

# Migrating from GitLab CI/CD to GitHub Actions

## In this article

Introduction

Jobs

Runners

Docker images

Condition and expression syntax

Dependencies between Jobs

Scheduling workflows

Variables and secrets

Caching

Artifacts

Databases and service containers

GitHub Actions and GitLab CI/CD share several configuration similarities, which makes migrating to GitHub Actions relatively straightforward.

**Note:** GitHub-hosted runners are not currently supported on GitHub Enterprise Server. You can see more information about planned future support on the [GitHub public roadmap](#).

## Introduction

GitLab CI/CD and GitHub Actions both allow you to create workflows that automatically build, test, publish, release, and deploy code. GitLab CI/CD and GitHub Actions share some similarities in workflow configuration:

- Workflow configuration files are written in YAML and are stored in the code's repository.
- Workflows include one or more jobs.
- Jobs include one or more steps or individual commands.
- Jobs can run on either managed or self-hosted machines.

There are a few differences, and this guide will show you the important differences so that you can migrate your workflow to GitHub Actions.

## Jobs

Jobs in GitLab CI/CD are very similar to jobs in GitHub Actions. In both systems, jobs have the following characteristics:

- Jobs contain a series of steps or scripts that run sequentially.
- Jobs can run on separate machines or in separate containers.

- Jobs run in parallel by default, but can be configured to run sequentially.

You can run a script or a shell command in a job. In GitLab CI/CD, script steps are specified using the `script` key. In GitHub Actions, all scripts are specified using the `run` key.

Below is an example of the syntax for each system.

## GitLab CI/CD syntax for jobs [↗](#)

```
job1:
  variables:
    GIT_CHECKOUT: "true"
  script:
    - echo "Run your script here"
```

## GitHub Actions syntax for jobs [↗](#)

```
jobs:
  job1:
    steps:
      - uses: actions/checkout@v4
      - run: echo "Run your script here"
```

## Runners [↗](#)

Runners are machines on which the jobs run. Both GitLab CI/CD and GitHub Actions offer managed and self-hosted variants of runners. In GitLab CI/CD, `tags` are used to run jobs on different platforms, while in GitHub Actions it is done with the `runs-on` key.

Below is an example of the syntax for each system.

## GitLab CI/CD syntax for runners [↗](#)

```
windows_job:
  tags:
    - windows
  script:
    - echo Hello, %USERNAME%!

linux_job:
  tags:
    - linux
  script:
    - echo "Hello, $USER!"
```

## GitHub Actions syntax for runners [↗](#)

```
windows_job:
  runs-on: windows-latest
  steps:
    - run: echo Hello, %USERNAME%!

linux_job:
  runs-on: ubuntu-latest
  steps:
    - run: echo "Hello, $USER!"
```

For more information, see "[Workflow syntax for GitHub Actions](#)."

## Docker images

---

Both GitLab CI/CD and GitHub Actions support running jobs in a Docker image. In GitLab CI/CD, Docker images are defined with an `image` key, while in GitHub Actions it is done with the `container` key.

Below is an example of the syntax for each system.

### GitLab CI/CD syntax for Docker images

```
my_job:
  image: node:10.16-jessie
```

### GitHub Actions syntax for Docker images

```
jobs:
  my_job:
    container: node:10.16-jessie
```

For more information, see "[Workflow syntax for GitHub Actions](#)."

## Condition and expression syntax

---

GitLab CI/CD uses `rules` to determine if a job will run for a specific condition. GitHub Actions uses the `if` keyword to prevent a job from running unless a condition is met.

Below is an example of the syntax for each system.

### GitLab CI/CD syntax for conditions and expressions

```
deploy_prod:
  stage: deploy
  script:
    - echo "Deploy to production server"
  rules:
    - if: '$CI_COMMIT_BRANCH == "master"'
```

### GitHub Actions syntax for conditions and expressions

```
jobs:
  deploy_prod:
    if: contains( github.ref, 'master')
    runs-on: ubuntu-latest
    steps:
      - run: echo "Deploy to production server"
```

For more information, see "[Expressions](#)."

## Dependencies between jobs

---

Both GitLab CI/CD and GitHub Actions allow you to set dependencies for a job. In both

systems, jobs run in parallel by default, but job dependencies in GitHub Actions can be specified explicitly with the `needs` key. GitLab CI/CD also has a concept of `stages`, where jobs in a stage run concurrently, but the next stage will start when all the jobs in the previous stage have completed. You can recreate this scenario in GitHub Actions with the `needs` key.

Below is an example of the syntax for each system. The workflows start with two jobs named `build_a` and `build_b` running in parallel, and when those jobs complete, another job called `test_ab` will run. Finally, when `test_ab` completes, the `deploy_ab` job will run.

## GitLab CI/CD syntax for dependencies between jobs [↗](#)

```
stages:
  - build
  - test
  - deploy

build_a:
  stage: build
  script:
    - echo "This job will run first."

build_b:
  stage: build
  script:
    - echo "This job will run first, in parallel with build_a."

test_ab:
  stage: test
  script:
    - echo "This job will run after build_a and build_b have finished."

deploy_ab:
  stage: deploy
  script:
    - echo "This job will run after test_ab is complete"
```

## GitHub Actions syntax for dependencies between jobs [↗](#)

```
jobs:
  build_a:
    runs-on: ubuntu-latest
    steps:
      - run: echo "This job will be run first."

  build_b:
    runs-on: ubuntu-latest
    steps:
      - run: echo "This job will be run first, in parallel with build_a"

  test_ab:
    runs-on: ubuntu-latest
    needs: [build_a,build_b]
    steps:
      - run: echo "This job will run after build_a and build_b have finished"

  deploy_ab:
    runs-on: ubuntu-latest
    needs: [test_ab]
    steps:
      - run: echo "This job will run after test_ab is complete"
```

For more information, see "[Workflow syntax for GitHub Actions](#)."

## Scheduling workflows

---

Both GitLab CI/CD and GitHub Actions allow you to run workflows at a specific interval. In GitLab CI/CD, pipeline schedules are configured with the UI, while in GitHub Actions you can trigger a workflow on a scheduled interval with the "on" key.

For more information, see "[Events that trigger workflows](#)."

## Variables and secrets

---

GitLab CI/CD and GitHub Actions support setting variables in the pipeline or workflow configuration file, and creating secrets using the GitLab or GitHub Enterprise Server UI.

For more information, see "[Variables](#)" and "[Using secrets in GitHub Actions](#)."

## Caching

---

GitLab CI/CD and GitHub Actions provide a method in the configuration file to manually cache workflow files.

Below is an example of the syntax for each system.

### GitLab CI/CD syntax for caching

```
image: node:latest

cache:
  key: $CI_COMMIT_REF_SLUG
  paths:
    - .npm/

before_script:
  - npm ci --cache .npm --prefer-offline

test_async:
  script:
    - node ./specs/start.js ./specs/async.spec.js
```

### GitHub Actions syntax for caching

```
jobs:
  test_async:
    runs-on: ubuntu-latest
    steps:
      - name: Cache node modules
        uses: actions/cache@v3
        with:
          path: ~/.npm
          key: v1-npm-deps-${{ hashFiles('**/package-lock.json') }}
          restore-keys: v1-npm-deps-
```

## Artifacts

---

Both GitLab CI/CD and GitHub Actions can upload files and directories created by a job as artifacts. In GitHub Actions, artifacts can be used to persist data across multiple jobs.

Below is an example of the syntax for each system.

## GitLab CI/CD syntax for artifacts [↗](#)

```
script:
artifacts:
  paths:
    - math-homework.txt
```

## GitHub Actions syntax for artifacts [↗](#)

```
- name: Upload math result for job 1
  uses: actions/upload-artifact@v3
  with:
    name: homework
    path: math-homework.txt
```

For more information, see "[Storing workflow data as artifacts](#)."

## Databases and service containers [↗](#)

Both systems enable you to include additional containers for databases, caching, or other dependencies.

In GitLab CI/CD, a container for the job is specified with the `image` key, while GitHub Actions uses the `container` key. In both systems, additional service containers are specified with the `services` key.

Below is an example of the syntax for each system.

## GitLab CI/CD syntax for databases and service containers [↗](#)

```
container-job:
  variables:
    POSTGRES_PASSWORD: postgres
    # The hostname used to communicate with the
    # PostgreSQL service container
    POSTGRES_HOST: postgres
    # The default PostgreSQL port
    POSTGRES_PORT: 5432
  image: node:10.18-jessie
  services:
    - postgres
  script:
    # Performs a clean installation of all dependencies
    # in the `package.json` file
    - npm ci
    # Runs a script that creates a PostgreSQL client,
    # populates the client with data, and retrieves data
    - node client.js
  tags:
    - docker
```

## GitHub Actions syntax for databases and service containers [↗](#)

```
jobs:
  container-job:
    runs-on: ubuntu-latest
    container: node:10.18-jessie
```

```
services:
  postgres:
    image: postgres
    env:
      POSTGRES_PASSWORD: postgres

steps:
  - name: Check out repository code
    uses: actions/checkout@v4

  # Performs a clean installation of all dependencies
  # in the `package.json` file
  - name: Install dependencies
    run: npm ci

  - name: Connect to PostgreSQL
    # Runs a script that creates a PostgreSQL client,
    # populates the client with data, and retrieves data
    run: node client.js
    env:
      # The hostname used to communicate with the
      # PostgreSQL service container
      POSTGRES_HOST: postgres
      # The default PostgreSQL port
      POSTGRES_PORT: 5432
```

For more information, see "[About service containers](#)."

## Legal

© 2023 GitHub, Inc. [Terms](#) [Privacy](#) [Status](#) [Pricing](#) [Expert services](#) [Blog](#)