5/2/2025

Thrive Technical Test

Muhammad Rehman Zafar

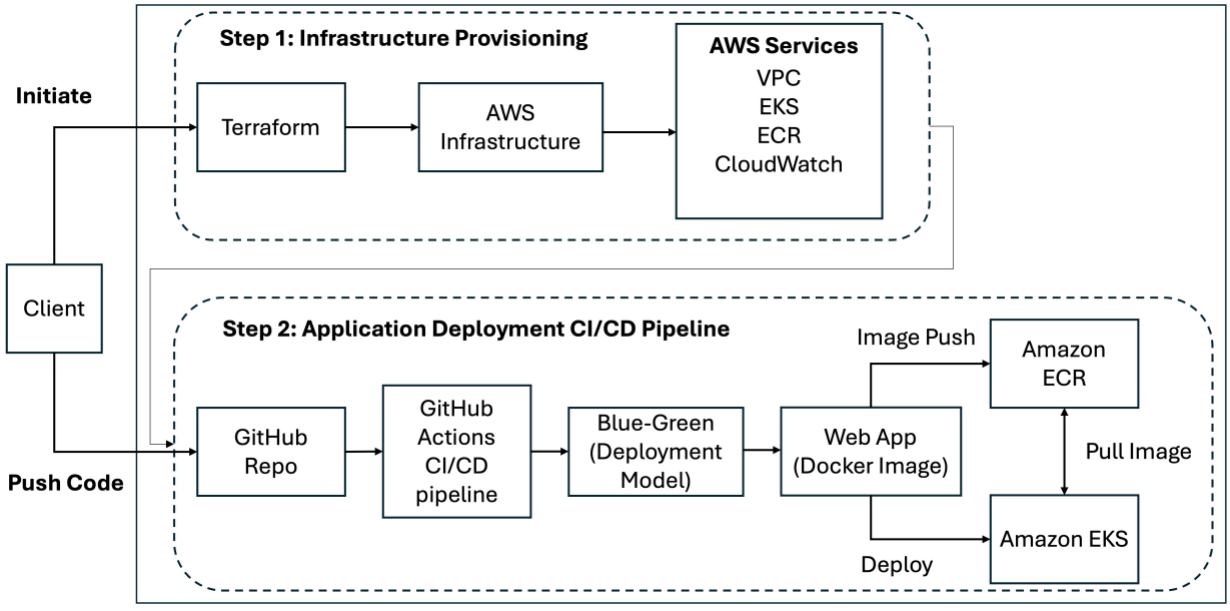
System Architecture Design:

The architecture diagram outlines a comprehensive cloud-native deployment workflow that is divided into two key stages: infrastructure provisioning and application deployment using CI/CD.

In the first stage, I used Terraform, an Infrastructure as Code (IaC) tool, to provision AWS infrastructure resources such as a Virtual Private Cloud (VPC), Elastic Kubernetes Service (EKS) for container orchestration, Elastic Container Registry (ECR) for storing Docker images, and CloudWatch for logging and monitoring.

Once the infrastructure is in place, the second stage of application deployment is triggered by pushing application code to a GitHub repository. This commit activates a GitHub Actions CI/CD pipeline that builds a Docker image of the application, pushes it to Amazon ECR, and then deploys it to the EKS cluster using a blue-green deployment strategy, which allows for zero-downtime upgrades by routing traffic between two environments.

The EKS cluster automatically pulls the Docker image from ECR and runs it in Kubernetes pods. This automated pipeline ensures a secure, repeatable, and scalable deployment process, with infrastructure and application lifecycles fully integrated and managed through code and version control.



Step 1: Infrastructure Provisioning

Set up the necessary AWS infrastructure using Infrastructure as Code (IaC).

- Client triggers Terraform execution.
- Terraform provisions the infrastructure in AWS.

- AWS Infrastructure includes:
    - VPC: A logically isolated network environment.
    - EKS: Managed Kubernetes service to run containers.
    - ECR: AWS-managed Docker container registry to store images.
    - CloudWatch: For monitoring and logging services.

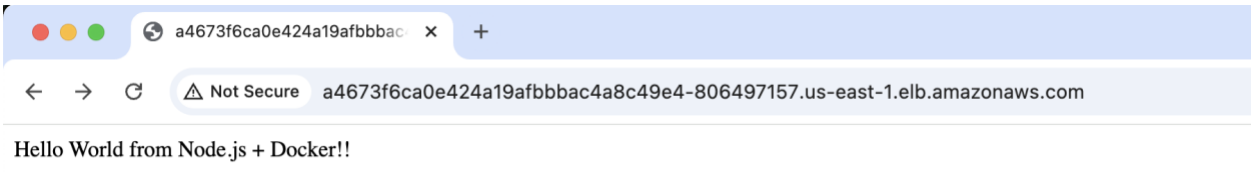
Step 2: Application Deployment via CI/CD Pipeline

Automate building, pushing, and deploying Docker images.

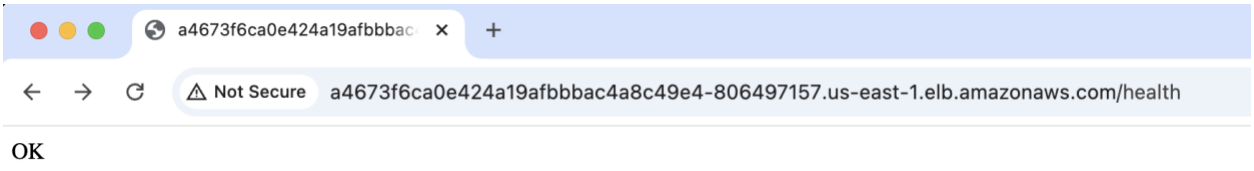
- Client pushes application code to GitHub Repo.
- This triggers GitHub Actions CI/CD pipeline:
    - Builds the Docker image
    - Pushes the image to Amazon ECR
    - Triggers a Blue-Green Deployment strategy (zero-downtime switch between old and new versions)
- Web App (Docker Image):
    - Gets built and pushed to Amazon ECR
    - Gets deployed to Amazon EKS, which pulls the image from ECR
    - Runs in Kubernetes Pods managed by EKS

Output:

Home Page

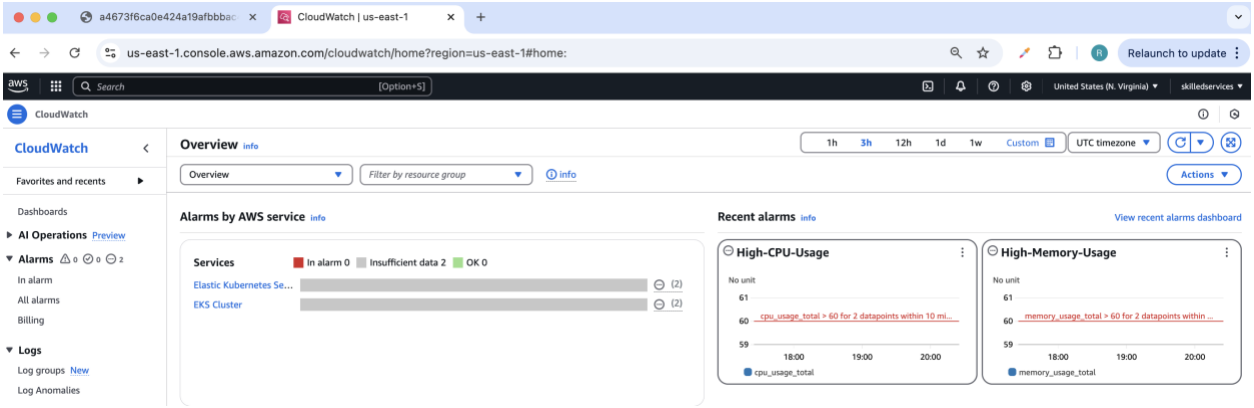


Health



Version (Blue-Green strategy)

Alerts:



Comments:

- I primarily defined all modules and resources in main.tf to quickly deliver a minimum viable product (MVP). However, for improved readability and maintainability, the configuration should be modularized into separate files (e.g., vpc.tf, eks.tf, iam.tf, etc.).
- I opted to use Amazon EKS for Kubernetes orchestration, which may incur higher costs compared to a simpler EC2 + Docker setup. The trade-off was made to take advantage of EKS's managed scalability and native integrations.
- Sensitive information such as the AWS account ID, access keys, and secret values were stored in GitHub Secrets. This approach improves security by avoiding hardcoded credentials in the codebase. While this adds management overhead, it can be further streamlined using GitHub OIDC integration with AWS IAM roles.
- CloudWatch alarms were configured to monitor CPU and memory usage. Alerts are triggered when usage exceeds 60%, and notifications are sent to the specified email address via Amazon SNS.