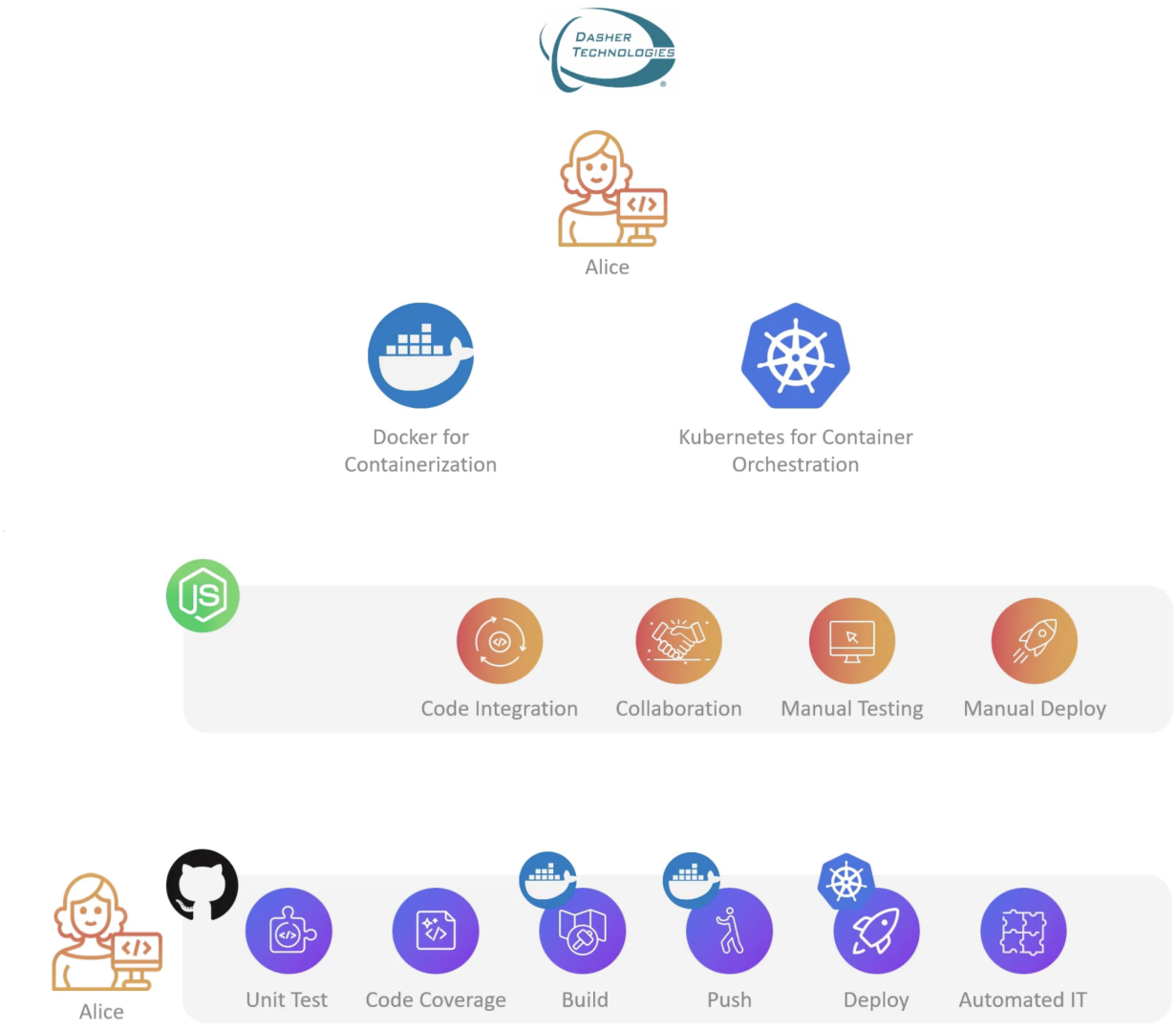
Introduction

We have a software provider which offers a platform to facilitate the connections of data applications and devices across on-premises for businesses. Recently there RD team has been exploring the possibilities of transiting their services to cloud and leveraging container technologies.

—> Alice for establishing the DEVOPS pipeline for the project right from it’s inception.



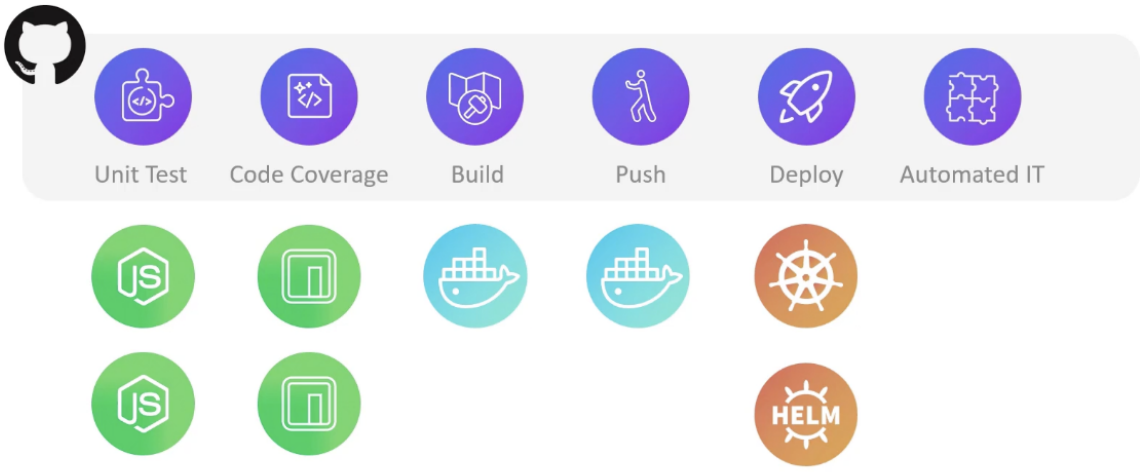
```
// Problem statement
to a node JS project. So the previous stream operated without a
system with developers, you know, independently writing and manu
 the code.
The testing process was also sluggish and ineffective due to the

 So collaboration among developers was often hampered
as they worked on separate code branches with infrequent integra
testings. And hence the software releases carried more significa
So deployment of software to various environments, including dev
 staging and production, was primarily a manual procedure. So to
challenges, Alice
and her team have opted to implement a continuous integration or
pipeline and have outlined the followina kev steps.
```

```
//solution
So they want to adopt, uh, GitHub for version control and develo
implementation of unit testing and code coverage measures to exp
and minimizing bugs, utilization of docker build and push proces
containerization. And the application is deployed to Kubernetes
or they also plan to, you know, incorporate automation like auto
testing as a final step. The successful execution of these steps
Resolve the existing issues, and nevertheless, the team now face
 hurdle of selecting the most suitable CI ICD tool.
So in the market, a wide array of CI ICD tools are available, wh
 Jenkins, driver CI, CircleCI, babmo CI
```

AS Jenkins is most popular for open source, the team had made the decision to proceed with Jenkins as it stands out as one of the most extensively used open source CICD tools. So to employ Jenkins, the team is required to undertake several tasks. The first thing is to, you know, do configuration operations and upkeep of a virtual machine with specific CPU memory and HTD capacity. And prior to Jenkins installation prerequisites includes Java, JDK installation, setting up firewall rules, and installing the Jenkins plugins. And finally, given that the project is centered around node jers, so node JERS and NPM should also be installed. So to facilitate unit testing with various node JS versions, the DevOps engineers must also install multiple node js and, uh, NPM versions packages, ensuring tests are run without any version conflicts. Docker installation is also essential for containerization for Kubernetes deployments, the installation of Cube CTL Health or other necessary binary assessment mandatory. Apart from this, the team often employs external tools for integration testing and reporting. So these installations of the tools or their CLI on the machine where Jen can execute DIM should also be done.
So it's also important to note that these tasks apply to a single, not just project with high level steps.

Traditional CI/CD Tools – Challenges



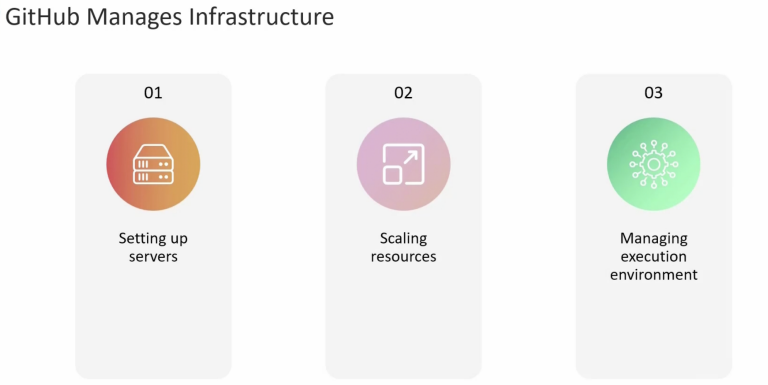
So the pipeline complexity may grow with additional steps proportionally increasing the manual configuration workload.

- simplify the setup and initiation without the need to install and configure numerous services.
- Focus on building pipelines without the burden of managing infrastructure or concerns regarding scalability.

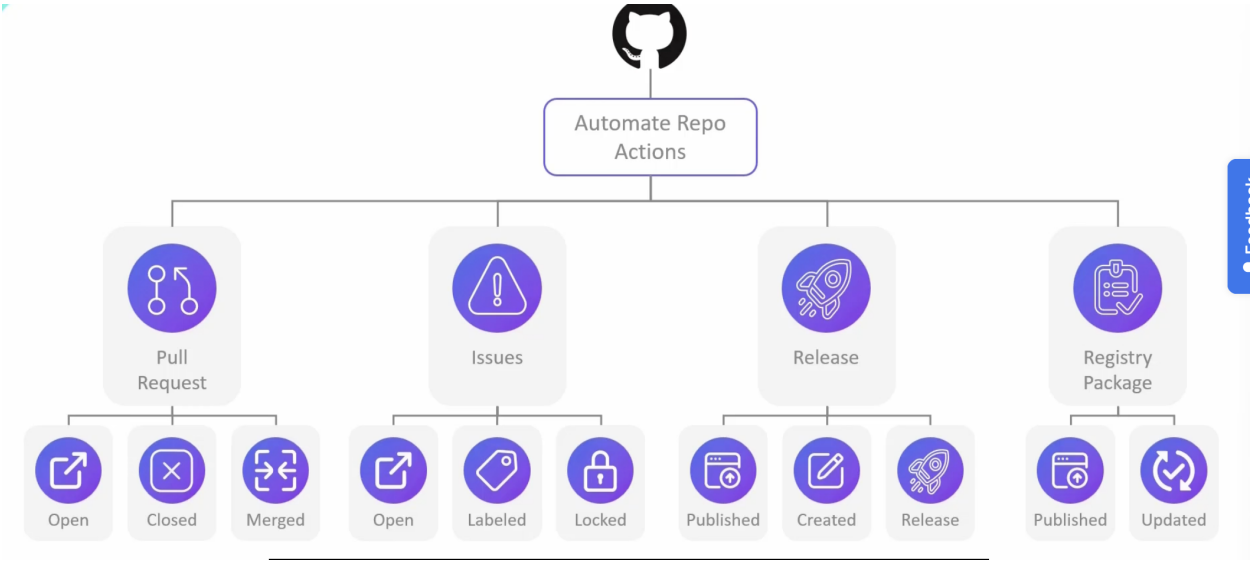
On a thorough review, we decided to use github actions:

GitHub Actions

Github actions is a powerful automation platform provided by github, we can automate tasks directly from the repositories, we can quickly create workflow to implement CICD pipelines that build test on every pull request, and deploy merged pull request to the production right beside the codebase.

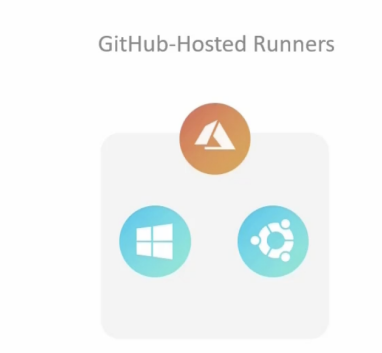


We write work flow configuration in the yaml file. They are located in .github/ workflows



```yaml
name: My Awesome App
on: push
jobs:
  unit-testing:
    name: Unit Testing
    strategy:
      matrix:
        os: [ubuntu-latest, macos-latest, windows-latest]
        cmd: [test]
    runs-on: ${{ matrix.os }}
    steps:
      - name: Checkout
        run: echo Code Checkout
      - name: Install NodeJS -  ${{ matrix.os }}
        run: echo Installing NodeJS
      - name: Run Tests
```

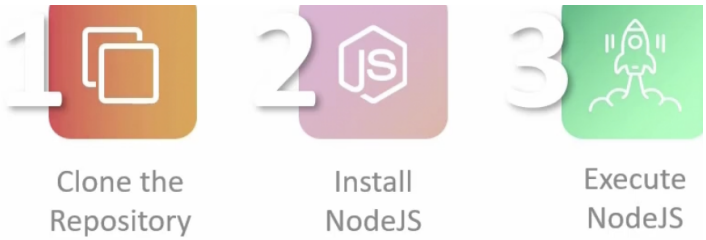
- Here a simple event like pushing code to the repository triggers the workflow, and within each workflow we define one or more jobs. A job consist of series of individual steps which are executed on a runner, and a runner is a VM responsible for executing the workflow upon triggering.



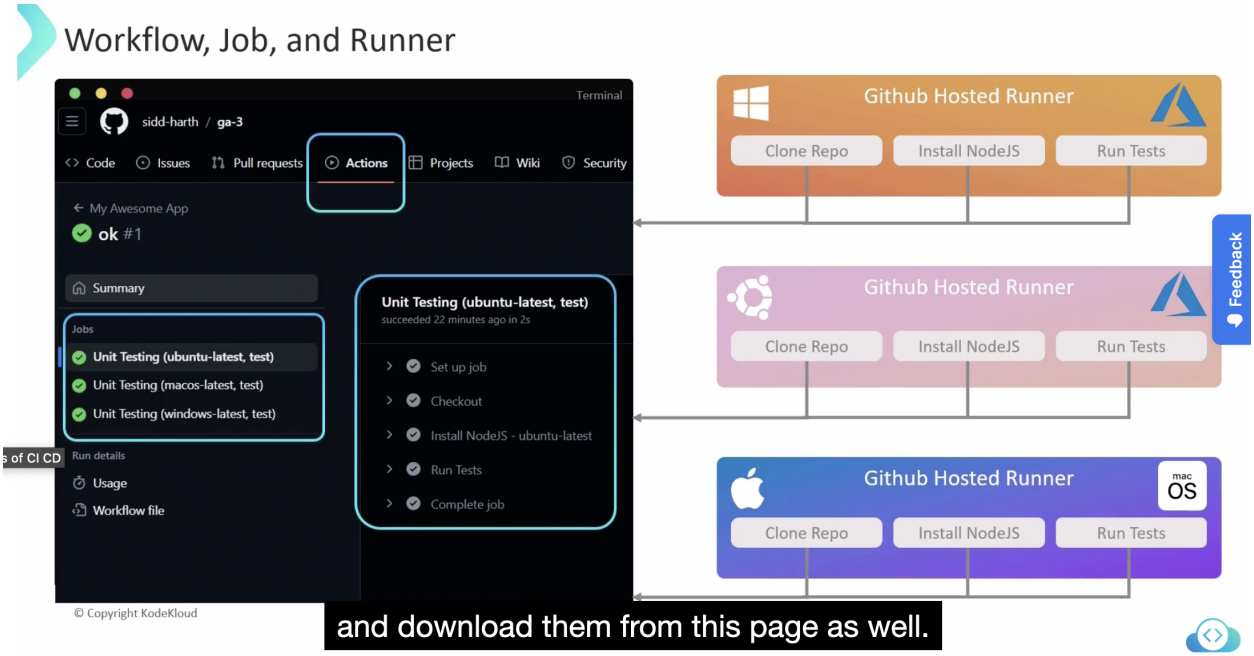
Macos are hosted on github own cloud, whereas other on microsoft azure.

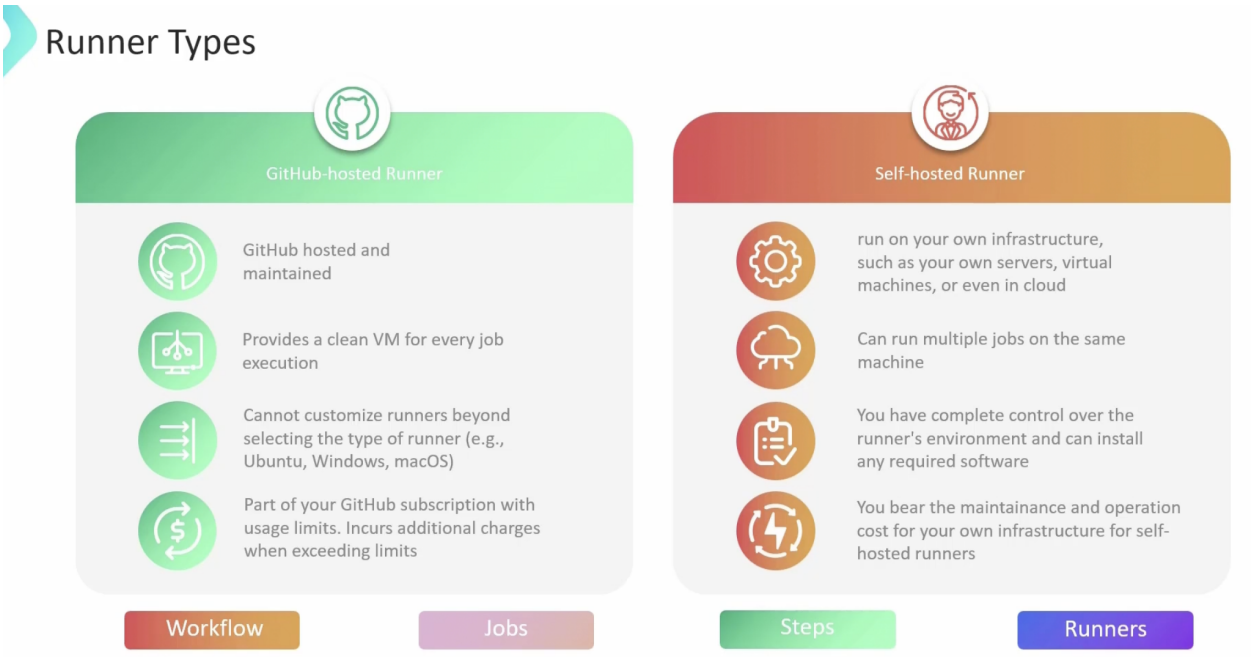
The steps are executed on series.

```yaml
unit-testing:
  name: Unit Testing
  strategy:
    matrix:
      os: [ubuntu-latest, macos-latest, windows-latest]
      cmd: [test]
  runs-on: ${{ matrix.os }}
  steps:
    - name: Checkout
      run: echo Code Checkout
    - name: Install NodeJS -  ${{ matrix.os }}
      run: echo Installing NodeJS
    - name: Run Tests
      run: echo npm ${{ matrix.cmd }}
```

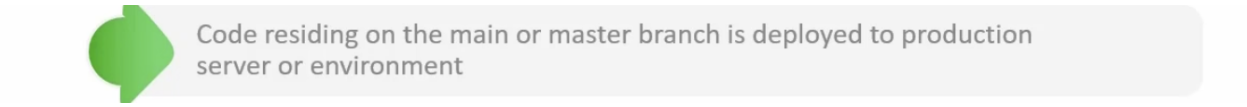


all the steps are executed concurently on all the 3 runners concurrently, and if one runner completes all the steps execution it will be marked as succesfull even if other runners are still in progress of executing the remaining steps.

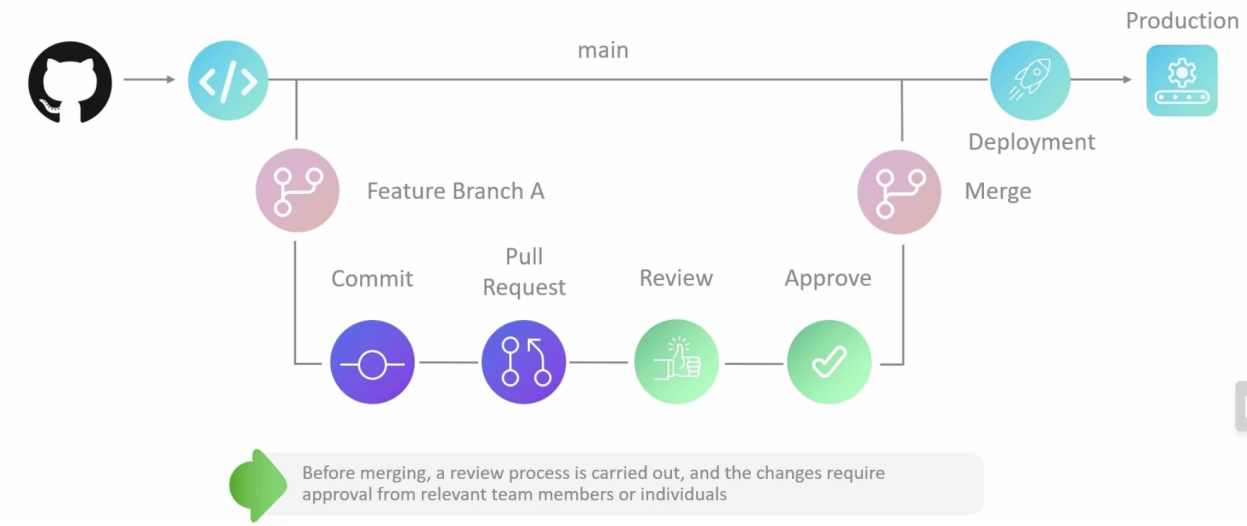




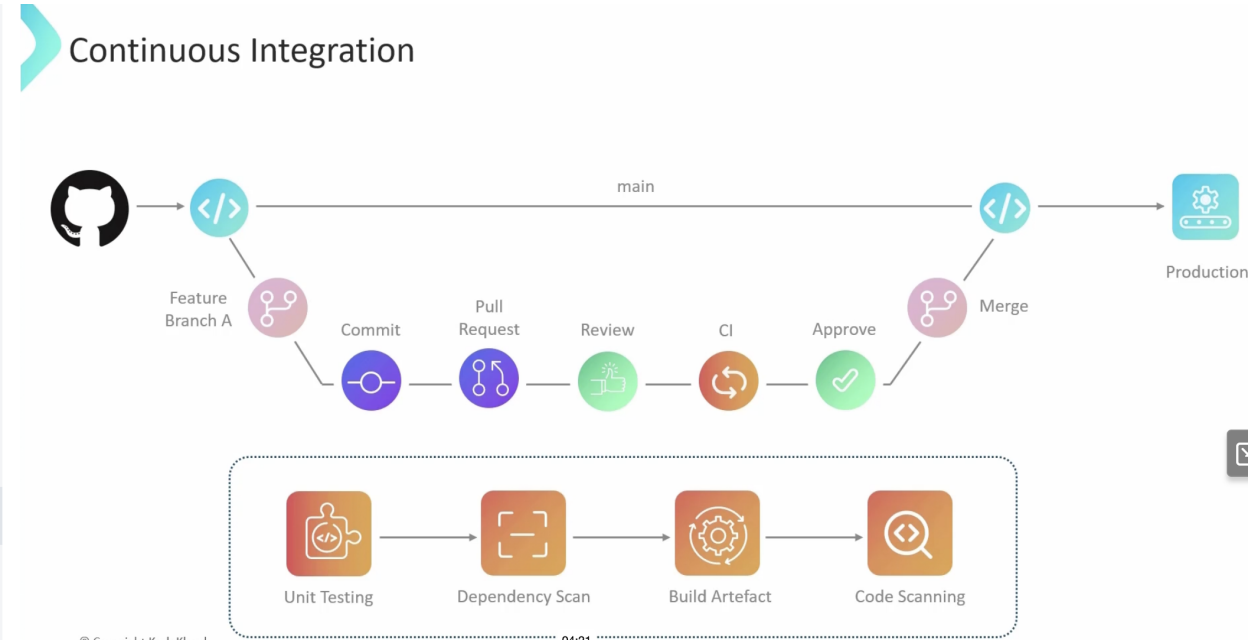
CI/CD Basics



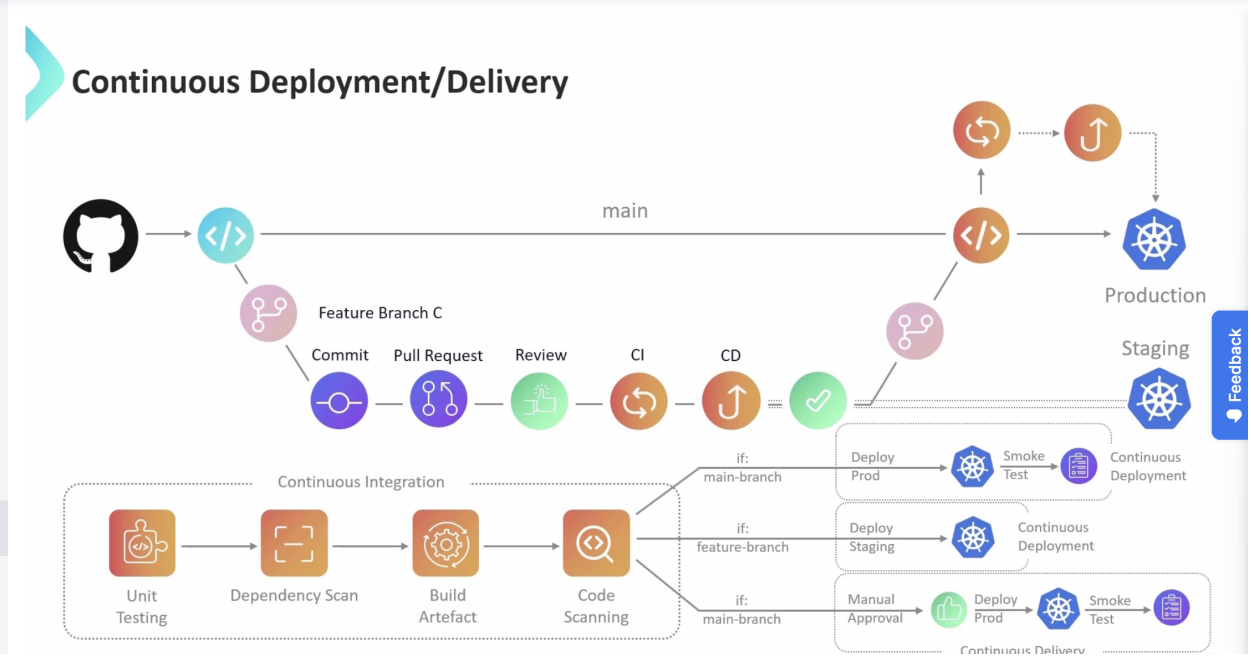
For new features, we have feature branch, which is the clone of the main base.



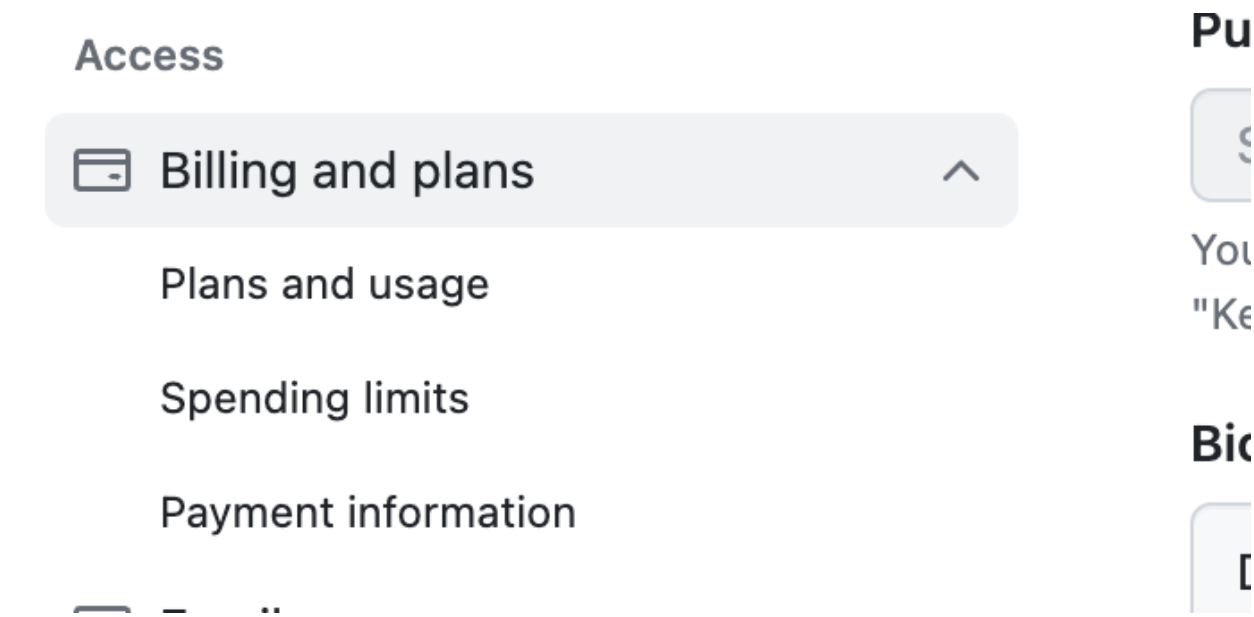
Copyright KodeKloud



- let's imagine a scenario where a developer one creates a feature branch and makes some modifications. And once the modifications are done, so the commit the code to the branch and a pull request is generated to merge these changes into the main branch. Again, before the merge, a team member reviews the code and then automated Cl pipeline is triggered. The CI pipeline proceeds through several stages, which includes unit testing, dependency scanning, uh, artifact building, and vulnerability scans. This ensures that the code changes from both feature branch A and B work together, and the entire process, which enables multiple developers to work on the same application, while ensuring that these new changes integrate smoothly Without, you know, uh, affecting the existing code or without introducing any new issues, it is called as continuous integration.



So in the previous instances, after the code integration into the main branch and successfully completion of the CI pipeline, we manually deploy it to the production environments. So in many scenarios, even after rigorous CI and the testing procedures, it is advisable to deploy the modified applications to a non- production environment that closely resembles the live environments. So this allows for live testing before proceeding with the prediction deployments. So within the feature branch itself, you know, uh, following a successful Cl pipeline, we can establish another continuous deployment pipeline. This pipeline is responsible for deploying the modified code automatically initiates the CD pipeline resulting in the deployment of the application to the production environment. This automatic deployment process, uh, following successful continuous integration is referred Continuous Deployment / Delivery.



https://github.com/settings/billing/summary

In a more laymen term, we can say that CICD will be as follows:

CI/CD, or Continuous Integration and Continuous Deployment/Delivery, is a method used in software development to more efficiently manage and deploy code changes. Here's a simplified explanation using the scenario you described:

Continuous Integration (CI): This happens when developers work on different parts of an application in separate branches (like feature branch A and B). They make changes and then commit (save) these changes to their branches. When they're ready to combine their changes with the main application code (the main branch), they create a pull request. This is like saying, "I think my code is ready to be added to the main project."

Before their code is merged into the main branch, it goes through several automated checks in the CI pipeline. This includes:

Unit testing: Checking small parts of the code to make sure they work as expected.
Dependency scanning: Making sure all the external code or libraries the application needs are secure and up-to-date.
Artifact building: Creating the runnable parts of the application from the code.
Vulnerability scans: Looking for any security weaknesses in the code.

These steps ensure that the new code works well with the existing code and doesn't introduce problems. This process lets multiple developers work on the same application simultaneously, smoothly integrating their changes without messing up the existing work.

Continuous Deployment/Delivery (CD): After the CI process, where code is integrated into the main branch and checked, the next step is to get this code into the actual production environment (where users can use the application). In many cases, it's a good idea to first deploy these changes to a non-production environment that's very similar to the live environment. This allows for additional testing in conditions that closely match where the application will eventually run.

If everything checks out, the CD pipeline takes over. This is an automated process that deploys the code to the production environment. So, after the CI has ensured the code is good to go, the CD automatically gets this code out to the actual users.

In short, CI/CD is all about making the process of adding new code to a software project as smooth, efficient, and safe as possible, with lots of automated checks along the way to ensure quality.