Scalable Energy-Aware Platform as a Service for Video Streaming

A Project Report
Presented to
The Faculty of the College of
Engineering
San Jose State University
In Partial Fulfillment
Of the Requirements for the Degree

Master of Science in Software Engineering

By
Fulbert Jong, Ayaz Khan, Sarthak Singhal, Mudambi Seshadri Srinivas

May/2020

APPROVED

DocuSigned by:
Park Younghee
504312F88D094ED...

5/8/2020

Dr. Younghee Park, Project Advisor

Dan Harkey, Director, MS Software Engineering

Xiao Su, Department Chair

Acknowledgment

We would like to thank our advisor, **Prof. Younghee Park**, for the patient guidance, encouragement, and advice she has provided throughout the course of the project. We have been extremely fortunate to have an advisor who cared and responded to our queries and proposals so promptly.

Table of Contents

ABSTRACT

Video Streaming constitutes more than half of the downstream traffic worldwide. Based on Sandvine Global Internet Phenomena Report on October 2018, streaming content makes up 57.7% of downstream traffic. With a huge amount of incoming requests, video streaming service requires a scalable distributed architecture to handle the traffic. A consistent, highly-available and fault-tolerant system can be developed and deployed on existing cloud solutions such as AWS.

To fulfill such demand, video streaming service providers need to develop an architecture for cost-effective resource utilization. Instead of focusing on the actual business plan, efforts are dissipated in designing coherent and complex architecture. Moreover, the lack of definitive analytics makes it difficult to maintain the system's availability and scalability at a dynamic pace. Therefore, there is a need to decouple existing architecture into a self-sustained cloud-assisted solution that matches the requirements of an ever-growing market of video services.

We propose a Platform as a Service (PaaS) solution to host video streaming content. The idea is to design a better cloud orchestration model that is inherently scalable in terms of delivery and storage. This elasticity can be achieved by capitalizing the container technology and integrating the load prediction mechanism for energy-aware scheduling. By adapting to this self-adjustable infrastructure, we aim to enhance the system throughput and improve content delivery experience.

Chapter 1. Project Overview

1.1 Introduction

Video streaming technologies have become a major focus as online content demands are growing exponentially. The evolution of tools and solutions to provide seamless video content to end users has left a trail of knowledge to be analyzed in this search. The work accomplished by video streaming giants like Netflix and Hulu in providing a quality-oriented subscription-based service is to be examined to dive deep into the latest solutions of the end to end video delivery process. Therefore, this literature search takes a systematic approach to study the various solutions adopted by these successful content providers. The aim is to answer the research questions arising from the abstract of the project and determining the relevance that our solution will hold in the current state of video provision.

Netflix and Hulu are the top content service providers on the international stage today. The downstream traffic in the US has been dominated by Netflix alone, ranging from 29.7% in 2011 to 30.71% and sometimes even 40% at peak hours today. According to the latest Global Internet Phenomena Report from Sandvine, a vendor of bandwidth-management systems, Netflix consumes a significant 15% of all internet bandwidth globally followed by HTTP media streams, representing 13.1% of all downstream traffic; YouTube (11.4%); web browsing (7.8%); and MPEG transport streams (4.4%). The report also discovered that video now accounts for 58% of consumer-side Internet traffic [2]. As Vijay K. Adhikari and others state that understanding the system and performance of Netflix and Hulu can reveal the design of such large-scale video streaming platforms, and help improve the design of future systems [1], it is imperative to dissect the Netflix architecture to formulate a robust design.

Both Netflix and Hulu video streaming platforms have crucial dependencies on third-party infrastructures, with Netflix relying on the Amazon Web Services, while Hulu hosts its services out of Akamai. However, both of them choose to maintain the set of content distribution networks for video delivery on their own. In this search, the focus shall be on studying these platforms to identify the strategies that are employed to support scalability and optimization as well as to discover the limitations and flaws that require new solutions.

Chapter 2. Related Works

2.1 Systematic Literature Review

The Systematic Literature Review guidelines described by Kitchenham [3] have been used to address the objectives of the literature search as described in the introduction above.

2.1.1 Research Questions

The following research questions were derived from the abstract. These research questions are the foundations of our search for content on video streaming platforms.

RQ1: What factors play important roles in designing a video streaming platform?

RQ2: What are the existing solution architectures for the provision of seamless streaming?

RQ3: What are the limitations and challenges faced by these architectures?

RQ4: What are the in-demand business requirements in the online video market today?

2.1.2 Search Strategies

The strategies used for the search of papers include the identification of keywords accumulated by using previous experience and relevant knowledge in the field. Some important key phrases used for searching papers were ‘cloud-assisted video streaming’, ‘video streaming platforms’, ‘Netflix architecture’, ‘resource management using containers’, ‘load prediction for video traffic’ and ‘video streaming through Kubernetes pods.’ The major databases used for the search were IEEE Xplore and ACM Digital Library. Also, an extensive search on the internet was performed to gather all the related knowledge with cutting edge technologies being employed in the field. Each key phrase with different variations was searched on each database and the most relevant papers were accepted.

2.1.3 Study Selection

The selection process was performed in three steps. (1) shortlisting based on the title, (2) filtering based on the abstract, and (3) further selection based on the extensive scrutiny of the paper. At step 1, around 30 papers, blogs, and journals were discovered, out of which, 20 were filtered out in step 2. In step 3, we accepted 3 papers and a blog that completely dissected the Netflix architecture. The reason for the selection of Netflix architecture as the backbone of our research is that Netflix has the most advanced set video streaming solutions in the present time. Therefore, an insight into its complex infrastructure would allow us to gain more focused information required for designing a scalable platform.

2.1.4 Study Analysis and Overview

The paper talks about the Netflix system design in detail and elaborates upon the key factors responsible for making it the best architecture in the video streaming industry. Netflix design revolves around 3 main components, the Open Connect, backend and client [4, 10]. Open Connect is Netflix's content delivery network that stores Netflix videos in different locations throughout the world [6]. This gives Netflix an advantage of a better quality of service and scalability. The backend follows a microservice-based architecture which gives Netflix tremendous advantages over the traditional video streaming systems. Netflix has more than 100 microservices working in synchronization to provide end to end video content delivery. This answers our RQ1 as microservices and content delivery networks are the major design factors that play a key role in building the video streaming service.

In the paper about the measurement study of Netflix and Hulu [1], the author monitors traffic while performing basic operations like account creation, login and streaming a video finds a dynamic change in the IP addresses of servers involved in the process. All account management is done on Netflix's own IP space, but the movie streaming is done via various other IPs hosted on AWS. This gives us more insight into the RQ2 and RQ3. One key component of Netflix's Architecture is TITUS, which is a container management system that provides Netflix with an Auto-scaling feature.

The containerized streaming aids in traffic distribution among the instances. Victor Medel and others aim to solve the key challenge of elastic behavior in cloud-assisted systems using Kubernetes containerization [9]. This can prove to be a key factor in the project design. Authors Yunyun Jiang and others propose a lightweight prediction algorithm to predict future video traffic and offer a hybrid configuration scheme to further leverage instances [5]. Leveraging traffic prediction can prove to be a critical source of improvement if integrated with container orchestration.

Another notable paper proposes a solution to the transcoding challenge faced by many video streaming services by a Cloud-assisted streaming environment, where scalable and adaptive high-definition video contents can be provided to users without requiring specialized decoding capabilities [8]. There are networking solutions proposed that provide dynamic multipath routing for a video delivery strategy in a large-scale mesh system [7].

2.2 State-of-the-Art Summary

After extensive scrutiny of technologies being used in the existing solutions, it is vital to pick the ones that have proven to be reliable. At the same time, to achieve the goals of the project, better solutions shall be developed using services that are currently not being employed. The microservice architecture has been well tested in terms of efficiency and scalability. Keeping this core idea of the Netflix design, Elastic Kubernetes Service has been selected as the provider of container orchestration. Kubernetes shall provide the auto-scaling feature to not only the streaming instances but also the many microservices that will come into play. Netflix uses the Titus container management service. As we aim to achieve load balancing at a container level apart from instance level, Kubernetes is recognized as the best option.

To provide a platform as a service with stateless microservices, an API gateway is required. The most commonly used API gateways are ZUUL, NGINX, and IBM API Connect. With the research, it was found that both of these are very close in terms of performance.

In a complex microservices design, there is a need for a circuit breaker to await a recovery in case of a single service failure. Netflix uses Hystrix as a solution that handles cascading failures as well as monitors configuration changes. Rate Limiting, a simple implementation of rate control, and Automatic Retrying, an encapsulation of automatic retry logic and exception recovery. As video streaming delivery systems encounter millions of events per day, it is essential to apply distributed system monitoring. This can be achieved through the use of Kafka pipelines to move data to various sinks like S3 or ElasticSearch.

The proposed solution intends to boost the load balancing among containers and provide dynamic resource allocation based on the on-demand model. Load prediction is an essential method that can help fulfill this objective. The research has revealed that Reinforcement Learning can provide an efficient prediction system in terms of video traffic. In addition, Collaborative Filtering can be used to determine the popularity of videos and thus, assist in resource allocation. The integration of these prediction models with the container level load balancing module will result in a self-sustained and energy-aware video streaming platform.

Chapter 3. Project Description

3.1 Project Justification

The impetus for the project is to design and provide a highly scalable, available, and distributed Platform as a Service (PaaS) solution to the clients for hosting their video content. While platforms focussing on content delivery networks, such as Netflix and Hulu, have been identified and studied in the previous research, there is a lack of knowledge regarding a platform that focuses on load-prediction as well as container and microservices orchestration to achieve a seamless and efficient video streaming user experience. The broad topic of video streaming platforms has received attention in recent years, but the underlying orchestration and energy-aware scheduling aren't taken into consideration. Instead, most of the literature concentrates on bandwidth allocation and transcoding.

The video content creators/providers like Netflix, Hulu, Amazon Prime Video, etc. create their own platform along with multiple microservices to manage their content. Although the big giants of the entertainment industry can do that due to the availability of financial prowess and resources, the upcoming streaming services struggle with this. Not only the newcomers waste their resources in developing the complex architecture to support their content, but they are also required to continuously monitor and maintain the interconnection among different components. This impedes them from concentrating on the business strategy, which later de-accelerate their growth in the market. For example, Netflix changed its focus from an online DVD store to a video streaming platform in 2009, but it took them almost 3 years to create a robust architecture to host their content and gain prominence in the market. The entire process involved them making multiple mistakes, from spending time on building their own cloud infrastructure to using monolithic architecture. Netflix gained popularity in the market post its migration to microservices and AWS.

The result of this project will prove to be important for the video content creators/providers because it provides them with a robust Platform as a Service (PaaS) solution for hosting the video streaming content. The project allows the content creators/providers to use an inherently scalable, distributed and fault-tolerant cloud orchestration model for anchoring their videos without worrying about the underlying architecture. Additionally, an energy-aware scheduling and load prediction will help in achieving an efficient-load balancing across the pods within the clusters. Instead of two-level load balancing, the proposed architecture will implement three-level load balancing enabling the application to handle multiple incoming requests from the

users. Consequently, the cost-effective solution will assist in better management and utilization of valuable resources like servers, memory, bandwidth, etc.

3.2 Dependencies and Deliverables

3.2.1 Dependencies

3.2.1.1 Cloud Service Provider

The project proposes a Platform-as-a-Service (PaaS) solution which will be deployed on a cloud service provider such as AWS. As the service provider is responsible for monitoring, maintaining and managing the cloud infrastructure, the project will rely heavily on the services offered by them. Additionally, the development cost of the project could pose a problem as we would be spinning multiple EC2 instances on AWS which will increase the cost exponentially.

3.2.1.2 Data Set for Load Prediction

The proposed solution intends to boost the load balancing among containers and provide dynamic resource allocation based on the on-demand model. Load prediction is an essential method that can help fulfill this objective. Techniques like Reinforcement Learning will provide an efficient prediction system in terms of video traffic. In order to train our model for efficient load balancing, we would require an extensive data set regarding the user traffic on the containers. To achieve a balanced dataset, we would have to generate at least twenty thousand realistic user records. Such a large data set creation could potentially slow down the development of load prediction models

3.2.1.3 Container Level Load Balancing

The project will deliver an energy-aware scheduling and load prediction which will help in achieving an efficient-load balancing across the pods within the clusters. Instead of two-level load balancing, the proposed architecture will implement three-level load balancing enabling the application to handle multiple incoming requests from the users. Server level load balancing will be handled by AWS Elastic Load Balancer to redirect traffic to healthy Amazon EC2 instances, but for managing the load across the containers, we would need to implement a new algorithm from scratch and optimize the internal load balancing in Kubernetes for auto-scaling.

3.2.2 Deliverables

3.2.2.1 User Profiling dashboard for clients

As a PaaS, our platform does not deal directly with the end-users. Instead, our targeted clients are content providers who are looking for a platform to deploy their applications. To provide a pleasant user experience, our project will offer a user dashboard to display information about the user's application that is being hosted. Information about the app such as the number of requests will be provided to the users.

3.2.2.2 Analysis of resource utilization and Metrics monitoring

Our project will also provide resource analysis and monitoring. Specifically, resource metrics such as CPU utilization, and memory usage will be monitored. This information can then be used to verify that our platform is using available resources efficiently. Furthermore, our load balancing and load prediction modules will be using this information to balance and predict the workload of the system.

3.2.2.3 Load Prediction

One module of our platform will be a load predictor. Learning from past resource utilization and metrics monitoring, our predictor will then preemptively predict the number of EC2 instances and containers needed at a given time to supply all the client's needs. Since a start-up time for a new EC2 instance can take around ten minutes, accurately predicting the number of instances needed will improve the response time and overall performance of the system.

Chapter 4. System Design and Architecture

4.1 Baseline Approaches

One of the State-of-the-art approaches to implement video streaming service is to use microservices architecture. Microservices architecture allows each service to be run independently with minimal dependency. This is a crucial factor to consider when utilizing a distributed system. The decoupling of services then allows deployment on the cloud.

This paper will use Netflix's approach to video streaming service as a baseline. Netflix deploys its application on Amazon Web Service (AWS). Primarily, Netflix utilizes Amazon Elastic Compute Cloud (EC2) instances as their compute resources. Each EC2 instance is a Virtual Machine based on a specific image and each individual image may correspond to a specific microservice. To balance the workload between instances, Netflix takes advantage of Amazon Elastic Load Balancer (ELB). The Amazon Auto Scaling group service is also utilized to scale the number of EC2 instances according to the workloads.

To further increase resource efficiency, Netflix has adopted containerization technology to deploy its microservices. Using docker containers, Netflix is able to fit multiple containers in a single EC2 instance. To manage all the containers, Netflix has developed its own container orchestration platform, Titus. Titus is built on top of Apache Mesos which was created to manage computer clusters. Similar to other container orchestration platforms, like Kubernetes, Titus is able to manage container deployment. Moreover, Titus is specifically built with Netflix's requirements. Many existing Netflix's frameworks are able to integrate seamlessly with Titus. Moreover, Titus is also integrating the AWS feature to manage resources. Specifically, Titus leverages Amazon EC2 Autoscaling to scale EC2 instances. Titus will then deploy containers on the EC2 instances based on the job requests (i.e. how many resources are needed? Who owns the job? In which server it should be run?). With this container orchestration, Netflix is able to run over a million containers across thousands of EC2 instances to fulfill the enormous workload.

Netflix has decided to open sourced Titus in April 2018. The baseline of the project will be implemented through the use of AWS features and Netflix Titus. Some available Netflix frameworks from Netflix Open Source Software Center will also be utilized as necessary to deploy and manage our microservices. A comparison will then be made between our proposed solution and the baseline.

4.2 System Infrastructure Design

Our proposed scalable video streaming platform solution strives to achieve enhanced resource utilization along with a high user's perceived quality of service(QoS). The cloud infrastructure has been designed to achieve this goal. The Virtual Private Cloud provides an abstract environment to construct the platform as a service solution. The cloud has been divided into two public and two private subnets. The public subnet space is connected to the internet via internet gateway and the private subnet space is connected via NAT gateway. There are two route tables to navigate between the subnet spaces. The application load balancer is provided to handle the traffic among different instances on the public subnet.

One of the public subnet spaces is used for hosting of User Dashboard, Security Layer and Apache Kafka Streaming and Message Queue service. These services directly interact with clients and external Softwares as Services. The other public subnet space is used for the internal processes which are Resource Utilization, Load Prediction Service and Content Management Service. All of these services are required to work in collaboration to provide Quality of Service and container orchestration. The result of this system is forwarded to the Elastic Kubernetes Service in the private subnet space. The Elastic Kubernetes Service hosts the clusters of containers running the microservices required for the video streaming provision. The repositories are saved in the other private subnet space. The Amazon S3 has been chosen for media storage. The user data and metadata will be stored on the MySQL server hosted on RDS. The volatile data generated by the regular tracking of cluster and traffic information is to be stored in the NoSQL database chosen as Apache Cassandra due to its auto-scalable properties.

There are four major microservices that will be part of the scope of the project. The media upload microservice will be a worker machine that handles the uploading of videos on the server. The transcoder handles the preprocessing of the video before it can be streamed over the internet. The media stream microservice is a highly available containerized microservice to stream the videos over the internet. The batch engine is the background microservice that keeps running scheduled tasks like data extraction, analytics provision, preprocessing jobs and other jobs that do not require manual intervention.

There are five major functional components identified for the development of this platform.

1. **Container Management System**
2. **Resource Utilization Manager**
3. **Load Prediction Module**
4. **Web Dashboard**

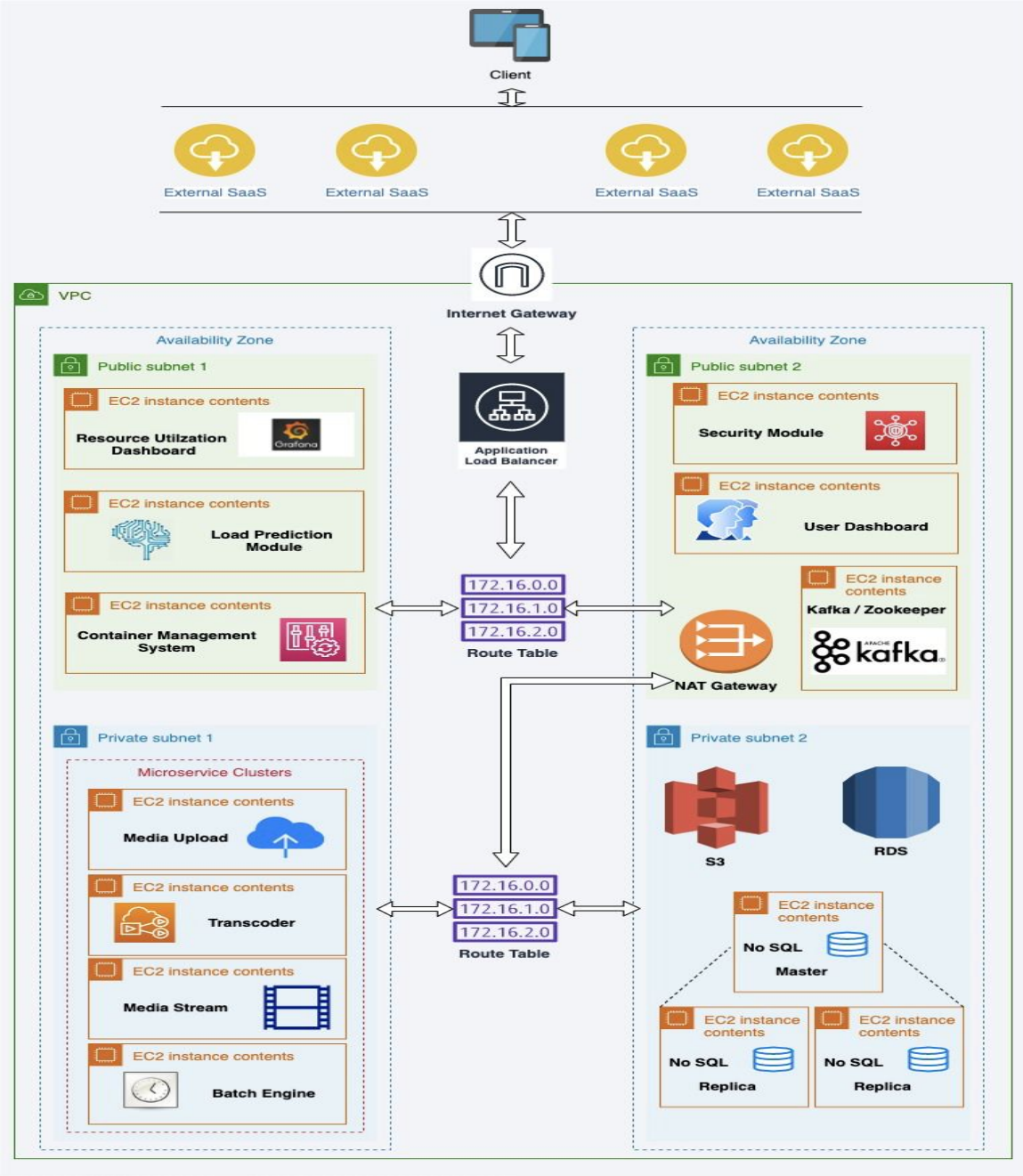


Fig. Cloud Infrastructure Design

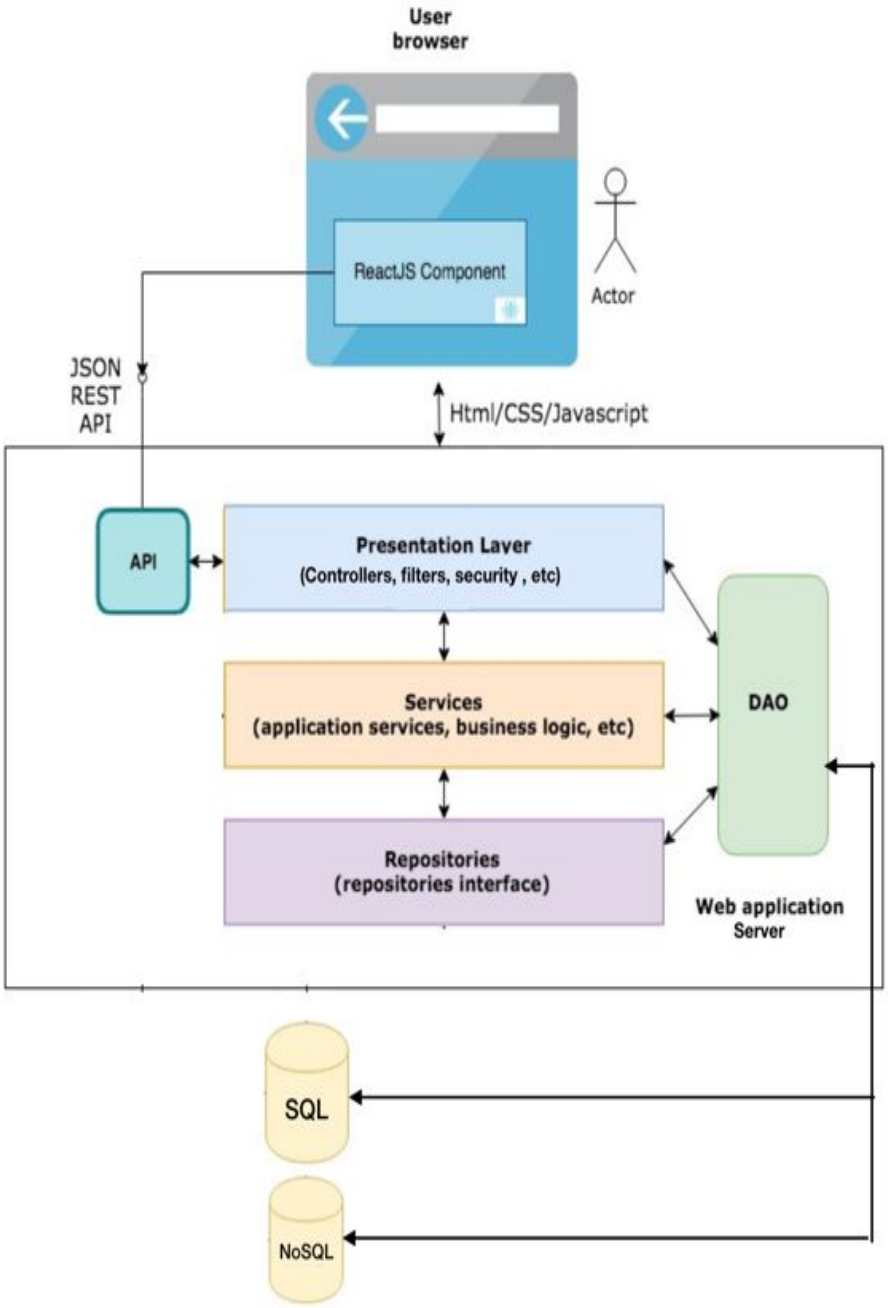


Fig. Dashboard Component API Design

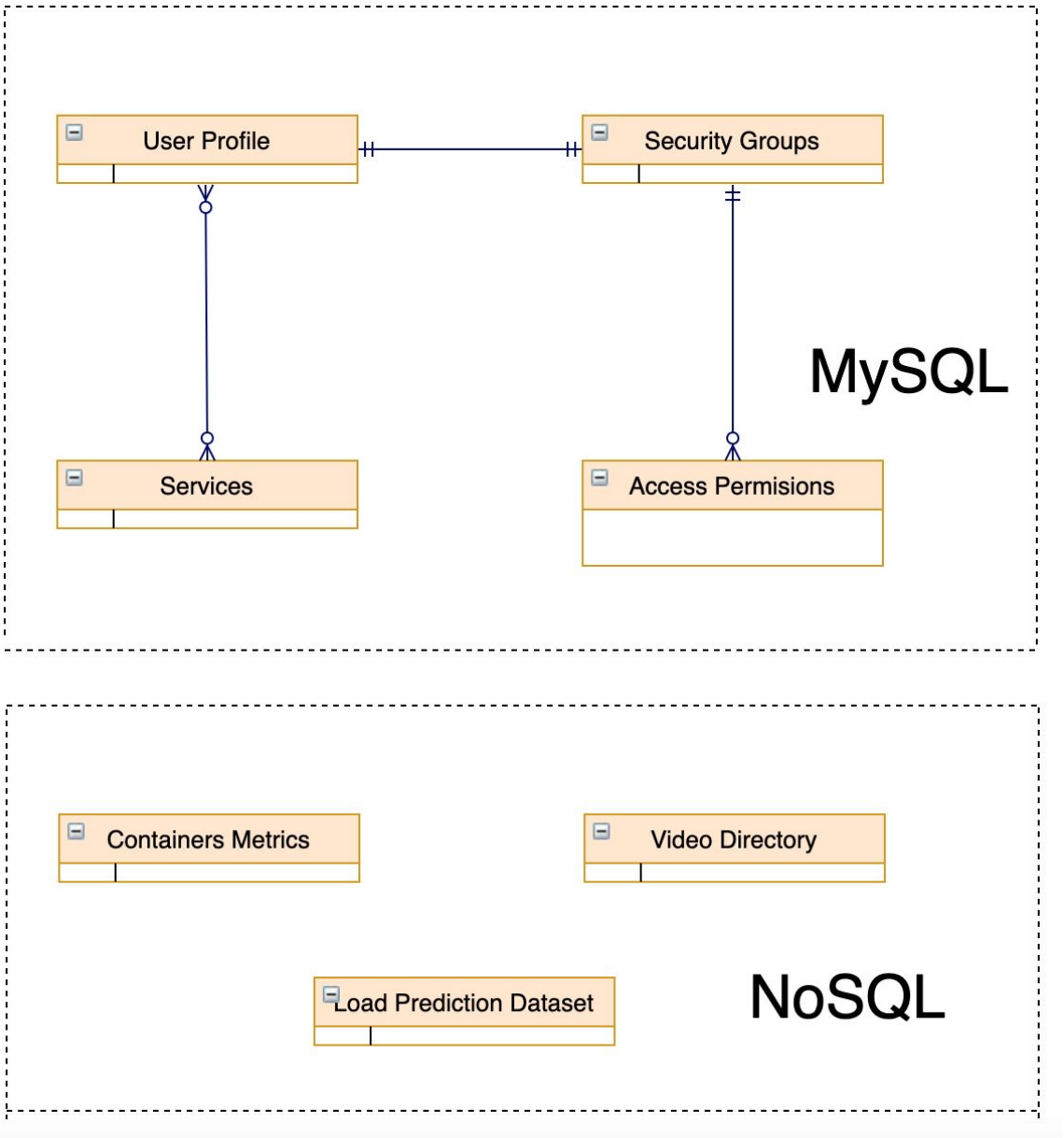


Fig. Overview of Database Design

4.3 System Architecture

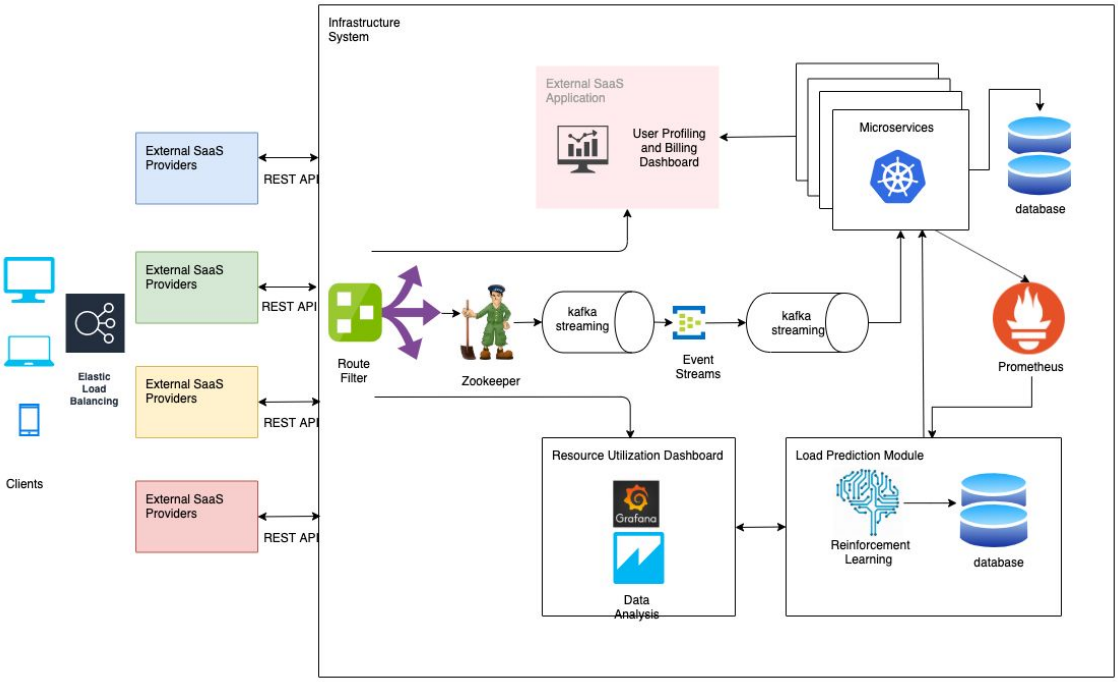


Figure 5.1: Architecture Diagram

A scalable video streaming platform demands dynamic adaptation to changes in the distributed environment and to be able to satisfy the CAP theorem. The system should be fault-tolerant, highly-available delivering high performance with low latency. In reference to the AKF Scale Cube, the resources are scaled in all axes based on their functionality. Container level orchestration of microservices has to be achieved and each of the services is decoupled into several clusters containing a certain number of functional modules. Each resource is a Kubernetes pod running on the cloud instance.

The project "**Scalable Energy-Aware Platform as a Service for Video Streaming**" strives to build a Platform as a Service for hosting video streaming services to achieve a scalable business model. The clients are accessible to different cloud resource provisioning schemes through a web application built to manage infrastructure via REST API. Clients submit the request for the resources through a web interface and based on the requirements, YAML descriptor files for different pods are generated and orchestration of different microservices is initiated.

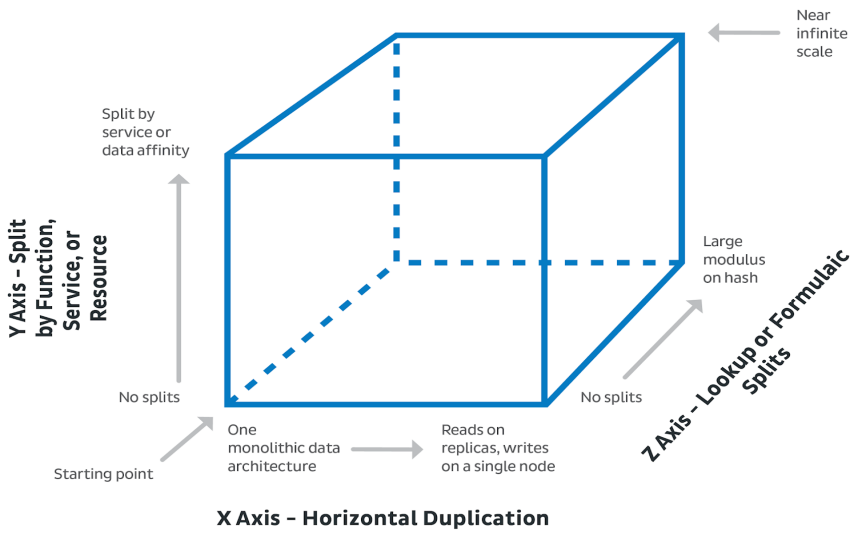


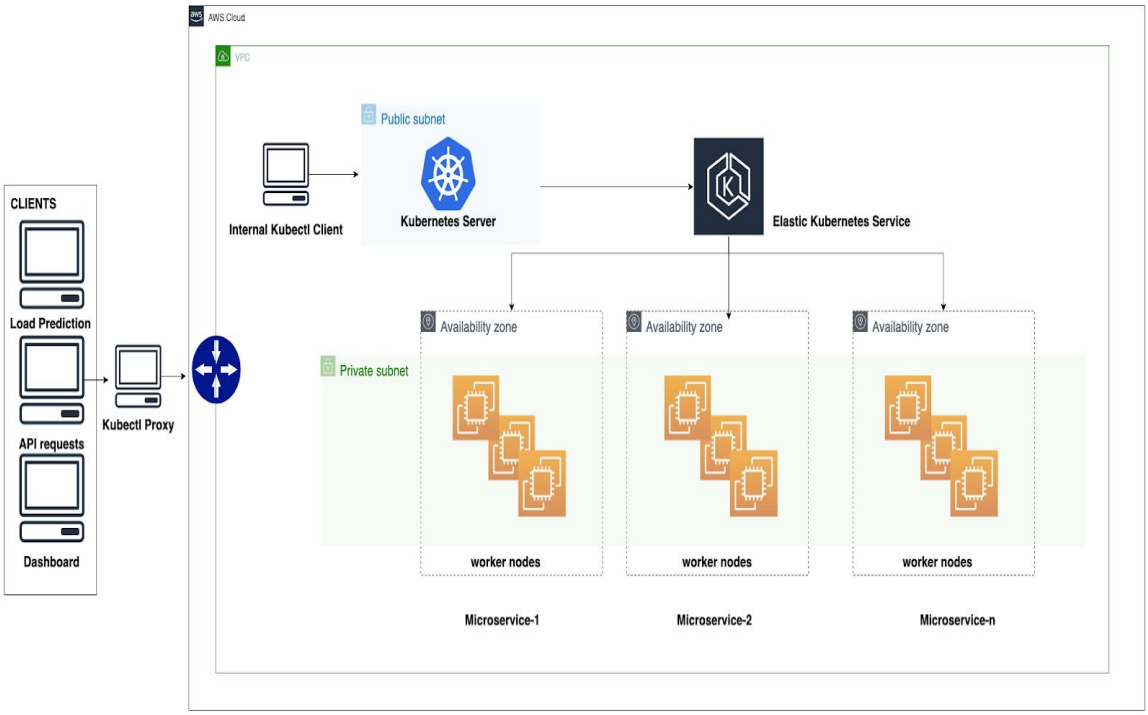
Figure 5.2: AKF Cube Reference

A namespace for each client is provisioned to host the streaming content and a set of Kafka topics are generated and assigned to handle different requests. The requests are redirected to different microservices running as Kubernetes pods. Each of the microservice is enabled with a metrics collection module with the help of Prometheus. Prometheus is an open-source tool used to continuously scrape the required data from different microservices and push it on to the Prometheus server which is later integrated with Grafana, a visualization tool to display different resource usage dashboards. Metrics such as CPU utilization, memory, the health of the pod, network latency, count of restarts of the service, etc. are actively monitored.

The Load prediction module is designed to predict the load on different modules developed as microservices based on the data collected and analyzed. This helps in re-route of the requests in a high traffic environment by effectively determining the characteristics of the service available for a read / write at any given instance. Based on the predicted outcome either migration of resources or circuit-breaking is enabled to maintain robustness across the system. The reinforcement learning model along with deep learning concepts is utilized to achieve this feature. Alarms are configured to be triggered as and when there is a need for new resources or rebalance of the resources. Both SQL and No-SQL databases are implemented based on different requirements. The databases are replicated and sharded to achieve scalability and high-availability.

Chapter 5. Component Design

5.1 Container Management System



Container Management System which is the heart of the system is supported by the Elastic Kubernetes Service of Amazon Web Services. Different modules in the system act as a client to this module that helps in scaling the resources. The set up is maintained by EKS, Kubectl, Kubernetes Server deployed on AWS. Worked nodes provide the base for the different microservices on the private subnet. These worker nodes span across different availability zones in the VPC. Microservices are differentiated by Kubernetes namespace. Each namespace has its own set of resources acting entirely isolated from other services. The main advantage of EKS is the underlying serverless architecture. Code development, Deployment, and Management can be easily done without concern about the underlying architecture.

The requests are queued in Apache Kafka that segregates the type of requests based on the channel. Kafka consumer deployed on this module subscribes to different channels available and performs necessary actions. The requests are redirected to different servers by the zookeeper technology which has information about different topics and partitions available in the streaming

system. The requests are queued in the messaging queue service of Kafka and are served accordingly that overcomes the problem of bombarding the system with requests.

5.1.1 Apache Kafka - Streaming and Messaging Queue Service

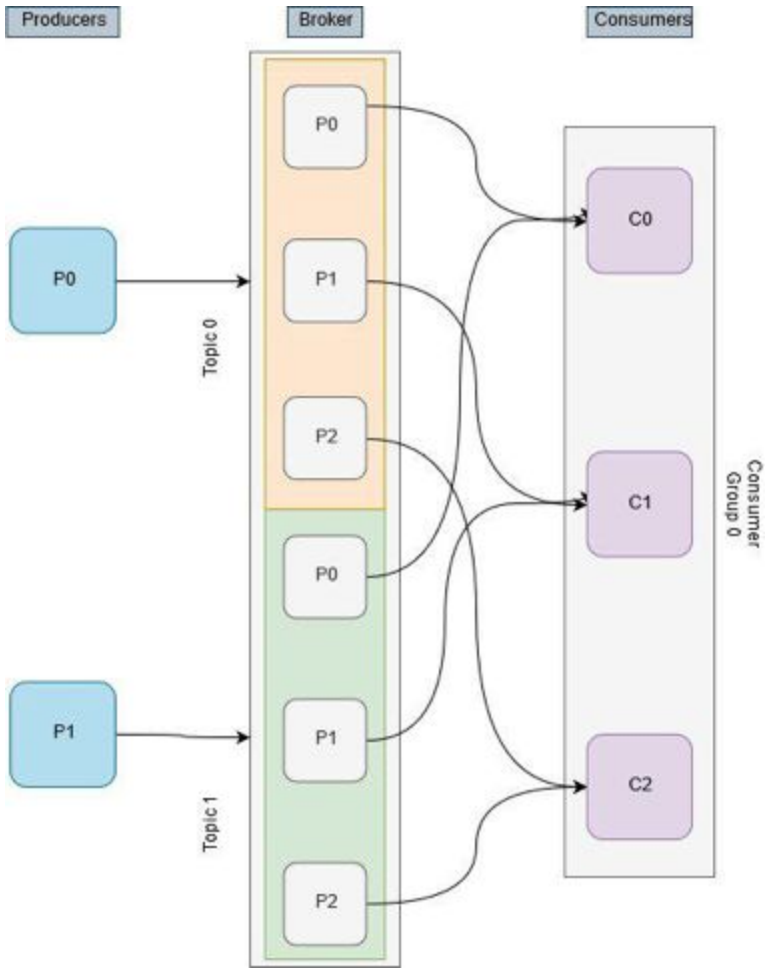


Fig. ***Apache Kafka***

Enterprise integration is one of the key building blocks of an enterprise system architecture, with many open-source offerings available. One of the most widely used messaging systems for integration available today is Apache Kafka. Kafka provides fault tolerance, high resiliency, and low latency for use in real-time applications. Apache Kafka is a distributed streaming application for real-time data processing. Like earlier messaging queue systems such as RabbitMQ, Kafka provides a publish-subscribe API which decouples the message consumption from production by publishing and consuming from topics. However, Kafka differs from messaging queues through its use of an append-only commit log which is replicated across the nodes running Kafka, otherwise known as the brokers. This allows the messages to be replayed

or read after being sent. On the consumer side, nodes read messages from topics, which can be split into partitions and divided amongst available consumers. Kafka also guarantees fault tolerance for both the consumers and brokers.

5.2 Resource Utilization Manager

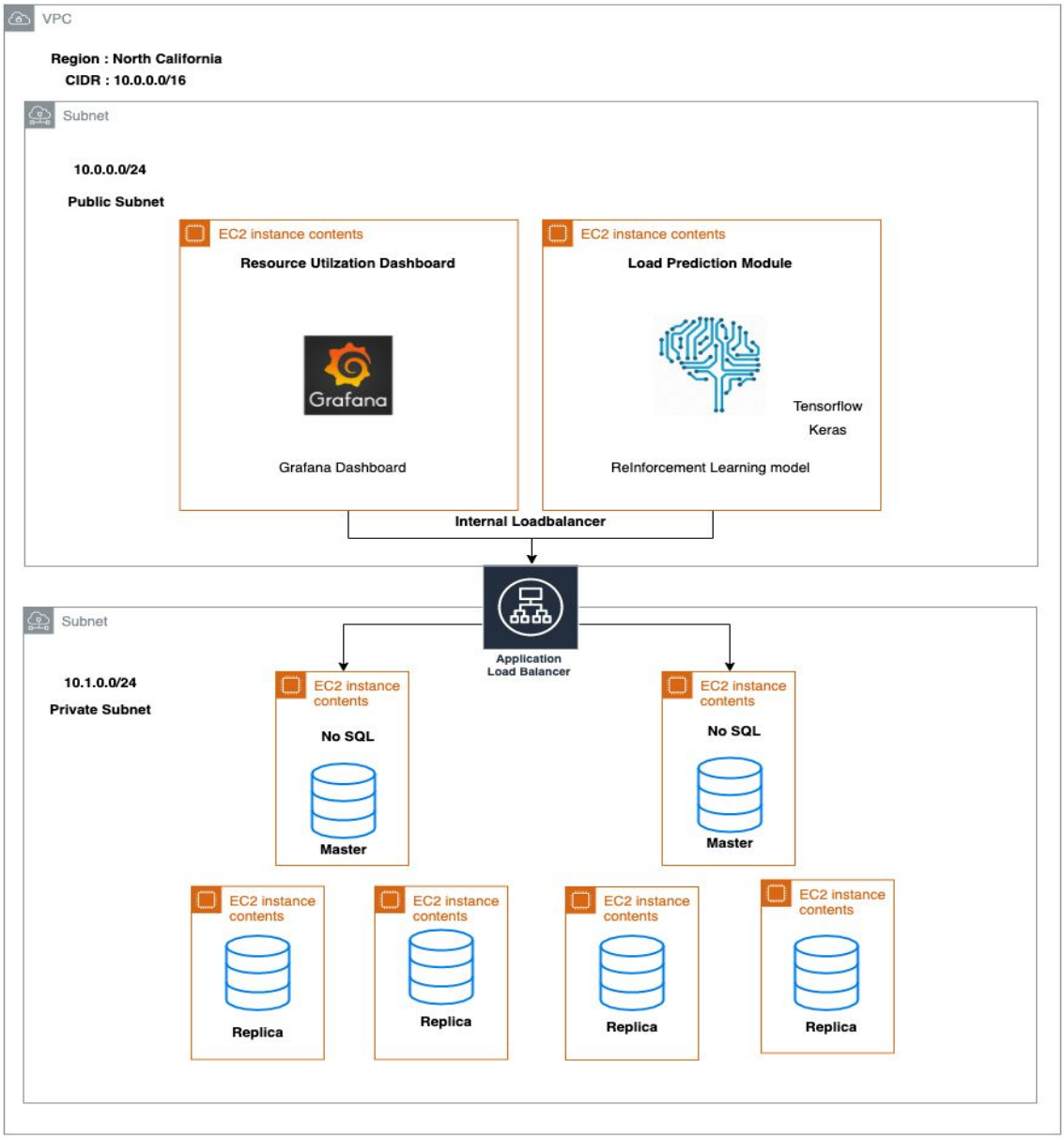


Fig. Depiction of Interactions between Load Prediction and Resource Visualization

5.2.1 Scraping server metrics

Prometheus is a system and services monitoring system and can be integrated easily with any data visualization technologies like Grafana, wavefront. Prometheus continuously collects data from predefined targets set up on a server using a “PULL” methodology and exports the information on to different other services for analysis and visualization. Targets either be configured manually depending on application endpoints or can be system related metrics. In Prometheus terminology, an endpoint that needs metric scraping is called an instance defined as a single process. A Job is a collection of instances offering scalability and reliability. The metrics scrapped from different microservices are parsed and are stored in the respective database. The load prediction module pulls the data from the database and predicts the future trend of the resources in the architecture and accordingly proposes the change. Based on the data produced by the load prediction module, the Kubectl client present on the Container Management System executes the commands that dynamically alters the existing configuration to be able to take up the load.

5.3 Load Prediction Module

Our solution provides a load prediction module that will scale our pods in Amazon EKS according to the predicted load. Our system will implement Prometheus to monitor the Kubernetes cluster. The monitoring metrics will then be periodically stored into our database. After a fixed time interval, the metrics will be passed into our reinforcement learning system. The reinforcement learning algorithm will be used to learn and predict the best action to take for our system.

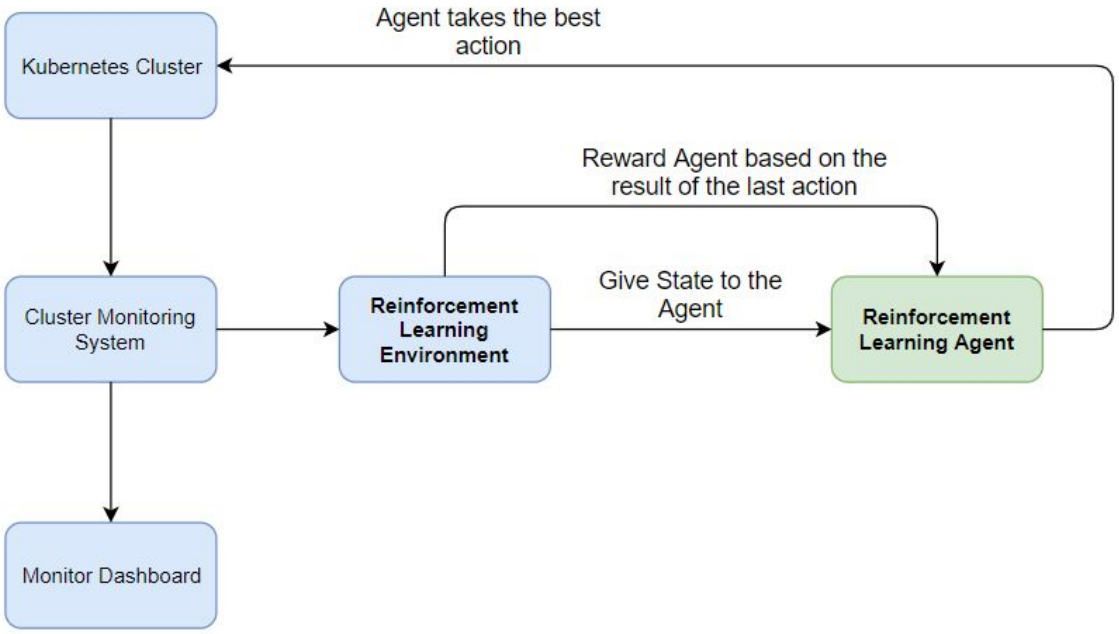


Fig.Reinforcement Learning Overview

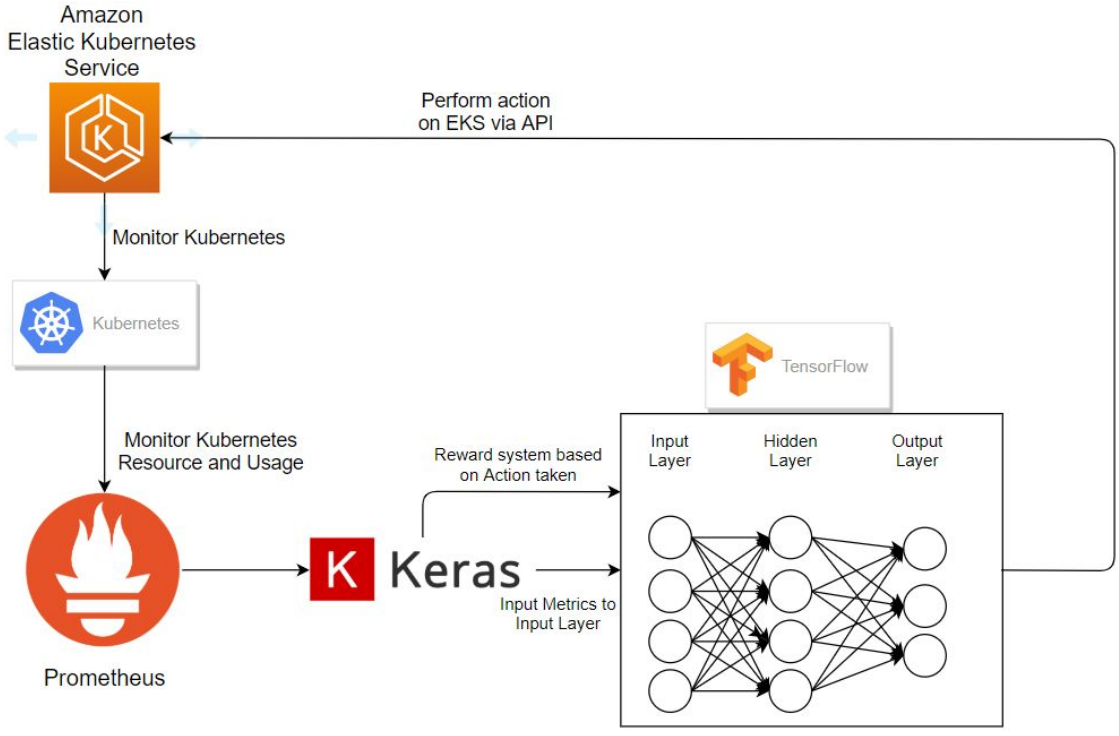


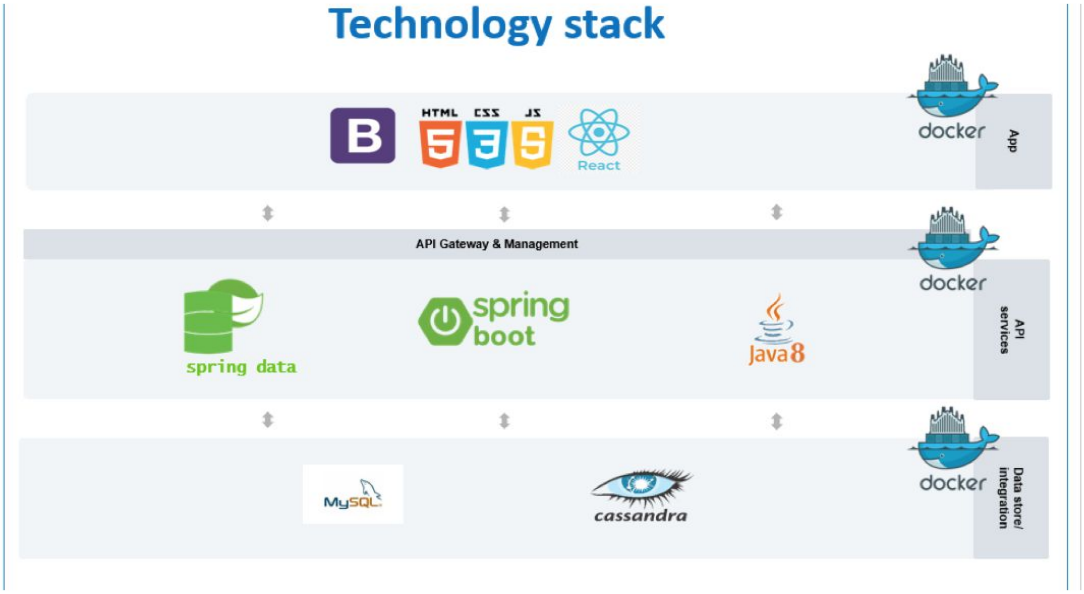
Fig.Reinforcement Learning Technology (DQN)

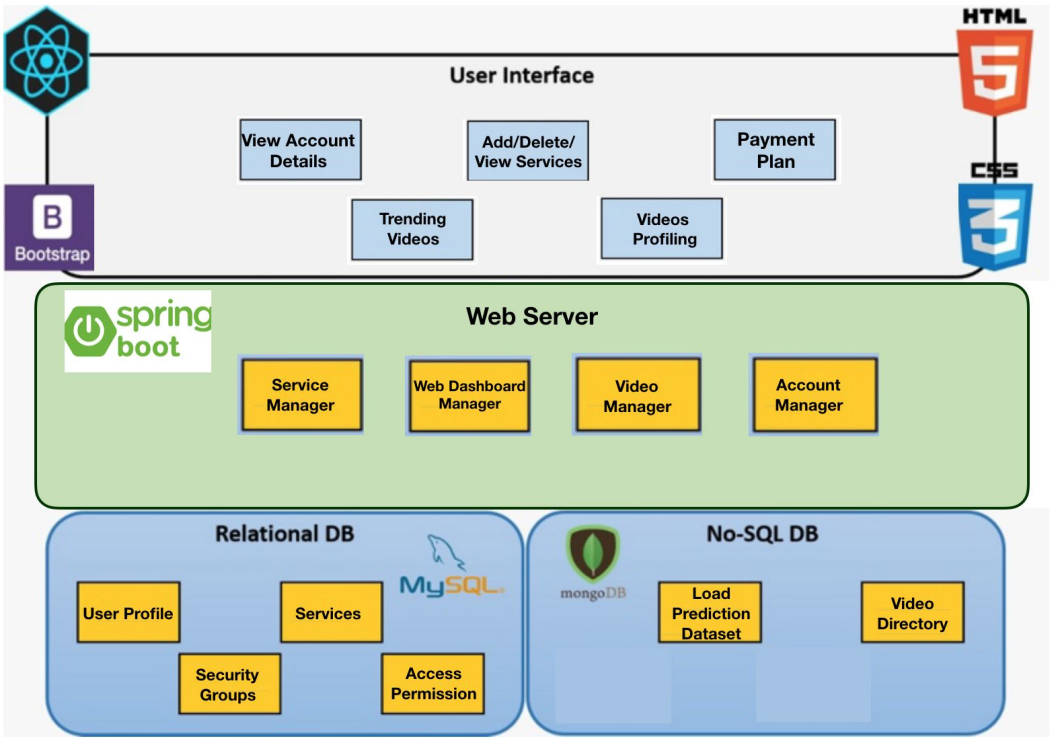
5.4 Web Dashboard

The user Dashboard module will enable the clients to interact with the PaaS for their respective Software-as-a-Service(SaaS) applications. The dashboard will provide a graphical view of the metrics regarding users' hosted videos as well as the microservices that are currently being used for the application. Additionally, different users will get different dashboard views available to them as per the assigned privileges. The interface will allow the users to select any of the following functions:

- View Account Details: The user can view the account details and make changes if necessary
- Payment Plan: The user can view the subscription cost as per the current payment plan they are enrolled in.
- Videos Profiling: The user can see the metrics regarding the hosted videos like the number of requests per video or video popularity per regions

React component is a JavaScript class or function that optionally accepts inputs i.e. properties(props) and returns a React element that describes how a section of the UI (User Interface) should appear. The component on the dashboard will interact with the controllers in the presentation layer via RESTful APIs. The controller will make appropriate service calls to DAO which are responsible for encapsulating the details of the persistence layer and provide a CRUD interface to the database.





Chapter 6. Project Implementation

6.1 Infrastructure Overview

Amazon Web Services based Kuberenetes service called Elastic Kubernetes Service has been employed to serve the infrastructure. A cluster is created on the EKS for the Video-Streaming platform which is hosted across different worker-group nodes that are under different availability zones. An IAM user is mapped to the cluster and the worker-groups for the cluster management.

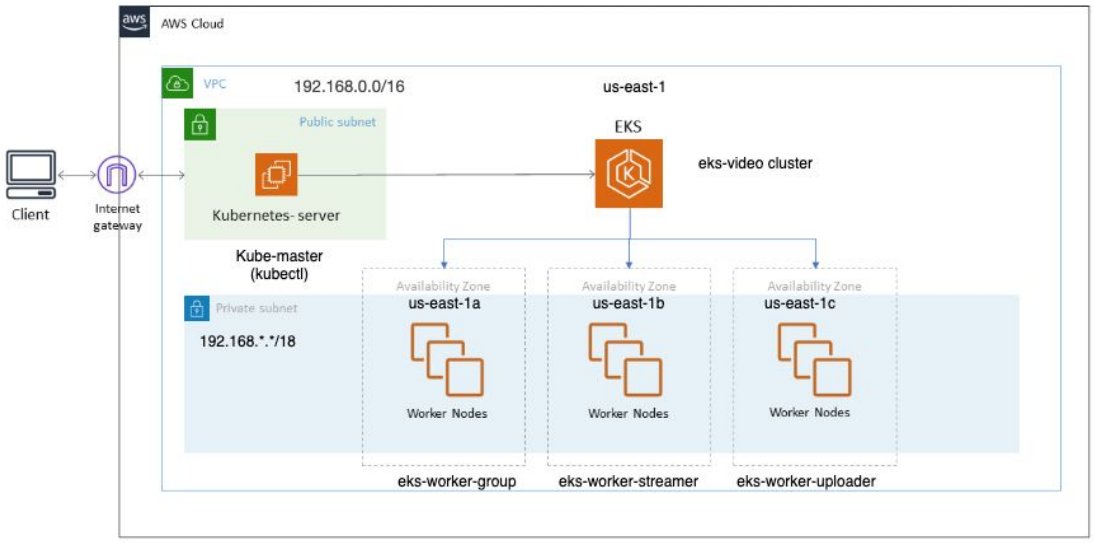


Figure: Overview of the EKS Infrastructure on Amazon Web Services

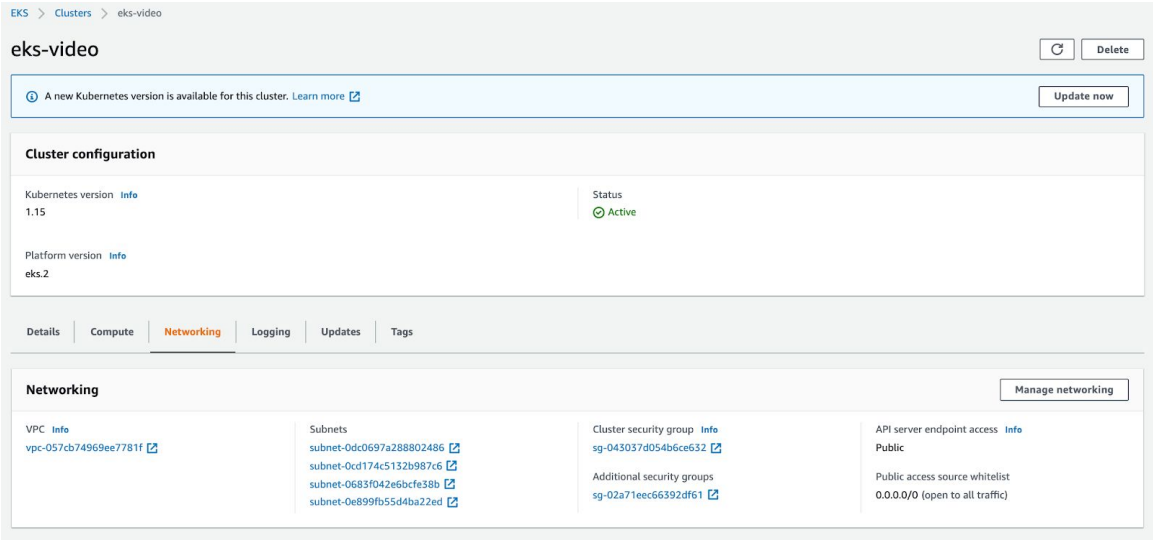


Figure: Overview of the EKS cluster setup

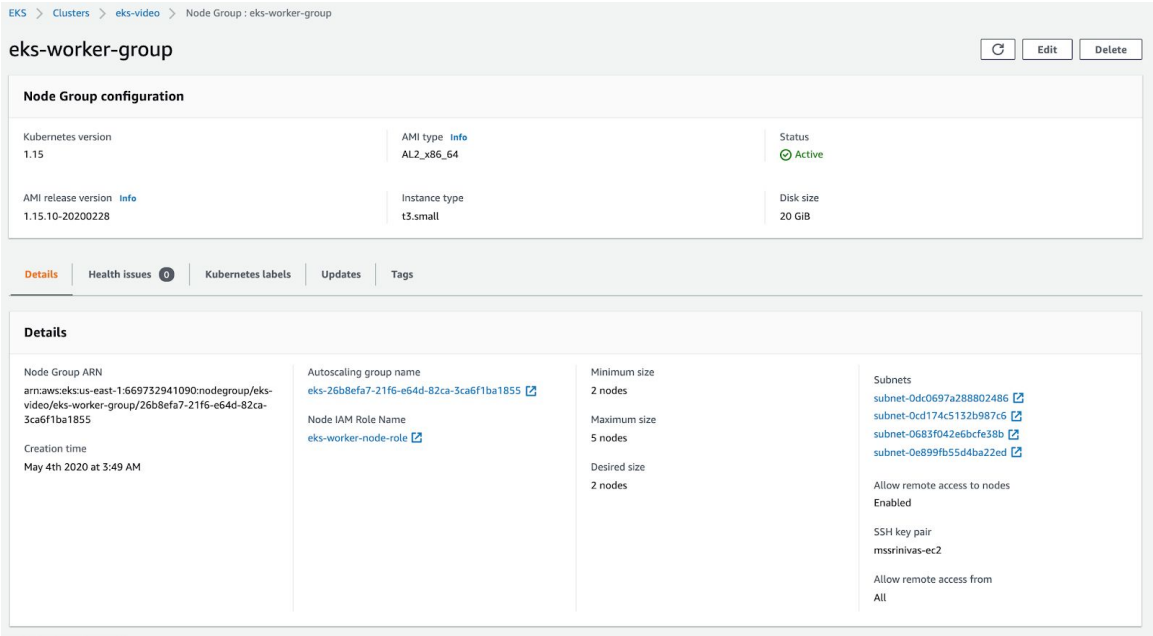


Figure: Overview of the EKS worker group setup

EKS Cluster Configuration:

```
[ec2-user@ip-172-31-47-5 ~]$ kubectl config view
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: DATA+OMITTED
    server: https://609C65EEB7BD5CD7D21390C308402895.gr7.us-east-1.eks.amazonaws.com
  name: arn:aws:eks:us-east-1:669732941090:cluster/eks-video
contexts:
- context:
    cluster: arn:aws:eks:us-east-1:669732941090:cluster/eks-video
    user: arn:aws:eks:us-east-1:669732941090:cluster/eks-video
  name: arn:aws:eks:us-east-1:669732941090:cluster/eks-video
current-context: arn:aws:eks:us-east-1:669732941090:cluster/eks-video
kind: Config
preferences: {}
users:
- name: arn:aws:eks:us-east-1:669732941090:cluster/eks-video
  user:
    exec:
      apiVersion: client.authentication.k8s.io/v1alpha1
      args:
      - --region
      - us-east-1
      - eks
      - get-token
      - --cluster-name
      - eks-video
      command: aws
      env: null
[ec2-user@ip-172-31-47-5 ~]$
```

Figure: Overview of the EKS config after the cluster setup

The system totally has five different microservices as different deployments. Each microservice is mapped to its respective namespace and **"service"** is enabled to expose the internal cluster of worker nodes to the outside world by having an internet-facing load balancer.

Currently available microservices:

1. **datamanager**
2. **video-uploader**
3. **video-streamer**
4. **prometheus**
5. **grafana**

```
[ec2-user@ip-172-31-47-5 ~]$ kubectl get namespaces
NAME              STATUS   AGE
datamanager       Active   7h39m
default           Active   3d11h
grafana           Active   3d10h
kube-node-lease   Active   3d11h
kube-public       Active   3d11h
kube-system       Active   3d11h
prometheus        Active   3d10h
videostreamer     Active   7h19m
videouploader     Active   7h26m
[ec2-user@ip-172-31-47-5 ~]$
```

Figure : Namespaces for all the deployments

Initial Configuration:

```
[ec2-user@ip-172-31-47-5 ~]$ kubectl get pods --all-namespaces
NAMESPACE       NAME                                        READY   STATUS             RESTARTS   AGE
datamanager     dms-5c64c4c87-99kt4                         1/1     Running            0          7h11m
datamanager     dms-5c64c4c87-q7jvl                         1/1     Running            0          7h11m
grafana         grafana-78ff696757-tv9x5                    1/1     Running            0          20h
kube-system     aws-node-4m7nr                              1/1     Running            0          20h
kube-system     aws-node-gppg9                              1/1     Running            0          38h
kube-system     cluster-autoscaler-55f677f8cd-wtwjj         0/1     ImagePullBackOff   0          20h
kube-system     cluster-autoscaler-7df7657c9b-ddjkq         1/1     Running            0          20h
kube-system     coredns-59dfd6b59f-9ddjv                    1/1     Running            0          38h
kube-system     coredns-59dfd6b59f-vlcfk                    1/1     Running            0          20h
kube-system     kube-proxy-hmpqx                            1/1     Running            0          38h
kube-system     kube-proxy-tzjbj                            1/1     Running            0          20h
prometheus      prometheus-alertmanager-7f49b47477-fcj5k    2/2     Running            0          38h
prometheus      prometheus-kube-state-metrics-685dccc6d8-5v2z4   1/1     Running            0          20h
prometheus      prometheus-node-exporter-m8vms              1/1     Running            0          20h
prometheus      prometheus-node-exporter-xrf8x              1/1     Running            0          38h
prometheus      prometheus-pushgateway-7f59d58894-9tfnc     1/1     Running            0          20h
prometheus      prometheus-server-79448cfdc6-hbm9t          2/2     Running            0          20h
videostreamer   streamer-94f9c4dd6-lqrcp                    1/1     Running            0          7h13m
videostreamer   streamer-94f9c4dd6-rggqh                    1/1     Running            0          7h13m
videouploader   uploader-578b99d468-9nmlp                   1/1     Running            0          7h9m
videouploader   uploader-578b99d468-smkkz                   1/1     Running            0          7h9m
[ec2-user@ip-172-31-47-5 ~]$ $#
```

Figure : Pods across all the deployments

Services:

```
[ec2-user@ip-172-31-47-5 ~]$ kubectl get services --all-namespaces
NAMESPACE       NAME                            TYPE           CLUSTER-IP       EXTERNAL-IP
          PORT(S)          AGE
datamanager     dms-service                     LoadBalancer   10.100.91.93     a7fac3082e07246d58f635932018ad79-586489182.us-east-1.elb.amazon
aws.com    8802:30327/TCP   7h40m
default         kubernetes                      ClusterIP      10.100.0.1       <none>
          443/TCP          3d11h
grafana         grafana                         LoadBalancer   10.100.201.192   a0981e7d7f27b4329840196562c00fbd-1676795059.us-east-1.elb.amazo
naws.com   80:30894/TCP     3d10h
kube-system     kube-dns                        ClusterIP      10.100.0.10      <none>
          53/UDP,53/TCP    3d11h
prometheus      prometheus-alertmanager         ClusterIP      10.100.118.214   <none>
          80/TCP           3d10h
prometheus      prometheus-kube-state-metrics   ClusterIP      10.100.128.243   <none>
          8080/TCP         3d10h
prometheus      prometheus-node-exporter        ClusterIP      None             <none>
          9100/TCP         3d10h
prometheus      prometheus-pushgateway          ClusterIP      10.100.229.140   <none>
          9091/TCP         3d10h
prometheus      prometheus-server               ClusterIP      10.100.94.64     <none>
          80/TCP           3d10h
videostreamer   video-streamer-service          LoadBalancer   10.100.30.193    a07894e148acc4baca28e5f053137be2-1212568822.us-east-1.elb.amazo
naws.com   8804:32303/TCP   7h22m
videouploader   video-uploader-service          LoadBalancer   10.100.242.151   a6fbf811aba3749efb42e5c02667f7ff-1801499846.us-east-1.elb.amazo
naws.com   8806:32261/TCP   7h29m
[ec2-user@ip-172-31-47-5 ~]$
```

Figure : Services for all the deployments

Deployments:

```
[ec2-user@ip-172-31-47-5 ~]$ kubectl get deployments --all-namespaces
NAMESPACE       NAME                            READY   UP-TO-DATE   AVAILABLE   AGE
datamanager     dms                             2/2     2            2           7h14m
default         eks-iam-test                    0/1     0            0           3d11h
grafana         grafana                         1/1     1            1           3d10h
kube-system     cluster-autoscaler              1/1     1            1           2d14h
kube-system     coredns                         2/2     2            2           3d11h
prometheus      prometheus-alertmanager         1/1     1            1           3d10h
prometheus      prometheus-kube-state-metrics   1/1     1            1           3d10h
prometheus      prometheus-pushgateway          1/1     1            1           3d10h
prometheus      prometheus-server               1/1     1            1           3d10h
videostreamer   streamer                        2/2     2            2           7h15m
videouploader   uploader                        2/2     2            2           7h12m
[ec2-user@ip-172-31-47-5 ~]$
```

Figure : Deployment for all the microservices

```
[ec2-user@ip-172-31-47-5 ~]$
[ec2-user@ip-172-31-47-5 ~]$ kubectl get nodes
NAME                                STATUS   ROLES    AGE   VERSION
ip-192-168-25-216.ec2.internal   Ready    <none>   21h   v1.15.10-eks-bac369
ip-192-168-78-255.ec2.internal   Ready    <none>   39h   v1.15.10-eks-bac369
ip-192-168-84-173.ec2.internal   Ready    <none>   19m   v1.15.10-eks-bac369
[ec2-user@ip-172-31-47-5 ~]$
```

Figure: Worker nodes

Deployments and Services :

Each of the microservice is spinned up with a replication factor of "2" and the respective port exposed. This deployment is further mapped with the "service" inorder to fetch the external load balancer mapping.

Data Management Service:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app.kubernetes.io/name: data-management-service
  name: dms
  namespace: datamanager
spec:
  replicas: 2
  selector:
    matchLabels:
      app.kubernetes.io/name: data-management-service
  template:
    metadata:
      labels:
        app.kubernetes.io/name: data-management-service
    spec:
      containers:
      - image: fjong1/dms:latest
        name: data-management
        ports:
        - containerPort: 8802
```

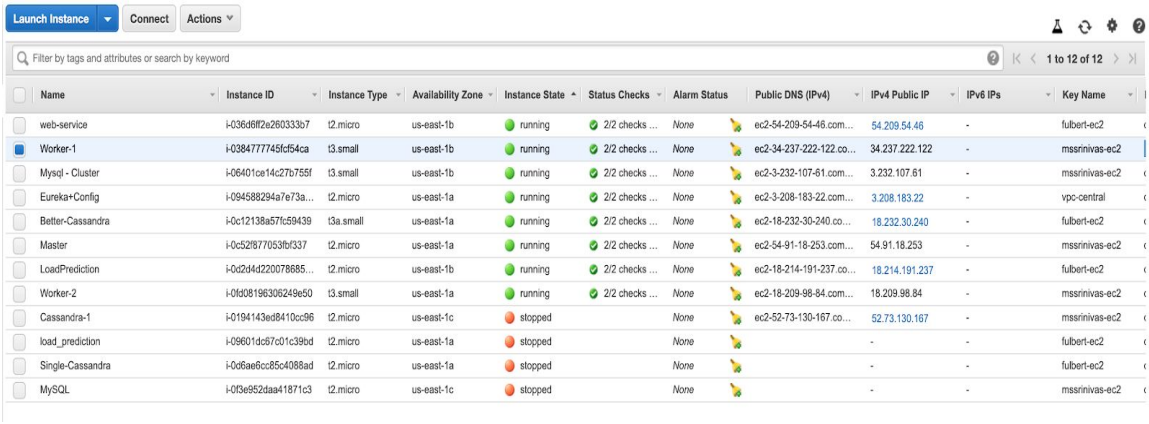
Video Streamer Service:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app.kubernetes.io/name: video-streamer-service
  name: streamer
  namespace: videostreamer
spec:
  replicas:
  selector:
    matchLabels:
      app.kubernetes.io/name: video-streamer-service
  template:
    metadata:
      labels:
        app.kubernetes.io/name: video-streamer-service
    spec:
      containers:
      - image: fjong1/streamer:latest
        name: streamer
        ports:
        - containerPort: 8804
```

Video Uploader Service:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app.kubernetes.io/name: video-uploader-service
  name: uploader
  namespace: videouploader
spec:
  replicas: 
  selector:
    matchLabels:
      app.kubernetes.io/name: video-uploader-service
  template:
    metadata:
      labels:
        app.kubernetes.io/name: video-uploader-service
    spec:
      containers:
      - image: fjong1/upload:latest
        name: data-management
        ports:
        - containerPort: 8806
```

EC2 Dashboard:



The container management service works hand-in-hand with the load prediction module which uses a reinforcement model to predict the next state of the infrastructure and helps in identifying the scaling up/scaling down of the resources.

Below is the visualization of the basic services dashboard representing the infrastructure, dependencies, health status, server metrics.

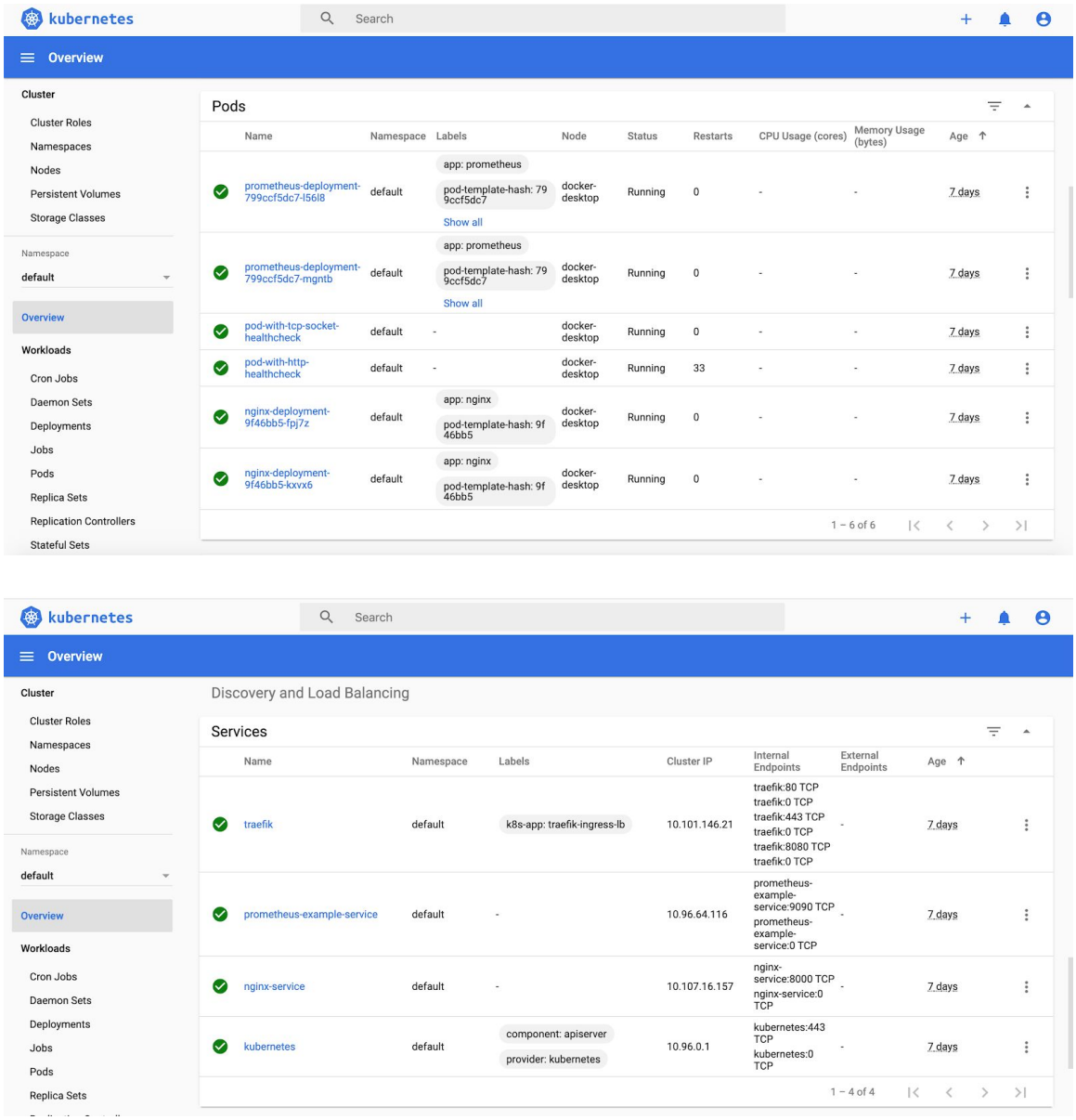
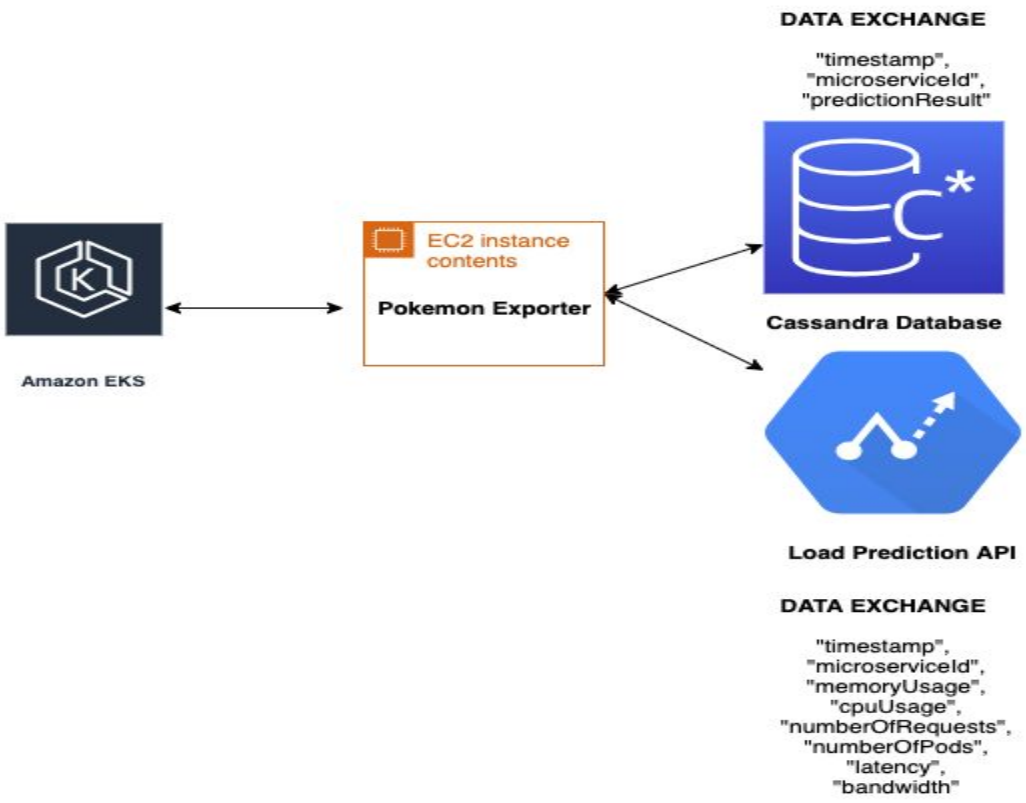


Figure : Kuberenetes dashboard depicting various metrics

Pokemon Exporter



This is a microservice that is independent of the Elastic Kubernetes service and the different deployments that are associated to it. This acts as a data exchange service between different modules. The customized prometheus query data that is being scraped and collected on the respective end points need to be served as input to the Load Prediction module via Cassandra database. The prediction of the module should be monitored continuously for any fluctuations and in order to do so this data should be published to a database. All of this data exchange between the different services is handled by Pokemon Exporter.

PokemonExporter is a python script that has multiple http requests and is scheduled to run for every 5 minutes interval. The data that is being exchanged is per kubernetes namespace.

6.2 Load Prediction

The reinforcement learning that we are using for our load prediction is the Q-learning algorithm. The reward and penalties are set to drive our model to utilize resources efficiently while still able to fulfill all the load requests. The metrics we will be using varies from CPU usage, memory usage, number of pods, latency, and Bandwidth. By monitoring and utilizing these metrics, our system will be able to perform its task.

$$Q_{st,at} = Q_{st,at} + \alpha * \left(r_t + \gamma * \max Q(st+1, a) - Q_{st,at}\right)$$

Since our state-action space is too massive to store, we will be using a neural network as a function approximator for our Q-learning algorithm. Using Stochastic Gradient Descent, our reinforcement learning algorithm will minimize the temporal difference of our model. As our model converges, it will be able to perform the best action in any given state.

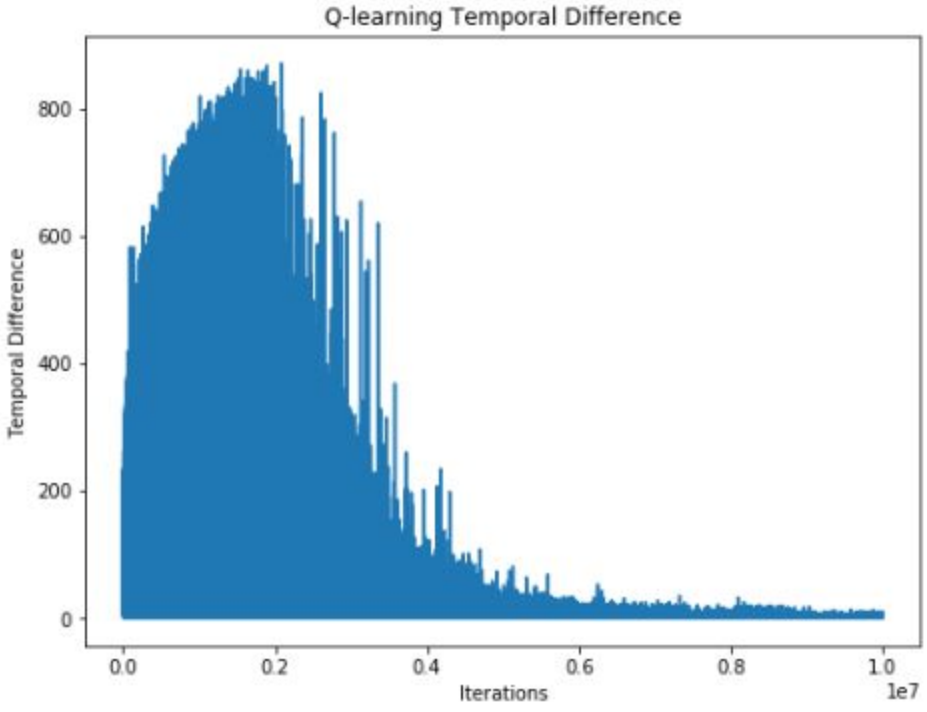


Fig.Q-Learning TD convergence

The metrics will be passed into the input layer of our neural network. The output of the neural network will be the predicted scores for each action at that given state. After extensive exploration, it was found that our system works best with eight hidden layers between the input and output layer to approximate our Q function. All these layers are components of load prediction modules that will suggest the best action based on predicted load by giving commands to Amazon EKS via an API. The change of state from the previous action will be used to reward or penalize the system for taking the action.

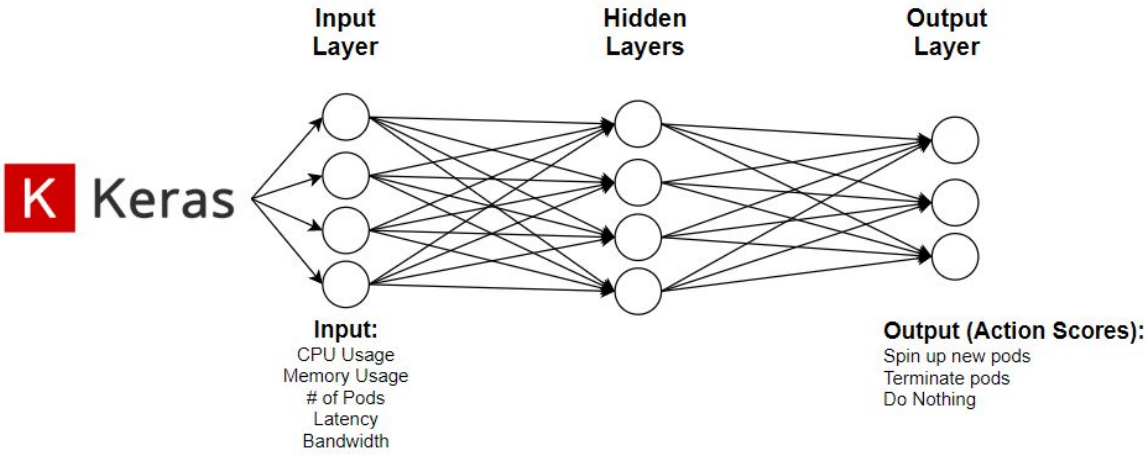
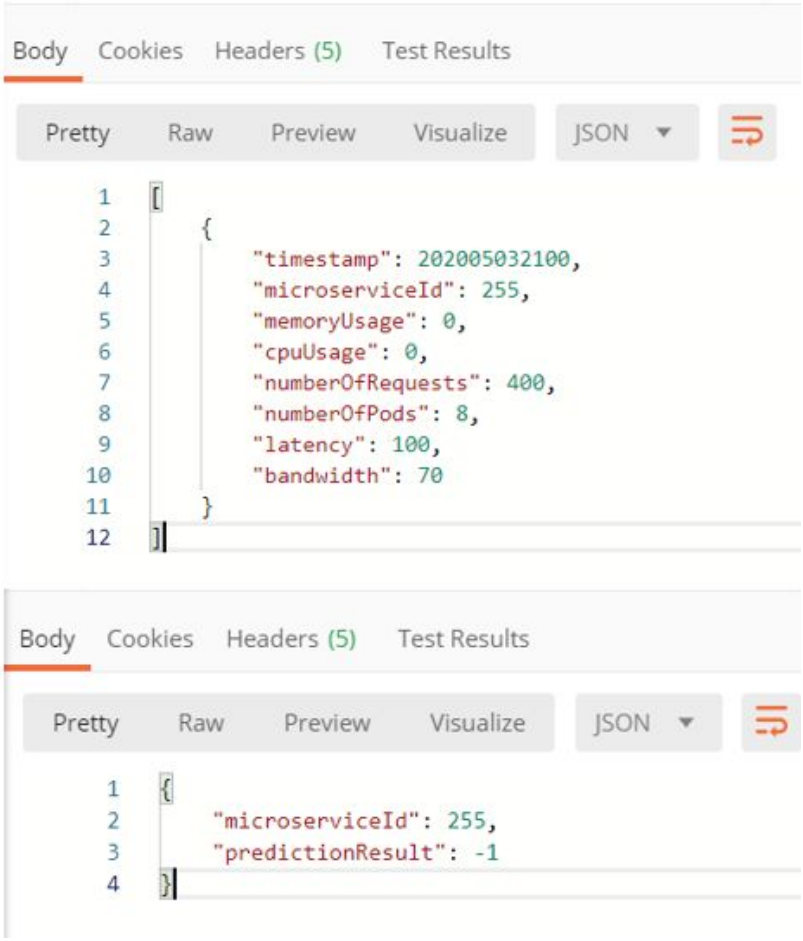


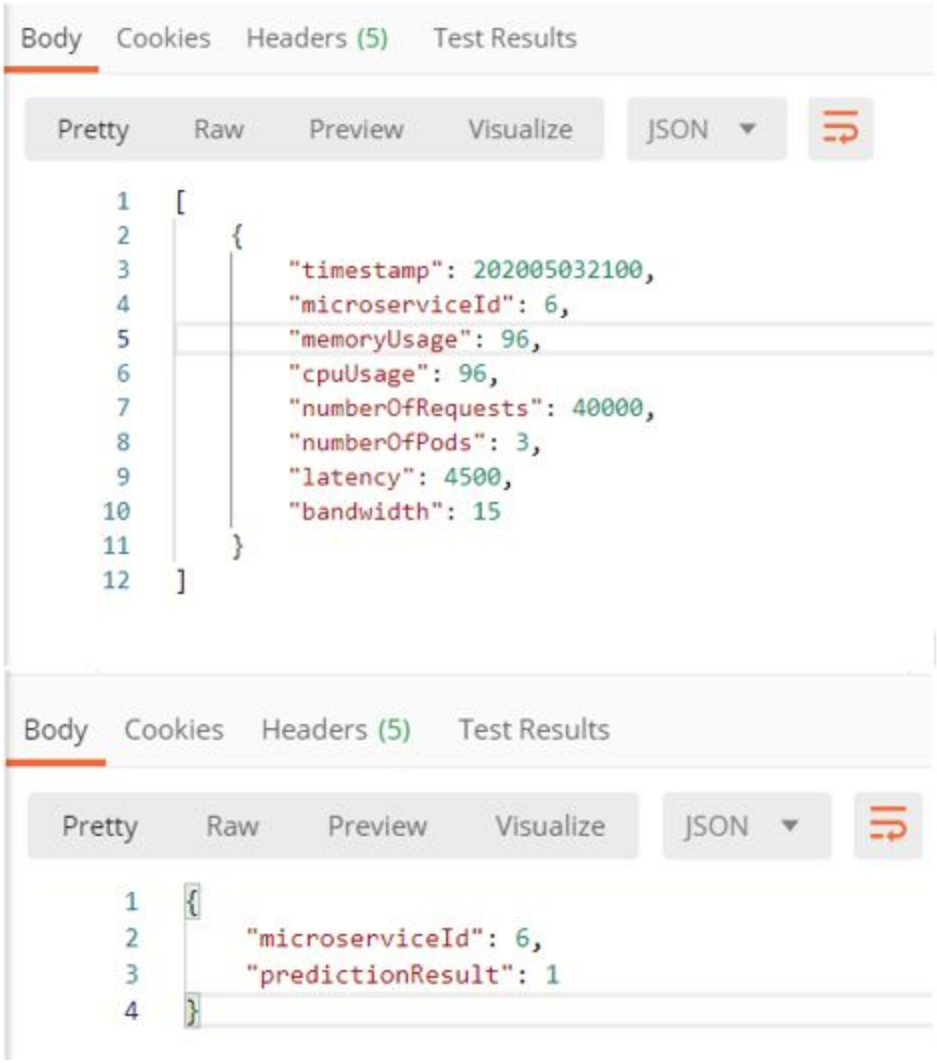
Fig.Neural Network Overview for Reinforcement Learning

Unlike a normal reinforcement learning application, our system does not have an end/done state. Therefore the number of training episodes need to be carefully adjusted to suit our system and time. After our model has explored our environment long enough, our model will switch to exploitation mode and start utilizing the learning experience.

System with excessive resources will correspond to -1 result from our load prediction to indicate the system should decrease the allocated resource.



System under load that lacks resources will correspond to +1 result from our load prediction to indicate the system should increase the allocated resource.



6.3 Web Dashboard

The Web Dashboard module enables the clients to interact with our proposed PaaS for their respective Software-as-a-Service(SaaS) applications. The module has 5 different functionalities -

1. Dashboard - The dashboard provides a graphical view of the metrics regarding users' hosted videos as well as the microservices that are currently being used for the application (video uploader/ video streamer/ data management service).

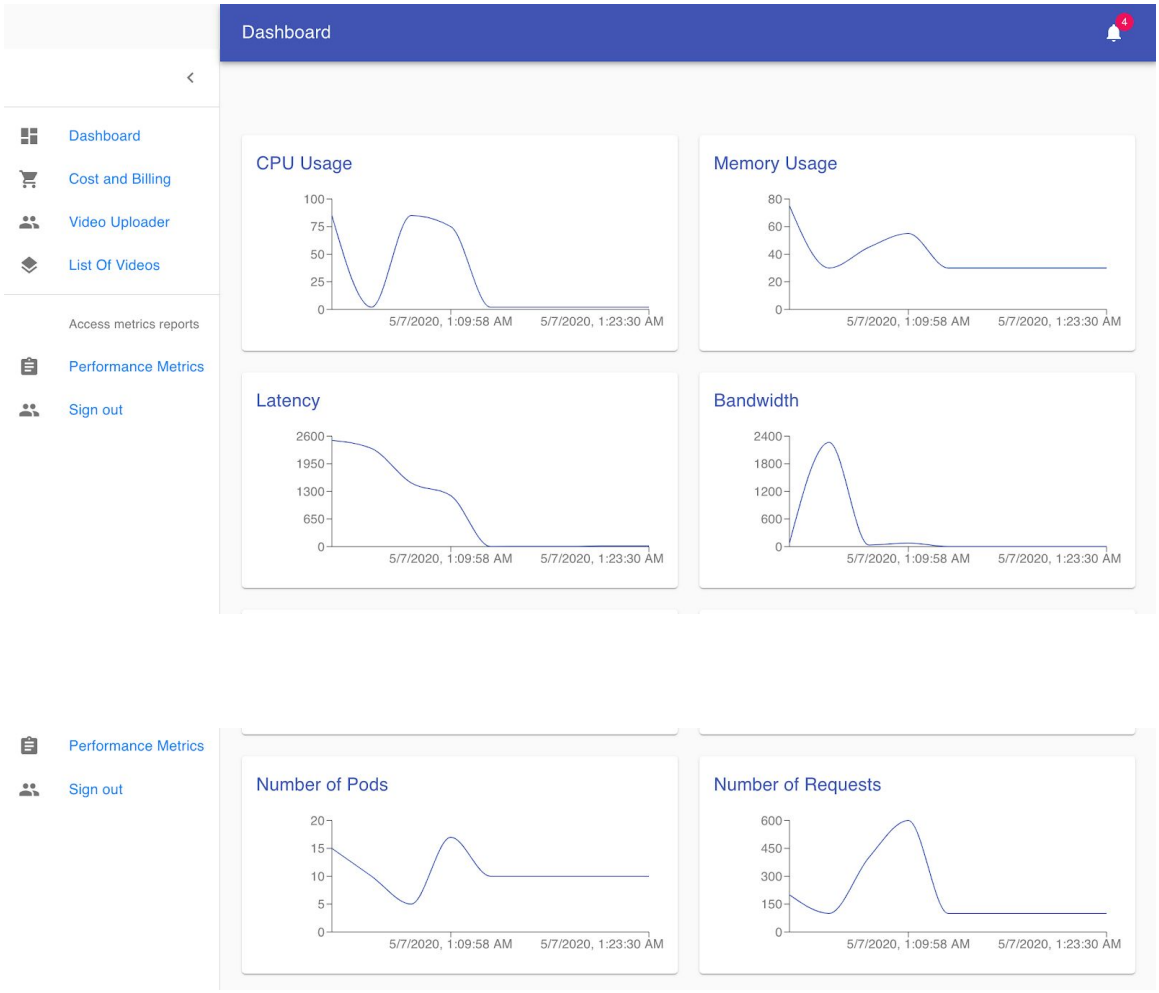


Figure: System and Video Metrics

2. Video Uploader - Once the client is registered with our Platform-as-a-Service (PaaS), they can upload the videos from their library which they want to be hosted on our application. When the videos are uploaded, the client will obtain a streaming url which can be integrated inside the stream tag of their SaaS.

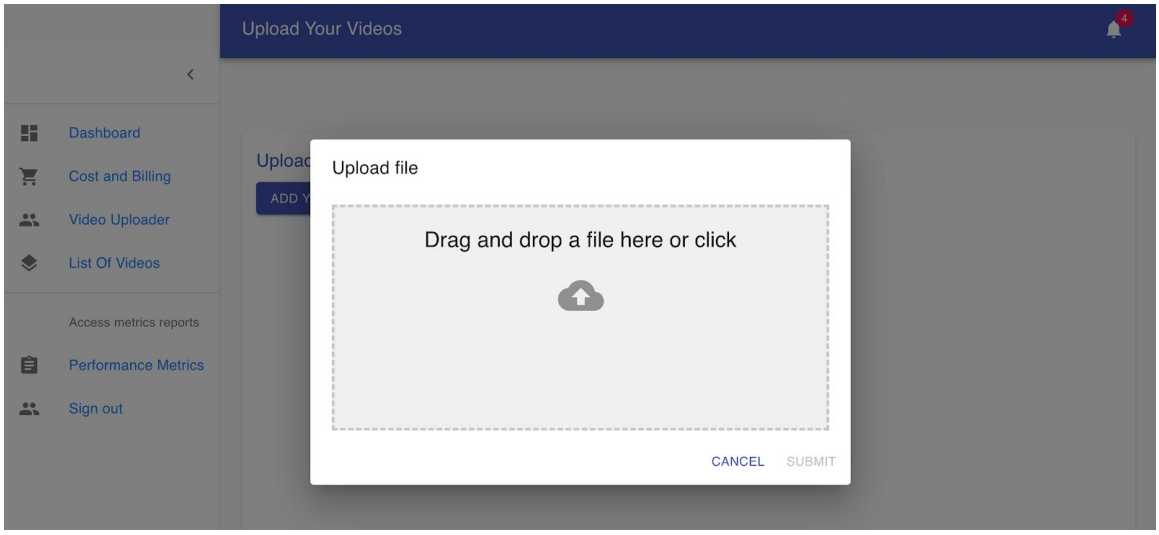


Figure: Video Upload

3.Videos List - This section provides the client with metadata of the client's videos that are hosted on our service. Information like access code, streaming url etc.

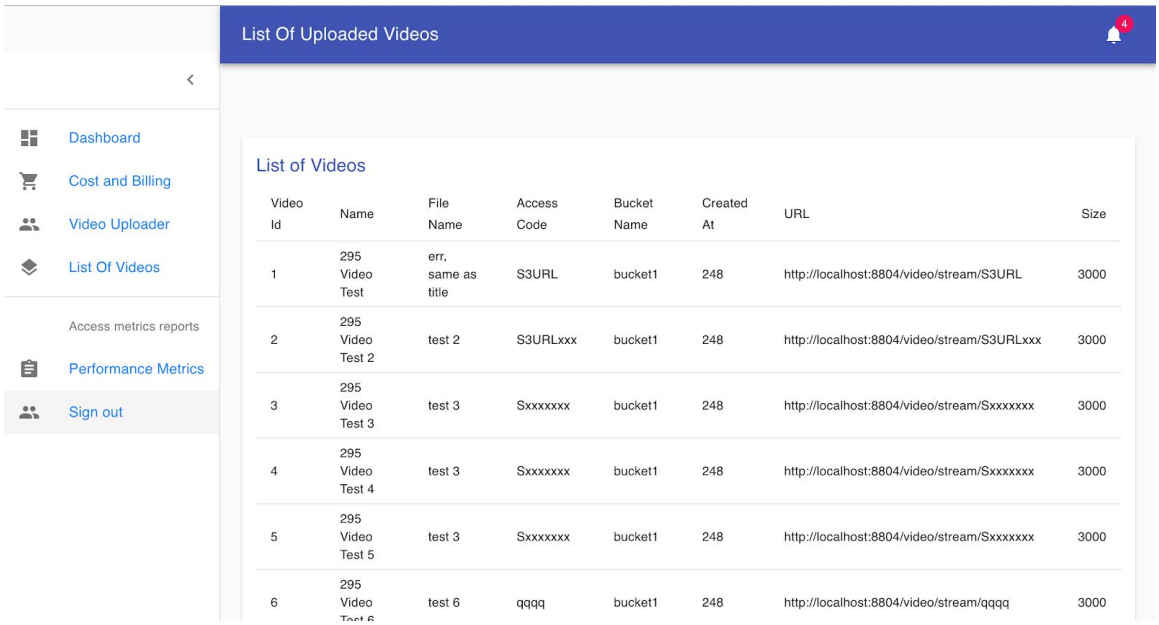


Figure: List of Videos uploaded by the client

4. Cost and Billing - The cost and Billing section enables the user to choose a subscription based plan for using the energy-aware PaaS. The different plans provide the clients with the flexibility to pick and choose a billing plan that suits the requirement of their Software-as-a-Service(SaaS) application.

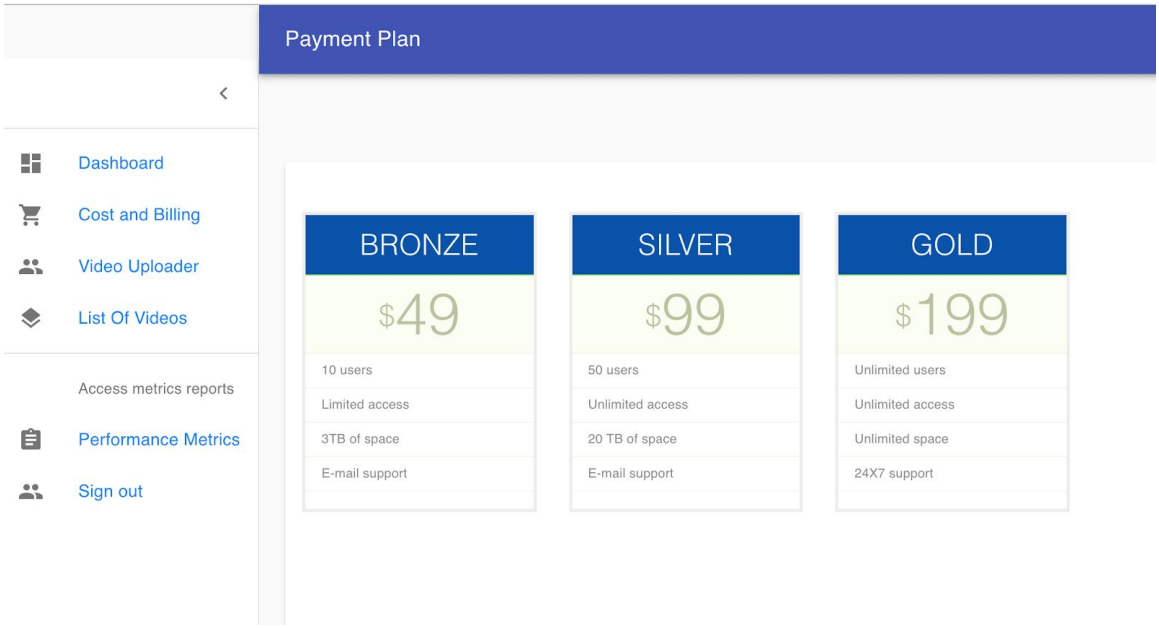
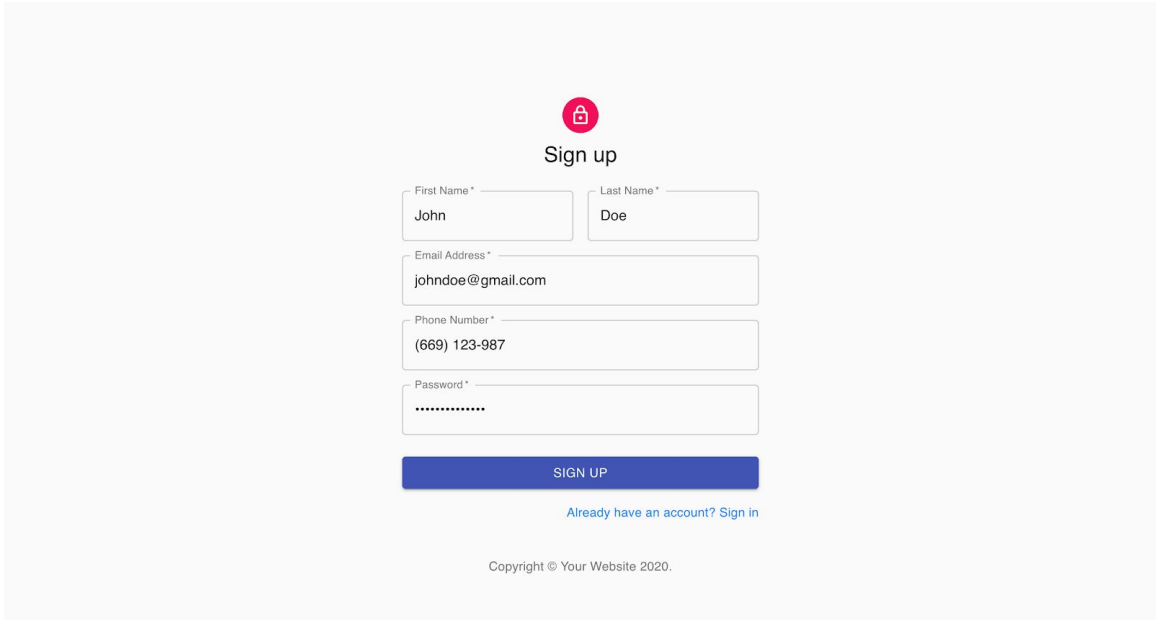


Figure: Subscription fees for the PaaS

5. User Registration: The client can register with our PaaS and on successful account creation, they would be able to host their videos on the platform



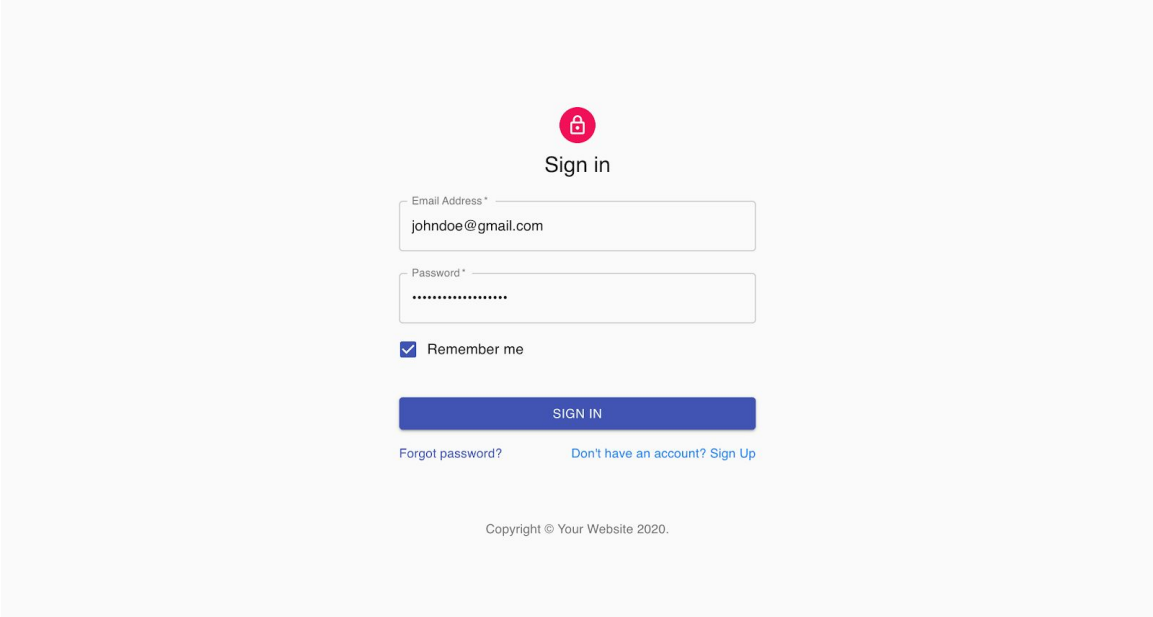
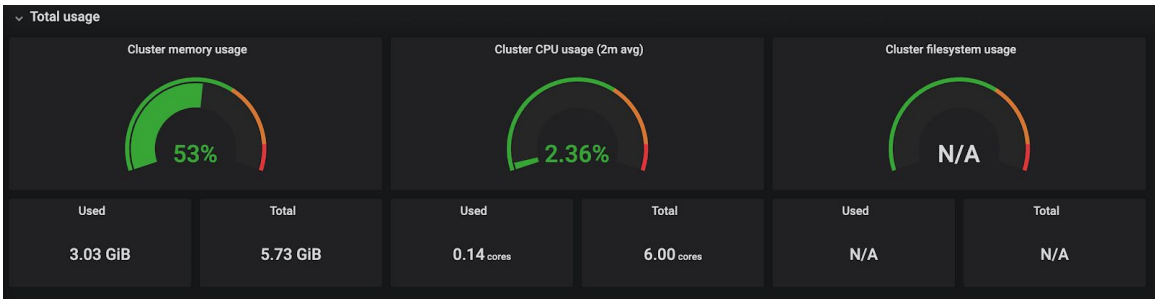


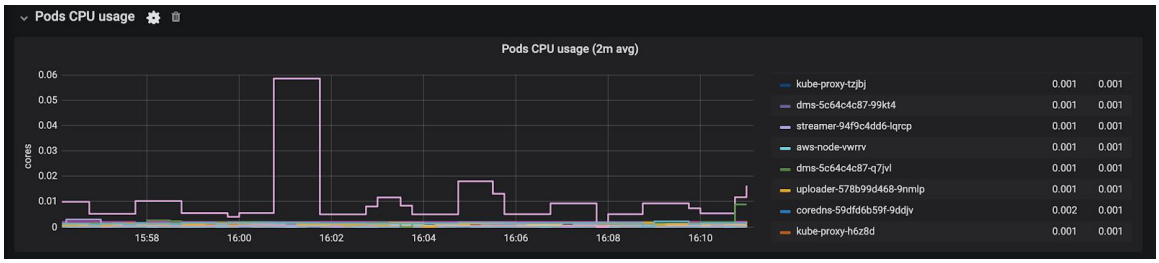
Figure: Client Registration and Login

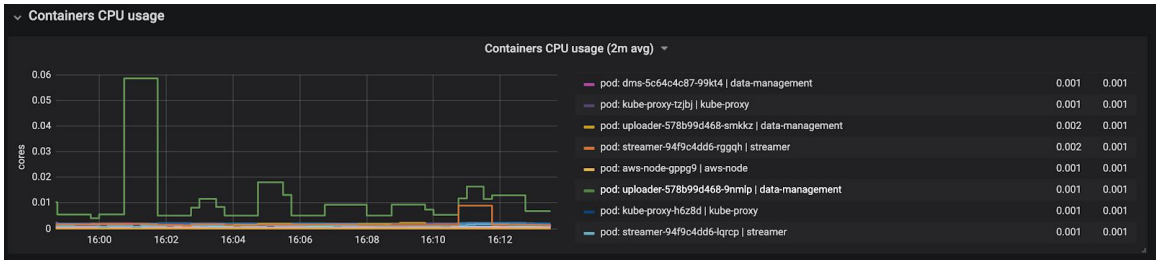
6.Grafana Dashboard - The client can actively monitor different system and performance metrics such as CPU utilization, memory, the health of the pod, network latency, count of restarts of the service, etc. for clear understanding of the current status of the application. Prometheus is used to continuously scrape the required data from different microservices and push it on to the Prometheus server which is later integrated with Grafana, a visualization tool to display different resource usage dashboards.

Total Usage of resources across all services :



Pod and Container CPU usage:





Query:

```
sum(rate(container_cpu_usage_seconds_total{namespace="microservice-name"}[5m]))/sum(machine_cpu_cores)*100
```

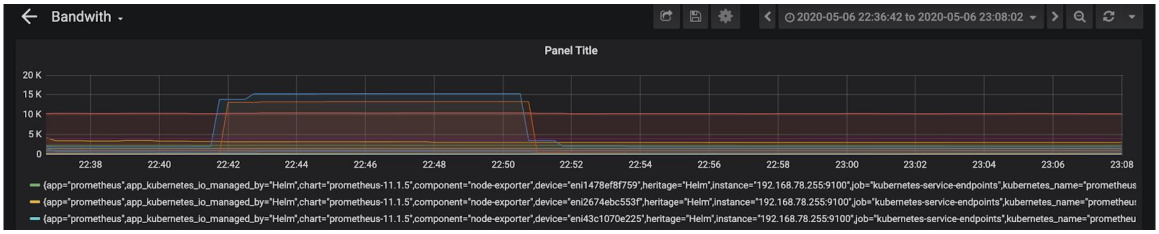
Pod and Container memory usage:

Memory Usage per microservice:

Query:

```
sum(container_memory_working_set_bytes{namespace="<microservice-name>"})/sum(machine_memory_bytes)*100
```

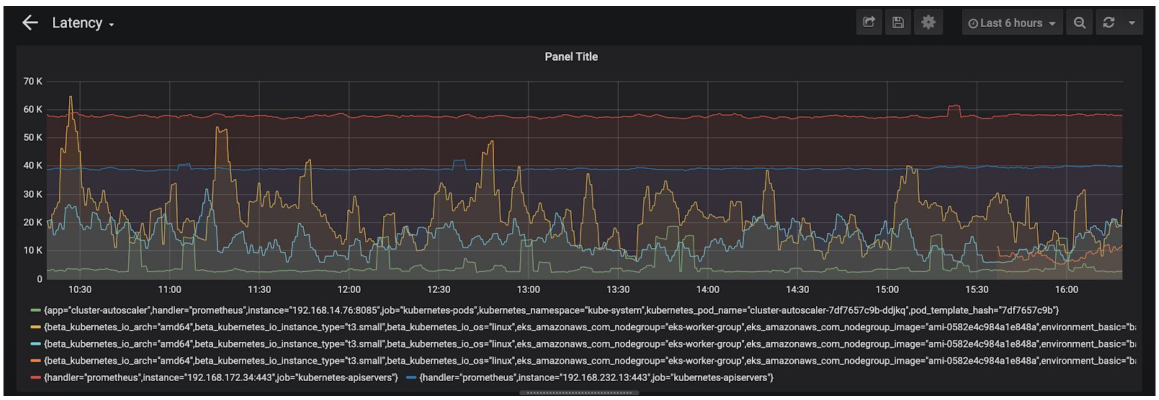
Cluster Bandwidth:



Query:

```
rate(node_network_transmit_bytes_total[5m])
```

Cluster Latency :



Query:

```
rate(http_request_duration_microseconds_sum[5m])/rate(http_reque
st_duration_microseconds_count[5m])
```

Chapter 7. Analysis and Result

7.1 Scalability

7.1.1 Network Traffic

Cluster-autoscaler, Horizontal-pod-autoscaler and Vertical-pod-autoscaler have been enabled for each of the microservice deployments. These default auto-scalers help in automatically adjusting to the size of the EKS cluster or increasing the deployment pods (horizontal scaling) or increasing the resources for the pods(vertical scaling). Enabling these features helped in adapting to the network traffic patterns that sometimes change abruptly.

Initial State :

```
[ec2-user@ip-172-31-47-5 VideoUploader]$ kubectl get hpa --namespace="videouploader"
NAME       REFERENCE              TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
uploader   Deployment/uploader   1%/2%     2         10        2          16s
[ec2-user@ip-172-31-47-5 VideoUploader]$ kubectl get hpa --namespace="videouploader"
NAME       REFERENCE              TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
uploader   Deployment/uploader   1%/2%     2         10        2          18s
[ec2-user@ip-172-31-47-5 VideoUploader]$ kubectl get hpa --namespace="videouploader"
NAME       REFERENCE              TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
uploader   Deployment/uploader   1%/2%     2         10        2          19s
[ec2-user@ip-172-31-47-5 VideoUploader]$ kubectl get hpa --namespace="videouploader"
NAME       REFERENCE              TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
uploader   Deployment/uploader   1%/2%     2         10        2          20s
```

Bulk load:(Heavy network traffic)

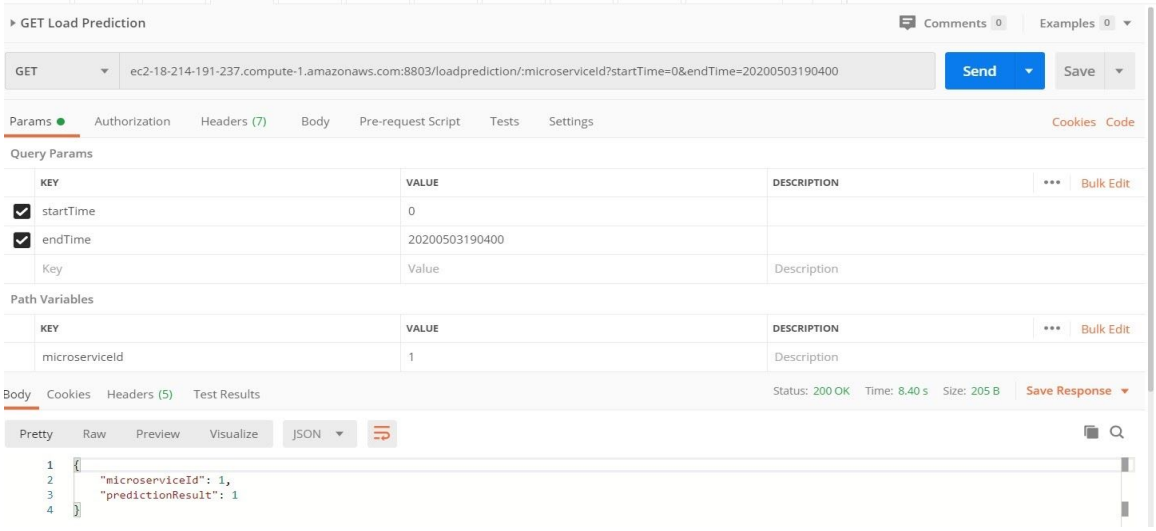

```
[ec2-user@ip-172-31-47-5 VideoUploader]$ kubectl get hpa --namespace="videouploader"
NAME       REFERENCE              TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
uploader   Deployment/uploader   1%/2%     2         10        2          19s
[ec2-user@ip-172-31-47-5 VideoUploader]$ kubectl get hpa --namespace="videouploader"
NAME       REFERENCE              TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
uploader   Deployment/uploader   1%/2%     2         10        2          20s
[ec2-user@ip-172-31-47-5 VideoUploader]$ kubectl get hpa --namespace="videouploader"
NAME       REFERENCE              TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
uploader   Deployment/uploader   6%/2%     2         10        6          113s
```

Pods automatically scale up from 2 to 6 on the basis of horizontal pod auto-scaler config

7.1.2 Load Prediction

Load Prediction module predicts the actions to be performed over an existing Infrastructure: “Scale-up”, “Scale-down”, and “No-action”. This prediction is based on metrics such as CPU usage, memory usage, number of allocated pods, network latency, and bandwidth.
Since the predictions are microservice based, a continuous prediction of “**Scale-up**” enumerated as “**1**” until convergence triggers an auto-scaling module which alters the existing microservice configuration. The same process is repeated for a continuous prediction of **“Scale-down”** resulting in management of the infrastructure resources with minimum or no human-intervention.

Output of the load prediction module for the above load-testing scenario:



7.2 Fault Tolerant

The different microservices across the platform are deployed on multiple pods on Kubernetes which makes the platform competent to cope up with a substantial spike in the network traffic enabling the platform to continue operating properly in case of a service failure. To allow the application to recover from the errors gracefully and avoid any single point of failures, we are maintaining 5 replicas in the cluster for each microservices which are being served by the load balancer. This enables the platform to not completely shut down in case any microservices pods go down.

7.3 High Availability

The platform is inherently reliable and stable as each microservice is deployed on multiple worker nodes within a namespace. As per the fluctuating user requests, we are spinning up or down the worker nodes to better serve the requests while not compromising on the resource utilization. Each service is deployed across multiple availability zones and subnets, making the system resilient to availability zone failures.

7.4 Load Prediction accuracy and adaptability

Our load prediction module uses a deep reinforcement learning algorithm to predict the best plan of action under different loads(states). As our module keeps predicting the best action for our application, it automatically updates our deep Q-learning model with the result of the predicted action. Our model might start as a generic load prediction model for a variety of applications. However, as the model keeps predicting and learning from the application's metrics, it adapts to become more suited for the application it's predicting. As the application keeps operating, our model will increasingly improve its prediction accuracy.

Chapter 8. Summary, Conclusions, and Recommendations

8.1 Conclusion

In this report, we have proposed an energy aware platform as a service for video streaming by integrating reinforcement learning with container management technology. As can be seen from the experiment results, our proposed solution efficiently manages the load among the pods within the cluster enabling users to better manage the resources. This helps in cutting down the costs of unused infrastructure in place and managing them in a cost-efficient manner.

8.2 Recommendation for future Research

For future study of reinforcement learning algorithms for load prediction, there are a couple recommendations that our study suggests. Our study uses three kubernetes metrics (CPU, Memory, and Pod numbers) and two network metrics (latency and bandwidth) to predict future load. Therefore, it will be desirable to see how the performance of the load prediction module changes as the input features are varied.

The second recommendation is dataset availability. With our limited resources and time, our system is only tested with a similar type of dataset (our video streaming platform). It will be of interest if our load prediction algorithm can be applied to another dataset of other types of application as well.

References

[1] Vijay K. Adhikari, Yang Guo, Fang Hao, Zhi-Li Zhang, Volker Hilt, Matteo Varvello, and Moritz Steiner, “Measurement Study of Netflix, Hulu, and a Tale of Three CDNs”, IEEE/ACM Transactions on Networking, vol. 23, no. 6, December 2015

[2] The Global Internet Phenomena Report October 2018, https://www.sandvine.com/hubfs/downloads/phenomena/2018-phenomena-report.pdf

[3] B. Kitchenham, Procedures for Performing Systematic Reviews, Joint Technical Report, Keele University Technical Report (TR/SE-0401) and NICTA Technical Report (0400011T.1) July 2004, 2004.

[4] Mayukh Nair, “How Netflix works: the (hugely simplified) complex stuff that happens every time you hit Play”, article posted on the medium platform in the https://medium.com/refraction-tech-everything/how-netflix-works-the-hugely-simplified-complex-stuff-that-happens-every-time-you-hit-play-3a40c9be254b

[5] Yunyun Jiang, Xiaosong Ma, and Wenguang Chen, “Cost-Effective Resource Configuration for Cloud Video Streaming Services”, 2015 IEEE 21st International Conference on Parallel and Distributed Systems

[6] System design architecture by Narendra L on Medium platform**, “**NETFLIX system design”, https://medium.com/@narengowda/netflix-system-design-dbec30fede8d

[7] Suoheng Li, Zuqing Zhu, Weiping Li, Houqiang Li, “Efficient and Scalable Cloud-Assisted SVC Video Streaming Through Mesh Networks”, International Conference on Computing, Networking and Communications Invited Position Paper Track

[8] Jer´ome Mendes de Figueiredo, Av. Prof.An´ıbal Cavaco Silva, “CASHED: Cloud-Assisted Adaptive and Scalable Video Streaming for Heterogeneous End-User Devices”

[9] Víctor Medel, Rafael Tolosana-Calasanza, José Ángel Bañaresa, Unai Arronategui, Omer F. Ranab “Characterizing resource management performance in Kubernetes”, Computers & Electrical Engineering, Volume 68, May 2018

[10] V.K Adhikari, Y. Guo, F. Hao, M. Varvello, V. Hi, M.Steiner, Z. Zhang “Unreeling netflix: Understanding and improving multi-CDN movie delivery” in the IEEE Computer Society.