Virtualization and Cloud computing

Amresh Kumar M23CSA004

Assignment- 03

Github link:https://github.com/m23csa004/vcc_assignment3

Video link: m23csa004_vcc_assign3.mkv

1. Introduction

This report outlines the step-by-step implementation of a local VM-based resource monitoring system that triggers autoscaling in Google Cloud Platform (GCP). The system monitors CPU utilization and automatically provisions additional instances when the CPU usage crosses a defined threshold.

I have used **GCP** as Public cloud and **virtual box** as hypervisor and **psutil** as a resource monitoring tool.

2. Step-by-Step Implementation

Step 1: Setting Up a Local VM

We begin by creating a virtual machine (VM) locally using VirtualBox:

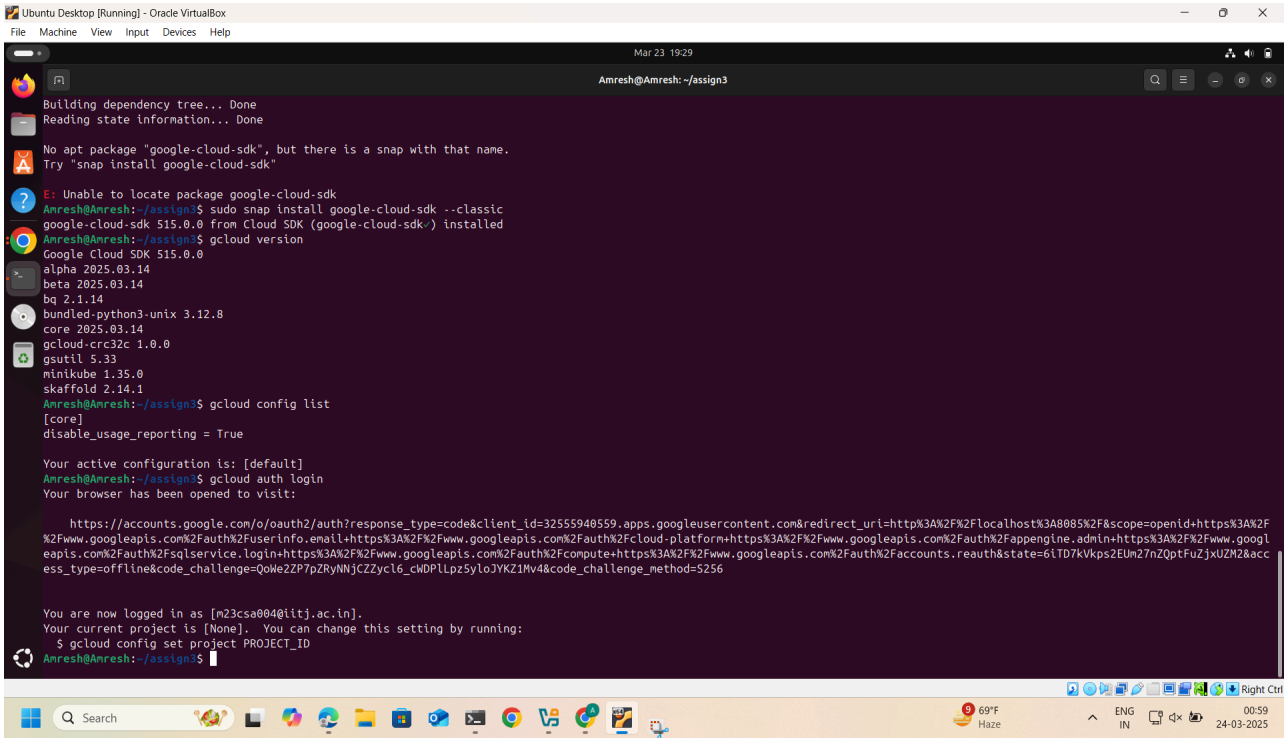
- Download and install **VirtualBox**.
- Use an **Ubuntu Desktop 24.04** ISO image to install Ubuntu.
- Complete the installation and upgrade **Guest Additions** for better performance.

Step 2: Install Google Cloud SDK

- Install Google Cloud SDK:**sudo snap install google-cloud-sdk --classic**
- Verify installation: **gcloud version**
- Authenticate with Google Cloud: **gcloud auth login**
- Get project lists and project id: **gcloud projects list**

  I have used the same project as i have used in the assignment 2 My First Project

- Set project configuration: **gcloud config set project <your-project-id>**
- Enable Compute Engine API: **gcloud services enable compute.googleapis.com**



Step 3: Setting Up Python Environment

- Install Python and required packages:
    - **sudo apt update && sudo apt install -y python3-pip python3-venv**
- Create a virtual environment:
    - **python3 -m venv venv**
    - **source venv/bin/activate**
- Install dependencies:
    - **pip install flask psutil google-auth-oauthlib google-auth-httplib2 google-api-python-client**
    - **Flask:** A lightweight web framework for building web applications in Python.
    - **psutil:** A library for retrieving system and process-related information such as CPU, memory, and disk usage.
    - **google-auth-oauthlib:** A library for handling OAuth 2.0 authentication in Python applications.
    - **google-auth-httplib2**: Provides authentication support for `httplib2`, enabling secure HTTP requests using Google authentication.

- **google-api-python-client:** A client library for accessing various Google APIs, such as Drive, Gmail, and Sheets.

Step 4: Implementing Resource Monitoring

We develop a **Flask-based monitoring application** (**`app.py`**) that:

- Monitors CPU usage.
- Runs a background thread that checks CPU utilization.
- Triggers the GCP VM creation script when CPU usage exceeds **75%.**

Step 5: Implementing Auto-Scaling Script

We create a Python script (**`create_gcp_vm.py`)** that

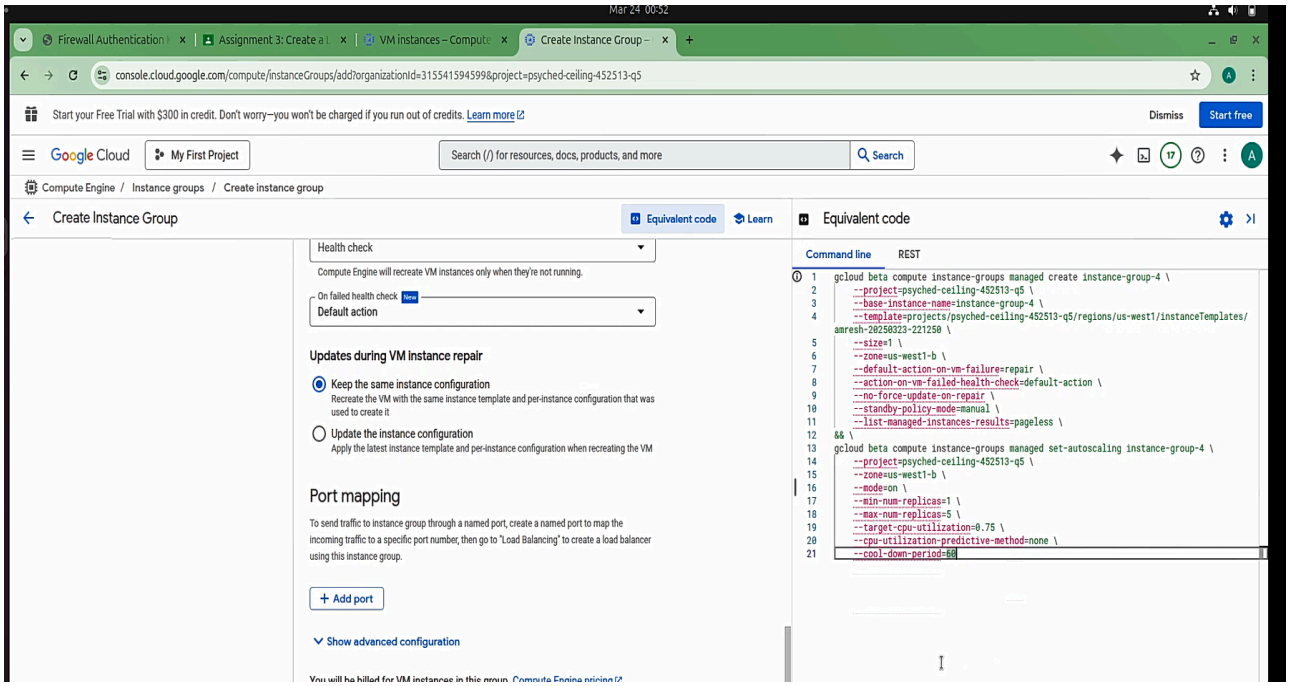
- Defines a **startup script** for VM instances.
- Creates an **Instance Group** with autoscaling rules.
- Uses **gcloud commands** to launch new instances when CPU utilization exceeds 75%.

Steps to Generate the Autoscaling Command

1. **Create an Instance Template in GCP**:
    - Go to **Compute Engine > Instance Templates** in the GCP Console.
    - Click **Create Instance Template** and configure the following:
        - **Name**: Choose a unique identifier.
        - **Region**: Select your preferred region.
        - **Machine Configuration**: Choose **e2-micro** for cost efficiency.
        - **Firewall Settings**: Enable **Allow HTTP and HTTPS traffic**.
    - Under **Advanced > Management > Automation**, paste the **startup script** that initializes the instance.
    - Click **Create**.

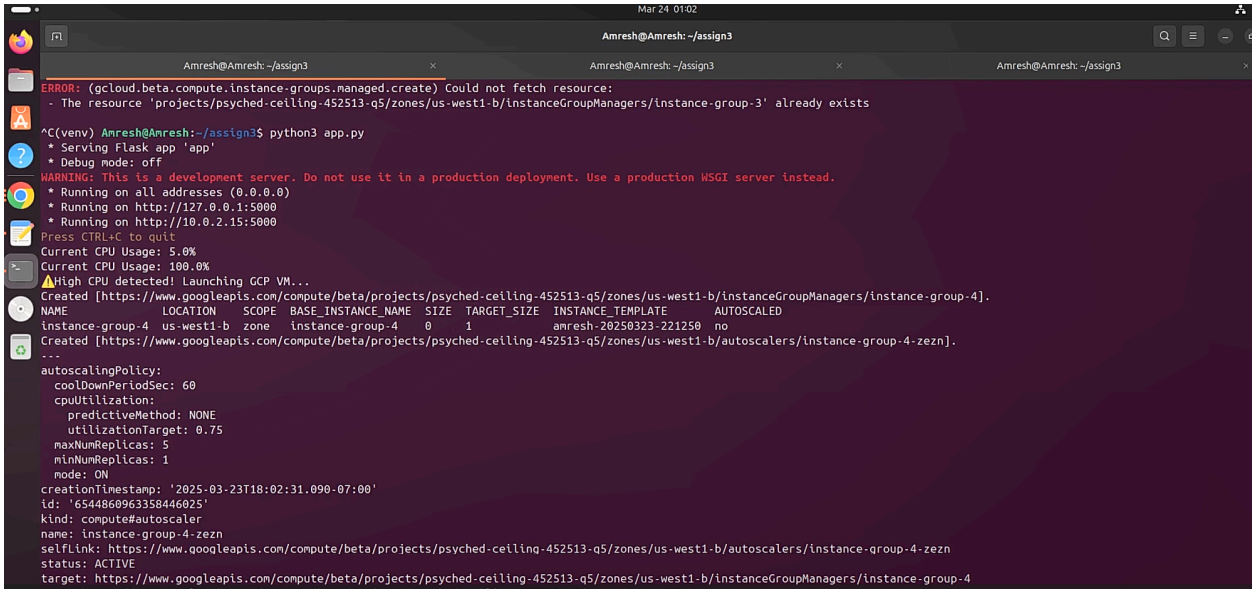
2. **Create a Managed Instance Group**:
    - Navigate to **Compute Engine > Instance Groups**.
    - Click **Create Instance Group** and select the previously created **Instance Template**.
    - Enable **Autoscaling** with the following parameters:
        - **Minimum instances**: 1
        - **Maximum instances**: 5
        - **CPU utilization threshold**: 75%
    - Save the configuration.
3. **Generate the Equivalent gcloud Command**:
    - Once the instance group is configured, click **Equivalent Code** (top-right corner of the GCP Console).
    - Copy the generated command and include it in the Python script.

This ensures that the autoscaling behavior is dynamically managed based on CPU usage.



Step 6: Running the Application Locally

- Start the Flask app: **python3 app.py**
- Open another terminal and simulate high CPU usage:
    - **sudo apt install -y stress**
    - **stress --cpu 4 --timeout 60s**
    -
- Observe that CPU usage spikes and triggers the GCP VM creation script.



Why Simulate Load?

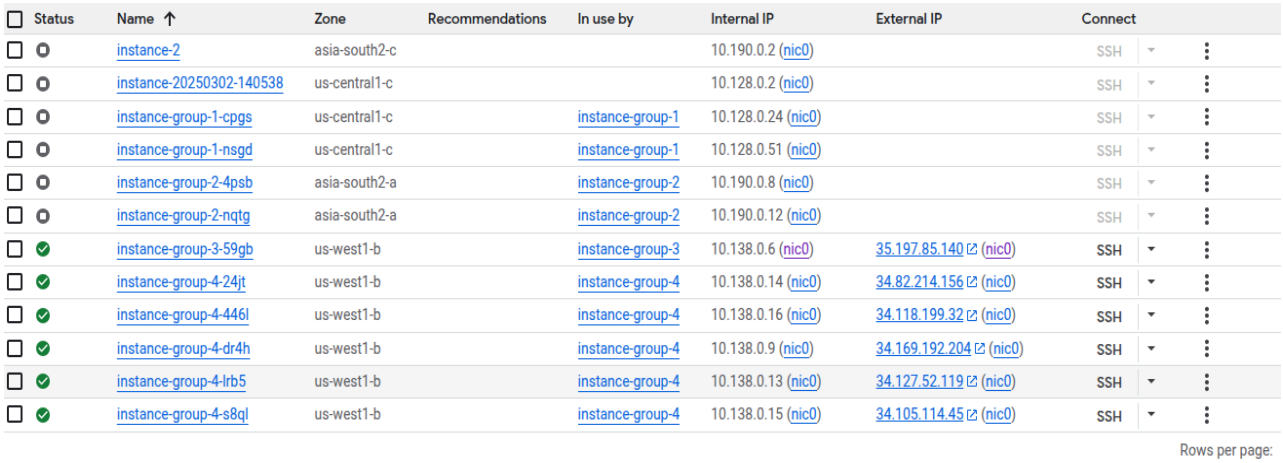
Load testing helps verify the effectiveness of auto-scaling policies and ensures that the infrastructure can handle high traffic scenarios without performance degradation.

I have performed only a cpu load test, other load tests like memory, network and disk can be performed.

Step 7: Verifying Autoscaling in GCP

- Check if the instance group is created: **gcloud compute instance-groups managed list**
- List instances within the group:
    - **gcloud compute instance-groups managed list-instances instance-group-4 --zone=us-west1-b**
- Connect to an instance using SSH:

    - **gcloud compute ssh <instance-name> --zone=us-west1-b**
- Monitor autoscaling activity:
    - **gcloud compute instance-groups managed describe instance-group-4 --zone=us-west1-b**



Status	Name ↑	Zone	Recommendations	In use by	Internal IP	External IP	Connect
	instance-2	asia-south2-c			10.190.0.2 (nic0)		SSH
	instance-20250302-140538	us-central1-c			10.128.0.2 (nic0)		SSH
	instance-group-1-cpgs	us-central1-c		instance-group-1	10.128.0.24 (nic0)		SSH
	instance-group-1-nsgd	us-central1-c		instance-group-1	10.128.0.51 (nic0)		SSH
	instance-group-2-4psb	asia-south2-a		instance-group-2	10.190.0.8 (nic0)		SSH
	instance-group-2-nqtg	asia-south2-a		instance-group-2	10.190.0.12 (nic0)		SSH
	instance-group-3-59gb	us-west1-b		instance-group-3	10.138.0.6 (nic0)	35.197.85.140 (nic0)	SSH
	instance-group-4-24jt	us-west1-b		instance-group-4	10.138.0.14 (nic0)	34.82.214.156 (nic0)	SSH
	instance-group-4-446l	us-west1-b		instance-group-4	10.138.0.16 (nic0)	34.118.199.32 (nic0)	SSH
	instance-group-4-dr4h	us-west1-b		instance-group-4	10.138.0.9 (nic0)	34.169.192.204 (nic0)	SSH
	instance-group-4-lrb5	us-west1-b		instance-group-4	10.138.0.13 (nic0)	34.127.52.119 (nic0)	SSH
	instance-group-4-s8ql	us-west1-b		instance-group-4	10.138.0.15 (nic0)	34.105.114.45 (nic0)	SSH

Rows per page:

Related actions

Step 8: Testing Deployment

1. Visit `http://<instance-external-ip>:5000` to check the application response.
2. Verify autoscaling by running the **stress** command and observing instance scaling behavior.

3. Conclusion

This implementation successfully demonstrates:

- Local VM-based CPU monitoring.
- Automated VM provisioning in GCP upon high CPU utilization.
- Deployment of a Flask-based application on auto-scaled instances.
- Firewall configuration for external accessibility.

By following these steps, we can efficiently scale resources dynamically based on workload demands, ensuring optimized cloud resource utilization.

You can also use tools like Prometheus, Grafana or node exporter for better graphics.

3. Architecture Design: