



KodeKloud

© Copyright KodeKloud

Follow us on <https://kodekloud.com/> to learn more about us.



Problem Statement (Meeting With Task Dash Team)

Task Dash Team DevOps Requirement



Alice



Docker for
Containerization



Kubernetes for
Container Orchestration

Let's start the course by understanding the DevOps prerequisites of a software provider. Over the duration of this course, we will check into and acquire knowledge about how Github actions can be used to meet these DevOps requirements.

Dasher technology operates as a software provider, offering a platform to facilitate the connection of data, applications, and devices across on-premises environments for businesses. Recently, their Research and Development (R&D) team has been exploring the possibilities of transitioning their services to the Cloud and leveraging container technologies. Initially, they aim to apply this to one of their NodeJS-based projects, with intentions to extend this approach to other projects

They've introduced a DevOps team for this initiative, led by Alice, who will be responsible for establishing the DevOps pipeline for the project right from its inception, adhering to industry best practices. Their work will revolve around a multi-cloud infrastructure, utilizing Docker for containerization and Kubernetes for container orchestration.

Task Dash Team DevOps Requirement



Alice



Version Control



Code Integration



Collaboration



Manual Testing



Manual Deploy



Unit Test



Code Coverage



Build



Push



Deploy



Automated IT

Alice conducted a swift evaluation and learned that the current requirement pertains to a NodeJS project. The previous team operated without a Version Control System, with developers independently writing and manually integrating code. The testing process was sluggish and ineffective due to manual execution. Collaboration among developers was often hampered as they worked on separate code branches. With infrequent integration and testing, software releases carried more significant risks. Deployment of software to various environments, including development, staging, and production, was primarily a manual procedure.

1. Adoption of Github for version control and developer collaboration.
- 2.Implementation of unit testing and code coverage measures to expedite testing and minimize bugs.
- 3.Utilization of Docker Build and Push processes for containerization, and the application is deployed to Kubernetes.
- 4.Incorporation of automated integration testing as a final step.

The successful execution of these steps is expected to resolve the existing issues. Nevertheless, the team now faces the additional hurdle of selecting the most suitable CI/CD tool.

Task Dash Team DevOps Requirement



Jenkins



Travis CI



Circle CI



Atlassian Bamboo



Spinnaker

In the market, a wide array of CI/CD tools is available, including Jenkins, Travis CI, Bamboo, Spinnaker, Circle CI, and more. After exploring several options, the team has made the decision to proceed with Jenkins, as it stands as one of the most extensively used open-source CI/CD tools.

Traditional CI/CD Tools – Challenges

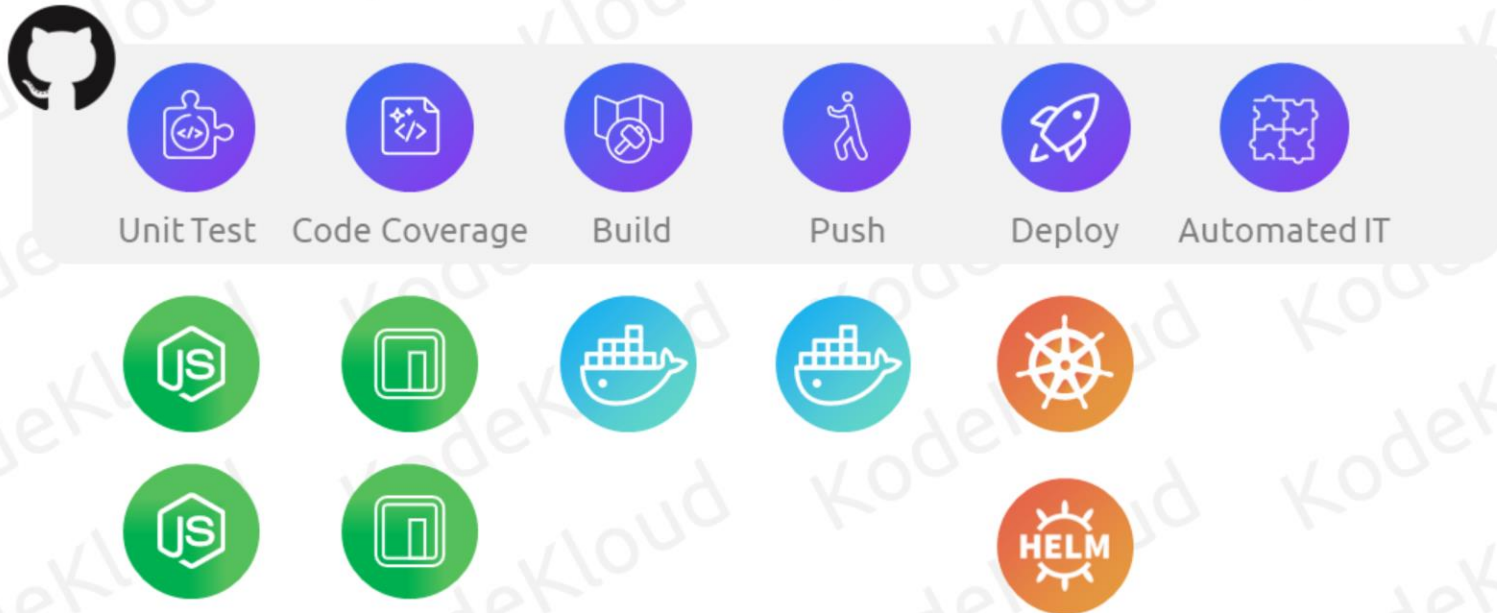


© Copyright KodeKloud

To employ Jenkins, the team is required to undertake several tasks:

1. Configuration, operation, and upkeep of a virtual machine with specific CPU, memory, and HDD capacity.
2. Prior to Jenkins installation, prerequisites include Java JDK installation, setting up firewall rules, and installing Jenkins Plugins.
3. Given that this project is centered around Node.js, Node.js and Npm must be installed.

Traditional CI/CD Tools – Challenges



© Copyright KodeKloud

4. To facilitate unit testing with various Node.js versions, the DevOps engineer must install multiple Node.js/Npm versions, ensuring tests run without version conflicts.
5. Docker installation is essential for containerization.
6. For Kubernetes deployment, the installation of kubectl, helm, or other necessary binaries is mandatory.
7. Teams often employ external tools for integration testing and reporting, necessitating the installation of these tools or their Command-Line Interfaces (CLIs) on the machine where Jenkins will execute them.

Traditional CI/CD Tools – Challenges



© Copyright KodeKloud

Furthermore, the organization intends to implement similar methodologies for other Java and Python projects/applications running on cloud platforms like AWS and Azure.

Traditional CI/CD Tools – Challenges



Java



Maven



Python



Azure



AWS CLI



Trivy



Kubesecc

Depending on the specific project, additional software requirements such as Java, Maven, Python, Azure/AWS CLI may be necessary. If the pipeline demands enhanced security in line with DevSecOps practices, more tools like Trivy and kubesecc must be configured.

Considering that this DevOps team is recently established, many team members lack familiarity with a broad spectrum of tools and services.

Alice has initiated a search for a tool that satisfies the following criteria:

- 1.Simplified setup and initiation without the need to install and configure numerous services.
- 2.A focus on building pipelines without the burden of managing infrastructure or concerns regarding scalability.

Traditional CI/CD Tools – Challenges



GitHub Actions

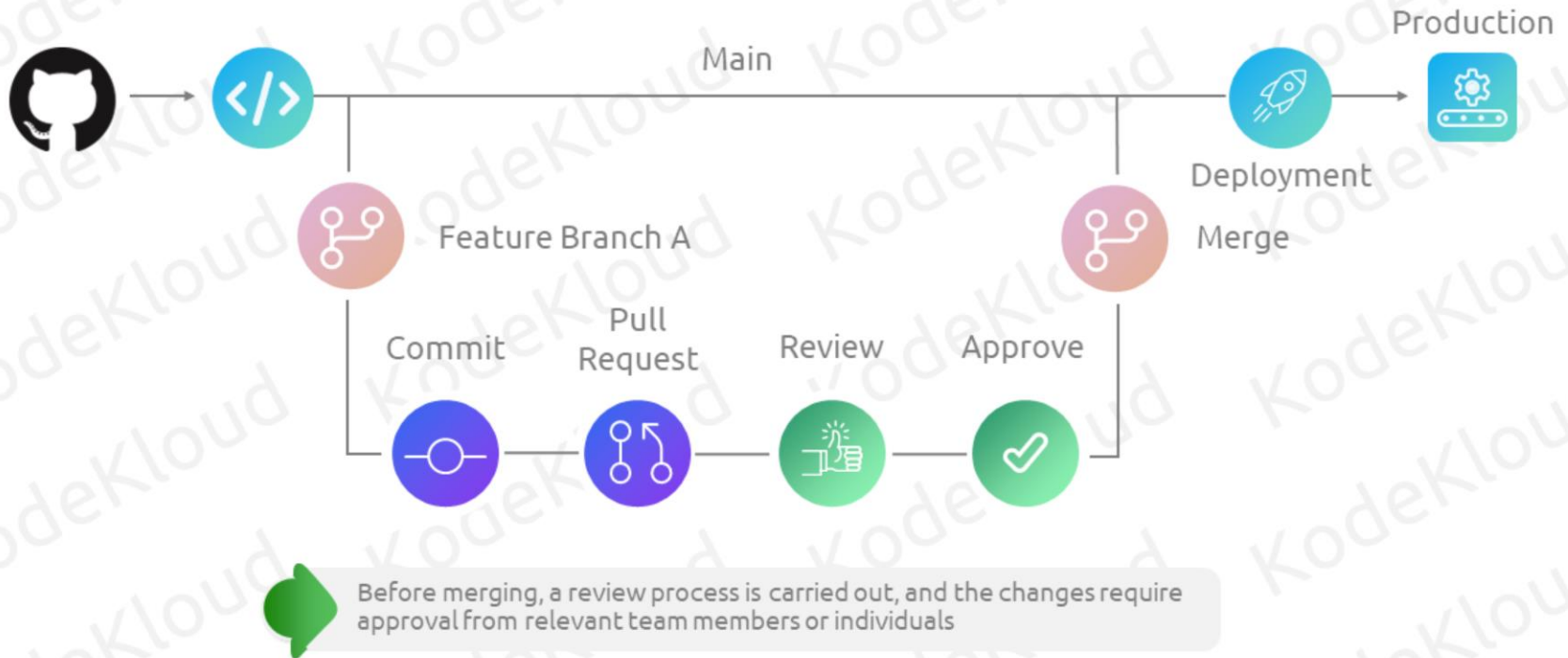
© Copyright KodeKloud

Following a thorough evaluation of multiple tools, Alice has chosen to delve into and implement Github Actions.

Over the course of this training program, we will check into the creation of Github Actions Workflows for a straightforward yet a real-time NodeJS application.

Basics of CI/CD

Why CI/CD?



© Copyright KodeKloud

Let's explore the importance of Continuous Integration and Continuous Deployment (CICD).

In a typical project, source code resides in a Git repository where it's both stored and versioned. GitHub, a web-based platform centered around Git, serves as a centralized hub for hosting these repositories and extends Git's capabilities with additional features.

Typically, all code residing on the main or master branch is frequently deployed to production servers or environments.

known as a "feature branch." This branch essentially functions as a clone of the main codebase, enabling a team of developers to work on a new feature until it's fully developed.

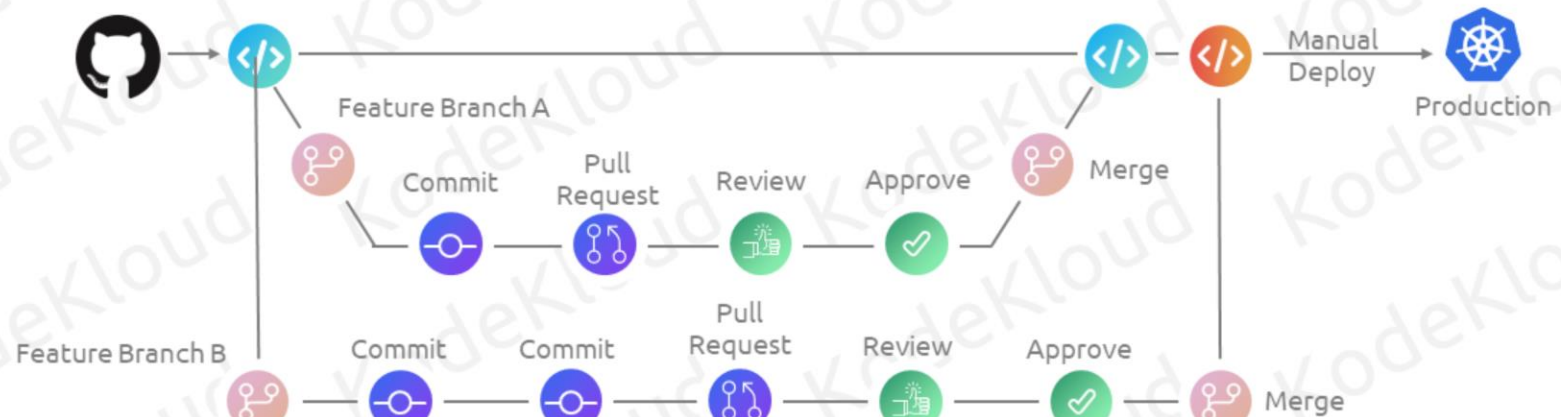
Once the necessary changes are made, the code is committed to the feature branch, and a pull request is initiated to merge this code back into the main branch.

Before the merge takes place, a review process is carried out, and the changes require approval from relevant team members or individuals.

Following a successful merge into the main branch, the code is then deployed to production, either manually or through an automated process.

This situation poses a significant risk to the application's stability, as there's often no testing conducted on the newly merged code before it reaches the production environment.

Need for Continuous Integration



Delayed Testing

Without CI, testing typically occurs late in the development cycle, often after multiple merges have taken place

Inefficient Deployment

In the absence of CI, deploying code to various environments (e.g., staging, production) often relies on manual processes

Quality Assurance Challenges

Without automated testing, ensuring quality becomes more reliant on manual testing, making it prone to human error

In real time scenarios you'll often find multiple developers working on different feature branches each focusing on a new enhancement.

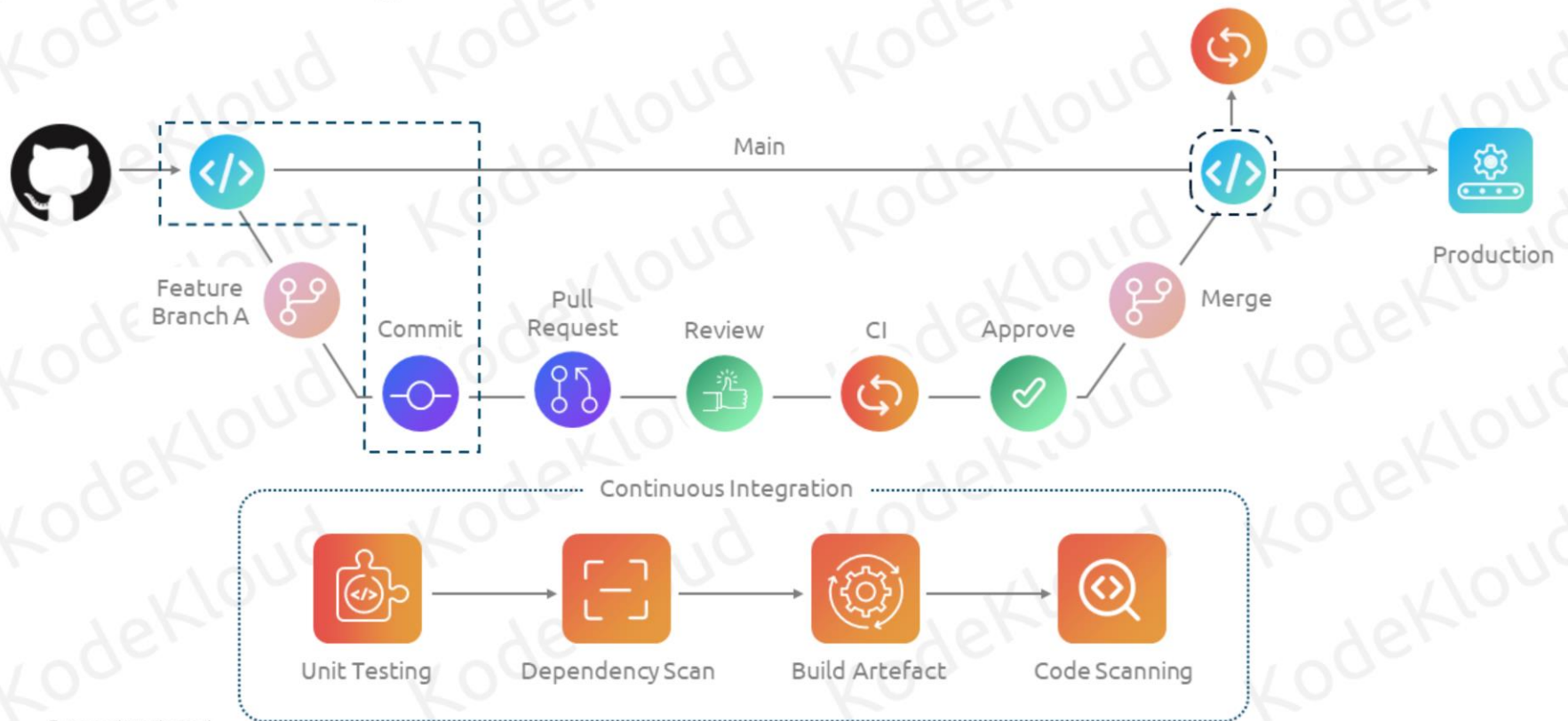
Performing multiple merges without the implementation of Continuous Integration (CI) in a software development workflow can lead to several significant problems and challenges such as:

Delayed Testing: Without CI, testing typically occurs late in the development cycle, often after multiple merges have taken place. This delay in testing can make it harder to identify and rectify issues early in the development process, increasing the risk of defects making their way into production.

Inefficient Deployment: In the absence of CI, deploying code to various environments (e.g., development, staging, production) often relies on manual processes. This can lead to inconsistencies in deployment and potential configuration errors.

Quality Assurance Challenges: Without automated testing as an integral part of the development process, ensuring software quality becomes more reliant on manual testing, making it prone to human error and subject to resource constraints.

Continuous Integration



© Copyright KodeKloud

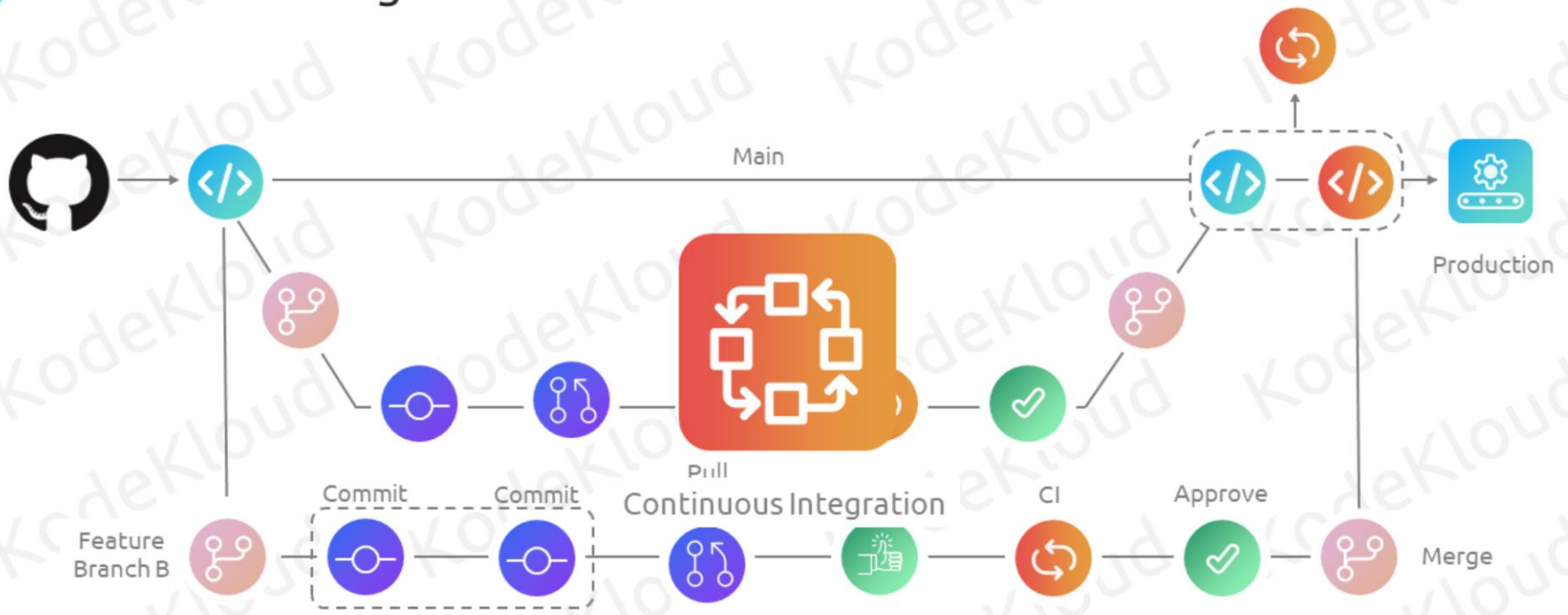
Let's imagine a scenario where Developer 1 creates a feature branch A, makes some modifications, and commits the code to this branch. A pull request is generated to merge these changes into the main branch. Before the merge, a team member reviews the code, and an automated CI pipeline is triggered.

The CI pipeline proceeds through several stages, including unit testing, dependency scanning, artifact building, and vulnerability code scanning. All these assessments are performed on both the newly added code and the existing code from the main branch.

If any of these tests fail, the developer is asked to make necessary adjustments and commit the changes to the same pull request. This action triggers the CI pipeline once again. If there are no failures this time, the pull request is approved and merged into the main branch.

Upon merging into the main branch, the same CI pipeline, or possibly a different one with additional steps and tests, is automatically executed to verify the merged code. At this point testing the same code again after merging may seem redundant, but I will try to clarify this by the end of this video.

Continuous Integration



© Copyright KodeKloud

While these developments are taking place in feature-branch A, Developer 2 is progressing on feature branch B. They commit their changes and create a pull request. An automated CI pipeline is initiated, testing the code committed in this branch. Once the CI pipeline successfully completes, the pull request is approved and merged into the main branch.

Following the successful merge, the main branch now contains code changes from both feature-branch A and B.

As mentioned earlier, any merges into the main branch automatically trigger a CI pipeline. In this instance, it once again runs all the tests, including unit testing, dependency scanning, artifact building, and vulnerability scans. This ensures that the code changes from both feature-branch B and A work seamlessly together.

This entire process, which enables multiple developers to work on the same application while ensuring that these new changes integrate smoothly without introducing any new issues, is known as continuous integration.

Continuous Deployment/Delivery

The diagram illustrates a Continuous Deployment/Delivery (CD) workflow. It starts with a GitHub repository (black cat icon) connected to a code editor (blue </> icon). A branch labeled 'Main' leads to a code editor (orange </> icon). From the 'Main' branch, a 'Feature Branch C' is created (pink icon with a branch symbol). The workflow proceeds through 'Commit' (blue icon with a circle and line), 'Pull Request' (blue icon with two arrows), 'Review' (green icon with a checkmark), 'CI' (orange icon with a circular arrow), and 'CD' (orange icon with a circular arrow). The 'CD' step is followed by a green checkmark icon, indicating a successful build. The workflow then branches into three paths: 'Deploy Prod' (orange icon with a circular arrow), 'Deploy Staging' (orange icon with a circular arrow), and 'Manual Approval' (green icon with a thumbs up). The 'Deploy Prod' path includes a 'Smoke Test' (blue icon with a checkmark) and 'Continuous Deployment' (blue icon with a circular arrow). The 'Deploy Staging' path includes 'Continuous Deployment' (blue icon with a circular arrow). The 'Manual Approval' path includes 'Deploy Prod' (orange icon with a circular arrow), 'Smoke Test' (blue icon with a checkmark), and 'Continuous Deployment' (blue icon with a circular arrow). The 'Continuous Deployment' step is represented by a blue icon with a circular arrow. The 'Manual Approval' step is represented by a green icon with a thumbs up. The 'Smoke Test' step is represented by a blue icon with a checkmark. The 'Continuous Deployment' step is represented by a blue icon with a circular arrow. The 'Manual Approval' step is represented by a green icon with a thumbs up. The 'Smoke Test' step is represented by a blue icon with a checkmark. The 'Continuous Deployment' step is represented by a blue icon with a circular arrow.

© Copyright KodeKloud

So far, we have explored the concept of continuous integration. Now, let's check into the CD aspect, which encompasses continuous deployment and/or continuous delivery.

In previous instances, after code integration into the main branch and successful completion of the CI pipeline, manual deployment to the production environment was performed.

production deployment.

Within the feature branch, following a successful CI pipeline run, we can establish another continuous deployment pipeline. This pipeline is responsible for deploying the modified code to a staging or development environment. Following deployment, a series of tests are executed to ensure the quality of the application.

Upon successful completion of CD, the pull request is approved and merged back into the main branch.

Within the main branch, the CI pipeline is triggered, assessing the newly merged changes. If successful, it automatically initiates the CD pipeline, resulting in the deployment of the application to the production environment. This automatic deployment process following successful continuous integration is referred to as continuous deployment.

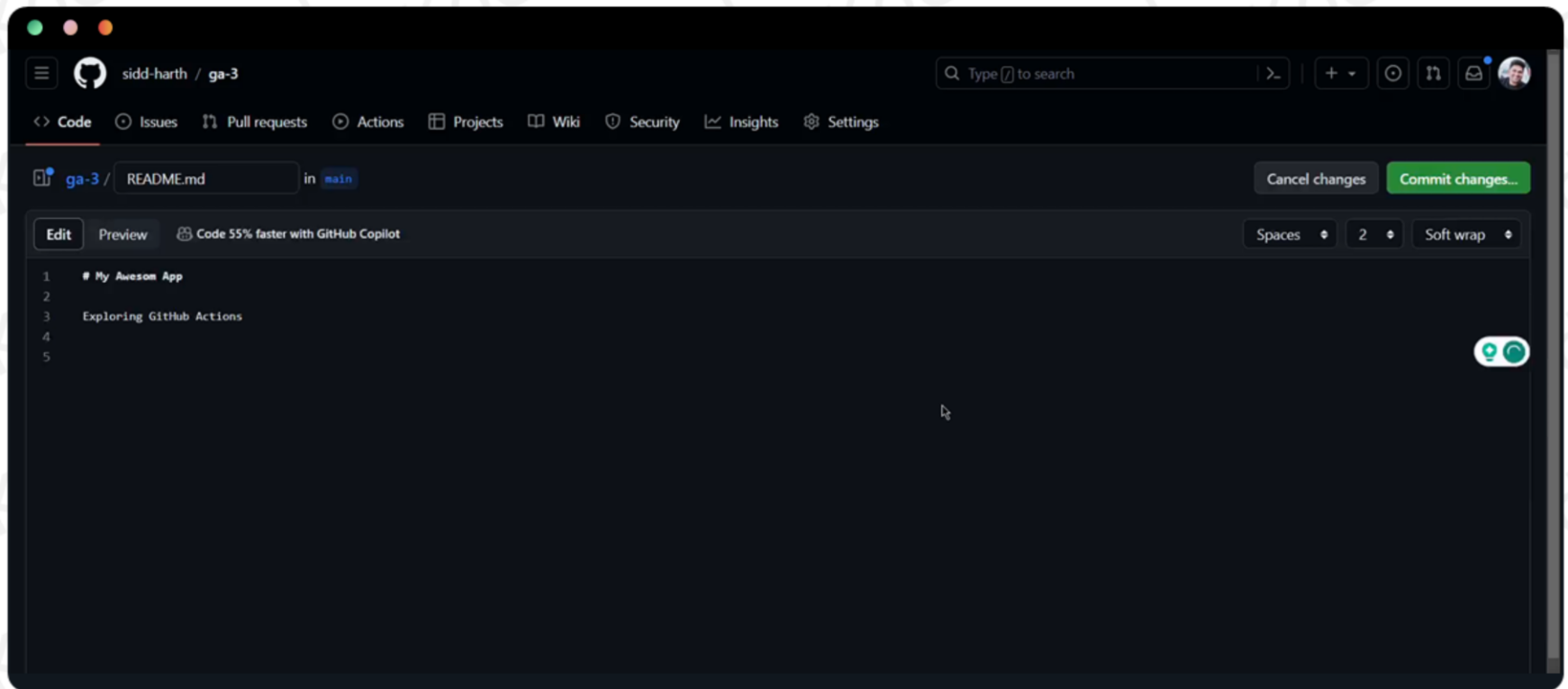
In certain instances, a manual approval step before production deployment is a critical aspect of the deployment process. This step serves to minimize risks, ensure quality, adhere to compliance requirements, and effectively coordinate changes. It offers a safety net and allows for human judgment and oversight within an otherwise automated process.

In this scenario, following the successful completion of the CI pipeline, the CD pipeline awaits human approval before proceeding with the production deployment. This process of manual approval prior to production deployment is known as continuous delivery.



GitHub Actions – Introduction

GitHub Actions



© Copyright KodeKloud

Lets assume an organization has decided to use Github as they code repository and and is in search of an automation solution.

There are numerous tools available in the market to automate CICD pipelines. Nevertheless, given the organization's commitment to GitHub as the code repository, let's **explore** how GitHub Actions offers the automation capabilities.

So what is github actions?

directly from their repositories.

You can quickly create workflows to implement a CI/CD pipeline that build, test on every pull request, and deploy merged pull requests to production right beside your code base.

So what is github actions?

GitHub Actions is a powerful and flexible automation platform provided by GitHub. It allows developers to **automate tasks** directly from their repositories.

You can quickly create workflows to implement a CI/CD pipeline that build, test on every pull request, and deploy merged pull requests to production right beside your code base. Out of the box, these steps can be executed in different operating systems like ubuntu, windows and macos.

GitHub Actions



Ubuntu



Windows



MacOS

Out of the box, these steps can be executed in different operating systems like ubuntu, windows and macos.

GitHub Manages Infrastructure

01



Setting up
servers

02



Scaling
resources

03



Managing
execution
environment

Another advantage of GitHub Actions is that GitHub manages the infrastructure for you, which includes setting up servers, scaling resources, and managing the execution environment for your workflows.

GitHub Handles the Rest

01



Tasks in virtual environments

02



Caching necessary dependencies

03



Providing reports on the outcomes

Your task is to write workflow configurations in YAML files, and GitHub Actions handles the rest.

This includes executing your tasks within virtual environments, caching necessary dependencies, and providing reports on the outcomes.

Automation in GitHub

01



Streamline
development

02



Reduce manual
errors

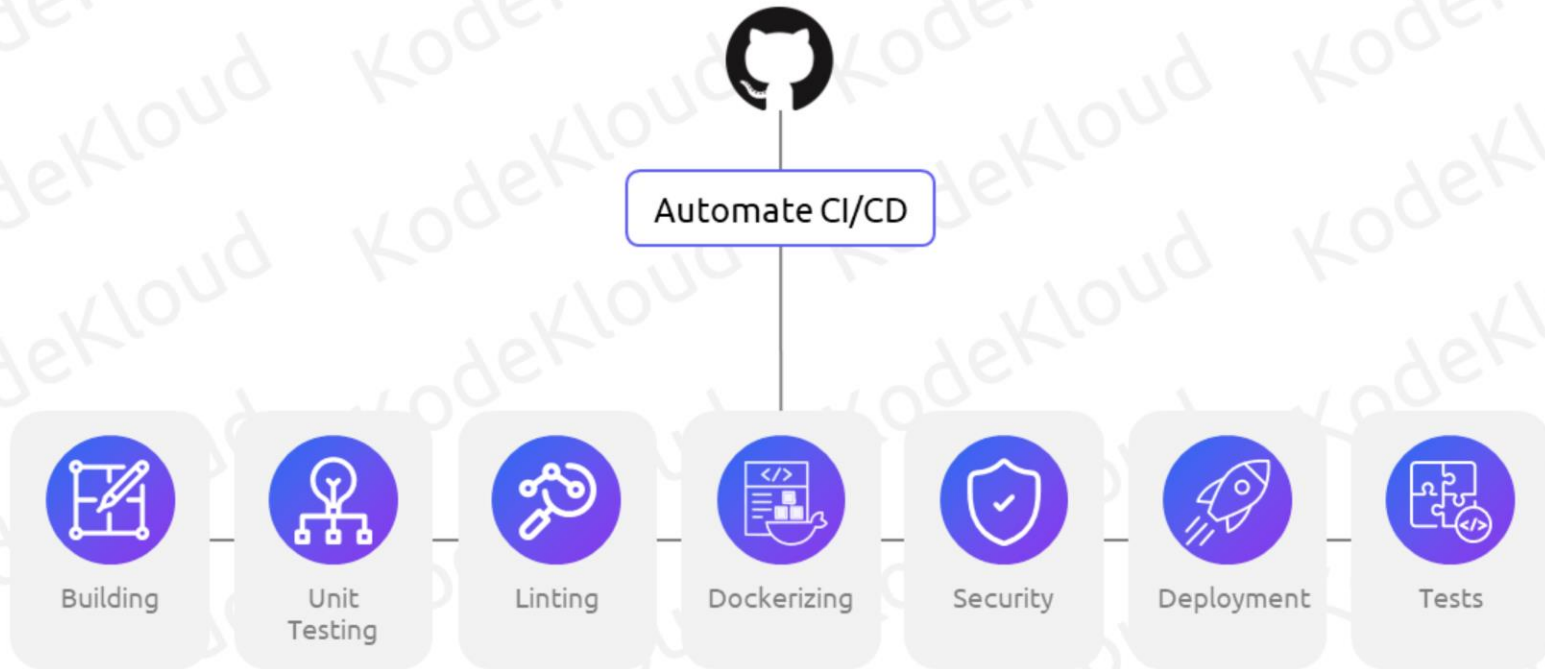
03



Increase
efficiency

Finally, this automation helps streamline development, reduce manual errors, and increase the efficiency of your software development workflow.

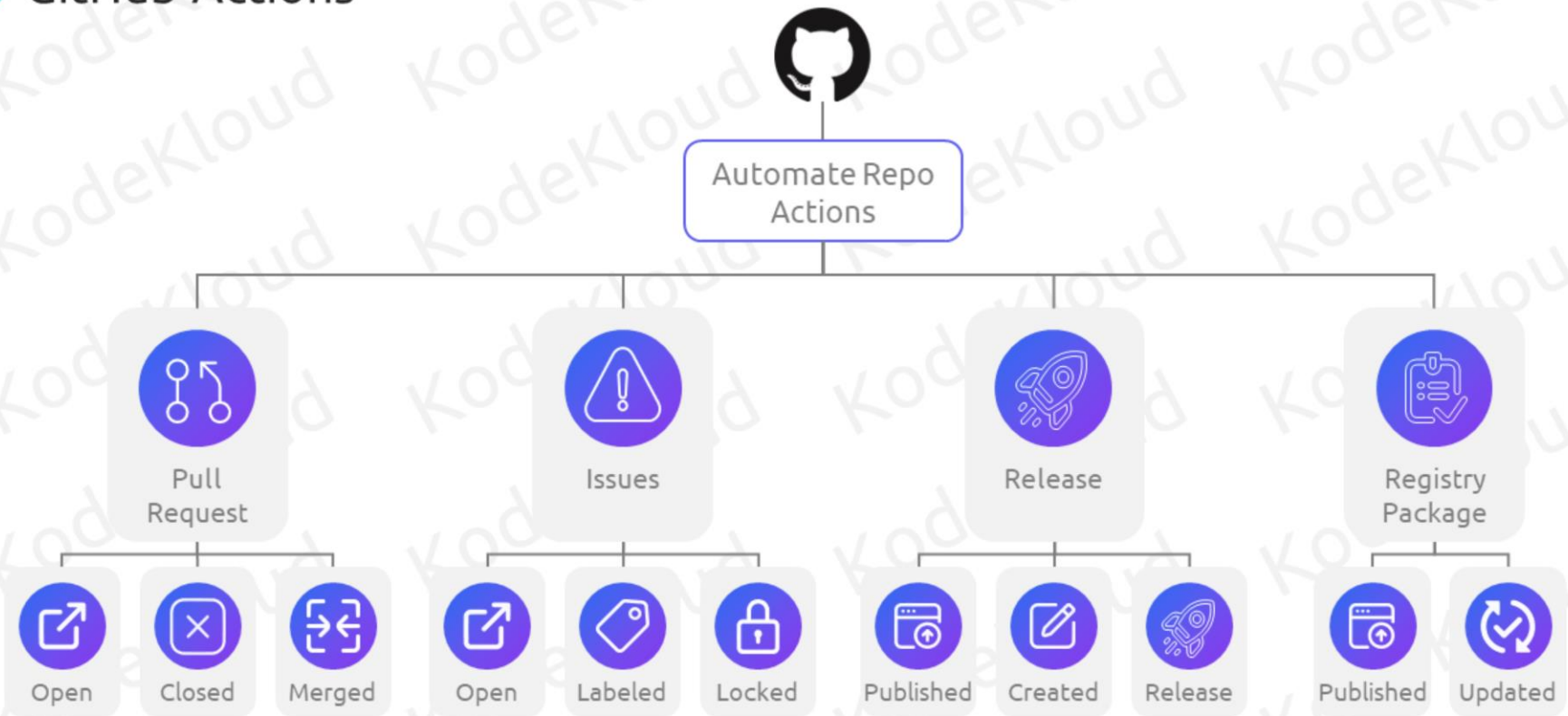
GitHub Actions



© Copyright KodeKloud

Is GitHub Actions solely used for automating CI/CD pipelines for building, testing, and releasing code?

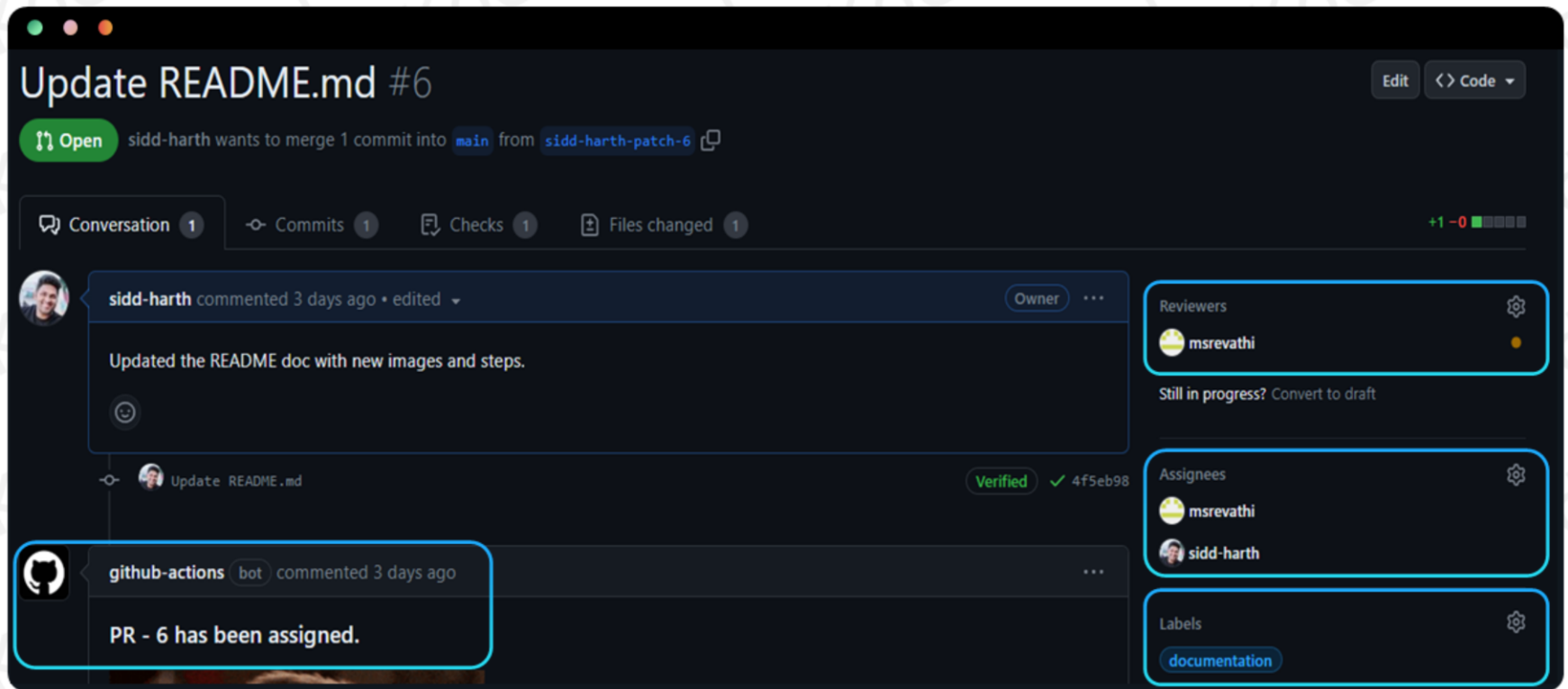
GitHub Actions



© Copyright KodeKloud

The answer is No, github actions goes beyond just CI/CD and lets you run workflows when other events happen in repository. It can automate all kinds of repository actions by listening to events such as push events, issues, pull requests, github registry-packager, or whenever a issues or pull requests get updated etc.,

GitHub Actions



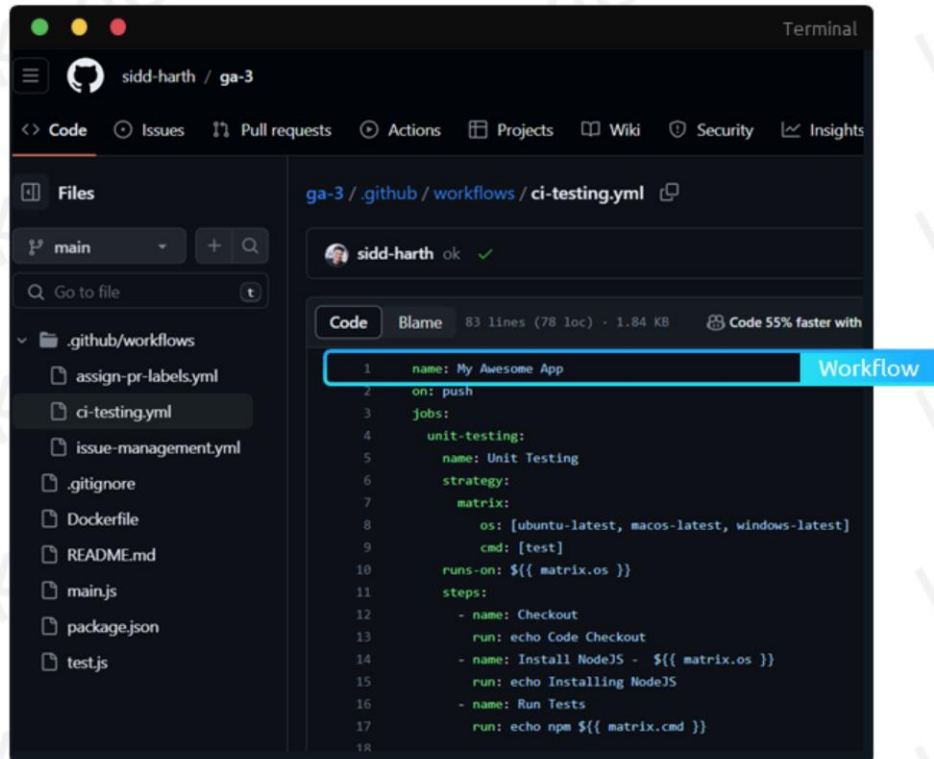
© Copyright KodeKloud

For instance, let's consider a scenario where a project contributor initiates a new pull request. By utilizing GitHub Actions, you can automatically post informative comments, apply labels, assign contributors, and even add reviewers to assess and approve the proposed changes.

These automation capabilities aren't limited solely to pull requests; you can employ similar actions for various other GitHub events.

Now that we've established the fundamentals of GitHub Actions, let's understand how and where these tasks are executed.

Workflow, Job, and Runner



The screenshot shows a GitHub repository interface for a user named 'sidd-harth' in a repository named 'ga-3'. The left sidebar displays the file explorer with the directory structure: `.github/workflows` (expanded), showing files like `assign-pr-labels.yml`, `ci-testing.yml` (selected), `issue-management.yml`, `.gitignore`, `Dockerfile`, `README.md`, `main.js`, `package.json`, and `test.js`. The main content area shows the selected file `ga-3 / .github / workflows / ci-testing.yml`. The file content is displayed in a code editor with a dark theme. A blue callout box labeled 'Workflow' points to the first line of the YAML file. The file content is as follows:

```
1 name: My Awesome App
2 on: push
3 jobs:
4   unit-testing:
5     name: Unit Testing
6     strategy:
7       matrix:
8         os: [ubuntu-latest, macos-latest, windows-latest]
9         cmd: [test]
10    runs-on: ${{ matrix.os }}
11    steps:
12      - name: Checkout
13        run: echo Code Checkout
14      - name: Install NodeJS - ${{ matrix.os }}
15        run: echo Installing NodeJS
16      - name: Run Tests
17        run: echo npm ${{ matrix.cmd }}
```

© Copyright KodeKloud

The core component of GitHub Actions is the workflow.

A workflow is an automated process capable of executing one or more tasks. These workflows are defined using YAML files and are located in the `.github/workflows` directory within your repository. A repository can have multiple workflows, each of which runs in response to specific events occurring in your repository.

For instance, in this example, a simple event like pushing code to the repository triggers the workflow.

Within each workflow, you define one or more jobs. A **job** consists of a series of individual steps, which are executed on a runner.

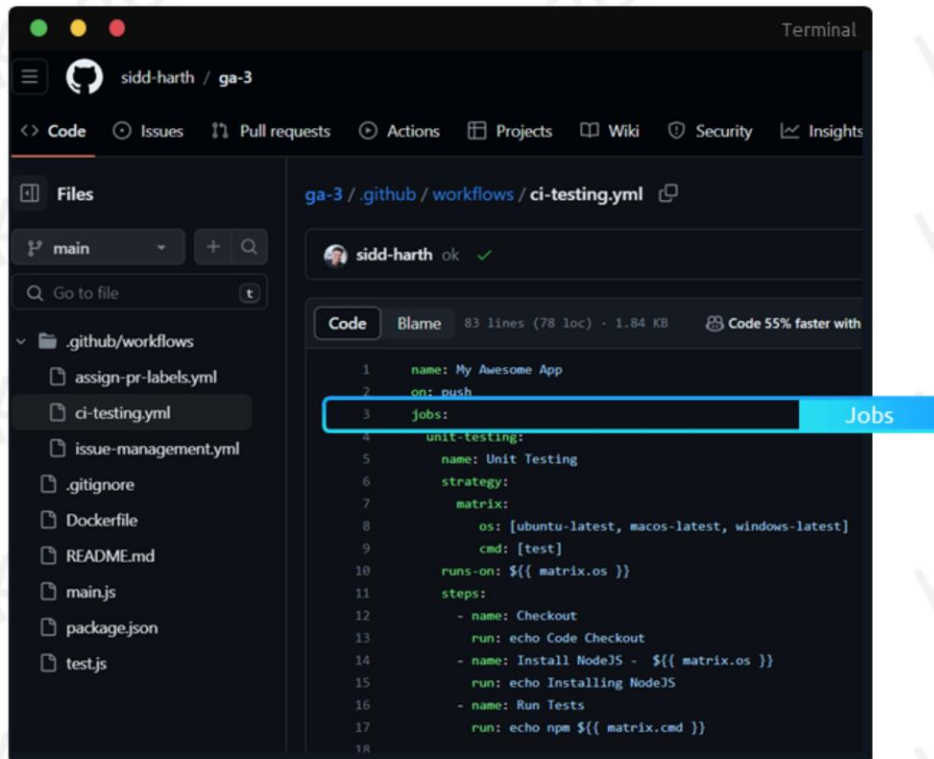
A runner is a virtual machine (VM) responsible for executing your workflows upon triggering. GitHub automatically provisions runners for each job based on the runs-on configuration specified.

In this example workflow, there's a job named 'unit-testing' that runs on three different machines: Windows, Ubuntu, and macOS. Consequently, GitHub will provision three VMs for these jobs. These VMs are known as GitHub Hosted Runners.

GitHub hosts Linux and Windows runners using Microsoft Azure, while macOS runners are hosted in GitHub's own macOS Cloud.

In this particular workflow example, all the runners are provisioned and operated in parallel.

Workflow, Job, and Runner



The screenshot shows a GitHub repository interface. On the left, the 'Files' sidebar lists the repository structure, including a '.github/workflows' directory. The file 'ci-testing.yml' is selected. The main content area displays the YAML code for this workflow. A blue box highlights the 'jobs' section, and a blue callout bubble with the word 'Jobs' points to it. The workflow is named 'My Awesome App' and is triggered on a 'push' event. It contains a single job named 'unit-testing' that runs on a matrix of operating systems (ubuntu-latest, macos-latest, windows-latest) and executes a series of steps: checkout, install NodeJS, and run tests.

```
1 name: My Awesome App
2 on: push
3 jobs:
4   unit-testing:
5     name: Unit Testing
6     strategy:
7       matrix:
8         os: [ubuntu-latest, macos-latest, windows-latest]
9         cmd: [test]
10    runs-on: ${{ matrix.os }}
11    steps:
12      - name: Checkout
13        run: echo Code Checkout
14      - name: Install NodeJS - ${{ matrix.os }}
15        run: echo Installing NodeJS
16      - name: Run Tests
17        run: echo npm ${{ matrix.cmd }}
```

© Copyright KodeKloud

The core component of GitHub Actions is the workflow.

A workflow is an automated process capable of executing one or more tasks. These workflows are defined using YAML files and are located in the `.github/workflows` directory within your repository. A repository can have multiple workflows, each of which runs in response to specific events occurring in your repository.

For instance, in this example, a simple event like pushing code to the repository triggers the workflow.

runner.

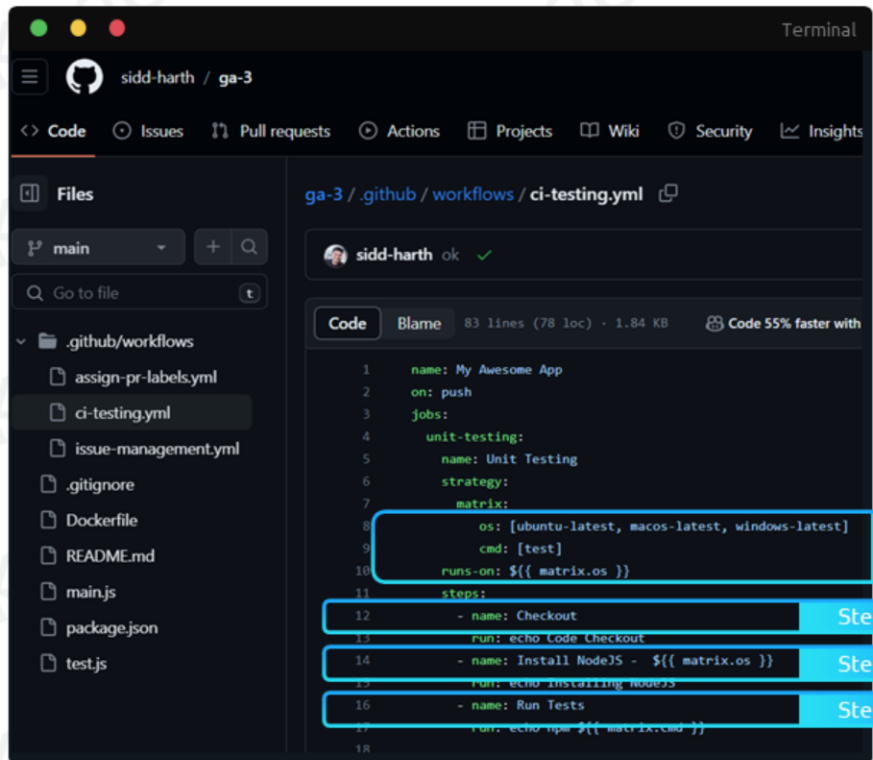
A runner is a virtual machine (VM) responsible for executing your workflows upon triggering. GitHub automatically provisions runners for each job based on the runs-on configuration specified.

In this example workflow, there's a job named 'unit-testing' that runs on three different machines: Windows, Ubuntu, and macOS. Consequently, GitHub will provision three VMs for these jobs. These VMs are known as GitHub Hosted Runners.

GitHub hosts Linux and Windows runners using Microsoft Azure, while macOS runners are hosted in GitHub's own macOS Cloud.

In this particular workflow example, all the runners are provisioned and operated in parallel.

Workflow, Job, and Runner

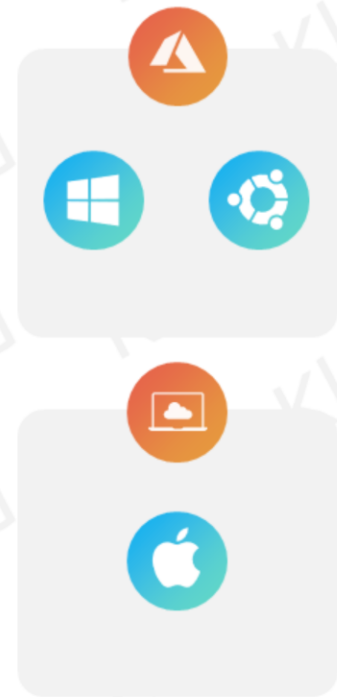


The screenshot shows a GitHub repository interface for a user named 'sidd-harth' in a repository named 'ga-3'. The 'Actions' tab is selected, displaying a workflow file named 'ci-testing.yml' located in the '.github/workflows' directory. The workflow file content is as follows:

```
1 name: My Awesome App
2 on: push
3 jobs:
4   unit-testing:
5     name: Unit Testing
6     strategy:
7       matrix:
8         os: [ubuntu-latest, macos-latest, windows-latest]
9         cmd: [test]
10    runs-on: ${{ matrix.os }}
11  steps:
12    - name: Checkout
13      run: echo Code Checkout
14    - name: Install NodeJS - ${{ matrix.os }}
15      run: echo installing nodejs
16    - name: Run Tests
17      run: echo npm ${{ matrix.cmd }}
```

On the right side of the workflow file, there are three blue callout boxes with white text: 'Runner' points to the 'runs-on' line, and 'Steps' points to the 'steps' section.

GitHub-Hosted Runners



The core component of GitHub Actions is the workflow.

A workflow is an automated process capable of executing one or more tasks. These workflows are defined using YAML files and are located in the `.github/workflows` directory within your repository. A repository can have multiple workflows, each of which runs in response to specific events occurring in your repository.

For instance, in this example, a simple event like pushing code to the repository triggers the workflow.

runner.

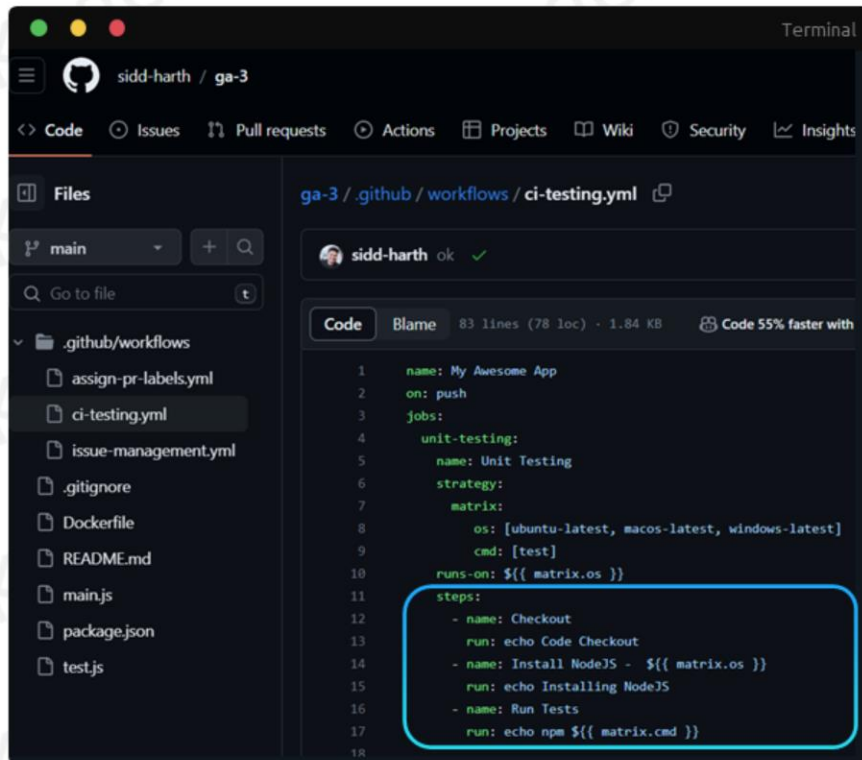
A runner is a virtual machine (VM) responsible for executing your workflows upon triggering. GitHub automatically provisions runners for each job based on the runs-on configuration specified.

In this example workflow, there's a job named 'unit-testing' that runs on three different machines: Windows, Ubuntu, and macOS. Consequently, GitHub will provision three VMs for these jobs. These VMs are known as GitHub Hosted Runners.

GitHub hosts Linux and Windows runners using Microsoft Azure, while macOS runners are hosted in GitHub's own macOS Cloud.

In this particular workflow example, all the runners are provisioned and operated in parallel.

Workflow, Job, and Runner



```
1 name: My Awesome App
2 on: push
3 jobs:
4   unit-testing:
5     name: Unit Testing
6     strategy:
7       matrix:
8         os: [ubuntu-latest, macos-latest, windows-latest]
9     cmd: [test]
10    runs-on: ${{ matrix.os }}
11    steps:
12      - name: Checkout
13        run: echo Code Checkout
14      - name: Install NodeJS - ${{ matrix.os }}
15        run: echo Installing NodeJS
16      - name: Run Tests
17        run: echo npm ${{ matrix.cmd }}
```

© Copyright KodeKloud



After the runners are up and running, they start the execution of the steps defined within the job.

In this example workflow, there are three steps and steps are executed in series.

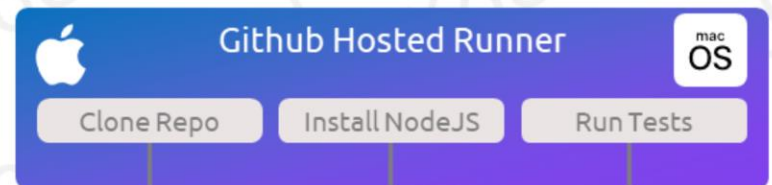
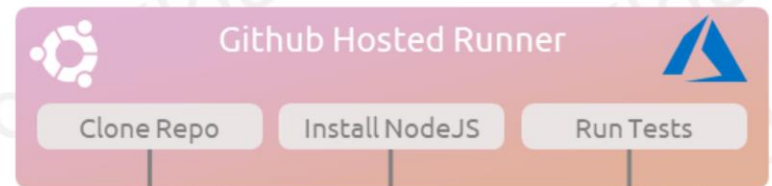
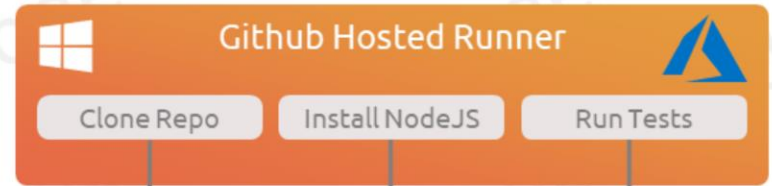
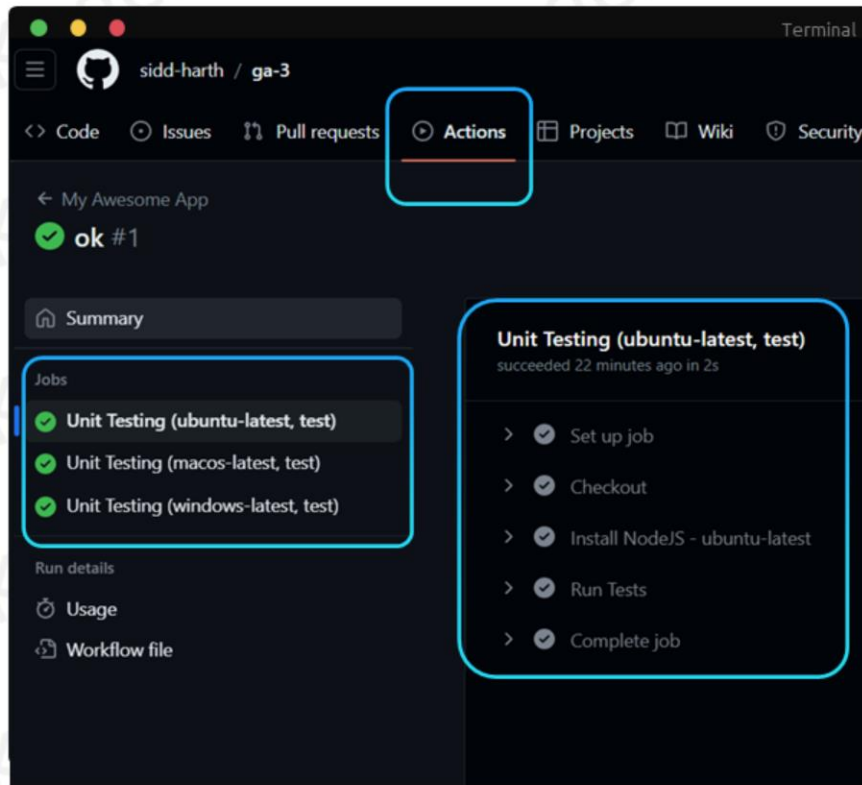
Firstly, the runners will clone the repository locally.

Then, they proceed to install Node.js and set up the necessary environment.

These same steps are executed on all three runners concurrently. Consequently, if one runner completes all the step executions, it will be marked as successful, even if the other runners are still in the process of executing their remaining steps.

The workflow as a whole is marked as successful only once all three runners have completed their tasks successfully.

Workflow, Job, and Runner



During and after the job's execution, you can access the logs, output, and any artifacts for each step via the GitHub User Interface.

This information is received from the runners where the jobs get executed

Within the Github repository, you can simply navigate to the Actions Tab to inspect the logs and artifacts.

Here, you'll find that the 'unit-testing' job has been executed on three different machines, and you can review the logs for each of them individually. This allows for detailed debugging of the workflow's progress and results.

If the job generates any artifacts, you can access and download them from this page as well. GitHub Actions provides a convenient interface for managing and retrieving any artifacts produced during the workflow's execution, enhancing the overall visibility and usability of your automated processes.

Runner Types



GitHub-hosted Runner



GitHub hosted and maintained



Provides a clean VM for every job execution



Cannot customize runners beyond selecting the type of runner (e.g., Ubuntu, Windows, macOS)



Part of your GitHub subscription with usage limits. Incurs additional charges when exceeding limits

Workflow

Jobs



Self-hosted Runner



run on your own infrastructure, such as your own servers, virtual machines, or even in cloud



Can run multiple jobs on the same machine



You have complete control over the runner's environment and can install any required software



You bear the maintainance and operation cost for your own infrastructure for self-hosted runners

Steps

Runners

There are two types of runners: GitHub-hosted runners and self-hosted runners. Let's explore the key differences between them:

GitHub-hosted runner: are hosted and maintained by GitHub. GitHub provides a set of virtual machines with various configurations, including different operating systems and software environments. You don't need to worry about managing the infrastructure for these runners.

Self-hosted runner: run on your own infrastructure, such as your own servers, virtual machines, or even cloud. You have full control over the environment, but you are responsible for setting up and maintaining these runners.

GitHub-hosted runner by default Provides a new clean instance for every job execution.

Self-hosted runner: can run multiple jobs on the same machine

within GitHub-hosted runner: You cannot customize the runners beyond selecting the type of runner (e.g., Ubuntu, Windows, macOS). You can't install additional software or make system-level changes. whereas in **Self-hosted runner** You have complete control over the self-hosted runner's environment. You can install any required software, configure the runner to your needs, and even use it to run workflows that require specific hardware or software configurations.

When it comes to the cost or pricing perspective,

•**GitHub-hosted runner:** are provided as part of your GitHub subscription, but there are usage limits. If your usage exceeds these limits, you may incur additional charges.

in Self-hosted runner: You bear the costs of maintaining and operating your own infrastructure

Choosing between GitHub-hosted runners and self-hosted runners depends on factors like your workflow requirements, security considerations, and infrastructure capabilities. Each has its own advantages, and the choice should align with your specific needs.

Conclusion



Workflow



Jobs



Steps



Runners

In the video, I've provided a brief overview of workflows, jobs, steps, and runners. These concepts will be explored in greater detail in our upcoming sessions.



KodeKloud

© Copyright KodeKloud

Follow us on <https://kodekloud.com/> to learn more about us.