# **Tutorial: Create and run your first GitLab CI/CD pipeline**

Before you start, make sure you have:

* A project in GitLab that you would like to use CI/CD for.
* The Maintainer or Owner role for the project.

If you don’t have a project, you can create a public project for free on [https://gitlab.com](https://gitlab.com/).

## **Steps**

To create and run your first pipeline:

[Ensure you have runners available](https://docs.gitlab.com/ee/ci/quick_start/" \l "ensure-you-have-runners-available) to run your jobs.

If you’re using GitLab.com, you can skip this step. GitLab.com provides shared runners for you.

[Create a .gitlab-ci.yml file](https://docs.gitlab.com/ee/ci/quick_start/" \l "create-a-gitlab-ciyml-file) at the root of your repository. This file is where you define the CI/CD jobs.

When you commit the file to your repository, the runner runs your jobs. The job results [are displayed in a pipeline](https://docs.gitlab.com/ee/ci/quick_start/" \l "view-the-status-of-your-pipeline-and-jobs).

## **Ensure you have runners available**

In GitLab, runners are agents that run your CI/CD jobs.

To view available runners:

* Go to ****Settings > CI/CD**** and expand ****Runners****.

As long as you have at least one runner that’s active, with a green circle next to it, you have a runner available to process your jobs.

### **If you don’t have a runner**

If you don’t have a runner:

1. [Install GitLab Runner](https://docs.gitlab.com/runner/install/) on your local machine.
2. [Register the runner](https://docs.gitlab.com/runner/register/) for your project. Choose the shell executor.

When your CI/CD jobs run, in a later step, they will run on your local machine.

## **Create a .gitlab-ci.yml file**

Now create a .gitlab-ci.yml file. It is a [YAML](https://en.wikipedia.org/wiki/YAML) file where you specify instructions for GitLab CI/CD.

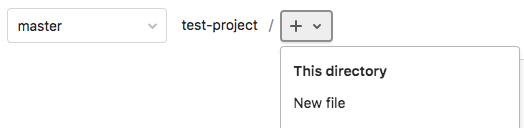
In this file, you define:

* The structure and order of jobs that the runner should execute.
* The decisions the runner should make when specific conditions are encountered.

To create a .gitlab-ci.yml file:

1. On the left sidebar, select ****Code > Repository****.

Above the file list, select the branch you want to commit to. If you’re not sure, leave master or main. Then select the plus icon () and ****New file****:

[](https://docs.gitlab.com/ee/ci/quick_start/img/new_file_v13_6.png)

For the ****Filename****, type .gitlab-ci.yml and in the larger window, paste this sample code:

build-job:

stage: build

script:

- echo "Hello, $GITLAB\_USER\_LOGIN!"

test-job1:

stage: test

script:

- echo "This job tests something"

test-job2:

stage: test

script:

- echo "This job tests something, but takes more time than test-job1."

- echo "After the echo commands complete, it runs the sleep command for 20 seconds"

- echo "which simulates a test that runs 20 seconds longer than test-job1"

- sleep **20**

deploy-prod:

stage: deploy

script:

- echo "This job deploys something from the $CI\_COMMIT\_BRANCH branch."

environment: production

This example shows four jobs: build-job, test-job1, test-job2, and deploy-prod. The comments listed in the echo commands are displayed in the UI when you view the jobs. The values for the [predefined variables](https://docs.gitlab.com/ee/ci/variables/predefined_variables.html) $GITLAB\_USER\_LOGIN and $CI\_COMMIT\_BRANCH are populated when the jobs run.

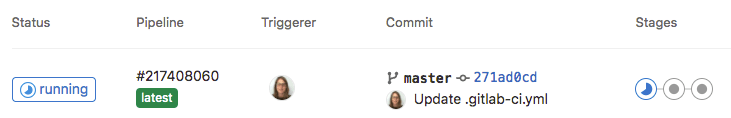
1. Select ****Commit changes****.

The pipeline starts and runs the jobs you defined in the .gitlab-ci.yml file.

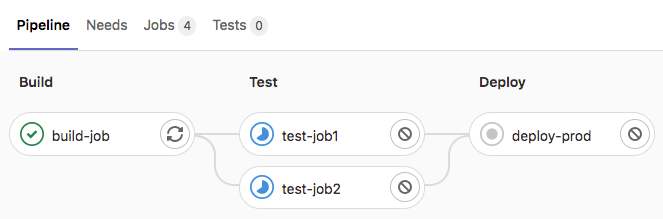
## **View the status of your pipeline and jobs**

Now take a look at your pipeline and the jobs within.

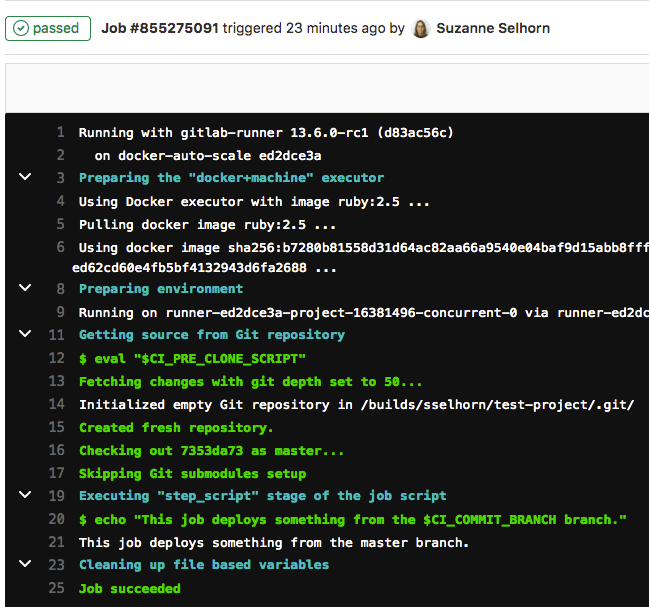
Go to ****Build > Pipelines****. A pipeline with three stages should be displayed:

[](https://docs.gitlab.com/ee/ci/quick_start/img/three_stages_v13_6.png)

View a visual representation of your pipeline by selecting the pipeline ID:

[](https://docs.gitlab.com/ee/ci/quick_start/img/pipeline_graph_v13_6.png)

View details of a job by selecting the job name. For example, deploy-prod:

[](https://docs.gitlab.com/ee/ci/quick_start/img/job_details_v13_6.png)

You have successfully created your first CI/CD pipeline in GitLab. Congratulations!

Now you can get started customizing your .gitlab-ci.yml and defining more advanced jobs.

## **.gitlab-ci.yml tips**

Here are some tips to get started working with the .gitlab-ci.yml file.

For the complete .gitlab-ci.yml syntax, see [the full .gitlab-ci.yml keyword reference](https://docs.gitlab.com/ee/ci/yaml/index.html).

* Use the [pipeline editor](https://docs.gitlab.com/ee/ci/pipeline_editor/index.html) to edit your .gitlab-ci.yml file.
* Each job contains a script section and belongs to a stage:
  + [stage](https://docs.gitlab.com/ee/ci/yaml/index.html" \l "stage) describes the sequential execution of jobs. If there are runners available, jobs in a single stage run in parallel.
  + Use the [needs keyword](https://docs.gitlab.com/ee/ci/yaml/index.html" \l "needs) to run jobs out of stage order. This creates a [Directed Acyclic Graph (DAG)](https://docs.gitlab.com/ee/ci/directed_acyclic_graph/index.html).
* You can set additional configuration to customize how your jobs and stages perform:
  + Use the [rules](https://docs.gitlab.com/ee/ci/yaml/index.html" \l "rules) keyword to specify when to run or skip jobs. The only and except legacy keywords are still supported, but can’t be used with rules in the same job.
  + Keep information across jobs and stages persistent in a pipeline with [cache](https://docs.gitlab.com/ee/ci/yaml/index.html" \l "cache) and [artifacts](https://docs.gitlab.com/ee/ci/yaml/index.html" \l "artifacts). These keywords are ways to store dependencies and job output, even when using ephemeral runners for each job.
  + Use the [default](https://docs.gitlab.com/ee/ci/yaml/index.html" \l "default) keyword to specify additional configurations that are applied to all jobs. This keyword is often used to define [before\_script](https://docs.gitlab.com/ee/ci/yaml/index.html" \l "before_script) and [after\_script](https://docs.gitlab.com/ee/ci/yaml/index.html" \l "after_script) sections that should run on every job.

# **Create a complex pipeline**

his tutorial walks you through configuring a progressively more complex CI/CD pipeline through small, iterative steps. The pipeline is always fully functional, but it gains more functionality with each step. The goal is to build, test, and deploy a documentation site.

When you finish this tutorial, you will have a new project on GitLab.com and a working documentation site using [Docusaurus](https://docusaurus.io/).

To complete this tutorial, you will:

1. Create a project to hold the Docusaurus files
2. Create the initial pipeline configuration file
3. Add a job to build the site
4. Add a job to deploy the site
5. Add test jobs
6. Start using merge request pipelines
7. Reduce duplicated configuration

## **Prerequisites**

* You need an account on GitLab.com.
* You should be familiar with Git.
* Node.js must be installed on your local machine. For example, on macOS you can [install node](https://formulae.brew.sh/formula/node) with brew install node.

## **Create a project to hold the Docusaurus files**

Before adding the pipeline configuration, you must first set up a Docusaurus project on GitLab.com:

1. Create a new project under your username (not a group):
   1. On the left sidebar, select ****Search or go to****.
   2. Select ****View all my projects****.
   3. On the right of the page, select ****New project****.
   4. Select ****Create blank project****.
   5. Enter the project details:
      * In the ****Project name**** field, enter the name of your project, for example My Pipeline Tutorial Project.
      * Select ****Initialize repository with a README****.
   6. Select ****Create project****.

On the right of the ****Project Overview**** page for your project, select ****Clone**** to find the clone paths for your project. Copy the SSH or HTTP path and use the path to clone the project locally.

For example, to clone with SSH into a pipeline-tutorial directory on your computer:

git clone git@gitlab.com:my-username/my-pipeline-tutorial-project.git pipeline-tutorial

Change to the project’s directory, then generate a new Docusaurus site:

cd pipeline-tutorial

npm init docusaurus

The Docusaurus initialization wizard prompts you with questions about the site. Use all the default options.

The initialization wizard sets up the site in website/, but the site should be in the root of the project. Move the files up to the root and delete the old directory:

mv website/**\*** .rm **-r** website

Update the Docusaurus configuration file with the details of your GitLab project. In docusaurus.config.js:

* + Set url: to a path with this format: https://<my-username>.gitlab.io/.
  + Set baseUrl: to your project name, like /my-pipeline-tutorial-project/.

Commit the changes, and push them to GitLab:

git add .

git commit **-m** "Add simple generated Docusaurus site"

git push origin

## **Create the initial CI/CD configuration file**

Start with the simplest possible pipeline configuration file to ensure CI/CD is enabled in the project and runners are available to run jobs.

This step introduces:

* [Jobs](https://docs.gitlab.com/ee/ci/jobs/index.html): These are self-contained parts of a pipeline that run your commands. Jobs run on [runners](https://docs.gitlab.com/ee/ci/runners/index.html), separate from the GitLab instance.
* [script](https://docs.gitlab.com/ee/ci/yaml/index.html" \l "script): This section of a job’s configuration is where you define the commands for jobs. If there are multiple commands (in an array), they run in order. Each command executes as if it was run as a CLI command. By default, if a command fails or returns an error, the job is flagged as failed and no more commands run.

In this step, create a .gitlab-ci.yml file in the root of the project with this configuration:

test-job:

script:

- echo "This is my first job!"

- date

Commit and push this change to GitLab, then:

1. Go to ****Build > Pipelines**** and make sure a pipeline runs in GitLab with this single job.
2. Select the pipeline, then select the job to view the job’s log and see the This is my first job! message followed by the date.

Now that you have a .gitlab-ci.yml file in your project, you can make all future changes to pipeline configuration with the [pipeline editor](https://docs.gitlab.com/ee/ci/pipeline_editor/index.html).

## **Add a job to build the site**

A common task for a CI/CD pipeline is to build the code in the project then deploy it. Start by adding a job that builds the site.

This step introduces:

* [image](https://docs.gitlab.com/ee/ci/yaml/index.html" \l "image): Tell the runner which Docker container to use to run the job in. The runner:
  1. Downloads the container image and starts it.
  2. Clones your GitLab project into the running container.
  3. Runs the script commands, one at a time.
* [artifacts](https://docs.gitlab.com/ee/ci/yaml/index.html" \l "artifacts): Jobs are self-contained and do not share resources with each other. If you want files generated in one job to be used in another job, you must save them as artifacts first. Then later jobs can retrieve the artifacts and use the generated files.

In this step, replace test-job with build-job:

* Use image to configure the job to run with the latest node image. Docusaurus is a Node.js project and the node image has the needed npm commands built in.
* Run npm install to install Docusaurus into the running node container, then run npm run build to build the site.
* Docusaurus saves the built site in build/, so save these files with artifacts.

build-job:

image: node

script:

- npm install

- npm run build

artifacts:

paths:

- "build/"

Use the pipeline editor to commit this pipeline configuration to the default branch, and check the job log. You can:

* See the npm commands run and build the site.
* Verify that the artifacts are saved at the end.
* Browse the contents of the artifacts file by selecting ****Browse**** to the right of the job log after the job completes.

## **Add a job to deploy the site**

After verifying the Docusaurus site builds in build-job, you can add a job that deploys it.

This step introduces:

* [stage](https://docs.gitlab.com/ee/ci/yaml/index.html" \l "stage) and [stages](https://docs.gitlab.com/ee/ci/yaml/index.html" \l "stage): The most common pipeline configurations group jobs into stages. Jobs in the same stage can run in parallel, while jobs in later stages wait for jobs in earlier stages to complete. If a job fails, the whole stage is considered failed and jobs in later stages do not start running.
* [GitLab Pages](https://docs.gitlab.com/ee/user/project/pages/index.html): To host your static site, you will use GitLab Pages.

In this step:

* Add a job that fetches the built site and deploys it. When using GitLab Pages, the job is always named pages. The artifacts from the build-job are fetched automatically and extracted into the job. Pages looks for the site in the public/ directory though, so add a script command to move the site to that directory.
* Add a stages section, and define the stages for each job. build-job runs first in the build stage, and pages runs after in the deploy stage.

stages: *# List of stages for jobs and their order of execution*

- build

- deploy

build-job:

stage: build *# Set this job to run in the `build` stage*

image: node

script:

- npm install

- npm run build

artifacts:

paths:

- "build/"

pages:

stage: deploy *# Set this new job to run in the `deploy` stage*

script:

- mv build/ public/

artifacts:

paths:

- "public/"

Use the pipeline editor to commit this pipeline configuration to the default branch, and view the pipeline details from the ****Pipelines**** list. Verify that:

* The two jobs run in different stages, build and deploy.
* After the pages job completes a pages-deploy job appears, which is the GitLab process that deploys the Pages site. When that job completes, you can visit your new Docusaurus site. The Pages documentation explains [the URL formatting](https://docs.gitlab.com/ee/user/project/pages/getting_started_part_one.html" \l "gitlab-pages-default-domain-names), which should be similar to https://<my-username>.gitlab.io/<my-pipeline-tutorial-project>/.

## **Add test jobs**

Now that the site builds and deploys as expected, you can add tests and linting. For example, a Ruby project might run RSpec test jobs. Docusaurus is a static site that uses Markdown and generated HTML, so this tutorial adds jobs to test the Markdown and HTML.

This step introduces:

* [allow\_failure](https://docs.gitlab.com/ee/ci/yaml/index.html" \l "allow_failure): Jobs that fail intermittently, or are expected to fail, can slow down productivity or be difficult to troubleshoot. Use allow\_failure to let jobs fail without halting pipeline execution.
* [dependencies](https://docs.gitlab.com/ee/ci/yaml/index.html" \l "dependencies): Use dependencies to control artifact downloads in individual jobs by listing which jobs to fetch artifacts from.

In this step:

* Add a new test stage that runs between build and deploy. These three stages are the default stages when stages is undefined in the configuration.
* Add a lint-markdown job to run [markdownlint](https://github.com/DavidAnson/markdownlint) and check the Markdown in your project. markdownlint is a static analysis tool that checks that your Markdown files follow formatting standards.
  + The sample Markdown files Docusaurus generates are in blog/ and docs/.
  + This tool scans the original Markdown files only, and does not need the generated HTML saved in the build-job artifacts. Speed up the job with dependencies: [] so that it fetches no artifacts.
  + A few of the sample Markdown files violate default markdownlint rules, so add allow\_failure: true to let the pipeline continue despite the rule violations.
* Add a test-html job to run [HTMLHint](https://htmlhint.com/) and check the generated HTML. HTMLHint is a static analysis tool that scans generated HTML for known issues.
* Both test-html and pages need the generated HTML found in the build-job artifacts. Jobs fetch artifacts from all jobs in earlier stages by default, but add dependencies: to make sure the jobs don’t accidentally download other artifacts after future pipeline changes.

stages:

- build

- test *# Add a `test` stage for the test jobs*

- deploy

build-job:

stage: build

image: node

script:

- npm install

- npm run build

artifacts:

paths:

- "build/"

lint-markdown:

stage: test

image: node

dependencies: [] *# Don't fetch any artifacts*

script:

- npm install markdownlint-cli2 --global *# Install markdownlint into the container*

- markdownlint-cli2 -v *# Verify the version, useful for troubleshooting*

- markdownlint-cli2 "blog/\*\*/\*.md" "docs/\*\*/\*.md" *# Lint all markdown files in blog/ and docs/*

allow\_failure: **true** *# This job fails right now, but don't let it stop the pipeline.*

test-html:

stage: test

image: node

dependencies:

- build-job *# Only fetch artifacts from `build-job`*

script:

- npm install --save-dev htmlhint *# Install HTMLHint into the container*

- npx htmlhint --version *# Verify the version, useful for troubleshooting*

- npx htmlhint build/ *# Lint all markdown files in blog/ and docs/*

pages:

stage: deploy

dependencies:

- build-job *# Only fetch artifacts from `build-job`*

script:

- mv build/ public/

artifacts:

paths:

- "public/"

Commit this pipeline configuration to the default branch, and view the pipeline details.

* The test-markdown job fails because the sample Markdown violates the default markdownlint rules, but is allowed to fail. You can:
  + Ignore the violations for now. They do not need to be fixed as part of the tutorial.
  + Fix the Markdown file violations. Then you can change allow\_failure to false, or remove allow\_failure completely because allow\_failure: false is the default behavior when not defined.
  + Add a markdownlint configuration file to limit which rule violations to alert on.
* You can also make changes to the Markdown file content and see the changes on the site after the next deployment.

## **Start using merge request pipelines**

With the pipeline configurations above, the site deploys every time a pipeline completes successfully, but this is not an ideal development workflow. It’s better to work from feature branches and merge requests, and only deploy the site when changes merge to the default branch.

This step introduces:

* [rules](https://docs.gitlab.com/ee/ci/yaml/index.html" \l "rules): Add rules to each job to configure in which pipelines they run. You can configure jobs to run in [merge request pipelines](https://docs.gitlab.com/ee/ci/pipelines/merge_request_pipelines.html), [scheduled pipelines](https://docs.gitlab.com/ee/ci/pipelines/schedules.html), or other specific situations. Rules are evaluated from top to bottom, and if a rule matches, the job is added to the pipeline.
* [CI/CD variables](https://docs.gitlab.com/ee/ci/variables/index.html): use these environment variables to configure job behavior in the configuration file and in script commands. [Predefined CI/CD variables](https://docs.gitlab.com/ee/ci/variables/predefined_variables.html) are variables that you do not need to manually define. They are automatically injected into pipelines so you can use them to configure your pipeline. Variables are usually formatted as $VARIABLE\_NAME. and predefined variables are usually prefixed with $CI\_.

In this step:

* Create a new feature branch and make the changes in the branch instead of the default branch.
* Add rules to each job:
  + The site should only deploy for changes to the default branch.
  + The other jobs should run for all changes in merge requests or the default branch.
* With this pipeline configuration, you can work from a feature branch without running any jobs, which saves resources. When you are ready to validate your changes, create a merge request and a pipeline runs with the jobs configured to run in merge requests.
* When your merge request is accepted and the changes merge to the default branch, a new pipeline runs which also contains the pages deployment job. The site deploys if no jobs fail.

stages:

- build

- test

- deploy

build-job:

stage: build

image: node

script:

- npm install

- npm run build

artifacts:

paths:

- "build/"

rules:

- if: $CI\_PIPELINE\_SOURCE == 'merge\_request\_event' *# Run for all changes to a merge request's source branch*

- if: $CI\_COMMIT\_BRANCH == $CI\_DEFAULT\_BRANCH *# Run for all changes to the default branch*

lint-markdown:

stage: test

image: node

dependencies: []

script:

- npm install markdownlint-cli2 --global

- markdownlint-cli2 -v

- markdownlint-cli2 "blog/\*\*/\*.md" "docs/\*\*/\*.md"

allow\_failure: **true**

rules:

- if: $CI\_PIPELINE\_SOURCE == 'merge\_request\_event' *# Run for all changes to a merge request's source branch*

- if: $CI\_COMMIT\_BRANCH == $CI\_DEFAULT\_BRANCH *# Run for all changes to the default branch*

test-html:

stage: test

image: node

dependencies:

- build-job

script:

- npm install --save-dev htmlhint

- npx htmlhint --version

- npx htmlhint build/

rules:

- if: $CI\_PIPELINE\_SOURCE == 'merge\_request\_event' *# Run for all changes to a merge request's source branch*

- if: $CI\_COMMIT\_BRANCH == $CI\_DEFAULT\_BRANCH *# Run for all changes to the default branch*

pages:

stage: deploy

dependencies:

- build-job

script:

- mv build/ public/

artifacts:

paths:

- "public/"

rules:

- if: $CI\_COMMIT\_BRANCH == $CI\_DEFAULT\_BRANCH *# Run for all changes to the default branch only*

Merge the changes in your merge request. This action updates the default branch. Verify that the new pipeline contains the pages job that deploys the site.

Be sure to use feature branches and merge requests for all future changes to pipeline configuration. Other project changes, like creating a Git tag or adding a pipeline schedule, do not trigger pipelines unless you add rules for those cases too.

## **Reduce duplicated configuration**

The pipeline now contains three jobs that all have identical rules and image configuration. Instead of repeating these rules, use extends and default to create single sources of truth.

This step introduces:

* [Hidden jobs](https://docs.gitlab.com/ee/ci/jobs/index.html" \l "hide-jobs): Jobs that start with . are never added to a pipeline. Use them to hold configuration you want to reuse.
* [extends](https://docs.gitlab.com/ee/ci/yaml/index.html" \l "extends): Use extends to repeat configuration in multiple places, often from hidden jobs. If you update the hidden job’s configuration, all jobs extending the hidden job use the updated configuration.
* [default](https://docs.gitlab.com/ee/ci/yaml/index.html" \l "default): Set keyword defaults that apply to all jobs when not defined.
* YAML overriding: When reusing configuration with extends or default, you can explicitly define a keyword in the job to override the extends or default configuration.

In this step:

* Add a .standard-rules hidden job to hold the rules that are repeated in build-job, lint-markdown, and test-html.
* Use extends to reuse the .standard-rules configuration in the three jobs.
* Add a default section to define the image default as node.
* The pages deployment job does not need the default node image, so explicitly use [busybox](https://hub.docker.com/_/busybox), an extremely tiny and fast image.

stages:

- build

- test

- deploy

default: *# Add a default section to define the `image` keyword's default value*

image: node

.standard-rules: *# Make a hidden job to hold the common rules*

rules:

- if: $CI\_PIPELINE\_SOURCE == 'merge\_request\_event'

- if: $CI\_COMMIT\_BRANCH == $CI\_DEFAULT\_BRANCH

build-job:

extends:

- .standard-rules *# Reuse the configuration in `.standard-rules` here*

stage: build

script:

- npm install

- npm run build

artifacts:

paths:

- "build/"

lint-markdown:

stage: test

extends:

- .standard-rules *# Reuse the configuration in `.standard-rules` here*

dependencies: []

script:

- npm install markdownlint-cli2 --global

- markdownlint-cli2 -v

- markdownlint-cli2 "blog/\*\*/\*.md" "docs/\*\*/\*.md"

allow\_failure: **true**

test-html:

stage: test

extends:

- .standard-rules *# Reuse the configuration in `.standard-rules` here*

dependencies:

- build-job

script:

- npm install --save-dev htmlhint

- npx htmlhint --version

- npx htmlhint build/

pages:

stage: deploy

image: busybox *# Override the default `image` value with `busybox`*

dependencies:

- build-job

script:

- mv build/ public/

artifacts:

paths:

- "public/"

rules:

- if: $CI\_COMMIT\_BRANCH == $CI\_DEFAULT\_BRANCH

Use a merge request to commit this pipeline configuration to the default branch. The file is simpler, but it should have the same behavior as the previous step.

You’ve just created a full pipeline and streamlined it to be more efficient. Nice work! Now you can take this knowledge, learn about [the rest of the .gitlab-ci.yml keywords](https://docs.gitlab.com/ee/ci/yaml/index.html), and build your own pipelines.

# **Plan a migration from another tool to GitLab CI/CD**

Before starting a migration from another tool to GitLab CI/CD, you should begin by developing a migration plan.

Review the advice on [managing organizational changes](https://docs.gitlab.com/ee/ci/migration/plan_a_migration.html" \l "manage-organizational-changes) first for advice on initial steps for larger migrations.

Users involved in the migration itself should review the [questions to ask before starting a migration](https://docs.gitlab.com/ee/ci/migration/plan_a_migration.html" \l "technical-questions-to-ask-before-starting-a-migration), as an important technical step for setting expectations. CI/CD tools differ in approach, structure, and technical specifics. While some concepts map one-to-one, others require interactive conversion.

It’s important to focus on your desired end state instead of strictly translating the behavior of your old tool.

## **Manage organizational changes**

An important part of transitioning to GitLab CI/CD is the cultural and organizational changes that come with the move, and successfully managing them.

A few things that organizations have reported as helping:

* Set and communicate a clear vision of what your migration goals are, which helps your users understand why the effort is worth it. The value is clear when the work is done, but people need to be aware while it’s in progress too.
* Sponsorship and alignment from the relevant leadership teams helps with the point above.
* Spend time educating your users on what’s different, and share this guide with them.
* Finding ways to sequence or delay parts of the migration can help a lot. Importantly though, try not to leave things in a non-migrated (or partially-migrated) state for too long.
* To gain all the benefits of GitLab, moving your existing configuration over as-is, including any current problems, isn’t enough. Take advantage of the improvements that GitLab CI/CD offers, and update your implementation as part of the transition.

## **Technical questions to ask before starting a migration**

Asking some initial technical questions about your CI/CD needs helps quickly define the migration requirements:

* How many projects use this pipeline?
* What branching strategy is used? Feature branches? Mainline? Release branches?
* What tools do you use to build your code? For example, Maven, Gradle, or NPM?
* What tools do you use to test your code? For example JUnit, Pytest, or Jest?
* Do you use any security scanners?
* Where do you store any built packages?
* How do you deploy your code?
* Where do you deploy your code?

# **Migrating from GitHub Actions**All tiersAll offerings

If you’re migrating from GitHub Actions to GitLab CI/CD, you are able to create CI/CD pipelines that replicate and enhance your GitHub Action workflows.

## **Key Similarities and Differences**

GitHub Actions and GitLab CI/CD are both used to generate pipelines to automate building, testing, and deploying your code. Both share similarities including:

* CI/CD functionality has direct access to the code stored in the project repository.
* Pipeline configurations written in YAML and stored in the project repository.
* Pipelines are configurable and can run in different stages.
* Jobs can each use a different container image.

Additionally, there are some important differences between the two:

* GitHub has a marketplace for downloading 3rd-party actions, which might require additional support or licenses.
* Self-managed GitLab instances support both horizontal and vertical scaling, while GitHub Enterprise Server only supports vertical scaling.
* GitLab maintains and supports all features in house, and some 3rd-party integrations are accessible through templates.
* GitLab provides a built-in container registry.
* GitLab has native Kubernetes deployment support.
* GitLab provides granular security policies.

## **Comparison of features and concepts**

Many GitHub features and concepts have equivalents in GitLab that offer the same functionality.

### **Configuration file**

GitHub Actions can be configured with a [workflow YAML file](https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions" \l "understanding-the-workflow-file). GitLab CI/CD uses a [.gitlab-ci.yml YAML file](https://docs.gitlab.com/ee/ci/yaml/gitlab_ci_yaml.html) by default.

For example, in a GitHub Actions workflow file:

on: [push]jobs:

hello:

runs-on: ubuntu-latest

steps:

- run: echo "Hello World"

The equivalent GitLab CI/CD .gitlab-ci.yml file would be:

stages:

- hello

hello:

stage: hello

script:

- echo "Hello World"

### **GitHub Actions workflow syntax**

A GitHub Actions configuration is defined in a workflow YAML file using specific keywords. GitLab CI/CD has similar functionality, also usually configured with YAML keywords.

| **GitHub** | **GitLab** | **Explanation** |
| --- | --- | --- |
| env | variables | env defines the variables set in a workflow, job, or step. GitLab uses variables to define [CI/CD variables](https://docs.gitlab.com/ee/ci/variables/index.html) at the global or job level. Variables can also be added in the UI. |
| jobs | stages | jobs groups together all the jobs that run in the workflow. GitLab uses stages to group jobs together. |
| on | Not applicable | on defines when a workflow is triggered. GitLab is integrated tightly with Git, so SCM polling options for triggers are not needed, but can be configured per job if required. |
| run | Not applicable | The command to execute in the job. GitLab uses a YAML array under the script keyword, one entry for each command to execute. |
| runs-on | tags | runs-on defines the GitHub runner that a job must run on. GitLab uses tags to select a runner. |
| steps | script | steps groups together all the steps that run in a job. GitLab uses script to group together all the commands run in a job. |
| uses | include | uses defines what GitHub Action to be added to a step. GitLab uses include to add configuration from other files to a job. |

### **Common configurations**

This section goes over commonly used CI/CD configurations, showing how they can be converted from GitHub Actions to GitLab CI/CD.

[GitHub Action workflows](https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions" \l "workflows) generate automated CI/CD jobs that are triggered when certain event take place, for example pushing a new commit. A GitHub Action workflow is a YAML file defined in the .github/workflows directory located in the root of the repository. The GitLab equivalent is the [.gitlab-ci.yml configuration file](https://docs.gitlab.com/ee/ci/yaml/gitlab_ci_yaml.html) which also resides in the repository’s root directory.

#### **Jobs**

Jobs are a set of commands that run in a set sequence to achieve a particular result, for example building a container or deploying to production.

For example, this GitHub Actions workflow builds a container then deploys it to production. The jobs runs sequentially, because the deploy job depends on the build job:

on: [push]jobs:

build:

runs-on: ubuntu-latest

container: golang:alpine

steps:

- run: apk update

- run: go build -o bin/hello

- uses: actions/upload-artifact@v3

with:

name: hello

path: bin/hello

retention-days: **7**

deploy:

if: contains( github.ref, 'staging')

runs-on: ubuntu-latest

container: golang:alpine

steps:

- uses: actions/download-artifact@v3

with:

name: hello

- run: echo "Deploying to Staging"

- run: scp bin/hello remoteuser@remotehost:/remote/directory

This example:

* Uses the golang:alpine container image.
* Runs a job for building code.
  + Stores build executable as artifact.
* Runs a second job to deploy to staging, which also:
  + Requires the build job to succeed before running.
  + Requires the commit target branch staging.
  + Uses the build executable artifact.

The equivalent GitLab CI/CD .gitlab-ci.yml file would be:

default:

image: golang:alpine

stages:

- build

- deploy

build-job:

stage: build

script:

- apk update

- go build -o bin/hello

artifacts:

paths:

- bin/hello

expire\_in: 1 week

deploy-job:

stage: deploy

script:

- echo "Deploying to Staging"

- scp bin/hello remoteuser@remotehost:/remote/directory

rules:

- if: $CI\_COMMIT\_BRANCH == 'staging'

##### **Parallel**

In both GitHub and GitLab, Jobs run in parallel by default.

For example, in a GitHub Actions workflow file:

on: [push]jobs:

python-version:

runs-on: ubuntu-latest

container: python:latest

steps:

- run: python --version

java-version:

if: contains( github.ref, 'staging')

runs-on: ubuntu-latest

container: openjdk:latest

steps:

- run: java -version

This example runs a Python job and a Java job in parallel, using different container images. The Java job only runs when the staging branch is changed.

The equivalent GitLab CI/CD .gitlab-ci.yml file would be:

python-version:

image: python:latest

script:

- python --version

java-version:

image: openjdk:latest

rules:

- if: $CI\_COMMIT\_BRANCH == 'staging'

script:

- java -version

In this case, no extra configuration is needed to make the jobs run in parallel. Jobs run in parallel by default, each on a different runner assuming there are enough runners for all the jobs. The Java job is set to only run when the staging branch is changed.

##### **Matrix**

In both GitLab and GitHub you can use a matrix to run a job multiple times in parallel in a single pipeline, but with different variable values for each instance of the job.

For example, in a GitHub Actions workflow file:

on: [push]jobs:

build:

runs-on: ubuntu-latest

steps:

- run: echo "Building $PLATFORM for $ARCH"

strategy:

matrix:

platform: [linux, mac, windows]

arch: [x64, x86]

test:

runs-on: ubuntu-latest

steps:

- run: echo "Testing $PLATFORM for $ARCH"

strategy:

matrix:

platform: [linux, mac, windows]

arch: [x64, x86]

deploy:

runs-on: ubuntu-latest

steps:

- run: echo "Deploying $PLATFORM for $ARCH"

strategy:

matrix:

platform: [linux, mac, windows]

arch: [x64, x86]

The equivalent GitLab CI/CD .gitlab-ci.yml file would be:

stages:

- build

- test

- deploy

.parallel-hidden-job:

parallel:

matrix:

- PLATFORM: [linux, mac, windows]

ARCH: [x64, x86]

build-job:

extends: .parallel-hidden-job

stage: build

script:

- echo "Building $PLATFORM for $ARCH"

test-job:

extends: .parallel-hidden-job

stage: test

script:

- echo "Testing $PLATFORM for $ARCH"

deploy-job:

extends: .parallel-hidden-job

stage: deploy

script:

- echo "Deploying $PLATFORM for $ARCH"

#### **Trigger**

GitHub Actions requires you to add a trigger for your workflow. GitLab is integrated tightly with Git, so SCM polling options for triggers are not needed, but can be configured per job if required.

Sample GitHub Actions configuration:

on:

push:

branches:

- main

The equivalent GitLab CI/CD configuration would be:

rules:

- if: '$CI\_COMMIT\_BRANCH == main'

Pipelines can also be [scheduled by using Cron syntax](https://docs.gitlab.com/ee/ci/pipelines/schedules.html).

#### **Container Images**

With GitLab you can [run your CI/CD jobs in separate, isolated Docker containers](https://docs.gitlab.com/ee/ci/docker/using_docker_images.html) by using the [image](https://docs.gitlab.com/ee/ci/yaml/index.html" \l "image) keyword.

For example, in a GitHub Actions workflow file:

jobs:

update:

runs-on: ubuntu-latest

container: alpine:latest

steps:

- run: apk update

In this example the apk update command runs in an alpine:latest container.

The equivalent GitLab CI/CD .gitlab-ci.yml file would be:

update-job:

image: alpine:latest

script:

- apk update

GitLab provides every project a [container registry](https://docs.gitlab.com/ee/user/packages/container_registry/index.html) for hosting container images. Container images can be built and stored directly from GitLab CI/CD pipelines.

For example:

stages:

- build

build-image:

stage: build

variables:

IMAGE: $CI\_REGISTRY\_IMAGE/$CI\_COMMIT\_REF\_SLUG:$CI\_COMMIT\_SHA

before\_script:

- docker login -u $CI\_REGISTRY\_USER -p $CI\_REGISTRY\_PASSWORD $CI\_REGISTRY

script:

- docker build -t $IMAGE .

- docker push $IMAGE

#### **Variables**

In GitLab, we use the variables keyword to define different [CI/CD variables](https://docs.gitlab.com/ee/ci/variables/index.html) at runtime. Use variables when you need to reuse configuration data in a pipeline. You can define variables globally or per job.

For example, in a GitHub Actions workflow file:

env:

NAME: "fern"

jobs:

english:

runs-on: ubuntu-latest

env:

Greeting: "hello"

steps:

- run: echo "$GREETING $NAME"

spanish:

runs-on: ubuntu-latest

env:

Greeting: "hola"

steps:

- run: echo "$GREETING $NAME"

In this example, variables provide different outputs for the jobs.

The equivalent GitLab CI/CD .gitlab-ci.yml file would be:

default:

image: ubuntu-latest

variables:

NAME: "fern"

english:

variables:

GREETING: "hello"

script:

- echo "$GREETING $NAME"

spanish:

variables:

GREETING: "hola"

script:

- echo "$GREETING $NAME"

Variables can also be set up through the GitLab UI, under CI/CD settings, where you can [protect](https://docs.gitlab.com/ee/ci/variables/index.html" \l "protect-a-cicd-variable) or [mask](https://docs.gitlab.com/ee/ci/variables/index.html" \l "mask-a-cicd-variable) the variables. Masked variables are hidden in job logs, while protected variables can only be accessed in pipelines for protected branches or tags.

For example, in a GitHub Actions workflow file:

jobs:

login:

runs-on: ubuntu-latest

env:

AWS\_ACCESS\_KEY: ${{ secrets.AWS\_ACCESS\_KEY }}

steps:

- run: my-login-script.sh "$AWS\_ACCESS\_KEY"

If the AWS\_ACCESS\_KEY variable is defined in the GitLab project settings, the equivalent GitLab CI/CD .gitlab-ci.yml file would be:

login:

script:

- my-login-script.sh $AWS\_ACCESS\_KEY

Additionally, [GitHub Actions](https://docs.github.com/en/actions/learn-github-actions/contexts) and [GitLab CI/CD](https://docs.gitlab.com/ee/ci/variables/predefined_variables.html) provide built-in variables which contain data relevant to the pipeline and repository.

#### **Conditionals**

When a new pipeline starts, GitLab checks the pipeline configuration to determine which jobs should run in that pipeline. You can use the [rules keyword](https://docs.gitlab.com/ee/ci/yaml/index.html" \l "rules) to configure jobs to run depending on conditions like the status of variables, or the pipeline type.

For example, in a GitHub Actions workflow file:

jobs:

deploy\_staging:

if: contains( github.ref, 'staging')

runs-on: ubuntu-latest

steps:

- run: echo "Deploy to staging server"

The equivalent GitLab CI/CD .gitlab-ci.yml file would be:

deploy\_staging:

stage: deploy

script:

- echo "Deploy to staging server"

rules:

- if: '$CI\_COMMIT\_BRANCH == staging'

#### **Runners**

Runners are the services that execute jobs. If you are using GitLab.com, you can use the [shared runner fleet](https://docs.gitlab.com/ee/ci/runners/index.html) to run jobs without provisioning your own self-managed runners.

Some key details about runners:

* Runners can be [configured](https://docs.gitlab.com/ee/ci/runners/runners_scope.html) to be shared across an instance, a group, or dedicated to a single project.
* You can use the [tags keyword](https://docs.gitlab.com/ee/ci/runners/configure_runners.html" \l "use-tags-to-control-which-jobs-a-runner-can-run) for finer control, and associate runners with specific jobs. For example, you can use a tag for jobs that require dedicated, more powerful, or specific hardware.
* GitLab has [autoscaling for runners](https://docs.gitlab.com/runner/configuration/autoscale.html). Use autoscaling to provision runners only when needed and scale down when not needed.

For example, in a GitHub Actions workflow file:

linux\_job:

runs-on: ubuntu-latest

steps:

- run: echo "Hello, $USER"

windows\_job:

runs-on: windows-latest

steps:

- run: echo "Hello, %USERNAME%"

The equivalent GitLab CI/CD .gitlab-ci.yml file would be:

linux\_job:

stage: build

tags:

- linux-runners

script:

- echo "Hello, $USER"

windows\_job:

stage: build

tags:

- windows-runners

script:

- echo "Hello, %USERNAME%"

#### **Artifacts**

In GitLab, any job can use the [artifacts](https://docs.gitlab.com/ee/ci/yaml/index.html" \l "artifacts) keyword to define a set of artifacts to be stored when a job completes. [Artifacts](https://docs.gitlab.com/ee/ci/jobs/job_artifacts.html) are files that can be used in later jobs.

For example, in a GitHub Actions workflow file:

on: [push]jobs:

generate\_cat:

steps:

- run: touch cat.txt

- run: echo "meow" > cat.txt

- uses: actions/upload-artifact@v3

with:

name: cat

path: cat.txt

retention-days: **7**

use\_cat:

needs: [generate\_cat]

steps:

- uses: actions/download-artifact@v3

with:

name: cat

- run: cat cat.txt

The equivalent GitLab CI/CD .gitlab-ci.yml file would be:

stage:

- generate

- use

generate\_cat:

stage: generate

script:

- touch cat.txt

- echo "meow" > cat.txt

artifacts:

paths:

- cat.txt

expire\_in: 1 week

use\_cat:

stage: use

script:

- cat cat.txt

#### **Caching**

A [cache](https://docs.gitlab.com/ee/ci/caching/index.html) is created when a job downloads one or more files and saves them for faster access in the future. Subsequent jobs that use the same cache don’t have to download the files again, so they execute more quickly. The cache is stored on the runner and uploaded to S3 if [distributed cache is enabled](https://docs.gitlab.com/runner/configuration/autoscale.html" \l "distributed-runners-caching).

For example, in a GitHub Actions workflow file:

jobs:

build:

runs-on: ubuntu-latest

steps:

- run: echo "This job uses a cache."

- uses: actions/cache@v3

with:

path: binaries/

key: binaries-cache-$CI\_COMMIT\_REF\_SLUG

The equivalent GitLab CI/CD .gitlab-ci.yml file would be:

cache-job:

script:

- echo "This job uses a cache."

cache:

key: binaries-cache-$CI\_COMMIT\_REF\_SLUG

paths:

- binaries/

#### **Templates**

In GitHub an Action is a set of complex tasks that need to be frequently repeated and is saved to enable reuse without redefining a CI/CD pipeline. In GitLab the equivalent to an action would be a the [include keyword](https://docs.gitlab.com/ee/ci/yaml/includes.html), which allows you to [add CI/CD pipelines from other files](https://docs.gitlab.com/ee/ci/yaml/includes.html), including template files built into GitLab.

Sample GitHub Actions configuration:

- uses: hashicorp/setup-terraform@v2.0.3

The equivalent GitLab CI/CD configuration would be:

include:

- template: Terraform.gitlab-ci.yml

In these examples, the setup-terraform GitHub action and the Terraform.gitlab-ci.yml GitLab template are not exact matches. These two examples are just to show how complex configuration can be reused.

### **Security Scanning features**

GitLab provides a variety of [security scanners](https://docs.gitlab.com/ee/user/application_security/index.html) out-of-the-box to detect vulnerabilities in all parts of the SLDC. You can add these features to your GitLab CI/CD pipeline by using templates.

for example to add SAST scanning to your pipeline, add the following to your .gitlab-ci.yml:

include:

- template: Security/SAST.gitlab-ci.yml

You can customize the behavior of security scanners by using CI/CD variables, for example with the [SAST scanners](https://docs.gitlab.com/ee/user/application_security/sast/index.html" \l "available-cicd-variables).

### **Secrets Management**

Privileged information, often referred to as “secrets”, is sensitive information or credentials you need in your CI/CD workflow. You might use secrets to unlock protected resources or sensitive information in tools, applications, containers, and cloud-native environments.

For secrets management in GitLab, you can use one of the supported integrations for an external service. These services securely store secrets outside of your GitLab project, though you must have a subscription for the service:

* [HashiCorp Vault](https://docs.gitlab.com/ee/ci/secrets/id_token_authentication.html" \l "automatic-id-token-authentication-with-hashicorp-vault).
* [Azure Key Vault](https://docs.gitlab.com/ee/ci/secrets/azure_key_vault.html).

GitLab also supports [OIDC authentication](https://docs.gitlab.com/ee/ci/secrets/id_token_authentication.html) for other third party services that support OIDC.

Additionally, you can make credentials available to jobs by storing them in CI/CD variables, though secrets stored in plain text are susceptible to accidental exposure. You should always store sensitive information in [masked](https://docs.gitlab.com/ee/ci/variables/index.html" \l "mask-a-cicd-variable) and [protected](https://docs.gitlab.com/ee/ci/variables/index.html" \l "protect-a-cicd-variable) variables, which mitigates some of the risk.

Also, never store secrets as variables in your .gitlab-ci.yml file, which is public to all users with access to the project. Storing sensitive information in variables should only be done in [the project, group, or instance settings](https://docs.gitlab.com/ee/ci/variables/index.html" \l "define-a-cicd-variable-in-the-ui).

Review the [security guidelines](https://docs.gitlab.com/ee/ci/variables/index.html" \l "cicd-variable-security) to improve the safety of your CI/CD variables.

## **Planning and Performing a Migration**

The following list of recommended steps was created after observing organizations that were able to quickly complete this migration.

### **Create a Migration Plan**

Before starting a migration you should create a [migration plan](https://docs.gitlab.com/ee/ci/migration/plan_a_migration.html) to make preparations for the migration.

### **Prerequisites**

Before doing any migration work, you should first:

1. Get familiar with GitLab.
   * Read about the [key GitLab CI/CD features](https://docs.gitlab.com/ee/ci/index.html).
   * Follow tutorials to create [your first GitLab pipeline](https://docs.gitlab.com/ee/ci/quick_start/index.html) and [more complex pipelines](https://docs.gitlab.com/ee/ci/quick_start/tutorial.html) that build, test, and deploys a static site.
   * Review the [.gitlab-ci.yml keyword reference](https://docs.gitlab.com/ee/ci/yaml/index.html).
2. Set up and configure GitLab.
3. Test your GitLab instance.
   * Ensure [runners](https://docs.gitlab.com/ee/ci/runners/index.html) are available, either by using shared GitLab.com runners or installing new runners.

### **Migration Steps**

1. Migrate Projects from GitHub to GitLab:
   * (Recommended) You can use the [GitHub Importer](https://docs.gitlab.com/ee/user/project/import/github.html) to automate mass imports from external SCM providers.
   * You can [import repositories by URL](https://docs.gitlab.com/ee/user/project/import/repo_by_url.html).
2. Create a .gitlab-ci.yml in each project.
3. Migrate GitHub Actions jobs to GitLab CI/CD jobs and configure them to show results directly in merge requests.
4. Migrate deployment jobs by using [cloud deployment templates](https://docs.gitlab.com/ee/ci/cloud_deployment/index.html), [environments](https://docs.gitlab.com/ee/ci/environments/index.html), and the [GitLab agent for Kubernetes](https://docs.gitlab.com/ee/user/clusters/agent/index.html).
5. Check if any CI/CD configuration can be reused across different projects, then create and share [CI/CD templates](https://docs.gitlab.com/ee/development/cicd/templates.html)
6. Check the [pipeline efficiency documentation](https://docs.gitlab.com/ee/ci/pipelines/pipeline_efficiency.html) to learn how to make your GitLab CI/CD pipelines faster and more efficient.

### **Additional Resources**

* [Video: How to migrate from GitHub to GitLab including Actions](https://youtu.be/0Id5oMl1Kqs?feature=shared)
* [Blog: GitHub to GitLab migration the easy way](https://about.gitlab.com/blog/2023/07/11/github-to-gitlab-migration-made-easy/)

If you have questions that are not answered here, the [GitLab community forum](https://forum.gitlab.com/) can be a great resource.

# **Migrating from Jenkins**All tiersAll offerings

If you’re migrating from Jenkins to GitLab CI/CD, you are able to create CI/CD pipelines that replicate and enhance your Jenkins workflows.

## **Key similarities and differences**

GitLab CI/CD and Jenkins are CI/CD tools with some similarities. Both GitLab and Jenkins:

* Use stages for collections of jobs.
* Support container-based builds.

Additionally, there are some important differences between the two:

* GitLab CI/CD pipelines are all configured in a YAML format configuration file. Jenkins uses either a Groovy format configuration file (declarative pipelines) or Jenkins DSL (scripted pipelines).
* GitLab can run either on SaaS (cloud) or self-managed deployments. Jenkins deployments must be self-managed.
* GitLab provides source code management (SCM) out of the box. Jenkins requires a separate SCM solution to store code.
* GitLab provides a built-in container registry. Jenkins requires a separate solution for storing container images.
* GitLab provides built-in templates for scanning code. Jenkins requires 3rd party plugins for scanning code.

## **Comparison of features and concepts**

Many Jenkins features and concepts have equivalents in GitLab that offer the same functionality.

### **Configuration file**

Jenkins can be configured with a [Jenkinsfile in the Groovy format](https://www.jenkins.io/doc/book/pipeline/jenkinsfile/). GitLab CI/CD uses a [.gitlab-ci.yml YAML file](https://docs.gitlab.com/ee/ci/yaml/gitlab_ci_yaml.html) by default.

Example of a Jenkinsfile:

pipeline {

agent any

stages {

stage('hello') {

steps {

echo "Hello World"

}

}

}}

The equivalent GitLab CI/CD .gitlab-ci.yml file would be:

stages:

- hello

hello-job:

stage: hello

script:

- echo "Hello World"

### **Jenkins pipeline syntax**

A Jenkins configuration is composed of a pipeline block with “sections” and “directives”. GitLab CI/CD has similar functionality, configured with YAML keywords.

#### **Sections**

| **Jenkins** | **GitLab** | **Explanation** |
| --- | --- | --- |
| agent | image | Jenkins pipelines execute on agents, and the agent section defines how the pipeline executes, and the Docker container to use. GitLab jobs execute on runners, and the image keyword defines the container to use. You can configure your own runners in Kubernetes or on any host. |
| post | after\_script or stage | The Jenkins post section defines actions that should be performed at the end of a stage or pipeline. In GitLab, use after\_script for commands to run at the end of a job, and before\_script for actions to run before the other commands in a job. Use stage to select the exact stage a job should run in. GitLab supports both .pre and .post stages that always run before or after all other defined stages. |
| stages | stages | Jenkins stages are groups of jobs. GitLab CI/CD also uses stages, but it is more flexible. You can have multiple stages each with multiple independent jobs. Use stages at the top level to the stages and their execution order, and use stage at the job level to define the stage for that job. |
| steps | script | Jenkins steps define what to execute. GitLab CI/CD uses a script section which is similar. The script section is a YAML array with separate entries for each command to run in sequence. |

#### **Directives**

| **Jenkins** | **GitLab** | **Explanation** |
| --- | --- | --- |
| environment | variables | Jenkins uses environment for environment variables. GitLab CI/CD uses the variables keyword to define CI/CD variables that can be used during job execution, but also for more dynamic pipeline configuration. These can also be set in the GitLab UI, under CI/CD settings. |
| options | Not applicable | Jenkins uses options for additional configuration, including timeouts and retry values. GitLab does not need a separate section for options, all configuration is added as CI/CD keywords at the job or pipeline level, for example timeout or retry. |
| parameters | Not applicable | In Jenkins, parameters can be required when triggering a pipeline. Parameters are handled in GitLab with CI/CD variables, which can be defined in many places, including the pipeline configuration, project settings, at runtime manually through the UI, or API. |
| triggers | rules | In Jenkins, triggers defines when a pipeline should run again, for example through cron notation. GitLab CI/CD can run pipelines automatically for many reasons, including Git changes and merge request updates. Use the rules keyword to control which events to run jobs for. Scheduled pipelines are defined in the project settings. |
| tools | Not applicable | In Jenkins, tools defines additional tools to install in the environment. GitLab does not have a similar keyword, as the recommendation is to use container images prebuilt with the exact tools required for your jobs. These images can be cached and can be built to already contain the tools you need for your pipelines. If a job needs additional tools, they can be installed as part of a before\_script section. |
| input | Not applicable | In Jenkins, input adds a prompt for user input. Similar to parameters, inputs are handled in GitLab through CI/CD variables. |
| when | rules | In Jenkins, when defines when a stage should be executed. GitLab also has a when keyword, which defines whether a job should start running based on the status of earlier jobs, for example if jobs passed or failed. To control when to add jobs to specific pipelines, use rules. |

### **Common configurations**

This section goes over commonly used CI/CD configurations, showing how they can be converted from Jenkins to GitLab CI/CD.

[Jenkins pipelines](https://www.jenkins.io/doc/book/pipeline/) generate automated CI/CD jobs that are triggered when certain event take place, such as a new commit being pushed. A Jenkins pipeline is defined in a Jenkinsfile. The GitLab equivalent is the [.gitlab-ci.yml configuration file](https://docs.gitlab.com/ee/ci/yaml/gitlab_ci_yaml.html).

Jenkins does not provide a place to store source code, so the Jenkinsfile must be stored in a separate source control repository.

#### **Jobs**

Jobs are a set of commands that run in a set sequence to achieve a particular result.

For example, build a container then deploy it to production, in a Jenkinsfile:

pipeline {

agent any

stages {

stage('build') {

agent { docker 'golang:alpine' }

steps {

apk update

go build -o bin/hello

}

post {

always {

archiveArtifacts artifacts: 'bin/hello'

onlyIfSuccessful: **true**

}

}

}

stage('deploy') {

agent { docker 'golang:alpine' }

when {

branch 'staging'

}

steps {

echo "Deploying to staging"

scp bin/hello remoteuser@remotehost:/remote/directory

}

}

}}

This example:

* Uses the golang:alpine container image.
* Runs a job for building code.
  + Stores the built executable as an artifact.
* Adds a second job to deploy to staging, which:
  + Only exists if the commit targets the staging branch.
  + Starts after the build stage succeeds.
  + Uses the built executable artifact from the earlier job.

The equivalent GitLab CI/CD .gitlab-ci.yml file would be:

default:

image: golang:alpine

stages:

- build

- deploy

build-job:

stage: build

script:

- apk update

- go build -o bin/hello

artifacts:

paths:

- bin/hello

expire\_in: 1 week

deploy-job:

stage: deploy

script:

- echo "Deploying to Staging"

- scp bin/hello remoteuser@remotehost:/remote/directory

rules:

- if: $CI\_COMMIT\_BRANCH == 'staging'

artifacts:

paths:

- bin/hello

##### **Parallel**

In Jenkins, jobs that are not dependent on previous jobs can run in parallel when added to a parallel section.

For example, in a Jenkinsfile:

pipeline {

agent any

stages {

stage('Parallel') {

parallel {

stage('Python') {

agent { docker 'python:latest' }

steps {

sh "python --version"

}

}

stage('Java') {

agent { docker 'openjdk:latest' }

when {

branch 'staging'

}

steps {

sh "java -version"

}

}

}

}

}}

This example runs a Python and a Java job in parallel, using different container images. The Java job only runs when the staging branch is changed.

The equivalent GitLab CI/CD .gitlab-ci.yml file would be:

python-version:

image: python:latest

script:

- python --version

java-version:

image: openjdk:latest

rules:

- if: $CI\_COMMIT\_BRANCH == 'staging'

script:

- java -version

In this case, no extra configuration is needed to make the jobs run in parallel. Jobs run in parallel by default, each on a different runner assuming there are enough runners for all the jobs. The Java job is set to only run when the staging branch is changed.

##### **Matrix**

In GitLab you can use a matrix to run a job multiple times in parallel in a single pipeline, but with different variable values for each instance of the job. Jenkins runs the matrix sequentially.

For example, in a Jenkinsfile:

matrix {

axes {

axis {

name 'PLATFORM'

values 'linux', 'mac', 'windows'

}

axis {

name 'ARCH'

values 'x64', 'x86'

}

}

stages {

stage('build') {

echo "Building $PLATFORM for $ARCH"

}

stage('test') {

echo "Building $PLATFORM for $ARCH"

}

stage('deploy') {

echo "Building $PLATFORM for $ARCH"

}

}}

The equivalent GitLab CI/CD .gitlab-ci.yml file would be:

stages:

- build

- test

- deploy

.parallel-hidden-job:

parallel:

matrix:

- PLATFORM: [linux, mac, windows]

ARCH: [x64, x86]

build-job:

extends: .parallel-hidden-job

stage: build

script:

- echo "Building $PLATFORM for $ARCH"

test-job:

extends: .parallel-hidden-job

stage: test

script:

- echo "Testing $PLATFORM for $ARCH"

deploy-job:

extends: .parallel-hidden-job

stage: deploy

script:

- echo "Testing $PLATFORM for $ARCH"

#### **Container Images**

In GitLab you can [run your CI/CD jobs in separate, isolated Docker containers](https://docs.gitlab.com/ee/ci/docker/using_docker_images.html) using the [image](https://docs.gitlab.com/ee/ci/yaml/index.html" \l "image) keyword.

For example, in a Jenkinsfile:

stage('Version') {

agent { docker 'python:latest' }

steps {

echo 'Hello Python'

sh 'python --version'

}}

This example shows commands running in a python:latest container.

The equivalent GitLab CI/CD .gitlab-ci.yml file would be:

version-job:

image: python:latest

script:

- echo "Hello Python"

- python --version

#### **Variables**

In GitLab, use the variables keyword to define [CI/CD variables](https://docs.gitlab.com/ee/ci/variables/index.html). Use variables to reuse configuration data, have more dynamic configuration, or store important values. Variables can be defined either globally or per job.

For example, in a Jenkinsfile:

pipeline {

agent any

environment {

NAME = 'Fern'

}

stages {

stage('English') {

environment {

GREETING = 'Hello'

}

steps {

sh 'echo "$GREETING $NAME"'

}

}

stage('Spanish') {

environment {

GREETING = 'Hola'

}

steps {

sh 'echo "$GREETING $NAME"'

}

}

}}

This example shows how variables can be used to pass values to commands in jobs.

The equivalent GitLab CI/CD .gitlab-ci.yml file would be:

default:

image: alpine:latest

stages:

- greet

variables:

NAME: "Fern"

english:

stage: greet

variables:

GREETING: "Hello"

script:

- echo "$GREETING $NAME"

spanish:

stage: greet

variables:

GREETING: "Hola"

script:

- echo "$GREETING $NAME"

Variables can also be [set in the GitLab UI, in the CI/CD settings](https://docs.gitlab.com/ee/ci/variables/index.html" \l "define-a-cicd-variable-in-the-ui). In some cases, you can use [protected](https://docs.gitlab.com/ee/ci/variables/index.html" \l "protect-a-cicd-variable) and [masked](https://docs.gitlab.com/ee/ci/variables/index.html" \l "mask-a-cicd-variable) variables for secret values. These variables can be accessed in pipeline jobs the same as variables defined in the configuration file.

For example, in a Jenkinsfile:

pipeline {

agent any

stages {

stage('Example Username/Password') {

environment {

AWS\_ACCESS\_KEY = credentials('aws-access-key')

}

steps {

sh 'my-login-script.sh $AWS\_ACCESS\_KEY'

}

}

}}

The equivalent GitLab CI/CD .gitlab-ci.yml file would be:

login-job:

script:

- my-login-script.sh $AWS\_ACCESS\_KEY

Additionally, GitLab CI/CD makes [predefined variables](https://docs.gitlab.com/ee/ci/variables/predefined_variables.html) available to every pipeline and job which contain values relevant to the pipeline and repository.

#### **Expressions and conditionals**

When a new pipeline starts, GitLab checks which jobs should run in that pipeline. You can configure jobs to run depending on factors like the status of variables, or the pipeline type.

For example, in a Jenkinsfile:

stage('deploy\_staging') {

agent { docker 'alpine:latest' }

when {

branch 'staging'

}

steps {

echo "Deploying to staging"

}}

In this example, the job only runs when the branch we are committing to is named staging.

The equivalent GitLab CI/CD .gitlab-ci.yml file would be:

deploy\_staging:

stage: deploy

script:

- echo "Deploy to staging server"

rules:

- if: '$CI\_COMMIT\_BRANCH == staging'

#### **Runners**

Like Jenkins agents, GitLab runners are the hosts that run jobs. If you are using GitLab.com, you can use the [shared runner fleet](https://docs.gitlab.com/ee/ci/runners/index.html) to run jobs without provisioning your own runners.

To convert a Jenkins agent for use with GitLab CI/CD, uninstall the agent and then [install and register a runner](https://docs.gitlab.com/ee/ci/runners/index.html). Runners do not require much overhead, so you might be able to use similar provisioning as the Jenkins agents you were using.

Some key details about runners:

* Runners can be [configured](https://docs.gitlab.com/ee/ci/runners/runners_scope.html) to be shared across an instance, a group, or dedicated to a single project.
* You can use the [tags keyword](https://docs.gitlab.com/ee/ci/runners/configure_runners.html" \l "use-tags-to-control-which-jobs-a-runner-can-run) for finer control, and associate runners with specific jobs. For example, you can use a tag for jobs that require dedicated, more powerful, or specific hardware.
* GitLab has [autoscaling for runners](https://docs.gitlab.com/runner/configuration/autoscale.html). Use autoscaling to provision runners only when needed and scale down when not needed.

For example, in a Jenkinsfile:

pipeline {

agent none

stages {

stage('Linux') {

agent {

label 'linux'

}

steps {

echo "Hello, $USER"

}

}

stage('Windows') {

agent {

label 'windows'

}

steps {

echo "Hello, %USERNAME%"

}

}

}}

The equivalent GitLab CI/CD .gitlab-ci.yml file would be:

linux\_job:

stage: build

tags:

- linux

script:

- echo "Hello, $USER"

windows\_job:

stage: build

tags:

- windows

script:

- echo "Hello, %USERNAME%"

#### **Artifacts**

In GitLab, any job can use the [artifacts](https://docs.gitlab.com/ee/ci/yaml/index.html" \l "artifacts) keyword to define a set of artifacts to be stored when a job completes. [Artifacts](https://docs.gitlab.com/ee/ci/jobs/job_artifacts.html) are files that can be used in later jobs, for example for testing or deployment.

For example, in a Jenkinsfile:

stages {

stage('Generate Cat') {

steps {

sh 'touch cat.txt'

sh 'echo "meow" > cat.txt'

}

post {

always {

archiveArtifacts artifacts: 'cat.txt'

onlyIfSuccessful: **true**

}

}

}

stage('Use Cat') {

steps {

sh 'cat cat.txt'

}

}

}

The equivalent GitLab CI/CD .gitlab-ci.yml file would be:

stages:

- generate

- use

generate\_cat:

stage: generate

script:

- touch cat.txt

- echo "meow" > cat.txt

artifacts:

paths:

- cat.txt

expire\_in: 1 week

use\_cat:

stage: use

script:

- cat cat.txt

artifacts:

paths:

- cat.txt

#### **Caching**

A [cache](https://docs.gitlab.com/ee/ci/caching/index.html) is created when a job downloads one or more files and saves them for faster access in the future. Subsequent jobs that use the same cache don’t have to download the files again, so they execute more quickly. The cache is stored on the runner and uploaded to S3 if [distributed cache is enabled](https://docs.gitlab.com/runner/configuration/autoscale.html" \l "distributed-runners-caching). Jenkins core does not provide caching.

For example, in a .gitlab-ci.yml file:

cache-job:

script:

- echo "This job uses a cache."

cache:

key: binaries-cache-$CI\_COMMIT\_REF\_SLUG

paths:

- binaries/

### **Jenkins plugins**

Some functionality in Jenkins that is enabled through plugins is supported natively in GitLab with keywords and features that offer similar functionality. For example:

| **Jenkins plugin** | **GitLab feature** |
| --- | --- |
| [Build Timeout](https://plugins.jenkins.io/build-timeout/) | [timeout keyword](https://docs.gitlab.com/ee/ci/yaml/index.html" \l "timeout) |
| [Cobertura](https://plugins.jenkins.io/cobertura/) | [Coverage report artifacts](https://docs.gitlab.com/ee/ci/yaml/artifacts_reports.html" \l "artifactsreportscoverage_report) and [Code coverage](https://docs.gitlab.com/ee/ci/testing/code_coverage.html) |
| [Code coverage API](https://plugins.jenkins.io/code-coverage-api/) | [Code coverage](https://docs.gitlab.com/ee/ci/testing/code_coverage.html) and [Test coverage visualization](https://docs.gitlab.com/ee/ci/testing/test_coverage_visualization.html) |
| [Embeddable Build Status](https://plugins.jenkins.io/embeddable-build-status/) | [Pipeline status badges](https://docs.gitlab.com/ee/user/project/badges.html" \l "pipeline-status-badges) |
| [JUnit](https://plugins.jenkins.io/junit/) | [JUnit test report artifacts](https://docs.gitlab.com/ee/ci/yaml/artifacts_reports.html" \l "artifactsreportsjunit) and [Unit test reports](https://docs.gitlab.com/ee/ci/testing/unit_test_reports.html) |
| [Mailer](https://plugins.jenkins.io/mailer/) | [Notification emails](https://docs.gitlab.com/ee/user/profile/notifications.html) |
| [Parameterized Trigger Plugin](https://plugins.jenkins.io/parameterized-trigger/) | [trigger keyword](https://docs.gitlab.com/ee/ci/yaml/index.html" \l "trigger) and [downstream pipelines](https://docs.gitlab.com/ee/ci/pipelines/downstream_pipelines.html) |
| [Role-based Authorization Strategy](https://plugins.jenkins.io/role-strategy/) | GitLab [permissions and roles](https://docs.gitlab.com/ee/user/permissions.html) |
| [Timestamper](https://plugins.jenkins.io/timestamper/) | [Job](https://docs.gitlab.com/ee/ci/jobs/index.html) logs are time stamped by default |

### **Security Scanning features**

You might have used plugins for things like code quality, security, or static application scanning in Jenkins. GitLab provides [security scanners](https://docs.gitlab.com/ee/user/application_security/index.html) out-of-the-box to detect vulnerabilities in all parts of the SDLC. You can add these plugins in GitLab using templates, for example to add SAST scanning to your pipeline, add the following to your .gitlab-ci.yml:

include:

- template: Security/SAST.gitlab-ci.yml

You can customize the behavior of security scanners by using CI/CD variables, for example with the [SAST scanners](https://docs.gitlab.com/ee/user/application_security/sast/index.html" \l "available-cicd-variables).

### **Secrets Management**

Privileged information, often referred to as “secrets”, is sensitive information or credentials you need in your CI/CD workflow. You might use secrets to unlock protected resources or sensitive information in tools, applications, containers, and cloud-native environments.

Secrets management in Jenkins is usually handled with the Secret type field or the Credentials Plugin. Credentials stored in the Jenkins settings can be exposed to jobs as environment variables by using the Credentials Binding plugin.

For secrets management in GitLab, you can use one of the supported integrations for an external service. These services securely store secrets outside of your GitLab project, though you must have a subscription for the service:

* [HashiCorp Vault](https://docs.gitlab.com/ee/ci/secrets/id_token_authentication.html" \l "automatic-id-token-authentication-with-hashicorp-vault)
* [Azure Key Vault](https://docs.gitlab.com/ee/ci/secrets/azure_key_vault.html).

GitLab also supports [OIDC authentication](https://docs.gitlab.com/ee/ci/secrets/id_token_authentication.html) for other third party services that support OIDC.

Additionally, you can make credentials available to jobs by storing them in CI/CD variables, though secrets stored in plain text are susceptible to accidental exposure, [the same as in Jenkins](https://www.jenkins.io/doc/developer/security/secrets/" \l "storing-secrets). You should always store sensitive information in [masked](https://docs.gitlab.com/ee/ci/variables/index.html" \l "mask-a-cicd-variable) and [protected](https://docs.gitlab.com/ee/ci/variables/index.html" \l "protect-a-cicd-variable) variables, which mitigates some of the risk.

Also, never store secrets as variables in your .gitlab-ci.yml file, which is public to all users with access to the project. Storing sensitive information in variables should only be done in [the project, group, or instance settings](https://docs.gitlab.com/ee/ci/variables/index.html" \l "define-a-cicd-variable-in-the-ui).

Review the [security guidelines](https://docs.gitlab.com/ee/ci/variables/index.html" \l "cicd-variable-security) to improve the safety of your CI/CD variables.

## **Planning and Performing a Migration**

The following list of recommended steps was created after observing organizations that were able to quickly complete this migration.

### **Create a Migration Plan**

Before starting a migration you should create a [migration plan](https://docs.gitlab.com/ee/ci/migration/plan_a_migration.html) to make preparations for the migration. For a migration from Jenkins, ask yourself the following questions in preparation:

* What plugins are used by jobs in Jenkins today?
  + Do you know what these plugins do exactly?
  + Do any plugins wrap a common build tool? For example, Maven, Gradle, or NPM?
* What is installed on the Jenkins agents?
* Are there any shared libraries in use?
* How are you authenticating from Jenkins? Are you using SSH keys, API tokens, or other secrets?
* Are there other projects that you need to access from your pipeline?
* Are there credentials in Jenkins to access outside services? For example Ansible Tower, Artifactory, or other Cloud Providers or deployment targets?

### **Prerequisites**

Before doing any migration work, you should first:

1. Get familiar with GitLab.
   * Read about the [key GitLab CI/CD features](https://docs.gitlab.com/ee/ci/index.html).
   * Follow tutorials to create [your first GitLab pipeline](https://docs.gitlab.com/ee/ci/quick_start/index.html) and [more complex pipelines](https://docs.gitlab.com/ee/ci/quick_start/tutorial.html) that build, test, and deploys a static site.
   * Review the [.gitlab-ci.yml keyword reference](https://docs.gitlab.com/ee/ci/yaml/index.html).
2. Set up and configure GitLab.
3. Test your GitLab instance.
   * Ensure [runners](https://docs.gitlab.com/ee/ci/runners/index.html) are available, either by using shared GitLab.com runners or installing new runners.

### **Migration Steps**

1. Migrate projects from your SCM solution to GitLab.
   * (Recommended) You can use the available [importers](https://docs.gitlab.com/ee/user/project/import/index.html) to automate mass imports from external SCM providers.
   * You can [import repositories by URL](https://docs.gitlab.com/ee/user/project/import/repo_by_url.html).
2. Create a .gitlab-ci.yml file in each project.
3. Migrate Jenkins configuration to GitLab CI/CD jobs and configure them to show results directly in merge requests.
4. Migrate deployment jobs by using [cloud deployment templates](https://docs.gitlab.com/ee/ci/cloud_deployment/index.html), [environments](https://docs.gitlab.com/ee/ci/environments/index.html), and the [GitLab agent for Kubernetes](https://docs.gitlab.com/ee/user/clusters/agent/index.html).
5. Check if any CI/CD configuration can be reused across different projects, then create and share CI/CD templates.
6. Check the [pipeline efficiency documentation](https://docs.gitlab.com/ee/ci/pipelines/pipeline_efficiency.html) to learn how to make your GitLab CI/CD pipelines faster and more efficient.

### **Additional Resources**

You can use the [JenkinsFile Wrapper](https://gitlab.com/gitlab-org/jfr-container-builder/) to run a complete Jenkins instance inside of a GitLab CI/CD job, including plugins. Use this tool to help ease the transition to GitLab CI/CD, by delaying the migration of less urgent pipelines.

The JenkinsFile Wrapper is not packaged with GitLab and falls outside of the scope of support. For more information, see the [Statement of Support](https://about.gitlab.com/support/statement-of-support/).

If you have questions that are not answered here, the [GitLab community forum](https://forum.gitlab.com/) can be a great resource.

# **Migrate a Maven build from Jenkins to GitLab CI/CD**

If you have a Maven build in Jenkins, you can use a [Java Spring](https://gitlab.com/gitlab-org/project-templates/spring) project template to migrate to GitLab. The template uses Maven for its underlying dependency management.

## **Sample Jenkins configurations**

The following three Jenkins examples each use different methods to test, build, and install a Maven project into a shell agent:

* Freestyle with shell execution
* Freestyle with the Maven task plugin
* A declarative pipeline using a Jenkinsfile

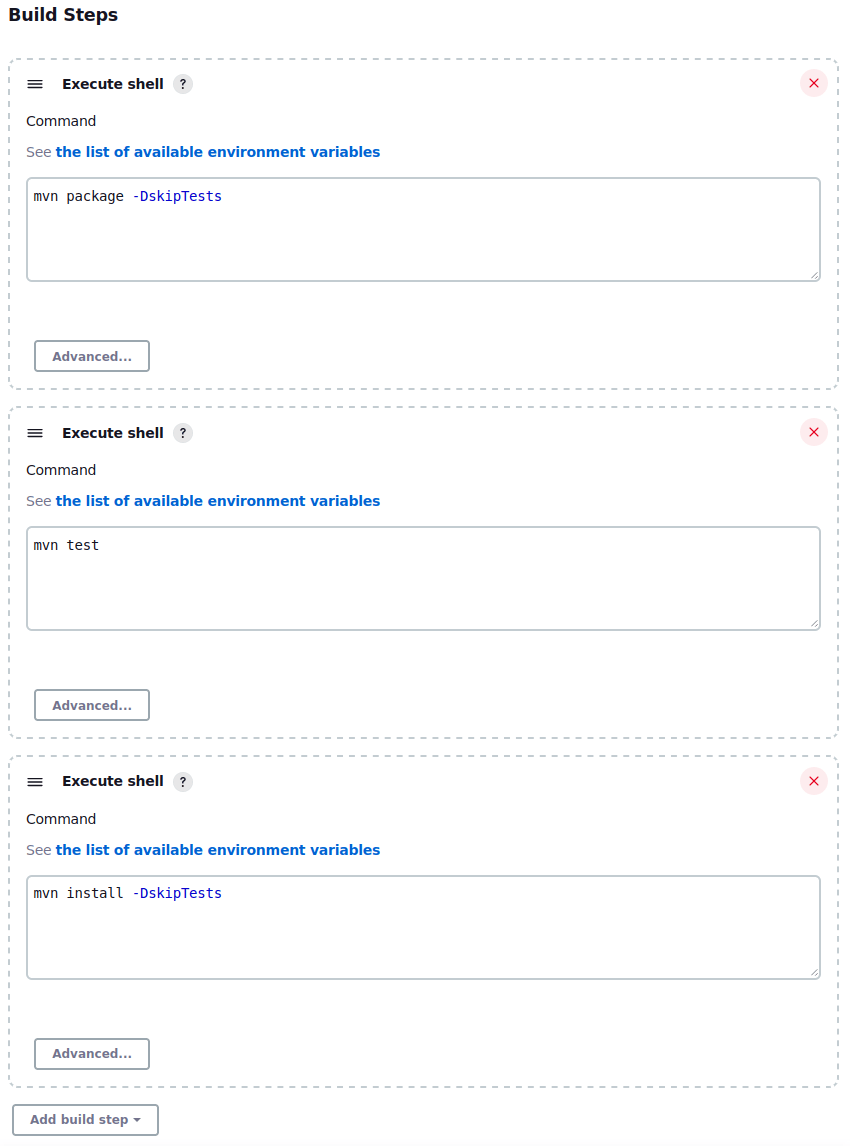
All three examples run the same three commands in order, in three different stages:

* mvn test: Run any tests found in the codebase
* mvn package -DskipTests: Compile the code into an executable type defined in the POM and skip running any tests because that was done in the first stage.
* mvn install -DskipTests: Install the compiled executable into the agent’s local Maven .m2 repository and again skip running the tests.

These examples use a single, persistent Jenkins agent, which requires Maven to be pre-installed on the agent. This method of execution is similar to a GitLab Runner using the [shell executor](https://docs.gitlab.com/runner/executors/shell.html).

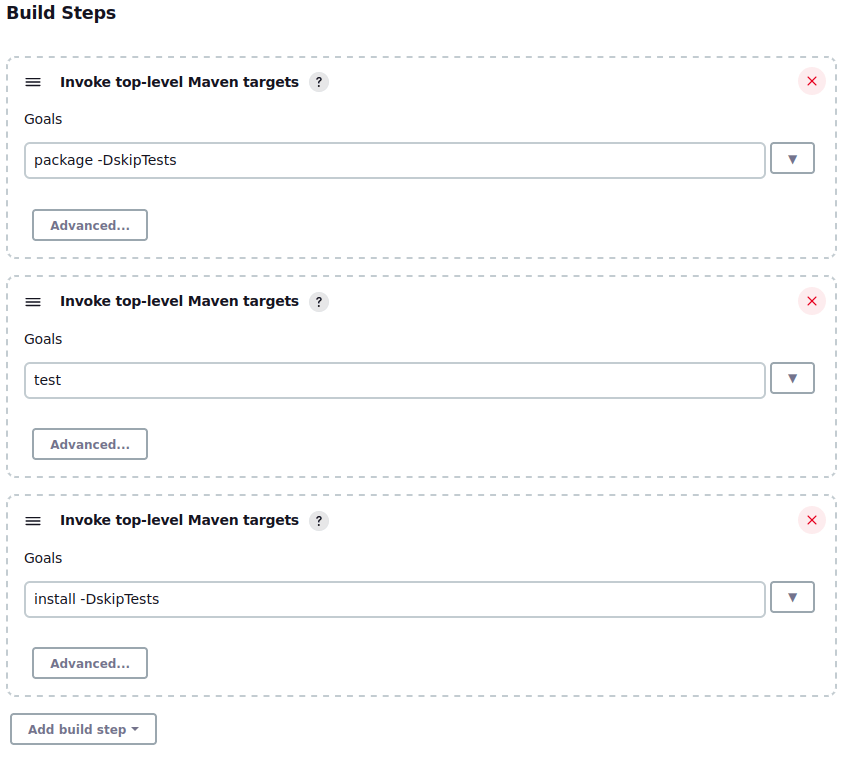
### **Freestyle with shell execution**

If using Jenkins’ built-in shell execution option to directly call mvn commands from the shell on the agent, the configuration might look like:

[](https://docs.gitlab.com/ee/ci/migration/examples/img/maven-freestyle-shell.png)

### **Freestyle with Maven task plugin**

If using the Maven plugin in Jenkins to declare and execute any specific goals in the [Maven build lifecycle](https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html), the configuration might look like:

[](https://docs.gitlab.com/ee/ci/migration/examples/img/maven-freestyle-plugin.png)

This plugin requires Maven to be installed on the Jenkins agent, and uses a script wrapper for calling Maven commands.

### **Using a declarative pipeline**

If using a declarative pipeline, the configuration might look like:

pipeline {

agent any

tools {

maven 'maven-3.6.3'

jdk 'jdk11'

}

stages {

stage('Build') {

steps {

sh "mvn package -DskipTests"

}

}

stage('Test') {

steps {

sh "mvn test"

}

}

stage('Install') {

steps {

sh "mvn install -DskipTests"

}

}

}}

This example uses shell execution commands instead of plugins.

By default, a declarative pipeline configuration is stored either in the Jenkins pipeline configuration or directly in the Git repository in a Jenksinfile.

## **Convert Jenkins configuration to GitLab CI/CD**

While the examples above are all slightly different, they can all be migrated to GitLab CI/CD with the same pipeline configuration.

Prerequisites:

* A GitLab Runner with a Shell executor
* Maven 3.6.3 and Java 11 JDK installed on the shell runner

This example mimics the behavior and syntax of building, testing, and installing on Jenkins.

In a GitLab CI/CD pipeline, the commands run in “jobs”, which are grouped into stages. The migrated configuration in the .gitlab-ci.yml configuration file consists of two global keywords (stages and variables) followed by 3 jobs:

stages:

- build

- test

- install

variables:

MAVEN\_OPTS: >-

-Dhttps.protocols=TLSv1.2

-Dmaven.repo.local=$CI\_PROJECT\_DIR/.m2/repository

MAVEN\_CLI\_OPTS: >-

-DskipTests

build-JAR:

stage: build

script:

- mvn $MAVEN\_CLI\_OPTS package

test-code:

stage: test

script:

- mvn test

install-JAR:

stage: install

script:

- mvn $MAVEN\_CLI\_OPTS install

In this example:

* stages defines three stages that run in order. Like the Jenkins examples above, the test job runs first, followed by the build job, and finally the install job.
* variables defines [CI/CD variables](https://docs.gitlab.com/ee/ci/variables/index.html) that can be used by all jobs:
  + MAVEN\_OPTS are Maven environment variables needed whenever Maven is executed:
    - -Dhttps.protocols=TLSv1.2 sets the TLS protocol to version 1.2 for any HTTP requests in the pipeline.
    - -Dmaven.repo.local=$CI\_PROJECT\_DIR/.m2/repository sets the location of the local Maven repository to the GitLab project directory on the runner, so the job can access and modify the repository.
  + MAVEN\_CLI\_OPTS are specific arguments to be added to mvn commands:
    - -DskipTests skips the test stage in the Maven build lifecycle.
* test-code, build-JAR, and install-JAR are the user-defined names for the jobs to run in the pipeline:
  + stage defines which stage the job runs in. A pipeline contains one or more stages and a stage contains one or more jobs. This example has three stages, each with a single job.
  + script defines the commands to run in that job, similar to steps in a Jenkinsfile. Jobs can run multiple commands in sequence, which run in the image container, but in this example the jobs run only one command each.

### **Run jobs in Docker containers**

Instead of using a persistent machine for handling this build process like the Jenkins samples, this example uses an ephemeral Docker container to handle execution. Using a container removes the need for maintaining a virtual machine and the Maven version installed on it. It also increases flexibility for expanding and extending the functionality of the pipeline.

Prerequisites:

* A GitLab Runner with the Docker executor that can be used by the project. If you are using GitLab.com, you can use the public shared runners.

This migrated pipeline configuration consists of three global keywords (stages, default, and variables) followed by 3 jobs. This configuration makes use of additional GitLab CI/CD features for an improved pipeline compared to the [example above](https://docs.gitlab.com/ee/ci/migration/examples/jenkins-maven.html" \l "convert-jenkins-configuration-to-gitlab-cicd):

stages:

- build

- test

- install

default:

image: maven:3.6.3-openjdk-11

cache:

key: $CI\_COMMIT\_REF\_SLUG

paths:

- .m2/

variables:

MAVEN\_OPTS: >-

-Dhttps.protocols=TLSv1.2

-Dmaven.repo.local=$CI\_PROJECT\_DIR/.m2/repository

MAVEN\_CLI\_OPTS: >-

-DskipTests

build-JAR:

stage: build

script:

- mvn $MAVEN\_CLI\_OPTS package

test-code:

stage: test

script:

- mvn test

install-JAR:

stage: install

script:

- mvn $MAVEN\_CLI\_OPTS install

In this example:

* stages defines three stages that run in order. Like the Jenkins examples above, the test job runs first, followed by the build job, and finally the install job.
* default defines standard configuration to reuse in all jobs by default:
  + image defines the Docker image container to use and execute commands in. In this example, it’s an official Maven Docker image with everything needed already installed.
  + cache is used to cache and reuse dependencies:
    - key is the unique identifier for the specific cache archive. In this example, it’s a shortened version of the Git commit ref, autogenerated as a [predefined CI/CD variable](https://docs.gitlab.com/ee/ci/variables/predefined_variables.html). Any job that runs for the same commit ref reuses the same cache.
    - paths are the directories or files to include in the cache. In this example, we cache the .m2/ directory to avoid re-installing dependencies between job runs.
* variables defines [CI/CD variables](https://docs.gitlab.com/ee/ci/variables/index.html) that can be used by all jobs:
  + MAVEN\_OPTS are Maven environment variables needed whenever Maven is executed:
    - -Dhttps.protocols=TLSv1.2 sets the TLS protocol to version 1.2 for any HTTP requests in the pipeline.
    - -Dmaven.repo.local=$CI\_PROJECT\_DIR/.m2/repository sets the location of the local Maven repository to the GitLab project directory on the runner, so the job can access and modify the repository.
  + MAVEN\_CLI\_OPTS are specific arguments to be added to mvn commands:
    - -DskipTests skips the test stage in the Maven build lifecycle.
* test-code, build-JAR, and install-JAR are the user-defined names for the jobs to run in the pipeline:
  + stage defines which stage the job runs in. A pipeline contains one or more stages and a stage contains one or more jobs. This example has three stages, each with a single job.
  + script defines the commands to run in that job, similar to steps in a Jenkinsfile. Jobs can run multiple commands in sequence, which run in the image container, but in this example the jobs run only one command each.

# **Migrating from CircleCI**All tiersAll offerings

If you are currently using CircleCI, you can migrate your CI/CD pipelines to [GitLab CI/CD](https://docs.gitlab.com/ee/ci/introduction/index.html), and start making use of all its powerful features.

We have collected several resources that you may find useful before starting to migrate.

The [Quick Start Guide](https://docs.gitlab.com/ee/ci/quick_start/index.html) is a good overview of how GitLab CI/CD works. You may also be interested in [Auto DevOps](https://docs.gitlab.com/ee/topics/autodevops/index.html) which can be used to build, test, and deploy your applications with little to no configuration needed at all.

For advanced CI/CD teams, [custom project templates](https://docs.gitlab.com/ee/administration/custom_project_templates.html) can enable the reuse of pipeline configurations.

If you have questions that are not answered here, the [GitLab community forum](https://forum.gitlab.com/) can be a great resource.

## **config.yml vs .gitlab-ci.yml**

CircleCI’s config.yml configuration file defines scripts, jobs, and workflows (known as “stages” in GitLab). In GitLab, a similar approach is used with a .gitlab-ci.yml file in the root directory of your repository.

### **Jobs**

In CircleCI, jobs are a collection of steps to perform a specific task. In GitLab, [jobs](https://docs.gitlab.com/ee/ci/jobs/index.html) are also a fundamental element in the configuration file. The checkout keyword is not necessary in GitLab CI/CD as the repository is automatically fetched.

CircleCI example job definition:

jobs:

job1:

steps:

- checkout

- run: "execute-script-for-job1"

Example of the same job definition in GitLab CI/CD:

job1:

script: "execute-script-for-job1"

### **Docker image definition**

CircleCI defines images at the job level, which is also supported by GitLab CI/CD. Additionally, GitLab CI/CD supports setting this globally to be used by all jobs that don’t have image defined.

CircleCI example image definition:

jobs:

job1:

docker:

- image: ruby:2.6

Example of the same image definition in GitLab CI/CD:

job1:

image: ruby:2.6

### **Workflows**

CircleCI determines the run order for jobs with workflows. This is also used to determine concurrent, sequential, scheduled, or manual runs. The equivalent function in GitLab CI/CD is called [stages](https://docs.gitlab.com/ee/ci/yaml/index.html" \l "stages). Jobs on the same stage run in parallel, and only run after previous stages complete. Execution of the next stage is skipped when a job fails by default, but this can be allowed to continue even [after a failed job](https://docs.gitlab.com/ee/ci/yaml/index.html" \l "allow_failure).

See [the Pipeline Architecture Overview](https://docs.gitlab.com/ee/ci/pipelines/pipeline_architectures.html) for guidance on different types of pipelines that you can use. Pipelines can be tailored to meet your needs, such as for a large complex project or a monorepo with independent defined components.

#### **Parallel and sequential job execution**

The following examples show how jobs can run in parallel, or sequentially:

1. job1 and job2 run in parallel (in the build stage for GitLab CI/CD).
2. job3 runs only after job1 and job2 complete successfully (in the test stage).
3. job4 runs only after job3 completes successfully (in the deploy stage).

CircleCI example with workflows:

version: **2**jobs:

job1:

steps:

- checkout

- run: make build dependencies

job2:

steps:

- run: make build artifacts

job3:

steps:

- run: make test

job4:

steps:

- run: make deploy

workflows:

version: **2**

jobs:

- job1

- job2

- job3:

requires:

- job1

- job2

- job4:

requires:

- job3

Example of the same workflow as stages in GitLab CI/CD:

stages:

- build

- test

- deploy

job1:

stage: build

script: make build dependencies

job2:

stage: build

script: make build artifacts

job3:

stage: test

script: make test

job4:

stage: deploy

script: make deploy

environment: production

#### **Scheduled run**

GitLab CI/CD has an easy to use UI to [schedule pipelines](https://docs.gitlab.com/ee/ci/pipelines/schedules.html). Also, [rules](https://docs.gitlab.com/ee/ci/yaml/index.html" \l "rules) can be used to determine if jobs should be included or excluded from a scheduled pipeline.

CircleCI example of a scheduled workflow:

commit-workflow:

jobs:

- buildscheduled-workflow:

triggers:

- schedule:

cron: "0 1 \* \* \*"

filters:

branches:

only: try-schedule-workflow

jobs:

- build

Example of the same scheduled pipeline using [rules](https://docs.gitlab.com/ee/ci/yaml/index.html" \l "rules) in GitLab CI/CD:

job1:

script:

- make build

rules:

- if: $CI\_PIPELINE\_SOURCE == "schedule" && $CI\_COMMIT\_REF\_NAME == "try-schedule-workflow"

After the pipeline configuration is saved, you configure the cron schedule in the [GitLab UI](https://docs.gitlab.com/ee/ci/pipelines/schedules.html" \l "add-a-pipeline-schedule), and can enable or disable schedules in the UI as well.

#### **Manual run**

CircleCI example of a manual workflow:

release-branch-workflow:

jobs:

- build

- testing:

requires:

- build

- deploy:

type: approval

requires:

- testing

Example of the same workflow using [when: manual](https://docs.gitlab.com/ee/ci/jobs/job_control.html" \l "create-a-job-that-must-be-run-manually) in GitLab CI/CD:

deploy\_prod:

stage: deploy

script:

- echo "Deploy to production server"

when: manual

environment: production

### **Filter job by branch**

[Rules](https://docs.gitlab.com/ee/ci/yaml/index.html" \l "rules) are a mechanism to determine if the job runs for a specific branch.

CircleCI example of a job filtered by branch:

jobs:

deploy:

branches:

only:

- main

- /rc-.\*/

Example of the same workflow using rules in GitLab CI/CD:

deploy:

stage: deploy

script:

- echo "Deploy job"

rules:

- if: $CI\_COMMIT\_BRANCH == "main" || $CI\_COMMIT\_BRANCH =~ /^rc-/

environment: production

### **Caching**

GitLab provides a caching mechanism to speed up build times for your jobs by reusing previously downloaded dependencies. It’s important to know the different between [cache and artifacts](https://docs.gitlab.com/ee/ci/caching/index.html" \l "how-cache-is-different-from-artifacts) to make the best use of these features.

CircleCI example of a job using a cache:

jobs:

job1:

steps:

- restore\_cache:

key: source-v1-< .Revision >

- checkout

- run: npm install

- save\_cache:

key: source-v1-< .Revision >

paths:

- "node\_modules"

Example of the same pipeline using cache in GitLab CI/CD:

test\_async:

image: node:latest

cache: *# Cache modules in between jobs*

key: $CI\_COMMIT\_REF\_SLUG

paths:

- .npm/

before\_script:

- npm ci --cache .npm --prefer-offline

script:

- node ./specs/start.js ./specs/async.spec.js

## **Contexts and variables**

CircleCI provides [Contexts](https://circleci.com/docs/contexts/) to securely pass environment variables across project pipelines. In GitLab, a [Group](https://docs.gitlab.com/ee/user/group/index.html) can be created to assemble related projects together. At the group level, [CI/CD variables](https://docs.gitlab.com/ee/ci/variables/index.html" \l "for-a-group) can be stored outside the individual projects, and securely passed into pipelines across multiple projects.

## **Orbs**

There are two GitLab issues open addressing CircleCI Orbs and how GitLab can achieve similar functionality.

* [issue #1151](https://gitlab.com/gitlab-com/Product/-/issues/1151)
* [issue #195173](https://gitlab.com/gitlab-org/gitlab/-/issues/195173)

## **Build environments**

CircleCI offers executors as the underlying technology to run a specific job. In GitLab, this is done by [runners](https://docs.gitlab.com/runner/).

The following environments are supported:

Self-managed runners:

* Linux
* Windows
* macOS

GitLab.com shared runners:

* Linux
* [Windows](https://docs.gitlab.com/ee/ci/runners/saas/windows_saas_runner.html) ([Beta](https://docs.gitlab.com/ee/policy/experiment-beta-support.html" \l "beta)).
* [macOS](https://docs.gitlab.com/ee/ci/runners/saas/macos_saas_runner.html) ([Beta](https://docs.gitlab.com/ee/policy/experiment-beta-support.html" \l "beta)).

### **Machine and specific build environments**

[Tags](https://docs.gitlab.com/ee/ci/yaml/index.html" \l "tags) can be used to run jobs on different platforms, by telling GitLab which runners should run the jobs.

CircleCI example of a job running on a specific environment:

jobs:

ubuntuJob:

machine:

image: ubuntu-1604:201903-01

steps:

- checkout

- run: echo "Hello, $USER!"

osxJob:

macos:

xcode: 11.3.0

steps:

- checkout

- run: echo "Hello, $USER!"

Example of the same job using tags in GitLab CI/CD:

windows job:

stage: build

tags:

- windows

script:

- echo Hello, %USERNAME%!

osx job:

stage: build

tags:

- osx

script:

- echo "Hello, $USER!"

# **GitLab CI/CD examples**All tiersAll offerings

This page contains links to a variety of examples that can help you understand how to implement [GitLab CI/CD](https://docs.gitlab.com/ee/ci/index.html) for your specific use case.

Examples are available in several forms. As a collection of:

* .gitlab-ci.yml [template files](https://docs.gitlab.com/ee/ci/examples/" \l "cicd-templates) maintained in GitLab, for many common frameworks and programming languages.
* Repositories with [example projects](https://gitlab.com/gitlab-examples) for various languages. You can fork and adjust them to your own needs. Projects include an example of using [Review Apps with a static site served by NGINX](https://gitlab.com/gitlab-examples/review-apps-nginx/).
* Examples and [other resources](https://docs.gitlab.com/ee/ci/examples/" \l "other-resources) listed below.

## **CI/CD examples**

The following table lists examples with step-by-step tutorials that are contained in this section:

| **Use case** | **Resource** |
| --- | --- |
| Deployment with Dpl | [Using dpl as deployment tool](https://docs.gitlab.com/ee/ci/examples/deployment/index.html). |
| GitLab Pages | See the [GitLab Pages](https://docs.gitlab.com/ee/user/project/pages/index.html) documentation for a complete example of deploying a static site. |
| End-to-end testing | [End-to-end testing with GitLab CI/CD and WebdriverIO](https://docs.gitlab.com/ee/ci/examples/end_to_end_testing_webdriverio/index.html). |
| Multi project pipeline | [Build, test deploy using multi project pipeline](https://gitlab.com/gitlab-examples/upstream-project). |
| npm with semantic-release | [Publish npm packages to the GitLab Package Registry using semantic-release](https://docs.gitlab.com/ee/ci/examples/semantic-release.html). |
| PHP with Laravel, Envoy | [Test and deploy Laravel applications with GitLab CI/CD and Envoy](https://docs.gitlab.com/ee/ci/examples/laravel_with_gitlab_and_envoy/index.html). |
| PHP with npm, SCP | [Running Composer and npm scripts with deployment via SCP in GitLab CI/CD](https://docs.gitlab.com/ee/ci/examples/deployment/composer-npm-deploy.html). |
| PHP with PHPUnit, atoum | [Testing PHP projects](https://docs.gitlab.com/ee/ci/examples/php.html). |
| Secrets management with Vault | [Authenticating and Reading Secrets With HashiCorp Vault](https://docs.gitlab.com/ee/ci/examples/authenticating-with-hashicorp-vault/index.html). |

### **Contributed examples**

You can help people that use your favorite programming language by submitting a link to a guide for that language. These contributed guides are hosted externally or in separate example projects:

| **Use case** | **Resource** |
| --- | --- |
| Clojure | [Test a Clojure application with GitLab CI/CD](https://gitlab.com/gitlab-examples/clojure-web-application). |
| Game development | [DevOps and Game Development with GitLab CI/CD](https://gitlab.com/gitlab-examples/gitlab-game-demo/). |
| Java with Maven | [How to deploy Maven projects to Artifactory with GitLab CI/CD](https://gitlab.com/gitlab-examples/maven/simple-maven-example). |
| Java with Spring Boot | [Deploy a Spring Boot application to Cloud Foundry with GitLab CI/CD](https://gitlab.com/gitlab-examples/spring-gitlab-cf-deploy-demo). |
| Parallel testing Ruby & JS | [GitLab CI/CD parallel jobs testing for Ruby & JavaScript projects](https://docs.knapsackpro.com/2019/how-to-run-parallel-jobs-for-rspec-tests-on-gitlab-ci-pipeline-and-speed-up-ruby-javascript-testing). |
| Python on Heroku | [Test and deploy a Python application with GitLab CI/CD](https://gitlab.com/gitlab-examples/python-getting-started). |
| Ruby on Heroku | [Test and deploy a Ruby application with GitLab CI/CD](https://gitlab.com/gitlab-examples/ruby-getting-started). |
| Scala on Heroku | [Test and deploy a Scala application to Heroku](https://gitlab.com/gitlab-examples/scala-sbt). |

## **CI/CD templates**

Get started with GitLab CI/CD and your favorite programming language or framework by using a .gitlab-ci.yml [template](https://gitlab.com/gitlab-org/gitlab/-/tree/master/lib/gitlab/ci/templates).

When you create a .gitlab-ci.yml file in the UI, you can choose one of these templates:

* [Android (Android.gitlab-ci.yml)](https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Android.gitlab-ci.yml)
* [Android with fastlane (Android-Fastlane.gitlab-ci.yml)](https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Android-Fastlane.gitlab-ci.yml)
* [Bash (Bash.gitlab-ci.yml)](https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Bash.gitlab-ci.yml)
* [C++ (C++.gitlab-ci.yml)](https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/C++.gitlab-ci.yml)
* [Chef (Chef.gitlab-ci.yml)](https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Chef.gitlab-ci.yml)
* [Clojure (Clojure.gitlab-ci.yml)](https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Clojure.gitlab-ci.yml)
* [Composer Composer.gitlab-ci.yml](https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Composer.gitlab-ci.yml)
* [Crystal (Crystal.gitlab-ci.yml)](https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Crystal.gitlab-ci.yml)
* [Dart (Dart.gitlab-ci.yml)](https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Dart.gitlab-ci.yml)
* [Django (Django.gitlab-ci.yml)](https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Django.gitlab-ci.yml)
* [Docker (Docker.gitlab-ci.yml)](https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Docker.gitlab-ci.yml)
* [dotNET (dotNET.gitlab-ci.yml)](https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/dotNET.gitlab-ci.yml)
* [dotNET Core (dotNET-Core.gitlab-ci.yml)](https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/dotNET-Core.gitlab-ci.yml)
* [Elixir (Elixir.gitlab-ci.yml)](https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Elixir.gitlab-ci.yml)
* [Flutter (Flutter.gitlab-ci.yml)](https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Flutter.gitlab-ci.yml)
* [Go (Go.gitlab-ci.yml)](https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Go.gitlab-ci.yml)
* [Gradle (Gradle.gitlab-ci.yml)](https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Gradle.gitlab-ci.yml)
* [Grails (Grails.gitlab-ci.yml)](https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Grails.gitlab-ci.yml)
* [iOS with fastlane (iOS-Fastlane.gitlab-ci.yml)](https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/iOS-Fastlane.gitlab-ci.yml)
* [Julia (Julia.gitlab-ci.yml)](https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Julia.gitlab-ci.yml)
* [Laravel (Laravel.gitlab-ci.yml)](https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Laravel.gitlab-ci.yml)
* [LaTeX (LaTeX.gitlab-ci.yml)](https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/LaTeX.gitlab-ci.yml)
* [Maven (Maven.gitlab-ci.yml)](https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Maven.gitlab-ci.yml)
* [Mono (Mono.gitlab-ci.yml)](https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Mono.gitlab-ci.yml)
* [npm (npm.gitlab-ci.yml)](https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/npm.gitlab-ci.yml)
* [Node.js (Nodejs.gitlab-ci.yml)](https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Nodejs.gitlab-ci.yml)
* [OpenShift (OpenShift.gitlab-ci.yml)](https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/OpenShift.gitlab-ci.yml)
* [Packer (Packer.gitlab-ci.yml)](https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Packer.gitlab-ci.yml)
* [PHP (PHP.gitlab-ci.yml)](https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/PHP.gitlab-ci.yml)
* [Python (Python.gitlab-ci.yml)](https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Python.gitlab-ci.yml)
* [Ruby (Ruby.gitlab-ci.yml)](https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Ruby.gitlab-ci.yml)
* [Rust (Rust.gitlab-ci.yml)](https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Rust.gitlab-ci.yml)
* [Scala (Scala.gitlab-ci.yml)](https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Scala.gitlab-ci.yml)
* [Swift (Swift.gitlab-ci.yml)](https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Swift.gitlab-ci.yml)
* [Terraform (Terraform.gitlab-ci.yml)](https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Terraform.gitlab-ci.yml)
* [Terraform (Terraform.latest.gitlab-ci.yml)](https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Terraform.latest.gitlab-ci.yml)

If a programming language or framework template is not in this list, you can contribute one. To create a template, submit a merge request to [the templates list](https://gitlab.com/gitlab-org/gitlab/-/tree/master/lib/gitlab/ci/templates).

### **Adding templates to your GitLab installation**[Premium](https://about.gitlab.com/pricing/?glm_source=docs.gitlab.com&glm_content=badges-docs)Self-managed

You can add custom examples and templates to your self-managed GitLab instance. Your GitLab administrator can [designate an instance template repository](https://docs.gitlab.com/ee/administration/settings/instance_template_repository.html) that contains examples and templates specific to your organization.

## **Other resources**

This section provides further resources to help you get familiar with various uses of GitLab CI/CD. Older articles and videos may not reflect the state of the latest GitLab release.

### **CI/CD in the cloud**

For examples of setting up GitLab CI/CD for cloud-based environments, see:

* [How to set up multi-account AWS SAM deployments with GitLab CI](https://about.gitlab.com/blog/2019/02/04/multi-account-aws-sam-deployments-with-gitlab-ci/)
* Video: [Automating Kubernetes Deployments with GitLab CI/CD](https://www.youtube.com/watch?v=wEDRfAz6_Uw)
* [How to autoscale continuous deployment with GitLab Runner on DigitalOcean](https://about.gitlab.com/blog/2018/06/19/autoscale-continuous-deployment-gitlab-runner-digital-ocean/)
* [How to create a CI/CD pipeline with Auto Deploy to Kubernetes using GitLab and Helm](https://about.gitlab.com/blog/2017/09/21/how-to-create-ci-cd-pipeline-with-autodeploy-to-kubernetes-using-gitlab-and-helm/)
* Video: [Demo - Deploying from GitLab to OpenShift Container Cluster](https://youtu.be/EwbhA53Jpp4)
* Tutorial: [Set up a GitLab.com Civo Kubernetes integration with GitPod](https://gitlab.com/k33g_org/k33g_org.gitlab.io/-/issues/82)

See also the following video overviews:

* Video: [Kubernetes, GitLab, and Cloud Native](https://www.youtube.com/watch?v=d-9awBxEbvQ)
* Video: [Deploying to IBM Cloud with GitLab CI/CD](https://www.youtube.com/watch?v=6ZF4vgKMd-g)

### **Customer stories**

For some customer experiences with GitLab CI/CD, see:

* [How Verizon Connect reduced data center deploys from 30 days to under 8 hours with GitLab](https://about.gitlab.com/blog/2019/02/14/verizon-customer-story/)
* [How Wag! cut their release process from 40 minutes to just 6](https://about.gitlab.com/blog/2019/01/16/wag-labs-blog-post/)
* [How Jaguar Land Rover embraced CI to speed up their software lifecycle](https://about.gitlab.com/blog/2018/07/23/chris-hill-devops-enterprise-summit-talk/)

### **Getting started**

For some examples to help get you started, see:

* [GitLab CI/CD’s 2018 highlights](https://about.gitlab.com/blog/2019/01/21/gitlab-ci-cd-features-improvements/)
* [A beginner’s guide to continuous integration](https://about.gitlab.com/blog/2018/01/22/a-beginners-guide-to-continuous-integration/)

### **Implementing GitLab CI/CD**

For examples of others who have implemented GitLab CI/CD, see:

* [How to streamline interactions between multiple repositories with multi-project pipelines](https://about.gitlab.com/blog/2018/10/31/use-multiproject-pipelines-with-gitlab-cicd/)
* [How we used GitLab CI to build GitLab faster](https://about.gitlab.com/blog/2018/05/02/using-gitlab-ci-to-build-gitlab-faster/)
* [Test all the things in GitLab CI with Docker by example](https://about.gitlab.com/blog/2018/02/05/test-all-the-things-gitlab-ci-docker-examples/)
* [A Craftsman looks at continuous integration](https://about.gitlab.com/blog/2018/01/17/craftsman-looks-at-continuous-integration/)
* [Go tools and GitLab: How to do continuous integration like a boss](https://about.gitlab.com/blog/2017/11/27/go-tools-and-gitlab-how-to-do-continuous-integration-like-a-boss/)
* [GitBot – automating boring Git operations with CI](https://about.gitlab.com/blog/2017/11/02/automating-boring-git-operations-gitlab-ci/)
* [How to use GitLab CI for Vue.js](https://about.gitlab.com/blog/2017/09/12/vuejs-app-gitlab/)
* Video: [GitLab CI/CD Deep Dive](https://youtu.be/pBe4t1CD8Fc?t=195)
* [Dockerizing GitLab Review Apps](https://about.gitlab.com/blog/2017/07/11/dockerizing-review-apps/)
* [Fast and natural continuous integration with GitLab CI](https://about.gitlab.com/blog/2017/05/22/fast-and-natural-continuous-integration-with-gitlab-ci/)
* [Demo: CI/CD with GitLab in action](https://about.gitlab.com/blog/2017/03/13/ci-cd-demo/)

### **Migrating to GitLab from third-party CI tools**

* [Migrating from CircleCI to GitLab](https://docs.gitlab.com/ee/ci/migration/circleci.html)
* [Migrating from Jenkins to GitLab](https://docs.gitlab.com/ee/ci/migration/jenkins.html)
* Video: [Migrating from Jenkins to GitLab](https://youtu.be/RlEVGOpYF5Y)

### **Integrating GitLab CI/CD with other systems**

To see how you can integrate GitLab CI/CD with third-party systems, see:

* [Streamline and shorten error remediation with Sentry’s new GitLab integration](https://about.gitlab.com/blog/2019/01/25/sentry-integration-blog-post/)
* [How to simplify your smart home configuration with GitLab CI/CD](https://about.gitlab.com/blog/2018/08/02/using-the-gitlab-ci-slash-cd-for-smart-home-configuration-management/)
* [Demo: GitLab + Jira + Jenkins](https://about.gitlab.com/blog/2018/07/30/gitlab-workflow-with-jira-jenkins/)
* [Introducing Auto Breakfast from GitLab (sort of)](https://about.gitlab.com/blog/2018/06/29/introducing-auto-breakfast-from-gitlab/)

### **Mobile development**

For help with using GitLab CI/CD for mobile application development, see:

* [How to publish Android apps to the Google Play Store with GitLab and fastlane](https://about.gitlab.com/blog/2019/01/28/android-publishing-with-gitlab-and-fastlane/)
* [Setting up GitLab CI for Android projects](https://about.gitlab.com/blog/2018/10/24/setting-up-gitlab-ci-for-android-projects/)
* [Working with YAML in GitLab CI from the Android perspective](https://about.gitlab.com/blog/2017/11/20/working-with-yaml-gitlab-ci-android/)
* [How to use GitLab CI and MacStadium to build your macOS or iOS projects](https://about.gitlab.com/blog/2017/05/15/how-to-use-macstadium-and-gitlab-ci-to-build-your-macos-or-ios-projects/)
* [Setting up GitLab CI for iOS projects](https://about.gitlab.com/blog/2016/03/10/setting-up-gitlab-ci-for-ios-projects/)