.gitlab-ci.yml keyword reference

Keywords

A GitLab CI/CD pipeline configuration includes:

- Global keywords that configure pipeline behavior:

  KeywordDescription `default`Custom default values for job keywords. `include`Import configuration from other YAML files. `stages`The names and order of the pipeline stages. `variables`Define CI/CD variables for all job in the pipeline. `workflow`Control what types of pipeline run.
- Jobs configured with job keywords:

  KeywordDescription
- `after_script`Override a set of commands that are executed after job.
- `allow_failure`Allow job to fail. A failed job does not cause the pipeline to fail.
- `artifacts`List of files and directories to attach to a job on success.
- `before_script`Override a set of commands that are executed before job.
- `cache`List of files that should be cached between subsequent runs.
- `coverage`Code coverage settings for a given job.
- `dast_configuration`Use configuration from DAST profiles on a job level.
- `dependencies`Restrict which artifacts are passed to a specific job by providing a list of jobs to fetch artifacts from.
- `environment`Name of an environment to which the job deploys.
- `except`Control when jobs are not created.
- `extends`Configuration entries that this job inherits from.
- `image`Use Docker images.
- `inherit`Select which global defaults all jobs inherit.
- `interruptible`Defines if a job can be canceled when made redundant by a newer run. `needs`Execute jobs earlier than the stage ordering.
- `only`Control when jobs are created.
- `pages`Upload the result of a job to use with GitLab Pages.
- `parallel`How many instances of a job should be run in parallel.
- `release`Instructs the runner to generate a release object.
- `resource_group`Limit job concurrency.
- `retry`When and how many times a job can be auto-retried in case of a failure.
- `rules`List of conditions to evaluate and determine selected attributes of a job, and whether or not it's created.
- `script`Shell script that is executed by a runner.
- `secrets`The CI/CD secrets the job needs.
- `services`Use Docker services images.
- `stage`Defines a job stage.
- `tags`List of tags that are used to select a runner.
- `timeout`Define a custom job-level timeout that takes precedence over the project-wide setting. `trigger`Defines a downstream pipeline trigger.
- `variables`Define job variables on a job level.
- `when`When to run job.

Global keywords

Some keywords are not defined in a job. These keywords control pipeline behavior or import additional pipeline configuration.

`default`

You can set global defaults for some keywords. Jobs that do not define one or more of the listed keywords use the value defined in the `default` section.

Keyword type: Global keyword.

Possible inputs: These keywords can have custom defaults:

- `after_script`
- `artifacts`
- `before_script`
- `cache`
- `image`
- `interruptible`
- `retry`
- `services`
- `tags`

- `timeout`

Example of `default`:

```
default:
  image: ruby:3.0

rspec:
  script: bundle exec rspec

rspec 2.7:
  image: ruby:2.7
  script: bundle exec rspec
```

In this example, `ruby:3.0` is the default `image` value for all jobs in the pipeline. The `rspec 2.7` job does not use the default, because it overrides the default with a job-specific `image` section:

Additional details:

- When the pipeline is created, each default is copied to all jobs that don't have that keyword defined.
- If a job already has one of the keywords configured, the configuration in the job takes precedence and is not replaced by the default.
- Control inheritance of default keywords in jobs with `inherit:default`.

`include`

Moved to GitLab Free in 11.4.

Use `include` to include external YAML files in your CI/CD configuration. You can split one long `.gitlab-ci.yml` file into multiple files to increase readability, or reduce duplication of the same configuration in multiple places.

You can also store template files in a central repository and include them in projects.

The `include` files are:

- Merged with those in the `.gitlab-ci.yml` file.
- Always evaluated first and then merged with the content of the `.gitlab-ci.yml` file, regardless of the position of the `include` keyword.

You can nest up to 100 includes. In GitLab 14.9 and later, the same file can be included multiple times in nested includes, but duplicates are ignored.

In GitLab 12.4 and later, the time limit to resolve all files is 30 seconds.

Keyword type: Global keyword.

Possible inputs: The `include` subkeys:

- `include:local`
- `include:file`
- `include:remote`
- `include:template`

Additional details:

- Use merging to customize and override included CI/CD configurations with local
- You can override included configuration by having the same job name or global keyword in the `.gitlab-ci.yml` file. The two configurations are merged together, and the configuration in the `.gitlab-ci.yml` file takes precedence over the included configuration.

Related topics:

- Use variables with `include`.
- Use `rules` with `include`.

`include:local`

Use `include:local` to include a file that is in the same repository as the `.gitlab-ci.yml` file. Use `include:local` instead of symbolic links.

Keyword type: Global keyword.

Possible inputs:

A full path relative to the root directory (`/`):

- The YAML file must have the extension `.yml` or `.yaml`.
- You can use `*` and `**` wildcards in the file path.
- You can use certain CI/CD variables.

Example of `include:local`:

```
include:
  - local: '/templates/.gitlab-ci-template.yml'
```

You can also use shorter syntax to define the path:

```
include: '.gitlab-ci-production.yml'
```

Additional details:

- The `.gitlab-ci.yml` file and the local file must be on the same branch.
- You can't include local files through Git submodules paths.
- All nested includes are executed in the scope of the same project, so you can use local, project, remote, or template includes.

`include:file`

Including multiple files from the same project introduced in GitLab 13.6. Feature flag removed in GitLab 13.8.

To include files from another private project on the same GitLab instance, use `include:file`. You can use `include:file` in combination with `include:project` only.

Keyword type: Global keyword.

Possible inputs:

A full path, relative to the root directory (`/`):

- The YAML file must have the extension `.yml` or `.yaml`.
- You can use certain CI/CD variables.

Example of `include:file`:

```
include:
  - project: 'my-group/my-project'
    file: '/templates/.gitlab-ci-template.yml'
```

You can also specify a `ref`. If you do not specify a value, the ref defaults to the `HEAD` of the project:

```
include:
  - project: 'my-group/my-project'
    ref: main
    file: '/templates/.gitlab-ci-template.yml'

  - project: 'my-group/my-project'
    ref: v1.0.0  # Git Tag
    file: '/templates/.gitlab-ci-template.yml'

  - project: 'my-group/my-project'
    ref: 787123b47f14b552955ca2786bc9542ae66fee5b  # Git SHA
    file: '/templates/.gitlab-ci-template.yml'
```

You can include multiple files from the same project:

```
include:
  - project: 'my-group/my-project'
    ref: main
    file:
      - '/templates/.builds.yml'
      - '/templates/.tests.yml'
```

Additional details:

- All nested includes are executed in the scope of the target project. You can use `local` (relative to the target project), `project`, `remote`, or `template` includes.
- When the pipeline starts, the `.gitlab-ci.yml` file configuration included by all methods is evaluated. The configuration is a snapshot in time and persists in the database. GitLab does not reflect any changes to the referenced `.gitlab-ci.yml` file configuration until the next pipeline starts.
- When you include a YAML file from another private project, the user running the pipeline must be a member of both projects and have the appropriate permissions to run pipelines. A `not found or access denied` error may be displayed if the user does not have access to any of the included files.

`include:remote`

Use `include:remote` with a full URL to include a file from a different location.

Keyword type: Global keyword.

Possible inputs:

A public URL accessible by an HTTP/HTTPS `GET` request:

- Authentication with the remote URL is not supported.
- The YAML file must have the extension `.yml` or `.yaml`.
- You can use certain CI/CD variables.

Example of `include:remote`:

```
include:
  - remote: 'https://gitlab.com/example-project/-/raw/main/.gitlab-ci.
yml'
```

Additional details:

- All nested includes execute without context as a public user, so you can only include public projects or templates.
- Be careful when including a remote CI/CD configuration file. No pipelines or notifications trigger when external CI/CD configuration files change. From a security perspective, this is similar to pulling a third-party dependency.

`include:template`

Use `include:template` to include `.gitlab-ci.yml` templates.

Keyword type: Global keyword.

Possible inputs:

A CI/CD template:

- Templates are stored in `lib/gitlab/ci/templates`. Not all templates are designed to be used with `include:template`, so check template comments before using one.
- You can use certain CI/CD variables.

Example of `include:template`:

```
# File sourced from the GitLab template collection
include:
  - template: Auto-DevOps.gitlab-ci.yml
```

Multiple `include:template` files:

```
include:
  - template: Android-Fastlane.gitlab-ci.yml
  - template: Auto-DevOps.gitlab-ci.yml
```

Additional details:

- All nested includes are executed only with the permission of the user, so it's possible to use `project`, `remote`, or `template` includes.

`stages`

Use `stages` to define stages that contain groups of jobs. Use `stage` in a job to configure the job to run in a specific stage.

If `stages` is not defined in the `.gitlab-ci.yml` file, the default pipeline stages are:

- `.pre`
- `build`
- `test`
- `deploy`
- `.post`

The order of the items in `stages` defines the execution order for jobs:

- Jobs in the same stage run in parallel.

- Jobs in the next stage run after the jobs from the previous stage complete successfully.

If a pipeline contains only jobs in the `.pre` or `.post` stages, it does not run. There must be at least one other job in a different stage. `.pre` and `.post` stages can be used in required pipeline configuration to define compliance jobs that must run before or after project pipeline jobs.

Keyword type: Global keyword.

Example of `stages`:

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

In this example:

1. All jobs in `build` execute in parallel.
2. If all jobs in `build` succeed, the `test` jobs execute in parallel.
3. If all jobs in `test` succeed, the `deploy` jobs execute in parallel.
4. If all jobs in `deploy` succeed, the pipeline is marked as `passed`.

If any job fails, the pipeline is marked as `failed` and jobs in later stages do not start. Jobs in the current stage are not stopped and continue to run.

Additional details:

- If a job does not specify a `stage`, the job is assigned the `test` stage.
- If a stage is defined but no jobs use it, the stage is not visible in the pipeline, which can help compliance pipeline configurations:
  - Stages can be defined in the compliance configuration but remain hidden if not used.
  - The defined stages become visible when developers use them in job definitions.

Related topics:

- To make a job start earlier and ignore the stage order, use the `needs` keyword.

`workflow`

Introduced in GitLab 12.5

Use `workflow` to control pipeline behavior.

Related topics:

- `workflow: rules` examples
- Switch between branch pipelines and merge request pipelines

`workflow:rules`

The `rules` keyword in `workflow` is similar to `rules` defined in jobs, but controls whether or not a whole pipeline is created.

When no rules evaluate to true, the pipeline does not run.

Possible inputs: You can use some of the same keywords as job-level `rules`:

- `rules: if`.
- `rules: changes`.
- `rules: exists`.
- when, can only be `always` or `never` when used with `workflow`.
- `variables`.

Example of `workflow:rules`:

```
workflow:
  rules:
    - if: $CI_COMMIT_TITLE =~ /-draft$/
      when: never
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
    - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH
```

In this example, pipelines run if the commit title (first line of the commit message) does not end with `-draft` and the pipeline is for either:

- A merge request
- The default branch.

Additional details:

- If your rules match both branch pipelines (other than the default branch) and merge request pipelines, duplicate pipelines can occur.

Related topics:

- You can use the `workflow:rules` templates to import a preconfigured `workflow: rules` entry.
- Common `if` clauses for `workflow:rules`.
- Use `rules` to run merge request pipelines.

`workflow:rules:variables`

Version history

You can use `variables` in `workflow:rules` to define variables for specific pipeline conditions.

When the condition matches, the variable is created and can be used by all jobs in the pipeline. If the variable is already defined at the global level, the `workflow` variable takes precedence and overrides the global variable.

Keyword type: Global keyword.

Possible inputs: Variable name and value pairs:

- The name can use only numbers, letters, and underscores (_).
- The value must be a string.

Example of `workflow:rules:variables`:

```
variables:
  DEPLOY_VARIABLE: "default-deploy"

workflow:
  rules:
    - if: $CI_COMMIT_REF_NAME == $CI_DEFAULT_BRANCH
      variables:
        DEPLOY_VARIABLE: "deploy-production"  # Override globally-
defined DEPLOY_VARIABLE
    - if: $CI_COMMIT_REF_NAME =~ /feature/
      variables:
        IS_A_FEATURE: "true"                  # Define a new variable.
    - when: always                            # Run the pipeline in
other cases

job1:
  variables:
    DEPLOY_VARIABLE: "job1-default-deploy"
  rules:
    - if: $CI_COMMIT_REF_NAME == $CI_DEFAULT_BRANCH
      variables:                                    # Override
DEPLOY_VARIABLE defined
        DEPLOY_VARIABLE: "job1-deploy-production"  # at the job level.
    - when: on_success                              # Run the job in
other cases
  script:
    - echo "Run script with $DEPLOY_VARIABLE as an argument"
    - echo "Run another script if $IS_A_FEATURE exists"

job2:
  script:
    - echo "Run script with $DEPLOY_VARIABLE as an argument"
    - echo "Run another script if $IS_A_FEATURE exists"
```

When the branch is the default branch:

- job1's `DEPLOY_VARIABLE` is `job1-deploy-production`.
- job2's `DEPLOY_VARIABLE` is `deploy-production`.

When the branch is `feature`:

- job1's `DEPLOY_VARIABLE` is `job1-default-deploy`, and `IS_A_FEATURE` is `true`.
- job2's `DEPLOY_VARIABLE` is `default-deploy`, and `IS_A_FEATURE` is `true`.

When the branch is something else:

- job1's `DEPLOY_VARIABLE` is `job1-default-deploy`.
- job2's `DEPLOY_VARIABLE` is `default-deploy`.

Job keywords

The following topics explain how to use keywords to configure CI/CD pipelines.

`after_script`

Use `after_script` to define an array of commands that run after each job, including failed jobs.

Keyword type: Job keyword. You can use it only as part of a job or in the `default` section.

Possible inputs: An array including:

- Single line commands.
- Long commands split over multiple lines.
- YAML anchors.

CI/CD variables are supported.

Example of `after_script`:

```
job:
  script:
    - echo "An example script section."
  after_script:
    - echo "Execute this command after the `script` section completes."
```

Additional details:

Scripts you specify in `after_script` execute in a new shell, separate from any `before_script` or `script` commands. As a result, they:

- Have the current working directory set back to the default (according to the variables which define how the runner processes Git requests).
- Don't have access to changes done by commands defined in the `before_script` or `script`, including:
  - Command aliases and variables exported in `script` scripts.
  - Changes outside of the working tree (depending on the runner executor), like software installed by a `before_script` or `script` script.
- Have a separate timeout, which is hard-coded to 5 minutes.
- Don't affect the job's exit code. If the `script` section succeeds and the `after_script` times out or fails, the job exits with code `0` (`Job Succeeded`).

If a job times out or is cancelled, the `after_script` commands do not execute. An issue exists to add support for executing `after_script` commands for timed-out or cancelled jobs.

Related topics:

- Use `after_script` with `default` to define a default array of commands that should run after all jobs.
- You can ignore non-zero exit codes.
- Use color codes with `after_script` to make job logs easier to review.
- Create custom collapsible sections to simplify job log output.

`allow_failure`

Use `allow_failure` to determine whether a pipeline should continue running when a job fails.

- To let the pipeline continue running subsequent jobs, use `allow_failure: true`.
- To stop the pipeline from running subsequent jobs, use `allow_failure: false`.

When jobs are allowed to fail (`allow_failure: true`) an orange warning () indicates that a job failed. However, the pipeline is successful and the associated commit is marked as passed with no warnings.

This same warning is displayed when:

- All other jobs in the stage are successful.
- All other jobs in the pipeline are successful.

The default value for `allow_failure` is:

- `true` for manual jobs.
- `false` for jobs that use `when: manual` inside `rules`.
- `false` in all other cases.

Keyword type: Job keyword. You can use it only as part of a job.

Possible inputs:

- `true` or `false`.

Example of `allow_failure`:

```
job1:
  stage: test
  script:
    - execute_script_1

job2:
  stage: test
  script:
    - execute_script_2
  allow_failure: true

job3:
  stage: deploy
  script:
    - deploy_to_staging
```

In this example, `job1` and `job2` run in parallel:

- If `job1` fails, jobs in the `deploy` stage do not start.
- If `job2` fails, jobs in the `deploy` stage can still start.

Additional details:

- You can use `allow_failure` as a subkey of `rules`.
- You can use `allow_failure: false` with a manual job to create a blocking manual job. A blocked pipeline does not run any jobs in later stages until the manual job is started and completes successfully.

`allow_failure:exit_codes`

Version history

Use `allow_failure:exit_codes` to control when a job should be allowed to fail. The job is `allow_failure: true` for any of the listed exit codes, and `allow_failure` false for any other exit code.

Keyword type: Job keyword. You can use it only as part of a job.

Possible inputs:

- A single exit code.
- An array of exit codes.

Example of `allow_failure`:

```
test_job_1:
  script:
    - echo "Run a script that results in exit code 1. This job fails."
    - exit 1
  allow_failure:
    exit_codes: 137

test_job_2:
  script:
    - echo "Run a script that results in exit code 137. This job is
allowed to fail."
    - exit 137
  allow_failure:
    exit_codes:
      - 137
      - 255
```

`artifacts`

Use `artifacts` to specify which files to save as job artifacts. Job artifacts are a list of files and directories that are attached to the job when it suc ceeds, fails, or always.

The artifacts are sent to GitLab after the job finishes. They are available for download in the GitLab UI if the size is smaller than the the maximum artifact size.

By default, jobs in later stages automatically download all the artifacts created by jobs in earlier stages. You can control artifact download behavior in jobs with `dependencies`.

When using the `needs` keyword, jobs can only download artifacts from the jobs defined in the `needs` configuration.

Job artifacts are only collected for successful jobs by default, and artifacts are restored after caches.

Read more about artifacts.

`artifacts:paths`

Paths are relative to the project directory (`$CI_PROJECT_DIR`) and can't directly link outside it.

Keyword type: Job keyword. You can use it only as part of a job or in the `default` section.

Possible inputs:

- An array of file paths, relative to the project directory.
- You can use Wildcards that use glob patterns and:
  - In GitLab Runner 13.0 and later, `doublestar.Glob`.
  - In GitLab Runner 12.10 and earlier, `filepath.Match`.

Example of `artifacts:paths`:

```
job:
  artifacts:
    paths:
      - binaries/
      - .config
```

This example creates an artifact with `.config` and all the files in the `binaries` directory.

Additional details:

- If not used with `artifacts:name`, the artifacts file is named `artifacts`, which becomes `artifacts.zip` when downloaded.

Related topics:

- To restrict which jobs a specific job fetches artifacts from, see `dependencies`.
- Create job artifacts.

`artifacts:exclude`

Version history

Use `artifacts:exclude` to prevent files from being added to an artifacts archive.

Keyword type: Job keyword. You can use it only as part of a job or in the `default` section.

Possible inputs:

- An array of file paths, relative to the project directory.
- You can use Wildcards that use glob or `doublestar.PathMatch` patterns.

Example of `artifacts:exclude`:

```
artifacts:
  paths:
    - binaries/
  exclude:
    - binaries/**/*.o
```

This example stores all files in `binaries/`, but not `*.o` files located in subdirectories of `binaries/`.

Additional details:

- `artifacts:exclude` paths are not searched recursively.
- Files matched by `artifacts:untracked` can be excluded using `artifacts:exclude` too.

Related topics:

- Exclude files from job artifacts.

`artifacts:expire_in`

Version history

Use `expire_in` to specify how long job artifacts are stored before they expire and are deleted. The `expire_in` setting does not affect:

- Artifacts from the latest job, unless keeping the latest job artifacts is:
  - Disabled at the project level.
  - Disabled instance-wide.
- Pipeline artifacts. You can't specify an expiration date for pipeline artifacts. See When pipeline artifacts are deleted for more information.

After their expiry, artifacts are deleted hourly by default (using a cron job), and are not accessible anymore.

Keyword type: Job keyword. You can use it only as part of a job or in the `default` section.

Possible inputs: The expiry time. If no unit is provided, the time is in seconds. Valid values include:

- `'42'`
- `42 seconds`
- `3 mins 4 sec`
- `2 hrs 20 min`
- `2h20min`
- `6 mos 1 day`

- `47 yrs 6 mos and 4d`
- `3 weeks and 2 days`
- `never`

Example of `artifacts:expire_in`:

```
job:
  artifacts:
    expire_in: 1 week
```

Additional details:

- The expiration time period begins when the artifact is uploaded and stored on GitLab. If the expiry time is not defined, it defaults to the instance wide setting.
- To override the expiration date and protect artifacts from being automatically deleted:
  - Select **Keep** on the job page.
  - In GitLab 13.3 and later, set the value of `expire_in` to `never`.

`artifacts:expose_as`

Introduced in GitLab 12.5.

Use the `artifacts:expose_as` keyword to expose job artifacts in the merge request UI.

Keyword type: Job keyword. You can use it only as part of a job or in the `default` section.

Possible inputs:

- The name to display in the merge request UI for the artifacts download link. Must be combined with `artifacts:paths`.

Example of `artifacts:expose_as`:

```
test:
  script: ["echo 'test' > file.txt"]
  artifacts:
    expose_as: 'artifact 1'
    paths: ['file.txt']
```

Additional details:

- If `artifacts:paths` uses CI/CD variables, the artifacts do not display in the UI.
- A maximum of 10 job artifacts per merge request can be exposed.
- Glob patterns are unsupported.
- If a directory is specified and there is more than one file in the directory, the link is to the job artifacts browser.
- If GitLab Pages is enabled, GitLab automatically renders the artifacts when the artifacts is a single file with one of these extensions:
  - `.html` or `.htm`
  - `.txt`
  - `.json`
  - `.xml`
  - `.log`

Related topics:

- Expose job artifacts in the merge request UI.

`artifacts:name`

Use the `artifacts:name` keyword to define the name of the created artifacts archive. You can specify a unique name for every archive.

If not defined, the default name is `artifacts`, which becomes `artifacts.zip` when downloaded.

Keyword type: Job keyword. You can use it only as part of a job or in the `default` section.

Possible inputs:

- The name of the artifacts archive. CI/CD variables are supported. Must be combined with `artifacts:paths`.

Example of `artifacts:name`:

To create an archive with a name of the current job:

```
job:
  artifacts:
    name: "job1-artifacts-file"
    paths:
      - binaries/
```

Related topics:

- Use CI/CD variables to define the artifacts name.

`artifacts:public`

Version history

On self-managed GitLab, by default this feature is not available. To make it available, ask an administrator to enable the feature flag named `non_public_artifacts`. On GitLab.com, this feature is not available.

Use `artifacts:public` to determine whether the job artifacts should be publicly available.

When `artifacts:public` is `true` (default), the artifacts in public pipelines are available for download by anonymous and guest users.

To deny read access for anonymous and guest users to artifacts in public pipelines, set `artifacts:public` to `false`:

Keyword type: Job keyword. You can use it only as part of a job or in the `default` section.

Possible inputs:

- `true` (default if not defined) or `false`.

Example of `artifacts:public`:

```
job:
  artifacts:
    public: false
```

`artifacts:reports`

Use `artifacts:reports` to collect artifacts generated by included templates in jobs.

Keyword type: Job keyword. You can use it only as part of a job or in the `default` section.

Possible inputs:

- See list of available artifacts reports types.

Example of `artifacts:reports`:

```
rspec:
  stage: test
  script:
    - bundle install
    - rspec --format RspecJunitFormatter --out rspec.xml
  artifacts:
    reports:
      junit: rspec.xml
```

Additional details:

- Combining reports in parent pipelines using artifacts from child pipelines is not supported. Track progress on adding support in this issue.
- To be able to browse the report output files, include the `artifacts:paths` keyword. Please note that this will upload and store the artifact twice.
- The test reports are collected regardless of the job results (success or failure). You can use `artifacts:expire_in` to set up an expiration date for artifacts reports.

`artifacts:untracked`

Use `artifacts:untracked` to add all Git untracked files as artifacts (along with the paths defined in `artifacts:paths`). `artifacts:untracked` ignores configuration in the repository's `.gitignore`, so matching artifacts in `.gitignore` are included.

Keyword type: Job keyword. You can use it only as part of a job or in the `default` section.

Possible inputs:

- `true` or `false` (default if not defined).

Example of `artifacts:untracked`:

Save all Git untracked files:

```
job:
  artifacts:
    untracked: true
```

Related topics:

- Add untracked files to artifacts.

`artifacts:when`

Use `artifacts:when` to upload artifacts on job failure or despite the failure.

Keyword type: Job keyword. You can use it only as part of a job or in the `default` section.

Possible inputs:

- `on_success` (default): Upload artifacts only when the job succeeds.
- `on_failure`: Upload artifacts only when the job fails.
- `always`: Always upload artifacts (except when jobs time out). For example, when uploading artifacts required to troubleshoot failing tests.

Example of `artifacts:when`:

```
job:
  artifacts:
    when: on_failure
```

`before_script`

Use `before_script` to define an array of commands that should run before each job's `script` commands, but after artifacts are restored.

Keyword type: Job keyword. You can use it only as part of a job or in the `default` section.

Possible inputs: An array including:

- Single line commands.
- Long commands split over multiple lines.
- YAML anchors.

CI/CD variables are supported.

Example of `before_script`:

```
job:
  before_script:
    - echo "Execute this command before any 'script:' commands."
  script:
    - echo "This command executes after the job's 'before_script'
commands."
```

Additional details:

- Scripts you specify in `before_script` are concatenated with any scripts you specify in the main `script`. The combined scripts execute together in a single shell.
- Using `before_script` at the top level, but not in the `default` section, is deprecated.

Related topics:

- Use `before_script` with `default` to define a default array of commands that should run before the `script` commands in all jobs.
- You can ignore non-zero exit codes.
- Use color codes with `before_script` to make job logs easier to review.
- Create custom collapsible sections to simplify job log output.

`cache`

Use `cache` to specify a list of files and directories to cache between jobs. You can only use paths that are in the local working copy.

Caching is shared between pipelines and jobs. Caches are restored before artifacts.

Learn more about caches in Caching in GitLab CI/CD.

`cache:paths`

Use the `cache:paths` keyword to choose which files or directories to cache.

Keyword type: Job keyword. You can use it only as part of a job or in the `default` section.

Possible inputs:

- An array of paths relative to the project directory (`$CI_PROJECT_DIR`). You can use wildcards that use glob patterns:
  - In GitLab Runner 13.0 and later, `doublestar.Glob`.

- In GitLab Runner 12.10 and earlier, `filepath.Match`.

Example of `cache:paths`:

Cache all files in `binaries` that end in `.apk` and the `.config` file:

```
rspec:
  script:
    - echo "This job uses a cache."
  cache:
    key: binaries-cache
    paths:
      - binaries/*.apk
      - .config
```

Related topics:

- See the common `cache` use cases for more `cache:paths` examples.

`cache:key`

Use the `cache:key` keyword to give each cache a unique identifying key. All jobs that use the same cache key use the same cache, including in different pipelines.

If not set, the default key is `default`. All jobs with the `cache` keyword but no `cache:key` share the `default` cache.

Must be used with `cache: path`, or nothing is cached.

Keyword type: Job keyword. You can use it only as part of a job or in the `default` section.

Possible inputs:

- A string.
- A predefined CI/CD variable.
- A combination of both.

Example of `cache:key`:

```
cache-job:
  script:
    - echo "This job uses a cache."
  cache:
    key: binaries-cache-$CI_COMMIT_REF_SLUG
    paths:
      - binaries/
```

Additional details:

- If you use **Windows Batch** to run your shell scripts you must replace `$` with `%`. For example: `key: %CI_COMMIT_REF_SLUG%`
- The `cache:key` value can't contain:
  - The `/` character, or the equivalent URI-encoded `%2F`.
  - Only the `.` character (any number), or the equivalent URI-encoded `%2E`.
- The cache is shared between jobs, so if you're using different paths for different jobs, you should also set a different `cache:key`. Otherwise cache content can be overwritten.

Related topics:

- You can specify a fallback cache key to use if the specified `cache:key` is not found.

- You can use multiple cache keys in a single job.
- See the common `cache` use cases for more `cache:key` examples.

`cache:key:files`

Introduced in GitLab 12.5.

Use the `cache:key:files` keyword to generate a new key when one or two specific files change. `cache:key:files` lets you reuse some caches, and rebuild them less often, which speeds up subsequent pipeline runs.

Keyword type: Job keyword. You can use it only as part of a job or in the `default` section.

Possible inputs:

- An array of one or two file paths.

Example of `cache:key:files`:

```
cache-job:
  script:
    - echo "This job uses a cache."
  cache:
    key:
      files:
        - Gemfile.lock
        - package.json
    paths:
      - vendor/ruby
      - node_modules
```

This example creates a cache for Ruby and Node.js dependencies. The cache is tied to the current versions of the `Gemfile.lock` and `package.json` files. When one of these files changes, a new cache key is computed and a new cache is created. Any future job runs that use the same `Gemfile.lock` and `package.json` with `cache:key:files` use the new cache, instead of rebuilding the dependencies.

Additional details:

- The cache `key` is a SHA computed from the most recent commits that changed each listed file. If neither file is changed in any commits, the fallback key is `default`.

`cache:key:prefix`

Introduced in GitLab 12.5.

Use `cache:key:prefix` to combine a prefix with the SHA computed for `cache:key:files`.

Keyword type: Job keyword. You can use it only as part of a job or in the `default` section.

Possible inputs:

- A string
- A predefined variables
- A combination of both.

Example of `cache:key:prefix`:

```
rspec:
  script:
    - echo "This rspec job uses a cache."
  cache:
    key:
      files:
        - Gemfile.lock
      prefix: $CI_JOB_NAME
    paths:
      - vendor/ruby
```

For example, adding a `prefix` of `$CI_JOB_NAME` causes the key to look like `rspec-feef9576d21ee9b6a32e30c5c79d0a0ceb68d1e5`. If a branch changes `Gemfile.lock`, that branch has a new SHA checksum for `cache:key:files`. A new cache key is generated, and a new cache is created for that key. If `Gemfile.lock` is not found, the prefix is added to `default`, so the key in the example would be `rspec-default`.

Additional details:

- If no file in `cache:key:files` is changed in any commits, the prefix is added to the `default` key.

`cache:untracked`

Use `untracked: true` to cache all files that are untracked in your Git repository:

Keyword type: Job keyword. You can use it only as part of a job or in the `default` section.

Possible inputs:

- `true` or `false` (default).

Example of `cache:untracked`:

```
rspec:
  script: test
  cache:
    untracked: true
```

Additional details:

- You can combine `cache:untracked` with `cache:paths` to cache all untracked files as well as files in the configured paths. For example:

```
  rspec:
    script: test
    cache:
      untracked: true
      paths:
        - binaries/
```

`cache:when`

Introduced in GitLab 13.5 and GitLab Runner v13.5.0.

Use `cache:when` to define when to save the cache, based on the status of the job.

Must be used with `cache: path`, or nothing is cached.

Keyword type: Job keyword. You can use it only as part of a job or in the `default` section.

Possible inputs:

- `on_success` (default): Save the cache only when the job succeeds.
- `on_failure`: Save the cache only when the job fails.
- `always`: Always save the cache.

Example of `cache:when`:

```
rspec:
  script: rspec
  cache:
    paths:
      - rspec/
    when: 'always'
```

This example stores the cache whether or not the job fails or succeeds.

`cache:policy`

To change the upload and download behavior of a cache, use the `cache:policy` keyword. By default, the job downloads the cache when the job starts, and uploads changes to the cache when the job ends. This caching style is the `pull-push` policy (default).

To set a job to only download the cache when the job starts, but never upload changes when the job finishes, use `cache:policy:pull`.

To set a job to only upload a cache when the job finishes, but never download the cache when the job starts, use `cache:policy:push`.

Use the `pull` policy when you have many jobs executing in parallel that use the same cache. This policy speeds up job execution and reduces load on the cache server. You can use a job with the `push` policy to build the cache.

Must be used with `cache: path`, or nothing is cached.

Keyword type: Job keyword. You can use it only as part of a job or in the `default` section.

Possible inputs:

- `pull`
- `push`
- `pull-push` (default)

Example of `cache:policy`:

```
prepare-dependencies-job:
  stage: build
  cache:
    key: gems
    paths:
      - vendor/bundle
    policy: push
  script:
    - echo "This job only downloads dependencies and builds the cache."
    - echo "Downloading dependencies..."

faster-test-job:
  stage: test
  cache:
    key: gems
    paths:
      - vendor/bundle
    policy: pull
  script:
    - echo "This job script uses the cache, but does not update it."
    - echo "Running tests..."
```

`coverage`

Use `coverage` with a custom regular expression to configure how code coverage is extracted from the job output. The coverage is shown in the UI if at least one line in the job output matches the regular expression.

To extract the code coverage value from the match, GitLab uses this smaller regular expression: `\d+(\.\d+)?`.

Possible inputs:

- A regular expression. Must start and end with `/`. Must match the coverage number. May match surrounding text as well, so you don't need to use a regular expression character group to capture the exact number.

Example of `coverage`:

```
job1:
  script: rspec
  coverage: '/Code coverage: \d+\.\d+/'
```

In this example:

1. GitLab checks the job log for a match with the regular expression. A line like `Code coverage: 67.89% of lines covered` would match.
2. GitLab then checks the matched fragment to find a match to `\d+(\.\d+)?`. The sample matching line above gives a code coverage of `67.89`.

Additional details:

- If there is more than one matched line in the job output, the last line is used (the first result of reverse search).
- If there are multiple matches in a single line, the last match is searched for the coverage number.
- If there are multiple coverage numbers found in the matched fragment, the first number is used.
- Leading zeros are removed.

- Coverage output from child pipelines is not recorded or displayed. Check the related issue for more details.

`dast_configuration` **ultimate**

Introduced in GitLab 14.1.

Use the `dast_configuration` keyword to specify a site profile and scanner profile to be used in a CI/CD configuration. Both profiles must first have been created in the project. The job's stage must be `dast`.

Keyword type: Job keyword. You can use only as part of a job.

Possible inputs: One each of `site_profile` and `scanner_profile`.

- Use `site_profile` to specify the site profile to be used in the job.
- Use `scanner_profile` to specify the scanner profile to be used in the job.

Example of `dast_configuration`:

```
stages:
  - build
  - dast

include:
  - template: DAST.gitlab-ci.yml

dast:
  dast_configuration:
    site_profile: "Example Co"
    scanner_profile: "Quick Passive Test"
```

In this example, the `dast` job extends the `dast` configuration added with the `include` keyword to select a specific site profile and scanner profile.

Additional details:

- Settings contained in either a site profile or scanner profile take precedence over those contained in the DAST template.

Related topics:

- Site profile.
- Scanner profile.

`dependencies`

Use the `dependencies` keyword to define a list of jobs to fetch artifacts from. You can also set a job to download no artifacts at all.

If you do not use `dependencies`, all artifacts from previous stages are passed to each job.

Keyword type: Job keyword. You can use it only as part of a job.

Possible inputs:

- The names of jobs to fetch artifacts from.
- An empty array (`[]`), to configure the job to not download any artifacts.

Example of `dependencies`:

```
build osx:
  stage: build
  script: make build:osx
  artifacts:
    paths:
      - binaries/

build linux:
  stage: build
  script: make build:linux
  artifacts:
    paths:
      - binaries/

test osx:
  stage: test
  script: make test:osx
  dependencies:
    - build osx

test linux:
  stage: test
  script: make test:linux
  dependencies:
    - build linux

deploy:
  stage: deploy
  script: make deploy
```

In this example, two jobs have artifacts: `build osx` and `build linux`. When `test osx` is executed, the artifacts from `build osx` are downloaded and extracted in the context of the build. The same thing happens for `test linux` and artifacts from `build linux`.

The `deploy` job downloads artifacts from all previous jobs because of the stage precedence.

Additional details:

- The job status does not matter. If a job fails or it's a manual job that isn't triggered, no error occurs.
- If the artifacts of a dependent job are expired or deleted, then the job fails.

`environment`

Use `environment` to define the environment that a job deploys to.

Keyword type: Job keyword. You can use it only as part of a job.

Possible inputs: The name of the environment the job deploys to, in one of these formats:

- Plain text, including letters, digits, spaces, and these characters: `-`, `_`, `/`, `$`, `{`, `}`.
- CI/CD variables, including predefined, project, group, instance, or variables defined in the `.gitlab-ci.yml` file. You can't use variables defined in a `script` section.

Example of `environment`:

```
deploy to production:
  stage: deploy
  script: git push production HEAD:main
  environment: production
```

Additional details:

- If you specify an `environment` and no environment with that name exists, an environment is created.

`environment:name`

Set a name for an environment.

Common environment names are `qa`, `staging`, and `production`, but you can use any name.

Keyword type: Job keyword. You can use it only as part of a job.

Possible inputs: The name of the environment the job deploys to, in one of these formats:

- Plain text, including letters, digits, spaces, and these characters: `-`, `_`, `/`, `$`, `{`, `}`.
- CI/CD variables, including predefined, project, group, instance, or variables defined in the `.gitlab-ci.yml` file. You can't use variables defined in a `script` section.

Example of `environment:name`:

```
deploy to production:
  stage: deploy
  script: git push production HEAD:main
  environment:
    name: production
```

`environment:url`

Set a URL for an environment.

Keyword type: Job keyword. You can use it only as part of a job.

Possible inputs: A single URL, in one of these formats:

- Plain text, like `https://prod.example.com`.
- CI/CD variables, including predefined, project, group, instance, or variables defined in the `.gitlab-ci.yml` file. You can't use variables defined in a `script` section.

Example of `environment:url`:

```
deploy to production:
  stage: deploy
  script: git push production HEAD:main
  environment:
    name: production
    url: https://prod.example.com
```

Additional details:

- After the job completes, you can access the URL by selecting a button in the merge request, environment, or deployment pages.

`environment:on_stop`

Closing (stopping) environments can be achieved with the `on_stop` keyword defined under `environment`. It declares a different job that runs to close the environment.

Keyword type: Job keyword. You can use it only as part of a job.

Additional details:

- See `environment:action` for more details and an example.

`environment:action`

Use the `action` keyword to specify how the job interacts with the environment.

Keyword type: Job keyword. You can use it only as part of a job.

Possible inputs: One of the following keywords:

Value	Description
`start`	Default value. Indicates that the job starts the environment. The deployment is created after the job starts.
`prepare`	Indicates that the job is only preparing the environment. It does not trigger deployments. Read more about preparing environments.
`stop`	Indicates that the job stops a deployment. For more detail, read Stop an environment.
`verify`	Indicates that the job is only verifying the environment. It does not trigger deployments. Read more about verifying environments.
`access`	Indicates that the job is only accessing the environment. It does not trigger deployments. Read more about accessing environments.

Example of `environment:action`:

```
stop_review_app:
  stage: deploy
  variables:
    GIT_STRATEGY: none
  script: make delete-app
  when: manual
  environment:
    name: review/$CI_COMMIT_REF_SLUG
    action: stop
```

`environment:auto_stop_in`

Introduced in GitLab 12.8.

The `auto_stop_in` keyword specifies the lifetime of the environment. When an environment expires, GitLab automatically stops it.

Keyword type: Job keyword. You can use it only as part of a job.

Possible inputs: A period of time written in natural language. For example, these are all equivalent:

- `168 hours`
- `7 days`
- `one week`

Example of `environment:auto_stop_in`:

```
review_app:
  script: deploy-review-app
  environment:
    name: review/$CI_COMMIT_REF_SLUG
    auto_stop_in: 1 day
```

When the environment for `review_app` is created, the environment's lifetime is set to `1 day`. Every time the review app is deployed, that lifetime is also reset to `1 day`.

Related topics:

- Environments auto-stop documentation.

`environment:kubernetes`

Introduced in GitLab 12.6.

Use the `kubernetes` keyword to configure deployments to a Kubernetes cluster that is associated with your project.

Keyword type: Job keyword. You can use it only as part of a job.

Example of `environment:kubernetes`:

```
deploy:
  stage: deploy
  script: make deploy-app
  environment:
    name: production
    kubernetes:
      namespace: production
```

This configuration sets up the `deploy` job to deploy to the `production` environment, using the `production` Kubernetes namespace.

Additional details:

- Kubernetes configuration is not supported for Kubernetes clusters that are managed by GitLab. To follow progress on support for GitLab-managed clusters, see the relevant issue.

Related topics:

- Available settings for `kubernetes`.

`environment:deployment_tier`

Introduced in GitLab 13.10.

Use the `deployment_tier` keyword to specify the tier of the deployment environment.

Keyword type: Job keyword. You can use it only as part of a job.

Possible inputs: One of the following:

- `production`
- `staging`
- `testing`
- `development`
- `other`

Example of `environment:deployment_tier`:

```
deploy:
  script: echo
  environment:
    name: customer-portal
    deployment_tier: production
```

Related topics:

- Deployment tier of environments.

Dynamic environments

Use CI/CD variables to dynamically name environments.

For example:

```
deploy as review app:
  stage: deploy
  script: make deploy
  environment:
    name: review/$CI_COMMIT_REF_SLUG
    url: https://$CI_ENVIRONMENT_SLUG.example.com/
```

The `deploy as review app` job is marked as a deployment to dynamically create the `review/$CI_COMMIT_REF_SLUG` environment. `$CI_COMMIT_REF_SLUG` is a CI/CD variable set by the runner. The `$CI_ENVIRONMENT_SLUG` variable is based on the environment name, but suitable for inclusion in URLs. If the `deploy as review app` job runs in a branch named `pow`, this environment would be accessible with a URL like `https://review-pow.example.com/`.

The common use case is to create dynamic environments for branches and use them as Review Apps. You can see an example that uses Review Apps at https://gitlab.com/gitlab-examples/review-apps-nginx/ .

`extends`

Use `extends` to reuse configuration sections. It's an alternative to YAML anchors and is a little more flexible and readable.

Keyword type: Job keyword. You can use it only as part of a job.

Possible inputs:

- The name of another job in the pipeline.
- A list (array) of names of other jobs in the pipeline.

Example of `extends`:

```
.tests:
  script: rake test
  stage: test
  only:
    refs:
      - branches

rspec:
  extends: .tests
  script: rake rspec
  only:
    variables:
      - $RSPEC
```

In this example, the `rspec` job uses the configuration from the `.tests` template job. When creating the pipeline, GitLab:

- Performs a reverse deep merge based on the keys.
- Merges the `.tests` content with the `rspec` job.
- Doesn't merge the values of the keys.

The result is this `rspec` job:

```
rspec:
  script: rake rspec
  stage: test
  only:
    refs:
      - branches
    variables:
      - $RSPEC
```

Additional details:

- In GitLab 12.0 and later, you can use multiple parents for `extends`.
- The `extends` keyword supports up to eleven levels of inheritance, but you should avoid using more than three levels.
- In the example above, `.tests` is a hidden job, but you can extend configuration from regular jobs as well.

Related topics:

- Reuse configuration sections by using `extends`.
- Use `extends` to reuse configuration from included configuration files.

`image`

Use `image` to specify a Docker image that the job runs in.

Keyword type: Job keyword. You can use it only as part of a job or in the `default` section.

Possible inputs: The name of the image, including the registry path if needed, in one of these formats:

- `<image-name>` (Same as using `<image-name>` with the `latest` tag)
- `<image-name>:<tag>`
- `<image-name>@<digest>`

CI/CD variables are supported.

Example of `image`:

```
default:
  image: ruby:3.0

rspec:
  script: bundle exec rspec

rspec 2.7:
  image: registry.example.com/my-group/my-project/ruby:2.7
  script: bundle exec rspec
```

In this example, the `ruby:3.0` image is the default for all jobs in the pipeline. The `rspec 2.7` job does not use the default, because it overrides the default with a job-specific `image` section.

Related topics:

- Run your CI/CD jobs in Docker containers.

`image:name`

The name of the Docker image that the job runs in. Similar to `image` used by itself.

Keyword type: Job keyword. You can use it only as part of a job or in the `default` section.

Possible inputs: The name of the image, including the registry path if needed, in one of these formats:

- `<image-name>` (Same as using `<image-name>` with the `latest` tag)
- `<image-name>:<tag>`
- `<image-name>@<digest>`

Example of `image:name`:

```
image:
  name: "registry.example.com/my/image:latest"
```

Related topics:

- Run your CI/CD jobs in Docker containers.

`image:entrypoint`

Command or script to execute as the container's entry point.

When the Docker container is created, the `entrypoint` is translated to the Docker `--entrypoint` option. The syntax is similar to the Dockerfile `ENTRYPOINT` directive, where each shell token is a separate string in the array.

Keyword type: Job keyword. You can use it only as part of a job or in the `default` section.

Possible inputs:

- A string.

Example of `image:entrypoint`:

```
image:
  name: super/sql:experimental
  entrypoint: [""]
```

Related topics:

- Override the entrypoint of an image.

`image:pull_policy`

Version history

On self-managed GitLab, by default this feature is not available. To make it available, ask an administrator to enable the feature flag named `ci_docker_image_pull_policy`. The feature is not ready for production use.

The pull policy that the runner uses to fetch the Docker image.

Keyword type: Job keyword. You can use it only as part of a job or in the `default` section.

Possible inputs:

- A single pull policy, or multiple pull policies in an array. Can be `always`, `if-not-present`, or `never`.

Examples of `image:pull_policy`:

```
job1:
  script: echo "A single pull policy."
  image:
    name: ruby:3.0
    pull_policy: if-not-present

job2:
  script: echo "Multiple pull policies."
  image:
    name: ruby:3.0
    pull_policy: [always, if-not-present]
```

Additional details:

- If the runner does not support the defined pull policy, the job fails with an error similar to: `ERROR: Job failed (system failure): the configured PullPolicies ([always]) are not allowed by AllowedPullPolicies ([never])`.

Related topics:

- Run your CI/CD jobs in Docker containers.
- How runner pull policies work.
- Using multiple pull policies.

`inherit`

Introduced in GitLab 12.9.

Use `inherit` to control inheritance of default keywords and variables.

`inherit:default`

Use `inherit:default` to control the inheritance of default keywords.

Keyword type: Job keyword. You can use it only as part of a job.

Possible inputs:

- `true` (default) or `false` to enable or disable the inheritance of all default keywords.
- A list of specific default keywords to inherit.

Example of `inherit:default`:

```
default:
  retry: 2
  image: ruby:3.0
  interruptible: true

job1:
  script: echo "This job does not inherit any default keywords."
  inherit:
    default: false

job2:
  script: echo "This job inherits only the two listed default keywords.
It does not inherit 'interruptible'."
  inherit:
    default:
      - retry
      - image
```

Additional details:

- You can also list default keywords to inherit on one line: `default: [keyword1, keyword2]`

`inherit:variables`

Use `inherit:variables` to control the inheritance of global variables keywords.

Keyword type: Job keyword. You can use it only as part of a job.

Possible inputs:

- `true` (default) or `false` to enable or disable the inheritance of all global variables.
- A list of specific variables to inherit.

Example of `inherit:variables`:

```
variables:
  VARIABLE1: "This is variable 1"
  VARIABLE2: "This is variable 2"
  VARIABLE3: "This is variable 3"

job1:
  script: echo "This job does not inherit any global variables."
  inherit:
    variables: false

job2:
  script: echo "This job inherits only the two listed global variables.
It does not inherit 'VARIABLE3'."
  inherit:
    variables:
      - VARIABLE1
      - VARIABLE2
```

Additional details:

- You can also list global variables to inherit on one line: `variables: [VARIABLE1, VARIABLE2]`

`interruptible`

Introduced in GitLab 12.3.

Use `interruptible` if a job should be canceled when a newer pipeline starts before the job completes.

This keyword has no effect if automatic cancellation of redundant pipelines is disabled. When enabled, a running job with `interruptible: true` is cancelled when starting a pipeline for a new change on the same branch.

You can't cancel subsequent jobs after a job with `interruptible: false` starts.

Keyword type: Job keyword. You can use it only as part of a job or in the `default` section.

Possible inputs:

- `true` or `false` (default).

Example of `interruptible`:

```
stages:
  - stage1
  - stage2
  - stage3

step-1:
  stage: stage1
  script:
    - echo "Can be canceled."
  interruptible: true

step-2:
  stage: stage2
  script:
    - echo "Can not be canceled."

step-3:
  stage: stage3
  script:
    - echo "Because step-2 can not be canceled, this step can never be
canceled, even though it's set as interruptible."
  interruptible: true
```

In this example, a new pipeline causes a running pipeline to be:

- Canceled, if only `step-1` is running or pending.
- Not canceled, after `step-2` starts.

Additional details:

- Only set `interruptible: true` if the job can be safely canceled after it has started, like a build job. Deployment jobs usually shouldn't be cancelled, to prevent partial deployments.
- To completely cancel a running pipeline, all jobs must have `interruptible: true`, or `interruptible: false` jobs must not have started.

needs

Version history

Use `needs` to execute jobs out-of-order. Relationships between jobs that use `needs` can be visualized as a directed acyclic graph.

You can ignore stage ordering and run some jobs without waiting for others to complete. Jobs in multiple stages can run concurrently.

Keyword type: Job keyword. You can use it only as part of a job.

Possible inputs:

- An array of jobs.
- An empty array (`[]`), to set the job to start as soon as the pipeline is created.

Example of `needs`:

```
linux:build:
  stage: build
  script: echo "Building linux..."

mac:build:
  stage: build
  script: echo "Building mac..."

lint:
  stage: test
  needs: []
  script: echo "Linting..."

linux:rspec:
  stage: test
  needs: ["linux:build"]
  script: echo "Running rspec on linux..."

mac:rspec:
  stage: test
  needs: ["mac:build"]
  script: echo "Running rspec on mac..."

production:
  stage: deploy
  script: echo "Running production..."
```

This example creates four paths of execution:

- Linter: The `lint` job runs immediately without waiting for the `build` stage to complete because it has no needs (`needs: []`).
- Linux path: The `linux:rspec` job runs as soon as the `linux:build` job finishes, without waiting for `mac:build` to finish.
- macOS path: The `mac:rspec` jobs runs as soon as the `mac:build` job finishes, without waiting for `linux:build` to finish.
- The `production` job runs as soon as all previous jobs finish: `linux:build`, `linux:rspec`, `mac:build`, `mac:rspec`.

Additional details:

- The maximum number of jobs that a single job can have in the `needs` array is limited:
  - For GitLab.com, the limit is 50. For more information, see our infrastructure issue.
  - For self-managed instances, the default limit is 50. This limit can be changed.
- If `needs` refers to a job that uses the `parallel` keyword, it depends on all jobs created in parallel, not just one job. It also downloads artifacts from all the parallel jobs by default. If the artifacts have the same name, they overwrite each other and only the last one downloaded is saved.
- In GitLab 14.1 and later you can refer to jobs in the same stage as the job you are configuring. This feature is enabled on GitLab.com and ready for production use. On self-managed GitLab 14.2 and later this feature is available by default.
- In GitLab 14.0 and older, you can only refer to jobs in earlier stages. Stages must be explicitly defined for all jobs that use the `needs` keyword, or are referenced in a job's `needs` section.
- In GitLab 13.9 and older, if `needs` refers to a job that might not be added to a pipeline because of `only`, `except`, or `rules`, the pipeline might fail to create.

`needs:artifacts`

Introduced in GitLab 12.6.

When a job uses `needs`, it no longer downloads all artifacts from previous stages by default, because jobs with `needs` can start before earlier stages complete. With `needs` you can only download artifacts from the jobs listed in the `needs` configuration.

Use `artifacts: true` (default) or `artifacts: false` to control when artifacts are downloaded in jobs that use `needs`.

Keyword type: Job keyword. You can use it only as part of a job. Must be used with `needs:job`.

Possible inputs:

- `true` (default) or `false`.

Example of `needs:artifacts`:

```
test-job1:
  stage: test
  needs:
    - job: build_job1
      artifacts: true

test-job2:
  stage: test
  needs:
    - job: build_job2
      artifacts: false

test-job3:
  needs:
    - job: build_job1
      artifacts: true
    - job: build_job2
    - build_job3
```

In this example:

- The `test-job1` job downloads the `build_job1` artifacts
- The `test-job2` job does not download the `build_job2` artifacts.
- The `test-job3` job downloads the artifacts from all three `build_jobs`, because `artifacts` is `true`, or defaults to `true`, for all three needed jobs.

Additional details:

- In GitLab 12.6 and later, you can't combine the `dependencies` keyword with `needs`.

`needs:project` premium

Introduced in GitLab 12.7.

Use `needs:project` to download artifacts from up to five jobs in other pipelines. The artifacts are downloaded from the latest successful pipeline for the specified ref. To specify multiple jobs, add each as separate array items under the `needs` keyword.

If there is a pipeline running for the specified ref, a job with `needs:project` does not wait for the pipeline to complete. Instead, the job downloads the artifact from the latest pipeline that completed successfully.

`needs:project` must be used with `job`, `ref`, and `artifacts`.

Keyword type: Job keyword. You can use it only as part of a job.

Possible inputs:

- `needs:project`: A full project path, including namespace and group.
- `job`: The job to download artifacts from.
- `ref`: The ref to download artifacts from.
- `artifacts`: Must be `true` to download artifacts.

Examples of `needs:project`:

```
build_job:
  stage: build
  script:
    - ls -lhR
  needs:
    - project: namespace/group/project-name
      job: build-1
      ref: main
      artifacts: true
    - project: namespace/group/project-name-2
      job: build-2
      ref: main
      artifacts: true
```

In this example, `build_job` downloads the artifacts from the latest successful `build-1` and `build-2` jobs on the `main` branches in the `group/project-name` and `group/project-name-2` projects.

In GitLab 13.3 and later, you can use CI/CD variables in `needs:project`, for example:

```
build_job:
  stage: build
  script:
    - ls -lhR
  needs:
    - project: $CI_PROJECT_PATH
      job: $DEPENDENCY_JOB_NAME
      ref: $ARTIFACTS_DOWNLOAD_REF
      artifacts: true
```

Additional details:

- To download artifacts from a different pipeline in the current project, set `project` to be the same as the current project, but use a different ref than the current pipeline. Concurrent pipelines running on the same ref could override the artifacts.
- The user running the pipeline must have at least the Reporter role for the group or project, or the group/project must have public visibility.
- You can't use `needs:project` in the same job as `trigger`.
- When using `needs:project` to download artifacts from another pipeline, the job does not wait for the needed job to complete. Directed acyclic graph behavior is limited to jobs in the same pipeline. Make sure that the needed job in the other pipeline completes before the job that needs it tries to download the artifacts.
- You can't download artifacts from jobs that run in `parallel`.
- Support for CI/CD variables in `project`, `job`, and `ref` was introduced in GitLab 13.3. Feature flag removed in GitLab 13.4.

Related topics:

- To download artifacts between parent-child pipelines, use `needs:pipeline:job`.

`needs:pipeline:job`

Introduced in GitLab 13.7.

A child pipeline can download artifacts from a job in its parent pipeline or another child pipeline in the same parent-child pipeline hierarchy.

Keyword type: Job keyword. You can use it only as part of a job.

Possible inputs:

- `needs:pipeline`: A pipeline ID. Must be a pipeline present in the same parent-child pipeline hierarchy.
- `job`: The job to download artifacts from.

Example of `needs:pipeline:job`:

- Parent pipeline (`.gitlab-ci.yml`):

```
create-artifact:
  stage: build
  script: echo "sample artifact" > artifact.txt
  artifacts:
    paths: [artifact.txt]

child-pipeline:
  stage: test
  trigger:
    include: child.yml
    strategy: depend
  variables:
    PARENT_PIPELINE_ID: $CI_PIPELINE_ID
```

Child pipeline (`child.yml`):

```
use-artifact:
  script: cat artifact.txt
  needs:
    - pipeline: $PARENT_PIPELINE_ID
      job: create-artifact
```

In this example, the `create-artifact` job in the parent pipeline creates some artifacts. The `child-pipeline` job triggers a child pipeline, and passes the `CI_PIPELINE_ID` variable to the child pipeline as a new `PARENT_PIPELINE_ID` variable. The child pipeline can use that variable in `needs:pipeline` to download artifacts from the parent pipeline.

Additional details:

- The `pipeline` attribute does not accept the current pipeline ID (`$CI_PIPELINE_ID`). To download artifacts from a job in the current pipeline, use `needs`.

`needs:optional`

Version history

To need a job that sometimes does not exist in the pipeline, add `optional: true` to the `needs` configuration. If not defined, `optional: false` is the default.

Jobs that use `rules`, `only`, or `except` might not always be added to a pipeline. GitLab checks the `needs` relationships before starting a pipeline:

- If the needs entry has `optional: true` and the needed job is present in the pipeline, the job waits for it to complete before starting.
- If the needed job is not present, the job can start when all other needs requirements are met.
- If the `needs` section contains only optional jobs, and none are added to the pipeline, the job starts immediately (the same as an empty `needs` entry: `needs: []`).
- If a needed job has `optional: false`, but it was not added to the pipeline, the pipeline fails to start with an error similar to: `'job1' job needs 'job2' job, but it was not added to the pipeline`.

Keyword type: Job keyword. You can use it only as part of a job.

Example of `needs:optional`:

```
build-job:
  stage: build

test-job1:
  stage: test

test-job2:
  stage: test
  rules:
    - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH

deploy-job:
  stage: deploy
  needs:
    - job: test-job2
      optional: true
    - job: test-job1

review-job:
  stage: deploy
  needs:
    - job: test-job2
      optional: true
```

In this example:

- `build-job`, `test-job1`, and `test-job2` start in stage order.
- When the branch is the default branch, `test-job2` is added to the pipeline, so:
  - `deploy-job` waits for both `test-job1` and `test-job2` to complete.
  - `review-job` waits for `test-job2` to complete.
- When the branch is not the default branch, `test-job2` is not added to the pipeline, so:
  - `deploy-job` waits for only `test-job1` to complete, and does not wait for the missing `test-job2`.
  - `review-job` has no other needed jobs and starts immediately (at the same time as `build-job`), like `needs: []`.

`needs:pipeline`

You can mirror the pipeline status from an upstream pipeline to a bridge job by using the `needs:pipeline` keyword. The latest pipeline status from the default branch is replicated to the bridge job.

Keyword type: Job keyword. You can use it only as part of a job.

Possible inputs:

- A full project path, including namespace and group. If the project is in the same group or namespace, you can omit them from the `project` keyword. For example: `project: group/project-name` or `project: project-name`.

Example of `needs:pipeline`:

```
upstream_bridge:
  stage: test
  needs:
    pipeline: other/project
```

Additional details:

- If you add the `job` keyword to `needs:pipeline`, the job no longer mirrors the pipeline status. The behavior changes to `needs:pipeline:job`.

`only` / `except`

`only` and `except` are not being actively developed. `rules` is the preferred keyword to control when to add jobs to pipelines.

You can use `only` and `except` to control when to add jobs to pipelines.

- Use `only` to define when a job runs.
- Use `except` to define when a job **does not** run.

See specify when jobs run with `only` and `except` for more details and examples.

`only:refs` / `except:refs`

Use the `only:refs` and `except:refs` keywords to control when to add jobs to a pipeline based on branch names or pipeline types.

`only:refs` and `except:refs` are not being actively developed. `rules:if` is the preferred keyword when using refs, regular expressions, or variables to control when to add jobs to pipelines.

Keyword type: Job keyword. You can use it only as part of a job.

Possible inputs: An array including any number of:

- Branch names, for example `main` or `my-feature-branch`.
- Regular expressions that match against branch names, for example `/^feature-.*/`.
- The following keywords:

  Value`api`**Description** For pipelines triggered by the pipelines API. `branches` When the Git reference for a pipeline is a branch. `chat` For pipelines created by using a GitLab ChatOps command. `external` When you use CI services other than GitLab. `external_pull_requests` When an external pull request on GitHub is created or updated (See Pipelines for external pull requests). `merge_requests` For pipelines created when a merge request is created or updated. Enables merge request pipelines, merged results pipelines, and merge trains. `pipelines` For multi-project pipelines created by using the API with `CI_JOB_TOKEN`, or the `trigger` keyword. `pushes` For pipelines triggered by a `git push` event, including for branches and tags. `schedules` For scheduled pipelines. `tags` When the Git reference for a pipeline is a tag. `triggers` For pipelines created by using a trigger token. `web` For pipelines created by selecting **Run pipeline** in the GitLab UI, from the project's **CI/CD > Pipelines** section.

Example of `only:refs` and `except:refs`:

```
job1:
  script: echo
  only:
    - main
    - /^issue-.*$/
    - merge_requests

job2:
  script: echo
  except:
    - main
    - /^stable-branch.*$/
    - schedules
```

Additional details:

- Scheduled pipelines run on specific branches, so jobs configured with `only: branches` run on scheduled pipelines too. Add `except: schedules` to prevent jobs with `only: branches` from running on scheduled pipelines.
- `only` or `except` used without any other keywords are equivalent to `only: refs` or `except: refs`. For example, the following two jobs configurations have the same behavior:

```
  job1:
    script: echo
    only:
      - branches

  job2:
    script: echo
    only:
      refs:
        - branches
```

If a job does not use `only`, `except`, or `rules`, then `only` is set to `branches` and `tags` by default.

For example, `job1` and `job2` are equivalent:

```
job1:
  script: echo "test"

job2:
  script: echo "test"
  only:
    - branches
    - tags
```

`only:variables` / `except:variables`

Use the `only:variables` or `except:variables` keywords to control when to add jobs to a pipeline, based on the status of CI/CD variables.

`only:variables` and `except:variables` are not being actively developed. `rules:if` is the preferred keyword when using refs, regular expressions, or variables to control when to add jobs to pipelines.

Keyword type: Job keyword. You can use it only as part of a job.

Possible inputs:

- An array of CI/CD variable expressions.

Example of `only:variables`:

```
deploy:
  script: cap staging deploy
  only:
    variables:
      - $RELEASE == "staging"
      - $STAGING
```

Related topics:

- `only:variables` and `except:variables` examples.

`only:changes` / `except:changes`

Use the `changes` keyword with `only` to run a job, or with `except` to skip a job, when a Git push event modifies a file.

Use `changes` in pipelines with the following refs:

- `branches`
- `external_pull_requests`
- `merge_requests` (see additional details about using `only:changes` with merge request pipelines)

`only:changes` and `except:changes` are not being actively developed. `rules:changes` is the preferred keyword when using changed files to control when to add jobs to pipelines.

Keyword type: Job keyword. You can use it only as part of a job.

Possible inputs: An array including any number of:

- Paths to files.
- Wildcard paths for single directories, for example `path/to/directory/*`, or a directory and all its subdirectories, for example `path/to/directory/**/*`.
- Wildcard glob paths for all files with the same extension or multiple extensions, for example `*.md` or `path/to/directory/*.{rb,py,sh}`. See the Ruby `fnmatch` documentation for the supported syntax list.
- Wildcard paths to files in the root directory, or all directories, wrapped in double quotes. For example `"*.json"` or `"**/*.json"`.

Example of `only:changes`:

```
docker build:
  script: docker build -t my-image:$CI_COMMIT_REF_SLUG .
  only:
    refs:
      - branches
    changes:
      - Dockerfile
      - docker/scripts/*
      - dockerfiles/**/*
      - more_scripts/*.{rb,py,sh}
      - "**/*.json"
```

Additional details:

- `changes` resolves to `true` if any of the matching files are changed (an `OR` operation).
- If you use refs other than `branches`, `external_pull_requests`, or `merge_requests`, `changes` can't determine if a given file is new or old and always returns `true`.
- If you use `only: changes` with other refs, jobs ignore the changes and always run.
- If you use `except: changes` with other refs, jobs ignore the changes and never run.

Related topics:

- `only: changes` and `except: changes` examples.
- If you use `changes` with only allow merge requests to be merged if the pipeline succeeds, you should also use `only:merge_requests`.
- Jobs or pipelines can run unexpectedly when using `only: changes`.

`only:kubernetes` / `except:kubernetes`

Use `only:kubernetes` or `except:kubernetes` to control if jobs are added to the pipeline when the Kubernetes service is active in the project.

`only:refs` and `except:refs` are not being actively developed. Use `rules:if` with the `CI_KUBERNETES_ACTIVE` predefined CI/CD variable to control if jobs are added to the pipeline when the Kubernetes service is active in the project.

Keyword type: Job-specific. You can use it only as part of a job.

Possible inputs:

- The `kubernetes` strategy accepts only the `active` keyword.

Example of `only:kubernetes`:

```
deploy:
  only:
    kubernetes: active
```

In this example, the `deploy` job runs only when the Kubernetes service is active in the project.

`pages`

Use `pages` to define a GitLab Pages job that uploads static content to GitLab. The content is then published as a website.

Keyword type: Job name.

Example of `pages`:

```
pages:
  stage: deploy
  script:
    - mkdir .public
    - cp -r * .public
    - mv .public public
  artifacts:
    paths:
      - public
  rules:
    - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH
```

This example moves all files from the root of the project to the `public/` directory. The `.public` workaround is so `cp` does not also copy `public/` to itself in an infinite loop.

Additional details:

You must:

- Place any static content in a `public/` directory.
- Define `artifacts` with a path to the `public/` directory.

`parallel`

Use `parallel` to run a job multiple times in parallel in a single pipeline.

Multiple runners must exist, or a single runner must be configured to run multiple jobs concurrently.

Parallel jobs are named sequentially from `job_name 1/N` to `job_name N/N`.

Keyword type: Job keyword. You can use it only as part of a job.

Possible inputs:

- A numeric value from 2 to `50`.

Example of `parallel`:

```
test:
  script: rspec
  parallel: 5
```

This example creates 5 jobs that run in parallel, named `test 1/5` to `test 5/5`.

Additional details:

- Every parallel job has a `CI_NODE_INDEX` and `CI_NODE_TOTAL` predefined CI/CD variable set.

Related topics:

- Parallelize large jobs.

`parallel:matrix`

Version history

Use `parallel:matrix` to run a job multiple times in parallel in a single pipeline, but with different variable values for each instance of the job.

Multiple runners must exist, or a single runner must be configured to run multiple jobs concurrently.

Keyword type: Job keyword. You can use it only as part of a job.

Possible inputs: An array of hashes of variables:

- The variable names can use only numbers, letters, and underscores (_).
- The values must be either a string, or an array of strings.
- The number of permutations cannot exceed 50.

Example of `parallel:matrix`:

```
deploystacks:
  stage: deploy
  script:
    - bin/deploy
  parallel:
    matrix:
      - PROVIDER: aws
        STACK:
          - monitoring
          - app1
          - app2
      - PROVIDER: ovh
        STACK: [monitoring, backup, app]
      - PROVIDER: [gcp, vultr]
        STACK: [data, processing]
```

The example generates 10 parallel `deploystacks` jobs, each with different values for `PROVIDER` and `STACK`:

```
deploystacks: [aws, monitoring]
deploystacks: [aws, app1]
deploystacks: [aws, app2]
deploystacks: [ovh, monitoring]
deploystacks: [ovh, backup]
deploystacks: [ovh, app]
deploystacks: [gcp, data]
deploystacks: [gcp, processing]
deploystacks: [vultr, data]
deploystacks: [vultr, processing]
```

Related topics:

- Run a one-dimensional matrix of parallel jobs.
- Run a matrix of triggered parallel jobs.
- Select different runner tags for each parallel matrix job.

`release`

Introduced in GitLab 13.2.

Use `release` to create a release.

The release job must have access to the `release-cli`, which must be in the `$PATH`.

If you use the Docker executor, you can use this image from the GitLab Container Registry: `registry.gitlab.com/gitlab-org/release-cli:latest`

If you use the Shell executor or similar, install `release-cli` on the server where the runner is registered.

Keyword type: Job keyword. You can use it only as part of a job.

Possible inputs: The `release` subkeys:

- `tag_name`
- `tag_message` (optional)
- `name` (optional)
- `description`
- `ref` (optional)
- `milestones` (optional)
- `released_at` (optional)
- `assets:links` (optional)

Example of `release` keyword:

```
release_job:
  stage: release
  image: registry.gitlab.com/gitlab-org/release-cli:latest
  rules:
    - if: $CI_COMMIT_TAG                 # Run this job when a tag is
created manually
  script:
    - echo "Running the release job."
  release:
    tag_name: $CI_COMMIT_TAG
    name: 'Release $CI_COMMIT_TAG'
    description: 'Release created using the release-cli.'
```

This example creates a release:

- When you push a Git tag.
- When you add a Git tag in the UI at **Repository > Tags**.

Additional details:

- All release jobs, except trigger jobs, must include the `script` keyword. A release job can use the output from script commands. If you don't need the script, you can use a placeholder:

  ```
  script:
    - echo "release job"
  ```

- An issue exists to remove this requirement.
- The `release` section executes after the `script` keyword and before the `after_script`.
- A release is created only if the job's main script succeeds.
- If the release already exists, it is not updated and the job with the `release` keyword fails.

Related topics:

- CI/CD example of the `release` keyword.
- Create multiple releases in a single pipeline.
- Use a custom SSL CA certificate authority.

`release:tag_name`

Required. The Git tag for the release.

If the tag does not exist in the project yet, it is created at the same time as the release. New tags use the SHA associated with the pipeline.

Keyword type: Job keyword. You can use it only as part of a job.

Possible inputs:

- A tag name.

CI/CD variables are supported.

Example of `release:tag_name`:

To create a release when a new tag is added to the project:

- Use the `$CI_COMMIT_TAG` CI/CD variable as the `tag_name`.
- Use `rules:if` or `only: tags` to configure the job to run only for new tags.

```
job:
  script: echo "Running the release job for the new tag."
  release:
    tag_name: $CI_COMMIT_TAG
    description: 'Release description'
  rules:
    - if: $CI_COMMIT_TAG
```

To create a release and a new tag at the same time, your `rules` or `only` should **not** configure the job to run only for new tags. A semantic versioning example:

```
job:
  script: echo "Running the release job and creating a new tag."
  release:
    tag_name: ${MAJOR}_${MINOR}_${REVISION}
    description: 'Release description'
  rules:
    - if: $CI_PIPELINE_SOURCE == "schedule"
```

`release:tag_message`

Introduced in GitLab 15.3. Supported by `release-cli` v0.12.0 or later.

If the tag does not exist, the newly created tag is annotated with the message specifed by `tag_message`. If omitted, a lightweight tag is created.

Keyword type: Job keyword. You can use it only as part of a job.

Possible inputs:

- A text string.

Example of `release:tag_message`:

```
release_job:
  stage: release
  release:
    tag_name: $CI_COMMIT_TAG
    description: 'Release description'
    tag_message: 'Annotated tag message'
```

`release:name`

The release name. If omitted, it is populated with the value of `release: tag_name`.

Keyword type: Job keyword. You can use it only as part of a job.

Possible inputs:

- A text string.

Example of `release:name`:

```
release_job:
  stage: release
  release:
    name: 'Release $CI_COMMIT_TAG'
```

`release:description`

The long description of the release.

Keyword type: Job keyword. You can use it only as part of a job.

Possible inputs:

- A string with the long description.
- The path to a file that contains the description. Introduced in GitLab 13.7.
  - The file location must be relative to the project directory (`$CI_PROJECT_DIR`).
  - If the file is a symbolic link, it must be in the `$CI_PROJECT_DIR`.
  - The `./path/to/file` and filename can't contain spaces.

Example of `release:description`:

```
job:
  release:
    tag_name: ${MAJOR}_${MINOR}_${REVISION}
    description: './path/to/CHANGELOG.md'
```

`release:ref`

The `ref` for the release, if the `release: tag_name` doesn't exist yet.

Keyword type: Job keyword. You can use it only as part of a job.

Possible inputs:

- A commit SHA, another tag name, or a branch name.

`release:milestones`

The title of each milestone the release is associated with.

`release:released_at`

The date and time when the release is ready.

Possible inputs:

- A date enclosed in quotes and expressed in ISO 8601 format.

Example of `release:released_at`:

```
released_at: '2021-03-15T08:00:00Z'
```

Additional details:

- If it is not defined, the current date and time is used.

`release:assets:links`

Introduced in GitLab 13.12.

Use `release:assets:links` to include asset links in the release.

Requires `release-cli` version v0.4.0 or later.

Example of `release:assets:links`:

```
assets:
  links:
    - name: 'asset1'
      url: 'https://example.com/assets/1'
    - name: 'asset2'
      url: 'https://example.com/assets/2'
      filepath: '/pretty/url/1' # optional
      link_type: 'other' # optional
```

`resource_group`

Introduced in GitLab 12.7.

Use `resource_group` to create a resource group that ensures a job is mutually exclusive across different pipelines for the same project.

For example, if multiple jobs that belong to the same resource group are queued simultaneously, only one of the jobs starts. The other jobs wait until the `resource_group` is free.

Resource groups behave similar to semaphores in other programming languages.

You can define multiple resource groups per environment. For example, when deploying to physical devices, you might have multiple physical devices. Each device can be deployed to, but only one deployment can occur per device at any given time.

Keyword type: Job keyword. You can use it only as part of a job.

Possible inputs:

- Only letters, digits, `-`, `_`, `/`, `$`, `{`, `}`, `.`, and spaces. It can't start or end with `/`. CI/CD variables are supported.

Example of `resource_group`:

```
deploy-to-production:
  script: deploy
  resource_group: production
```

In this example, two `deploy-to-production` jobs in two separate pipelines can never run at the same time. As a result, you can ensure that concurrent deployments never happen to the production environment.

Related topics:

- Pipeline-level concurrency control with cross-project/parent-child pipelines.

`retry`

Use `retry` to configure how many times a job is retried if it fails. If not defined, defaults to `0` and jobs do not retry.

When a job fails, the job is processed up to two more times, until it succeeds or reaches the maximum number of retries.

By default, all failure types cause the job to be retried. Use `retry:when` to select which failures to retry on.

Keyword type: Job keyword. You can use it only as part of a job or in the `default` section.

Possible inputs:

- `0` (default), `1`, or `2`.

Example of `retry`:

```
test:
  script: rspec
  retry: 2
```

`retry:when`

Use `retry:when` with `retry:max` to retry jobs for only specific failure cases. `retry:max` is the maximum number of retries, like `retry`, and can be `0`, `1`, or `2`.

Keyword type: Job keyword. You can use it only as part of a job or in the `default` section.

Possible inputs:

- A single failure type, or an array of one or more failure types:

- `always`: Retry on any failure (default).
- `unknown_failure`: Retry when the failure reason is unknown.
- `script_failure`: Retry when the script failed.
- `api_failure`: Retry on API failure.
- `stuck_or_timeout_failure`: Retry when the job got stuck or timed out.
- `runner_system_failure`: Retry if there is a runner system failure (for example, job setup failed).
- `runner_unsupported`: Retry if the runner is unsupported.
- `stale_schedule`: Retry if a delayed job could not be executed.
- `job_execution_timeout`: Retry if the script exceeded the maximum execution time set for the job.
- `archived_failure`: Retry if the job is archived and can't be run.
- `unmet_prerequisites`: Retry if the job failed to complete prerequisite tasks.
- `scheduler_failure`: Retry if the scheduler failed to assign the job to a runner.
- `data_integrity_failure`: Retry if there is a structural integrity problem detected.

Example of `retry:when` (single failure type):

```
test:
  script: rspec
  retry:
    max: 2
    when: runner_system_failure
```

If there is a failure other than a runner system failure, the job is not retried.

Example of `retry:when` (array of failure types):

```
test:
  script: rspec
  retry:
    max: 2
    when:
      - runner_system_failure
      - stuck_or_timeout_failure
```

Related topics:

You can specify the number of retry attempts for certain stages of job execution using variables.

`rules`

Introduced in GitLab 12.3.

Use `rules` to include or exclude jobs in pipelines.

Rules are evaluated when the pipeline is created, and evaluated *in order* until the first match. When a match is found, the job is either included or excluded from the pipeline, depending on the configuration.

You cannot use dotenv variables created in job scripts in rules, because rules are evaluated before any jobs run.

`rules` replaces `only/except` and they can't be used together in the same job. If you configure one job to use both keywords, the GitLab returns a `key may not be used with rules` error.

`rules` accepts an array of rules defined with:

- `if`
- `changes`
- `exists`
- `allow_failure`
- `variables`
- `when`

You can combine multiple keywords together for complex rules.

The job is added to the pipeline:

- If an `if`, `changes`, or `exists` rule matches and also has `when: on_success` (default), `when: delayed`, or `when: always`.
- If a rule is reached that is only `when: on_success`, `when: delayed`, or `when: always`.

The job is not added to the pipeline:

- If no rules match.
- If a rule matches and has `when: never`.

You can use `!reference` tags to reuse `rules` configuration in different jobs.

`rules:if`

Use `rules:if` clauses to specify when to add a job to a pipeline:

- If an `if` statement is true, add the job to the pipeline.
- If an `if` statement is true, but it's combined with `when: never`, do not add the job to the pipeline.
- If no `if` statements are true, do not add the job to the pipeline.

`if` clauses are evaluated based on the values of predefined CI/CD variables or custom CI/CD variables.

Keyword type: Job-specific and pipeline-specific. You can use it as part of a job to configure the job behavior, or with `workflow` to configure the pipeline behavior.

Possible inputs:

- A CI/CD variable expression.

Example of `rules:if`:

```
job:
  script: echo "Hello, Rules!"
  rules:
    - if: $CI_MERGE_REQUEST_SOURCE_BRANCH_NAME =~ /^feature/ &&
$CI_MERGE_REQUEST_TARGET_BRANCH_NAME != $CI_DEFAULT_BRANCH
      when: never
    - if: $CI_MERGE_REQUEST_SOURCE_BRANCH_NAME =~ /^feature/
      when: manual
      allow_failure: true
    - if: $CI_MERGE_REQUEST_SOURCE_BRANCH_NAME
```

Additional details:

- If a rule matches and has no `when` defined, the rule uses the `when` defined for the job, which defaults to `on_success` if not defined.
- In GitLab 14.5 and earlier, you can define `when` once per rule, or once at the job-level, which applies to all rules. You can't mix `when` at the job-level with `when` in rules.
- In GitLab 14.6 and later, you can mix `when` at the job-level with `when` in rules. `when` configuration in `rules` takes precedence over `when` at the job-level.
- Unlike variables in `script` sections, variables in rules expressions are always formatted as `$VARIABLE`.
  - You can use `rules:if` with `include` to conditionally include other configuration files.
- In GitLab 15.0 and later, variables on the right side of `=~` and `!~` expressions are evaluated as regular expressions.

Related topics:

- Common `if` expressions for `rules`.
- Avoid duplicate pipelines.
- Use `rules` to run merge request pipelines.

`rules:changes`

Use `rules:changes` to specify when to add a job to a pipeline by checking for changes to specific files.

You should use `rules: changes` only with **branch pipelines** or **merge request pipelines**. You can use `rules: changes` with other pipeline types, but `rules: changes` always evaluates to true when there is no Git `push` event. Tag pipelines, scheduled pipelines, manual pipelines, and so on do **not** have a Git `push` event associated with them. A `rules: changes` job is **always** added to those pipelines if there is no `if` that limits the job to branch or merge request pipelines.

Keyword type: Job keyword. You can use it only as part of a job.

Possible inputs:

- An array of file paths. In GitLab 13.6 and later, file paths can include variables.
- Alternatively, the array of file paths can be in `rules:changes:paths`.

Example of `rules:changes`:

```
docker build:
  script: docker build -t my-image:$CI_COMMIT_REF_SLUG .
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
      changes:
        - Dockerfile
      when: manual
      allow_failure: true
```

- If the pipeline is a merge request pipeline, check `Dockerfile` for changes.
- If `Dockerfile` has changed, add the job to the pipeline as a manual job, and the pipeline continues running even if the job is not triggered (a `llow_failure: true`).
- If `Dockerfile` has not changed, do not add job to any pipeline (same as `when: never`).
- `rules:changes:paths` is the same as `rules:changes` without any subkeys.

Additional details:

- `rules: changes` works the same way as `only: changes` and `except: changes`.
- You can use `when: never` to implement a rule similar to `except:changes`.
- `changes` resolves to `true` if any of the matching files are changed (an `OR` operation).

Related topics:

- Jobs or pipelines can run unexpectedly when using `rules: changes`.

`rules:changes:paths`

Introduced in GitLab 15.2.

Use `rules:changes` to specify that a job only be added to a pipeline when specific files are changed, and use `rules:changes:paths` to specify the files.

`rules:changes:paths` is the same as using `rules:changes` without any subkeys. All additional details and related topics are the same.

Keyword type: Job keyword. You can use it only as part of a job.

Possible inputs:

- An array of file paths. In GitLab 13.6 and later, file paths can include variables.

Example of `rules:changes:paths`:

```
docker-build-1:
  script: docker build -t my-image:$CI_COMMIT_REF_SLUG .
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
      changes:
        - Dockerfile

docker-build-2:
  script: docker build -t my-image:$CI_COMMIT_REF_SLUG .
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
      changes:
        paths:
          - Dockerfile
```

In this example, both jobs have the same behavior.

`rules:changes:compare_to`

Introduced in GitLab 15.3 with a flag named `ci_rules_changes_compare`. Enabled by default.

Use `rules:changes:compare_to` to specify which ref to compare against for changes to the files listed under `rules:changes:paths`.

Keyword type: Job keyword. You can use it only as part of a job, and it must be combined with `rules:changes:paths`.

Possible inputs:

- A branch name, like `main`, `branch1`, or `refs/heads/branch1`.
- A tag name, like `tag1` or `refs/tags/tag1`.
- A commit SHA, like `2fg31ga14b`.

Example of `rules:changes:compare_to`:

```
docker build:
  script: docker build -t my-image:$CI_COMMIT_REF_SLUG .
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
      changes:
        paths:
          - Dockerfile
        compare_to: 'refs/heads/branch1'
```

In this example, the `docker build` job is only included when the `Dockerfile` has changed relative to `refs/heads/branch1` and the pipeline source is a merge request event.

`rules:exists`

Introduced in GitLab 12.4.

Use `exists` to run a job when certain files exist in the repository.

Keyword type: Job keyword. You can use it only as part of a job.

Possible inputs:

- An array of file paths. Paths are relative to the project directory (`$CI_PROJECT_DIR`) and can't directly link outside it. File paths can use glob patterns.

Example of `rules:exists`:

```
job:
  script: docker build -t my-image:$CI_COMMIT_REF_SLUG .
  rules:
    - exists:
        - Dockerfile
```

`job` runs if a `Dockerfile` exists anywhere in the repository.

Additional details:

- Glob patterns are interpreted with Ruby `File.fnmatch` with the flags `File::FNM_PATHNAME | File::FNM_DOTMATCH | File::FNM_EXTGLOB`.
- For performance reasons, GitLab matches a maximum of 10,000 `exists` patterns or file paths. After the 10,000th check, rules with patterned globs always match. In other words, `exists` always reports `true` if more than 10,000 checks run. Repositories with less than 10,000 files might still be impacted if the `exists` rules are checked more than 10,000 times.
- `exists` resolves to `true` if any of the listed files are found (an `OR` operation).

`rules:allow_failure`

Introduced in GitLab 12.8.

Use `allow_failure: true` in `rules` to allow a job to fail without stopping the pipeline.

You can also use `allow_failure: true` with a manual job. The pipeline continues running without waiting for the result of the manual job. `allow_failure: false` combined with `when: manual` in rules causes the pipeline to wait for the manual job to run before continuing.

Keyword type: Job keyword. You can use it only as part of a job.

Possible inputs:

- `true` or `false`. Defaults to `false` if not defined.

Example of `rules:allow_failure`:

```
job:
  script: echo "Hello, Rules!"
  rules:
    - if: $CI_MERGE_REQUEST_TARGET_BRANCH_NAME == $CI_DEFAULT_BRANCH
      when: manual
      allow_failure: true
```

If the rule matches, then the job is a manual job with `allow_failure: true`.

Additional details:

- The rule-level `rules:allow_failure` overrides the job-level `allow_failure`, and only applies when the specific rule triggers the job.

`rules:variables`

Version history

Use `variables` in `rules` to define variables for specific conditions.

Keyword type: Job-specific. You can use it only as part of a job.

Possible inputs:

- A hash of variables in the format `VARIABLE-NAME: value`.

Example of `rules:variables`:

```
job:
  variables:
    DEPLOY_VARIABLE: "default-deploy"
  rules:
    - if: $CI_COMMIT_REF_NAME == $CI_DEFAULT_BRANCH
      variables:                              # Override
DEPLOY_VARIABLE defined
        DEPLOY_VARIABLE: "deploy-production"  # at the job level.
    - if: $CI_COMMIT_REF_NAME =~ /feature/
      variables:
        IS_A_FEATURE: "true"                  # Define a new variable.
  script:
    - echo "Run script with $DEPLOY_VARIABLE as an argument"
    - echo "Run another script if $IS_A_FEATURE exists"
```

`script`

Use `script` to specify commands for the runner to execute.

All jobs except trigger jobs require a `script` keyword.

Keyword type: Job keyword. You can use it only as part of a job.

Possible inputs: An array including:

- Single line commands.
- Long commands split over multiple lines.
- YAML anchors.

CI/CD variables are supported.

Example of `script`:

```
job1:
  script: "bundle exec rspec"

job2:
  script:
    - uname -a
    - bundle exec rspec
```

Additional details:

- When you use these special characters in `script`, you must use single quotes (`'`) or double quotes (`"`) .

Related topics:

- You can ignore non-zero exit codes.
- Use color codes with `script` to make job logs easier to review.
- Create custom collapsible sections to simplify job log output.

`secrets` premium

Introduced in GitLab 13.4.

Use `secrets` to specify CI/CD secrets to:

- Retrieve from an external secrets provider.
- Make available in the job as CI/CD variables (`file` type by default).

This keyword must be used with `secrets:vault`.

`secrets:vault`

Introduced in GitLab 13.4 and GitLab Runner 13.4.

Use `secrets:vault` to specify secrets provided by a HashiCorp Vault.

Keyword type: Job keyword. You can use it only as part of a job.

Possible inputs:

- `engine:name`: Name of the secrets engine.
- `engine:path`: Path to the secrets engine.
- `path`: Path to the secret.
- `field`: Name of the field where the password is stored.

Example of `secrets:vault`:

To specify all details explicitly and use the KV-V2 secrets engine:

```
job:
  secrets:
    DATABASE_PASSWORD:  # Store the path to the secret in this CI/CD
variable
      vault:  # Translates to secret: `ops/data/production/db`, field:
`password`
        engine:
          name: kv-v2
          path: ops
        path: production/db
        field: password
```

You can shorten this syntax. With the short syntax, `engine:name` and `engine:path` both default to `kv-v2`:

```
job:
  secrets:
    DATABASE_PASSWORD:  # Store the path to the secret in this CI/CD
variable
      vault: production/db/password  # Translates to secret: `kv-v2/data
/production/db`, field: `password`
```

To specify a custom secrets engine path in the short syntax, add a suffix that starts with `@`:

```
job:
  secrets:
    DATABASE_PASSWORD:  # Store the path to the secret in this CI/CD
variable
      vault: production/db/password@ops  # Translates to secret: `ops
/data/production/db`, field: `password`
```

`secrets:file`

Introduced in GitLab 14.1 and GitLab Runner 14.1.

Use `secrets:file` to configure the secret to be stored as either a `file` or `variable` type CI/CD variable

By default, the secret is passed to the job as a `file` type CI/CD variable. The value of the secret is stored in the file and the variable contains the path to the file.

If your software can't use `file` type CI/CD variables, set `file: false` to store the secret value directly in the variable.

Keyword type: Job keyword. You can use it only as part of a job.

Possible inputs:

- `true` (default) or `false`.

Example of `secrets:file`:

```
job:
  secrets:
    DATABASE_PASSWORD:
      vault: production/db/password@ops
      file: false
```

Additional details:

- The `file` keyword is a setting for the CI/CD variable and must be nested under the CI/CD variable name, not in the `vault` section.

`services`

Use `services` to specify an additional Docker image to run scripts in. The `services` image is linked to the image specified in the `image` keyword.

Keyword type: Job keyword. You can use it only as part of a job or in the `default` section.

Possible inputs: The name of the services image, including the registry path if needed, in one of these formats:

- `<image-name>` (Same as using `<image-name>` with the `latest` tag)
- `<image-name>:<tag>`
- `<image-name>@<digest>`

CI/CD variables are supported, but not for `alias`.

Example of `services`:

```
default:
  image:
    name: ruby:2.6
    entrypoint: ["/bin/bash"]

  services:
    - name: my-postgres:11.7
      alias: db-postgres
      entrypoint: ["/usr/local/bin/db-postgres"]
      command: ["start"]

  before_script:
    - bundle install

test:
  script:
    - bundle exec rake spec
```

In this example, the job launches a Ruby container. Then, from that container, the job launches another container that's running PostgreSQL. Then the job then runs scripts in that container.

Related topics:

- Available settings for `services`.
- Define `services` in the `.gitlab-ci.yml` file.
- Run your CI/CD jobs in Docker containers.
- Use Docker to build Docker images.

`service:pull_policy`

Version history

On self-managed GitLab, by default this feature is available. To hide the feature, ask an administrator to disable the feature flag named `ci_docker_image_pull_policy`.

The pull policy that the runner uses to fetch the Docker image.

Keyword type: Job keyword. You can use it only as part of a job or in the `default` section.

Possible inputs:

- A single pull policy, or multiple pull policies in an array. Can be `always`, `if-not-present`, or `never`.

Examples of `service:pull_policy`:

```
job1:
  script: echo "A single pull policy."
  services:
    - name: postgres:11.6
      pull_policy: if-not-present

job2:
  script: echo "Multiple pull policies."
  services:
    - name: postgres:11.6
      pull_policy: [always, if-not-present]
```

Additional details:

- If the runner does not support the defined pull policy, the job fails with an error similar to: `ERROR: Job failed (system failure): the configured PullPolicies ([always]) are not allowed by AllowedPullPolicies ([never])`.

Related topics:

- Run your CI/CD jobs in Docker containers.
- How runner pull policies work.
- Using multiple pull policies.

`stage`

Use `stage` to define which stage a job runs in. Jobs in the same `stage` can execute in parallel (see **Additional details**).

If `stage` is not defined, the job uses the `test` stage by default.

Keyword type: Job keyword. You can use it only as part of a job.

Possible inputs: An array including any number of stage names. Stage names can be:

- The default stages.
- User-defined stages.

Example of `stage`:

```
stages:
  - build
  - test
  - deploy

job1:
  stage: build
  script:
    - echo "This job compiles code."

job2:
  stage: test
  script:
    - echo "This job tests the compiled code. It runs when the build
stage completes."

job3:
  script:
    - echo "This job also runs in the test stage".

job4:
  stage: deploy
  script:
    - echo "This job deploys the code. It runs when the test stage
completes."
```

Additional details:

- Jobs can run in parallel if they run on different runners.
- If you have only one runner, jobs can run in parallel if the runner's `concurrent` setting is greater than `1`.

`stage: .pre`

Introduced in GitLab 12.4.

Use the `.pre` stage to make a job run at the start of a pipeline. `.pre` is always the first stage in a pipeline. User-defined stages execute after `.pre`. You do not have to define `.pre` in `stages`.

If a pipeline contains only jobs in the `.pre` or `.post` stages, it does not run. There must be at least one other job in a different stage.

Keyword type: You can only use it with a job's `stage` keyword.

Example of `stage: .pre`:

```
stages:
  - build
  - test

job1:
  stage: build
  script:
    - echo "This job runs in the build stage."

first-job:
  stage: .pre
  script:
    - echo "This job runs in the .pre stage, before all other stages."

job2:
  stage: test
  script:
    - echo "This job runs in the test stage."
```

`stage: .post`

Introduced in GitLab 12.4.

Use the `.post` stage to make a job run at the end of a pipeline. `.post` is always the last stage in a pipeline. User-defined stages execute before `.post`. You do not have to define `.post` in `stages`.

If a pipeline contains only jobs in the `.pre` or `.post` stages, it does not run. There must be at least one other job in a different stage.

Keyword type: You can only use it with a job's `stage` keyword.

Example of `stage: .post`:

```
stages:
  - build
  - test

job1:
  stage: build
  script:
    - echo "This job runs in the build stage."

last-job:
  stage: .post
  script:
    - echo "This job runs in the .post stage, after all other stages."

job2:
  stage: test
  script:
    - echo "This job runs in the test stage."
```

`tags`

Version history

Use `tags` to select a specific runner from the list of all runners that are available for the project.

When you register a runner, you can specify the runner's tags, for example `ruby`, `postgres`, or `development`. To pick up and run a job, a runner must be assigned every tag listed in the job.

Keyword type: Job keyword. You can use it only as part of a job or in the `default` section.

Possible inputs:

- An array of tag names.
- CI/CD variables are supported in GitLab 14.1 and later.

Example of `tags`:

```
job:
  tags:
    - ruby
    - postgres
```

In this example, only runners with *both* the `ruby` and `postgres` tags can run the job.

Additional details:

- In GitLab 14.3 and later, the number of tags must be less than `50`.

Related topics:

- Use tags to control which jobs a runner can run.
- Select different runner tags for each parallel matrix job.

`timeout`

Introduced in GitLab 12.3.

Use `timeout` to configure a timeout for a specific job. If the job runs for longer than the timeout, the job fails.

The job-level timeout can be longer than the project-level timeout. but can't be longer than the runner's timeout.

Keyword type: Job keyword. You can use it only as part of a job or in the `default` section.

Possible inputs: A period of time written in natural language. For example, these are all equivalent:

- `3600 seconds`
- `60 minutes`
- `one hour`

Example of `timeout`:

```
build:
  script: build.sh
  timeout: 3 hours 30 minutes

test:
  script: rspec
  timeout: 3h 30m
```

`trigger`

Version history

Use `trigger` to start a downstream pipeline that is either:

- A multi-project pipeline.
- A child pipeline.

Keyword type: Job keyword. You can use it only as part of a job.

Possible inputs:

- For multi-project pipelines, path to the downstream project. CI/CD variables are supported in GitLab 15.3 and later.
- For child pipelines, path to the child pipeline CI/CD configuration file.

Example of `trigger` for multi-project pipeline:

```
rspec:
  stage: test
  script: bundle exec rspec

staging:
  stage: deploy
  trigger: my/deployment
```

Example of `trigger` for child pipelines:

```
trigger_job:
  trigger:
    include: path/to/child-pipeline.yml
```

Additional details:

- Jobs with `trigger` can only use a limited set of keywords. For example, you can't run commands with `script`, `before_script`, or `after_script`. Also, `environment` is not supported with `trigger`.
- You cannot use the API to start `when:manual` trigger jobs.
- In GitLab 13.5 and later, you can use `when:manual` in the same job as `trigger`. In GitLab 13.4 and earlier, using them together causes the error `jobs:#{job-name} when should be on_success, on_failure or always`.
- In GitLab 13.2 and later, you can view which job triggered a downstream pipeline in the pipeline graph.
- Manual pipeline variables and scheduled pipeline variables are not passed to downstream pipelines by default. Use trigger:forward to forward these variables to downstream pipelines.
- Job-level persisted variables are not available in trigger jobs.

Related topics:

- Multi-project pipeline configuration examples.
- Child pipeline configuration examples.
- To run a pipeline for a specific branch, tag, or commit, you can use a trigger token to authenticate with the pipeline triggers API. The trigger token is different than the `trigger` keyword.

`trigger:strategy`

Use `trigger:strategy` to force the `trigger` job to wait for the downstream pipeline to complete before it is marked as **success**.

This behavior is different than the default, which is for the `trigger` job to be marked as **success** as soon as the downstream pipeline is created.

This setting makes your pipeline execution linear rather than parallel.

Example of `trigger:strategy`:

```
trigger_job:
  trigger:
    include: path/to/child-pipeline.yml
    strategy: depend
```

In this example, jobs from subsequent stages wait for the triggered pipeline to successfully complete before starting.

Additional details:

- Optional manual jobs in the downstream pipeline do not affect the status of the downstream pipeline or the upstream trigger job. The downstream pipeline can complete successfully without running any optional manual jobs.
- Blocking manual jobs in the downstream pipeline must run before the trigger job is marked as successful or failed. The trigger job shows **pending** () if the downstream pipeline status is **waiting for manual action** () due to manual jobs. By default, jobs in later stages do not start until the trigger job completes.

`trigger:forward`

Version history

Use `trigger:forward` to specify what to forward to the downstream pipeline. You can control what is forwarded to both parent-child pipelines and multi-project pipelines.

Possible inputs:

- `yaml_variables`: `true` (default), or `false`. When `true`, variables defined in the trigger job are passed to downstream pipelines.
- `pipeline_variables`: `true` or `false` (default). When `true`, manual pipeline variables and scheduled pipeline variables are passed to downstream pipelines.

Example of `trigger:forward`:

Run this pipeline manually, with the CI/CD variable `MYVAR = my value`:

```
variables: # default variables for each job
  VAR: value

# Default behavior:
# - VAR is passed to the child
# - MYVAR is not passed to the child
child1:
  trigger:
    include: .child-pipeline.yml

# Forward pipeline variables:
# - VAR is passed to the child
# - MYVAR is passed to the child
child2:
  trigger:
    include: .child-pipeline.yml
    forward:
      pipeline_variables: true

# Do not forward YAML variables:
# - VAR is not passed to the child
# - MYVAR is not passed to the child
child3:
  trigger:
    include: .child-pipeline.yml
    forward:
      yaml_variables: false
```

`variables`

CI/CD variables are configurable values that are passed to jobs. Use `variables` to create custom variables.

Variables are always available in `script`, `before_script`, and `after_script` commands. You can also use variables as inputs in some job keywords.

Keyword type: Global and job keyword. You can use it at the global level, and also at the job level.

If you define `variables` at the global level, each variable is copied to every job configuration when the pipeline is created. If the job already has that variable defined, the job-level variable takes precedence.

Possible inputs: Variable name and value pairs:

- The name can use only numbers, letters, and underscores (_). In some shells, the first character must be a letter.
- The value must be a string.

CI/CD variables are supported.

Examples of `variables`:

```
variables:
  DEPLOY_SITE: "https://example.com/"

deploy_job:
  stage: deploy
  script:
    - deploy-script --url $DEPLOY_SITE --path "/"

deploy_review_job:
  stage: deploy
  variables:
    REVIEW_PATH: "/review"
  script:
    - deploy-review-script --url $DEPLOY_SITE --path $REVIEW_PATH
```

Additional details:

- All YAML-defined variables are also set to any linked Docker service containers.
- YAML-defined variables are meant for non-sensitive project configuration. Store sensitive information in protected variables or CI/CD secrets.
- Manual pipeline variables and scheduled pipeline variables are not passed to downstream pipelines by default. Use trigger:forward to forward these variables to downstream pipelines.

Related topics:

- You can use YAML anchors for variables.
- Predefined variables are variables the runner automatically creates and makes available in the job.
- You can configure runner behavior with variables.

`variables:description`

Introduced in GitLab 13.7.

Use the `description` keyword to define a pipeline-level (global) variable that is prefilled when running a pipeline manually.

Must be used with `value`, for the variable value.

Keyword type: Global keyword. You cannot set job-level variables to be pre-filled when you run a pipeline manually.

Possible inputs:

- A string.

Example of `variables:description`:

```
variables:
  DEPLOY_ENVIRONMENT:
    value: "staging"
    description: "The deployment target. Change this variable to
'canary' or 'production' if needed."
```

`when`

Use `when` to configure the conditions for when jobs run. If not defined in a job, the default value is `when: on_success`.

Keyword type: Job keyword. You can use it as part of a job. `when: always` and `when: never` can also be used in `workflow:rules`.

Possible inputs:

- `on_success` (default): Run the job only when all jobs in earlier stages succeed or have `allow_failure: true`.
- `manual`: Run the job only when triggered manually.
- `always`: Run the job regardless of the status of jobs in earlier stages. Can also be used in `workflow:rules`.
- `on_failure`: Run the job only when at least one job in an earlier stage fails.
- `delayed`: Delay the execution of a job for a specified duration.
- `never`: Don't run the job. Can only be used in a `rules` section or `workflow: rules`.

Example of `when`:

```
stages:
  - build
  - cleanup_build
  - test
  - deploy
  - cleanup

build_job:
  stage: build
  script:
    - make build

cleanup_build_job:
  stage: cleanup_build
  script:
    - cleanup build when failed
  when: on_failure

test_job:
  stage: test
  script:
    - make test

deploy_job:
  stage: deploy
  script:
    - make deploy
  when: manual

cleanup_job:
  stage: cleanup
  script:
    - cleanup after jobs
  when: always
```

In this example, the script:

1. Executes `cleanup_build_job` only when `build_job` fails.
2. Always executes `cleanup_job` as the last step in pipeline regardless of success or failure.
3. Executes `deploy_job` when you run it manually in the GitLab UI.

Additional details:

- In GitLab 13.5 and later, you can use `when:manual` in the same job as `trigger`. In GitLab 13.4 and earlier, using them together causes the error `jobs:#{job-name} when should be on_success, on_failure or always`.
- The default behavior of `allow_failure` changes to `true` with `when: manual`. However, if you use `when: manual` with `rules`, `allow_failure` defaults to `false`.

Related topics:

- `when` can be used with `rules` for more dynamic job control.
- `when` can be used with `workflow` to control when a pipeline can start.

Deprecated keywords

The following keywords are deprecated.

Globally-defined `types` (removed)

The `types` keyword was deprecated in GitLab 9.0, and removed in GitLab 15.0. Use `stages` instead.

Job-defined `type` (removed)

The `type` keyword was deprecated in GitLab 9.0, and removed in GitLab 15.0. Use `stage` instead.

Globally-defined `image`, `services`, `cache`, `before_script`, `after_script`

Defining `image`, `services`, `cache`, `before_script`, and `after_script` globally is deprecated. Support could be removed from a future release.

Use `default` instead. For example:

```
default:
  image: ruby:3.0
  services:
    - docker:dind
  cache:
    paths: [vendor/]
  before_script:
    - bundle config set path vendor/bundle
    - bundle install
  after_script:
    - rm -rf tmp/
```