Report: CI/CD Pipeline for Deploying Microservices on EC2 Using Docker and GitHub Actions

Camilo Bueno - A00219928

Juan Pablo Uribe - A00376657

Summary

A complete CI/CD solution was built to deploy a microservices architecture on an EC2 instance using container images hosted on GitHub Container Registry (GHCR). The system is divided into two key pipelines: one for publishing images and another for automated deployment.

Work Methodology - Scrum

Scrum is an agile methodology used to manage complex projects, focusing on incremental deliverables, constant collaboration and continuous improvement.

Benefits of using Scrum in this project

Adaptability to frequent changes

- In an environment with multiple microservices and continuous deployments, requirements can change rapidly. Scrum facilitates constant adaptation through short iterations (sprints).

Incremental delivery of value

- At the end of each sprint, a functional increment of the system is delivered, allowing frequent progress to be shown and bugs to be caught before they accumulate.

Continuous visibility and collaboration

- Scrum ceremonies (such as Daily Scrums and Sprint Reviews) promote continuous communication among team members, which is key to coordinating the development of mutually dependent components.

Focus on continuous improvement

- Through retrospectives, the team reflects on what worked and what didn't, thus improving the process in each iteration.

Facilitates integration with DevOps and GitOps practices

- Scrum fits seamlessly with deployment, test and infrastructure automation, allowing the team to focus on delivering value while maintaining quality.

Branching Develop Strategy - Simplified Git Flow

This strategy organizes collaborative work in Git using a clear and predictable branching model. It is based on three main types of branches:

main: This is the main branch that contains the stable, production code. It is only updated when a new version has been fully tested and is ready to be deployed.

develop: Serves as the integration branch for development. It groups the changes of all new features before they are integrated into production. This is where joint developments are tested before being released.

feature/{name}: These are branches created from develop to work in isolation on specific new features. Once completed and tested, they are merged back into develop.

This model improves development organization, allows pre-production testing and facilitates teamwork without conflicts in the main code.

Operations Strategy - Infrastructure as Code + GitOps

This model applies GitOps principles to manage the infrastructure through versioned and automated code, ensuring consistency, traceability and control.

infra/main: Contains the stable configuration of the infrastructure that has already been deployed in production. It is a true reflection of the real environment.

infra/dev: Used to test infrastructure changes in development environments. New configurations are validated here before being promoted to production.

infra/feature/{name}: These are temporary branches created from infra/dev to experiment with new resources or adjust specific configurations (such as pipelines, clusters, or networks). Once approved, they are merged to infra/dev or directly to infra/main if necessary.

This strategy allows good development practices to be applied to the infrastructure, making every change reviewable, reversible and automatic through CI/CD tools.

Pipeline 1: Docker Image Publishing to GHCR (Publish to Docker)

This pipeline is triggered by pull requests to the `main` branch. It includes authentication with GHCR and image versioning using `latest` and `sha` tags.

Dockerized Services:

- **auth-api**
- **frontend**
- **log-message-processor**
- **todos-api**
- **users-api**

Security: Authentication is handled via `${{ secrets.GITHUB_TOKEN }}` to ensure private access to GHCR.

Pipeline 2: Deployment to EC2 (Deploy to EC2)

This second workflow is automatically triggered when the "Publish to Docker" workflow completes successfully (via the `workflow_run` event with successful conclusion).

Key Steps:

File Transfer:
The `scp-action` is used to copy the `docker-compose.yml` file to the EC2 instance.

Remote Execution:
Using `ssh-action`, the workflow logs in to GHCR, pulls the latest images, and recreates containers with `docker-compose up -d --force-recreate`.

Architecture Diagram

1. AWS Cloud

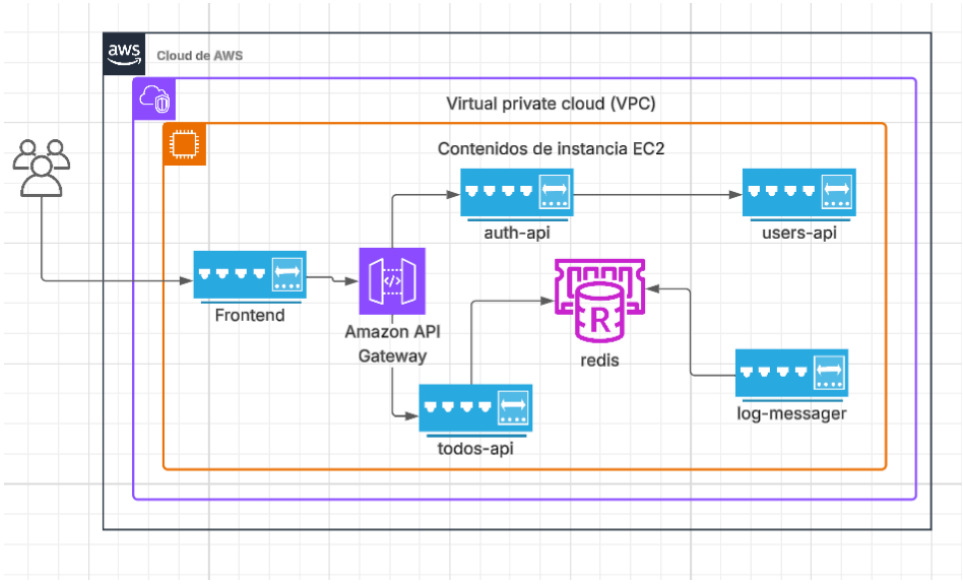
   - The entire environment is hosted in the Amazon Web Services (AWS) cloud, providing scalability and high availability.

2. Virtual Private Cloud (VPC)

   - A virtual private network where all system resources are isolated, providing security and internal traffic control between services.

3. EC2 Instance

- This is the virtual server where microservices are executed. This is where Docker and docker-compose are used to orchestrate the containers.



This architecture was created using an AWSTemplateFormat.yaml file. This file is in charge of defining the infrastructure (IaC) and must be executed through the AWS Cloudformation service. It allows you to execute .yaml files to automatically create the infrastructure.

Result:

The remote EC2 environment always runs the latest versions of the defined services.

Service Definition in Docker Compose

```
services:

  users-api:
    image: ghcr.io/solosoyjuan/users-api:latest

  auth-api:
    image: ghcr.io/solosoyjuan/auth-api:latest

  todos-api:
    image: ghcr.io/solosoyjuan/todos-api:latest

  frontend:
    image: ghcr.io/solosoyjuan/frontend:latest

  log-message-processor:
```

image: ghcr.io/solosoyjuan/log-message-processor:latest

All services are interconnected via the Docker network `msa-network`. Additionally, Redis is included as a shared dependency between `todos-api` and `log-message-processor`.

Cloud Design Patterns Used

- **Container as a deployment unit:** Clear separation of concerns and decoupled service deployment.
- **Authentication gateway:** `auth-api` centralizes access using JWT.

Production Recommendations

- Add image scanning in GHCR.
- Incorporate monitoring and alerting (e.g., Prometheus and Grafana).
- Use a custom DNS for the frontend.
- Separate environments (`dev`, `staging`, `prod`) using tags and multiple `docker-compose` files.