions and provide flexibility
5. Expertise/Experience of the team: Team members should be aware and comfortable with services and having prior experience could ease the design & development process
6. SLA: Agreements offered by providers should align with our requirements and be acceptable
7. Reputation: The provider should be well-reputed in this industry
8. Exit strategies: There should be ways to mitigate vendor lock-in and migration of data outside of the cloud should be easy

3.2 Provider Comparison

	AWS		**GCP**		**Azure**	
Criteria	Justification	Rank	Justification	Rank	Justification	Rank
Service Offerings	Has over 200+ services	1	Has around 100 service offerings	3	Has largest catalog after AWS	2
Pricing	Offers discounts up to 72% for commitment models. Overall storage options offered are cheaper compared to others	1	Offers discounts up to 70% for commitment models	2	Offers discounts up to 65% for commitment models	3
Geographical Presence	AWS Cloud spans 102 Availability Zones within 32 geographic regions around the world	3	GCP offers 36 regions with 109 availability zones	2	Azure offers 60+ regions and 113 availability zones	1
Data Storage	Offers various storage options along with	1	Offers various storage options along with	1	Offers various storage options along with relational	1

	relational & managed NoSQL database that suits our requirements		relational & serverless NoSQL database that suits our requirements		& distributed NoSQL databases that suits our requirements	
Expertise/ Experience of the team	Most experience of working with various AWS services	1	Basic experience of working with some GCP services	2	Basic experience of working with some Azure services	2
SLA	For downtime 95%-99% in computer, service credits offered are 30%, and monthly uptime of 99.99%. For storage services, credits offered by AWS are lucrative	1	For downtime 95%-99% in compute, service credits offered are 25%, and monthly uptime of 99.99%	2	For downtime 95%-99% in compute, service credits offered are 25%, and monthly uptime of 99.99%	2
Reputation	AWS is widely regarded as the market leader and has been in the cloud computing industry for a longer period than its competitors	1	GCP has a smaller market share compared to others but known for its innovation and contributions to open-source technologies. It is often considered developer-friendly	1	Azure has 2nd largest market share and is often seen as a strong choice for enterprises, especially those already using Microsoft technologies	1
Exit Strategies	AWS offers support for various migration services and container orchestration tools facilitating application migration	1	GCP is known for its commitment to open standards and has a strong focus on Kubernetes, and also provide support for transfer services	1	Azure has a strong focus on hybrid cloud scenarios, well-suited for on-premises infrastructure. They also offers Azure Migrate service	1

3.3 The final selection

There are some aspects where all 3 providers have a tie. However, we have decided to go with AWS as our cloud provider since it has a large service catalog, competitive pricing model, suitable SLAs, and we are familiar with AWS. It also scored the highest average rank.

3.3.1 The list of services offered by the winner

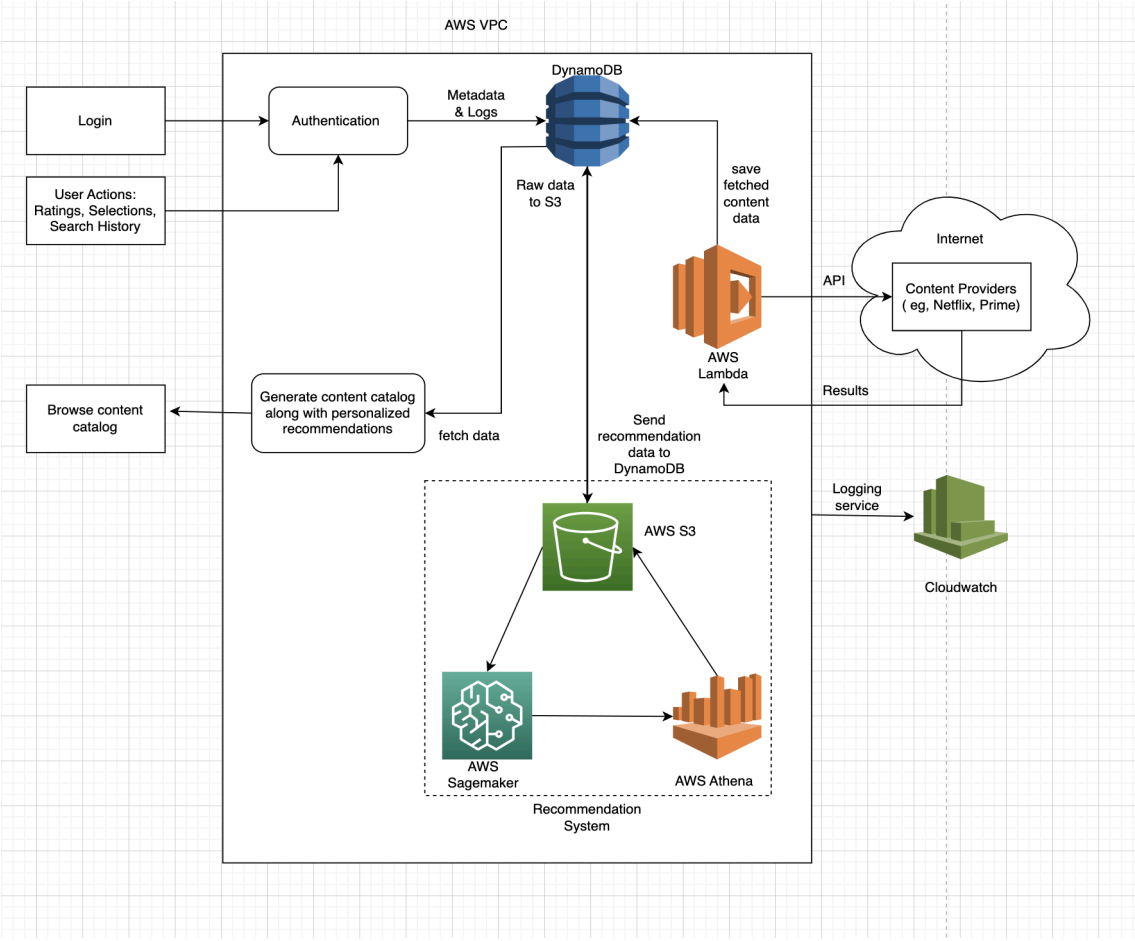
- TR 1.1: AWS Lambda, EventBridge
- TR 2.1: AWS Cloudwatch, EKS, ELB
- TR 2.2: AWS EKS, VPC
- TR 2.3: AWS CloudWatch Alarms, AWS Lambda (for executing automated recovery)
- TR 2.4: AWS Backup
- TR 1.2, 3.1, 4.1: AWS EKS
- TR 3.2: AWS EC2, DynamoDB
- TR 1.3, 3.3, 5.1: AWS Lambda, EventBridge
- TR 5.2: AWS Cloudwatch
- TR 6.1: AWS Identity and Access Management (IAM)
- TR 6.2: AWS VPC, EC2, S3, DynamoDB
- TR 7.1: AWS Athena, AWS SageMaker, AWS S3
- TR 6.3, 8.1, 13.1: Amazon Cognito
- TR 10.1: AWS EC2, DynamoDB, EKS
- TR 7.2, 11.1: AWS Multi-region deployment for all services
- TR 8.2: AWS EC2, EKS, ELB
- TR 12.1: Amazon API Gateway
- TR 6.4, 13.2: AWS DynamoDB
- TR 1.2, 2.5, 4.2, 5.3 14.1, 9.1: Amazon EKS
- TR 11.2: AWS Cloudwatch
- TR 2.6, 3.4, 4.3: AWS ELB

4 The first design draft

We are selecting the following TRs for our design:

- TR 1: Continuously sync catalog changes like new shows/movies added or deleted across all major content providers
- TR 2: The system availability should be 99%
- TR 3: Deploy the applications in multiple availability zones across different locations (AWS WAF: Reliability)
- TR 4: Identify the tenant and authorize access (Tenant Identification TR)
- TR 5: Implement robust monitoring and analytics tools to analyze resource usage, daily traffic, and associated costs, and configure resource usage limits. (Monitoring TR) (conflicts with TR 3)
- TR 6: The resource capacity for CPU, memory, and storage should rapidly grow and shrink according to varying workloads (Elasticity TR)
- TR 7: Distribute load across multiple servers to increase performance and improve fault tolerance (Load Balancing TR) & (AWS WAF: Performance Efficiency)
- TR 8: Implement analysis tool for tenants' viewing patterns for a strong recommendation model
- TR 9: Store user data, all the content data, metadata, and logs in appropriate secure storage (conflicts with TR 8) (AWS WAF: Performance Efficiency)
- TR 10: Latency to load the catalog should not be more than 1 second (conflicts with TR 9)
- TR 11: Ensure that the data and network traffic is encrypted (AWS WAF: Security)

This is the initial version of our design:



4.1 The first building blocks

1. **Storage**
   i. AWS S3: For storing content thumbnails for displaying on the catalog for a better viewing experience. S3 is also required for running SQL queries using Athena.
   ii. AWS DynamoDB: Storing user-specific recommendations, user information, and logs

2. **Compute**
   i. Amazon Elastic Kubernetes Service (EKS): EKS handles the Kubernetes Control Plane in AWS VPC so scaling, and backups are handled automatically
   ii. Amazon EC2: used together with EKS to run Kubernetes worker nodes in the data plane.

3. **Networking**
   i. Amazon Elastic Load Balancer: Used to distribute the load from multiple users to different server nodes for computation.
   ii. AWS VPC: To provide an isolated virtual environment for running EKS worker nodes. The internet gateway will be used to access the VPC from the internet for server user requests

4. **Others**

   i. AWS Lambda + AWS EventBridge: To periodically run a job that will fetch the latest updates and changes from all the content platforms and update the same in DynamoDB.

   ii. AWS CloudWatch: Storing and analyzing logs

   iii: AWS Athena: To run analytical queries to learn and gather knowledge about the usage patterns stored in S3 storage required for training ML models.

   iv: Amazon Sagemaker: To simplify the process of building, training, and deploying ML models at scale get data generated from Athena and answer recommendations.

4.2 Top-level, informal validation of the design

We present arguments that the above-mentioned solution will work.

1. **Storage:**

   i. AWS S3 for storing thumbnails: AWS S3 is an object storage that offers scalable, secure, and low-latency storage, making it well-suited for hosting and serving image data in applications. We will also need S3 for running SQL queries using Athena which is required to generate training data for the recommendation system.

   ii. AWS DynamoDB for storing user-specific information and logs: After the aggregating and clustering of the data, the recommendation results for each user will be stored in AWS DynamoDB and it provides low-latency data access, making the application fast and responsive.

2. **Compute:**

   An EKS cluster can be configured to use EC2 instances as worker nodes that run the containerized applications. In an Amazon EKS (Elastic Kubernetes Service) environment, EC2 instances running as worker nodes are organized into Auto Scaling groups. These Auto Scaling groups dynamically adjust the number of EC2 instances based on scaling policies and triggers, responding to changes in demand. Kubernetes Horizontal Pod Autoscaling (HPA) automatically adjusts the number of pod replicas based on workload. Thus, the application will scale up and scale down based on the number of users, which provides elasticity.

3. **Networking:**

   i. Amazon EKS (Elastic Kubernetes Service) is used with AWS ALB (Application Load Balancer), where the ALB is typically used as the ingress controller to distribute incoming traffic to the EC2 worker nodes running in the EKS cluster.

   ii. AWS VPC provides an isolated virtual environment for running EKS worker nodes. We will attach an Internet Gateway to the VPC to allow communication between instances in the VPC and the Internet.

4. **Other:**

i. We will schedule AWS Lambda functions using EventBridge. The code to aggregate the changes from different platforms will be packaged in an AWS Lambda function which will be triggered periodically from the AWS EventBridge rule
ii. The user action data from S3 will be queried using Athena to generate the data structure of use for ML models. The ML recommendation model will be trained in Amazon Sagamaker and then will be deployed. Using Athena, recommendation queries will run to generate personalized recommendations for each user.
iii. AWS CloudWatch will be used to store logs, generate metrics, and trigger alarms.

4.3 Action items and rough timeline

Skipped

5 The second design

5.1 Use of the Well-Architected framework

Distinct steps suggested by the AWS Well-Architected framework [22]:

1. Operational Excellence:
This pillar refers to the proficiency in supporting the development and operation of workloads effectively, gaining operational insights, and perpetually enhancing supporting processes to deliver business value. To incorporate this, we have added mechanisms to continuously monitor and analyze performance (such as CloudWatch metrics) to gain insights and improve the existing processes.

2. Security:
This pillar outlines the utilization of cloud technologies to safeguard data, systems, and assets, thereby enhancing overall security posture. We have incorporated various AWS services that provide data security and encryption (at rest or in transit) such as VPC, EC2, S3, and DynamoDB.

3. Reliability:
Encompassing a workload's ability to consistently and correctly perform its intended function, including operations and testing throughout its lifecycle. We have incorporated this by deploying applications into multiple availability zones which will improve reliability using redundancy.

4. Performance Efficiency:
The capacity to utilize computing resources effectively to meet system requirements, maintaining efficiency amidst changing demand and evolving technologies. We have designed our application to distribute load across multiple servers to increase performance and improve fault tolerance.

5. Cost Optimization:
The proficiency in running systems to deliver business value at the most economical price point. We will implement robust monitoring and analytics tools to analyze resource usage so that the application will deploy only the necessary amount of resources, in order to keep costs at the minimum.

6. Sustainability:
The capability to continually enhance sustainability impacts by reducing energy consumption and increasing efficiency across all workload components. This is achieved by maximizing benefits from provisioned resources and minimizing total resource requirements.

5.2 Discussion of pillars

The AWS well-architected framework provides a set of best practices for designing and operating reliable, secure, efficient, and cost-effective systems in the cloud. The framework provides a consistent approach to evaluating systems against the qualities expected from modern cloud-based systems, and the remediation that would be required to accomplish those qualities. This is achieved by documenting a set

of foundational questions that allow an understanding of a specific architecture to align well with cloud best practices.

The AWS Well-Architected Framework is based on six pillars:
1. Operational Excellence
2. Performance Efficiency
3. Cost Optimization
4. Reliability
5. Security
6. Sustainability

We now discuss 2 pillars in detail.

1. Operational Excellence[7]

The Operational Excellence pillar includes the ability to support development and run workloads effectively, gain insight into their operations, and to continuously improve supporting processes and procedures to deliver business value. It provides an overview of design principles, best practices, and questions.

Design Principles

There are five design principles for operational excellence in the cloud:

1. Perform operations as code
2. Make frequent, small, reversible changes
3. Refine operations procedures frequently
4. Anticipate failure
5. Learn from all operational failures

Perform operations as code: In the cloud, you can apply the same engineering discipline to your environment as you use it for application code. You can define your entire workload (infrastructure, applications, etc) as code. With operations as code, you can script the operations and automate their execution by triggering events. By using performing operations as code, you can minimize human error and allow consistency in the operations.

Make frequent, small, reversible changes: The design should adapt to changes and updations if needed. In order to achieve this, make changes in small increments that can be reversed if they fail.

Refine Operations Procedures Frequently: As the operations and workload proceed, they should be continuously evaluated and improved. This can be achieved by setting up regular “game days” to review all the processes and make sure the team is familiar with them.

Anticipate failure: Potential sources of failures should be identified through “pre-mortem” exercises so that they can be removed or mitigated. Test failure scenarios and validate the understanding of their

impact. Test response procedures to ensure that they are effective and that teams are familiar with their execution.

Learn from all operational failures: Drive improvement through lessons learned from all operational events and failures. Share what is learned across teams and through the entire organization.

AWS mentions four ways to achieve operational excellence in the cloud:
● Organization
● Prepare
● Operate
● Evolve

Organization:
The teams need to have a shared understanding of the entire workload and well-defined priorities. teams must understand their part in achieving business outcomes. Teams need to understand their roles in the success of other teams, the role of other teams in their success, and have shared goals.Ensure that there are identified owners for each application, workload, platform, and infrastructure component and that each process and procedure has an identified owner responsible for its definition, and owners responsible for their performance. AWS has mentioned some questions that focus on these considerations for operational excellence:

- How do you determine what your priorities are?
- How do you structure your organization to support your business outcomes?
- How does your organizational culture support your business outcomes?

Prepare:
To achieve operational excellence, it is crucial to design workloads that provide insight into their status, implement procedures to support them and adopt approaches that improve the flow of changes into production while enabling rapid feedback on quality and recovery. Additionally, it is essential to evaluate operational readiness and utilize operations as code to view the entire workload as code, allowing for consistent and controlled experimentation. AWS has mentioned some questions that focus on these considerations for operational excellence:

- How do you design your workload so that you can understand its state?
- How do you reduce defects, ease remediation, and improve flow into production?
- How do you reduce defects, ease remediation, and improve flow into production?
- How do you mitigate deployment risks?
- How do you know that you are ready to support a workload?

Operate:
To measure the successful operation of a workload, define expected outcomes, determine how success will be measured, and establish metrics baselines. Manage operational events efficiently by using runbooks and playbooks, prioritizing responses based on their impact. Communicate the operational

status of workloads through dashboards and notifications tailored to the target audience. AWS has mentioned some questions that focus on these considerations for operational excellence:

- How do you understand the health of your workload?
- How do you understand the health of your operations?
- How do you manage workload and operations events?

Evolve:
To sustain operational excellence, continuously improve by dedicating work cycles to improvements, performing post-incident analysis, evaluating opportunities for improvement, and including feedback loops. Share lessons learned across teams and analyze trends within lessons learned to identify opportunities for improvement. Implement changes and evaluate their success. “How do you evolve operations?” is a question that can be asked to focus on these considerations.

2. Performance Efficiency[8]

The Performance Efficiency pillar includes the ability to use computing resources efficiently to meet system requirements and to maintain that efficiency as demand changes and technologies evolve.
It provides an overview of design principles, best practices, and questions.

AWS mentions five design principles for performance efficiency in the cloud:

1. Democratize Advanced Technologies
2. Go global in Minutes
3. Use Serverless Architectures
4. Experiment More Often
5. Consider Mechanical Sympathy

Democratize Advanced Technologies: To make advanced technology implementation easier for the team, consider consuming the technology as a service. This means that to delegate complex tasks to cloud vendors, such as hosting and running NoSQL databases, media transcoding, and machine learning. This will allow the team to focus on product development rather than resource provisioning and management.
Go global in Minutes: Deploying your workload in multiple AWS Regions Around world provides lower latency and a better experience for customers at minimal cost.

Use Serverless Architectures: Serverless architectures eliminate the need to run and maintain physical servers for traditional computing activities. This removes the operational burden of managing physical servers and can lower transaction costs because managed services operate at cloud scale. Serverless storage services can act as static websites (removing the need for web servers) and event services can host code.

Experiment More Often: With virtual and automatable resources,you can quickly carry out comparative testing using different types of instances, storage, or configurations.

Consider Mechanical Sympathy: Understand how cloud services are consumed and always use the technology approach that aligns best with your workload goals.

AWS mentions four best practice areas for performance efficiency in the cloud:
• Selection
• Review
• Monitoring
• Trade-Offs

Selection:
The optimal solution for a particular workload varies, and solutions often combine multiple approaches. Well-architected workloads use multiple solutions and enable different features to improve performance. The question "How do you select the best performing architecture?" be asked to focus on these considerations for performance efficiency.

i. Compute:
When selecting compute resources, it is important to consider your workload performance and cost requirements. By choosing the compute option that best meets your needs, you can optimize your resource utilization and achieve greater efficiency. The question "How do you select your compute solution?" can be asked to compute related solutions considering performance efficiency.

ii. Storage:
Cloud storage is an essential aspect of cloud computing, housing the data utilized by your workload. It offers enhanced reliability, scalability, and security compared to conventional on-premises storage solutions. Choose from object, block, and file storage services, along with cloud data migration options, to suit your workload's needs. The question "How do you select your storage solution?" can be asked for storage-related solutions considering performance efficiency.

iii. Database:
The cloud provides a diverse range of purpose-built database services tailored to specific workload requirements. Choose from various database engines, including relational, key-value, document, in-memory, graph, time series, and ledger databases. By selecting the most suitable database for a particular problem or group of problems, you can break free from the limitations of monolithic, one-size-fits-all databases and focus on developing applications that meet the performance expectations of your customers. The question "How do you select your database solution?" can be asked for database-related solutions considering performance efficiency.

iv. Network:
The network plays a critical role in cloud computing, impacting workload performance and behavior. Workloads like High-Performance Computing (HPC) are heavily reliant on network performance, making network understanding crucial for optimizing cluster performance. Network requirements for bandwidth, latency, jitter, and throughput must be determined to ensure optimal

workload performance. The question “How do you select your networking solution?” can be asked for networking-related solutions considering performance efficiency.

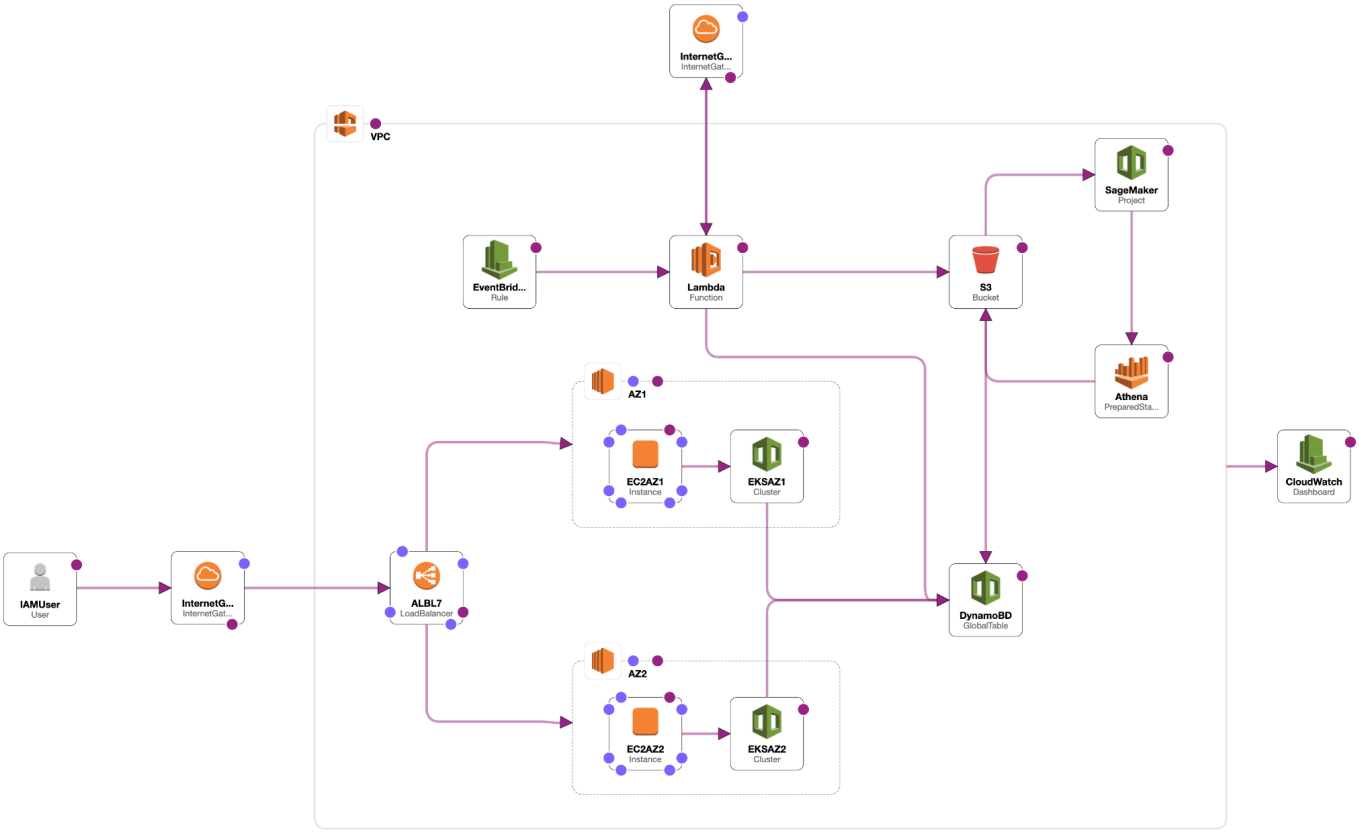
Review:
As cloud technologies advance, it's crucial to adapt your workload components to leverage the latest advancements and optimize performance. Continuously evaluate and consider changes to ensure you meet performance and cost goals. Emerging technologies like machine learning and artificial intelligence (AI) can revolutionize customer experiences and drive innovation across your business workloads. The question “How do you evolve your workload to take advantage of new releases?” focuses on these considerations for performance efficiency.

Monitoring:
After you implement your workload, you must monitor its performance so that you can remediate any issues before they impact your customers. Monitoring metrics should be used to raise alarms when thresholds are breached.

The question “How do you monitor your resources to ensure they are performing?” can be asked for monitoring-related solutions considering performance efficiency. The question “How do you monitor your resources to ensure they are performing?” focuses on these considerations for performance efficiency.

Tradeoffs:
Tradeoffs must be evaluated to ensure an optimal approach. Depending on the situation, there could be a tradeoff between consistency, durability, and space for time or latency, to deliver higher performance. The question “How do you use tradeoffs to improve performance?” focuses on these considerations for performance efficiency.

5.3 Use of Cloudformation Diagrams



5.4 Validation of the design

We now present a justification of how our design aims to achieve the TRs mentioned in section 4.

TR 1: One of the fundamental requirements of a movie catalog is the timely (almost real-time) integration of new offerings/updates provided by all the content providers. To achieve this, we need to constantly query the APIs provisioned by all the content providers and keep syncing any updates to our database. This can be implemented by having a cron-job-like behavior, where we would periodically query the APIs for changes and update the same in our database. AWS lambda is an optimal service for achieving this as it is a serverless, event-driven compute service that enables users to run code without provisioning or managing infrastructure. We can have a code logic, packaged as an AWS Lambda function, which will be responsible for fetching changes and updates from content providers by making API calls and updating them in our database. Now, we would also need a scheduler of an "event" to trigger the AWS Lambda function periodically. For this, we can use an EventBridge Rule that can run in response to an event, or at certain time intervals. We would need a rule type as "Schedule", that will always run once within the specified time interval, and choose the Lambda function as the target. [11]

TR 2: We aim to achieve 99% availability through the following:[12]

- We will use metrics and alarms to constantly monitor new updates and the overall health of the application. AWS Cloudwatch can be used here, which is a monitoring service that collects and tracks metrics from AWS resources and custom applications. It enables users to set alarms based on predefined thresholds, facilitating proactive management and automated responses to potential issues. Setting up proper alerts helps keep a check on the overall status of the application. In case the alarms trigger, it enables us to take quick action and damage control, thus keeping the availability of the system under control.
- Amazon EKS facilitates automatic scaling of EC2 instances within Kubernetes clusters by integrating with Auto Scaling Groups. Users can define and manage the desired number of worker nodes, and EKS dynamically adjusts the EC2 instances based on workload demands. Thus, auto-scaling EC2 instances helps in adapting to time-varying loads, keeping the overall availability high. EKS also helps to gracefully roll back faulty or buggy changes, making sure that there is minimum downtime.
- Amazon Elastic Load Balancer (ALB) enhances availability by distributing incoming traffic across multiple nodes and pods, preventing overloads and automatically redirecting to healthy instances.

TR 3: EKS helps to deploy, manage, and scale containerized applications using Kubernetes. When deploying applications across multiple availability zones, EKS facilitates the distribution of Kubernetes worker nodes across those zones, ensuring redundancy and fault tolerance. VPC provides a logically isolated section of the AWS Cloud to launch AWS resources. AWS VPC empowers users to define their own IP address range, create subnets in various availability zones, and configure route tables and network gateways. This capability enables them to construct a resilient network architecture that spans multiple locations.[1] [13] [14]

TR 4: AWS also allows for tenant identification and isolation through Identity and Access Management(IAM). Using this, the access for each resource can be configured on a per-tenant basis. We can write policies for each tenant for tenant isolation.

TR 5: AWS CloudWatch facilitates robust monitoring and analytics by collecting metrics, setting alarms for resource thresholds, and offering customizable dashboards. It enables analysis of daily traffic and resource usage, integrates with logs for troubleshooting, and supports resource usage limit configurations to prevent over utilization. CloudWatch's integration with Cost Explorer allows users to analyze and optimize costs associated with resource usage. [18]

TR 6: AWS EKS helps with dynamic resource scaling by automatically adjusting the capacity for CPU, memory, and storage based on varying workloads. EKS supports HPA (Horizontal Pod Autoscaler), a Kubernetes feature that automatically adjusts the number of running instances (pods) based on observed CPU utilization or other custom metrics. This ensures that the application scales horizontally to meet changing performance demands. EKS also utilizes AWS Auto Scaling Groups to dynamically add or remove EC2 instances in response to demand. This ensures that the overall compute capacity of the cluster scales in or out based on the application's resource requirements. [13]

TR 7: AWS Elastic Load Balancing (ELB) helps distribute load across multiple servers, enhancing performance and fault tolerance. ELB evenly distributes incoming traffic across multiple servers or instances, ensuring that no single server is overwhelmed. ELB automatically detects unhealthy instances and redirects traffic to healthy instances, contributing to fault tolerance. If an instance becomes unavailable or experiences issues, ELB routes traffic to other healthy instances, minimizing downtime. [20]

TR 8: We would need a tool to analyze the user's rating pattern, watching behavior, and history and create personalized recommendations based on it. We can use AWS Athena, S3, and SageMaker. Firstly, the insights gathered from users' watching history, ratings, and behavior on the application would be stored in DynamoDB, along with other metrics. We can export the metrics necessary for the recommendation model, from DynamoDB to S3. Athena allows querying data stored in Amazon S3 using standard SQL queries. Tenants' viewing patterns, stored in S3 as structured or semi-structured data, can be easily queried and analyzed using Athena. Amazon SageMaker provides a machine-learning platform that allows the development, training, and deployment of recommendation models. Using the insights gained from Athena and S3, SageMaker can be utilized to build and train a recommendation model based on tenants' viewing patterns. By integrating these services, AWS enables the creation of a comprehensive analysis tool. Athena facilitates querying and analyzing data, S3 acts as a scalable and durable storage solution, and SageMaker brings machine learning capabilities to develop and deploy a recommendation model tailored to tenants' viewing preferences.[5] [4] [3] [15]

TR 9: Amazon DynamoDB serves as a secure and scalable solution for storing user data, content information, metadata, and logs. Its automatic scalability, encryption features, and integration with AWS IAM make it well-suited for a robust and managed storage solution. AWS S3 is suitable for storing image files of content thumbnails as it is best to work with for object storage. [21]

TR 10: We can ensure latency of a maximum of 1 second by incorporating services offering minimum latency. DynamoDB's managed and serverless nature, along with features like provisioned throughput and SSD storage, contributes to low-latency data access. Meanwhile, EC2 offers a range of instance types, enhanced networking options, and placement group capabilities for low-latency compute resources. The combination of these services allows users to interact with the movie catalog interface with minimal latency.

TR 11: Various AWS services offer the capability to secure data through encryption:

- Encryption at rest is inherent in both S3 and DynamoDB [16] [17]
- Encryption is applied to all network traffic within Virtual Private Clouds (VPCs) [1]
- AWS automatically encrypts data in transit between EC2 instances [19]

5.5 Design principles and best practices used

We used the following design patterns from notes and WAF:

1. Perform operations as code[7, page 2]: We use automation to deploy and scale the application in TR 14.1, use automation for load balancing across servers in TR 4.3, and implement automated failure recovery using logs in TR 2.3

2. Optimize for cost[9, page 88]: We implement monitoring and analysis tools for resource usage and associated cost in TR 5.2 so we can set resource limit as an upper bound to optimize cost
3. Scale horizontally to increase aggregate workload availability[6, pg 4]: We use multiple small resources instead of using one large resource and distribute load among those resources in TR 4.3 to avoid a single point of failure
4. Favor-managed services[9, page 90]: We used managed services for automated deployment and management of containers in TR 14.1, and used managed services like Sagemaker and analysis tools like Athena for TR 7.1 instead of setting up infrastructure manually and writing scripts for deployment.
5. Protect data in transit and at rest[8, pg 2]: We use built-in encryption mechanisms to securely store data in appropriate locations in TR 13.2 and implement access control mechanisms for all resources in TR 6.1, and since we deploy VPC, all the network traffic is also encrypted
6. Stop guessing capacity[6, pg 4]: We use monitoring tools to continuously monitor the resource usage in TR 5.2 and based on that we grow or shrink our resources based on varying workloads in TR 4.1

5.6 Tradeoffs revisited

1. Deploying the application in multiple regions & configuring resource usage limit to optimize cost
In order to maintain high availability and comply with regional content restrictions in each region globally, we need to deploy our application in all supported regions. However, deploying the application in multiple regions will lead to overall high resource usage, for example when AWS Lambda fetches content data from the internet it needs to perform multiple writes. All such services must perform duplicate operations to achieve wide region deployment and support. This requires that we set resource limits on the higher side for proper functioning. Hence higher resource limits will lead to higher costs because the parameter we want to optimize (resource limit) is itself on the higher side.

2. Storing all the data in the appropriate storage service and minimizing latency to load the catalog
We use storage options like S3 and DynamoDB based on the nature of the data to store object-based data. We require S3 buckets, however, to store user data and content data, an optimized way of storing it is DynamoDB. This leads to overall performance efficiency which should minimize latency. However, the recommendation tasks require us to connect DynamoDB and S3 and move data between these storages to run our model. This adds additional delays in networking between these components and export delays. However, storing data in appropriate services is more important as it will optimize the overall operational time. There are multiple reads and writes happening in these storage locations in which DynamoDB should perform better in key-value pair data format. Recommendation models require training data which needs to be generated using SQL queries for which Athena and S3 perform better. Both of these tasks are majority operations running in the backend and optimizing these individually should lead to overall performance optimization even if it adds some latency to interconnect these storages.

3. Setting resource limit and high performance
In order to reduce costs beyond a particular level we need to set an upper limit on resource consumption. This can also provide a great analysis tool to identify when the system was over-utilized and was under extreme load which can help in refactoring the limit or study conditions causing such spikes. However,

such a resource limit means the system would fail in extreme workloads that push the system beyond this limit. Since such types of spikes are occasional and random, we can let the system fail and analyze to take further steps instead of having inflated limits at all times.

4. Enhanced security features and high performance

All actions inside the system must be secure and the user data and logs should be stored securely using encryption. Data sent between processes should also go through a secure network and must be encrypted before transmission. All actions should be authorized and users must be authenticated. All these can lead to a decrease in performance due to additional overhead of all these security measures apart from the normal functioning of the application. However, without security measures, the system can be exposed to malicious attacks or security threats. This can heavily impact user experience and cause serious harm to reputation along with many other problems. Some users can also require some additional security measures like AWS private link or advanced encryption methods which can further degrade the performance by increasing latency. However, this is something that cannot be negotiated and should be in place. Hence we need to let performance affect without compromising on the security of the system.

5. Logging data like user actions and optimizing storage costs

The recommendation model heavily relies on the logs stored in DynamoDB that are generated by user actions on the application. Hence for the high accuracy of the recommendation model, the more logs are better, even historical ones. This means the application needs to log all major user interactions, generating a large volume of data that will only grow over time. This can incur huge storage costs to accommodate all these logs. However strong recommendation systems can heavily impact user experience and increase user's usage of the application. Hence we need to make a tradeoff between the cost of storing these logs over the positive experience excessive logging can provide to the users.

5.7 Discussion of an alternate design

Skipped

6 Kubernetes experimentation

6.1 Experiment Design

We selected this subset of TRs from our design:

- TR1: The system availability should be 99% (high availability)
- TR2: The resource capacity for CPU, memory, and storage should rapidly grow and shrink according to varying workloads
- TR3: The system should handle a large number of users and scale with the growing number of users
- TR4: Cost reduction by efficient utilization of resources by dynamically adjusting the number of replicas based on metrics like CPU and memory.

Configuration of the experiment:

- In this experiment we used a subset of TRs of our main application
- The aim of this experiment is to test system performance during heavy load which can be generated when a large number of user generates load on the application or when the application fetched catalog data from numerous sources.
- This type of traffic can be bursty and the system cannot predict this load at the start. Hence we need an autoscaling mechanism to adjust system resources to balance the bursty traffic.
- This autoscaling mechanism also helps to achieve high availability as new servers are spawned to handle increased load and hence reduce downtime.

Experiment Description[2]

To validate the design, we'll set up an experiment that involves deploying a scalable PHP-Apache application using Kubernetes and experimenting with varying workloads to observe the behavior of the Horizontal Pod Autoscaler (HPA) in a 2 cluster minikube environment. Horizontal scaling means to increased load is to deploy more Pods, which is horizontal scaling.

- Installed Minikube with two cluster nodes and kubectl command-line tools
- Deployed and configured Metrics Server to collect metrics from kubelets
- We started a Deployment that runs a container using the hpa-example image, and exposed it as a Service. In the spec for this deployment, we set the following resources:
    - limits:
      cpu: 500m
      memory: 1Gi
    - requests:
      cpu: 200m
      memory: 100Mi
- Then we created a HorizontalPodAutoscaler that maintains between 1 and 25 replicas of the Pods for php-apache deployment and specifies average CPU utilization of 50% and average memory utilization of 20% for each pod
- Finally, load was generated to php-apache service to test the horizontal scaling.

Input:

- Setup an environment, deploy service and HPA instance for scaling
- Generation of load on the above-created service
- Gradually increase the number of load generation pods to simulate a growing user base

Output:

- Monitor service deployment and initial setup and traffic load
- Dynamic scaling of pods to meet resource requirements in response to varying workloads
- Validate that the system can handle a large number of users by scaling up resources and maintaining performance

6.2 Workload generation

Deployed a load generation pod using busy-box image in the Minikube cluster, continuously sending HTTP requests to the PHP-Apache service in an infinite loop and terminated the load to stop sending request so HPA can scale down the replicas

Gradually increased the number of load generation pods similar to above one to simulate increasing user base scenario concurrently accessing the application:

```
pranavmanbhekar@Pranavs-MBP-2 Desktop % kubectl get pods
NAME                          READY   STATUS    RESTARTS   AGE
load-generator                1/1     Running   0          4m43s
load-generator2               1/1     Running   0          2m43s
load-generator3               1/1     Running   0          95s
php-apache-597b96665d-2h6nk   1/1     Running   0          4m10s
php-apache-597b96665d-5xzfj   1/1     Running   0          4m10s
php-apache-597b96665d-6n4lr   1/1     Running   0          108m
php-apache-597b96665d-6xpw6   1/1     Running   0          70s
php-apache-597b96665d-bxmps   1/1     Running   0          70s
php-apache-597b96665d-fn2k5   1/1     Running   0          109m
php-apache-597b96665d-fq6cd   1/1     Running   0          70s
php-apache-597b96665d-jk4ck   1/1     Running   0          4m25s
php-apache-597b96665d-l27nk   1/1     Running   0          70s
php-apache-597b96665d-m9xtl   1/1     Running   0          2m25s
php-apache-597b96665d-ppn86   1/1     Running   0          4m10s
php-apache-597b96665d-q8gnn   1/1     Running   0          2m10s
php-apache-597b96665d-xhskz   1/1     Running   0          2m10s
php-apache-597b96665d-xjqsj   1/1     Running   0          2m10s
```

6.3 Analysis of the results

1. Initially, we have only 2 clusters in Minikube and 1 pod running for our service
2. We created a deployment: php-apache and exposed it as a service
3. We added HPA by setting CPU_utilization=50 i.e. mean CPU utilization=60%, memory_utilization=20%, minimum pod=1, maximum pods=25. So initially we have only 1 replica running as at this moment the load is 0 and the minimum pods are set to 1

```
pranavmanbhekar@Pranavs-MBP-2 Desktop % kubectl get hpa
NAME         REFERENCE                  TARGETS          MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache   0%/50%, 8%/20%   1         25        1          34s
```

4. Now our load generation pod is triggered which continuously sends HTTP requests to our service

```
pranavmanbhekar@Pranavs-MBP-2 Desktop % kubectl get hpa php-apache --watch
NAME         REFERENCE                  TARGETS          MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache   0%/50%, 8%/20%   1         25        1          43s
php-apache   Deployment/php-apache   74%/50%, 11%/20%   1         25        1          60s
php-apache   Deployment/php-apache   250%/50%, 11%/20%   1         25        2          75s
php-apache   Deployment/php-apache   132%/50%, 11%/20%   1         25        4          90s
php-apache   Deployment/php-apache   62%/50%, 11%/20%    1         25        5          105s
php-apache   Deployment/php-apache   58%/50%, 11%/20%    1         25        5          2m
php-apache   Deployment/php-apache   53%/50%, 11%/20%    1         25        5          2m15s
php-apache   Deployment/php-apache   53%/50%, 11%/20%    1         25        5          2m30s
php-apache   Deployment/php-apache   52%/50%, 11%/20%    1         25        5          2m45s
php-apache   Deployment/php-apache   52%/50%, 11%/20%    1         25        5          3m
php-apache   Deployment/php-apache   54%/50%, 11%/20%    1         25        5          3m15s
php-apache   Deployment/php-apache   54%/50%, 11%/20%    1         25        5          3m30s
php-apache   Deployment/php-apache   55%/50%, 11%/20%    1         25        5          3m45s
php-apache   Deployment/php-apache   52%/50%, 11%/20%    1         25        6          4m
php-apache   Deployment/php-apache   44%/50%, 11%/20%    1         25        6          4m15s
php-apache   Deployment/php-apache   46%/50%, 11%/20%    1         25        6          4m30s
php-apache   Deployment/php-apache   44%/50%, 11%/20%    1         25        6          4m45s
php-apache   Deployment/php-apache   45%/50%, 11%/20%    1         25        6          5m
php-apache   Deployment/php-apache   45%/50%, 11%/20%    1         25        6          5m15s
php-apache   Deployment/php-apache   44%/50%, 11%/20%    1         25        6          5m45s
php-apache   Deployment/php-apache   14%/50%, 11%/20%    1         25        6          6m30s
php-apache   Deployment/php-apache   0%/50%, 11%/20%     1         25        6          6m45s
php-apache   Deployment/php-apache   0%/50%, 11%/20%     1         25        6          11m
php-apache   Deployment/php-apache   0%/50%, 11%/20%     1         25        4          11m
php-apache   Deployment/php-apache   0%/50%, 11%/20%     1         25        4          16m
php-apache   Deployment/php-apache   0%/50%, 11%/20%     1         25        3          16m
php-apache   Deployment/php-apache   0%/50%, 11%/20%     1         25        3          21m
php-apache   Deployment/php-apache   0%/50%, 11%/20%     1         25        2          21m
```

5. Once the load was generated, our CPU utilization jumped to 74% which exceeded the target CPU utilization, also memory utilization increased to 11%. So HPA spawns 1 more replica after a minute to handle the incoming traffic.
6. Since load is still generated, CPU utilization further jumps to 250% after 1.5 mins and so HPA spawns more pods, so current replicas = 4
7. After around 2 mins, utilization drops down to 58%, however, the load is still generated so HPA adds 1 more pod to handle the load, and eventually after spawning 6 replicas, the utilization stabilized at 45% after 5 minutes.
8. Now we stop generating the load and HPA should scale down since incoming traffic has stopped
9. After 10 mins, CPU utilization drops down to 0% and the number of replicas gradually decreases from 6 to 4 then to 3 and 2 which takes around another 10 mins to eventually come back to the initial scenario of no load and minimum resources.

10. So as load increased, HPA spawned new pods to keep CPU utilization to 50%, and when load stopped, it scaled down the pods to meet minimum resource requirements
11. Now we created multiple load generators that were sending traffic to the php-apache service to simulate multiple user scenarios.
12. In this case HPA behaves as expected by spawning new pods as the load increases. After 107m unlike before CPU utilization does not stabilize and replicas=6 do not meet the new increased load demands as now there are more users generating the load. So HPA scaled up and spawned 4 more replicas which led to bringing utilization to < 50% after 108m
13. At this more one more user comes in which again generates a new load by sending traffic to the server. To handle this HPA spawns 4 more pods making replicas = 14 which eventually brings utilization close to < 50% after 112m. So in total 14 replicas are running with <50% CPU utilization to server three users sending traffic to the application

```
pranavmanbhekar@Pranavs-MBP-2 Desktop % kubectl get hpa php-apache --watch
NAME         REFERENCE                  TARGETS           MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache   0%/50%, 11%/20%   1         25        2          105m
php-apache   Deployment/php-apache   62%/50%, 11%/20%   1         25        2          105m
php-apache   Deployment/php-apache   133%/50%, 11%/20%   1         25        3          105m
php-apache   Deployment/php-apache   75%/50%, 11%/20%    1         25        6          105m
php-apache   Deployment/php-apache   45%/50%, 11%/20%    1         25        6          106m
php-apache   Deployment/php-apache   44%/50%, 11%/20%    1         25        6          106m
php-apache   Deployment/php-apache   44%/50%, 11%/20%    1         25        6          106m
php-apache   Deployment/php-apache   45%/50%, 11%/20%    1         25        6          106m
php-apache   Deployment/php-apache   45%/50%, 11%/20%    1         25        6          107m
php-apache   Deployment/php-apache   58%/50%, 11%/20%    1         25        6          107m
php-apache   Deployment/php-apache   80%/50%, 12%/20%    1         25        7          107m
php-apache   Deployment/php-apache   65%/50%, 12%/20%    1         25        10         107m
php-apache   Deployment/php-apache   48%/50%, 12%/20%    1         25        10         108m
php-apache   Deployment/php-apache   48%/50%, 12%/20%    1         25        10         108m
php-apache   Deployment/php-apache   66%/50%, 12%/20%    1         25        10         108m
php-apache   Deployment/php-apache   67%/50%, 12%/20%    1         25        14         108m
php-apache   Deployment/php-apache   51%/50%, 12%/20%    1         25        14         109m
php-apache   Deployment/php-apache   52%/50%, 12%/20%    1         25        14         109m
php-apache   Deployment/php-apache   50%/50%, 12%/20%    1         25        14         109m
php-apache   Deployment/php-apache   46%/50%, 12%/20%    1         25        14         109m
php-apache   Deployment/php-apache   49%/50%, 12%/20%    1         25        14         110m
php-apache   Deployment/php-apache   49%/50%, 12%/20%    1         25        14         110m
php-apache   Deployment/php-apache   47%/50%, 12%/20%    1         25        14         110m
php-apache   Deployment/php-apache   48%/50%, 12%/20%    1         25        14         110m
php-apache   Deployment/php-apache   50%/50%, 12%/20%    1         25        14         111m
php-apache   Deployment/php-apache   51%/50%, 12%/20%    1         25        14         111m
php-apache   Deployment/php-apache   50%/50%, 12%/20%    1         25        14         111m
php-apache   Deployment/php-apache   53%/50%, 12%/20%    1         25        14         111m
php-apache   Deployment/php-apache   51%/50%, 12%/20%    1         25        14         112m
php-apache   Deployment/php-apache   52%/50%, 12%/20%    1         25        14         112m
```

Validation:

TR1: As the load increases, new replicas are added to serve the incoming traffic which minimizes downtime of the current service and maintains CPU utilization making sure that other services running in the same pod are not affected, so eventually not affecting their availability. This ensures that overall application availability is high.

TR2: When the load increases, HPA scales up by adding new replicas, which provide more resources like CPU and memory for running the service, also it scales down as the load is reduced /stopped. So our resource capacity for the application rapidly grows and shrinks based on varying workloads.

TR3: As the number of users increases, and the load increases, the total number of replicas increases to handle the increased incoming traffic. So our design is able to handle multiple users concurrently and it can scale as the number of users increases.

TR4: Using autoscaling we can save cost by optimizing the resources based on the workload. As we can see in our design maximum replicas needed is 14 at one point at peak load but our initial configuration starts with 1 replica at minimum load. We only increase replicas when the load increases and do not keep running all replicas at all times. Also when load decreases we reduce the replicas so we can optimize costs. With this autoscaling, we were able to keep the system running during peak load but also optimize cost during low/no load by optimizing resource utilization.

7 Ansible playbooks

Skipped

8 Demonstration

Skipped

9 Comparisons

Skipped

10 Conclusion

10.1 The lessons learned

1. The process of converting business requirements into technical requirements is quite a difficult part as we need to mention a concrete statement to achieve an abstract requirement.
2. Identification of conflicting requirements is the most difficult part as it requires a thorough understanding & analysis of the design and requires a lot of thought.
3. The AWS Well-Architected Framework is an excellent resource for good design principles and proven best practices which helps to form a solid foundation for the design and provides insightful ways on how to evaluate the design. We will use this framework in the future to design well-architected and scalable projects.
4. The interesting part we got to learn is the broad array of services provided by AWS and how well they can be integrated with each other that optimize each part of the design, one example would be DynamoDB to S3 data export can be performed automatically in scheduled intervals without any additional tools. AWS has already provided solutions for many use cases required in design
5. The thought process that we needed to align cloud services with our design's requirements and using WAF principles was complex and that was unexpected but also a good learning experience

10.2 Possible continuation of the project

We provide some ideas for the possible continuation of this project.

- We can carry out implementation in incremental stages by connecting different services that will interact with each other. For example:
  - Exporting data from DynamoDB to S3 as mentioned in section 5
  - Writing a lambda function and executing it
  - Creating VPC subnets in different availability zones and deploying EC2 instances

- After implementing different components, we can piece them together and create a demo of a fully working application.

11 References

[1] Amazon VPC
https://aws.amazon.com/vpc/features/

[2] Kubernetes HPA
https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/

[3] DynamoDB to S3 export
https://aws.amazon.com/blogs/database/introducing-incremental-export-from-amazon-dynamodb-to-amazon-s3/

[4] DynamoDB to S3 export
https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/S3DataExport.HowItWorks.html

[5] Athena Querying Model
https://docs.aws.amazon.com/athena/latest/ug/querying-mlmodel.html

[6] Amazon Well-Architected Framework - Reliability
https://docs.aws.amazon.com/pdfs/wellarchitected/latest/reliability-pillar/wellarchitected-reliability-pillar.pdf#welcome

[7] AWS Well-Architected Framework - Operational Excellence
https://docs.aws.amazon.com/pdfs/wellarchitected/latest/operational-excellence-pillar/wellarchitected-operational-excellence-pillar.pdf#welcome

[8] AWS Well-Architected Framework - Performance efficiency
https://docs.aws.amazon.com/wellarchitected/latest/framework/performance-efficiency.html

[9] AWS Well-Architected Framework - Security
https://docs.aws.amazon.com/pdfs/wellarchitected/latest/security-pillar/wellarchitected-security-pillar.pdf#welcome

[10] Lecture notes
https://moodle-courses2324.wolfware.ncsu.edu/pluginfile.php/118093/course/section/7542/CloudArchitectureNotes2023.pdf?time=1687793847340

[11] Run Lambda function triggered EventBridge rule
https://docs.aws.amazon.com/eventbridge/latest/userguide/eb-run-lambda-schedule.html

[12] Run EC2 using EKS
https://bluexp.netapp.com/blog/aws-kubernetes-cluster-quick-setup-with-ec2-and-eks-aws-cvo-blg

[13] AWS EKS
https://docs.aws.amazon.com/whitepapers/latest/overview-deployment-options/amazon-elastic-kubernetes-service.html

[14] AWS VPC
https://docs.aws.amazon.com/vpc/latest/userguide/what-is-amazon-vpc.html

[15] Scheduling using SageMaker
https://medium.com/analytics-vidhya/a-guide-to-schedule-sagemaker-notebooks-a7a09eb641f6

[16] DynamoDB encryption
https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/EncryptionAtRest.html

[17] S3 Security
https://aws.amazon.com/s3/security/?nc=sn&loc=5

[18] AWS Cloudwatch user guide
https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/WhatIsCloudWatch.html

[19] Encryption in transit
https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/data-protection.html#encryption-transit

[20] AWS ELB user guide
https://docs.aws.amazon.com/elasticloadbalancing/latest/userguide/what-is-load-balancing.html

[21] AWS DynamoDB documentation
https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html

[22] AWS Well Architected Framework
https://aws.amazon.com/architecture/well-architected/?wa-lens-whitepapers.sort-by=item.additionalFields.sortDate&wa-lens-whitepapers.sort-order=desc&wa-guidance-whitepapers.sort-by=item.additionalFields.sortDate&wa-guidance-whitepapers.sort-order=desc

“Referred Intelli-Image Fall 2022 project for guidance on content structure and expectations”