`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 CI/CD variables or predefined CI/CD variables, with some exceptions.

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.
- CI/CD 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 including any number of:

- Paths to files. In GitLab 13.6 and later, file paths can include variables. A file path array can also be in `rules:changes:paths`.
- Wildcard paths for:
  - Single directories, for example `path/to/directory/*`.
  - 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 `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 (`allow_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. 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`

Version history

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`

Version history

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 and CI/CD variables.

  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"
```