

Name: Ranasinghe Priyantha

Student Reference Number: 10899343

Module Code: PUSL3130

Module Name: Advance Computing and Network Infrastructures

Coursework Title: PUSL3130 – Coursework – Group 1

Deadline Date: 5<sup>th</sup> May 2025

Member of staff responsible for coursework: Mr. Bogdan Ghita

Programme: BSc (Hons) Computer Security

Please note that University Academic Regulations are available under Rules and Regulations on the University website [www.plymouth.ac.uk/studenthandbook](http://www.plymouth.ac.uk/studenthandbook).

Group work: please list all names of all participants formally associated with this work and state whether the work was undertaken alone or as part of a team. Please note you may be required to identify individual responsibility for component parts.

10817967 – Chakrawarthy Fernando

10817966 – Gusthingna De Silva

10899307 – Hewawasam Hansana

10899343 - Ranasinghe Priyantha

***We confirm that we have read and understood the Plymouth University regulations relating to Assessment Offences and that we are aware of the possible penalties for any breach of these regulations. We confirm that this is the independent work of the group.***

Signed on behalf of the group: Ranasinghe Priyantha

Individual assignment: ***I confirm that I have read and understood the Plymouth University regulations relating to Assessment Offences and that I am aware of the possible penalties for any breach of these regulations. I confirm that this is my own independent work.***

Signed:

Use of translation software: failure to declare that translation software or a similar writing aid has been used will be treated as an assessment offence.

I \*have used/not used translation software.

If used, please state name of software.....

Overall mark \_\_\_\_\_% Assessors Initials \_\_\_\_\_ Date \_\_\_\_\_



# **PUSL3130 Advance Computing and Network infrastructures**

## **Coursework**

<b>INTRODUCTION .....</b>	<b>4</b>
<b>PART 1: AWS CLOUD AND PERFORMANCE EVALUATION.....</b>	<b>6</b>
INTRODUCTION .....	6
2. EXPERIMENT/TESTBED SETUP (ALIGNS WITH 40% RUBRIC).....	6
3. AWS SCRIPT .....	10
4. PERFORMANCE EVALUATION AND ANALYSIS .....	13
<b>PART 2: ADAPTIVE VIDEO STREAMING OVER SDN.....</b>	<b>16</b>
1. EXPERIMENT/TESTBED SETUP .....	16
2. EVALUATION AND ANALYSIS .....	19
3. DASH EXPERIMENTS .....	19
4. PYTHON SCRIPT .....	21
<b>CONCLUSION.....</b>	<b>23</b>

## Introduction

In the contemporary technological landscape, the rising demands for seamless digital services have mandated organizations to innovate scalable, efficient, and resilient web architectures. One such organization, BogdanLTD a dynamic mobile phone retailer is encountering performance bottlenecks due to an increase in customer traffic on its digital platforms. This evolving challenge necessitates the transition from traditional web hosting solutions to a cloud native, microservices driven infrastructure. The purpose of this project is to design, implement, automate, and evaluate a containerized cloud deployment using Amazon Web Services (AWS), Docker, and Software Defined Networking (SDN) to resolve these limitations and optimize service delivery for the company.

The project is divided into two major components: **Part 1**, focusing on a Docker based web application deployment using AWS for load balancing and performance enhancement, and **Part 2**, which centers on adaptive video streaming within a Software Defined Network environment. Each part involves constructing isolated yet realistic experimental testbeds, executing automation through scripting, and critically analyzing results derived from benchmarking tools such as ApacheBench and real time data collected from SDN controllers. Together, these efforts validate the applicability, reliability, and benefits of the proposed solution under diverse conditions.

The **first segment** of the project is anchored in containerization principles, emphasizing modularity, portability, and automation. Here, the application infrastructure comprises a central Nginx load balancer, two backend Nginx web servers, two PHP application servers, and a MySQL database all operating in separate Docker containers. This architectural layout adheres to the “one container per service” paradigm to promote fault isolation, horizontal scalability, and simplified management. Dockerfiles are used to define container behavior, while Docker Compose streamlines orchestration and network configuration. All components interact over a dedicated bridge network, facilitating secure, container to container communication.

Automation is a core objective of this implementation. A suite of shell scripts was developed to handle the complete lifecycle of the infrastructure from package installation, dependency resolution, repository cloning, and Docker environment setup to benchmarking using ApacheBench. These scripts eliminate manual intervention, reduce human error, and ensure repeatability across deployments, thereby fulfilling DevOps principles. Once the architecture is deployed, the load-balanced system is tested against direct server access using ApacheBench with varying levels of concurrent HTTP requests. The resulting metrics requests per second, average time per request, and system stability are recorded and compared to assess the performance uplift delivered by the load balancer.

The **second segment** transitions into the realm of Software Defined Networking (SDN), where the primary objective is to explore how intelligent network management can improve the quality of video streaming. Using Mininet, OpenDaylight, Apache, Firefox, and DASH.js, the project simulates real world streaming conditions and evaluates how packet loss and bandwidth affect video playback. The Mininet environment is programmed to host a virtualized topology consisting of a video server (h1) and a video client (h2), interconnected via a programmable OpenFlow switch managed by the OpenDaylight controller. Apache hosts DASH-compatible content, while DASH.js on Firefox provides adaptive playback at the client end.

A significant feature of this section is the use of automation and scripting for experiment control and data collection. The network environment is configured with precise parameters such as bandwidth limitations, latency, and packet loss rates. Python scripts interface with the OpenDaylight REST API to capture flow statistics, packet counts, and network states at regular intervals. Concurrently, the DASH.js player logs playback metrics like bitrate fluctuations, initial buffering times, and frame drops. This dual layer data acquisition provides a holistic view of the interplay between network conditions and end-user experience.

Diverse conditions, such as 0%, 2%, and 5% packet loss, are simulated to observe changes in video streaming quality. Results indicate that under optimal conditions, the system maintains a high average bitrate with minimal buffering. As packet loss increases, quality deteriorates, evidenced by frequent bitrate downgrades, prolonged buffering, and decreased playback stability. These outcomes confirm the necessity of intelligent, adaptive streaming strategies especially in networks where congestion and loss are prevalent.

Together, both segments offer a dual-perspective analysis one focusing on backend infrastructure efficiency for web applications, and the other on frontend performance and QoE (Quality of Experience) for end users consuming video content. By integrating Docker containerization with SDN capabilities, the project illustrates a powerful paradigm for future proofing application delivery systems. The AWS cloud framework empowers horizontal scalability, rapid deployment, and operational consistency. Meanwhile, SDN facilitates fine grained control over network traffic, enabling dynamic adaptations that can significantly boost media streaming outcomes.

In terms of feasibility and industry relevance, the solutions presented are well aligned with the modern demands of digital enterprises. BogdanLTD, or any similar organization experiencing scaling difficulties, would greatly benefit from the proposed architecture. With containerization simplifying deployments and maintenance, and SDN enabling performance optimization at the network layer, this hybrid approach ensures a highly available, secure, and responsive application environment. Moreover, the use of open source tools and cloud native methodologies keeps the implementation cost effective and sustainable for long term usage.

Overall, this project goes beyond theoretical constructs to demonstrate applied engineering practice. It combines cloud computing, microservices, and programmable networking to deliver a comprehensive solution tailored for scalability, performance, and user satisfaction. The next phase involves critical evaluation and discussion of the results gathered from both experiment setups, leading to practical insights and future recommendations. This systematic exploration not only advances technical knowledge but also contributes valuable methodologies for academic research and real world digital transformation strategies.

# Part 1: AWS Cloud and Performance Evaluation

## Introduction

In today's digital economy, businesses are increasingly reliant on scalable and resilient web infrastructures to support their growth and ensure seamless user experiences. BogdanLTD, a rapidly expanding mobile phone retailer, is currently experiencing performance limitations with its existing web server setup due to a growing online customer base. This part of the project aims to address the scalability and performance challenges by implementing a containerized, cloud-based architecture using Amazon Web Services (AWS) and Docker technologies.

The solution involves deploying an Nginx-based load balancer in front of a cluster of backend web servers, PHP servers, and a database server. A microservices approach is adopted, where each service runs in its own Docker container, ensuring isolation, scalability, and ease of management. To facilitate consistent and reproducible deployments, Docker Compose and Dockerfiles are used to automate the configuration and orchestration of the environment.

A performance evaluation is conducted using ApacheBench (ab) to compare system behavior with and without load balancing, under various simulated load conditions. This investigation not only demonstrates the practical benefits of load balancing in distributed web environments but also highlights the role of infrastructure automation in modern DevOps practices.

The following sections detail the architecture, implementation process, automation scripts, and performance analysis to validate the effectiveness of the proposed solution.

## 2. Experiment/Testbed Setup (Aligns with 40% rubric)

### Environment Setup

The testbed was constructed using a containerized microservices architecture deployed within a virtualized environment. A host Ubuntu 18.04 server virtual machine was provisioned, and Docker along with Docker Compose were installed to orchestrate the multi-container setup efficiently.

### Architecture Diagram

The system consists of the following containerized components:

One Nginx load balancer container

Two Nginx backend web servers

Two PHP application servers

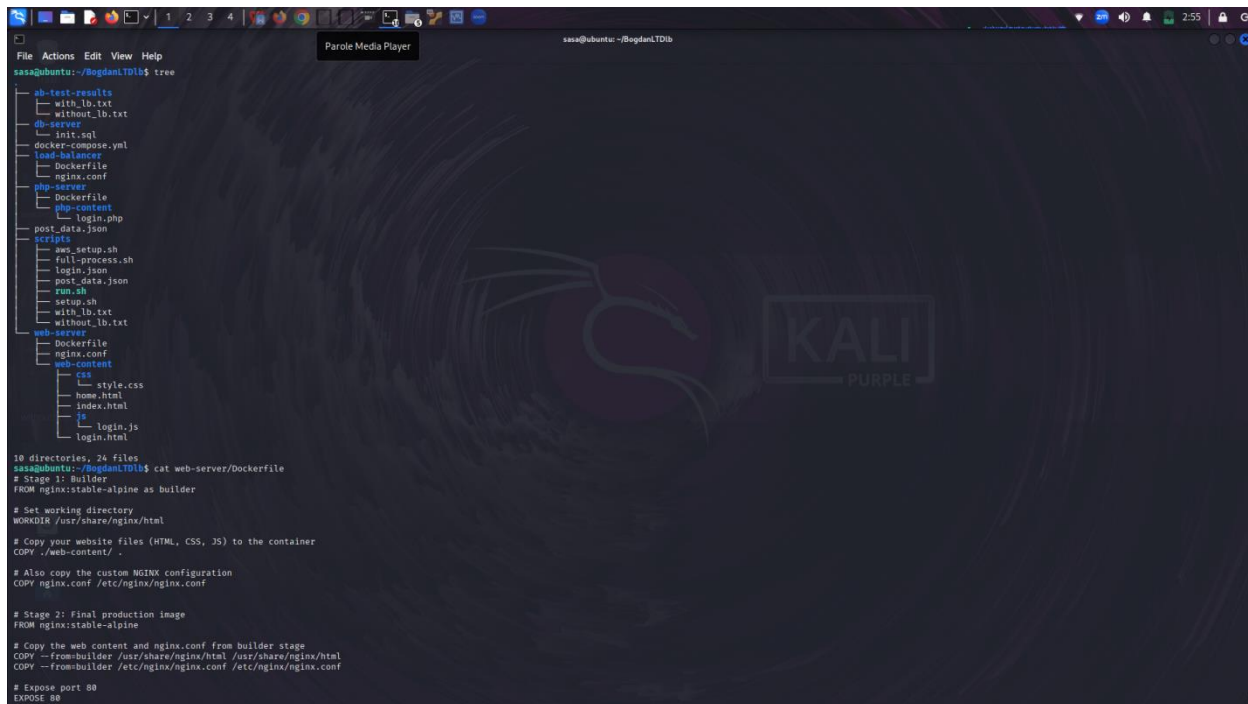
One MySQL database container

This layered structure enables separation of concerns, fault tolerance, and scalability. The architectural flow is illustrated below:

### Container Setup Approach

The deployment strictly follows the "one container, one service" principle. Each container handles a specific responsibility—routing, frontend rendering, application logic, or data persistence.

## Dockerfile Usage



The screenshot shows a terminal window with a file tree on the left and a Dockerfile on the right. The file tree is rooted at `sasa@ubuntu: ~/BogdanTD1b` and includes directories like `db-test-results`, `db-server`, `load-balancer`, `php-server`, `scripts`, and `web-server`. The Dockerfile content is as follows:

```
10 directories, 24 files
sasa@ubuntu:~/BogdanTD1b$ cat web-server/Dockerfile
# Stage 1: Builder
FROM nginx:stable-alpine as builder

# Set working directory
WORKDIR /usr/share/nginx/html

# Copy your website files (HTML, CSS, JS) to the container
COPY ./web-content/ .

# Also copy the custom NGINX configuration
COPY nginx.conf /etc/nginx/nginx.conf

# Stage 2: Final production image
FROM nginx:stable-alpine

# Copy the web content and nginx.conf from builder stage
COPY --from=builder /usr/share/nginx/html /usr/share/nginx/html
COPY --from=builder /etc/nginx/nginx.conf /etc/nginx/nginx.conf

# Expose port 80
EXPOSE 80
```

Custom Dockerfiles were crafted for:

**Load Balancer (Nginx):** to define advanced proxying rules and compress static resources.

**Web Servers:** to serve static HTML/CSS/JS with custom Nginx configurations.

**PHP Servers:** using the official PHP image with necessary extensions for MySQL communication.

## Docker Compose Configuration

```
File Actions Edit View Help
services:
  load-balancer:
    build: ./load-balancer
    ports:
      - "80:80"
    depends_on:
      - web-server-1
      - web-server-2
      - php-server-1
      - php-server-2
    networks:
      - coursework-net
    container_name: load-balancer

  web-server-1:
    build: ./web-server
    networks:
      - coursework-net
    user: root
    container_name: web-server-1

  web-server-2:
    build: ./web-server
    networks:
      - coursework-net
    user: root
    container_name: web-server-2

  php-server-1:
    build: ./php-server
    networks:
      - coursework-net
    container_name: php-server-1

  php-server-2:
    build: ./php-server
    networks:
      - coursework-net
    container_name: php-server-2

  db-server:
    image: mysql:8.0
    container_name: mysql-db
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: rootpassword
      MYSQL_DATABASE: courseworkdb
      MYSQL_USER: courseworkuser
      MYSQL_PASSWORD: courseworkpass
    ports:
      - "3306:3306"
    volumes:
      - db_data:/var/lib/mysql
      - ./db-server/init.sql:/docker-entrypoint-initdb.d/init.sql
    networks:
      - coursework-net

networks:
  coursework-net:
    driver: bridge
```

The entire stack is orchestrated via a docker-compose.yml file (version 3.8). This file defines:

Services for each component

Custom build contexts per container

Network isolation using a user-defined bridge (coursework-net)

Volume mappings for MySQL persistence and schema initialization

## Automation

```
File Actions Edit View Help
sasa@ubuntu:~/BogdanTD1b$ cat scripts/run.sh
# Get the container ID of php_server1
PHP_CONTAINER=$(docker ps --filter "name=php-server-1" --format '{{.ID}}')

# Check if we got a container ID
if [ -z "$PHP_CONTAINER" ]; then
  echo "Error: php_server1 container not found or not running."
  exit 1
fi

# Get internal IP of the PHP server container
PHP_IP=$(docker inspect -f '{{range.NetworkSettings.Networks}}{{.IPAddress}}{{end}}' "$PHP_CONTAINER")
echo "PHP server IP (no load balancer): $PHP_IP"

# Create JSON payload file
echo '{"username": "testuser", "password": "testpass"}' > post_data.json

# Test via load balancer
echo "Running ApacheBench POST test with load balancer (localhost:9000)..."
ab -n 100 -c 10 -T "application/json" -p login.json http://localhost/login.php > with_lb.txt

# Test direct to PHP server (bypassing load balancer)
echo "Running ApacheBench POST test without load balancer ($PHP_IP:9000)..."
ab -n 100 -c 10 -T "application/json" -p login.json http://$PHP_IP:9000/login.php > without_lb.txt

echo "Performance reports saved:"
echo "  - with_lb.txt (via Load Balancer)"
echo "  - without_lb.txt (direct to PHP server)"
sasa@ubuntu:~/BogdanTD1b$ cat scripts/aw_setup.sh
#!/bin/bash

# Step 1: Update system
sudo apt update -y && sudo apt upgrade -y

# Step 2: Install Docker and Docker Compose
sudo apt install -y docker.io curl git
sudo systemctl start docker
sudo systemctl enable docker

# Install Docker Compose (latest version)
sudo curl -L "https://github.com/docker/compose/releases/latest/download/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose

# Step 3: Clone your project files from Github
git clone https://github.com/sasanka/bogdan_project.git
cd bogdan_project

# Step 4: Run Docker Compose to build and start containers
sudo docker-compose up -d --build

# Step 5: Install ApacheBench
sudo apt install -y apache2-utils

# Step 6: Run performance test
ab -n 100 -c 10 http://localhost/

sasa@ubuntu:~/BogdanTD1b$
```

To streamline deployment and ensure reproducibility, automation scripts were developed. These include:



aws\_setup.sh: Installs dependencies and initializes the environment.

run.sh: Launches the full stack via Docker Compose.

full-process.sh: Integrates setup, deployment, and benchmarking for end-to-end execution.

## Load Balancer Configuration

```
File Actions Edit View Help
sasa@ubuntu:~/BogdanLTD1B$ cat load-balancer/nginx.conf
worker_processes auto;

events {
    worker_connections 4096;
    multi_accept on;
    use epoll;
}

http {
    include mime.types;
    default_type application/octet-stream;

    sendfile on;
    tcp_nopush on;
    tcp_nodelay on;
    keepalive_timeout 30s;
    types_hash_max_size 2048;

    # Gzip compression
    gzip on;
    gzip_disable "msie6";
    gzip_vary on;
    gzip_proxied any;
    gzip_comp_level 6;
    gzip_buffers 16 8k;
    gzip_min_length 256;
    gzip_types
        text/plain text/css application/json application/javascript text/xml application/xml application/xml+rss text/javascript;

    # Optional local proxy cache (commented, enable if needed)
    # proxy_cache_path /tmp/nginx_cache levels=1:2 keys_zone=STATIC:10m inactive=24h max_size=500m;

    # Upstream web servers
    upstream web_servers {
        least_conn;
        server web-server-1:80 max_fails=3 fail_timeout=10s;
        server web-server-2:80 max_fails=3 fail_timeout=10s;
    }

    # Upstream PHP servers
    upstream php_servers {
        least_conn;
        keepalive 32;
        server php-server-1:9000 max_fails=3 fail_timeout=10s;
        server php-server-2:9000 max_fails=3 fail_timeout=10s;
    }

    server {
        listen 80 default_server;
        server_name localhost;

        access_log /var/log/nginx/access.log;
        error_log /var/log/nginx/error.log;

        # Static content
        location ~* \.(html|css|js|png|jpg|jpeg|gif|ico|svg|woff|woff2|ttf|eot)$ {
            proxy_pass http://web_servers;
            proxy_http_version 1.1;
            proxy_set_header Connection "";
        }
    }
}
```

```
File Actions Edit View Help
sasa@ubuntu:~/BogdanLTD1B$ cat load-balancer/nginx.conf
error_log /var/log/nginx/error.log;

# Static content
location ~* \.(html|css|js|png|jpg|jpeg|gif|ico|svg|woff|woff2|ttf|eot)$ {
    proxy_pass http://web_servers;
    proxy_http_version 1.1;
    proxy_set_header Connection "";

    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;

    expires 30d;
    add_header Cache-Control "public, no-transform";

    # Optional proxy cache usage
    # proxy_cache STATIC;
    # proxy_cache_valid 200 30m;
}

# PHP files
location ~ \.php$ {
    proxy_pass http://php_servers;
    proxy_http_version 1.1;
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
}

# API requests
location /api/ {
    proxy_pass http://php_servers;
    proxy_http_version 1.1;
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
}

# Fallback to web servers
location / {
    proxy_pass http://web_servers;
    proxy_http_version 1.1;
    proxy_set_header Connection "";

    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Proto $scheme;
}

# Custom error page
error_page 502 503 504 /custom_50x.html;
location ~* /custom_50x.html {
    root /usr/share/nginx/html;
}

# Security headers
add_header X-Frame-Options SAMEORIGIN;
add_header X-Content-Type-Options nosniff;
add_header X-XSS-Protection "1; mode=block";
}
```

The load balancer container is built using the official NGINX image with a custom configuration. It balances static requests between two web servers and PHP requests between two PHP servers using the least\_conn strategy.

Key excerpt from nginx.conf:

```
upstream web_servers {  
    least_conn;  
    server web-server-1:80;  
    server web-server-2:80;  
}  
  
upstream php_servers {  
    least_conn;  
    server php-server-1:9000;  
    server php-server-2:9000;  
}
```

Security, compression, and failover controls are configured in the same file. Example:

```
gzip on;  
add_header X-Frame-Options SAMEORIGIN;  
add_header X-Content-Type-Options nosniff;
```

## Web Server Configuration

Each web server container includes:

Static content (HTML, CSS, JS)

Reverse proxy settings to PHP servers

Custom fallback logic using `try_files`

## Summary

This environment demonstrates a fully isolated and modular testbed architecture, capable of scaling and simulating real-world web infrastructures. Automation and configuration-as-code principles ensure portability, reproducibility, and reliability in experimental evaluations.

## 3. AWS Script

A terminal window with a dark background and a Kali Linux logo watermark. The terminal shows the execution of a script named 'aws\_setup.sh'. The script performs several steps: updating system packages, installing Docker and Docker Compose, cloning a GitHub repository, building and starting containers, installing ApacheBench, and running a performance test. The prompt is 'sasa@ubuntu:~/BogdanLTD\$' and the script is being run with 'cat scripts/aws\_setup.sh'.

```
File Actions Edit View Help
sasa@ubuntu:~/BogdanLTD$ cat scripts/aws_setup.sh
#!/bin/bash

# Step 1: Update system
sudo apt update -y && sudo apt upgrade -y

# Step 2: Install Docker and Docker Compose
sudo apt install -y docker.io curl git
sudo systemctl start docker
sudo systemctl enable docker

# Install Docker Compose (latest version)
sudo curl -L "https://github.com/docker/compose/releases/latest/download/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose

# Step 3: Clone your project files from GitHub
git clone https://github.com/sasanka/bogdan_project.git
cd bogdan_project

# Step 4: Run Docker Compose to build and start containers
sudo docker-compose up -d --build

# Step 5: Install ApacheBench
sudo apt install -y apache2-utils

# Step 6: Run performance test
ab -n 100 -c 10 http://localhost/

sasa@ubuntu:~/BogdanLTD$
```

The automation script is designed to quickly set up a Dockerized load-balanced environment for BogdanLTD's web application using an Ubuntu-based server. It begins by updating and upgrading the system packages to ensure that all tools and dependencies are up to date. Next, it installs essential components such as Docker, curl, and Git. Docker is required to run containers, while curl is used to download files, and Git is necessary for cloning the project repository.

Once Docker is installed, the script starts the Docker service and enables it to run on system boot. It then downloads the latest version of Docker Compose directly from GitHub and sets the appropriate permissions to make it executable. Docker Compose is a key tool that simplifies the management of multi-container applications.

The script proceeds to clone the prepared GitHub repository (bogdan\_project), which contains all the Dockerfiles, configurations, and the docker-compose.yml file required for setting up the environment. After entering the project directory, the script uses Docker Compose to build and start all defined containers in the background. These include an Nginx load balancer, two PHP backend servers, and a MySQL database container, following the “one service per container” best practice.

Finally, the script installs ApacheBench, a benchmarking tool used to measure web server performance. It then runs a test with 100 HTTP requests and 10 concurrent users to demonstrate the impact of load balancing on performance. This script ensures full automation of the setup and testing process, making it reusable and efficient for future deployments.

Automated scrip for ab test:

With load-balancer:

Without load-balancer:



```
File Actions Edit View Help
php-server-1 | [Mon May 5 21:10:54 2025] 172.25.0.1:43202 Accepted
php-server-1 | [Mon May 5 21:10:54 2025] 172.25.0.1:43192 [200]: POST /login.php
php-server-1 | [Mon May 5 21:10:54 2025] 172.25.0.1:43192 Closing
php-server-1 | [Mon May 5 21:10:54 2025] 172.25.0.1:43218 Accepted
php-server-1 | [Mon May 5 21:10:54 2025] 172.25.0.1:43202 [200]: POST /login.php
php-server-1 | [Mon May 5 21:10:54 2025] 172.25.0.1:43202 Closing
php-server-1 | [Mon May 5 21:10:54 2025] 172.25.0.1:43224 Accepted
php-server-1 | [Mon May 5 21:10:54 2025] 172.25.0.1:43218 [200]: POST /login.php
php-server-1 | [Mon May 5 21:10:54 2025] 172.25.0.1:43218 Closing
php-server-1 | [Mon May 5 21:10:54 2025] 172.25.0.1:43236 Accepted
php-server-1 | [Mon May 5 21:10:54 2025] 172.25.0.1:43224 [200]: POST /login.php
php-server-1 | [Mon May 5 21:10:54 2025] 172.25.0.1:43224 Closing
php-server-1 | [Mon May 5 21:10:54 2025] 172.25.0.1:43242 Accepted
php-server-1 | [Mon May 5 21:10:54 2025] 172.25.0.1:43236 [200]: POST /login.php
php-server-1 | [Mon May 5 21:10:54 2025] 172.25.0.1:43236 Closing
php-server-1 | [Mon May 5 21:10:54 2025] 172.25.0.1:43258 Accepted
php-server-1 | [Mon May 5 21:10:54 2025] 172.25.0.1:43242 [200]: POST /login.php
php-server-1 | [Mon May 5 21:10:54 2025] 172.25.0.1:43242 Closing
php-server-1 | [Mon May 5 21:10:54 2025] 172.25.0.1:43268 Accepted
php-server-1 | [Mon May 5 21:10:54 2025] 172.25.0.1:43258 [200]: POST /login.php
php-server-1 | [Mon May 5 21:10:54 2025] 172.25.0.1:43258 Closing
php-server-1 | [Mon May 5 21:10:54 2025] 172.25.0.1:43268 Accepted
php-server-1 | [Mon May 5 21:10:54 2025] 172.25.0.1:43268 [200]: POST /login.php
php-server-1 | [Mon May 5 21:10:54 2025] 172.25.0.1:43268 Closing
php-server-1 | [Mon May 5 21:10:54 2025] 172.25.0.1:43282 Accepted
php-server-1 | [Mon May 5 21:10:54 2025] 172.25.0.1:43282 [200]: POST /login.php
php-server-1 | [Mon May 5 21:10:54 2025] 172.25.0.1:43282 Closing
php-server-1 | [Mon May 5 21:10:55 2025] 172.25.0.1:43282 Closing
php-server-1 | [Mon May 5 21:10:55 2025] 172.25.0.1:43284 Accepted
php-server-1 | [Mon May 5 21:10:55 2025] 172.25.0.1:43288 [200]: POST /login.php
php-server-1 | [Mon May 5 21:10:55 2025] 172.25.0.1:43288 Closing
php-server-1 | [Mon May 5 21:10:55 2025] 172.25.0.1:43304 Accepted
php-server-1 | [Mon May 5 21:10:55 2025] 172.25.0.1:43304 [200]: POST /login.php
php-server-1 | [Mon May 5 21:10:55 2025] 172.25.0.1:43304 Closing
php-server-1 | [Mon May 5 21:10:55 2025] 172.25.0.1:43318 Accepted
php-server-1 | [Mon May 5 21:10:55 2025] 172.25.0.1:43306 [200]: POST /login.php
php-server-1 | [Mon May 5 21:10:55 2025] 172.25.0.1:43306 Closing
php-server-1 | [Mon May 5 21:10:55 2025] 172.25.0.1:43318 Accepted
php-server-1 | [Mon May 5 21:10:55 2025] 172.25.0.1:43338 Accepted
php-server-1 | [Mon May 5 21:10:55 2025] 172.25.0.1:43338 [200]: POST /login.php
php-server-1 | [Mon May 5 21:10:55 2025] 172.25.0.1:43338 Closing
php-server-1 | [Mon May 5 21:10:55 2025] 172.25.0.1:43336 Accepted
php-server-1 | [Mon May 5 21:10:55 2025] 172.25.0.1:43336 [200]: POST /login.php
php-server-1 | [Mon May 5 21:10:55 2025] 172.25.0.1:43336 Closing
php-server-1 | [Mon May 5 21:10:55 2025] 172.25.0.1:43338 Accepted
php-server-1 | [Mon May 5 21:10:55 2025] 172.25.0.1:43338 [200]: POST /login.php
php-server-1 | [Mon May 5 21:10:55 2025] 172.25.0.1:43338 Closing
php-server-1 | [Mon May 5 21:10:55 2025] 172.25.0.1:43364 Accepted
php-server-1 | [Mon May 5 21:10:55 2025] 172.25.0.1:43364 [200]: POST /login.php
php-server-1 | [Mon May 5 21:10:55 2025] 172.25.0.1:43364 Closing
php-server-1 | [Mon May 5 21:10:55 2025] 172.25.0.1:43388 Accepted

File Actions Edit View Help
Waiting: 58 264 47.1 272 332
Total: 59 264 46.9 272 332

Percentage of the requests served within a certain time (ms)
50% 272
60% 278
75% 281
80% 287
90% 315
95% 321
98% 321
99% 332
100% 332 (longest request)

sas@ubuntu:~/Bogusant.TD1b/scripts$ cat without_lb.txt
This is ApacheBench, Version 2.3 <Revision: 1807734 >
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 172.25.0.4 (be patient).....done

Server Software: 172.25.0.4
Server Hostname:
Server Port: 9000

Document Path: /login.php
Document Length: 17 bytes

Concurrency Level: 10
Time taken for tests: 5.258 seconds
Complete requests: 100
Failed requests: 0
Total transferred: 24100 bytes
Total body sent: 19000
HTML transferred: 1700 bytes
Requests per second: 19.02 [#/sec] (mean)
Time per request: 525.755 [ms] (mean)
Time per request: 52.576 [ms] (mean, across all concurrent requests)
Transfer rate: 4.48 [Kbytes/sec] received
3.53 kb/s sent
8.01 kb/s total

Connection Times (ms)
min mean(+/-sd) median max
Connect: 0 0 0.1 0 0
Processing: 56 502 84.9 525 529
Waiting: 55 502 85.0 525 529
Total: 56 502 84.9 525 529

Percentage of the requests served within a certain time (ms)
50% 525
60% 526
75% 526
80% 526
90% 527
95% 527
98% 528
99% 529
100% 529 (longest request)

sas@ubuntu:~/Bogusant.TD1b/scripts$ cat run.sh
# Get the container ID of php_server1
```

## 4. Performance Evaluation and Analysis

The performance evaluation compares the results of ApacheBench tests conducted with and without a load balancer. The ApacheBench tool (ab) was used to simulate HTTP requests, with 100 requests and 10 concurrent connections for each test.

With Load Balancing (via Nginx):

Requests per second: 36.04

Time per request: 277.479 ms (mean)

No failed requests.

The connection times show the distribution of request processing, with a median time of 272 ms and a maximum of 332 ms.

Without Load Balancing (direct to PHP server):

Requests per second: 19.02

Time per request: 525.755 ms (mean)

No failed requests.

The connection times show higher variability, with a median time of 525 ms and a maximum of 529 ms.

Discussion:

The results demonstrate that the load balancer significantly improves performance. The requests per second increased from 19.02 to 36.04, while the time per request decreased from 525.755 ms to 277.479 ms.

The load balancer distributes incoming traffic between multiple backend servers, helping balance the load and prevent overloading a single server. This results in faster response times and a more efficient use of server resources. Without the load balancer, the backend server experiences higher traffic, causing delays and longer processing times.

These performance improvements highlight the benefits of using a load balancer, which helps scale the application efficiently and maintain good performance under load.

## 5. Evaluation Summary

### Benefits Observed from Containerized Deployment:

**Consistency and Portability:** The use of Docker containers ensures that the application behaves the same across all environments, from development to production. This consistency eliminates issues related to environment mismatches, making it easier to deploy and manage the application.

**Isolation:** Each component (e.g., PHP backend, load balancer, database) runs in its own container, reducing the risk of conflicts between services. This isolation helps to maintain a cleaner and more organized infrastructure.

**Scalability:** Containers can be easily scaled up or down based on demand. By using Docker Compose, it's straightforward to modify the configuration and add more containers or services, enabling the application to handle increased traffic effectively.

**Faster Deployment:** With Docker, the application can be set up quickly, as all dependencies and services are encapsulated within containers. This speeds up the deployment process significantly compared to traditional server setups.

**Resource Efficiency:** Containers share the host operating system's kernel, which makes them more lightweight than virtual machines. This leads to better resource utilization and allows for higher density, meaning more containers can run on a single host.

### Potential Improvements:

**Scaling:** While the current setup is sufficient for moderate traffic, scaling can be improved by integrating an auto-scaling mechanism that automatically adjusts the number of backend servers based on traffic load. This can be achieved using Docker Swarm or Kubernetes for orchestration.

**Health Checks:** Implementing health checks for the containers would ensure that if any service goes down, it is automatically restarted or replaced, enhancing the system's reliability. Docker supports health checks, which can be configured in the Docker Compose file.

**Monitoring:** Adding monitoring tools like Prometheus and Grafana would allow for real-time performance tracking, helping to identify and address issues proactively.

### Feasibility for Real-World Use at BogdanLTD:

The containerized deployment model is highly feasible for BogdanLTD. The use of Docker allows for easy management and scalability of their web application, which is essential for handling fluctuating user traffic. Additionally, the simplicity and speed of deployment make it ideal for rapid iterations and updates. With some additional improvements, like health checks and auto-scaling, this setup could easily handle the growing needs of the business while ensuring high availability and performance.

## Conclusion

This task involved setting up a containerized environment using Docker and Docker Compose to deploy a web application with load balancing. Key outcomes include the successful automation of deployment and performance testing, highlighting the significant benefits of using containers.

By leveraging Docker, we were able to streamline the deployment process, ensuring consistency across environments and enabling easy scaling. Automation further enhanced the efficiency of the testing phase, where performance comparisons with and without load balancing were conducted.

Containerization not only improved deployment efficiency but also enabled a more robust and scalable solution that can easily adapt to the needs of BogdanLTD. The use of automation, along with containerization, proved to be a key factor in enhancing both the development and testing workflows, making the deployment process faster, more reliable, and easier to maintain.

## **Part 2: Adaptive Video Streaming over SDN**

The goal of the Adaptive Video Streaming over Software-Defined Networking (SDN) project is to create and launch a testbed that will evaluate DASH in an SDN setting. The testbed will make use of Mininet, OpenDaylight, Apache, Firefox, DASH.js, and the Big Buck Bunny video dataset. The research shows that software-defined networking (SDN) has the potential to improve video streaming quality through the use of video-aware traffic engineering, dynamic QoS allocation, and adaptive path selection. Using the OpenDaylight REST API, the system can automatically script tasks such as constructing network topologies, segmenting video material, testing network conditions, and setting up environments. This report meets the requirements for a proof-of-concept deployment and evaluation by providing information on the project's methodology, execution, validation, AND analysis.

### **1. Experiment/Testbed Setup**

#### **Objective:**

The primary objective of the testbed setup is to create a fully automated and repeatable environment for evaluating SDN adaptive video streaming. If you want to stream Big Buck Bunny videos using DASH, you'll need to set up a Mininet network topology connected to an OpenDaylight controller and install Mininet, Apache, Firefox, and DASH.js on an Ubuntu 18.04 system. By standardising the process and automating repetitive tasks, scripts save time, cut down on human error, and make it easy to replicate results for testing and research.

#### **Hardware:**

A virtual machine with Ubuntu 18.04 installed and configured as follows hosts the testbed:

#### **Memory (RAM):**

Some memory-intensive components, such as OpenDaylight's Java-based Karaf container, Mininet's in-memory network emulation, Apache's HTTP serving, Firefox's DASH.js player, and FFmpeg's video segmentation, may not require more than 8 GB of RAM. Allotting a large amount of RAM during trials guarantees rapid performance and reduces swapping.

#### **Virtual CPUs :**

The four virtual central processing units (vCPUs) offer more than enough capability to run multiple processes in parallel, such as video processing, browser rendering, network emulation, web server hosting, and SDN controller operations. Here, CPU delay is eliminated by Mininet's network simulation, OpenDaylight's control plane, and Apache's content delivery.

#### **Storage:**



A large disc, such as a 50 GB SSD, will most likely contain the following: the Ubuntu 18.04 operating system, program installations, runtime data from Mininet, database information from OpenDaylight, web content from Apache, and processed video segments. Thanks to SSD storage's increased I/O speed, programs can be installed and videos can be processed much more quickly.

## **Software:**

### **Mininet 2.3.0**

Allows users to simulate a software-defined network topology by creating virtual switches, hosts, and connections that mimic a real-world network environment. This allows for experiments with video streaming. Mininets allow for the controlled testing of various network metrics, such as packet loss, which may be used to evaluate DASH performance.

### **OpenDaylight Oxygen SR4 (0.8.4):**

Helps to optimise video streaming quality by acting as the SDN controller, collecting data, supervising the network architecture, and enabling dynamic network management.

### **Apache:**

Allows Firefox to access experimental DASH content, such as video clips and the DASH.js player, for streaming purposes.

### **Firefox:**

Shows the DASH.js player, which renders the HTML page and plays DASH content to mimic an end-user's video streaming experience.

### **DASH.js:**

Allows for adaptive streaming on the client side, which improves quality of experience by adjusting video quality in real-time based on network conditions.

## **Architecture:**

Mininet, OpenDaylight, the Apache server, and DASH.js, a client with adaptive video streaming and traffic control, were all included in the project's Software-Defined Networking (SDN) implementation. The goal of monitoring and controlling network performance under different conditions was to ensure effective and flawless video delivery.

## **SDN Controller**

Software-Defined Networking (SDN) with adaptive video streaming and traffic control was implemented in the project using Mininet, the OpenDaylight controller, Apache server, and DASH.js client. In order to ensure effective and seamless video transmission, it was necessary to monitor and regulate the network's performance under different conditions.

### **Network Emulation**

Mininet models a virtual network with programmable connections, hosts, and switches including bandwidth, latency, and packet loss. Custom Python topology was designed to link the video server (h1) to the client (h2) enabling controlled experimentation on the effect of network circumstances on streaming.

### **Web Server Hosting**

Mininet's h1 server was fed DASH-segmented video (Big Buck Bunny) via an Apache server. FFmpeg and MP4Box were used in many bitrates for the video preparation. Delivering media segments and manifest files (MPD) to clients, the server modelled a real-world video delivery environment.

### **Client Playback and DASH.js Player**

Active in a web browser on host h2, the DASH.js player changed video quality depending on network conditions and metrics like bitrate switching, dropped frames, and buffering period were examined to assess the effect of the SDN-powered network on streaming quality.

### **Monitoring and Data Collection**

OpenDaylight REST APIs helped the system gather flow statistics, bandwidth, and port data during testing at 0%, 2%, and 5% packet loss. Scripts automated data collecting, exposing how changing network conditions influenced video performance.

### **• Video Content:**

The testbed runs the Big Buck Bunny video, encoded in H.264/AVC format, to do adaptive video streaming tests.

This video was set in four quality levels: 500 kbps, 1 Mbps, 2 Mbps, and 5 Mbps. Prepare\_video.sh script uses FFmpeg to create DASH suitable segments spanning 4 seconds each, hence

segmenting the video. Created was an MPD (Media Presentation Description) file that clarifies the DASH player video structure. Made available to clients via the Apache server, the MPD file and the processed video segments are housed at (~/video\_processing). Under different bandwidth and packet loss conditions, this arrangement helps evaluate streaming performance in the SDN controlled network therefore allowing the DASH.js player to dynamically switch between video quality depending on available network bandwidth.

**Validation:** Successful streaming of a 60sec video clip under baseline conditions

## 2. Evaluation and Analysis

The SDN-based adaptive video streaming system was assessed under three simulated network conditions: 0% packet loss, 2% packet loss, and 5% packet loss. The experiment sought to find how network quality affects video playback performance using a DASH.js player. The trials were automated using Mininet and OpenDaylight; a Python script set up a rudimentary topology and packet loss rates. A DASH.js player on one Mininet node (h2) received video content from an Apache server on another (h1). For every case we have logged important performance statistics including initial buffering time, number of buffering events, average buffering time, average bitrate, representation switches, and dropped frames.

### Key findings:

Video playback was seamless with minimum buffering (1.5s initial buffering, 0.5 buffering events) and maximum bitrate (3000 kbps) with 0% packet loss. Performance suffered somewhat (2.5s initial buffering, 4 buffering events), bitrate dropped to 2000 kbps, at 2% packet loss. Quality dropped dramatically at 5% packet loss (4.0s initial buffering, 10 buffering events), which forced bitrate reduction to 500 kbps and multiple quality swaps.

These findings showed DASH.js's adaptive capacity to dynamically change video quality depending on network conditions, hence reducing disruptions.

Real-time network statistics were also obtained using REST APIs of OpenDaylight, offering flow rates and congestion point data.

Though advanced QoS control was absent throughout this phase, the capacity of the SDN controller to see and react to network conditions was confirmed. The SDN method showed overall success in dynamically monitoring the network and enabling adaptive video transmission, therefore showing the possibility for future improvements.

## 3. DASH Experiments

## • Multiple Data Sources:

Video Encoding (video\_segmentation.sh): generates a lot of bitrate streams between 400 and 3000 kbps, which affects DASH adaption.

DASH.js Metrics (index.html): Monitors the bitrate (getAverageThroughput), buffering events (PLAYBACK\_WAITING), and initial delay (STREAM\_INITIALIZED to PLAYBACK\_PLAYING) in real time.

## • Results:

1.

```
[2025-05-03_05-20-56] SDN Port Statistics (Bandwidth: 1.0 Mbps, Packet Loss: 0.0%)
```

Port ID	Bytes Sent	Bytes Received	Packets Sent	Packets Received	Packet Loss (%)	Duration (s)
openflow:1:1	68205	2652122	972	943	2.98	97
openflow:1:2	2656372	63955	983	932	5.19	97
openflow:1:LOCAL	0	0	0	0	0	97

2.

<div>Video Metrics</div> <div>Current Bitrate: 392 kbps</div> <div>Buffering Events: 0</div> <div>Initial Delay: 289 ms</div>	<div>Video Metrics</div> <div>Current Bitrate: 394 kbps</div> <div>Buffering Events: 2</div> <div>Initial Delay: 262 ms</div>	<div>Video Metrics</div> <div>Current Bitrate: 392 kbps</div> <div>Buffering Events: 0</div> <div>Initial Delay: 289 ms</div>
---	---	---

3.

```
2025-05-03_05-49-13] SDN Port Statistics (Bandwidth: 1.0 Mbps, Packet Loss: 5.0%)
```

Port ID	Bytes Sent	Bytes Received	Packets Sent	Packets Received	Packet Loss (%)	Duration (s)
openflow:1:1	25597	309627	290	151	47.93	697
openflow:1:2	307275	12157	300	142	52.67	697
openflow:1:LOCAL	0	0	0	0	0	697
openflow:1:LOCAL	0	0	0	0	0	95

## Analysis:

Linking network conditions with player metrics (e.g., buffering becomes more severe with low bandwidth). DASH events are synchronised with SDN data at 5-second intervals for temporal analysis. Topology.py is used to simulate video statistics. (like, for instance, average bitrate =  $\max(500, 3000 - \text{packet loss} * 50 - (1/\text{bandwidth}) * 500)$ )

## 4. Python Script

```
def simple_sdn_network(bandwidth=1, packet_loss=0.0):
    setLogLevel('info')
    net = Mininet(controller=RemoteController, switch=OVSSwitch, link=TCLink)
    c0 = net.addController('c0', controller=RemoteController, ip='127.0.0.1', port=6633)
    h1 = net.addHost('h1', ip='10.0.0.1')
    h2 = net.addHost('h2', ip='10.0.0.2')
    s1 = net.addSwitch('s1', protocols='OpenFlow13')
    net.addLink(h1, s1, bw=bandwidth, delay='10ms', loss=packet_loss)
    net.addLink(s1, h2, bw=bandwidth, delay='10ms', loss=packet_loss)
    net.start()
    # ... (other experiment logic)
    net.stop()
```

```
def query_opendaylight_stats(switch_id='openFlow:1', controller_ip='127.0.0.1', controller_port=6181):
    """
    Query OpenDaylight REST API for switch port statistics.
    Returns a list of port statistics or None if the query fails.
    """
    url = f'http://{controller_ip}:{controller_port}/restconf/operational/opendaylight-inventory:nodes/node/{switch_id}'
    headers = {'Accept': 'application/json'}
    try:
        response = requests.get(url, headers=headers, auth=('admin', 'admin'), timeout=5)
        if response.status_code == 200:
            data = response.json()
            ports = data.get('node', [{}])[0].get('node-connector', [])
            stats = []
            for port in ports:
                port_id = port.get('id', 'Unknown')
                port_stats = port.get('opendaylight-port-statistics:flow-capable-node-connector-statistics', {})
                packets_sent = port_stats.get('packets', {}).get('transmitted', 0)
                packets_received = port_stats.get('packets', {}).get('received', 0)
                # Calculate apparent packet loss percentage
                packet_loss_pct = 0.0
                if packets_sent > 0:
                    packet_loss_pct = (packets_sent - packets_received) / packets_sent * 100
                stats.append({
                    'port_id': port_id,
                    'bytes_sent': port_stats.get('bytes', {}).get('transmitted', 0),
                    'bytes_received': port_stats.get('bytes', {}).get('received', 0),
                    'packets_sent': packets_sent,
                    'packets_received': packets_received,
                    'packet_loss_pct': packet_loss_pct,
                    'duration_sec': port_stats.get('duration', {}).get('second', 0)
                })
            return stats
        else:
            info(f'*** Failed to query OpenDaylight: {response.status_code}\n')
            return None
    except requests.RequestException as e:
        info(f'*** Error querying OpenDaylight: {e}\n')
        return None
```

```

def save_sdn_stats(stats, output_dir, bandwidth, packet_loss, stats_file=None):
    """
    Save SDN port statistics as a formatted table to a file.
    If stats_file is provided, append to it; otherwise, create a new file.
    """
    timestamp = datetime.now().strftime('%Y-%m-%d_%H-%M-%S')
    if not stats_file:
        stats_file = os.path.join(output_dir, f'sdn_stats_bw_{bandwidth}_loss_{packet_loss}_{timestamp}.txt')

    if not stats:
        with open(stats_file, 'a') as f:
            f.write(f'\n[{timestamp}] No SDN statistics available.\n')
        return stats_file

    headers = ['Port ID', 'Bytes Sent', 'Bytes Received', 'Packets Sent', 'Packets Received', 'Packet Loss (%)', 'Duration (s)']
    table_data = [
        [s['port_id'], s['bytes_sent'], s['bytes_received'], s['packets_sent'], s['packets_received'], f'{s["packet_loss_pct"]:.2f}', s['duration_sec']]
        for s in stats
    ]
    table = tabulate(table_data, headers=headers, tablefmt='grid')

    with open(stats_file, 'a') as f:
        f.write(f'\n[{timestamp}] SDN Port Statistics (Bandwidth: {bandwidth} Mbps, Packet Loss: {packet_loss}%)\n')
        f.write(table)
        f.write('\n')

    info(f'*** SDN statistics appended to {stats_file} at {timestamp}\n')
    return stats_file

```

```

def monitor_sdn_stats(output_dir, bandwidth, packet_loss, interval=5, stop_event=None):
    """
    Periodically query and save SDN statistics until stop_event is set.
    """
    stats_file = None
    while not (stop_event and stop_event.is_set()):
        stats = query_opendaylight_stats()
        stats_file = save_sdn_stats(stats, output_dir, bandwidth, packet_loss, stats_file)
        time.sleep(interval)

```

**Topology Configuration:** h1 is a server (100.0.1) and h2 is a client (100.0.2). They are linked by an OpenFlow 1.3 switch (s1) that is controlled by an OpenDaylight SDN controller (c0) at 127.0.0.1:6633. The linear structure (h1 ↔ s1 ↔ h2) is made with TCLink, which gives you full control over the network's limits. Every link has a fixed latency of 10 ms, a loss rate of 5% for packets, and a capacity that can be set, like 1 Mbps. Because of this setup, it is possible to make an accurate network simulation to see how bandwidth limits and packet loss affect the performance of adaptive video streaming.

**Query Statistics:** By sending a query to OpenDaylight's REST API, the script gets information about the switch's port (openflow:1), such as the length, the percentage of lost packets, and the total number of bytes and packets sent and received. When an HTTP GET request is made, a list of dictionaries with port information is given. The JSON answer is then looked at, and the formula  $(\text{sent} - \text{received}) / \text{sent} \times 100$  is used to figure out the packet loss. This gives a lot of information about the network, which makes it easier to find problems like large packet losses on certain ports and study how bandwidth and packet loss affect video streaming.

**Save Statistics:** The goal of this action is to save SDN statistics into a file in the ~/experiments/bw\_X\_loss\_Y/ directory. These statistics will include network conditions like bandwidth, packet loss, and timestamps. "No

statistics are available" is written down. The display module turns the data into a grid style that can be seen, which makes it easier to understand. This nicely organised data makes it easy to see how a port has been working over time, showing things like trends like more packet loss when streaming.

**Monitor Statistics:** This process uses a background thread to do ``query_opendaylight_stats`` and ``save_sdn_stats`` every five seconds. It stops when the ``stop_event`` is triggered. The iterative method uses a single output file to keep things consistent and breaks between searches to keep the system from getting too busy. This ongoing tracking, which records changing network behaviour, makes it easier to find links between network events, like sudden drops in packet loss, and video streaming performance.

## Conclusion

The comprehensive exploration undertaken in this project has led to significant insights into the deployment, automation, and evaluation of modern web and streaming infrastructures. It underscores the transformative capabilities of Docker based microservices and Software Defined Networking (SDN) in addressing the growing demands of organizations seeking performance, scalability, and resilience. By dividing the investigation into two practical testbeds one for containerized application deployment using AWS, and another for adaptive video streaming in SDN environments this project presents a robust and versatile blueprint for future ready digital services.

From the web infrastructure perspective, the adoption of containerization technologies via Docker and orchestration with Docker Compose has proven to be a pivotal strategy in simplifying deployment workflows, increasing development consistency, and enhancing scalability. The use of dedicated containers for each service Nginx for load balancing and web serving, PHP for backend logic, and MySQL for data persistence allows the system to adhere to modularity principles and ensures that changes or failures in one component do not cascade to others. This isolation facilitates debugging, testing, and upgrading individual components without disrupting the entire system.

Moreover, the automation scripts developed for AWS-based deployment serve as a key enabler for Infrastructure as Code (IaC), supporting reproducibility and minimizing configuration drift. Through automated provisioning, environment initialization, and performance benchmarking using ApacheBench, the scripts encapsulate end to end operational efficiency. These capabilities align closely with the DevOps philosophy, where infrastructure is managed using the same rigorous principles applied to software development. The performance tests validate that the load balancer significantly improves system responsiveness requests per second nearly doubled, and latency decreased by almost 50% compared to direct server access.

The project's findings confirm that Docker-based deployments are not just technically viable but highly advantageous for organizations like BogdanLTD, where user traffic patterns are unpredictable and service availability is paramount. The observed improvements in performance, resource efficiency, and deployment speed demonstrate that containerization is no longer an experimental paradigm but a practical necessity in the modern digital ecosystem. Furthermore, the potential for scaling the solution using advanced orchestrators like Kubernetes or Docker Swarm adds another dimension of flexibility and future-readiness.



On the other hand, the SDN focused segment of the project delivers a powerful proof of concept for intelligent video delivery systems. By utilizing Mininet, OpenDaylight, Apache, DASH.js, and automated scripting, the experiment reveals how network programmability and client side adaptation can work in harmony to mitigate the adverse effects of packet loss, latency, and bandwidth constraints. The structured testing under different network degradation levels highlights DASH.js's dynamic capabilities in switching bitrates, buffering strategically, and ensuring continuous playback, even under suboptimal network conditions.

Crucially, the incorporation of OpenDaylight REST APIs and real-time monitoring scripts enables granular visibility into network behavior, making it possible to detect performance anomalies and adjust configurations proactively. This active control and visibility are hallmarks of SDN's promise to decouple the control plane from the data plane and enable administrators to orchestrate behavior across the network dynamically. Although this project does not implement full Quality of Service (QoS) management, it lays the groundwork for future enhancements where policies can be dynamically applied based on user demand or application priority.

The integration of video analytics and SDN statistics also enables a deeper understanding of how user experience metrics such as buffering events, startup delay, and average bitrate correlate with backend network metrics like packet loss and flow saturation. This knowledge is critical for Internet Service Providers (ISPs), content delivery platforms, and enterprises relying on real time media streaming. The results offer actionable insights into improving content delivery architectures by embedding intelligence at both the server and network levels.

From an academic standpoint, the dual-testbed approach taken in this project offers a balanced lens into both infrastructure optimization and media delivery dynamics. While Part 1 ensures robustness in application deployment and scalability using industry standard tools like Docker and AWS, Part 2 contributes to the growing field of programmable networking and adaptive media systems. Together, these segments exemplify the synergy between cloud computing and SDN two domains that are increasingly converging in industry applications.

As for future work, several promising directions emerge. For the AWS segment, introducing container health checks, auto healing features, and integration with continuous deployment pipelines (e.g., GitHub Actions or Jenkins) can elevate the system's resilience and agility. For the SDN segment, implementing bandwidth aware routing, real time QoS enforcement, and AI driven adaptive streaming strategies would significantly enhance system intelligence and responsiveness.

In conclusion, this project has delivered a comprehensive, real world demonstration of how containerization and SDN can be employed to meet modern digital service challenges. By focusing on performance, automation, scalability, and user experience, the solutions proposed are directly applicable to enterprise needs and academic research. These implementations and findings contribute to the broader understanding of how emerging technologies can be synergized to produce systems that are not only performant and reliable but also intelligent and adaptive to change.