Ahmedabad University

VIDEO STREAMING APPLICATION USING KVM VIRTUALISATION

CSE540 Cloud Computing

Submitted to:- Prof. Sanjay Chaudhary

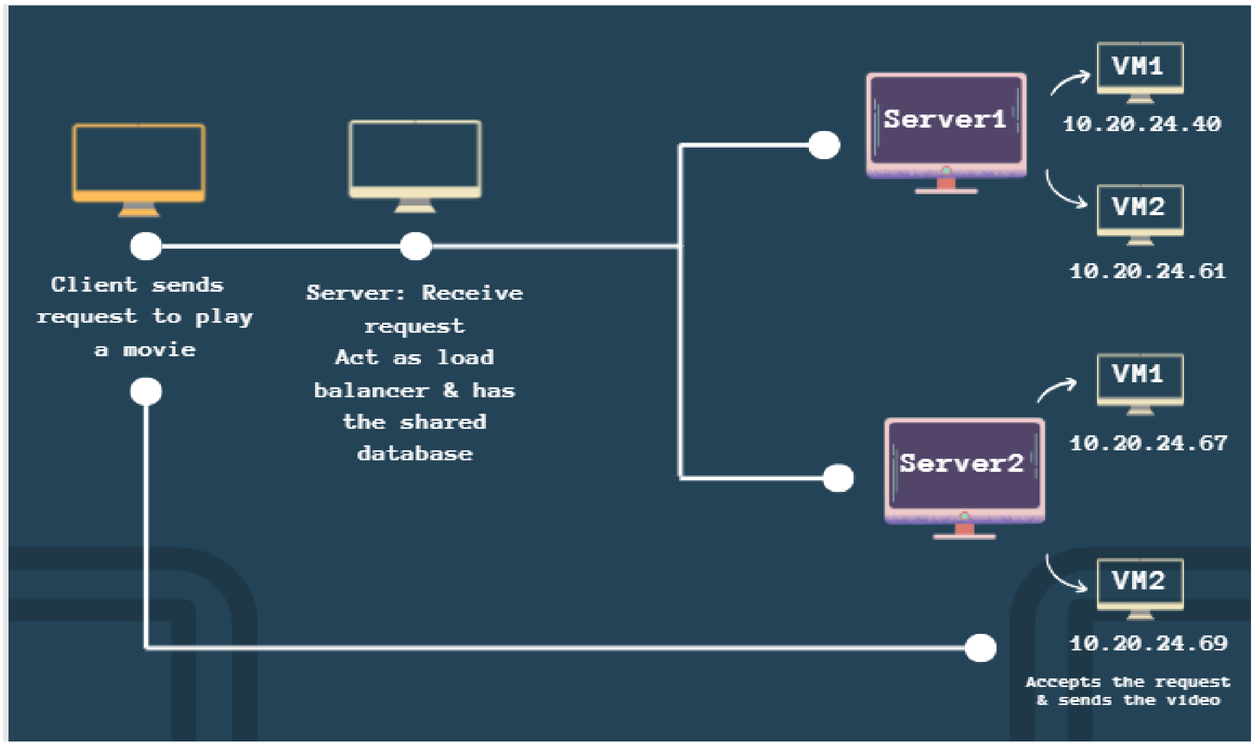
Group Members	Enrollment No.	Email ID
Pratham Mehta	AU2040203	pratham.m1@ahduni.edu.in
Aditi Vasa	AU2040122	aditi.v1@ahduni.edu.in
Purvansh Barodia	AU2040188	purvansh.b@ahduni.edu.in
Vandan Shah	AU2040196	vandan.s@ahduni.edu.in

Project Statement:

The demand for video streaming services is ever-increasing. Traditional infrastructure can be costly to set up and maintain. Leveraging cloud services, provides a cost-effective approach, allowing for efficient resource utilization and pay-as-you-go models. Users expect seamless and high-quality video streaming experiences. By incorporating load balancing, shared storage with GlusterFS, and a responsive user interface, the project aims to enhance the overall user experience. This project presents a cloud-based video streaming application designed for optimal performance and scalability. Leveraging KVM virtualization, Flask backend, and GlusterFS shared storage, the system ensures efficient resource utilization and seamless streaming. Load balancing and redundancy measures enhance reliability, while an intuitive user interface facilitates a user-friendly experience.

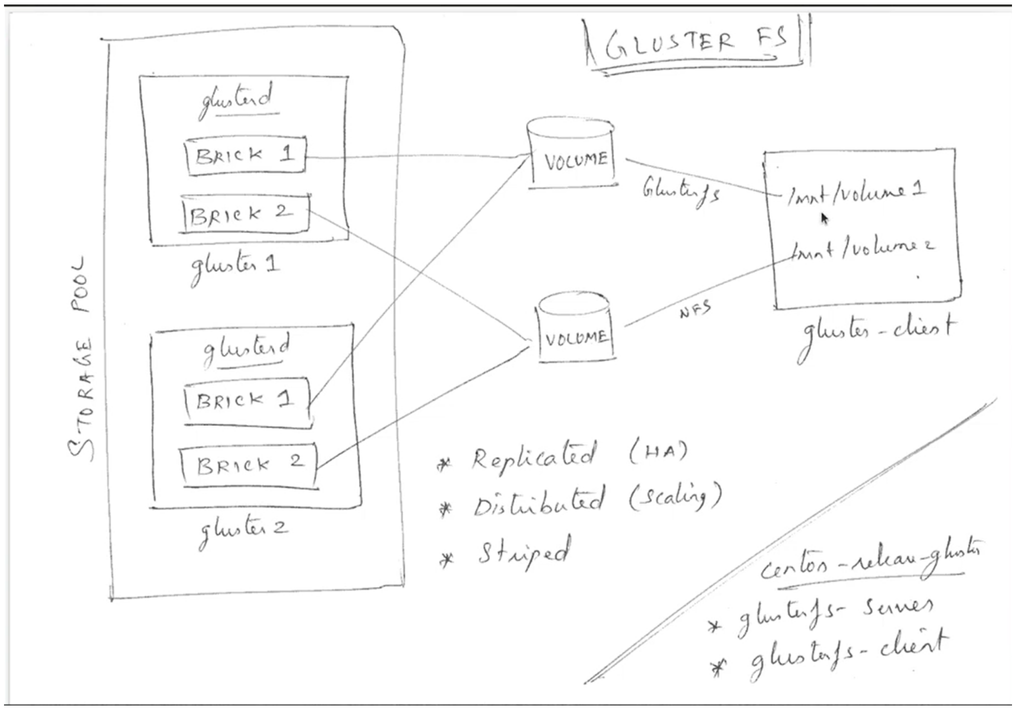
Project Design:

Shown below is the project architecture.



Client sends request to play a movie
Server: Receive request Act as load balancer & has the shared database
Server1
VM1
10.20.24.40
VM2
10.20.24.61
Server2
VM1
10.20.24.67
VM2
10.20.24.69
Accepts the request & sends the video

Gluster FS Architecture (Shared Database):



The architecture of our project setup is as shown in the figure above. Here, we have used virtualization on the server machines to efficiently serve the streaming service.

Virtual Machines:
We used virtualization and created 4 virtual machines on top of two servers. These VMs will host the streaming service and on request will stream the video files from the storage.

Database and Shared File:
A shared file system is created between the virtual machines hosting the service. This file system stores the video files and a shared database which contains the metadata about those video files.

Client Application:
As a part of the front end, we developed a web-based application in Python Flask, which will call the service APIs. This application is hosted on a separate server.

Implementation :

- Server Setup and Shared Database Creation:
    - Description: Initiated the project by configuring a main server, serving as the central hub. Established a shared database on this server using technologies like GlusterFS for consistent data access among the virtual machines (VMs).

- Frontend and Backend Development:
    - Description: Implemented the user interface using HTML, CSS, and JS to create a responsive and engaging frontend. The backend logic was crafted using the Flask framework, facilitating efficient handling of client requests and interactions with the shared database.

- Client-Main Server Interaction:
    - Description: When a client selects a movie, the request is sent to the main server. This server acts as a coordinator for movie streaming. It manages the communication flow between the client, the shared database, and the virtual machines.

- Virtual Machines and Movie Streaming:
    - Description: The main server communicates with two additional servers, each hosting two virtual machines (VMs). These VMs are in constant listening mode, awaiting instructions from the main server. Upon receiving a movie request, the first available VM accepts it and initiates the movie. The URL of the movie is then relayed back to the main server, which, in turn, facilitates the streaming of the selected movie for the client.

PSEUDOCODE:

Flask application interacts with an SQLite database of movies and communicates with multiple servers to play them. It has routes for rendering an index page with a list of movies, a movie details page, and an API endpoint for retrieving selected movie names and determining the next server(VM) in a round-robin fashion. When the client goes to this URL with the GET method, the server provides a JSON response containing all the details about all movies. The /movie route sends requests to different servers for movie playback based on conditions. Global variables track the selected movie and the current server index. The code utilizes Flask for web functionality and requests for server communication. Below is the pseudocode for the same.

```
1. Define Flask application
2. Initialize global variables: selected_movie, servers, current_server_index
3. Define function get_movies():
   - Connect to SQLite database 'movies.db'
   - Execute query to retrieve all movies from 'movies' table
   - Print each movie
   - Close database connection
   - Return list of movies

4. Define route '/' for rendering the index page:
   - Call get_movies to retrieve the movie list
   - Render index.html with movie list

5. Define route '/movie_page' for rendering the movie details page:
   - Render page.html

6. Define API endpoint '/get-movie-name/' for retrieving selected movie and server:
   - Retrieve selected_movie from request query parameters
   - Determine the next server in a round-robin fashion
   - Return JSON object with 'name' (selected_movie) and 'server' (current_server)

7. Define route '/movie' for playing movies:
   - Send requests to different servers based on conditions
   - Handle movie playback and server responses.
```

HARDWARE PLATFORM:

- Server:
    - Two PCs of Cloud Computing Lab
    - (Ubuntu 20.04)- Linux OS
- Virtual Machine:
    - Each server had two virtual machine
    - Each VM had 2GB RAM
- Client:
    - Single PC of Cloud Computing Lab
    - (Ubuntu 20.04)- Linux OS

SOFTWARE PLATFORM:

- Database:
    - SQLite for local database
    - GlusterFS for distributed and shared database storage
- Backend
    - Flask for the backend framework

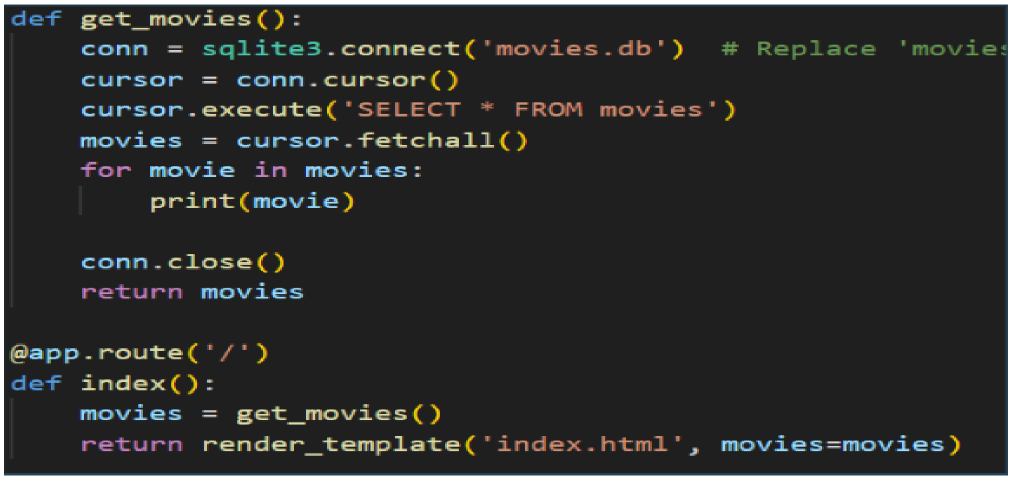
    - Python for server-side scripting
- Frontend
    - HTML, CSS, and JavaScript for building the user interface
- API
    - RESTful APIs for communication between frontend and backend
    - HTTP for client-server communication

LIST OF CLOUD SERVICE:

In the world of Software as a Service (SaaS), our Video Streaming Application lets users watch videos without worrying about the complicated tech stuff in the background. With SaaS, users don't have to deal with tricky things like managing servers or databases. They can easily access the video streaming service through web browsers or apps without needing to install any software on their devices. SaaS makes things easy by using a subscription model, which means the service provider takes care of all the techie tasks like keeping things up-to-date and running smoothly. Users can just focus on enjoying videos while the SaaS provider handles the behind-the-scenes work, making the video streaming experience friendly and straightforward.

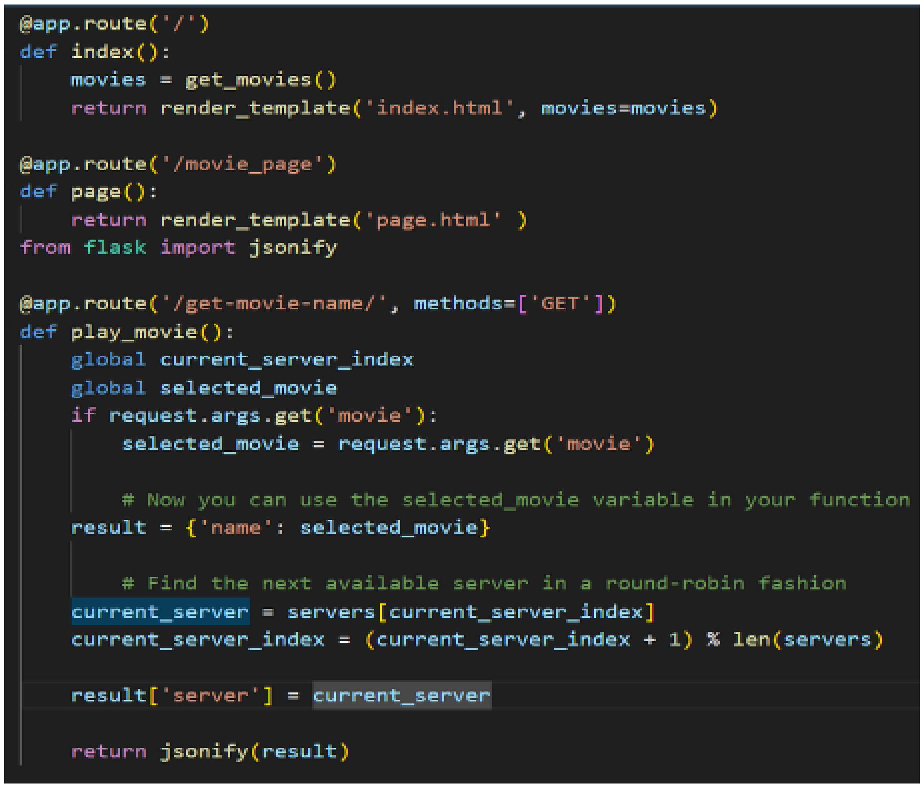
SAMPLE SOURCE CODES:

1. **Fetching all movies:**



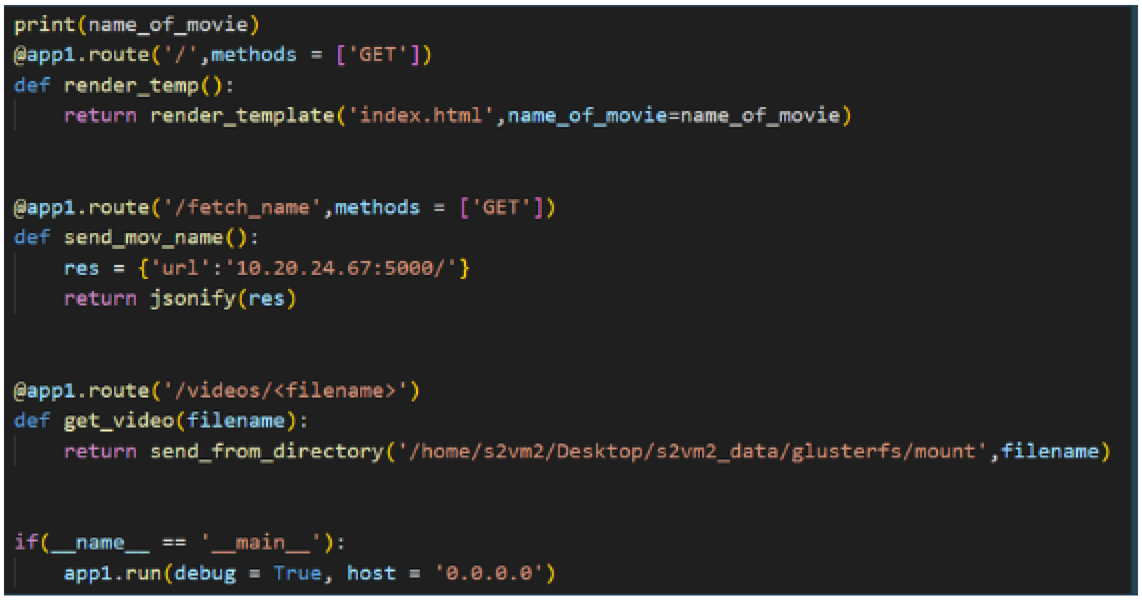
```python
def get_movies():
    conn = sqlite3.connect('movies.db')  # Replace 'movie
    cursor = conn.cursor()
    cursor.execute('SELECT * FROM movies')
    movies = cursor.fetchall()
    for movie in movies:
        print(movie)

    conn.close()
    return movies

@app.route('/')
def index():
    movies = get_movies()
    return render_template('index.html', movies=movies)
```

2. Getting the name of the movie:



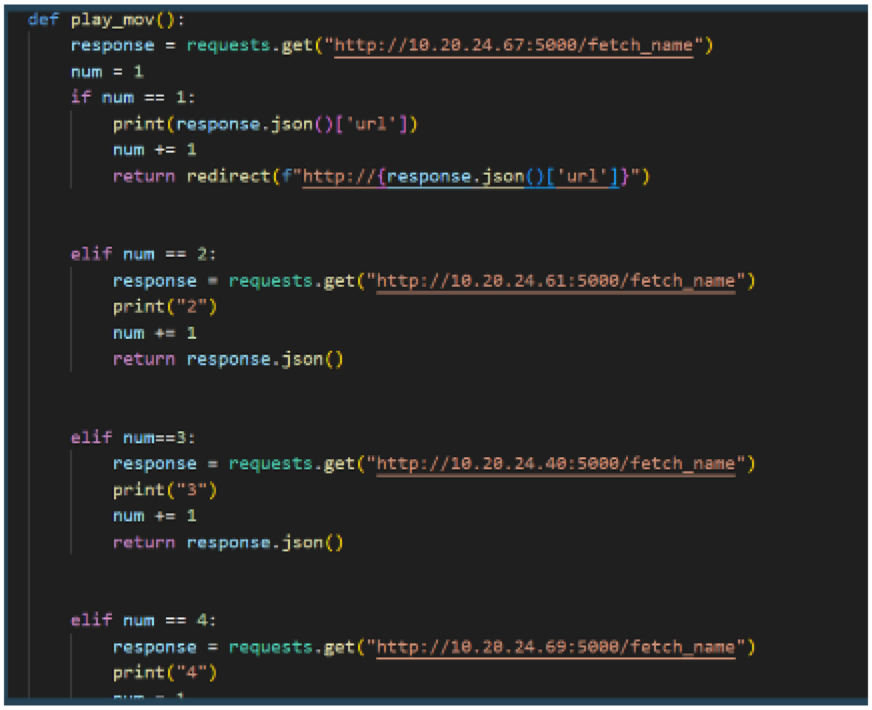
```python
@app.route('/')
def index():
    movies = get_movies()
    return render_template('index.html', movies=movies)

@app.route('/movie_page')
def page():
    return render_template('page.html' )
from flask import jsonify

@app.route('/get-movie-name/', methods=['GET'])
def play_movie():
    global current_server_index
    global selected_movie
    if request.args.get('movie'):
        selected_movie = request.args.get('movie')

        # Now you can use the selected_movie variable in your function
    result = {'name': selected_movie}

        # Find the next available server in a round-robin fashion
    current_server = servers[current_server_index]
    current_server_index = (current_server_index + 1) % len(servers)

    result['server'] = current_server

    return jsonify(result)
```

3. VM communication:



```python
print(name_of_movie)
@app1.route('/',methods = ['GET'])
def render_temp():
    return render_template('index.html',name_of_movie=name_of_movie)


@app1.route('/fetch_name',methods = ['GET'])
def send_mov_name():
    res = {'url':'10.20.24.67:5000/'}
    return jsonify(res)


@app1.route('/videos/<filename>')
def get_video(filename):
    return send_from_directory('/home/s2vm2/Desktop/s2vm2_data/glusterfs/mount',filename)


if(__name__ == '__main__'):
    app1.run(debug = True, host = '0.0.0.0')
```

4. **Fetching URL of available VM**



```
def play_mov():
    response = requests.get("http://10.20.24.67:5000/fetch_name")
    num = 1
    if num == 1:
        print(response.json()['url'])
        num += 1
        return redirect(f"http://{response.json()['url']}")


    elif num == 2:
        response = requests.get("http://10.20.24.61:5000/fetch_name")
        print("2")
        num += 1
        return response.json()


    elif num==3:
        response = requests.get("http://10.20.24.40:5000/fetch_name")
        print("3")
        num += 1
        return response.json()


    elif num == 4:
        response = requests.get("http://10.20.24.69:5000/fetch_name")
        print("4")
```

CONCLUSION:

In this project, we were able to install and configure KVM Virtualization and achieve full virtualization. 4 servers were set up (on virtual machines) on two machines in the lab so that multiple requests can be served concurrently. These servers were used for delivering video content in the web application. The movies were stored in shared storage between the servers which are served as per the request. The load balancer checks the load of each server and redirects user requests to the server with the minimum load. We also implemented a client-side web application for our streaming service in which the user can get the list of available movies, their details and watch any movie.

ANNEXURES:

1. How can you scale the project without introducing more VMs as that would mean higher cost?
   Scaling a project without introducing more virtual machines (VMs) can be achieved through various strategies that optimize resource utilization, improve performance, and enhance efficiency. Here are some approaches to consider:

   Caching:
   - Implement caching mechanisms to store frequently accessed data. This reduces the need to generate or fetch data repeatedly, improving response times and reducing server load.

Load Balancing:

- Implement load balancing techniques to distribute incoming traffic across multiple servers efficiently. This ensures that no single server bears the brunt of excessive load.

Horizontal Scaling:

- Instead of adding more VMs, consider scaling horizontally by adding more instances of your application across multiple servers. Load balancers can distribute incoming traffic across these instances.

Auto-Scaling:

- Implement auto-scaling solutions that dynamically adjust the number of instances based on demand. This can help ensure that we have the right amount of resources available at all times without manual intervention.

2. Explain the installation or use of XCP-ng in this project instead of KVM.
   Here is the installation documentation for XCP-ng. It requires creating a bootable USB. Even after changing boot order, the machine was running on the host OS.

   Installation guide : https://docs.xcp-ng.org/installation/install-xcp-ng/

3. If 4 users are watching 4 different movies, another user comes and wants to see the same movie 1st one is watching, then what does the system do? Will it again go to main memory to get the movie or will it store recently watched movie in some cache?

   The behavior of a system when a user wants to watch the same movie as another user depends on the architecture and design of the streaming system. Here are some systems that can be designed:

   Streaming from Main Memory or Storage:

   - In a typical scenario, when a user starts watching a movie, the content is streamed from the main storage (like a server or cloud storage).
   - If another user wants to watch the same movie, the system might check if the content is already in the cache or main memory to avoid fetching it again from the storage. However, if the content is not in the cache, the system may need to retrieve it from the main storage.

Caching Mechanisms:

- Some streaming systems implement caching mechanisms to store recently accessed content in a cache. If the movie being watched is popular or has been recently viewed by another user, there's a chance that it's already in the cache.
- If the movie is in the cache, the system can serve it directly from there, reducing the need to fetch it again from the main storage.

Dynamic Streaming and Adaptive Bitrate:

- Some streaming services use adaptive bitrate streaming, where the quality of the video adjusts based on the viewer's network conditions. In such cases, the system may fetch different segments of the video based on the viewer's network speed and device capabilities.

4. Providing a catalog service page for user to know which movies are available without using database?

   We can consider an alternative approach to provide a catalog service page to know which movies are availabe. One option is to use a static file or files that contain the information about the available movies. Another option is to create one or more json or yaml files that stores information about each movie, such as title,gener,release year, and any other relevant details. Catalog service can read and parse these files to dynamically generate catalog page.

REFERENCES:

1. Live Streaming - Media & Entertainment Solutions | Google Cloud. (n.d.). Retrieved from https://cloud.google.com/solutions/media-entertainment/use-cases/live-streaming/
2. A Practical Guide to Deploying Cloud-Based Video Services. (n.d.). Retrieved from https://pages.awscloud.com/GLOBAL_IND_practical_guide_whitepaper_2019.html
3. REST API: What It Is and How to Use One for Streaming: Wowza. Retrieved from https://www.wowza.com/blog/rest-api-streaming
4. Building a RESTful Web Service. (n.d.). Retrieved from https://spring.io/guides/gs/rest-service/
5. Dosoukey, M., & Dosoukey, M. (2022, October 21). *How to create virtual machines in Linux using KVM (Kernel-based Virtual Machine) - Part 1*. https://www.tecmint.com/install-and-configure-kvm-in-linux/
6. Just me and Opensource. (2020, March 4). *[GlusterFS 3] Creating replicated volumes in Gluster FS* [Video]. YouTube. https://www.youtube.com/watch?v=DfX1AAuWPKQ