ECE547/CSC547 - Cloud Computing Project

Submitted by:
Pankhi Saini (psaini2 - 200540542)
Aniket Singh Shaktawat (ashakta - 200532430)

Table of Contents

Table of Contents 2

1 Introduction 4
1.1 Motivation 4
1.2 Executive Summary 4

2 Problem Description
2.1 The problem 5
2.2 Business Requirements 5
2.3 Technical Requirements 6
2.4 Tradeoffs and Conflicting Requirements 8

3 Provider Selection 9
3.1 Criteria for choosing a provider 9
3.2 Provider Comparison 10
3.3 The final selection 11
3.4 The list of services offered by the winner 12

4 The first design draft 15
4.1 Basic building blocks of the design 15
4.2 Top-level informal validation 18
4.3 Action items and rough timeline 20

5 Second design draft 21
5.1 Use of the Well-Architected Framework 21
5.2 Discussion of pillars 22
5.3 Use of CloudFormation diagrams 26
5.4 Validation of the design 27
5.5 Design principles and best practices used 30
5.6 Tradeoffs revisited 31
5.7 Discussion of an alternate design 32

6 Kubernetes experimentation 33
6.1 Experiment Design 33
6.2 Workload Generation with Locust 34
6.3 Analysis of the results 37

7 Ansible Playbooks 39

8 Demonstration 40

9 Comparisons **41**

10 Conclusion **42**

10.1 Lessons Learned **42**

10.2 Possible continuation of the project **42**

11 References **43**

1 Introduction

1.1 Motivation

In India, cricket is the most popular sport that is watched by a population of 1.4 billion people. The application that streams live cricket matches and handles a large amount of load which is approximately more than 100 million spectators watching live stream at a time from various devices and the application providing this video streaming service at a lower price than current market competitors is our motivation to pursue this project. The current highest record of live stream traffic for any live sports match ever recorded according to the “Disney+ Hotstar” is around 53 million concurrent viewers, that’s why we thought of designing a cloud architecture for deploying a video streaming application that ensures high availability even in situations where load peaks and reaches around 100 million. We chose the number 100 million keeping in mind the current world record because it is important to check the breakpoint of a design as it is not possible to scale the system for an infinite amount of load because of the hardware capacities and the budget limitations.

1.2 Executive summary

This architecture is being designed from the perspective of a cloud architect of a cloud consumer and this summary is written for a cloud provider. The application that we want to run in a cloud is a video streaming application that will stream live cricket matches, it will also have content related to movies and web series and will also give user-specific suggestions related to movies or web series genres according to the user’s watching preferences that will be extracted from the watch history. This application intends to provide a seamless experience of watching different types of video content online, specifically live cricket matches, and can handle a load of approximately 100 million concurrent viewers watching the live stream from various devices at a time. This application must fulfill certain business requirements listed below.

2 Problem Description

2.1 The problem

The problem that we are designing is the problem that exists in video streaming applications where the application fails to perform to its potential and fails to handle requests more than the expectation when the load increases exponentially in a short period of time and specifically in live match streams where these live video stream applications face challenges when it comes to scale the viewership after a certain point. The issues include scalability problems, network congestion, server overload, inconsistent streaming quality, downtime and significantly impacting user experience. These challenges also lead to bad user experience, loss of users and also damages the reputation of the company. Thus solutions are needed to maintain and enhance the platform's performance, reliability and competitiveness during high demand live streaming events.

2.2 Business Requirements:

Below are some business requirements that this application needs to fulfill:

BR-1: The application should be highly Scalable according to the number of requests
BR-2: Ensure 24/7 Availability
BR-3: Reduce Latency and minimize buffering during the live stream of the matches
BR-4: Ensure Security for the user data and payment information
BR-5: Accommodate time-varying loads
BR-6: Provide High Quality of Service for the live stream and other video content
BR-7: Manage billing and subscription for tenants
BR-8: Restrict content for users based on their subscription plans
BR-9: Provide mobile support such that there is no difference in shifting from laptop to mobile.
BR-10: Integrate monitoring for the deployed cloud infrastructure.
BR-11: Identify each tenant for traffic analysis
BR-12: Reduce cost and optimize utilization of cloud resources.
BR-13: Ensure Geographic Restrictions on videos based on video licensing.
BR-14: The application should be compliant with the data laws of the country
BR-15: Provide scalable storage to store user data generated during the live stream
BR-16: Assure data retention of a particular live streamed match for a time period
BR-17: Provide customer reporting and analytics
BR-18: Ensure backup and disaster management

2.3 Technical Requirements:

TR-1.1: The Application should be horizontally scalable which means that the number of servers or instances should increase as the load increases.When the average CPU usage of an existing instance goes above 75%, new compute instances will be created. If the CPU utilization goes below 25% then that compute instance will be terminated.
TR-1.2: The Application should be able to handle a maximum of 100 million concurrent users approximately during the high demand events (Live stream of cricket matches)

TR-2.1: The availability of the system should be 99.9%, which corresponds to an annual downtime of approximately 8.76 hours. Along with the live stream feature which happens occasionally, it also has features of watching videos and web series which should be always available to view on the application and also there should be no downtime for the payment interface page to maintain the reputation of the application among customers.
TR-2.2: Movies and web series should also be available 99.9% of the time even when they have less traffic as compared to the live stream of a cricket match as the majority of the traffic is on the live stream.

TR-3.1: The latency of the live stream of the cricket match should not be more than 5 seconds.
TR-3.2: The end user should not experience buffering due to the streaming infrastructure, buffering is acceptable if it is experienced due to the poor network of the end user.

TR-4.1: As the resources of the cloud provider are shared among multiple consumers, the data from our application should be secured and must not be accessed by other applications
TR-4.2: The payment information should be stored with utmost security to ensure the integrity of subscription payment operations.

TR-5.1: During high demand, the requests per second are more, and thus the application should be able to accommodate these requests smoothly. Also, there are times when the demand is low and so the application should be flexible to accommodate user fluctuations and streaming demands.

TR-6.1: Different settings for the quality of the video will be present in the application for the users, the different quality settings of the video may be 4K, 1080p, 720p, 360p, so the application needs to ensure that if user selects a particular setting than the video should be delivered in that quality only.

TR-7.1: Different payment gateways should be integrated in the application to support billing and subscription.
TR-7.2: Dedicated storage should be provided for billing history and generating transaction ID for the payments and storing other payment related information to manage payments efficiently and securely.

TR-8.1: Live stream feature will be completely available only to those users who will pay the subscription fees otherwise live stream will only be available for 5 minutes as a demo to the unpaid users.
TR-8.2: Some of the videos and web series will require subscription fees to access, So the application needs to ensure that paid content should not be available to every user.

TR-9.1: Application should have cloud based session managements so that users can easily shift from one device to another without losing their current video progress
TR-9.2: The application should run in the same way in both laptop and mobile devices, for example if the user is using the application on the mobile and then shifts to the web application on the laptop, then there should not be any difference in terms of user experience.

TR-10.1: Monitoring should be there to monitor how the traffic is distributed among the different microservices and which microservice needs to be scaled up.
TR-10.2: Monitoring should be done for keeping track of the health, performance, and security of our cloud resources

TR-11.1: Each user is the tenant of the application and each tenant should be identified by the application and the data corresponding to this should be used for analyzing that which tenant is requesting which service in order to analyze the traffic on the application

TR-12.1: Application should be able to efficiently utilize the cloud resources and should avoid overprovisioning and wastage. Efficient use of resources will further help in reducing the cost.

TR-13.1: Restricted content delivery is required according to the geographical demands. Users should only have access to the content which abides by the rules set by a specific location.

TR-14.1: The application should fulfill the legal requirements of storing the user data of a particular country in the data centers residing in that country.

TR-15.1: User data generated during the live stream such as data from the live chat sessions or the number of API calls should be stored in a storage that can be scaled up or down depending on the traffic received in the live stream.

TR-16.1: The content of the live stream of a particular match should remain on the application for 1 month only, after that it should automatically be removed from the database.

TR-17.1: Application should store user-specific data like watching history and use analytics for suggesting similar content or managing user's watchlist and also keep this information up to date.

TR-18.1: Application should have regular data backups and a dedicated storage for backups with versioning so as retention to a particular version is possible in case of any disaster.

2.4 Tradeoff and Conflicting Requirements:

1. **Horizontal Scalability vs Cost Optimization (TR-1.1 and TR-12.1)**
   Achieving horizontal scalability will help in managing high traffic but it will also lead to increased number of instances which will eventually lead to increased cost at peak load times.

2. **Handling 100 million concurrent users vs avoiding overprovisioning(TR-1.2 and TR-12.1)**
   Handling a large number of users during live streams or at important matches will need extra infrastructure setup and this setup will not be utilized to its full capacity on the normal days and thus overprovisioning of resources cannot be avoided to meet such requirements.

3. **Low latency vs High-quality video (TR-3.1 and TR-6.1)**
   Achieving low latency during live streaming is an essential requirement for this application and as mentioned above that the delay should not be more than 5 seconds. Achieving this might lead to a decrease in the quality of the video.

4. **Customer Analytics vs Ensuring security (TR-17.1 and TR-4.1)**
   Aiming customer analytics is not that straightforward. It requires users personal information and user history which might compromise security. Thus maintaining secure data and performing analytics will require a tradeoff.

5. **Handling Fluctuating Requests vs User Experience(TR-5.1):**
   Designing the application to handle both high and low demand introduces the challenge of resource optimization. Over-provisioning for peak demand may lead to increased costs, while under-provisioning during low demand could impact user experience.

6. **Subscription-based Access vs User Experience(TR-8.1):**
   Restricting live stream features for unpaid users helps in revenue generation but might limit user engagement. Providing a demo for unpaid users could entice them to subscribe, but it might also impact the user experience.

7. **Content Retention Period vs Storage Costs(TR-16.1 and 12.1):**
   Automatically removing content after a month ensures freshness but may lead to data loss for users. Balancing content retention with storage costs and user expectations is challenging.

8. **Scalable Storage for Live Stream Data vs Storage Cost(TR-15.1 and 12.1):**
   Choosing scalable storage ensures flexibility but may result in higher storage costs. Properly managing the scaling of storage resources is essential to avoid unnecessary expenses.

3. Provider Selection

3.1 Criteria for choosing a provider

Following are the criterias set according to our technical requirements that should be considered while making a selection:

1. **Scalability and Elasticity:** In respect to TR 1.1 and TR 1.2, the provider should offer scalable services like autoscaling, allowing the infrastructure to automatically adjust to meet varying demands.

2. **High Availability:** In respect to TR-2.1 and TR-2.2, the provider should ensure 99.9% uptime and have services that ensure fault tolerance.

3. **Low Latency and High Performance:** The provider should offer Content Delivery Network (CDN) services and infrastructure optimizations for low-latency streaming meeting the requirements of TR-3.1 and TR-3.2.

4. **Flexible Resource Allocation:** The services provided should be flexible enough to scale up and down according to demand fluctuations. Flexible pricing models and resource allocation will also help in cost optimization ensuring TR-12.1.

5. **Security and Isolation:** The provider should ensure data security and encryption and have features like Virtual Private Cloud (VPC), encryption, access controls, and dedicated instances or storage for sensitive data like payment information as mentioned in TR-4.1.

6. **Dedicated storage for different requirements:** Different parts of the application require dedicated storages to meet specific technical requirements such as user specific data(TR-17.1) and Restricted content delivery(TR-13.1)

7. **Cost and Pricing Models:** With respect to TR-12.1, If all the required services are provided by the provider then we need to check the pricing of the services and choose the best and the lowest price offered.

8. **Global Presence:** The provider should have a worldwide presence and should be accessible in largest number of countries

3.1 Provider Comparison

The following table provides a comparison of the “big 3” cloud providers with respect to the above criteria. We give each provider a score from 0 (worst) to 2 (best).

Provider	AWS	GCP	Azure	
Criteria	**Score**	**Score**	**Score**	**Justification**
Scalability and Elasticity	2	1.5	1.5	AWS leads with a well-established Auto Scaling mechanism, offering a diverse set of scaling policies and groups for dynamic adjustments. GCP and Azure provide similar features but might have slightly fewer options or restrictions in certain scenarios. They might not have as many integrations or services supported as AWS Auto Scaling
High Availability	2	1.5	2	AWS and Azure boast robust architectures ensuring high availability with Multi-AZ deployments or fault-tolerant setups. GCP offers reliable load balancing but might have slightly less extensive fault tolerance compared to AWS and Azure.
Low Latency and High Performance	2	1.5	1.5	AWS provides an extensive CDN service with optimized latency strategies. GCP and Azure offer competitive CDN services but might have regional limitations impacting latency optimization
Flexible Resource Allocation	2	1.5	1.5	AWS stands out with numerous pricing models and instance types offering flexibility. GCP and Azure provide flexibility but might have restrictions in certain service offerings. Some specialized instance types available on AWS might not have direct equivalents on GCP

Security and Isolation	2	1.5	1.5	AWS leads with comprehensive VPC, encryption, and access controls. GCP and Azure offer similar security features but might have restrictions in certain services or RBAC models.
Dedicated Storage	2	1.5	1.5	AWS offers diverse storage options catering to various requirements. GCP and Azure provide similar services but might have specific limitations in certain aspects or service offerings.
Cost and Pricing Models	1.5	2	1.5	GCP stands out with transparent pricing tools and competitive pricing models. AWS and Azure offer varied pricing but might have slightly higher costs in specific usage scenarios.
Global Presence	2	2	1.5	AWS and GCP have extensive global presence with multiple regions and availability zones. Azure offers a broad presence but might have slightly fewer edge sites compared to AWS and GCP.

3.3 The Final Selection

The major three providers offer comparable services, yet AWS stands out due to its appealing pricing structures, an extensive array of services in its repertoire, and our greater familiarity with AWS compared to alternative cloud providers. Consequently, we opt for AWS, contributing to its top cumulative score.

3.4 The list of services offered by the winner

A. **AWS Auto Scaling:** **https://us-east-1.console.aws.amazon.com/awsautoscaling/home?region=us-east-1#home**
   AWS Auto Scaling enables to quickly discover all of the scalable resources underlying our application and set up application scaling in minutes using built-in scaling recommendations. It helps to configure and manage scaling for our scalable AWS resources through a scaling plan. It lets us to choose scaling strategies to define how to optimize our resource utilization. We can optimize for availability, for cost, or a balance of both. Alternatively, we can leverage custom strategies for greater flexibility.

B. **Amazon EC2: https://aws.amazon.com/ec2/**
   Amazon Elastic Compute Cloud (Amazon EC2) offers the broadest and deepest compute platform, with over 700 instances and choice of the latest processor, storage, networking, operating system, and purchase model to help us best match the needs of our workload. AWS Elastic Compute Cloud ensures high availability through Multi-AZ deployments. It replicates instances across multiple availability zones to provide fault tolerance and 99.9% uptime, meeting TR-2.1's requirement.

C. **Multi-AZ deployments: https://aws.amazon.com/rds/features/multi-az/**
   Deploying instances in multiple Availability Zones (AZs) is a fundamental strategy within Amazon Web Services (AWS) and is commonly associated with various services, including Amazon EC2 (Elastic Compute Cloud). Amazon EC2 instances can be deployed in multiple Availability Zones within a chosen AWS region. This multi-AZ deployment strategy offers increased fault tolerance and high availability for applications or services running on EC2 instances.
   Here's how multi-AZ deployment works with EC2:
   a. **Launching Instances:** When provisioning EC2 instances, users have the option to select the specific AZs where they want to deploy their instances.
   b. **High Availability:** By launching instances in different AZs within the same region, users ensure that if one AZ experiences a failure or disruption, the instances in other AZs remain unaffected. This redundancy helps maintain the availability of the application or service.
   c. **Load Balancing:** To further enhance availability and distribute traffic among instances in different AZs, users often combine multi-AZ deployment with a load balancer service (such as AWS Elastic Load Balancing) that can intelligently distribute incoming traffic across instances in different AZs.
   d. **Data Replication:** For data redundancy and resilience, applications often use additional AWS services like Amazon RDS (Relational Database Service) or Amazon EBS (Elastic Block Store) with multi-AZ configurations to replicate data across AZs.

D. **Elastic Load Balancing(ELB): https://aws.amazon.com/elasticloadbalancing/**
Elastic Load Balancing (ELB) is an AWS service that automatically distributes incoming application traffic across multiple targets, such as Amazon EC2 instances, containers, IP addresses, or Lambda functions, in multiple Availability Zones. It ensures high availability and fault tolerance for applications by evenly distributing traffic and scaling resources in response to incoming requests.
Here are key aspects of the Elastic Load Balancing service:
Load Balancing Algorithms:
- ELB uses various load balancing algorithms to distribute traffic among registered targets. These algorithms include round robin, least connections, and least time.
- These algorithms help balance the incoming traffic load across the registered targets, optimizing performance and ensuring that no single instance is overwhelmed.

High Availability and Fault Tolerance:
- ELB operates across multiple Availability Zones within a region to ensure high availability and fault tolerance. It automatically routes traffic to healthy instances in case of failures or unavailability in one zone.
- By distributing traffic across multiple zones, ELB helps in creating resilient architectures that can withstand zone failures or disruptions.

E. **Amazon Virtual Private Cloud(VPC): https://aws.amazon.com/vpc/**
VPC is a service facilitating the creation of a logically isolated virtual environment. Through VPCs, control inbound and outbound traffic, segregate application components, enhance security with security groups, manage internal IP addressing, and implement load balancing.

F. **Amazon S3 Overview:** https://aws.amazon.com/s3/
Amazon's S3, or Simple Storage Service, serves as a block storage solution capable of accommodating "objects" up to 5TB in size, encompassing images, videos, text files, etc. These objects find their place in buckets. AWS guarantees 11 9s of durability for S3 and presents various features, including:
Security: Control access to buckets and their contents through access control lists and bucket policies.
Replication: Achieve greater redundancy by replicating S3 objects across AWS regions using S3 replication.
Storage Classes: AWS automatically categorizes our data into tiers (frequent, infrequent, archive, deep-archive) based on access frequency, accompanied by a fixed monthly monitoring cost.
Batch Operations: Efficiently perform operations on billions of objects with AWS's S3 Batch Operations

G. **Elastic Kubernetes Service (EKS) Overview: https://aws.amazon.com/eks/**
EKS, a managed Kubernetes service, allows running containers in AWS EC2 or Fargate, ensuring high scalability for applications.

H. **CloudWatch Overview:** https://aws.amazon.com/cloudwatch/
CloudWatch, a logging and monitoring service, enables users to store, access, and analyze logs from various services within the AWS catalog.

I. **DynamoDB Highlights:** https://aws.amazon.com/dynamodb/
DynamoDB, a managed key-value database service, provides millisecond to microsecond latency. Key features include global replication, auto-scaling, encryption at rest, and seamless integration with other AWS services.

J. **Identity and Access Management (IAM) Significance:** https://aws.amazon.com/iam/
IAM allows organizations to centrally and efficiently define access permissions for individuals or entities regarding specific AWS services.

K. **Amazon Relational Database Service (RDS) Summary:** https://aws.amazon.com/rds/
Amazon RDS simplifies the setup, operation, and scaling of relational databases in the cloud.

L. **Amazon API Gateway:** https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html
Amazon API Gateway is an AWS service that simplifies the creation, management, and securing of APIs. It allows developers to build RESTful or WebSocket APIs with defined endpoints, integrates easily with various AWS services or external systems, and provides robust security measures.

M. **AWS Lambda:** https://docs.aws.amazon.com/lambda/latest/dg/welcome.html
Lambda can be used to trigger events or perform actions based on user interactions. For instance, Lambda functions can update user-specific data in DynamoDB or S3 whenever there's a change in the user's watchlist or preferences.

N. **Amazon Redshift:** https://docs.aws.amazon.com/redshift/latest/gsg/new-user-serverless.html
Redshift can be utilized for more in-depth analytics and reporting. It's a data warehousing solution that enables data aggregation and analysis, allowing for deeper insights into user behavior over time.

O. **Amazon Route 53:** https://docs.aws.amazon.com/Route53/latest/DeveloperGuide/Welcome.html
Route 53, combined with geolocation routing policies, allows us to route users to different versions of our application based on their geographic location. This can be used to direct users to specific content based on their location or restrict access as needed.

4. The first design draft

The first design draft of our solution is as follows::

1. Our application mainly provides two features to the customers

    a. **Live stream of the cricket match:** This application will have a microservice which is dedicated only for the live cricket stream. This microservice will require multiple compute instances and databases to host and serve the requests of the different users. This microservice will receive the majority of the traffic in high-demand events, So this will require high scalability. The compute instance hosting this microservice will have two major responsibilities, 1) Getting content from the video cameras recording the live stream and storing it in the corresponding databases and 2) Responding to the requests coming from different users by delivering these live video streams.

    b. Movies and Web Series: This will be the second microservice of our application and this microservice requires to handle less traffic as compared to the live stream microservice but this also needs to have high availability and scalability and fulfill many TRs that are listed above.

4.1 The basic building blocks of the design

The following are the basic building blocks of our design:

AWS Autoscaling:
AWS Autoscaling automatically adjusts the number of computing resources (such as EC2 instances) to match our application's demand. It works through Autoscaling Groups, where we define rules for scaling based on metrics like CPU usage or traffic. This ensures the application stays responsive during peaks and saves costs by scaling down during low-traffic periods.

Elastic Load Balancer:
Elastic Load Balancing (ELB) in AWS efficiently distributes incoming application traffic across multiple targets, like EC2 instances, containers, or IPs, across different Availability Zones. It ensures high availability, scalability, and fault tolerance by balancing loads based on three main types: Application Load Balancer (ALB), Network Load Balancer (NLB), and Classic Load Balancer (CLB).

Amazon VPC (Virtual Private Cloud)
VPC will help us in creating isolated sections of the AWS Cloud dedicated to our resources. This ensures that our application's data remains separate and inaccessible from other applications.

AWS IAM (Identity and access management)
IAM helps manage user access to AWS services and resources. It sets up strict access controls, restricting permissions only to authorized users or services.

AWS KMS (Key Management Services)
We can utilize AWS Key Management Service (KMS) to manage encryption keys and encrypt sensitive data at rest and in transit. Encrypting data ensures that it remains secure even if accessed by unauthorized parties.

Amazon RDS (Relational Database Service)
Amazon RDS (Relational Database Service) is a managed service provided by AWS that simplifies the setup, operation, and scaling of relational databases in the cloud. It offers a range of widely-used database engines like MySQL, PostgreSQL, MariaDB, Oracle, Microsoft SQL Server.

Amazon S3 (Simple Storage Service)
Amazon S3 (Simple Storage Service) is a cloud storage service offered by Amazon Web Services (AWS) that allows businesses and individuals to store and retrieve data from anywhere on the web. It's designed to provide scalable, secure, and highly available storage infrastructure. It enables server-side encryption to safeguard the stored data, ensuring that it remains encrypted and secure.

AWS CloudFront:
Amazon CloudFront is a content delivery network (CDN) service provided by AWS. Its primary purpose is to deliver content, including web pages, videos, images, and other static or dynamic assets, to users with low latency and high transfer speeds. To ensure that users receive videos in the selected quality settings, we will use Amazon CloudFront integrated with AWS Elemental MediaConvert, a file-based video transcoding service, to dynamically transcode and deliver videos in different quality settings.

DynamoDB:
Amazon DynamoDB is a fully managed NoSQL database service provided by AWS. It is designed for applications requiring high performance, scalability, and low-latency access to data. DynamoDB offers a flexible data model, seamless scalability, predictable performance, and automatic management by AWS. It's well-suited for real-time applications, gaming, IoT, and scenarios demanding fast and reliable data storage and retrieval at any scale.

Amazon EKS (Elastic Kubernetes Service):
Amazon EKS is an AWS-managed platform that simplifies deploying, managing, and scaling containerized applications using Kubernetes on AWS infrastructure. It handles the Kubernetes control plane, enabling users to focus on deploying their applications and integrating seamlessly with various AWS services for security, scalability, and ease of management.

AWS Elemental Media Services: AWS Elemental Media Services are a suite of tools by Amazon Web Services (AWS) for managing video content at scale. These services cover various aspects of video processing, including transcoding (MediaConvert), live video processing (MediaLive), packaging and delivery (MediaPackage), optimized storage (MediaStore), and personalized ad insertion (MediaTailor). They help streamline video workflows, making it easier to process, deliver, and monetize video content across different platforms and devices.

Amazon Aurora:
Amazon Aurora is a fully managed relational database service provided by Amazon Web Services (AWS). When it comes to security, Amazon Aurora is considered to be a strong choice among the list of databases provided by AWS for storing payment and billing information. The reasons for this include encryption, network isolation, authentication, authorization, and audit logging that allows capturing of database activity for compliance and security monitoring.

Amazon API Gateway:
Amazon API Gateway is a fully managed service provided by AWS that makes it easy for developers to create, publish, maintain, monitor, and secure APIs at any scale. This helps in integrating payment gateways with the AWS application and helps in the subscription process. Connecting API Gateway endpoints to third-party payment gateway APIs (e.g., Stripe, PayPal) using Lambda functions or HTTP integrations enables communication between the application and the payment gateway for processing payments

Amazon Cognito:
Amazon Cognito is a service provided by AWS that has session management functionalities. In the context of our streaming application, the user session includes preferences, watch history, and other relevant data, that will be stored in an AWS database. When a user switches from one device to another, the application retrieves their session information from the cloud, ensuring a consistent and continuous user experience across devices. When a user logs in, the application running on the EC2 instances can use the Amazon Cognito API to authenticate the user's credentials. The EC2 instances can then receive tokens from Cognito, validating the user's identity. We can use Cognito tokens to manage sessions on the server side, allowing users to switch between devices while maintaining their session state.

AWS Cost Explorers and AWS Budgets:
By using Amazon EC2 Auto Scaling in conjunction with Amazon CloudWatch, AWS Cost Explorer, and AWS Budgets, we can optimize the utilization of EC2 instances in our streaming application. This combination allows for dynamic scaling based on demand, efficient monitoring of resource utilization, and proactive cost management to reduce unnecessary expenses.

4.2 Top-level, informal validation of the design

Our application has two microservices that are:

1. Microservice that contains the logic for live streaming Cricket matches.
2. Microservice that contains the logic for the content delivery that is movies and web series.

The 24/7 availability requirement is for the second microservice. The live streaming delivery feature of the first microservice only needs to be available when the live match goes on and it is scheduled and pre-planned so this 99.9% availability requirement does not apply to the first microservice.
Let's discuss the services available to meet the technical requirements:

- **Scalability:**
  AWS Auto Scaling enables us to automatically adjust the number of compute instances based on the traffic coming on the application. The metrics used by the auto-scaler will be the average CPU utilization of the EC2 instances. Thus using AWS Autoscaling service with Amazon Cloud Watch, We can achieve horizontal scalability.
  Another service that provides scalability is **Amazon EKS (Elastic Kubernetes Service)**. Amazon EKS is a managed Kubernetes service that simplifies the deployment, management, and scaling of containerized applications using Kubernetes. Thus EKS is a good option when the application is containerized. Kubernetes provides native features for the horizontal scaling of containers. While AWS Auto Scaling is valuable for managing the scaling of traditional compute resources like EC2 instances, when it comes to scaling containerized applications, especially in a dynamic and container-centric environment, Amazon EKS with Kubernetes offers a more tailored and efficient solution. It provides the necessary tools and capabilities to effectively scale containerized workloads horizontally based on demand.
  Using Amazon EKS service along with Amazon Cloud Watch, **TR-1.1, TR-1.2, TR-5.1, TR-10.1, and TR-10.2 can be achieved**

- **24/7 Availability requirement:**
  Application Load Balancer(ALB) helps ensure high availability by automatically distributing incoming traffic across healthy instances. It monitors the health of registered instances and directs traffic only to healthy instances. Hence, It can distribute requests smoothly during high demand and can adjust to fluctuations in user demand.
  ALB will help achieve the high availability technical requirements TR-2.1 and TR-2.2

- The reasons for choosing ALB over NLB and CLB are as follows:
  1. **Content-Based Routing:** ALB operates at the application layer, allowing for content-based routing. This is crucial for video streaming applications where different content or streams may need to be handled differently based on URL paths, hostnames, or headers.

2. **WebSocket Support:** ALB supports WebSocket, which is essential for real-time communication in video streaming applications. This enables bidirectional communication between the client and the server, which is often required for live interactions during streaming.

- **Low Latency:**
  CloudFront uses a global network of edge locations (edge servers) to cache and distribute content closer to end-users. This reduces the latency in delivering content, providing a faster and more responsive experience for users. The edge locations are spread across multiple continents and these edge locations cache our content closer to end-users, reducing latency and improving performance. **This will help in meeting the requirements mentioned in TR-3.1**

- **Content delivery:**
  MediaPackage, when integrated with CloudFront, provides a reliable and scalable solution for delivering live video streams, minimizing buffering due to the streaming infrastructure. **This will help in meeting the technical requirement TR-3.2**. Any buffering experienced by end-users can be attributed to their individual network conditions.

- **Payment and security:**
  Amazon VPC, AWS IAM, Amazon API Gateways, and AWS KMS collectively help in securing data access which is mentioned in TR 4.1.
  1. **Handle Payment Request:** API Gateway helps in defining API methods and resources within the application to handle payment requests such as creating invoices, processing payments, handling subscriptions, or managing user payment profiles.
  2. **Subscription Management:** IAM can control access to subscription-related resources or databases containing user subscription details. VPC can isolate these resources within the network for additional security.
  3. **Payment Security:** AWS KMS encryption can be used to secure payment-related information, ensuring that payment data is encrypted when stored and transmitted.
  4. **Network Security:** VPC helps in creating a secure environment where the live streaming app infrastructure and payment systems are logically isolated from other networks.
  5. **Access Control:** IAM manages access to various components of the app, ensuring that only authorized users or services can access payment information or sensitive areas of the application.

- **Storage:**
  For storage purposes, we can use Amazon S3, Amazon Aurora, or DynamoDB. Choosing Amazon DynamoDB over Amazon S3 and Amazon Aurora depends on

specific application requirements, performance needs, and the nature of the data being stored or accessed. Here are scenarios where DynamoDB might be preferred:

1. **Low-Latency, High-Throughput Access:** DynamoDB is designed for applications that demand single-digit millisecond response times. Our application requires rapid and consistent read/write access to data with low latency, DynamoDB's performance characteristics might be more suitable than S3 or Aurora.
2. **Scalability and Managed Service:** DynamoDB is a fully managed NoSQL database service that automatically scales based on workload demands. It's capable of handling massive traffic spikes and scaling both read and write throughput without manual intervention.

This helps in achieving TR-4.2, TR-7.2, TR-15.1 and TR-18.1

4.3 Action items and rough timeline

Skipped

5 Second design draft

5.1 Use of the Well-Architected framework

The AWS Well-Architected Framework is a set of best practices and guidelines provided by Amazon Web Services (AWS) to help architects design and build secure, high-performing, resilient, and efficient infrastructure for their applications. It provides a structured approach for evaluating architectures and ensuring they align with industry best practices.
The five pillars on which the this architecture is based on are as follows:

1. **Operational Excellence Pillar:**
    a. This pillar focuses on operational practices that enable efficient management of applications and infrastructure, emphasizing the ability to run and monitor systems to deliver business value continually.
    b. Key considerations include automating processes, understanding performance metrics, responding to events, and making improvements over time.

2. **Security Pillar:**
    a. Security is paramount in any architecture. This pillar emphasizes the implementation of robust security controls and mechanisms to protect data, systems, and assets.
    b. It involves implementing strong identity and access management, applying data encryption, establishing monitoring and detection mechanisms, and ensuring compliance and governance.

3. **Reliability Pillar:**
    a. Reliability refers to the ability of a system to recover from failures and continue to function as expected. It involves building architectures that mitigate disruptions and reduce the impact of failures.
    b. This pillar includes strategies such as implementing fault tolerance, redundancy across different Availability Zones, automating recovery procedures, and testing failure scenarios.

4. **Performance Efficiency Pillar:**
    a. This pillar focuses on optimizing performance and resource utilization to meet system requirements and maximize efficiency. It involves selecting the right resource types, using scalable designs, and monitoring performance.
    b. Strategies include using the appropriate instance types, caching, optimizing storage and databases, and applying performance testing and monitoring.

5. **Cost Optimization Pillar:**
    a. Cost optimization involves maximizing the value of IT spending by understanding and controlling costs. It emphasizes the efficient use of resources without sacrificing performance or security.

   b. Key considerations include adopting a pay-as-you-go model, analyzing and optimizing spending, using managed services to reduce operational overhead, and implementing cost-effective architectures.

6. **Sustainability Pillar:**
   a. The Sustainability Pillar, introduced by AWS as an addition to the Well-Architected Framework in 2021, focuses on integrating sustainability principles into the design, operation, and lifecycle of cloud architectures. This pillar emphasizes how organizations can build and run workloads on AWS while considering environmental impacts and promoting sustainability initiatives.

5.2 Discussion of Pillars

Operational Excellence Pillar:

The Operational Excellence pillar within the AWS Well-Architected Framework focuses on implementing best practices and processes to enable efficient operations and continuous improvement in managing and maintaining AWS workloads. It emphasizes delivering business value through well-defined procedures, automation, and iterative refinement of operational processes.

Design Principle:

Design principles provide guidance for designing, implementing, and maintaining AWS workloads with a focus on operational efficiency, reliability, and agility. The 5 design principles discussed in the Operational Excellence Pillar are as follows:

Perform Operations as Code:

- Automate operational procedures and tasks by representing them as code (Infrastructure as Code - IaC). Use tools like AWS CloudFormation or AWS CDK to define and manage infrastructure and configurations programmatically.
- Apply version control, reuse patterns, and treat operational procedures as software artifacts to ensure consistency, repeatability, and traceability.

Make Frequent, Small, and Reversible Changes:

- Adopt a practice of making small, incremental changes to systems and infrastructure to minimize risk and improve agility.
- Enable the ability to rollback changes easily and quickly in case of unexpected issues (e.g., through automated deployment strategies or infrastructure-as-code practices).

Refine Operations Procedures Frequently:

- Continuously refine and optimize operational procedures and workflows through feedback loops, iterative reviews, and continuous improvement practices.
- Encourage a culture of experimentation and innovation to explore new, more efficient ways of performing operations.

Anticipate Failure:

- Design architectures and operational procedures with the assumption that failures will occur. Implement fault-tolerant architectures and automated recovery mechanisms to minimize disruptions.
- Conduct failure testing (chaos engineering, fault injection) to validate resilience and identify potential weak points in the system.

Learn from Operational Events:

- Establish robust monitoring, logging, and alerting mechanisms to capture operational events and performance metrics.
- Analyze data from operational events to derive insights, improve systems, and drive operational enhancements.

Best Practices

AWS says that there are four best practices to achieve operational excellence in the cloud:

- Organization
- Prepare
- Operate
- Evolve

Organization: This practice area focuses on establishing a foundation for operational excellence by aligning organizational structures, culture, and processes with cloud best practices.

- **Roles and Responsibilities:** Clearly define roles, responsibilities, and ownership for operational tasks and processes within teams.
- **Governance and Compliance:** Implement governance controls, policies, and compliance frameworks to ensure adherence to standards and regulations.
- **Leadership Support:** Foster a culture that values operational excellence and secures leadership buy-in to drive organizational change.

Prepare: Prepare emphasizes planning and readiness, ensuring that the organization is equipped with the necessary tools, skills, and processes to effectively operate in the cloud environment.

- **Training and Skill Development:** Invest in training programs to upskill teams and equip them with the knowledge needed to leverage cloud services effectively.
- **Documentation and Runbooks:** Develop comprehensive documentation, runbooks, and standard operating procedures (SOPs) to guide teams in operational tasks.
- **Risk Management:** Identify potential risks, conduct risk assessments, and develop mitigation strategies to address operational challenges proactively.

Operate: Operate focuses on implementing and optimizing operational processes, tools, and automation to efficiently manage cloud resources and applications.

- **Automation and Tooling:** Leverage automation tools, scripting, and infrastructure-as-code practices to automate routine operational tasks and workflows.

- **Monitoring and Incident Response:** Set up robust monitoring, logging, and alerting mechanisms to detect, respond to, and remediate incidents promptly.
- **Performance Optimization:** Continuously optimize resource utilization, performance, and costs through performance tuning and optimization strategies.

Evolve: Evolve emphasizes continuous improvement and innovation, encouraging organizations to evolve their operational practices in response to changing business needs and technological advancements.

- **Continuous Improvement:** Foster a culture of continuous learning, experimentation, and feedback to drive iterative improvements in operational processes.
- **Innovation and Experimentation:** Encourage teams to innovate, experiment with new technologies, and adopt emerging best practices to enhance operational efficiency.
- **Feedback and Iteration:** Collect feedback from operational experiences, analyze data, and iterate on processes to adapt to evolving requirements and trends.

Reliability Pillar:

The Reliability pillar within the AWS Well-Architected Framework focuses on designing and implementing architectures that deliver consistent and predictable performance while mitigating the impact of failures. It aims to ensure that systems can recover quickly from disruptions and maintain availability for users and applications.
The best practice guidance for implementing reliability pillar in AWS are as follows:

Automatically Recover from Failure:

- Design systems with automated recovery mechanisms to ensure quick and automated responses to failures. Leverage AWS services like Auto Scaling, AWS Elastic Load Balancing, and AWS Lambda to automatically handle failures by replacing unhealthy instances, distributing traffic, or executing code in response to events.
- Implement fault-tolerant architectures that can automatically detect failures and recover without manual intervention, minimizing downtime and maintaining system availability.

Test Recovery Procedures:

- Regularly test and validate recovery procedures to ensure they effectively restore systems to a working state. Conduct disaster recovery drills, failover tests, or chaos engineering experiments to simulate failures and validate the effectiveness of recovery mechanisms.
- Testing recovery procedures helps identify weaknesses or gaps in the system's resilience and provides insights into improving recovery mechanisms.

Scale Horizontally to Increase Aggregate Workload Availability:

- Embrace horizontal scaling by distributing workloads across multiple instances or resources rather than relying on a single large instance. Utilize auto-scaling groups to automatically add or remove instances based on demand.

- Horizontal scaling enhances availability by reducing the impact of failures on individual components and allowing the system to handle increased load by adding more resources.

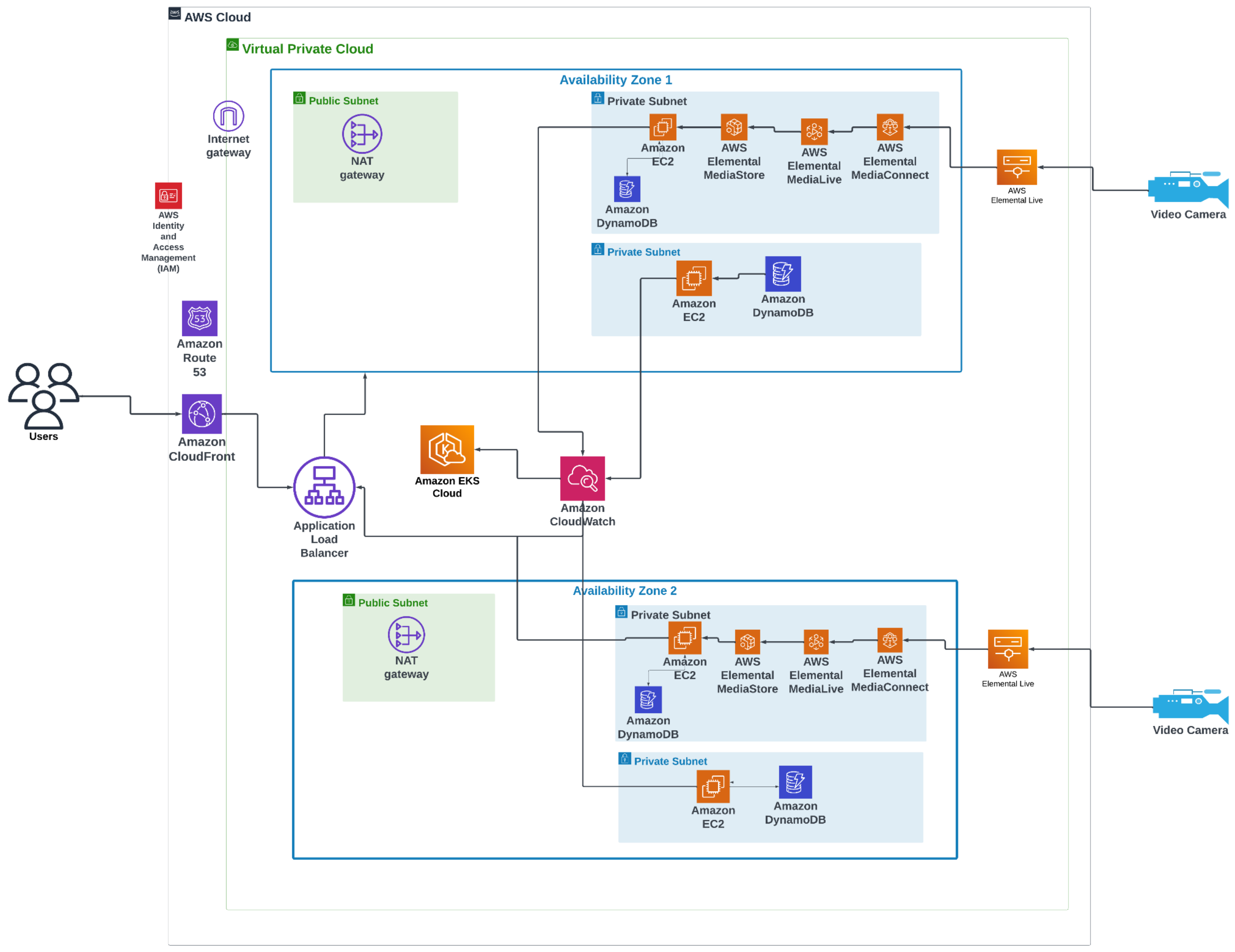
Stop Guessing Capacity:

- Utilize AWS services like AWS Auto Scaling and AWS Lambda to automatically adjust resources based on workload demand. Implement monitoring and metrics to dynamically scale resources in response to changes in traffic patterns or demand.
- Avoid over-provisioning or under-provisioning resources by leveraging elasticity and scaling features, ensuring that the system always has the right amount of resources to handle varying workloads.

Manage Change in Automation:

- Implement automation for change management processes, including deployment, configuration, and updates. Use infrastructure as code (IaC) tools like AWS CloudFormation or AWS Systems Manager to automate and version-control infrastructure changes.
- Automating change management reduces the risk of human error, ensures consistency, and facilitates faster and more reliable deployments while maintaining system reliability.

5.3 Use of Cloudformation Diagram



5.4 Validation of the design

The explanation of the design is as follows:

The above is the cloud formation diagram which represents an AWS architecture of a video streaming application that has two components contained in two different microservices, the details of both microservices are as follows:

Microservice 1: It contains the application logic for streaming a live video of a cricket match that is being fetched from the live video camera source
Microservice 2: It contains the application logic for delivering movies and web series that are already stored in a database.

This application is housed on the AWS Cloud and everything inside the "AWS Cloud" outline is the process happening inside that. The end users will run this application on a web browser and they will send HTTP requests to either get the live stream or the movies and web series.

The outermost box is an outline of the AWS cloud, everything inside is the function and process happening on the AWS cloud, two things outside the cloud outline are the user icon and video camera.

The Flow of a request from a user is as follows:

1) The user's browser initiates a request for the application by entering a domain name in the web browser
2) IAM plays a crucial role in controlling and managing access to the AWS resources involved in serving the application through Amazon Route 53 and CloudFront. IAM identifies the user and based on the permission access is granted.
3) Amazon Route 53 resolves the domain name and the request is then directed to CloudFront which acts as a content delivery network (CDN)
4) CloudFront helps improve the performance of the application by caching and distributing content across a global network of edge locations. It checks its cache to see if the requested content is already present at an edge location. If the content is in the cache and is still valid (according to cache settings), CloudFront can respond directly to the user from the edge location, reducing latency.
5) If the content is not in the cache or is expired, CloudFront forwards the request to the Application Load Balancer.
6) The ALB receives the request and, based on its load-balancing configuration, directs the request to one of the backend EC2 instances according to the type of request (Cricket Live Stream or Movie and Web Series) in one of the Availability zones.
7) And then the response is sent back by the corresponding microservice and this is how a request is fulfilled.

Following are the arguments on how the services we chose in our design will help achieve the TRs listed above:

- **Scalability:** Amazon EKS helps in achieving the scalability requirements. EKS allows us to manage Kubernetes clusters that automatically scale based on CPU utilization, making it easier to handle increased load dynamically. Kubernetes Horizontal Pod Autoscaler (HPA) can be configured to scale pods (containers) within the application automatically based on CPU metrics. As the CPU usage increases beyond the defined threshold (e.g., 75%), EKS can trigger the creation of new pods to accommodate the higher load. Conversely, when CPU utilization decreases (e.g., below 25%), EKS can terminate excess pods, effectively scaling down resources. EKS, with its ability to auto-scale and manage clusters efficiently, provides the infrastructure to handle spikes in user demand during high-profile events like live streaming of cricket matches. EKS is a fully managed Kubernetes service. It abstracts away the complexities of deploying, managing, and scaling Kubernetes clusters. This will allow our developers and operators to focus more on the applications and less on the underlying infrastructure. EKS excels at container orchestration. It provides features such as rolling updates, auto-scaling, and self-healing, making it well-suited for our use case. So, EKS will achieve the Technical requirements related to scalability (TR-1.1)

- **24/7 Availability requirement:** Achieving 24/7 availability for a video streaming application demands robust infrastructure, and Application Load Balancer plays a pivotal role in ensuring uninterrupted service. ALB is a service provided by AWS that automatically distributes incoming application traffic across multiple targets, such as EC2 instances, containers, and IP addresses, within one or more availability zones.

  Let's discuss how ALB can help achieve the technical requirements mentioned in TR-2.1 and TR-2.2.

  Incoming 'traffic' or 'load' refers to the requests and data that are sent to the application, or network within a specific timeframe. In the context of our application, incoming traffic typically consists of HTTP requests generated by users or other systems accessing the application. For our Live Streaming microservice incoming traffic includes the continuous flow of data for features like live chatting and commenting on the live video

  ALB handles occasional spikes in traffic associated with live streaming by distributing requests across multiple targets, preventing any single server from becoming a bottleneck.ALB can be configured with health checks to ensure that the live-streaming service remains available. If a target (e.g., a server handling live streaming) becomes unhealthy, ALB automatically redirects traffic to healthy targets.
  Monitoring CPU utilization on backend instances helps in understanding the load on each server. So CPU utilization is the metric that will be used by the load balancer for decision making

ALB can dynamically adjust to the number of healthy instances based on health checks and traffic patterns. Configuring health checks with high sensitivity may lead to false positives, triggering unnecessary scaling actions. Balancing health check sensitivity is crucial to ensure accurate detection of unhealthy targets without causing unnecessary scaling actions. ALB's sophisticated load-balancing algorithms may introduce a minimal amount of latency due to the decision-making process. This latency is usually negligible but should be considered.

- **Monitoring:** CloudWatch is a monitoring and observability service provided by AWS that helps us to collect and track metrics, collect and monitor log files, and set alarms. It can be used to achieve the technical requirements mentioned in TR-10.2, which involve monitoring the distribution of traffic among different microservices and tracking the health, performance, and security of cloud resources.CloudWatch integrates seamlessly with AWS EC2 which allows us to monitor and gain insights into the performance and health of the EC2 instances housing our application that can be used by EKS to make the scaling decisions.

- **Low Latency:** Amazon CloudFront, AWS's content delivery network (CDN), plays a crucial role in achieving low latency for a video streaming application by employing several strategies to optimize content delivery and reduce the time it takes for users to access the content. CloudFront integrates seamlessly with AWS Elemental Media Services, offering capabilities for live video streaming and on-demand media processing. This integration optimizes the delivery of video streams, reduces buffering, and ensures low-latency live streaming experiences for viewers and this will help achieve out Technical requirements related to Low Latency(TR-3.1).

- **Tenant Identification:** AWS IAM(Identity and Access Management) helps majorly in tenant identification. IAM allows defining roles with specific permissions for different tenants. By assigning unique IAM roles to each tenant, we can control their access to resources and data within the application. Implementing custom identification methods within the application logic can also help in distinguishing tenants. It also provides the foundation for implementing access controls and user authentication mechanisms required for subscription models. By using IAM we are able to fulfill Tenant Identification technical requirements. (TR-11.1). Once tenants are identified and access is granted, CloudWatch can capture logs and metrics related to their activities. This data can be used for traffic analysis, monitoring, and identifying patterns or trends specific to each tenant.

- **Storage:** DynamoDB's features like low-latency access, scalability, consistent performance, flexible data modeling, and built-in security contribute significantly to meeting requirements in scenarios where rapid and consistent data access, scalability without management overhead, and robust security measures are crucial for the success of a live streaming application. DynamoDB offers built-in security features like fine-grained access controls, encryption at rest and in transit, and data backups for durability. This ensures data integrity, confidentiality, and resilience against potential threats or data loss, critical for managing sensitive information, including user preferences, subscriptions, or transaction details in a live streaming app. This works closely with the following requirements: TR-4.2, TR-7.2, TR-15.1 and TR-18.1.

- **Content Delivery:** MediaPackage supports ABR, a technique that delivers video content in multiple bitrates and resolutions concurrently. It dynamically adjusts the video quality based on the viewer's available bandwidth and device capabilities. CloudFront, when integrated with MediaPackage, acts as a content delivery network (CDN) to cache and distribute the live video content closer to end-users. This minimizes latency by delivering content from edge locations, reducing the need for data to traverse long distances and mitigating buffering caused by high latency meeting requirements TR-6.1 and TR-3.1

5.5 Design principles and best practices used

We use the following design principles taken from the AWS WAF:

1. **Perform operations as code:** We have applied this approach in the AWS application by using tools like CloudFormation to define and manage infrastructure, automate deployment, configure monitoring, set up backups, and ensure compliance. By treating operational tasks as code, it improves consistency, repeatability, and efficiency in managing the application's infrastructure and operations.

2. **Use managed services:** Utilizing managed services in the design of the application involves leveraging AWS's pre-built and managed offerings rather than building and maintaining infrastructure from scratch. This approach was achieved by using services like AWS Elemental Media Services for streaming, Amazon CloudFront for content delivery, IAM for security, managed databases for data storage, CloudWatch for monitoring, and more. By relying on these managed services, the application benefits from AWS's expertise, reducing operational burden, ensuring scalability, and meeting technical requirements more efficiently

3. **Apply security at all layers**: Securing the application across all layers while meeting the requirements involves automating security measures, continuous compliance checks, robust monitoring, encryption, access controls, threat detection, and secure application design. These practices, aligned with each requirement, ensure a comprehensive security posture covering infrastructure, data, access, monitoring, and incident response.

4. **Scale horizontally to increase aggregate workload availability:** Scaling horizontally for increased availability involves dynamically adding or removing compute instances based on demand (Auto Scaling), using load balancing for even distribution of traffic,

adopting stateless architectures for seamless scaling, utilizing microservices and containerization, leveraging CDN for global content delivery, implementing database strategies for workload distribution, and ensuring failover mechanisms for continuous operation. These practices enable the application to handle varying workloads efficiently and maintain availability during peak demand.

5.6 Tradeoff Revisited

Horizontal Scalability vs Cost Optimization (TR-1.1 and TR-12.1):

- Employing Amazon EKS with target tracking or dynamic scaling policies based on CPU utilization which can be done by CloudWatch helps in managing instance counts. Using Amazon EC2 Spot Instances during low-demand periods further helps in reducing costs. It is not an easy task to reduce the cost as scaling down the number of instances can affect the user experience but Amazon EKS helped us meeting this tradeoff.

Handling 100 Million Concurrent Users vs Overprovisioning (TR-1.2 and TR-12.1):

- Using Application Load Balancing(ALB) to distribute traffic across AWS regions closer to users, dynamically scaling resources based on user demand was an optimal solution which helped us meet both the requirements of handling concurrent users and overprovisioning. The application will not have traffic all the time and thus enabling ALB will definitely help in reducing the wastage of resources to some extent.

Low Latency vs High-Quality Video (TR-3.1 and TR-6.1):

- Leveraging AWS Elemental Media Services, such as MediaLive for live video processing, helped in ensuring adaptive bitrate streaming for different resolutions without compromising latency.

Customer Analytics and Security (TR-17.1 and TR-4.1):

When implementing customer analytics, we ensured that IAM controls are configured to grant access only to the necessary personnel or systems. We Used IAM roles to delegate permissions based on specific responsibilities.
By carefully configuring IAM for access control and security measures and utilizing CloudWatch for analytics and performance monitoring, we achieved a trade-off that ensures both customer analytics and security in our AWS environment.

Subscription-based Access and User experience(TR-8.1 and TR-12.1):

- Implementing a subscription-based access model involves offering exclusive live stream features to paying subscribers while restricting access for unpaid users. To strike a balance between revenue generation and user engagement, various strategies can be employed. Providing a limited-time demo or restricted access for unpaid users can showcase premium content, potentially enticing them to subscribe. AWS IAM for access control facilitated us with efficient user management, enabling fine-grained control over feature access based on subscription status.

Content Retention Period vs Storage Costs (TR-16.1 and TR-12.1):

- Automatically removing content after a month ensures freshness but may lead to data loss for users. Balancing content retention with storage costs and user expectations is crucial. Content retention period management involves a delicate balance between ensuring content freshness and preventing potential data loss for users. Automatically removing content after a specified duration, such as a month, guarantees content relevance but raises concerns about user data loss. Our solution distinguished between archiving and deletion strategies and notified users in advance of content removal. So we used a retention period as 1 month only so that the data storage is not required for a large amount of days which surely helped in the storage cost reduction.

Scalable Storage for Live Stream Data vs Storage Costs(TR-15.1 and TR-12.1):

- Scalable storage for live stream data involves choosing flexible storage solutions that accommodate data growth but balancing scalability with costs is vital. This strategy is met by selecting dynamo DB implementation with lifecycle management to transition data based on usage, scaling storage dynamically using auto-scaling, employing compression, and monitoring costs to optimize storage usage efficiently while controlling expenses.

5.7 Discussion of an alternate design

Skipped

6 Kubernetes experimentation [0%]

6.1 Experiment Design

The Technical requirements that need to be fulfilled are as follows:

TR-1: The application needs to exhibit horizontal scalability, ensuring that the number of nodes in the Minikube cluster increases proportionally with the rising workload. This scalability requirement is driven by CPU utilization metrics. Specifically, when the average CPU usage of an existing node within the Minikube cluster surpasses 75%, the system should dynamically create new compute instances to accommodate the increased load. Conversely, when the CPU utilization drops below 25%, the system should intelligently terminate the corresponding compute instance to optimize resource utilization.

TR-2: The application needs to be available 24/7 which means that it should respond to the requests all the time

Following are the experiment details

- We used Minikube for our experiment. Minikube is an open-source tool that helps to run the Kubernetes cluster locally for development and testing purposes. In our Minikube cluster, there is only one node that will contain both the control plane and data plane and will house different control pods and worker pods on a single node.

- “php-apache” application was used for this experiment. This application was deployed in one pod in the given node using the yaml file present in this URL “https://k8s.io/examples/application/php-apache.yaml”.

- After that we deployed HPA(Horizontal Pod Autoscaler) in the minikube. The HPA is a Kubernetes resource that automatically adjusts the number of running pods in a deployment or replica set based on observed metrics, such as CPU utilization that we used in this case.

- We have set the threshold average CPU utilization as 75% for the application pods, which means that if the average CPU utilization of the application pods goes above 75% the HPA will scale up the number of pods.

- We have also installed a metrics server in the node which fetches the CPU utilization of the pods, and using this metric the HPA will scale up or scale down the number of pods.’

- After the scale-up of the application pods, the load is distributed among the multiple application pods using the rules set by the Kube-Proxy. Kube Proxy is mainly responsible for providing networking functionalities, but in this case, it indirectly helps in distributing the load among different pods. Using these configurations, the load was applied to the application using Locust and then the HPA started its work and the details of how the scalability was achieved are mentioned in the below section. Using HPA the technical requirement 1 was achieved

- The minimum and maximum number of pods are mentioned in the HPA configuration. The minimum number of pods set here is 1, and the maximum number is 10

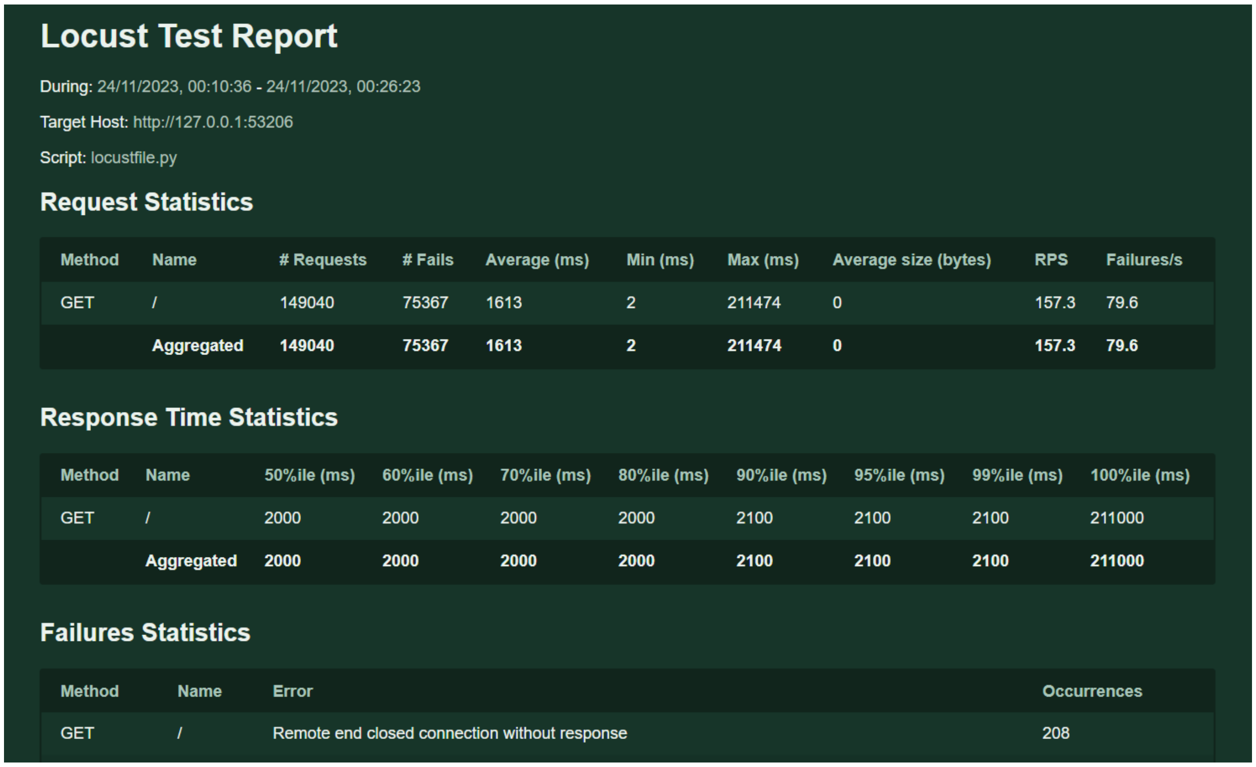
- One application pod will always exist in the node which will help achieve the second technical requirement.

6.2 Workload generation with Locust

We used Locust in our experiment to generate load for the application. The performance of the php-application is measured with Locust. It is an open-source and scalable performance testing tool. It enables us to create a load on our web application and simulate a high number of concurrent users (virtual users) in order to evaluate how the program responds to varying degrees of stress and also to test our HPA in order to achieve the mentioned Technical Requirements

The Process is as follows:

- Using Locust we generated load on the application php-apache running on the local host. The URL used for the same was http://127.0.0.1:54559. On this URL the load was sent through Locust which was also running on the local host on a different port number 8089 using the file locust.py and added peak concurrency, spawn rate, and target host URL mentioned above
- Number of users (peak concurrency): 1000
- Spawn rate (users added/stopped per second): 100
- Total Number of Requests: 149040
- Requests/second: 157



Locust Test Report

During: 24/11/2023, 00:10:36 - 24/11/2023, 00:26:23

Target Host: http://127.0.0.1:53206

Script: locustfile.py

Request Statistics

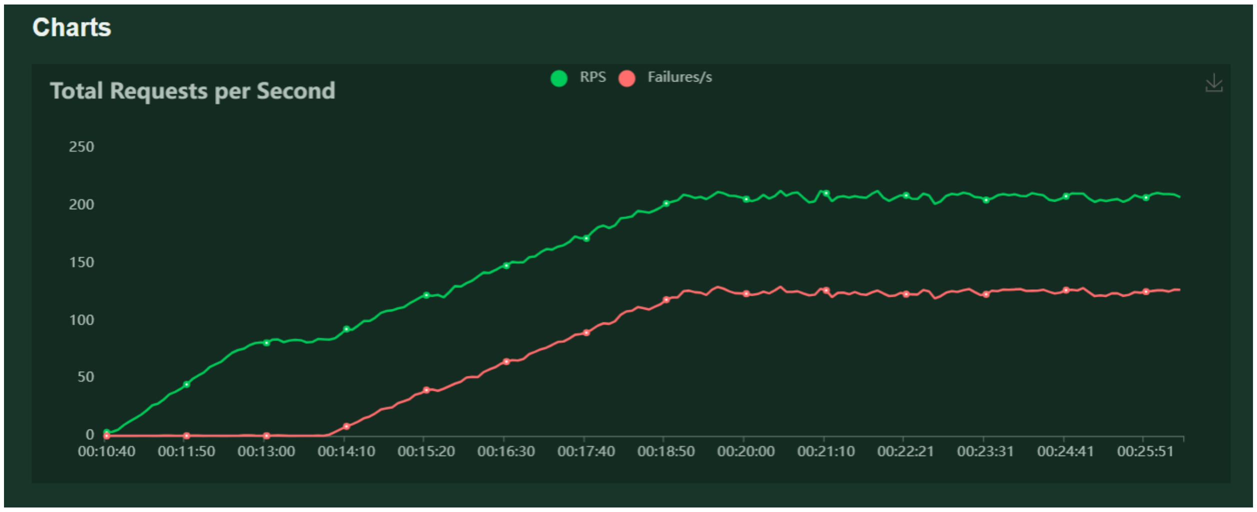
Method	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
GET	/	149040	75367	1613	2	211474	0	157.3	79.6
	Aggregated	**149040**	**75367**	**1613**	**2**	**211474**	**0**	**157.3**	**79.6**

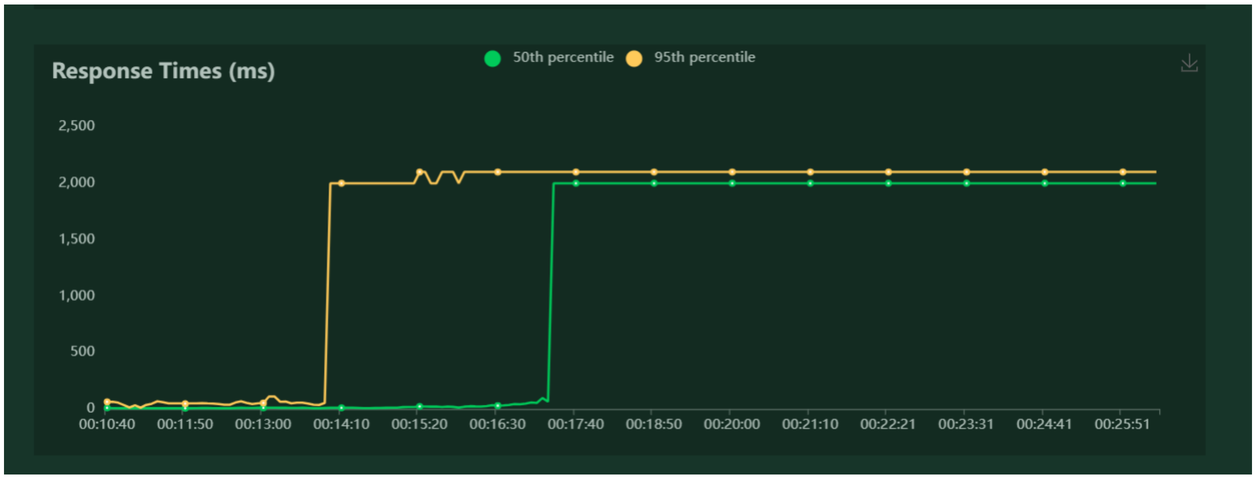
Response Time Statistics

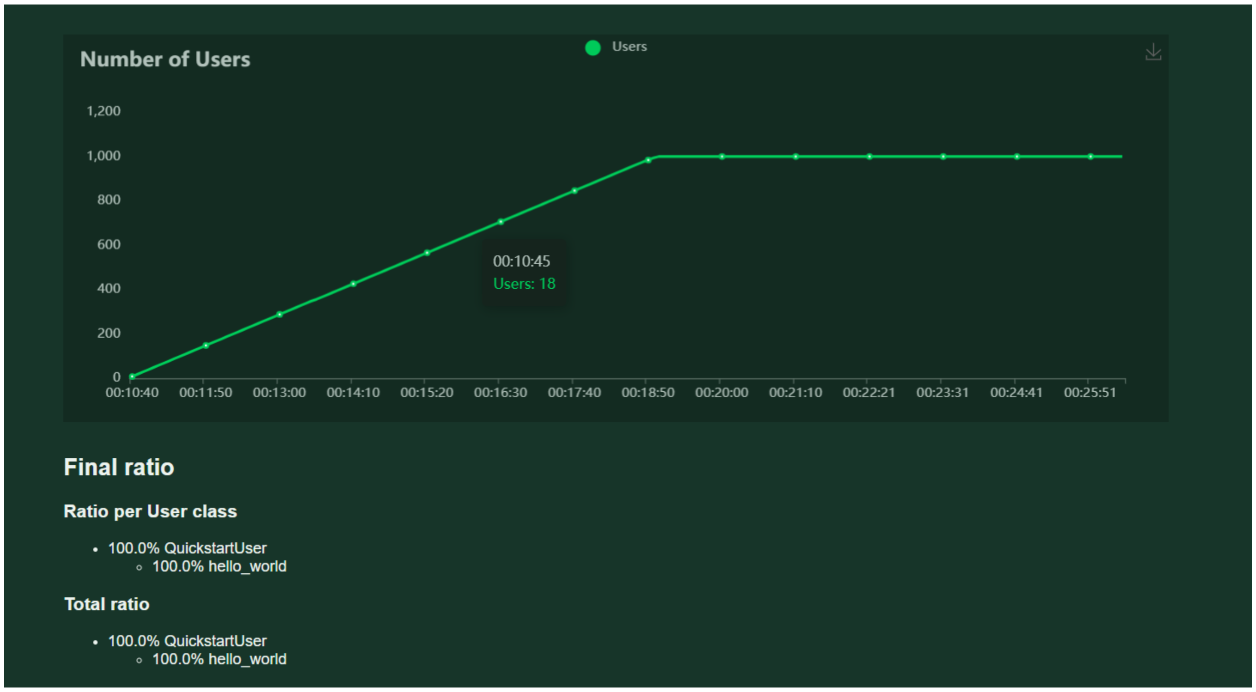
Method	Name	50%ile (ms)	60%ile (ms)	70%ile (ms)	80%ile (ms)	90%ile (ms)	95%ile (ms)	99%ile (ms)	100%ile (ms)
GET	/	2000	2000	2000	2000	2100	2100	2100	211000
	Aggregated	**2000**	**2000**	**2000**	**2000**	**2100**	**2100**	**2100**	**211000**

Failures Statistics

Method	Name	Error	Occurrences
GET	/	Remote end closed connection without response	208







Final ratio

Ratio per User class

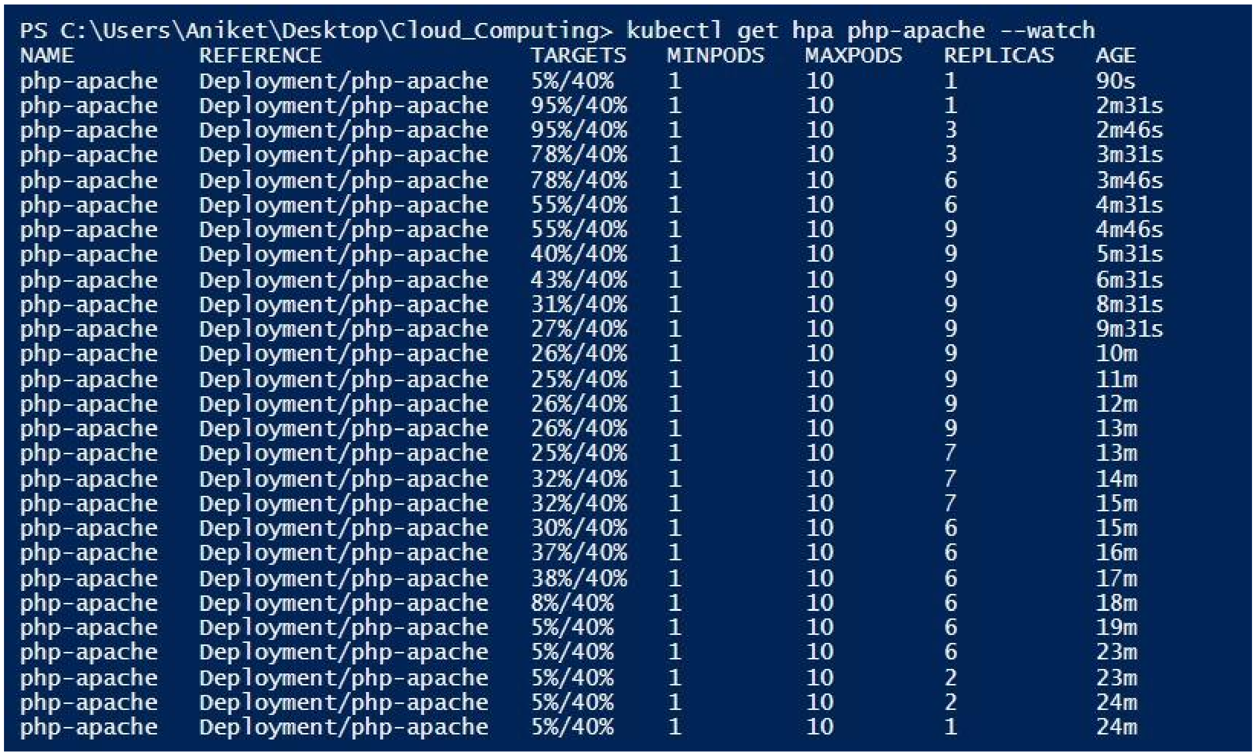
- 100.0% QuickstartUser
  - 100.0% hello_world

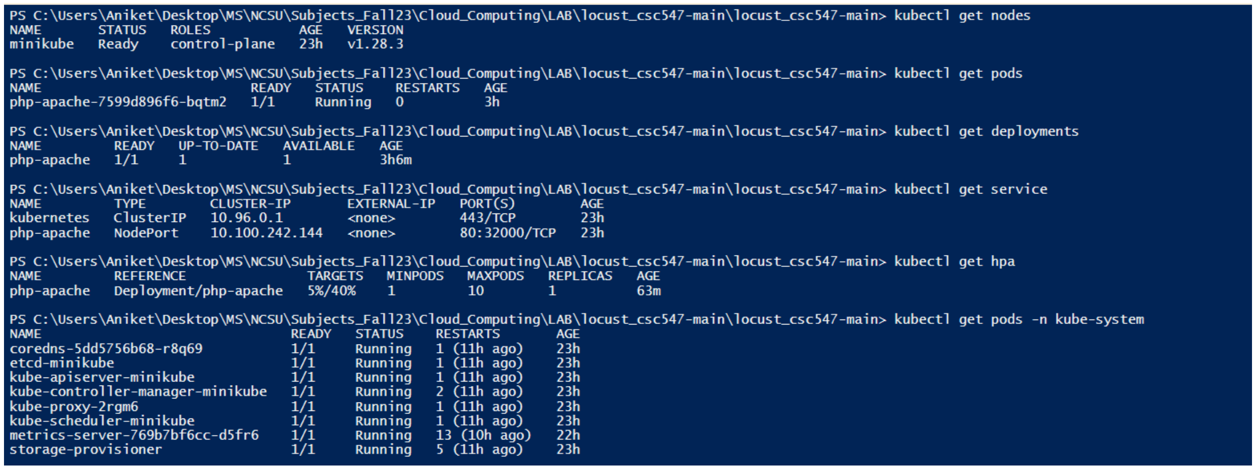
Total ratio

- 100.0% QuickstartUser
  - 100.0% hello_world

6.3 Analysis of the results

- Initially we have only one node which has one pod deployed for our application
- Horizontal Pod Autoscaler (HPA) is then deployed for auto scaling purposes which has the configuration that includes a minimum number of pods = 1 and a maximum number of pods = 10 and the threshold average CPU utilization = 40%, which means that if the average CPU utilization of the application pods goes above 40%, the HPA will scale up the number of pods, and if it is below 40% then it will scale down to maintain the minimum number of pods that is 1.
- When the load started generating with the peak concurrency = 1000 users and requests per seconds = 157.3, the average CPU utilization went to 95% and then the HPA recalculated the desired number of pods and scaled up to 3, and after that, the load started distributing among different pods, after that the CPU utilization was still above 40%, so it scaled up to 6 and then 9. When the average CPU utilization started to come down below 40%, the HPA started scaling down the number of pods as expected and ultimately, the number of pods became 1 which is the minimum number of pods configured by the HPA.
- Response time became constant in the graph after the number of users came to a constant value that is 1000. This is expected because the system gets stabilized after a certain point by the HPA and the load gets evenly distributed among different pods.
- The number of failures/second increased to a certain extent after the number of requesting users became 1000, this is because we used a basic application and only one node in the minikube and since the resources were limited, some failure is expected as the infrastructure cannot handle load after a certain point because this environment was only for testing purposes.


```
PS C:\Users\Aniket\Desktop\Cloud_Computing> kubectl get hpa php-apache --watch
NAME         REFERENCE               TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache   5%/40%    1         10        1          90s
php-apache   Deployment/php-apache   95%/40%   1         10        1          2m31s
php-apache   Deployment/php-apache   95%/40%   1         10        3          2m46s
php-apache   Deployment/php-apache   78%/40%   1         10        3          3m31s
php-apache   Deployment/php-apache   78%/40%   1         10        6          3m46s
php-apache   Deployment/php-apache   55%/40%   1         10        6          4m31s
php-apache   Deployment/php-apache   55%/40%   1         10        9          4m46s
php-apache   Deployment/php-apache   40%/40%   1         10        9          5m31s
php-apache   Deployment/php-apache   43%/40%   1         10        9          6m31s
php-apache   Deployment/php-apache   31%/40%   1         10        9          8m31s
php-apache   Deployment/php-apache   27%/40%   1         10        9          9m31s
php-apache   Deployment/php-apache   26%/40%   1         10        9          10m
php-apache   Deployment/php-apache   25%/40%   1         10        9          11m
php-apache   Deployment/php-apache   26%/40%   1         10        9          12m
php-apache   Deployment/php-apache   26%/40%   1         10        9          13m
php-apache   Deployment/php-apache   25%/40%   1         10        7          13m
php-apache   Deployment/php-apache   32%/40%   1         10        7          14m
php-apache   Deployment/php-apache   32%/40%   1         10        7          15m
php-apache   Deployment/php-apache   30%/40%   1         10        6          15m
php-apache   Deployment/php-apache   37%/40%   1         10        6          16m
php-apache   Deployment/php-apache   38%/40%   1         10        6          17m
php-apache   Deployment/php-apache   8%/40%    1         10        6          18m
php-apache   Deployment/php-apache   5%/40%    1         10        6          19m
php-apache   Deployment/php-apache   5%/40%    1         10        6          23m
php-apache   Deployment/php-apache   5%/40%    1         10        2          23m
php-apache   Deployment/php-apache   5%/40%    1         10        2          24m
php-apache   Deployment/php-apache   5%/40%    1         10        1          24m
```



```
PS C:\Users\Aniket\Desktop\MS\NCSU\Subjects_Fall23\Cloud_Computing\LAB\locust_csc547-main\locust_csc547-main> kubectl get nodes
NAME       STATUS   ROLES           AGE   VERSION
minikube   Ready    control-plane   23h   v1.28.3

PS C:\Users\Aniket\Desktop\MS\NCSU\Subjects_Fall23\Cloud_Computing\LAB\locust_csc547-main\locust_csc547-main> kubectl get pods
NAME                          READY   STATUS    RESTARTS   AGE
php-apache-7599d896f6-bqtm2   1/1     Running   0          3h

PS C:\Users\Aniket\Desktop\MS\NCSU\Subjects_Fall23\Cloud_Computing\LAB\locust_csc547-main\locust_csc547-main> kubectl get deployments
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
php-apache   1/1     1            1           3h6m

PS C:\Users\Aniket\Desktop\MS\NCSU\Subjects_Fall23\Cloud_Computing\LAB\locust_csc547-main\locust_csc547-main> kubectl get service
NAME         TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
kubernetes   ClusterIP   10.96.0.1        <none>        443/TCP        23h
php-apache   NodePort    10.100.242.144   <none>        80:32000/TCP   23h

PS C:\Users\Aniket\Desktop\MS\NCSU\Subjects_Fall23\Cloud_Computing\LAB\locust_csc547-main\locust_csc547-main> kubectl get hpa
NAME         REFERENCE               TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache   5%/40%    1         10        1          63m

PS C:\Users\Aniket\Desktop\MS\NCSU\Subjects_Fall23\Cloud_Computing\LAB\locust_csc547-main\locust_csc547-main> kubectl get pods -n kube-system
NAME                               READY   STATUS    RESTARTS       AGE
coredns-5dd5756b68-r8q69           1/1     Running   1 (11h ago)    23h
etcd-minikube                      1/1     Running   1 (11h ago)    23h
kube-apiserver-minikube            1/1     Running   1 (11h ago)    23h
kube-controller-manager-minikube   1/1     Running   2 (11h ago)    23h
kube-proxy-2rgm6                   1/1     Running   1 (11h ago)    23h
kube-scheduler-minikube            1/1     Running   1 (11h ago)    23h
metrics-server-769b7bf6cc-d5fr6    1/1     Running   13 (10h ago)   22h
storage-provisioner                1/1     Running   5 (11h ago)    23h
```

Results

- When the load was not there, the expectation was to maintain a minimum number of pods=1 configured by HPA when the average CPU utilization was less than 40%. The same is the actual output
- When the load was applied, the average CPU utilization went above 40%, and the HPA scaled up the pods as expected and the load distributed among different pods and again when the CPU utilization came down the HPA scales down the number of pods as expected.

7 Ansible playbooks

Skipped

8 Demonstration

Skipped

9 Comparisons

Skipped

10 Conclusion

10.1 The lessons learned

The lessons we have learned during the designing of this project are as follows:

- Designing a live streaming application often involves trade-offs between various technical requirements like scalability, low latency, cost optimization, and high availability. Achieving one requirement might affect others, emphasizing the importance of balancing these needs to meet overall objectives. This proved to be quite challenging and helped learning crucial aspects of different criterias.
- Exploring and implementing AWS services like Elastic Load Balancing (ELB), CloudFront CDN, was fascinating. The ability to optimize content delivery and reduce latency through edge computing was exciting to learn about.
- Understanding the importance of each section and preparing the design documentation based on the understanding and requirements of the application made us dig deeper in concepts of AWS and Kubernetes. This surely helped in enhancing our knowledge of the subject.
- When we first started the project it was a bit difficult for us to differentiate between the requirements from the cloud architecture perspective and software engineer perspective. But during the process of the project our understanding got a lot better and now we can clearly differentiate the two.

In summary, the project had its mix of challenging aspects, intriguing explorations, routine tasks, and unforeseen discoveries. The balance between technical challenges and interesting implementations made the process both demanding and enriching.

10.2 Possible continuation of the project

Skipped

11 References

1) **AWS Auto Scaling: https://us-east-1.console.aws.amazon.com/awsautoscaling/home?region=us-east-1#home**

2) **Amazon EC2: https://aws.amazon.com/ec2/**

3) **Multi-AZ deployments: https://aws.amazon.com/rds/features/multi-az/**

4) **Elastic Load Balancing(ELB): https://aws.amazon.com/elasticloadbalancing/**

5) **Amazon Virtual Private Cloud(VPC): https://aws.amazon.com/vpc/**

6) **Amazon S3 Overview: https://aws.amazon.com/s3/**

7) **Elastic Kubernetes Service (EKS) Overview: https://aws.amazon.com/eks/**

8) **CloudWatch Overview: https://aws.amazon.com/cloudwatch/**

9) **DynamoDB Highlights: https://aws.amazon.com/dynamodb/**

10) **Identity and Access Management (IAM) Significance: https://aws.amazon.com/iam/**

11) **Amazon Relational Database Service (RDS) Summary: https://aws.amazon.com/rds/**

12) **Amazon API Gateway: https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html**

13) **AWS Lambda: https://docs.aws.amazon.com/lambda/latest/dg/welcome.html**

14) **Amazon Redshift: https://docs.aws.amazon.com/redshift/latest/gsg/new-user-serverless.html**

15) **Amazon Route 53: https://docs.aws.amazon.com/Route53/latest/DeveloperGuide/Welcome.html**

16) **Cloud Pricing Comparison - Cast AI - https://cast.ai/blog/cloud-pricing-comparison-aws-vs-azure-vs-google-cloud-platform/**

17) **Security Pillar - AWS Well-Architected Framework -** **https://docs.aws.amazon.com/pdfs/wellarchitected/latest/security-pillar/wellarchitected-security-pillar.pdf#welcome**

18) **Operational Excellence Pillar - AWS Well-Architected Framework -** **https://docs.aws.amazon.com/pdfs/wellarchitected/latest/security-pillar/wellarchitected-security-pillar.pdf#welcome**

19) **Amazon Well Architected Framework - Operational Excellence Whitepaper -** **https://docs.aws.amazon.com/pdfs/wellarchitected/latest/operational-excellence-pillar/wellarchitected-operational-excellence-pillar.pdf#welcome**

20) **Amazon Well Architected Framework - Reliability Whitepaper -** **https://docs.aws.amazon.com/pdfs/wellarchitected/latest/reliability-pillar/wellarchitected-reliability-pillar.pdf#welcome**