COMPREHENSIVE GUIDE FOR INTEGRATING AZURE WITH GITHUB

DHAIRYA PAHWA

REDBACK OPERATIONS

Table of Contents

What is CI/CD pipelines, and why are they important in the software Development?

CI/CD pipelines, which stand for Continuous Integration and Continuous Deployment (or Continuous Delivery), are a cornerstone of modern software development. These pipelines automate the integration, testing, and deployment of code changes, offering several significantbenefits:

Accelerated Development: CI/CD pipelines enable faster delivery of software updates and newfeatures by automating repetitive tasks like building, testing, and deployment.

Enhanced Quality Assurance: By automating testing processes, CI/CD pipelines help catch bugsearly in the development cycle, ensuring that only high-quality code reaches production environments.

Reduced Manual Effort: Manual tasks are minimized or eliminated altogether, reducing the likelihood of human error and freeing up developers to focus on more strategic activities.

Consistency and Reliability: CI/CD pipelines enforce consistent development and deployment practices across teams, resulting in more predictable outcomes and higher reliability in productionenvironments.

Continuous Feedback Loop: Developers receive rapid feedback on their code changes through automated testing, facilitating faster iterations and improvements to the software.

Scalability and Adaptability: CI/CD pipelines can scale to accommodate projects of any size or complexity, from small startups to large enterprise applications, making them adaptable to diversedevelopment environments.

Which CI/CD platforms are covered in this guide?

This guide covers two major CI/CD platforms:

Azure DevOps: Microsoft's integrated set of tools for building, testing, and deploying software applications, providing comprehensive CI/CD capabilities along with version control, project management, and collaboration features.

GitHub Actions: GitHub's native CI/CD solution that allows developers to automate workflows directly within their GitHub repositories, seamlessly integrating with their existing codebase anddevelopment processes.

What prerequisites are required to set up CI/CD pipelines?

Before setting up CI/CD pipelines, it's important to ensure that you have the following prerequisitesin place:

Accounts: You'll need accounts for the CI/CD platforms you plan to use, such as Azure DevOps, GitHub, or Bitbucket. Make sure you have appropriate permissions to create pipelines and accessrepositories.

Git Repository: You should have a Git repository set up to store your codebase. This can be hosted on platforms like GitHub, GitLab, Bitbucket, or Azure Repos.

Access to Source Code: Ensure you have access to the source code repository where your application code resides. This may involve permissions to clone, push, and pull code from the repository.

Software Dependencies: Depending on your project requirements, you may need to install specific software dependencies or tools required for building, testing, and deploying your application. This could include programming language runtimes, build tools, testing frameworks, and deployment utilities.

Environment Configuration: Set up the necessary development, staging, and production environments where you'll deploy your application. Ensure these environments are properly configured and accessible to your CI/CD pipelines.

Build Scripts or Configuration Files: Prepare any build scripts or configuration files required for building and testing your application. For example, you may need a package.json file for a Node.js project, a pom.xml for a Maven project, or a Dockerfile for containerized applications.

Understanding of CI/CD Concepts: Familiarize yourself with the basic concepts of CI/CD, including version control, automated testing, continuous integration, and continuous deployment.This will help you understand how CI/CD pipelines work and how to configure them effectively.

What topics are covered in the documentation beyond pipeline creation?

The documentation covers several essential topics beyond pipeline creation, including:

Pipeline Configuration: Instructions for configuring pipelines, defining build and deployment steps, and setting up environments.
Adding Tasks and Actions: Guidance on adding tasks or actions, with examples of commonly used ones.
Customization: Information on customizing pipelines using variables, templates, and other advanced features.
Testing: Best practices for implementing automated testing within CI/CD pipelines.
Deployment: Instructions for configuring deployment steps and strategies.
Security and Permissions: Recommendations for securing pipelines and managing access controls.
Monitoring and Logging: Explanation of monitoring pipeline runs, viewing logs, and troubleshooting.
Best Practices: Tips for optimizing pipelines, organizing repositories, and managing dependencies.

PREREQUISITES

To get started with setting up CI/CD pipelines, ensure you have the following prerequisites:

Accounts:
Create accounts for the CI/CD platforms you plan to use, such as Azure DevOps, GitHub, orBitbucket.

Sign up for Azure DevOps here.

Sign up for GitHub here.

Git Repository:
Set up a Git repository to store your code. You can use platforms like GitHub, GitLab, Bitbucket, orAzure Repos. If you haven't already, create a new repository on your chosen platform and ensure you have access to it.

Access to Source Code:
Ensure you have access to the repository where your application code resides. This may involvepermissions to clone, push, and pull code from the repository.

Software Dependencies:
Install any necessary software dependencies required for building, testing, and deploying your application. This may include programming language runtimes, build tools, testing frameworks, anddeployment utilities.

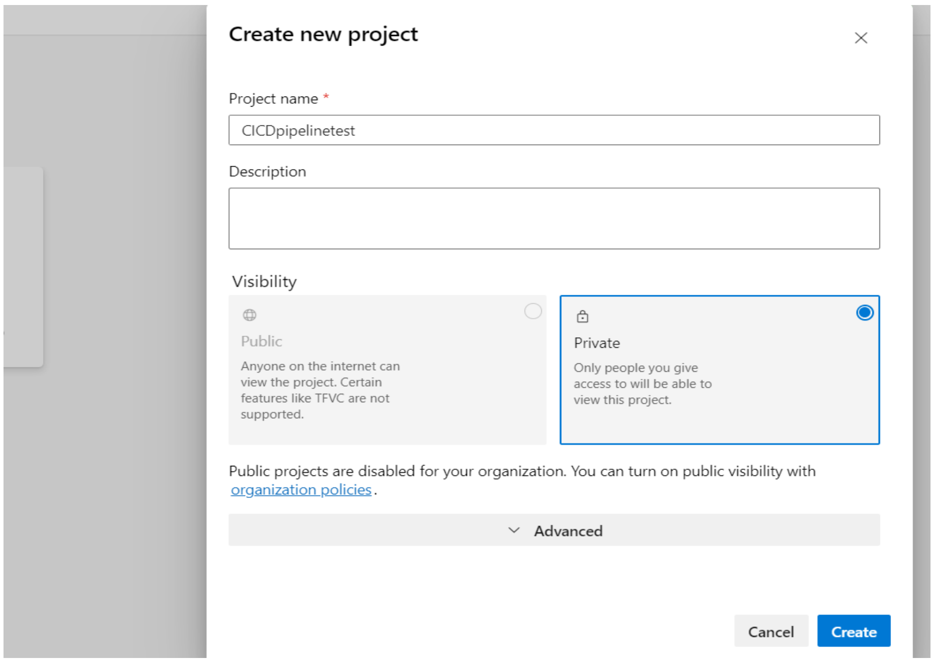
By ensuring you have these prerequisites in place, you'll be ready to start setting up CI/CD pipelinesfor your software development projects.

CREATING A PIPELINE

In this step, we would be learning how to create pipelines for different platforms we are using in thisguide.

Azure Devops

1) In your organization, create a new project and navigate to it. We will create a new project with thename "CI/CDpipelinetest"



Create new project

Project name *

CICDpipelinetest

Description

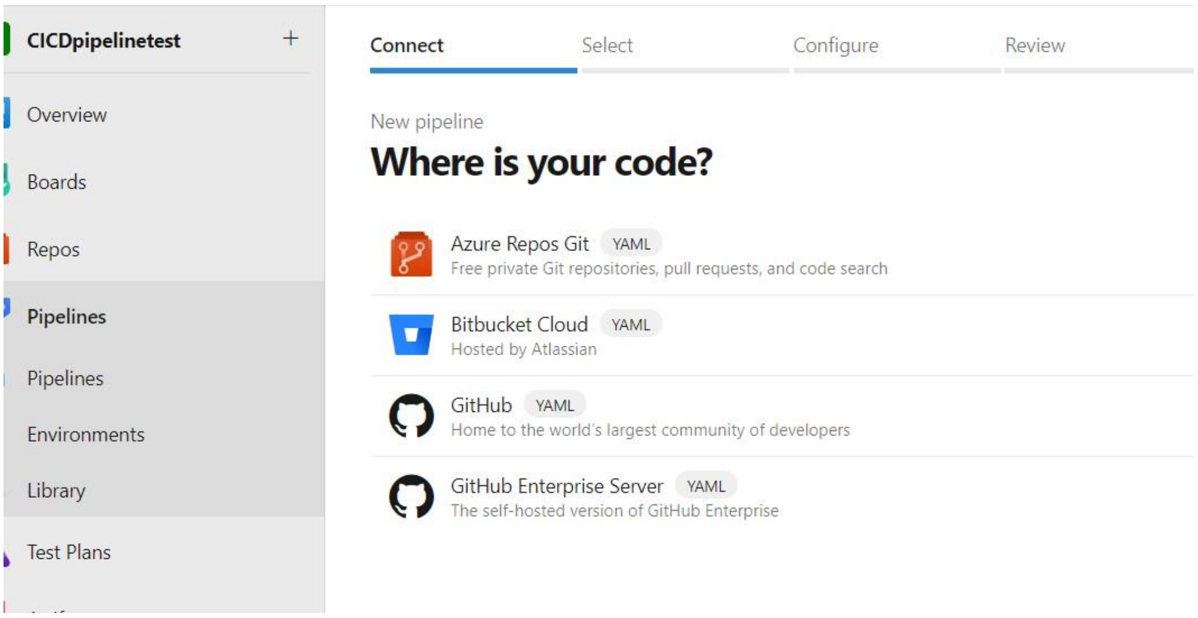
Visibility

Public
Anyone on the internet can view the project. Certain features like TFVC are not supported.

Private
Only people you give access to will be able to view this project.

Public projects are disabled for your organization. You can turn on public visibility with organization policies.

2) Go to Pipeline -> Click on Create Pipeline -> Select where your code is stored



New pipeline

Where is your code?

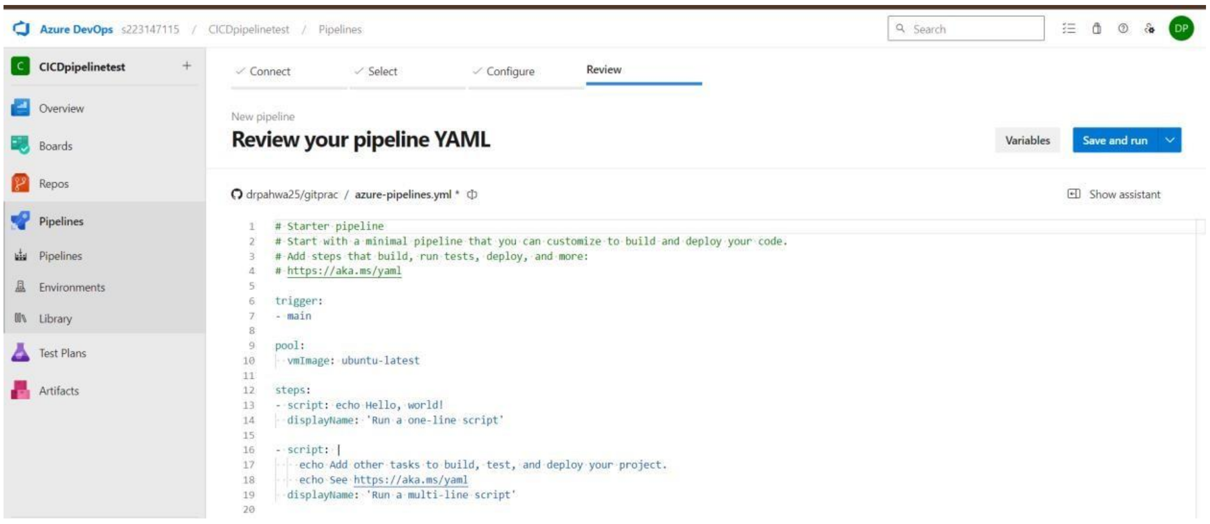
Azure Repos Git YAML
Free private Git repositories, pull requests, and code search

Bitbucket Cloud YAML
Hosted by Atlassian

GitHub YAML
Home to the world's largest community of developers

GitHub Enterprise Server YAML
The self-hosted version of GitHub Enterprise

3) Connect your Github and select the repo where the code is stored Configure your pipeline

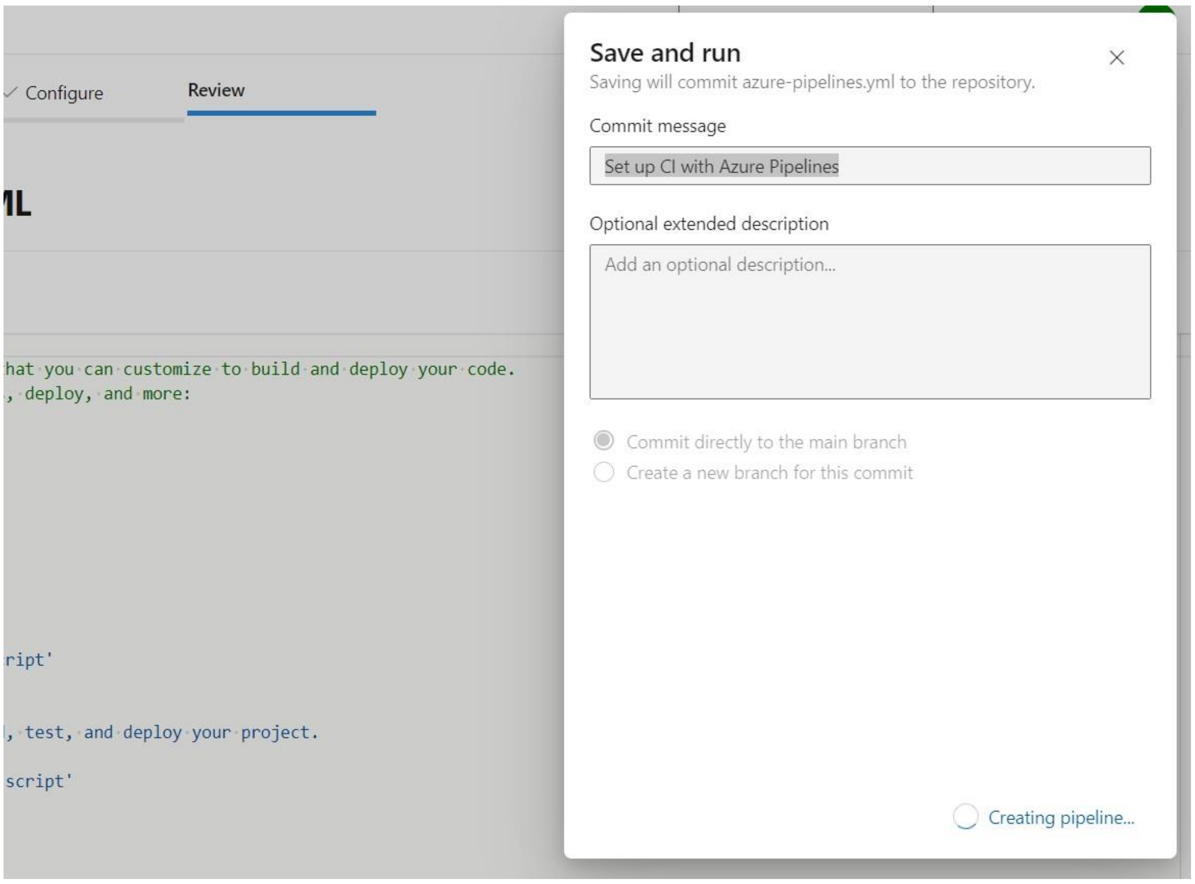


Review your pipeline YAML

drpahwa25/gitprac / azure-pipelines.yml *

```yaml
# Starter pipeline
# Start with a minimal pipeline that you can customize to build and deploy your code.
# Add steps that build, run tests, deploy, and more:
# https://aka.ms/yaml

trigger:
- main

pool:
  vmImage: ubuntu-latest

steps:
- script: echo Hello, world!
  displayName: 'Run a one-line script'

- script: |
    echo Add other tasks to build, test, and deploy your project.
    echo See https://aka.ms/yaml
  displayName: 'Run a multi-line script'

```

4) Create your pipeline



```
hat can customize to build and deploy your code.
, deploy, and more:

ript'

, test, and deploy your project.

script'
```

Save and run

Saving will commit azure-pipelines.yml to the repository.

Commit message: Set up CI with Azure Pipelines

Optional extended description: Add an optional description...

Commit directly to the main branch [✓]
Create a new branch for this commit

Creating pipeline...

Configurations for the Pipeline :-

Choose your pipeline configuration option:

YAML-based Pipeline: Recommended for more flexibility and version-controlled configuration. You'll need to create a azure-pipelines.yml file in your repository.

Classic Pipeline: Offers a visual editor for defining your pipeline without writing YAML. This may bepreferred for simpler pipelines or for users who are less familiar with YAML syntax.

Follow the guided steps to configure your pipeline:

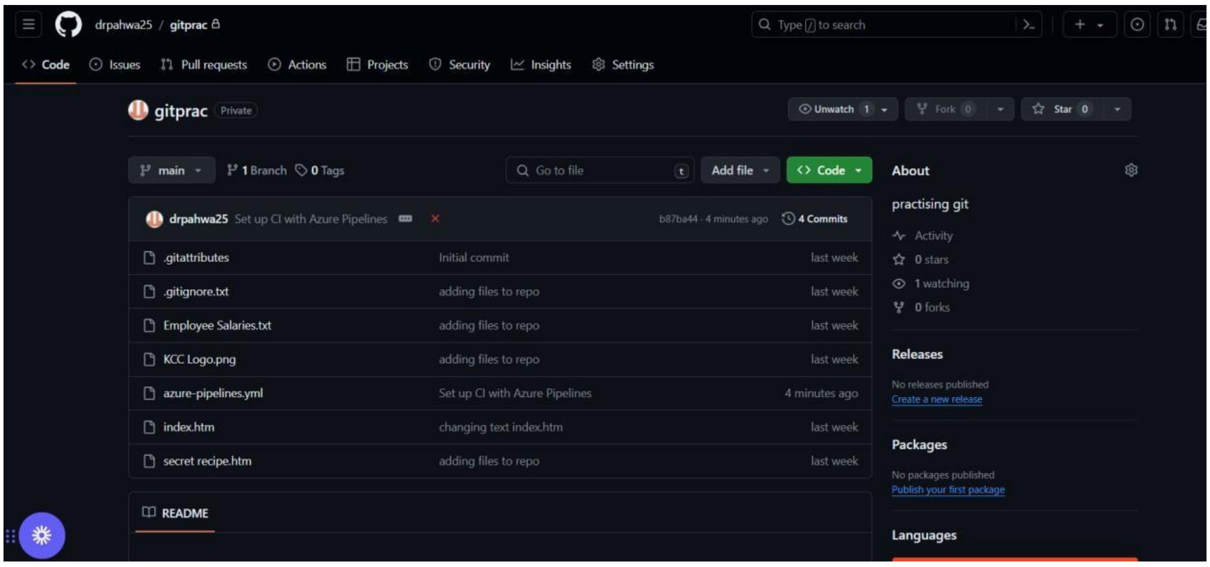
Define triggers, such as triggers on branch updates or pull requests.

Add stages and jobs to define the tasks your pipeline will perform, such as building, testing, anddeploying your application.

Specify agent pools and job execution environments.

<u>GitHub Actions:</u>

1. Go to your GitHub repository.



drpahwa25 Set up CI with Azure Pipelines — b87ba44 · 4 minutes ago — 4 Commits

.gitattributes	Initial commit	last week
.gitignore.txt	adding files to repo	last week
Employee Salaries.txt	adding files to repo	last week
KCC Logo.png	adding files to repo	last week
azure-pipelines.yml	Set up CI with Azure Pipelines	4 minutes ago
index.htm	changing text index.htm	last week
secret recipe.htm	adding files to repo	last week

About

practising git

Activity
0 stars
1 watching
0 forks

Releases

No releases published
Create a new release

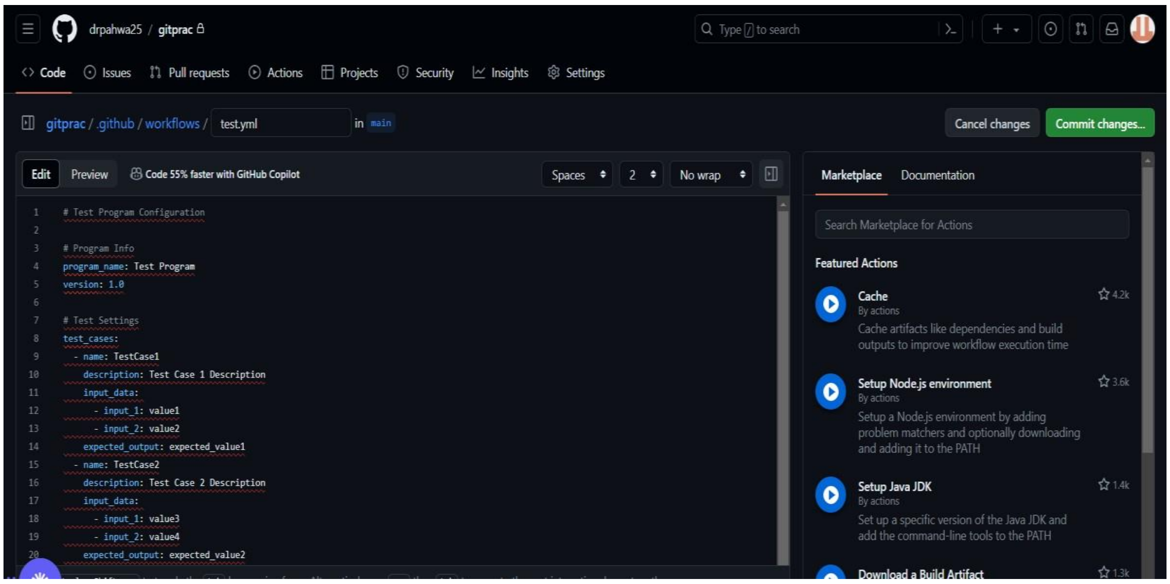
Packages

No packages published
Publish your first package

Languages

2. Navigate to the "Actions" tab.

3. Click on "Set up a workflow yourself" or choose from a template.



```
# Test Program Configuration

# Program Info
program_name: Test Program
version: 1.0

# Test Settings
test_cases:
  - name: TestCase1
    description: Test Case 1 Description
    input_data:
      - input_1: value1
      - input_2: value2
    expected_output: expected_value1
  - name: TestCase2
    description: Test Case 2 Description
    input_data:
      - input_1: value3
      - input_2: value4
    expected_output: expected_value2
```

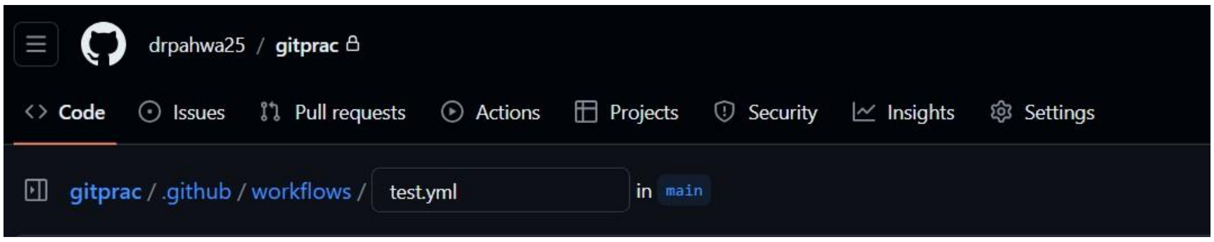
Cache
By actions
Cache artifacts like dependencies and build outputs to improve workflow execution time

Setup Node.js environment
By actions
Setup a Node.js environment by adding problem matchers and optionally downloading and adding it to the PATH

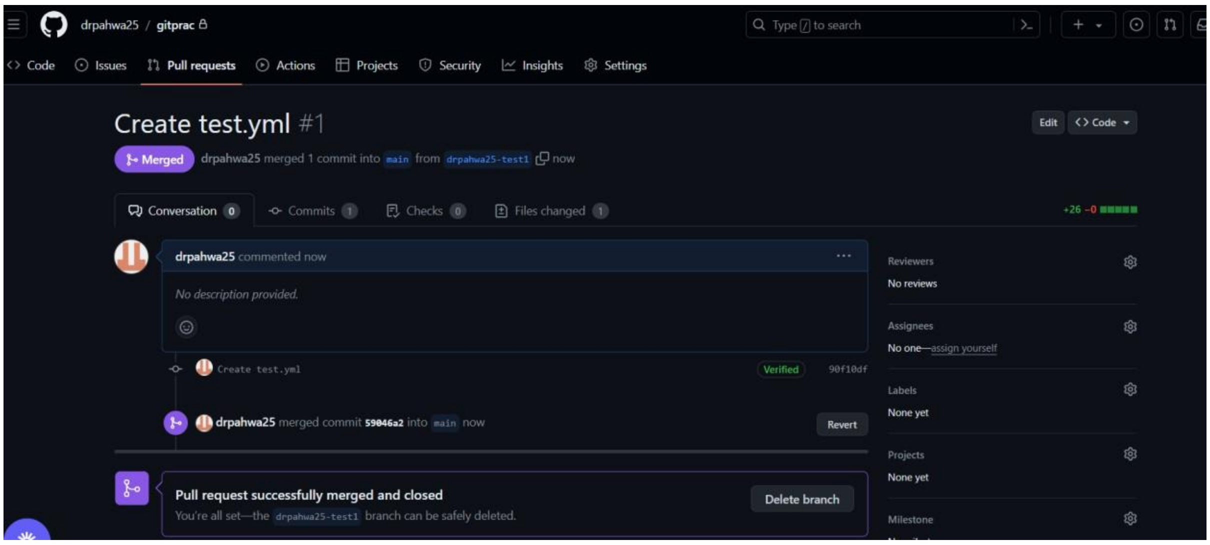
Setup Java JDK
By actions
Set up a specific version of the Java JDK and add the command-line tools to the PATH

Download a Build Artifact

4. Define your workflow using YAML syntax in the .github/workflows directory of your repository.



5. Configure triggers, jobs, and steps in your workflow file to define the tasks your pipeline will perform.
6. Commit and push your workflow file to trigger the pipeline.



Create test.yml #1

Merged drpahwa25 merged 1 commit into main from drpahwa25-test1 now

drpahwa25 commented now

No description provided.

Create test.yml Verified 90f10df

drpahwa25 merged commit 59046a2 into main now

Pull request successfully merged and closed

You're all set—the drpahwa25-test1 branch can be safely deleted.

These steps provide a basic guide for creating pipelines in Azure DevOps and GitHub Actions. Depending on your project's requirements and familiarity with YAML syntax, you can choose between YAML-based or visual pipelines in Azure DevOps, while GitHub Actions and BitbucketPipelines primarily utilize YAML-based configurations.

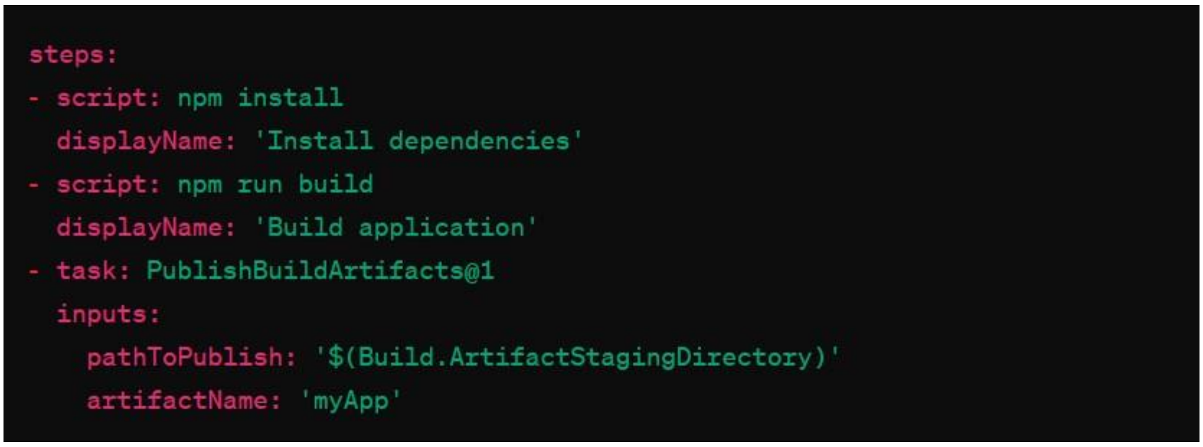
CONFIGURATION OF PIPELINE

In pipeline configuration, you have various options to define the behaviour of your CI/CD pipeline.Here's a breakdown of the configuration options and examples of common scenarios:

Defining Build and Deployment Steps:

Specify the tasks or actions your pipeline will perform during the build, test, and deployment stages.

Example: In Azure DevOps YAML pipeline, you can define steps using the steps keyword, whereeach step executes a specific command or action. For instance:



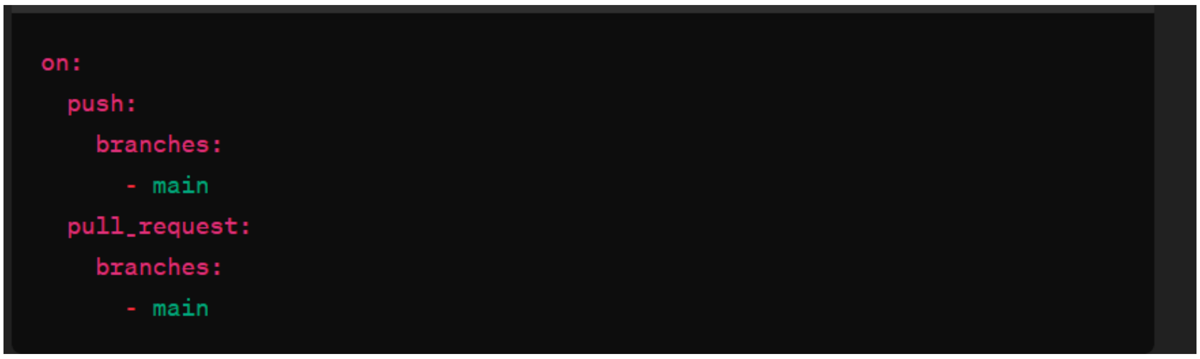
```
steps:
- script: npm install
  displayName: 'Install dependencies'
- script: npm run build
  displayName: 'Build application'
- task: PublishBuildArtifacts@1
  inputs:
    pathToPublish: '$(Build.ArtifactStagingDirectory)'
    artifactName: 'myApp'
```

This example installs dependencies, builds the application, and publishes artifacts for deployment.

Specifying Triggers:

Define the conditions that trigger your pipeline to run, such as code commits, pull requests, orscheduled builds.

Example: In GitHub Actions, you can use the on keyword to specify triggers. For instance:



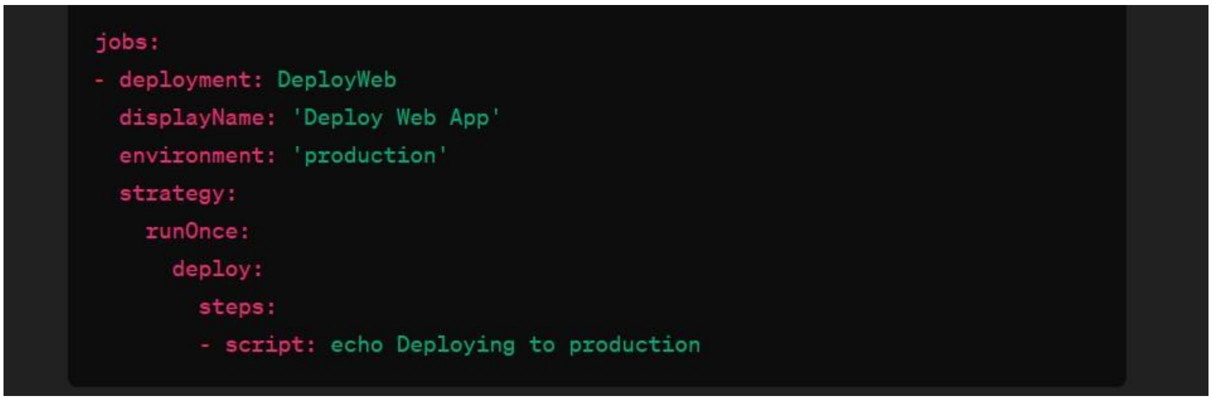
```
on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main
```

This configuration triggers the pipeline on pushes to the main branch and pull requests targeting the main branch.

Setting Up Environments:

Define environments for deployment, such as development, staging, and production.

Example: In Azure DevOps, you can define environments in the pipeline YAML file or in the Azure Pipelines UI. For instance:



```
jobs:
- deployment: DeployWeb
  displayName: 'Deploy Web App'
  environment: 'production'
  strategy:
    runOnce:
      deploy:
        steps:
        - script: echo Deploying to production
```

This configuration defines a deployment job named "DeployWeb" targeting the "production" environment.

Common Configuration Scenarios:

Parallel Jobs: Run multiple jobs concurrently to speed up the pipeline execution.

Conditional Steps: Execute steps based on conditions such as the branch being built or the outcome of previous steps.

Artifacts: Publish and consume build artifacts for sharing files between pipeline stages or jobs.

Environment Variables: Define and use environment variables to customize pipeline behavior or pass sensitive information securely.

Integration with External Services: Integrate with external services like Docker Hub, AWS, or Azure for container registry, cloud deployment, or other tasks.

ADDING TASKS AND ACTIONS

Here are step-by-step instructions on how to add tasks or actions to your CI/CD pipeline for Azure DevOps, GitHub Actions, and Bitbucket Pipelines:

Azure DevOps:

1. Navigate to Pipeline Editor:
2. Go to your Azure DevOps project.
3. Click on Pipelines > Pipelines. Select your pipeline.

4. Edit Pipeline Configuration: If you're using a YAML-based pipeline, locate the YAML configuration file. If not, you may need to edit the pipeline using the visual editor. Find thesection where you want to add tasks or actions. This is typically within the jobs section.
5. Add Tasks: Use predefined tasks provided by Azure DevOps or create custom tasks usingscripts.
6. Save and Run Pipeline: Save the changes to your pipeline configuration. Trigger the pipeline manually or wait for the defined triggers to initiate the pipeline execution.

<u>GitHub Actions:</u>

1. Navigate to Workflow Editor:
2. Go to your GitHub repository.
3. Click on the "Actions" tab.
   Select the workflow file you want to edit or create a new one.
4. Edit Workflow YAML:
   Add steps within the jobs section of the workflow YAML file.
5. Commit and Push Changes:
6. Save the changes to the workflow file.
7. Commit and push the changes to your GitHub repository.

TESTING

Implementing testing within CI/CD pipelines is crucial for ensuring the quality and reliability of yoursoftware. Here are some best practices for setting up and running automated tests as part of your pipeline, along with guidance on integrating different types of tests:

Define Test Strategy: Before implementing tests in your pipeline, define a clear test strategy outlining the types of tests needed for your application, such as unit tests, integration tests, andend-to-end tests. Consider the testing pyramid concept, where you have a larger number of unit tests at the base,followed by fewer integration tests, and even fewer end-to-end tests at the top.

Automate Tests: Automate as many tests as possible to ensure they can be run consistently andrepeatably as part of your pipeline.
Use testing frameworks and libraries appropriate for your programming language and technologystack to write automated tests.

Parallelize Tests: Parallelize test execution to speed up the overall testing process and reduce thetime taken for feedback. Divide tests into smaller groups and run them concurrently on separate agents or containers.

Integrate Tests into Pipeline Stages:
Integrate tests into different stages of your pipeline, such as build, test, and deploy stages. Run fastand lightweight tests, such as unit tests, in earlier stages to provide quick feedback on code changes.

Reserve heavier and longer-running tests, such as integration and end-to-end tests, for later stagesto avoid blocking the pipeline unnecessarily.

Use Docker for Test Environment: Use Docker containers to create isolated and consistent test environments. Define Docker images containing the necessary dependencies and configurations forrunning tests.

Instrumentation and Code Coverage:

Measure code coverage to ensure that your tests adequately cover your codebase. Use codecoverage tools to identify areas of your code that require additional testing.

Implement Continuous Testing:

Implement continuous testing practices where tests are triggered automatically on code changes.Use pre-commit hooks or pull request validations to ensure that tests pass before code is merged into the main branch.

Handle Test Failures Gracefully:

Implement mechanisms to handle test failures gracefully, such as retrying failed tests or notifyingteam members.
Ensure that failing tests are investigated and addressed promptly to maintain the integrity of thepipeline.

DEPLOYMENT

Deployment is a critical aspect of the CI/CD pipeline, ensuring that tested and validated codechanges are successfully released to various environments. Here's a guide to the deployment process and strategies:

Deployment Process:
The deployment process involves taking the application artifacts generated during the build processand deploying them to the target environment, such as staging, testing, or production.
It typically includes steps like transferring files, configuring infrastructure, and starting or updatingservices.

Deployment Strategies:

Rolling Deployments: Gradually replace instances of the previous version with instances of the newversion, ensuring minimal downtime and continuous availability.
Blue-Green Deployments: Route traffic between two identical production environments - onerunning the current version (blue) and the other running the new version (green). This allows forinstant rollback in case of issues.
Canary Deployments: Introduce the new version to a small subset of users or servers before rollingit out to the entire infrastructure, allowing for monitoring of its performance and stability.
Feature Flags: Gradually enable new features for users by toggling feature flags, allowing for

controlled rollouts and quick rollbacks if necessary.

Configuring Deployment Steps in the Pipeline:

Define deployment steps in the CI/CD pipeline after successful testing.Use
deployment tools or scripts to automate the deployment process.
Ensure that deployment steps are idempotent, meaning they can be run multiple times withoutcausing unintended side effects.

Handling Deployment-related Tasks:

Rollback: Implement rollback mechanisms to revert to the previous version in case of deploymentfailures or issues.
Health Checks: Perform health checks on the deployed application to ensure it's running asexpected before directing traffic to it.
Database Migrations: Handle database schema changes or data migrations as part of thedeployment process, ensuring data consistency and integrity.
Monitoring and Logging: Monitor the deployment process and application performance in real-time,and log deployment activities for auditing and troubleshooting purposes.

SECURITY AND PERMISSIONS

Securing pipelines and managing access controls within CI/CD pipelines are essential to protectsensitive information and ensure the integrity of the software delivery process. Here are some recommendations and best practices:

Use Role-Based Access Control (RBAC):

Implement RBAC to control access to pipelines, repositories, and other resources based on rolesand responsibilities.
Define roles such as administrators, developers, and testers, and grant appropriate permissions toeach role.

Limit Access to Secrets and Credentials:

Avoid storing sensitive information such as passwords, API keys, or connection strings directly inpipeline configuration files.
Use secure vaults or secret management services provided by the CI/CD platform to store andmanage secrets.
Grant access to secrets on a need-to-know basis, ensuring that only authorized users or servicescan access them.

Encrypt Communication and Data:

Enable encryption for communication between CI/CD agents and the platform's servers to protectdata in transit.

Utilize encryption mechanisms provided by the platform to encrypt sensitive data stored within thepipeline configuration or secret management systems.

Implement Two-Factor Authentication (2FA):

Enable 2FA for user accounts accessing the CI/CD platform to add an extra layer of security against unauthorized access.

Monitor Pipeline Activities:

Set up monitoring and auditing mechanisms to track pipeline activities, including who triggered thepipeline, what changes were made, and when they occurred.
Regularly review audit logs to detect any suspicious activities or unauthorized access attempts.

Use Service Connections:

Configure service connections to securely authenticate and authorize external services or resources accessed by the pipeline, such as cloud providers, container registries, or databases.

Use platform-specific mechanisms to create and manage service connections securely.

Manage Secrets and Environment Variables:

Store sensitive information like API keys or passwords as environment variables or pipeline variablesrather than hardcoding them in scripts or configuration files.

Encrypt environment variables or secrets at rest and in transit to prevent unauthorized access.

Regularly Update and Patch CI/CD Tools:

Keep CI/CD tools and associated plugins up to date with the latest security patches and updates toaddress known vulnerabilities and security issues.

MONITORING AND LOGGING

Monitoring and logging are crucial aspects of CI/CD pipelines to ensure visibility into pipeline runs,detect issues promptly, and troubleshoot any problems that arise. Here's how users can monitor pipeline runs, view logs, and troubleshoot issues within the CI/CD process:

Monitoring Pipeline Runs:

Monitor pipeline runs in real-time to track their progress and status.
Receive notifications or alerts for pipeline successes, failures, or other events.
Use dashboards or visualizations to gain insights into pipeline performance and trends over time.

Viewing Logs:

Access detailed logs generated during pipeline execution to understand each step's outcome andidentify any errors or issues.
View logs for specific pipeline runs, jobs, or individual steps to pinpoint the source of problems.

Filter and search logs based on criteria such as timestamps, keywords, or error codes to streamline troubleshooting.

Troubleshooting Issues:

Analyze log messages and error outputs to diagnose and troubleshoot issues encountered duringpipeline runs.
Identify failed or stalled pipeline steps and investigate the root causes of failures.
Utilize built-in diagnostic tools or third-party integrations for advanced troubleshooting anddebugging.

As for the monitoring and logging capabilities available within the chosen CI/CD platform, here's anoverview:

Azure DevOps:

Azure DevOps provides comprehensive monitoring and logging capabilities through its Pipelinesfeature.
Users can monitor pipeline runs, view detailed logs, and troubleshoot issues directly within theAzure DevOps portal.
Built-in dashboards and reports offer insights into pipeline performance, test results, anddeployment trends.

GitHub Actions:

GitHub Actions offers robust monitoring and logging capabilities for CI/CD workflows.
Users can monitor workflow runs, view detailed execution logs, and analyze performance metricsusing the GitHub web interface.
Integration with GitHub's native features allows for seamless collaboration and issue trackingduring troubleshooting.

CONCLUSION

Implementing a robust CI/CD pipeline is essential for modern software development practices, enabling teams to automate and streamline the delivery of high-quality software. Throughout thisdocument, we have explored various aspects of CI/CD pipelines, including setting up pipelines, integrating testing, managing security, monitoring pipeline runs, and troubleshooting issues.

By defining clear test strategies, automating tests, and integrating them into pipeline stages, teamscan ensure the reliability and quality of their software. Deployment strategies such as rolling deployments and blue-green deployments enable smooth releases to different environments whileminimizing downtime and risks.

Security and permissions play a critical role in protecting sensitive information and ensuring the integrity of the pipeline. By implementing role-based access control, limiting access to secrets, andencrypting communication and data, teams can mitigate security risks effectively.

Monitoring pipeline runs and viewing logs are essential for gaining visibility into the pipeline's performance and troubleshooting issues promptly. Leveraging the monitoring and logging capabilities offered by CI/CD platforms enables teams to track pipeline progress, analyze logs, anddiagnose issues efficiently.

In essence, by following best practices and leveraging the capabilities of CI/CD platforms, teams can accelerate software delivery, improve collaboration, and deliver value to customers faster andmore reliably. Continuous improvement and optimization of CI/CD pipelines are key to staying competitive in today's fast-paced software development landscape.