CI/CD implementation

At EED we have two k8s cluster ***accel-webapp-dev*** and ***accel-webapp*** respectively for testing and production deployment. In this article we are going to describe how CD/CD has been implemented to achieve CI/CD (*Continuous Integration* and *Continuous Delivering*). each one environment is identified by his hostname accel-webapp-dev.slac.stanford.edu and accel-webapp.slac.stanford.edu.

Overview

This document provides a detailed description of the continuous integration (CI) and continuous deployment (CD) pipeline applied across all projects. Our pipeline orchestrates the workflow from development to production, utilizing Kubernetes clusters to ensure smooth transitions and scalability. We've established a process that necessitates manual approval for production deployments, enhancing control and stability. The structure comprises two distinct Git repositories for each application: the Development Project for active development and the Deployment Project for managing deployment processes.

Pipeline Stages

The deployment workflow is segmented into three primary phases: Development, Pre-Production, and Production.

Development Stage

Within the Development Project, the development phase adheres to the trunk-based development model, maintaining the `main` branch as the most stable and release-ready version of the code. This approach emphasizes short-lived branches for small features, which are then merged into the `main` branch post-verification, thereby streamlining the merging and integration steps. This strategy is instrumental in achieving CI/CD objectives, consequently enhancing software delivery and organizational performance.

To support this methodology, two pipelines have been instituted:

Merge Request Pipeline

Executed within the Development Project, this pipeline automates the following tasks:

- **Compile**: Compilation of the codebase.
- **Test**: Execution of automated tests to validate code quality and functionality. This includes initializing the required backend services using a Docker Compose architecture.
- **Merge**: Automatic merging of the merge request (MR) by the system upon successful completion of the prior steps.

Main Branch Update Pipeline

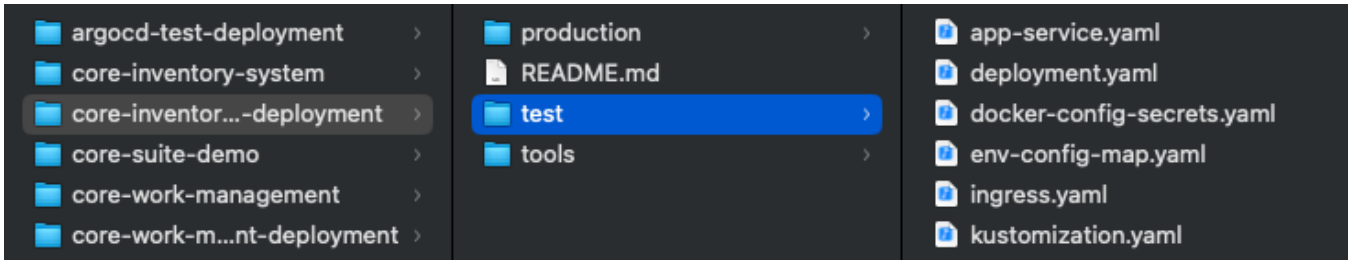
Triggered by updates to the `main` branch within the Development Project, this pipeline performs the following:

- **Compile**: Code compilation.
- **Test**: Automated testing to verify code quality and functionality, which involves spinning up backend services using Docker Compose.
- **Build Docker Image**: Creation of a Docker image from the compiled and tested code.
- **Notify Deployment Pipeline**: Invocation of a subsequent pipeline within the Deployment Project to manage deployment readiness.

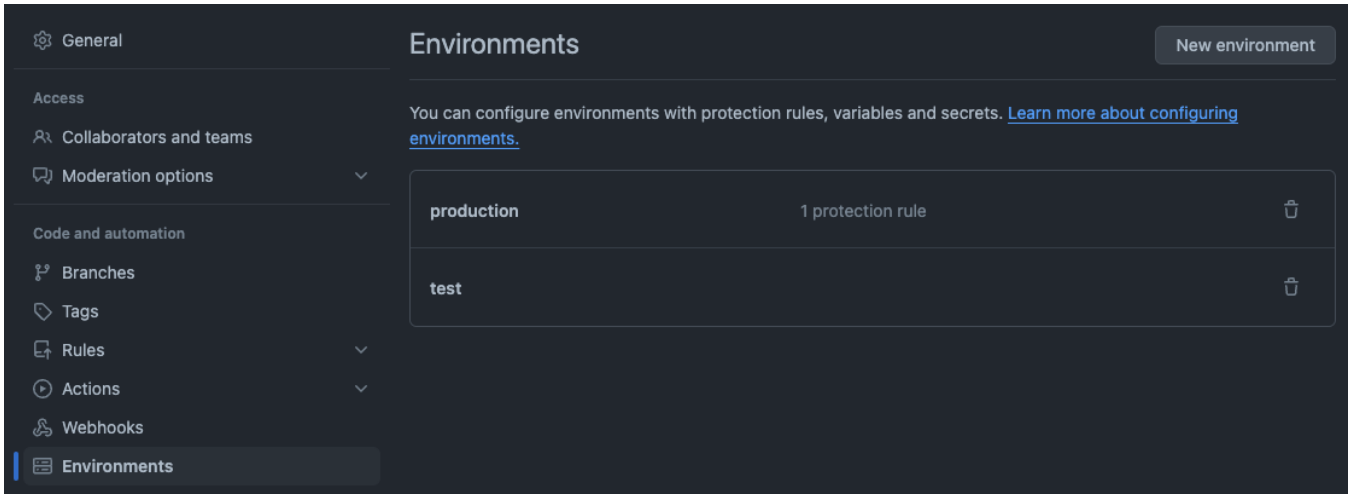
> By maintaining distinct pipelines for merge requests and main branch updates, we ensure rigorous validation at every stage, thereby preserving the integrity and deployability of the `main` branch. This separation of concerns also facilitates a more controlled and monitored transition to the subsequent phases of Pre-Production and Production.

Deployment Stage

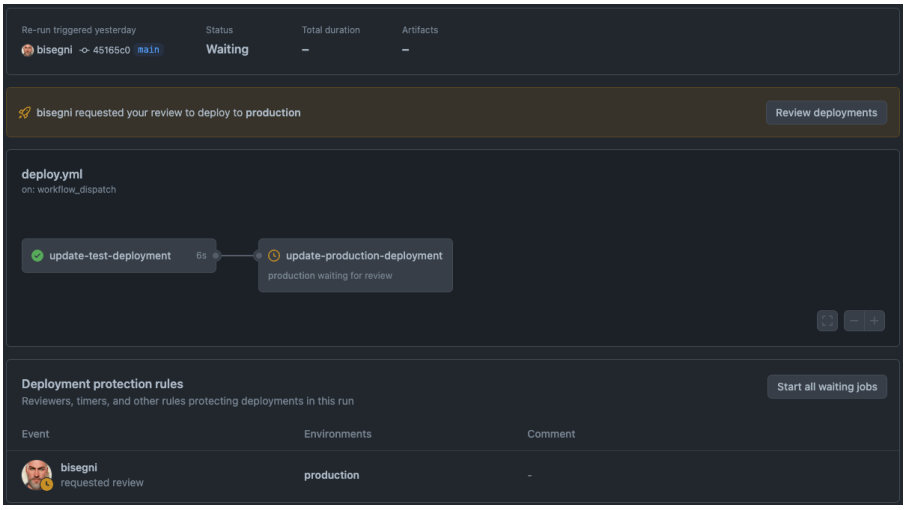
The Deployment Project is a dedicated Git repository that houses the configuration for both the test and production Kubernetes environments. This architecture is strategically designed to segregate duties, granting differentiated access levels to development personnel and deployment managers. Such a setup enhances security and governance, ensuring that only authorized individuals manage the deployment aspect of the application lifecycle. Usually our deployment project looks like the below image:



The deployment procedure commences with a designated initiation trigger with a parameter: the complete Docker image name accompanied by its corresponding tag (e.g., `docker-image:tag`). This image and tag denote the specific build that has been vetted and is ready for deployment. Upon activation by this trigger, the deployment workflow engages in an automated sequence to update the Kubernetes deployment resource. This is meticulously carried out within the test environment's configuration file directory located in the Git repository. Following the automated update, the process enters a holding state, mandating a manual review and intervention. This step serves as a quality gate, necessitating a deliberate action to authorize the promotion of the updated configuration to the production environment. On github this implemented using the 'environment' and protecting the deployment one giving authorization:



the **wait** trigger, on github looks like the one on the below image:



This enforced pause is integral to our deployment strategy, introducing a critical checkpoint to validate system stability and operational readiness. It ensures that all updates undergo a thorough vetting process, reinforcing the integrity of our production environment. Finally the synchronization between the state of the cluster and the git configuration is automated using argocd that is a declarative, GitOps continuous delivery tool for Kubernetes. Each environment (test, production) has his designed argocd instance that manages the specific cluster.

Deployment Flow Recap

1. Code changes are pushed to `Project A GitRepo`.
2. A CI pipeline compiles the code and runs tests.
3. Upon successful testing, a Docker image is built.
4. The image is tagged for pre-production and deployed to the pre-production cluster using Argo CD.
5. After successful deployment and testing in pre-production, a human promotes the configuration for production deployment.
6. Argo CD synchronizes the production cluster with the changes from the production Git repository.

- Develop a NodeJS Backend
- Develop a React UI frontend
- EED git workflow
- CI/CD implementation
- How to deploy webapp on kubernetes