GitLab

# Intro to GitLab CI/CD

For teams Getting Started with GitLab CI/CD

## Agenda

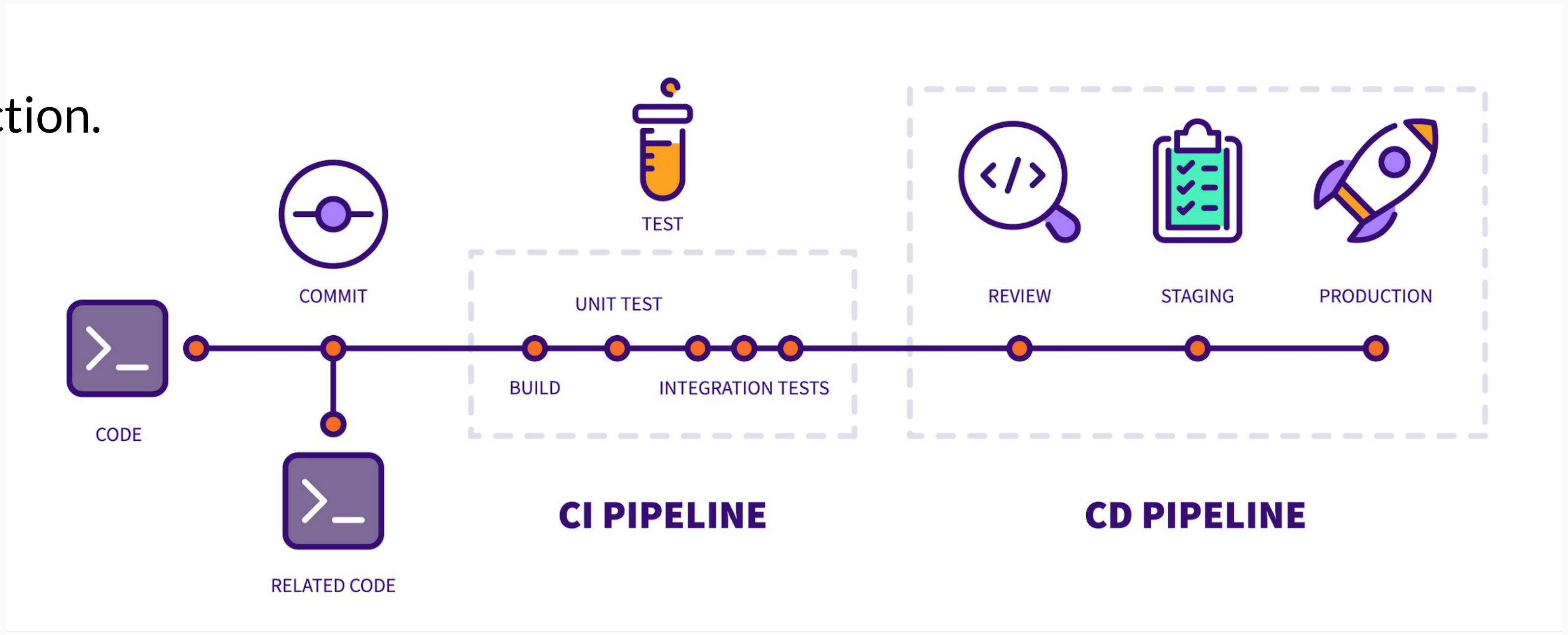
- What is CI/CD?
- GitLab CI/CD Overview
- GitLab CI/CD Setup
- GitLab CI/CD Runners
- Q&A

## What is GitLab?

- DevOps Lifecycle Tool
  - Bridge dev and Ops.
- Git Repository Manager
  - Integrate code provided by your team in a shared repository.
- CI/CD
  - Empowers all teams to work together  efficiently.
  - Powerful, scalable, end-to-end automation.

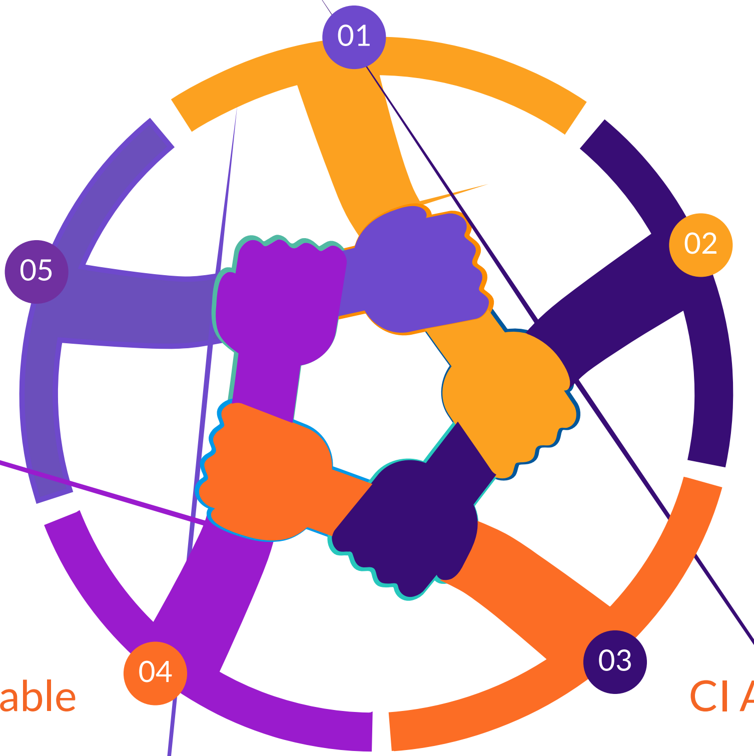
## What is CI/CD?

- Continuous Integration (CI)
  - Integrate code provided by your team in a shared repository.
- Continuous Delivery
  - Software released to production  automatically.
- Continuous Deployment
  - Pushes changes to production.



## Why use CI/CD?

CI/CD encourages collaboration across all departments and makes code creation and management easy, as well as provides the following specific benefits.



### CI Detects Errors Quickly

Fix errors while they are fresh in your mind

### CI Reduces Integration Problems

Smaller problems are easier to digest and it ensures the problems don't compound

### CI Allows Teams to Develop Faster

More confidence among the developers allows for less bottlenecking

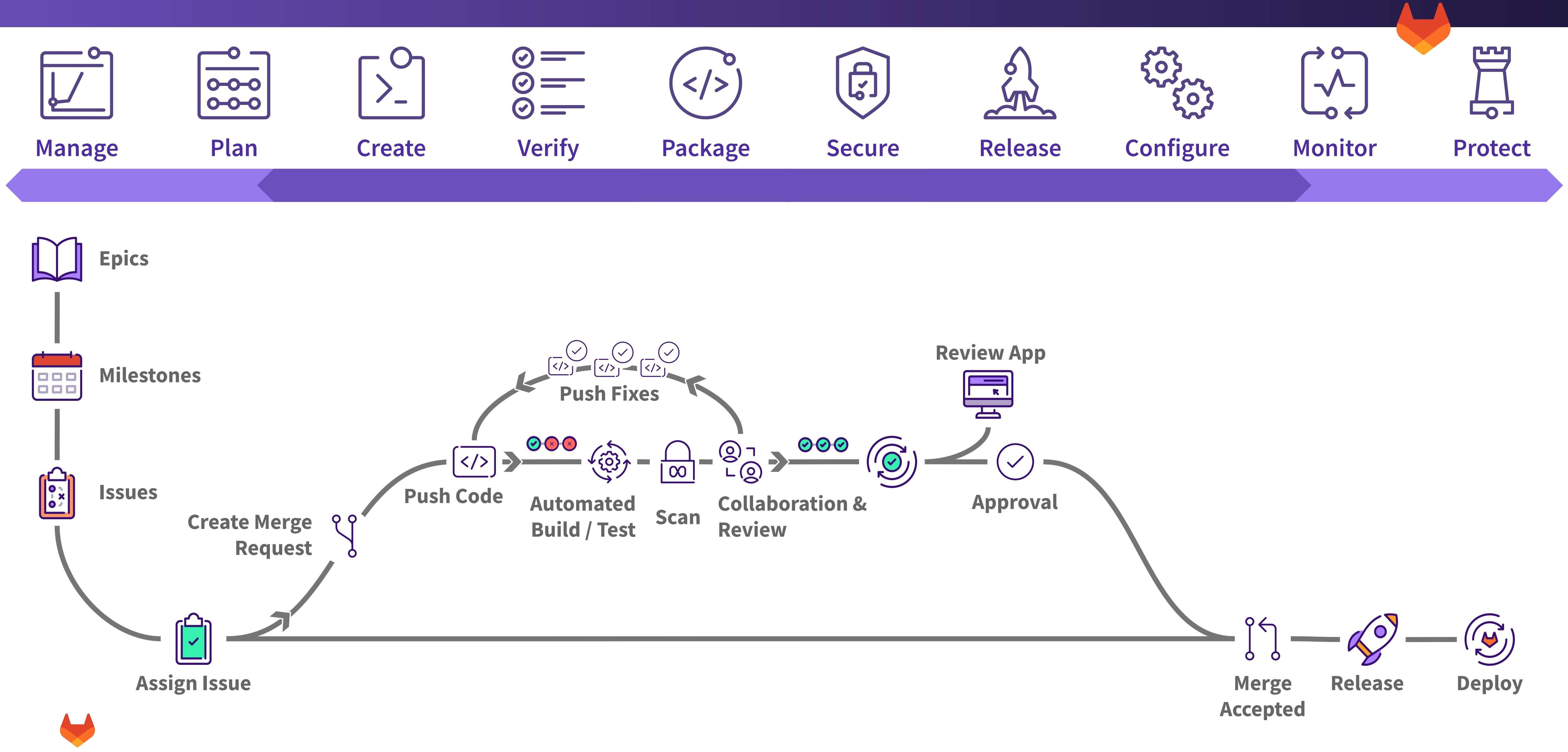
### CD Ensures Every Change is Releasable

This lowers the risk of each release- allows releases to be "boring"

### CD Delivers Value Quickly and More Often

Get fast feedback on what your end users care about

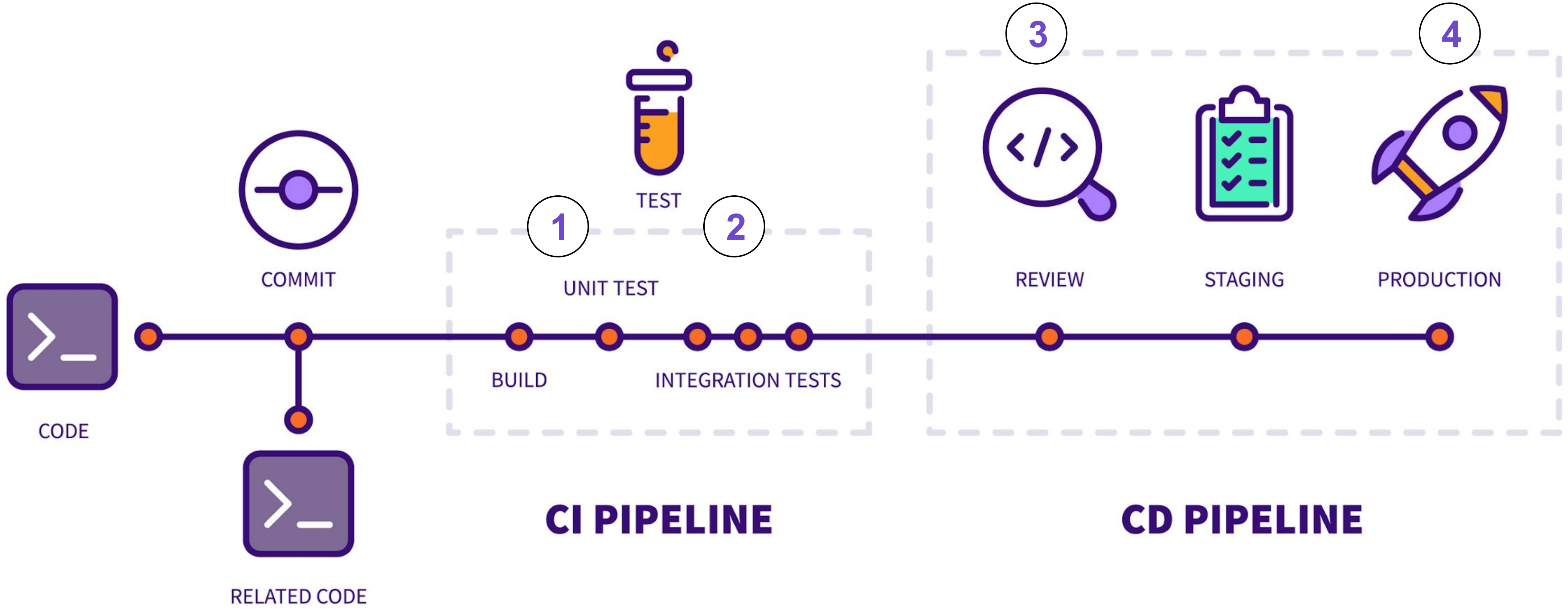
# GitLab Recommended Process



GitLab

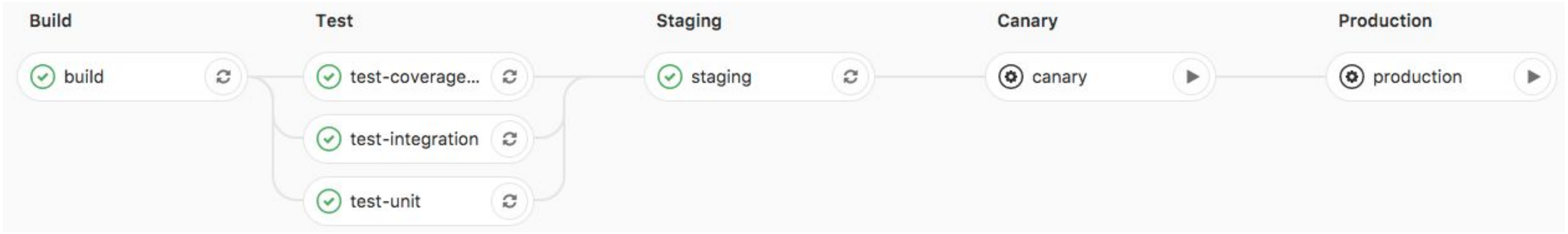
# GitLab CI/CD Overview

## Anatomy of a CI/CD Pipeline



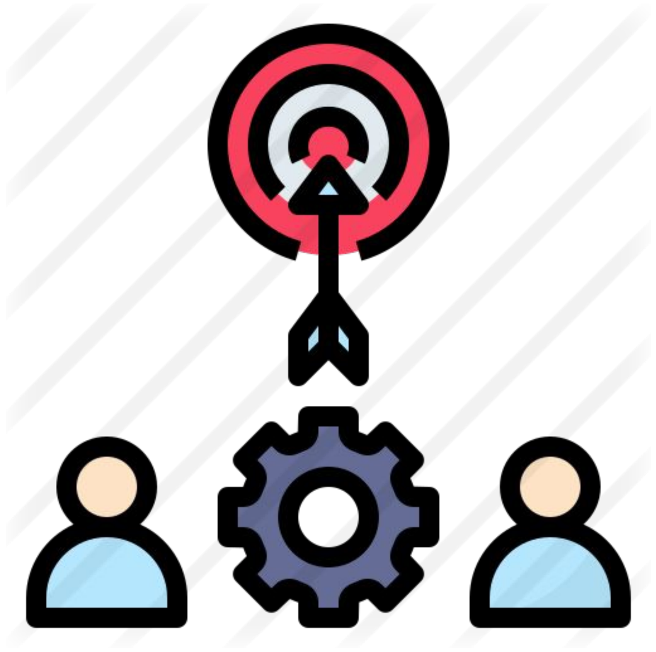
# GitLab Pipeline Graph

- Jobs define what we want to accomplish in our pipeline.
  - Executed by Runners
  - Executed in Stages
- Stages define when and how  to run jobs.
  - Stages that run tests after stages that compile the code.
- Jobs in each stage are executed in parallel
  - If *all* jobs in a stage succeed, the pipeline moves on to the next stage.
  - if one job in a stage fails, the next stage is not (usually) executed



## Ways to trigger GitLab pipeline

- Push your code to GitLab repository*
- Run it manually from the UI
- Schedule it to run at later time
- “Trigger”ed by upstream pipeline
- Use API to launch a pipeline with “trigger”



GitLab

# GitLab CI/CD Set-up

# .gitlab-ci.yml Example

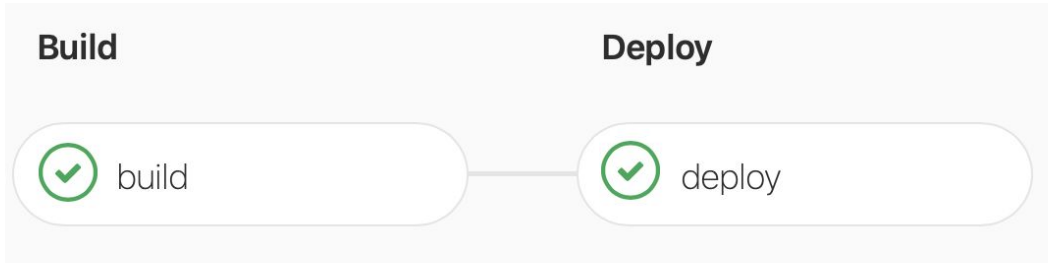
```yaml
image:
registry.gitlab.com/gitlab-examples/kubernete
s-deploy

stages:
  - build
  - deploy

variables:
  KUBE_DOMAIN: example.com

build:
  stage: build
  script:
    - command build
  only:
    - main

deploy:
  stage: deploy
  script:
    - command deploy
  environment:
    name: production
    url: http://production.example.com
  variables:
    DISABLE_POSTGRES: "yes"
  only:
    - main
```



## GitLab CI/CD pipeline configuration reference

- A job is defined as a list of keywords that define the job's behavior.
- Configuration options for your GitLab .gitlab-ci.yml file.
- The keywords available for jobs are:
    - https://docs.gitlab.com/ee/ci/yaml/

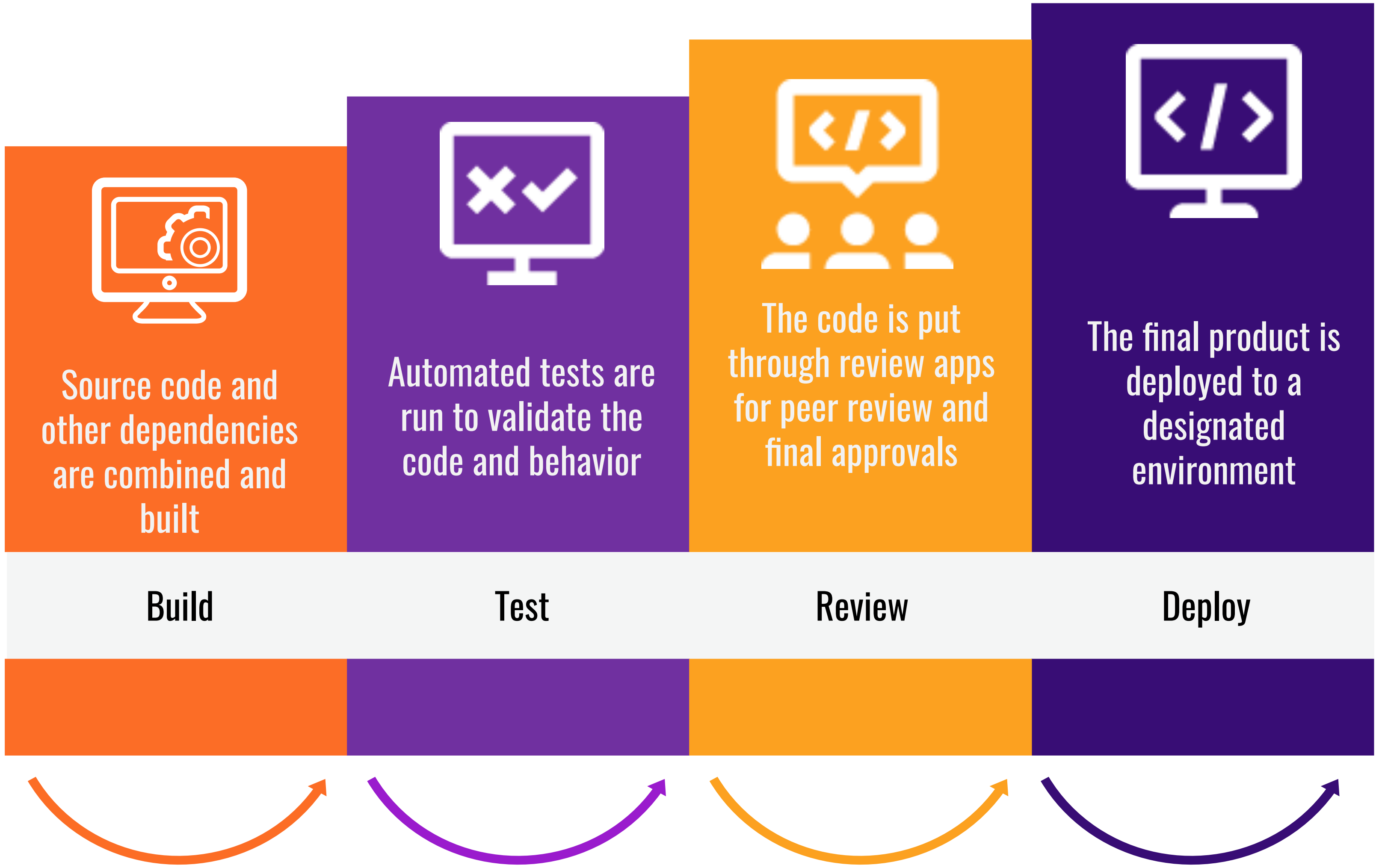
- `image`
- `services`
- `script`
- `before_script & after_script`
- `variables`
- `Environment`
- `cache`
- `artifacts`
- `rules`
- `tags`
- `when`

# Stages

**Default Stages**: Build, Test, Review, & Deploy

User can define custom stages & any number of jobs per stage

```
stages:
  - build
  - test
  - review
  - deploy
```



Source code and other dependencies are combined and built

Automated tests are run to validate the code and behavior

The code is put through review apps for peer review and final approvals

The final product is deployed to a designated environment

**Stages** seperate jobs into logical sections while **Jobs** perform the actual tasks

## Jobs and Scripts

Each Stage Can Have Multiple Jobs

Jobs Run In Parallel

```
build-code:
  stage: build
  script: build-it.sh

build-other-code:
  stage: build
  script: src/other/code/build-it.sh
```

Scripts Can Be Defined Several Different Ways

```
script: command build
```

Script Examples

```
script:
  - npm install
  - npm build
```

```
script:

 -  scripts/build_script.sh
```

## Basic Parameters

```yaml
test:
  script:
  - apt-get update -qy
  - bundle install --path /cache
  - bundle exec rake test

staging:
  stage: deploy
  script:
  - gem install dpl
  - dpl --provider=heroku --app=ruby-test-staging --api-key=$HEROKU_KEY
  only:
  - main

production:
  stage: deploy
  script:
  - gem install dpl
  - dpl --provider=heroku --app=ruby-prod --api-key=$HEROKU_PROD_KEY
  only:
  - tags
```

## Image

**Images are pulled from Docker Hub by default**

Use of a public image:

```
image: ruby:2.3
```

**Images stored in the GitLab Container Registry**

Use of a custom image:

```
image:
'registry.gitlab.com/gitl
ab-org/ci-training-sample
:latest'
```

.gitlab-ci.yml build so far

```
image: registry.example.com/k8-deploy:latest
```

## Services & Variables

Services

```
services:
  - postgre
```

Services lines tell the Runner that additional images are needed

Variables

```
variables:
  - POSTGRES_DB:
rails-sample-1_test
  - POSTGRES_USER: root
    POSTGRES_PASSWORD:
```

Variables also defined in Project > Settings > CI/CD > Variables

.gitlab-ci.yml build so far

```
image: registry.example.com/k8-deploy:latest
services:
  - postgres
variables:
  - POSTGRES_DB: rails-sample-1_test
```

## What Our **.gitlab-ci.yml** looks like so far...

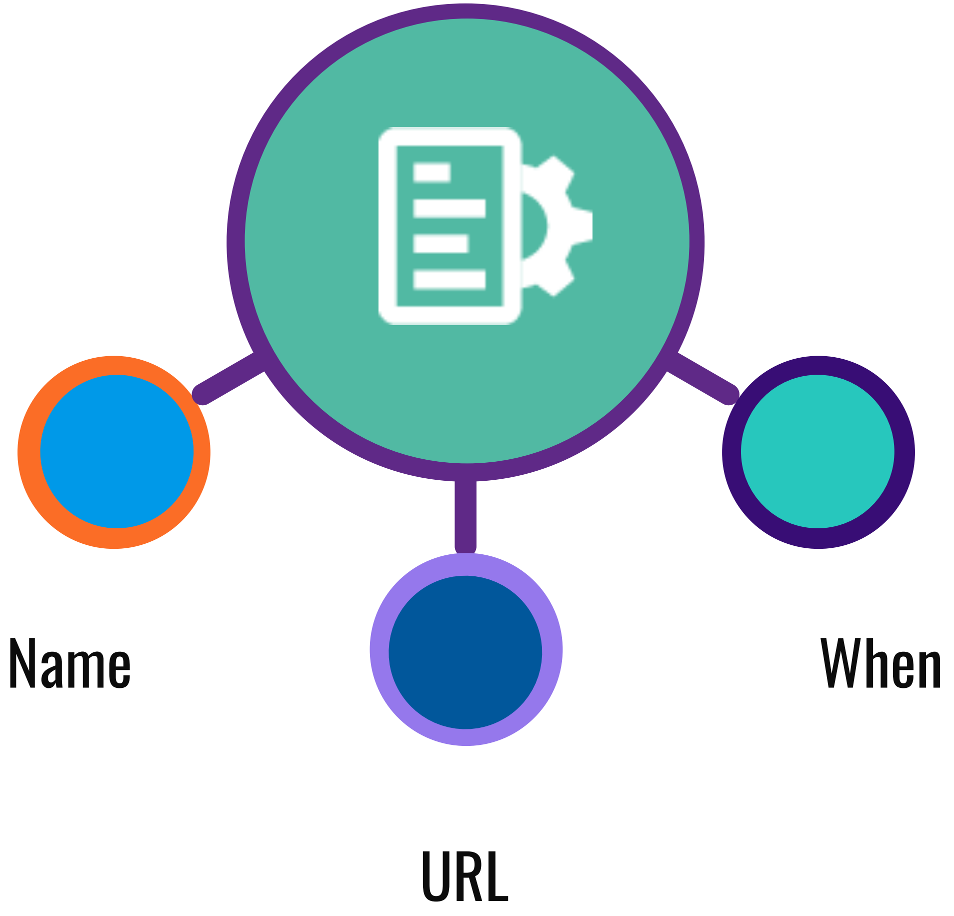
```yaml
image: registry.example.com/k8-deploy:latest
services:
  - postgres
variables:
  - POSTGRES_DB: rails-sample-1_test
stages:
  - build
  - test
  - deploy
deploy-code:
  stage: deploy
  script:
  - command deploy
```

## Environments

The environment keyword defines where the app is deployed and is defined by 3 parts.

```
environment:
  name: prod
  url: http://$CI_PROJECT_NAME.$KUBE_DOMAIN
when: manual
```

**When** triggers jobs & stages manually (e.g. deploy to production)



## Only & Except- Restricting When a Job is Executed

```
pseudo-deploy:
  stage: deploy
  script:
  - command deploy_review
  only:
  - branches
  except:
  - main
  environment:
    name: review
    url: http://$CI_PROJECT_NAME-review.$KUBE_DOMAIN
```

**Only**

The name of branch to execute on (in this case all branches)

**Except**

Branches NOT to execute on with exception to the main Branch

The rules syntax is an improved, more powerful solution for defining when jobs should run or not. *Consider using rules instead of only/except to get the most out of your pipelines.*

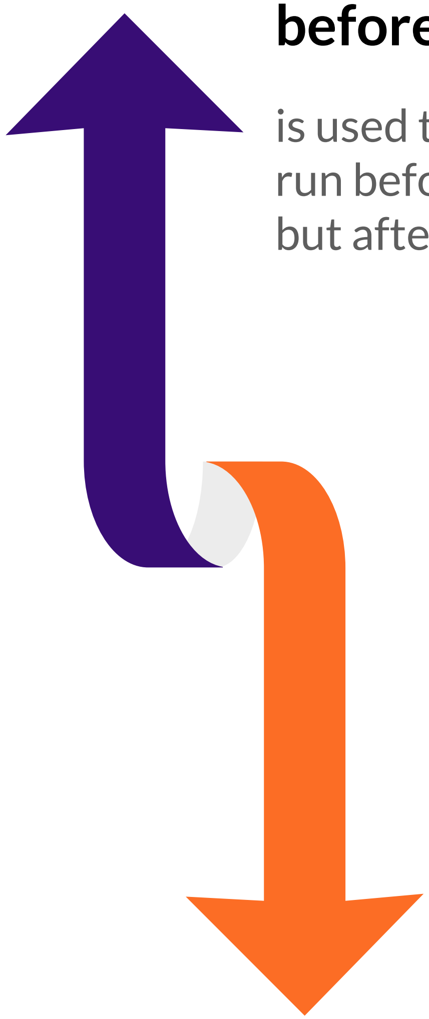
## Rules - Restricting When a Job is Executed

```yaml
pseudo-deploy:
  stage: deploy
  script:
  - command deploy_review
  rules:
    - if: '$CI_COMMIT_REF_NAME == "main"'
      when: never
    - when: always
  environment:
    name: review
    url: http://$CI_PROJECT_NAME-review.$KUBE_DOMAIN
```

## before_script & after_script

Run before and after the script defined in each job

- Can update the image with the latest version of components
- They run within the job and can interact with the job



**before_script**

is used to define a command that should be run before each job, including deploy jobs, but after the restoration of any artifacts

```
before_script:
  - echo $CI_BUILD_STAGE
  - apt-get update
  - apt-get install node-js -y
  - bundle install
  - npm install
after_script:
  - rm temp/*.tmp
```

**after_script**

is used to define the command that will be run after each job, including failed ones.

## Cache & Artifacts

Cache is used to pass information between jobs & stages by storing project dependencies

```
cache:
  paths:
     - binary/
     - .config
```

There may be build artifacts you want to save

```
artifacts:
  when: on_success
  paths:
     - bin/target
```

## What Our .gitlab-ci.yml looks like so far...

```
image: registry.example.com/k8-deploy:latest
services:
  - postgres
variables:
  - POSTGRES_DB: rails-sample-1_test
cache:
  paths:
  - binary/
stages:
  - build
  - test
  - deploy
deploy-code:
  stage: deploy
  script:
  - command deploy
  environment:
    name:  production
    url:  http://$CI_PROJECT_NAME.$KUBE_DOMAIN
  when:  manual
  only:
  - main
```

```
build-it:
  stage: build
  script:
  - command build
  only:
  - main
  artifacts:
    when:  on_success
    paths:
    - bin/target
```

## Tags

- Tags are used to select a specific runner
    - CI tags are different from Git tags
- Runners with the required tags can pick-up the job
    - If a Runner has more tags than required, it can still run that particular job; including if the job requires no tags at all

```yaml
job-name:
  tags:
    - ruby
    - test
```

## What Our **.gitlab-ci.yml** looks like so far...

```
image: registry.example.com/k8-deploy:latest
services:
  - postgres
variables:
  - POSTGRES_DB: rails-sample-1_test
cache:
  paths:
  - binary/
stages:
  - build
  - test
  - deploy
deploy-code:
  stage: deploy
  script:
  - command deploy
  environment:
    name:  production
    url:  http://$CI_PROJECT_NAME.$KUBE_DOMAIN
  when:  manual
  only:
  - main
```

```
build-it:
  stage: build
  script:
  - command build
  only:
  - main
  tags:
    - osx
    - ios
  artifacts:
    when:  on_success
    paths:
    - bin/target
```

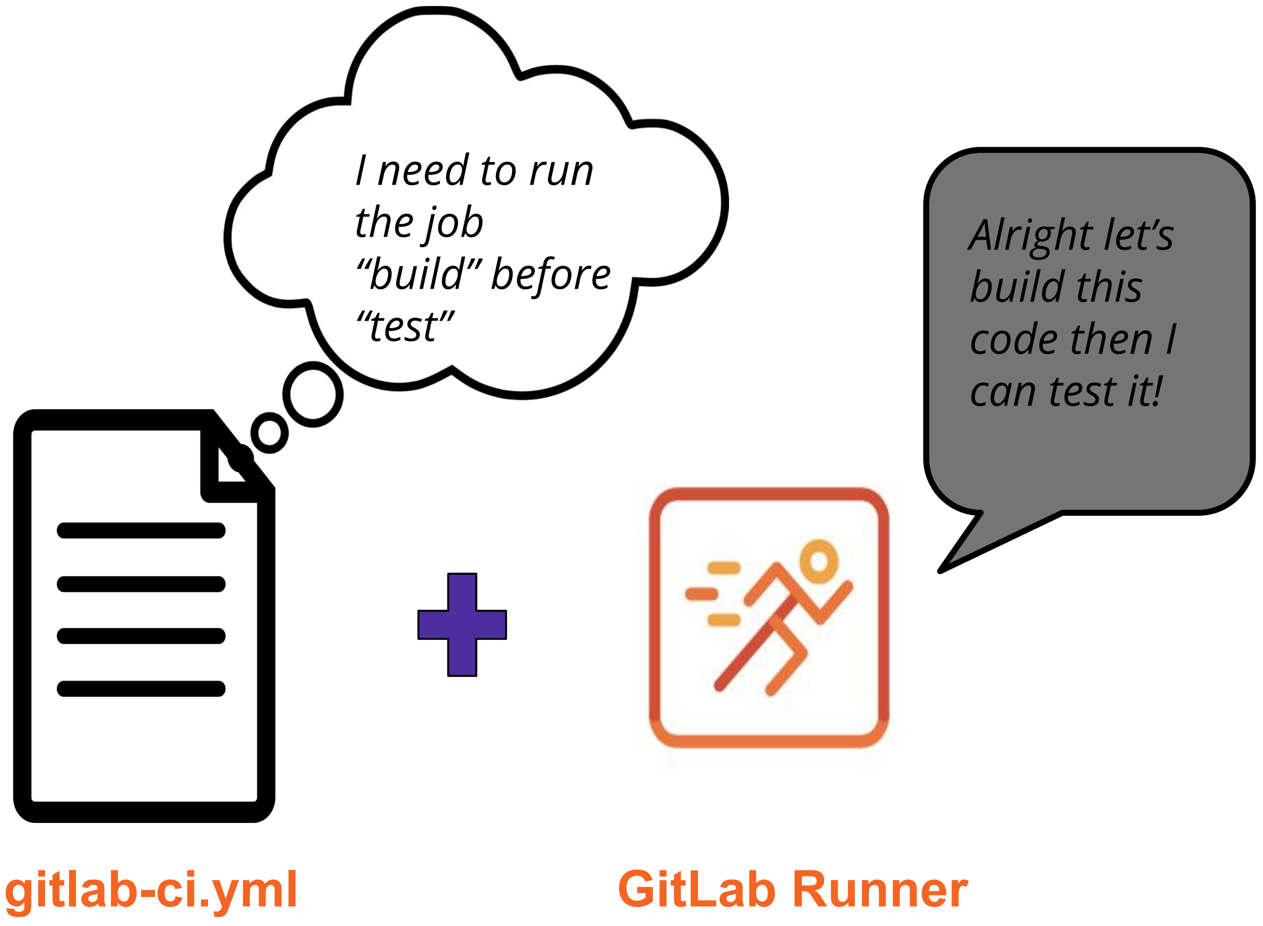
GitLab

# GitLab CI/CD Runners

## Configuration File + Runner

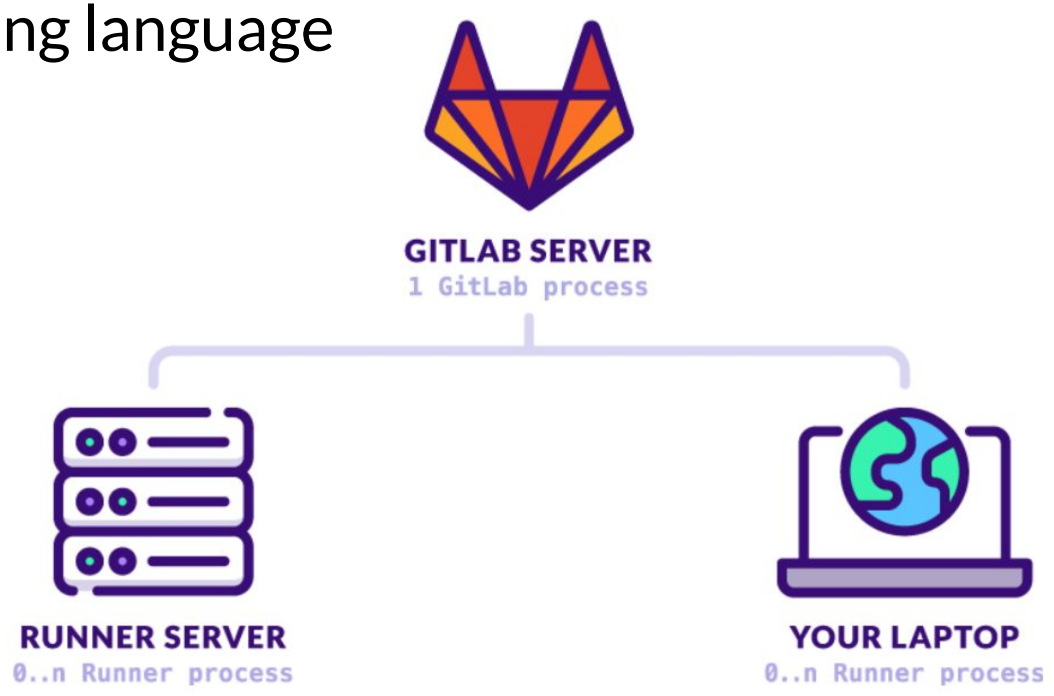
- .gitlab-ci.yml file
  - Instructions for GitLab CI/CD jobs.
  - Lives in the root of the repository

- GitLab Runner
  - Lightweight agent that runs CI/CD jobs.


I need to run the job "build" before "test"
Alright let's build this code then I can test it!
.gitlab-ci.yml
GitLab Runner

## Runner Architecture

- The GitLab runner can be installed on any platform where you build Go binaries.
  - Linux, macOS, Windows, FreeBSD, Cloud Provider, Bare Metal, Your work station and Docker
- The GitLab runner can test any programming language
  - .Net, Java, Python, C, PHP and others.
- Created by an Administrator



## A Runner Can Be....

**Shared or Specific**

**Tagged or Untagged**

**Protected or Not Protected**

## Shared vs. Specific Runners

**Shared Runners**

Available to every project with similar requirements

**Description**

Included in the pool for all projects

Managed by GitLab Admin

Typically auto-scaling or otherwise scaled

VS

**Specific Runners**

Tied to one or more specific projects

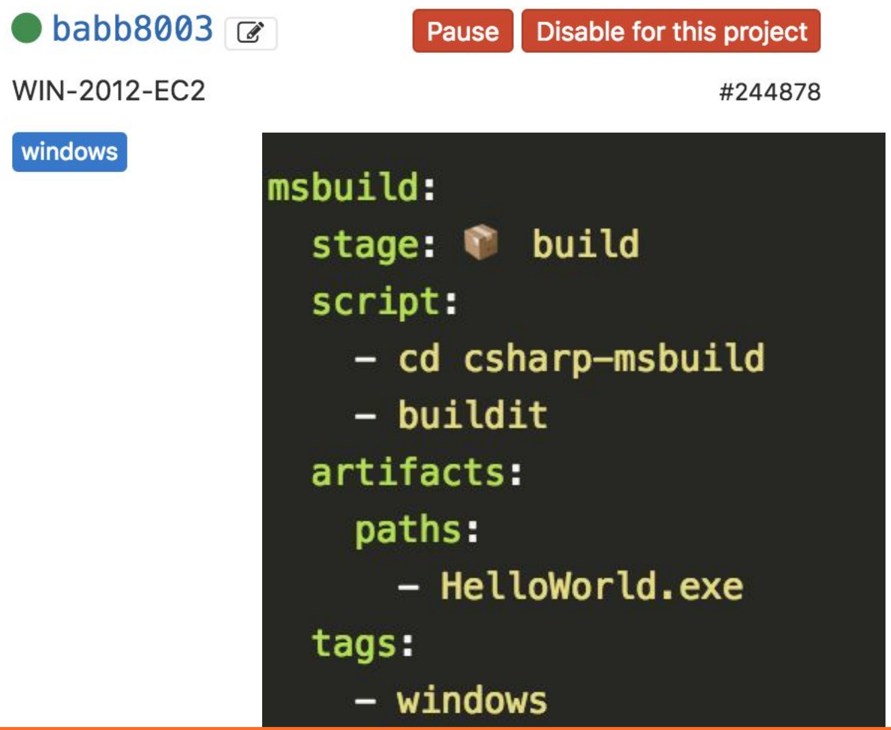
**Description**

In the pool for ONLY specific projects

Managed by Runner Owner(s)

Typically for specialized builds, or if an org needs to do so for billing

## Tagged vs. Untagged

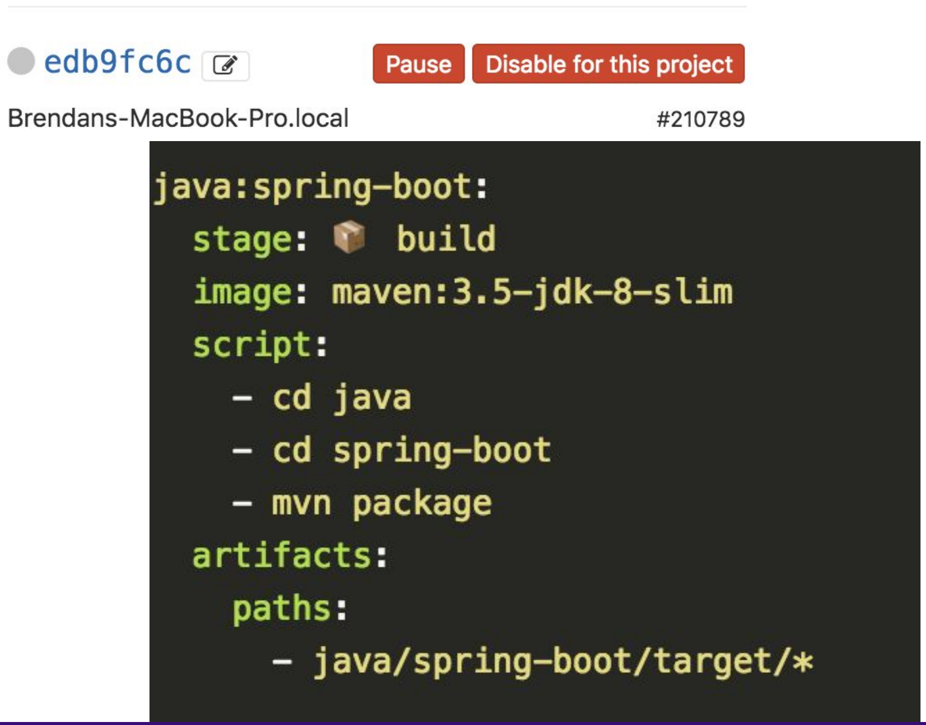


● babb8003 Pause Disable for this project

WIN-2012-EC2 #244878

windows

```
msbuild:
  stage: 📦 build
  script:
    - cd csharp-msbuild
    - buildit
  artifacts:
    paths:
      - HelloWorld.exe
  tags:
    - windows
```



● edb9fc6c Pause Disable for this project

Brendans-MacBook-Pro.local #210789

```
java:spring-boot:
  stage: 📦 build
  image: maven:3.5-jdk-8-slim
  script:
    - cd java
    - cd spring-boot
    - mvn package
  artifacts:
    paths:
      - java/spring-boot/target/*
```

# Tagged

Only used to run jobs tagged with same tag

# Untagged

Used to run jobs with no tags

## Protected vs. Non-Protected

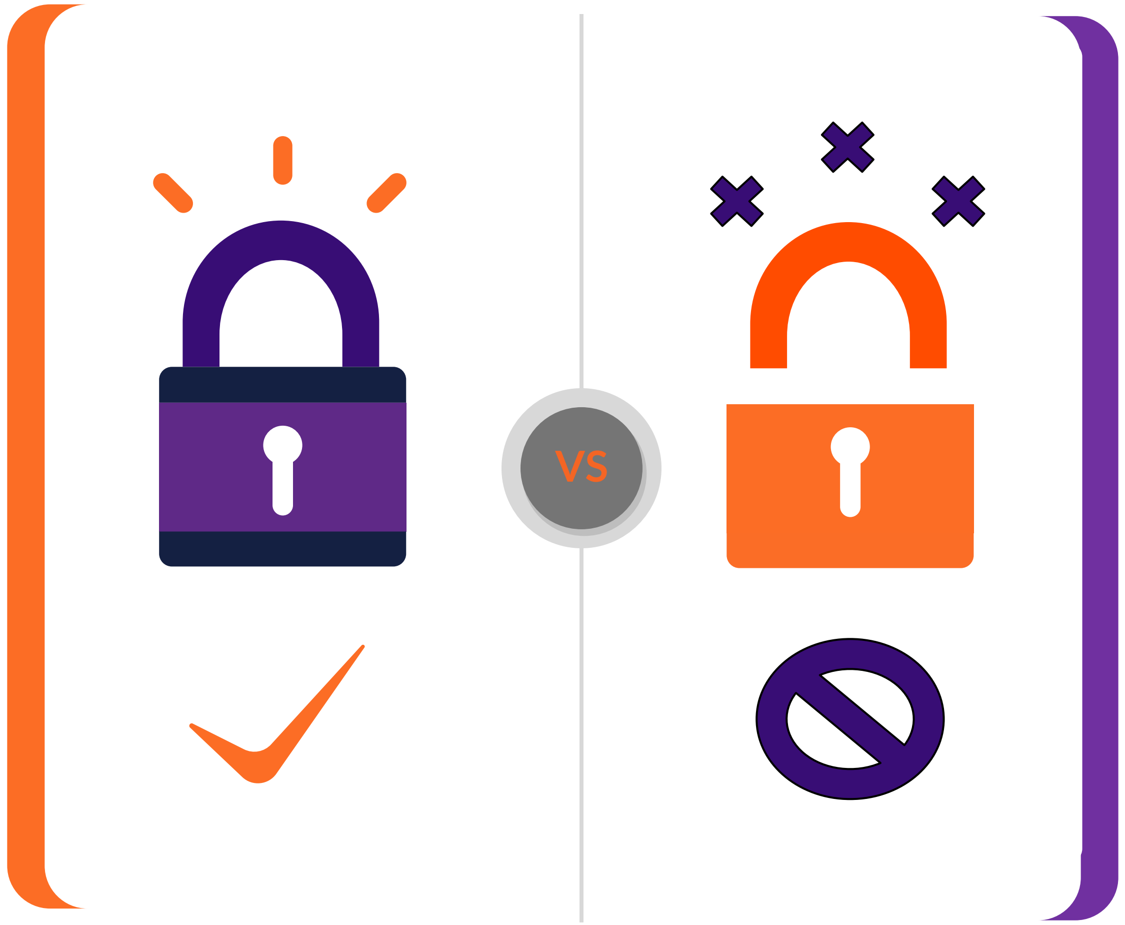
### Protected

**Characteristics**

ONLY runs jobs from

- Protected Branches
- Protected Tags

Typically used for runners containing deploy keys or other sensitive capabilities

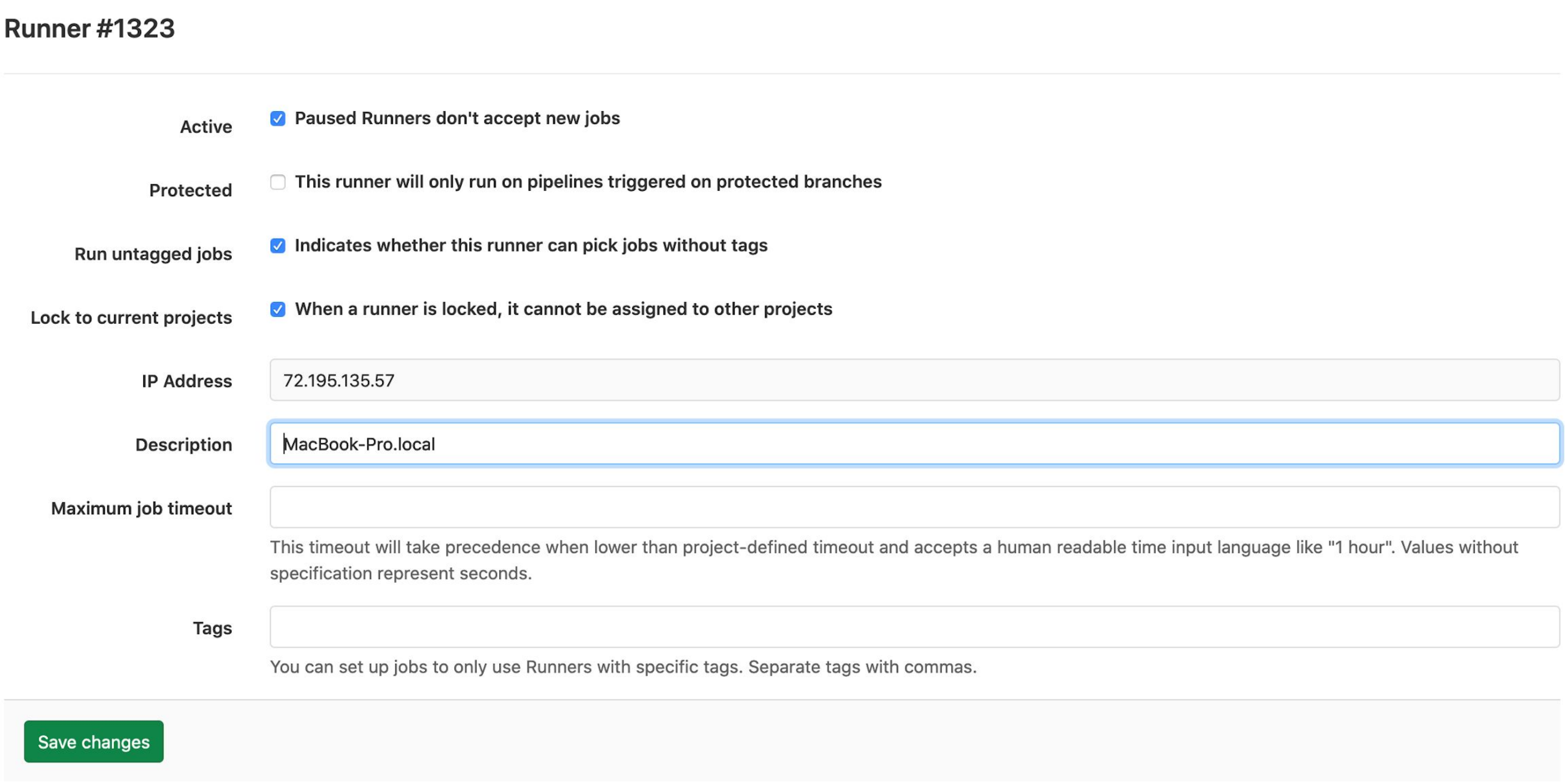


### Non-Protected

**Characteristics**

- Runs jobs from ANY branch
- Used for ANY build

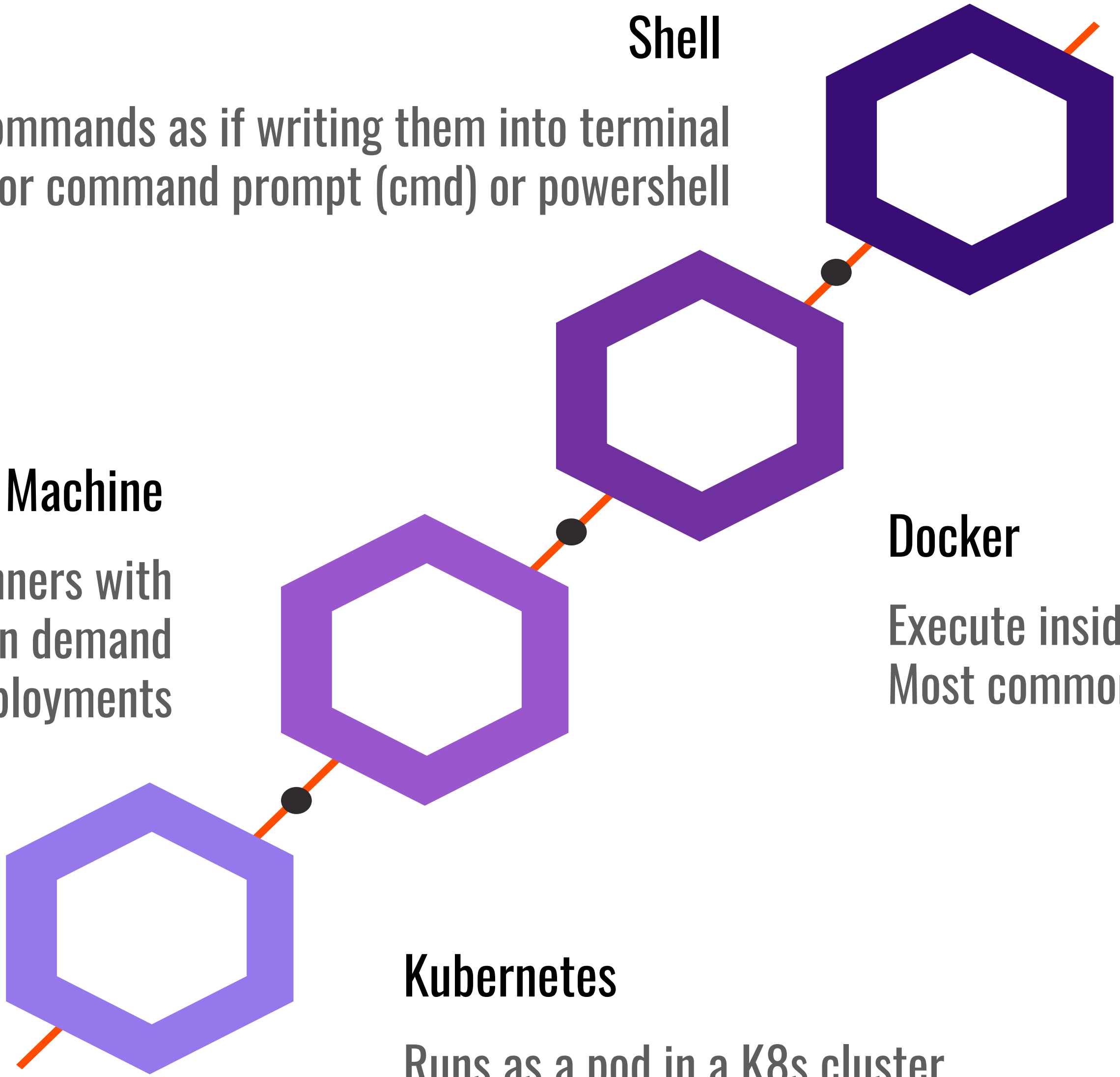
# Additional Runner Options



## Runner #1323

| | |
|---|---|
| Active | [✓] Paused Runners don't accept new jobs |
| Protected | [ ] This runner will only run on pipelines triggered on protected branches |
| Run untagged jobs | [✓] Indicates whether this runner can pick jobs without tags |
| Lock to current projects | [✓] When a runner is locked, it cannot be assigned to other projects |
| IP Address | 72.195.135.57 |
| Description | MacBook-Pro.local |
| Maximum job timeout | This timeout will take precedence when lower than project-defined timeout and accepts a human readable time input language like "1 hour". Values without specification represent seconds. |
| Tags | You can set up jobs to only use Runners with specific tags. Separate tags with commas. |

## Executors: Common



**Shell**

Directly run commands as if writing them into terminal (bash or sh) or command prompt (cmd) or powershell

**Docker**

Execute inside of a docker image
Most common!

**Docker Machine**

“Main” machine scales up runners with *any* executor on demand
Typical in cloud deployments

**Kubernetes**

Runs as a pod in a K8s cluster
Can also feature auto-scaling

## Executors: Less Common

### 1. VirtualBox

Base VM for runner
"Main" creates a new VM for each needed runner

### 2. Parallels

Hint: Parallels is a nice platform on top of VirtualBox

### 3. SSH

Similar to shell, but not as many features (bash only, no caching)
Does allow you to SSH and execute commands on a machine you might not want to install runner on

GitLab

## Q&A