Secure Infrastructure Automation with CI/CD Deployment of a Dockerized Web App on AWS EC2[1]

Twesha Thakur
Department of Computer Science and Engineering
Lovely Professional University
Phagwara, Punjab 144411
tweshathakur@gmail.com

***Abstract* - This project aims to demonstrate the application of DevSecOps principles by deploying a containerized web application using CI/CD automation. Infrastructure is provisioned using Terraform on AWS EC2, and deployment is triggered via GitHub Actions upon code changes. The project showcases secure key handling, environment isolation using Docker, and reliable deployment of a lightweight to-do list application served over a public EC2 instance.**

Index Terms - DevSecOps, CI/CD, Terraform, Docker, GitHub Actions, AWS EC2

I. Introduction

In this project, we design and implement an automated DevSecOps pipeline that provisions infrastructure using Terraform and deploys a Dockerized frontend web application through GitHub Actions CI/CD workflows. The web application is a simple to-do list built with HTML/CSS, containerized with Docker, and served on port 3000. The infrastructure is deployed on an Amazon EC2 instance running Amazon Linux, and private key-based authentication is securely managed. The entire deployment workflow is triggered upon GitHub pushes, demonstrating a complete infrastructure-as-code and DevOps cycle.

II. Tech Used

1) *HTML/CSS* – For creating a simple, static to-do web interface
2) *Docker* – Containerization of the frontend application
3) *Terraform* – Provisioning AWS EC2 infrastructure as code
4) *Git & GitHub* – Version control and codebase hosting
5) *GitHub Actions* – CI/CD automation triggered on push
6) *Amazon EC2* – Hosting environment for deployed Docker app
7) *SSH & SCP* – Secure remote access and deployment via GitHub pipeline

III. Architecture

The architecture of this DevSecOps project consists of three core components: the application layer, infrastructure provisioning, and the continuous integration/continuous deployment (CI/CD) pipeline. Each component is modular and interacts seamlessly to ensure a fully automated, secure deployment process.

A. Application Layer

The application is a static to-do list built using HTML and CSS. It is containerized using Docker and served using a minimal HTTP server mapped to port 3000. This container is built locally and later deployed to an EC2 instance through automation.

B. Infrastructure *Provisioning*

Infrastructure is provisioned using Terraform, adhering to Infrastructure as Code (IaC) principles. The configuration includes:

i. Creation of an AWS EC2 instance with the Amazon Linux 2 AMI.
ii. Security Groups that allow traffic over port 22 (SSH) and port 3000 (App access).
iii. Automatic installation of Docker via a user_data.sh script.
iv. Deployment of a public SSH key to the EC2 instance, allowing secure access via a corresponding private key stored as a GitHub Secret.

Terraform stores state locally and ensures idempotent provisioning — meaning repeated executions produce the same result without duplicating resources.

C. CI/CD Pipeline

The CI/CD pipeline is managed using GitHub Actions. It monitors the main branch of the GitHub repository for changes. Upon detecting a new push, the workflow performs the following steps:

i. Code Checkout: Clones the repository inside the GitHub runner.
ii. File Transfer: Uses the appleboy/scp-action to copy the updated To-do-List/ folder to the EC2 instance using SCP over SSH.
iii. Deployment: SSHs into the EC2 instance and:
    - Stops the running Docker container (if any)

- Builds a new Docker image from the transferred files
- Starts a fresh container using the updated image

This entire process is secured using GitHub Secrets, which store:

i. The EC2 instance's public IP (EC2_HOST)
ii. The username (EC2_USER)
iii. The corresponding private key (EC2_KEY)

D. *Integration Flow*

Each component is decoupled but integrated in a streamlined sequence:

i. Terraform ensures the EC2 infrastructure is consistent and secure.
ii. GitHub Actions ensures the latest version of the application is always deployed.
iii. Docker provides a consistent runtime environment, both locally and on the EC2 host.

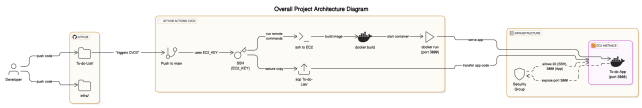

Fig. 1 Architecture and Workflow

IV. Development Workflow

A. *Local Development*

The frontend to-do list web application is built using simple HTML and CSS. It is containerized locally using Docker, allowing the application to run consistently across different environments, regardless of the host operating system. The following commands are used for containerization and local testing:

i. docker build -t todo-image .
   This command builds a Docker image named todo-image using the Dockerfile located inside the To-do-List/ folder. The image encapsulates the application and its serving configuration.
ii. docker run -p 3000:80 todo-image
   This command runs the Docker container in the background and maps port 80 inside the container to port 3000 on the local host. This allows the application to be accessed via http://localhost:3000 during development.

This local setup ensures that the application behaves the same in both local and production environments, fulfilling a key DevOps principle: "build once, run anywhere."

B. *Infrastructure Provisioning*

To host the application on a cloud platform, Terraform is used to provision an EC2 instance on Amazon Web Services (AWS). Terraform enables the creation, update, and destruction of infrastructure using declarative configuration files.

The infrastructure setup is initiated using:

i. terraform init
   This command initializes the Terraform working directory. It downloads the necessary provider plugins (in this case, AWS) and prepares the backend to store the Terraform state.
ii. terraform apply
   This command applies the configuration defined in the .tf files. It provisions the EC2 instance, configures security groups, installs Docker via a user-data script, and embeds the public SSH key for remote access.

The public SSH key is passed to AWS to allow secure access to the instance, while the corresponding private key is stored securely in GitHub as a secret (EC2_KEY). This sets the foundation for a secure and automated deployment pipeline via GitHub Actions.

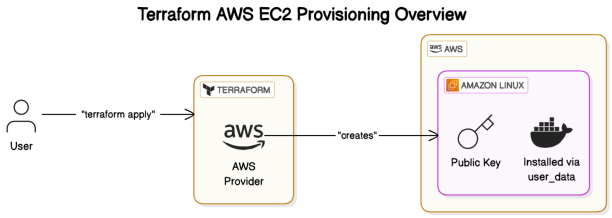

Fig. 2 Infrastructure Provisioning

V. CI/CD PIPELINE

The continuous integration and continuous deployment (CI/CD) pipeline is implemented using **GitHub Actions**, which automates the end-to-end deployment process for the Dockerized web application. The pipeline is triggered automatically upon any code push to the main branch of the repository, ensuring that the most recent changes are always reflected in the deployed application.

The GitHub Actions workflow begins by checking out the latest code and building the Docker image from the contents of the To-do-List/ folder. This image contains the complete application and its runtime configuration. The pipeline then securely transfers the updated source files to the EC2 instance using the appleboy/scp-action, which uses SCP over SSH. Following this, the pipeline executes commands remotely on the EC2 instance via appleboy/ssh-action. These commands stop any previously running container, rebuild the Docker image on the EC2 instance, and run a new container to serve the latest version of the application.

To enable secure, non-interactive deployment, three critical secrets are stored in the GitHub repository: EC2_HOST (the public IP address of the EC2 instance), EC2_USER (the SSH username, typically ec2-user for Amazon Linux), and EC2_KEY (the private SSH key corresponding to the public key embedded via Terraform). These secrets are used by the GitHub Actions runner to authenticate and securely communicate with the remote instance.

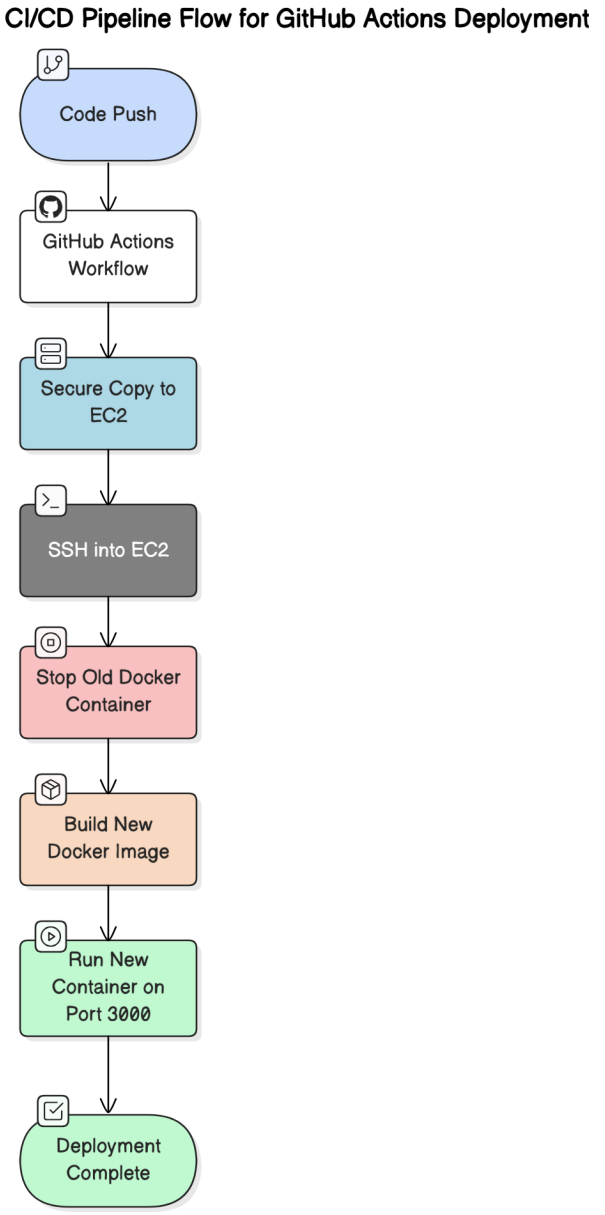

Fig. 3 CI/CD Pipeline Flowchart

VI. Hosting and Infrastructure

The hosting infrastructure is provisioned using Terraform, which automates the creation and configuration of a virtual server on Amazon EC2. The Terraform script defines a lightweight, general-purpose EC2 instance with an open port 3000, allowing the Dockerized web application to be accessed publicly over HTTP. The instance is also configured with a security group that allows SSH access (port 22) for administrative control.

Terraform uses a user_data shell script during instance provisioning to install Docker on the EC2 machine automatically. This ensures that the server is container-ready immediately upon boot. Additionally, a public SSH key is injected into the EC2 instance's authorized keys, which aligns with the corresponding private key stored as a GitHub secret. This key-based authentication forms the backbone of the secure deployment process.

Once provisioned, the EC2 instance hosts the Dockerized to-do web application, which becomes accessible over the internet at http://<EC2-PUBLIC-IP>:3000. The use of Docker and Terraform together creates a fully reproducible, isolated, and scalable environment suitable for both development and production workloads.

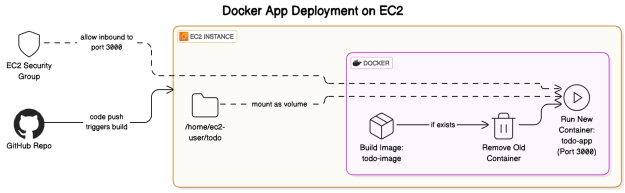

Fig. 4 Deployment on EC2 using Docker

VII. Future Scope

The current implementation can be extended to include a backend component that supports dynamic data handling through APIs. Automating such backend deployments would provide a more complete full-stack deployment pipeline.

Integrating SSL certificates using services like Let's Encrypt or an AWS Application Load Balancer (ALB) would enhance security by enabling HTTPS access to the application.

Mapping the deployment to an Elastic IP and linking it with a custom domain name would improve accessibility and make the setup more production-ready.

Container orchestration can be further refined using Docker Compose, enabling multi-container deployments with services like databases or APIs.

Additionally, monitoring tools such as Amazon CloudWatch or Prometheus can be integrated to track application performance, container health, and resource usage for proactive maintenance and reliability.

VIII. Conclusion

This project successfully demonstrates a practical and efficient DevSecOps pipeline by integrating Terraform for infrastructure provisioning, Docker for application containerization, and GitHub Actions for CI/CD automation. It reflects how modern DevOps practices can reduce manual effort, enforce reproducibility, and ensure secure deployment workflows. The modular design also makes it easy to extend and scale as needed, aligning with industry standards for cloud-native deployments.

REFERENCES

[1] Terraform Docs: *https://developer.hashicorp.com/terraform/docs*
[2] GitHub Actions Docs*:* *https://docs.github.com/actions*
[3] Docker Docs*:* *https://docs.docker.com*
[4] AWS EC2 Docs: *https://docs.aws.amazon.com/ec2/*