Task 1)

- refer my github repository
- Refer my deploy.sh file from the github repo
- The environment variables I used in the deploy.sh is
    - export EC2_USER=root
    - export IP=3.91.56.154
    - export SOURCE=/Users/imac/task
    - export DESTINATION=/var/www/html
- Save the following script on deploy.sh file
- Make the file executable by chmod +x deploy.sh
- Add the machine public key to server authorized_keys

```
#!/bin/bash
rsync -avz --exclude '.git/' $SOURCE $EC2_USER@$IP:$DESTINATION
```

Task 2)

The following terraform configuration 2 resource(EC2 and RDS database) on AWS

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_instance" "app_server" {
  ami           = "ami-080e1f13689e07408"
  instance_type = "t2.micro"

  tags = {
    Name = "terraform-server"
  }
}

resource "aws_db_instance" "myrds" {
```

```
  allocated_storage   = 1
  storage_type        = "gp2"
  identifier          = "smrds"
  engine              = "mysql"
  engine_version      = "8.0.27"
  instance_class      = "db.t2.micro"
  username            = "admin"
  password            = "Passw0rd!123"
  publicly_accessible = true
  skip_final_snapshot = true

  tags = {
    Name = "MyRDS"
  }
}
```

#after save the above file run the following commands

- **terraform init**: Initializes a Terraform working directory.
- **terraform validate**: Checks the Terraform configuration for syntax errors and internal consistency.
- **terraform plan**: Generates an execution plan without making any changes to the infrastructure.
- **terraform apply**: Applies the changes defined in the Terraform configuration to the infrastructure. We can add **-auto-approve** tag when we run the apply command if we already checked the plan

By using Terraform, we can ensure that our infrastructure is:

- **Consistent**: The same configuration can be applied multiple times, resulting in the same end state.
- **Scalable**: we can easily update the configurations to scale up or down based on requirements.
- **Reproducible**: The code can be shared and reused to replicate the same infrastructure setup elsewhere, such as different environments or regions.

To apply best practices for security and compliance in our infrastructure configuration

- Create specific IAM roles for each task to ensure that only the necessary permissions are granted.
- Do not share IAM roles with unauthorized users to prevent security risks.
- Keep IAM credentials out of the configuration code. Use environment variables or a secure secrets management service to handle IAM access.
- Be cautious with resource deletion roles. Avoid using roles that have permissions to delete resources unless absolutely necessary. Running the terraform destroy command will remove all resources defined in the Terraform configuration, which can be risky.
- Regularly review and update IAM policies to remove outdated permissions and ensure that the current policies align with the principle of least privilege.

Task 3)

I used gitlab CI/CD that has 4 stages

- Build
- Test
- Deploy

Build Stage

- The build stage builds a Docker image based on the Dockerfile in the repository (project).

Test Stage

- In the test stage, the following integrations have been included:

  Sonarqube Integration

  - Sonarqube has been integrated for static code analysis.

  Gitlab Included Templates Automation Testing

  The following Gitlab CI/CD templates have been included:

  - Security/SAST.gitlab-ci.yml: For Static Application Security Testing (SAST)
  - Jobs/SAST-IaC.gitlab-ci.yml: For SAST of Infrastructure as Code (IaC)
  - Security/Secret-Detection.gitlab-ci.yml: For detection of secrets in the codebase

Deploy Stage

- The Deploy stage includes two different deployment strategies:

  Deployment on Main Branch Change

  - When there is a change on the main branch, the code from the main branch is copied to the target server in the production environment.

  Deployment on Stage Branch Change

  - When there is a change on the stage branch, the previous running Docker container is removed, and a new container is run with the same name and IP using the new image.

```yaml
image: docker:latest
stages:
 - build
 - test
 - scan
 - deploy

variables:
 CONTAINER_TEST_IMAGE: $CI_REGISTRY_IMAGE:$CI_COMMIT_REF_SLUG
 CONTAINER_RELEASE_IMAGE: $CI_REGISTRY_IMAGE:latest

build:
 stage: build
 script:
   - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD $CI_REGISTRY
   - docker build -t $CONTAINER_RELEASE_IMAGE .
   - docker push $CONTAINER_RELEASE_IMAGE
 sonarqube-check:
 image:
   name: sonarsource/sonar-scanner-cli:latest
   entrypoint: [""]
 variables:
   SONAR_USER_HOME: "${CI_PROJECT_DIR}/.sonar"
   GIT_DEPTH: "0"
 cache:
   key: "${CI_JOB_NAME}"
   paths:
     - .sonar/cache
 script:
   - sonar-scanner
 allow_failure: true
 rules:
   - if: $CI_COMMIT_BRANCH == 'main'

include:
 - template: Security/SAST.gitlab-ci.yml
 - template: Jobs/SAST-IaC.gitlab-ci.yml
 - template: Security/Secret-Detection.gitlab-ci.yml

deploy-copyfile:
 stage: deploy
```

```
 tags:
  - prod
 only:
  - main
 before_script:
   - apk update && apk add openssh-client bash rsync
   - eval $(ssh-agent -s)
   - echo "$SSH_PRIVATE_KEY" | tr -d '\r' | ssh-add -

 script:
   - mkdir -p ~/.ssh
   - echo "${SSH_HOST_KEY1}" > ~/.ssh/known_hosts
   - echo -e "Host *\n\tStrictHostKeyChecking no\n\n" > ~/.ssh/config
   - rsync -vv -rz --checksum --delete . root@ip-address:/var/www/html/dashboard
 environment:
   name: prod
   url: https://main.test.com

run-docker:
 stage: deploy
 only:
  - stage
 script:
   - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD $CI_REGISTRY
   - docker pull $CONTAINER_RELEASE_IMAGE
   - docker container rm -f test || true
   - docker run -d -p 9010:9010 --name test --restart unless-stopped
$CONTAINER_RELEASE_IMAGE
 environment:
   name: stage
   url: https://stage.test.com
```

Task 4)

Ansible is a powerful IT automation tool that we can use to manage server configurations. It uses playbooks written in YAML to describe the desired state of our server's configuration, Here's a general steps of how we can use Ansible to manage server configuration:

1. **Install Ansible** on our control node (the machine from which we'll manage our servers).
2. **Modify an inventory file** to list the servers we want to manage
    - Using a Custom Inventory File: If you have a custom inventory file, you can specify it with the -i option:
        i. ansible-playbook -i /path/to/inventory playbook.yml
    - Using the Default Inventory File (/etc/ansible/hosts): If we specify our servers in the default inventory file located at /etc/ansible/hosts, we don't need to use the -i option. we can simply run:
        i. ansible-playbook playbook.yml
    - Defining Servers in a Group: In the inventory file, we can define groups by using brackets [] and listing the servers under the group name. Here's an example of how to define servers in a group:
      [dev]
      server1.name
      server2.name
      [production]
      server3.name
      server4.name
3. **Write a playbook** that defines the tasks to be performed on the servers.

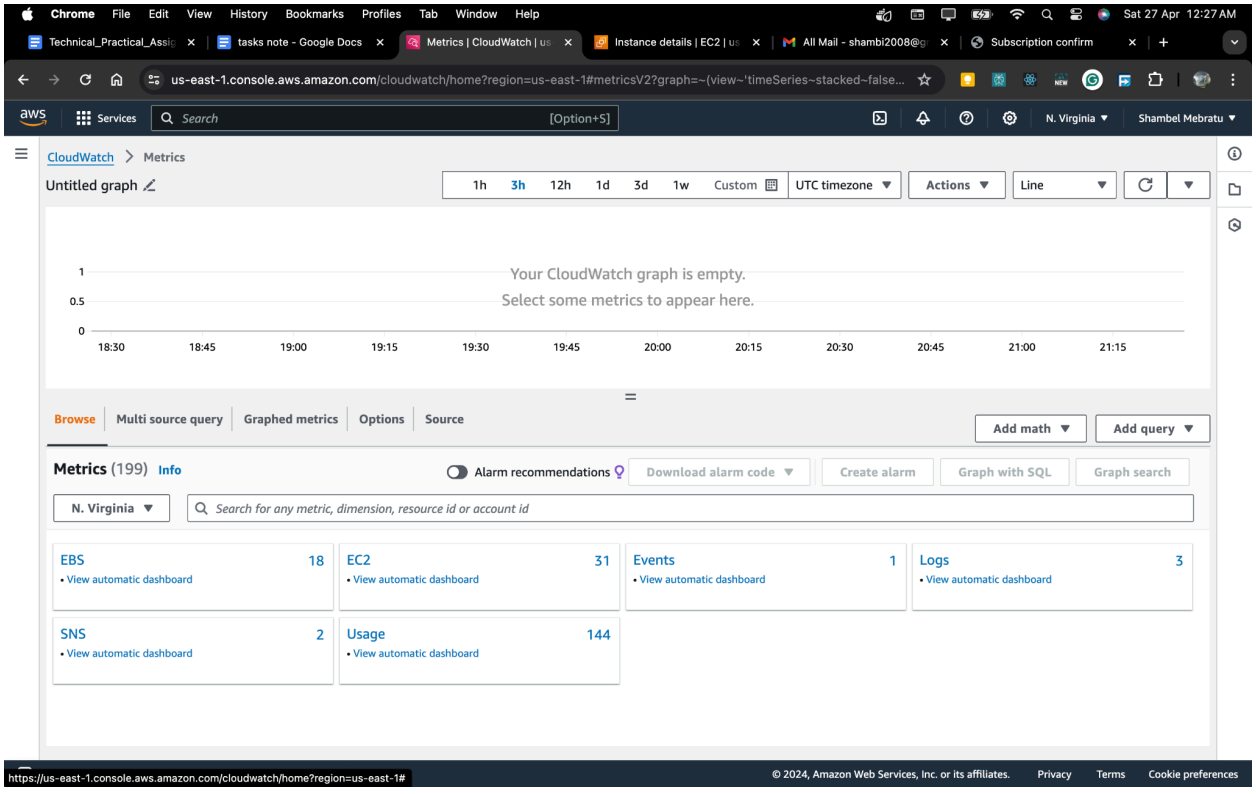
The following playbook will create file, delete file in dev servers and install nginx, start nginx service in production servers

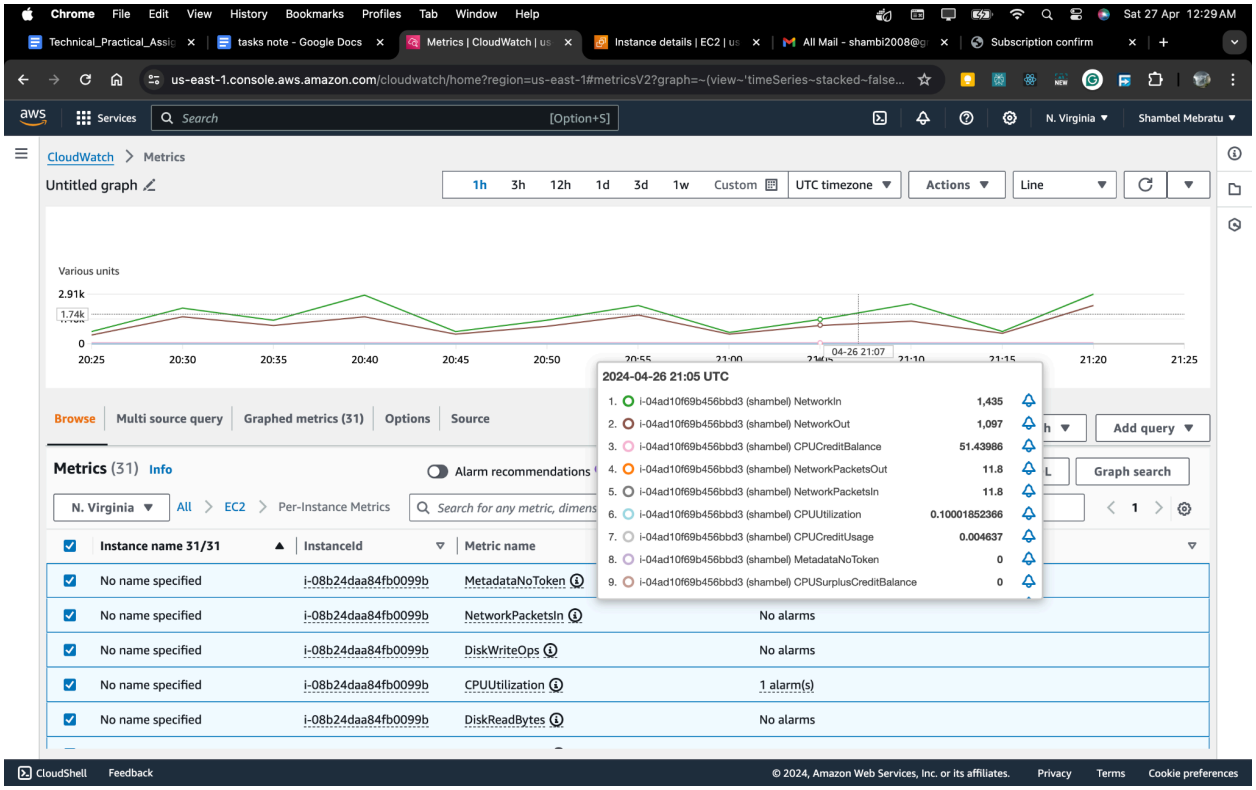
```yaml
---
- name: Create and delete a file
 hosts: dev
 become: yes
 tasks:
   - name: Create a file
     file:
       path: /tmp/file.txt
       state: touch
   - name: Add content to the file
     lineinfile:
       path: /tmp/file.txt
       line: "Hello, this is my sample file content."
   - name: Delete the file
     file:
       path: /tmp/file.txt
       state: absent
---
- name: Install Nginx
 hosts: production
 become: yes
 tasks:
   - name: Install Nginx
     apt:
       name: nginx
       state: present
   - name: Start Nginx service
     service:
       name: nginx
       state: started
```

Ansible ad-hoc commands that create and delete a file without using a playbook
ansible all -m file -a "path=~/tmp-file state=touch" ..... uses the file module with the -m flag to create a file named tmp-file in the home directory (~) of all hosts
ansible all -m file -a "path=~/tmp-file state=absent" ..... uses the file module to delete the tmp-file from the home directory of all hosts.The state=absent ensures that the file is removed

Task 5)

Amazon CloudWatch is the best monitoring tool for tracking metrics, logs, setting alarms for specific activities, and taking actions based on those alarms. It offers a clear and descriptive dashboard where all metrics, dashboards, and alarms can be accessed.
You can access Amazon CloudWatch within the CloudWatch service to view and manage all monitoring aspects centrally. Additionally, you can access specific service metrics directly from each service's page. For example, you can view the EC2 CloudWatch dashboard within the EC2 page by clicking on the instance name.





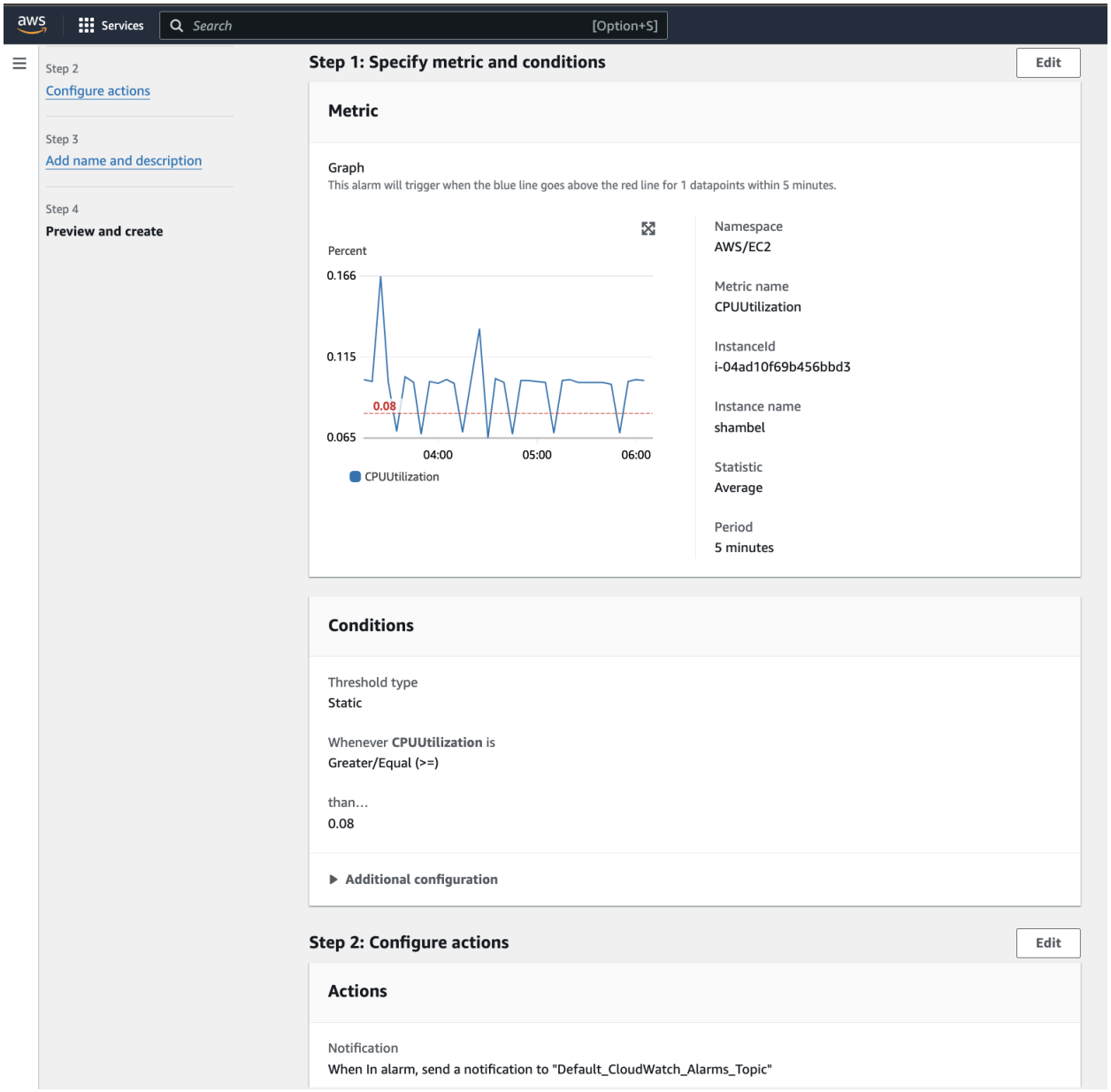
To set an alarm

CloudWatch Alarm for Instance ‘shambel’

An alarm has been configured in AWS CloudWatch to monitor the CPUUtilization metric of an EC2 instance named ‘shambel’. The metric represents the average CPU load, measured over 5-minute intervals.

The alarm condition is set to trigger when the CPU utilization (depicted as the blue line on the graph) exceeds the threshold (the red line) for one data point within a 5-minute period. This means that if the CPU load goes above the predefined limit for at least 5 minutes, an alarm state will be initiated.

This monitoring tool is crucial for maintaining the health and performance of the EC2 instance by ensuring that the CPU usage remains within acceptable parameters, thus preventing potential overloads.



Task 6)

I will share the document via Microsoft team for addisu.a@kachadfs.com and tsegaye.t@kachadfs.com