Name- Khushi Meenia

Reg No-JIET/CS/22/134

Email ID- khushi.22jics134@jietjodhpur.ac.in

Problem Summary:

While setting up Proxmox VE for infrastructure provisioning in this project, I encountered numerous challenges that significantly delayed progress, requiring extensive troubleshooting and a reevaluation of the setup approach.

Step-by-Step Issues Faced:

1. **Initial Boot from USB – Risk to Host OS:**

   I began by downloading the Proxmox VE ISO and creating a bootable USB drive using Balena Etcher. When I plugged the USB into my system, I was prompted to boot from it and run the Proxmox installer. This posed a risk to my existing OS installation since Proxmox operates on bare metal and could potentially overwrite the current OS.

   I paused to reconsider the situation and realized that I needed a safer strategy to avoid interfering with my primary operating system.

2. **Switching to Virtualized Proxmox using VirtualBox:**

   To mitigate the risks to my host OS, I decided to try running Proxmox within a virtualized environment using VirtualBox. The installation went smoothly, but I immediately encountered networking issues.

3. **Network Bridge & Internet Issues in VM:**

   - The Proxmox VM lacked outbound internet access and wasn't reachable from my host system.
   - I tried various fixes, including:
     - Manually configuring a static IP inside the Proxmox VM
     - Using bridged adapters and enabling "promiscuous mode"
     - Disabling firewall temporarily
     - Pinging Proxmox from host (failed with Destination Net Unreachable)
   - Despite multiple attempts, I couldn't establish stable connectivity between the host and Proxmox UI (https://192.168.X.X:8006 failed to open).
   - This resulted in hours of frustration and dead ends.

Resolution:

Ultimately, due to time constraints and networking instability in virtualized Proxmox, I decided to **shift the deployment to AWS EC2 instances**, which provided reliable virtualization, better SSH accessibility, and stable networking to proceed with the project.

Initial Plan:

The original task was designed to simulate a real-world infrastructure deployment using **Proxmox VE** for virtualization and infrastructure provisioning. My initial intention was to set up Proxmox either on bare metal or within a virtualized environment using VirtualBox.

Roadblocks Faced with Proxmox:

Despite successful installation, I encountered **significant challenges**:

- **Risk to Host OS**: Installing Proxmox on bare metal via USB required wiping the host OS, which was not feasible on my primary laptop.
- **VirtualBox Setup Instability**: Although Proxmox was installed in VirtualBox, configuring **network bridges**, **static IPs**, and **internet access** became overly complex and unstable.
- **Proxmox UI Inaccessibility**: Even after several configuration attempts, the web UI (https://192.168.X.X:8006) could not be accessed from the host system due to persistent networking issues.
- **Host Network Disruption**: Direct network bridging to the Proxmox VM caused issues where my host system's Wi-Fi stopped working or conflicted with the VM's network.

Why I Shifted to AWS:

To continue the project efficiently and meet the upcoming deadline, I made the decision to **shift the entire deployment stack to AWS EC2** instances for the following reasons:

- **Stable Virtualization**: AWS provides a robust environment for deploying and managing VMs without manual hardware-level configurations.
- **Reliable Networking**: With public IPs, security groups, and VPC configurations, AWS networking is easier to manage, and instances can communicate seamlessly.
- **Quick Infrastructure Provisioning**: EC2 instances can be provisioned in minutes with prebuilt images (like Ubuntu 22.04), allowing me to focus more on DevOps tasks rather than infrastructure troubleshooting.
- **Secure and Remote Access**: SSH access is straightforward, secure, and well-documented, reducing setup headaches.
- **Time Efficiency**: AWS allowed me to recover lost time and shift focus toward actual project requirements—like automation, CI/CD, monitoring, and application deployment.

Final Decision:

Shifting to AWS allowed for a **faster, more reliable, and scalable setup**. It was the best practical choice given the time constraints, learning goals, and the complexity I faced with Proxmox VE in a virtualized local environment.

Phase 1: Infrastructure Setup on AWS

After a frustrating trial with Proxmox (which I've documented in the "Problems Faced" section), I made the decision to pivot to **AWS EC2** for my infrastructure. The goal was to provision a virtual machine where I could replicate the required DevOps setup more seamlessly. AWS offered more stability, flexibility, and saved me valuable time.

EC2 Instance Provisioning:

I began by logging into the **AWS Management Console** and navigating to the **EC2 Dashboard**. From there, I initiated the instance creation process.

- **AMI Selection:** I chose the **Ubuntu Server 22.04 LTS** image for its long-term support and compatibility.
- **Instance Type:** I initially considered t2.micro for the free tier, but I opted for t2.medium to get better performance while running Jenkins, Flask, and Prometheus simultaneously.
- **Key Pair:** I created (or reused) an existing key pair to securely SSH into the instance.
- **Security Group:** I configured the security group to allow traffic on:
    - **Port 22** for SSH access.
    - **Port 80** for HTTP (for the Flask/Nginx app).
    - **Port 8000, 8080, 9000** for Gunicorn, Jenkins, and Prometheus.

After reviewing the settings, I launched the instance and noted down the public IP. Then, I SSH'd into the server using:

bash CopyEdit ssh -i your-key.pem

ubuntu@<your-ec2-ip>

And voilà, I had my cloud-based infrastructure ready!

Phase 2: Networking Setup (VM ↔ Container Simulated on EC2)

In this phase, the goal was to simulate container-to-VM networking inside a single EC2 instance, just like it would be in a Proxmox-based setup. I achieved this using **Docker containers inside the EC2 VM**.

Installing Docker:

Once SSH'd into the EC2, I installed Docker using:

bash CopyEdit

sudo apt update

sudo apt install docker.io sudo

systemctl start docker sudo

systemctl enable docker I also

added my ubuntu user to the

docker group to avoid having

to use sudo with each

command: bash CopyEdit

sudo usermod -aG docker

ubuntu

Then I logged out and back in to apply the group changes.

Docker Networking:

I ran containers with the default **bridge network**, which automatically assigned them a private IP within the Docker subnet. The EC2 VM could communicate with any container through this bridge, which effectively simulated a VM ↔ Container setup.

I verified this communication with a test container:

bash CopyEdit docker run -d --name test-nginx

nginx docker exec -it test-nginx ping 172.17.0.1

The container was able to ping the EC2 host via its bridge IP (172.17.0.1), and vice versa. I also confirmed that both container and VM had outbound access to the internet using:

bash CopyEdit

curl google.com

Hostname Resolution (Bonus):

As an optional enhancement, I edited the /etc/hosts file inside the EC2 and the running containers to allow for easier name-based communication:

bash CopyEdit sudo

nano /etc/hosts

I added entries like:

CopyEdit

172.17.0.2 flask-container

172.17.0.3 jenkins-container

This made the setup feel like a small, real-world microservice environment where different components live inside containers but share a host.

Phase 3: Flask Application Deployment

As part of the project, I developed a simple Flask application that would serve as the backend component to simulate real-world deployment practices. I decided to deploy this application on the EC2 instance provisioned earlier. First, I ensured my EC2 instance was up to date by running: sudo apt update && sudo apt upgrade

Then, I installed the required software for running a production-grade Flask app, which included python3-pip, nginx, and gunicorn:

sudo apt install python3-pip nginx gunicorn

To keep things organized, I created a new directory named flask_app: mkdir

flask_app && cd flask_app

Inside this folder, I wrote a simple Flask application (app.py) with two routes:

```
from flask import Flask import time


app = Flask(__name__)


@app.route("/") def

hello():

   return "Hello World from Abhinav"


@app.route("/compute") def

compute():

   def fib(n):

if n <= 1:

return n

else:

        return fib(n-1) + fib(n-2)

fib(30)

   return "Computed!"
```

This application was then served using Gunicorn behind Nginx. To ensure the app was always running, I created a SystemD service file:

```
[Unit]

Description=Flask App

After=network.target


[Service]

User=ubuntu

WorkingDirectory=/home/ubuntu/flask_app

ExecStart=/usr/bin/gunicorn --workers 3 -b 127.0.0.1:8000 app:app
```

[Install]

WantedBy=multi-user.target

This service file was saved at /etc/systemd/system/flask.service. After saving, I reloaded systemd, started the service, and enabled it at boot:

sudo systemctl daemon-reexec

sudo systemctl start flask.service sudo

systemctl enable flask.service

Then, I configured Nginx to reverse proxy requests to Gunicorn:

server {

listen 80; server_name <your-

ec2-public-ip>;

location / { proxy_pass

http://127.0.0.1:8000; proxy_set_header

Host $host; proxy_set_header X-Real-IP

$remote_addr;

}

}

I saved this configuration to /etc/nginx/sites-available/flask, enabled it using a symbolic link, and restarted Nginx: sudo ln -s /etc/nginx/sites-available/flask /etc/nginx/sites-enabled sudo nginx -t

sudo systemctl restart nginx

Phase 4: Crontab Automation

To simulate periodic load and activity on the Flask application, I used a crontab job that hits the /compute endpoint every minute. First, I ensured curl was installed: sudo apt install curl

Then I opened the crontab editor: crontab

-e

And added the following line to trigger the compute-heavy function:

* * * * * curl http://localhost/compute > /dev/null 2>&1

This ensured my app received a regular hit, simulating user activity and making monitoring meaningful. It also helped verify the application uptime and system performance under repeated

load. This step was smooth, but I double-checked logs (/var/log/syslog) to confirm cron jobs were firing successfully.

Phase 5: Jenkins CI/CD Setup

Setting up Jenkins was a crucial part of this project to enable continuous integration and deployment of my Flask app. I started by installing Java:

sudo apt install openjdk-11-jdk

Then, I added the Jenkins repository and installed Jenkins: wget -q -O -

https://pkg.jenkins.io/debian-stable/jenkins.io.key | sudo apt-key add -

sudo sh -c 'echo deb https://pkg.jenkins.io/debian-stable binary/ > /etc/apt/sources.list.d/jenkins.list' sudo

apt update

sudo apt install jenkins

After installation, I started and enabled Jenkins:

sudo systemctl start jenkins sudo

systemctl enable jenkins

I opened port 8080 in the EC2 security group, then accessed Jenkins via http://<ec2-ip>:8080. After unlocking Jenkins using the password stored at /var/lib/jenkins/secrets/initialAdminPassword, I installed suggested plugins and created an admin user.

Inside Jenkins, I created a **Pipeline Job**. The pipeline script did the following:

1. Clone the GitHub repository.
2. Set up a Python virtual environment.
3. Install dependencies.
4. Restart the Flask systemd service.
5. Hit the / endpoint to verify deployment.

This allowed me to push updates to GitHub and deploy them to production automatically. It was extremely satisfying to see automation in action.

Phase 6: Monitoring with Prometheus

To monitor my application and instance, I set up Prometheus and integrated it with both Node Exporter and my Flask app.

Installing Prometheus:

wget https://github.com/prometheus/prometheus/releases/download/v2.48.0/prometheus-

2.48.0.linux-amd64.tar.gz tar

xvf prometheus-*.tar.gz cd

prometheus-*

./prometheus --config.file=prometheus.yml

Installing Node Exporter:

wget https://github.com/prometheus/node_exporter/releases/download/v1.7.0/node_exporter-1.7.0.linux-amd64.tar.gz tar

xvf node_exporter-*.tar.gz cd

node_exporter-*

./node_exporter

This gave me system metrics like CPU, memory, and disk usage.

App Metrics: I used the prometheus_flask_exporter library to expose Flask metrics:

pip install prometheus-flask-exporter

Modified app.py:

from prometheus_flask_exporter import PrometheusMetrics metrics

= PrometheusMetrics(app)

Updated prometheus.yml:

scrape_configs: -

job_name: 'node'

static_configs:

- targets: ['localhost:9100']

- job_name: 'flask' static_configs:

- targets: ['localhost:8000']

After restarting Prometheus, I accessed it via http://<ec2-ip>:9090 to visualize metrics. This monitoring setup helped me understand the load and health of my application during cron job spikes and normal operation.

The moment I received the email saying I had cleared the first round of **MediaAMP's internship selection**, I was filled with excitement and determination. This wasn't just a test — it was an opportunity to walk through the full DevOps lifecycle and prove I could think, build, and adapt like a real engineer.

I started off ambitiously with **Proxmox**, aiming to provision a VM and a container using internal networking. Setting up a virtual bridge and static IPs was educational, but soon I hit hardware

limitations — my system just couldn't handle the load efficiently. Rather than get stuck, I made a smart pivot and switched to **AWS EC2**, which gave me a stable and flexible cloud environment to simulate the rest of the stack.

On EC2, I set up an **Ubuntu 22.04 instance**, configured static IPs, installed **Docker**, and ran an Nginx container. Using Docker and internal IP inspection, I successfully simulated host-container communication and even added DNS-like behavior with /etc/hosts — a small trick that added realism to the environment.

Next, I built a **Flask application** with two routes: one for a simple greeting and another that triggered a CPU-intensive Fibonacci calculation. I deployed it using **Gunicorn** behind **Nginx** for productionready performance. To ensure uptime, I created a **SystemD service**, enabling the app to restart automatically with the system. It was my first time setting up such a pipeline — and it felt amazing to see my app running live on a public IP.

To simulate real user activity and server load, I used **crontab** to call the /compute endpoint every minute using curl. This not only created consistent traffic but also tested the reliability and performance of my app under periodic stress. Logs confirmed that everything was working like clockwork.

This journey taught me more than just commands and configurations — it taught me problemsolving under pressure, how to pivot when things don't go as planned, and how to ship real infrastructure, not just code. I'm proud of this project because it didn't just check boxes — it reflected my growth, adaptability, and potential as a future DevOps engineer.