The objective of this project is to develop and showcase a fully automated CI/CD testing pipeline that ensures quality and stability of applications before deployment. The pipeline was designed to be flexible and able to be used with any application and testing framework making it reusable and scalable.

1. Key Features

- Robust automated testing workflow. This pipeline supports unit tests, integration tests and end-to-end tests ensuring wide testing coverage and code quality.

- Infrastructure as Code (IaC). The entire infrastructure, including AWS EC2 instances, VPCs, subnets, DynamoDB and others is provisioned and managed using Terraform.

- Containerized deployment. The application is packaged into a Docker container and deployed on an AWS EC2 instance for testing.

- Cloud-based log storage. Test results and logs are stored in AWS Cloudwatch, allowing users to monitor and analyze test outcomes efficiently.

- Flexible log management. The pipeline has a built-in functionality to create, append and delete the logs of a specific application being tested.

- Extensibility. For the purposes of this project - a Python-based API was built to showcase the pipeline. But it can be easily adapted for any application written in different programming languages or frameworks.

This project presents a practical implementation of automated testing utilizing a CI/CD pipeline, ensuring the applications meet quality standards before deployment and saving the time of developers, testers and DevOps engineers alike.

2. Prerequisites

This pipeline is designed to be application-agnostic therefore it can be used to test any application, regardless of its technology stack or testing framework. However - to successfully run the pipeline - several tools and configurations are required. The following section outlines the necessary software and tools but does not cover the specifics of the application or test suite used. For the purposes of this project a simple Python-based API and associated tests were built as a demonstration.

To set up and run the pipeline - ensure you have the following installed and configured:

- Terraform (> 1.0). Infrastructure provisioning to deploy cloud resources.
  For download/installation instructions refer to: Install Terraform | Terraform | HashiCorp Developer

- AWS CLI (>2.x). Authentication and management of AWS resources.
  For download/installation instructions refer to: Installing or updating to the latest version of the AWS CLI - AWS Command Line Interface

- Ansible (>2.10). Automating configuration and deployment.
  For download/installation instructions refer to: Installing Ansible — Ansible Community Documentation

- Docker (>24.x). Containerizing the application for deployment.
  For download/installation instructions refer to: Install | Docker Docs

- GitHub Actions. Running CI/CD workflows.
  For download/installation instructions refer to: Writing workflows - GitHub Docs

- Cloudwatch Logs. Collecting and reviewing test logs.
  For usage instructions refer to: What is Amazon CloudWatch Logs? - Amazon CloudWatch Logs

- IAM credentials. AWS access keys with sufficient permissions.
  For usage instructions refer to: Manage access keys for IAM users - AWS Identity and Access Management

Since the pipeline provisions infrastructure on AWS - make sure you have:

- An AWS account with permissions to create EC2, VPC, Subnets, Security Groups, IAM roles and CloudWatch logs.
- The AWS CLI configured with proper IAM credentials.

While the pipeline is application-agnostic and flexible, users must provide:

- The application code
- A Dockerfile of the application
- Any required configuration files (e.g., environment variables, dependencies, etc)

Without these the pipeline cannot deploy or test the application properly!

3. **Deployment and usage**

   The whole testing pipeline consists of four separate pipelines:

- Infrastructure setup in AWS cloud via Terraform.
- Deploying Docker on the provisioned EC2 instance.
- Building and deploying the target application in a container in the EC2 instance.
- Running the selected tests on the running application.

Each pipeline (besides other required files) has an Ansible playbook and a GitHub Workflows file (except in the case of Terraform that has *<file_name>.tf* files). This section will

explain what each pipeline does in detail and how to use them properly. Below is the file structure of the project:

Infrastructure provisioning with Terraform

As mentioned before - the AWS infrastructure used for the purposes of the project is provisioned using Terraform. The folder *AWS_Infra* contains the required files: *main.tf* and *variables.tf*. The *main.tf* file describes all the infrastructure components that will be provisioned and includes:

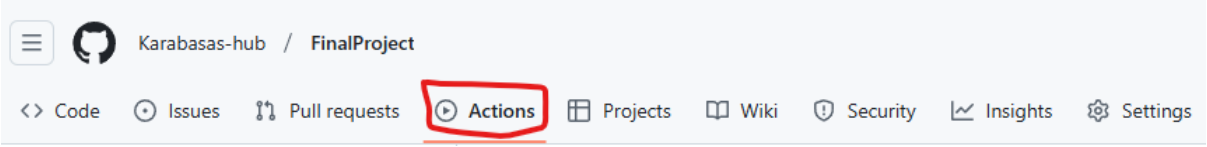
- VPC - a dedicated virtual network for the infrastructure
- Subnets for resource segmentation
- Internet Gateway (IGW) that enables outbound internet access
- Security groups configured to allow necessary traffic ingress

- EC2 instance (VM) to run the application
- IAM roles and policies to grant permissions to required AWS services
- CloudWatch Logs to collect logs from testing sessions
- DynamoDB table to store application data and tfstate lock
- S3 bucket to store the tfstate itself

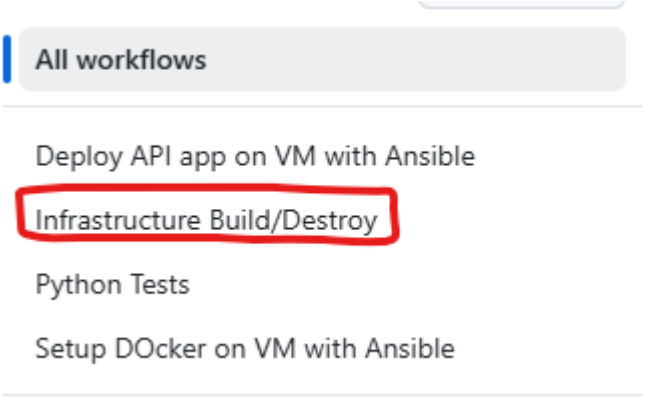
Prior to running the pipeline - it is required to store your AWS Access credentials and region in GitHub Actions Secrets.

The pipeline creates an S3 bucket and DynamoDB table to store the Terraform tfstate and a enable state locking to prevent concurring changes to take place while also enabling the user to not store the tfstate locally. Then all the resources listed above are initialised and provisioned.

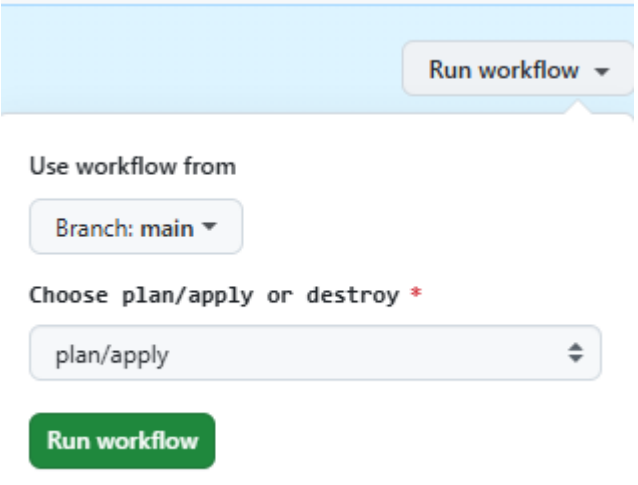
To run the pipeline - navigate to GitHub Actions:



Select the *Infrastructure Build/Destroy* workflow:



Above the history of workflow runs notice a choice drop-down:



Select *plan/apply* to provision all the infrastructure and *destroy* to destroy it.

After making the selection - wait for the runner to provision/destroy the infrastructure and view the logs.

After the pipeline is successful - notice an IP address at the bottom of the logs just before *Post job cleanup*.

```
Outputs:
instance_ip = "3.125.6.72"
```

This is the IP address of the VM that was just provisioned. It is required by the following pipelines, therefore - remember it. Note that your IP address will differ from the one in the example.

The AWS infrastructure has been provisioned. You can check all the resources via the AWS Console. Next step - deploying Docker on the provisioned VM.

Deploying Docker on the provisioned EC2 instance (VM)

The folder *Ansible_playbooks* contains two playbooks - one for deploying the application and another one for deploying Docker. In this section we will focus on the latter. The playbook performs the following actions:

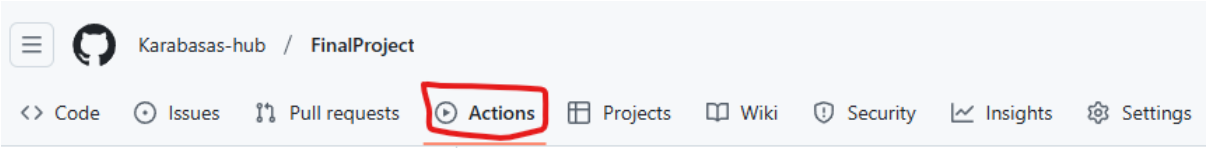
- Installs required packages
- Adds Docker GPG key and repository
- Installs Docker and adds the user to the Docker group
- Verifies Docker installation
- Passes your AWS credentials from GitHub Actions Secrets to your container

The workflow file for installing Docker is *Docker_deploy.yml*. Prior to running the pipeline - you will need to create and SSH key pair in AWS and store the Private SSH key in GitHub Actions Secrets as well.

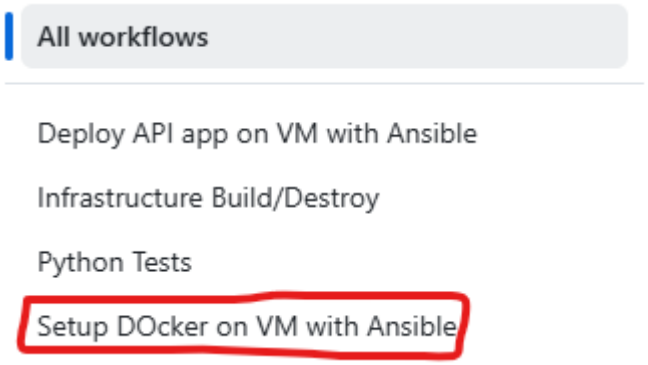
This pipeline will perform the following actions:

- Create an SSH directory and add the host (the IP address of your VM) to known hosts
- Check and install Ansible
- Ensure everything is up to date
- Set up an SSH key for authentication with the VM and set the AWS credentials
- SSH into the VM and run the Ansible playbook

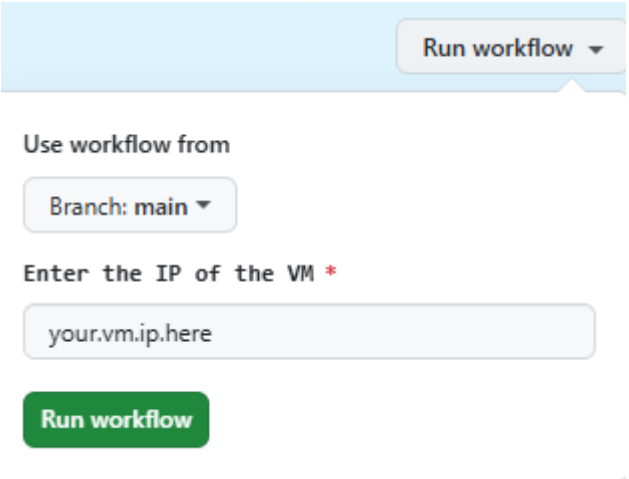
To run the pipeline - once again navigate to the *Actions* tab:



Select the *Setup Docker on VM with Ansible* option:



Above the history of workflow runs you will notice another drop-down. This is where you need to enter the IP address of your VM:



Click *Run workflow* and view the logs. Once the pipeline is finished - Docker is successfully installed and running on the VM.

Building and deploying the application on the VM

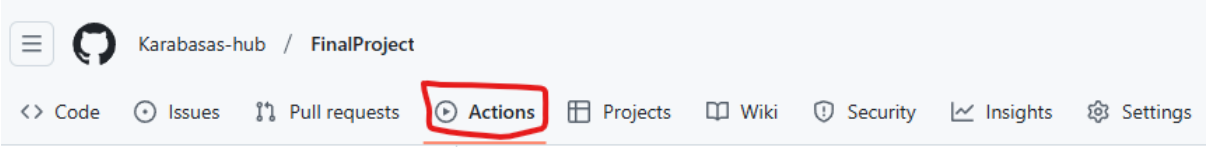
In the folder *Ansible_playbooks* there exists an Ansible playbook named *API_on_docker_on_vm.yml*. That is the playbook to build and deploy the application on the VM in a container. The playbook performs the following actions:

- Starts the Docker service in case it has seized
- Copies the application file, the Dockerfile for the application and the requirements file onto the VM
- Builds the Docker image of the application in the VM
- Stops and removes any pre-existing containers that are running the application
- Starts the container with the application and maps the container port 5000 with port 5000 of the VM

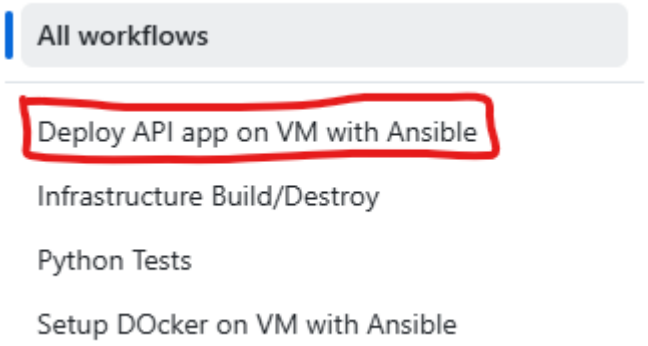
The workflow file works very similarly to the workflow file do deploy Docker since all it really has to do is ensure Ansible is installed and run the playbook. Below are all the actions performed by this pipeline:

- Create an SSH directory and add the host (the IP address of your VM) to known hosts
- Check and install Ansible
- Ensure everything is up to date
- Set up an SSH key for authentication with the VM and set the AWS credentials
- SSH into the VM and run the Ansible playbook

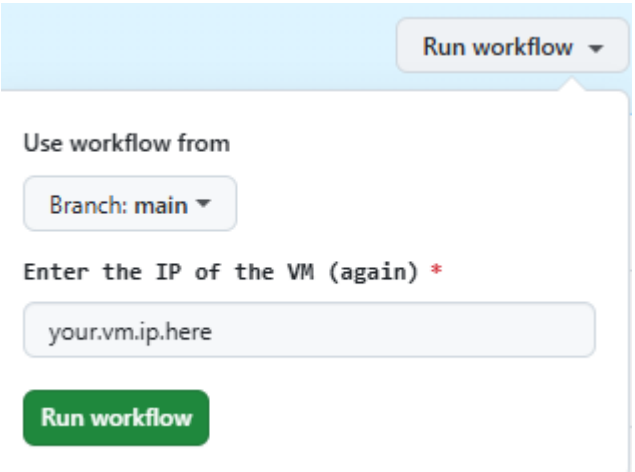
Notice that both workflows work pretty much identically. For running the pipeline - the procedure remains the same. Once again navigate to the *Actions* tab in GitHub:



This time select *Deploy API app on VM with Ansible* workflow:



Find the drop-down menu above the workflow history and enter the IP of the VM:



Click *Run workflow*.

Wait for the pipeline to finish the job and view the logs. After the job is complete - the application is running inside a container inside the VM.

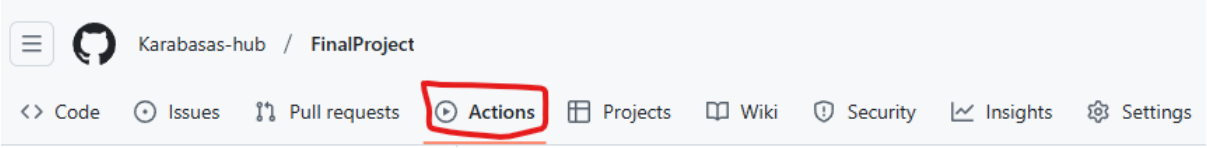
Running tests on the deployed application

At this point the infrastructure is provisioned, Docker and the application deployed on the VM and it is time to run the tests. Since the project called for a robust testing pipeline - unit, integration and end-to-end tests are performed. For the sake of simplicity - two tests of each kind are performed.

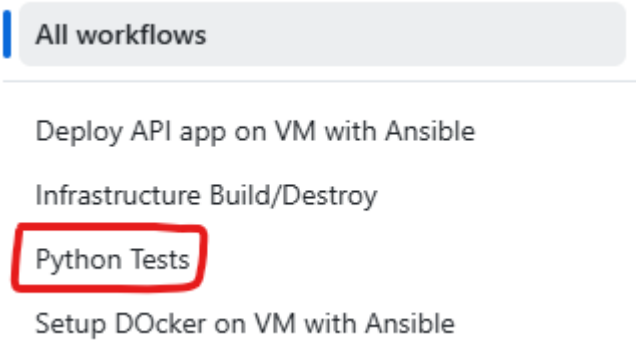
In the folder *Tests* are the test files used for this project. The workflow file for running the tests is located alongside the other workflow files in *.github/workflows* folder and is called *Testing.yml*. In essence it performs the following actions:

- Sets up Python and installs dependencies
- Sets a BASE_URL variable that is built like: *http://<vm-ip>:5000"*
- Runs selected tests
- Formats collected logs to pass them to CloudWatch
- Gives CloudWatch your AWS credentials to manage logs
- Based on user selection creates, appends or deletes logs and log groups

To run the testing workflow - navigate to the *Actions* tab:



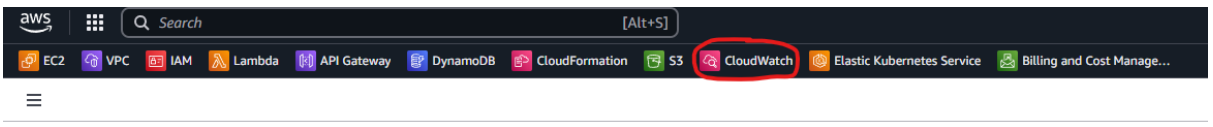
Select the *Python tests* pipeline:



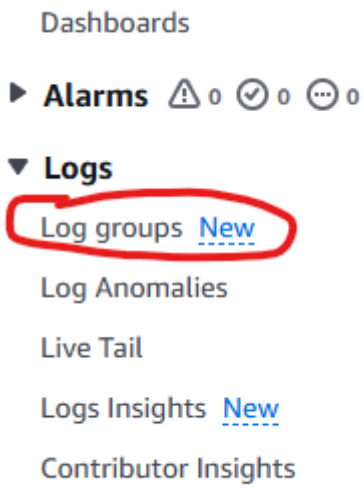
Locate the vastly different drop-down menu above the workflow run history:

You will need to enter the IP of the VM, select wether to create, keep or delete the logs and list the tests you want to run. Note that the tests have to be provided **by names, separated with commas without file extensions** - e.g. "*test_unit_1, test_unit_2".*

Press *Run workflow* and it will run all your selected tests. After running them - the logs will be displayed in the workflow window but also collected and stored in AWS Cloudwatch. So in order to view and analyze the logs - navigate to AWS CloudWatch:



Select the log group:



Locate your created log group, click on it and you will find your stored logs.

Troubleshooting and common FAQs

In this section we will briefly go over the most common issues that can arise while setting up and using this pipeline.

Q1: Terraform workflow fails with AWS authentication error

Possible causes: Incorrect AWS credentials, missing IAM permissions for the GitHub Actions runner.

Possible fixes:

- Check that the AWS credentials in GitHub Actions secrets are set correctly
- Ensure the IAM role has permissions for Terraform actions
- Run:

```
aws sts get-caller-identity
```

in the terminal to verify authentication.

Q2: Docker installation workflow fails due to SSH issues

Possible causes: The VM's IP address is incorrect or unreachable, the SSH private key is missing from GitHub secrets.

Possible fixes:

- Verify the VM's IP in the AWS EC2 dashboard
- Check SSH connectivity:

```
ssh -i "mock_ssh.pem"\
ubuntu@ec2-<IP-of-VM>.eu-central-1.compute.amazonaws.com
```

notice that when trying to SSH into the VM - you are using the DNS of the VM therefore the IP address of the VM has to be separated by dashes and not dots.

- Ensure that the correct private key is stored in GitHub Secrets

Q3: The application is running, but I can't access it in the browser

Possible causes: The application is not exposed on the correct port, AWS security groups are blocking inbound traffic.

Possible fixes:

- SSH into the VM as shown above and check if the docker container with the application is running and what port is it listening on:

  ```
  docker ps
  ```

- Check the logs of the Docker container:

  ```
  docker logs <container_id>
  ```

- Ensure the AWS security group allows traffic on the application's port