Workflow syntax for GitHub Actions

In this article

jobs.<job_id>.steps[*].uses

jobs.<job_id>.steps[*].run

jobs.<job_id>.steps[*].working-directory

jobs.<job_id>.steps[*].shell

jobs.<job_id>.steps[*].with

jobs.<job_id>.steps[*].with.args

jobs.<job_id>.steps[*].with.entrypoint

jobs.<job_id>.steps[*].env

jobs.<job_id>.steps[*].continue-on-error

jobs.<job_id>.steps[*].timeout-minutes

jobs.<job_id>.timeout-minutes

jobs.<job_id>.strategy

jobs.<job_id>.strategy.matrix

jobs.<job_id>.strategy.matrix.include

jobs.<job_id>.strategy.matrix.exclude

jobs.<job_id>.strategy.fail-fast

jobs.<job_id>.strategy.max-parallel

jobs.<job_id>.continue-on-error

jobs.<job_id>.container

jobs.<job_id>.container.image

jobs.<job_id>.container.credentials

jobs.<job_id>.container.env

jobs.<job_id>.container.ports

jobs.<job_id>.container.volumes

jobs.<job_id>.container.options

jobs.<job_id>.services

Example: Using localhost

jobs.<job_id>.services.<service_id>.image

jobs.<job_id>.services.<service_id>.credentials

jobs.<job_id>.services.<service_id>.env

jobs.<job_id>.services.<service_id>.ports

jobs.<job_id>.services.<service_id>.volumes

jobs.<job_id>.services.<service_id>.options

jobs.<job_id>.uses

jobs.<job_id>.with

jobs.<job_id>.with.<input_id>

jobs.<job_id>.secrets

jobs.<job_id>.secrets.inherit

jobs.<job_id>.secrets.<secret_id>

Filter pattern cheat sheet

A workflow is a configurable automated process made up of one or more jobs. You must create a YAML file to define your workflow configuration.

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

About YAML syntax for workflows

Workflow files use YAML syntax, and must have either a `.yml` or `.yaml` file extension. If you're new to YAML and want to learn more, see "Learn YAML in Y minutes."

You must store workflow files in the `.github/workflows` directory of your repository.

`name`

The name of the workflow. GitHub displays the names of your workflows under your repository's "Actions" tab. If you omit `name`, GitHub displays the workflow file path relative to the root of the repository.

`run-name`

The name for workflow runs generated from the workflow. GitHub displays the workflow run name in the list of workflow runs on your repository's "Actions" tab. If `run-name` is omitted or is only whitespace, then the run name is set to event-specific information for the workflow run. For example, for a workflow triggered by a `push` or `pull_request` event, it is set as the commit message.

This value can include expressions and can reference the `github` and `inputs` contexts.

Example of `run-name`

```yaml
run-name: Deploy to ${{ inputs.deploy_target }} by @${{ github.actor }}
```

`on`

To automatically trigger a workflow, use `on` to define which events can cause the workflow to run. For a list of available events, see "Events that trigger workflows."

You can define single or multiple events that can trigger a workflow, or set a time schedule. You can also restrict the execution of a workflow to only occur for specific files, tags, or branch changes. These options are described in the following sections.

Using a single event

For example, a workflow with the following `on` value will run when a push is made to any branch in the workflow's repository:

```yaml
on: push
```

Using multiple events

You can specify a single event or multiple events. For example, a workflow with the following `on` value will run when a push is made to any branch in the repository or when someone forks the repository:

```yaml
on: [push, fork]
```

If you specify multiple events, only one of those events needs to occur to trigger your workflow. If multiple triggering events for your workflow occur at the same time, multiple

workflow runs will be triggered.

Using activity types

Some events have activity types that give you more control over when your workflow should run. Use `on.<event_name>.types` to define the type of event activity that will trigger a workflow run.

For example, the `issue_comment` event has the `created`, `edited`, and `deleted` activity types. If your workflow triggers on the `label` event, it will run whenever a label is created, edited, or deleted. If you specify the `created` activity type for the `label` event, your workflow will run when a label is created but not when a label is edited or deleted.

```yaml
on:
  label:
    types:
      - created
```

If you specify multiple activity types, only one of those event activity types needs to occur to trigger your workflow. If multiple triggering event activity types for your workflow occur at the same time, multiple workflow runs will be triggered. For example, the following workflow triggers when an issue is opened or labeled. If an issue with two labels is opened, three workflow runs will start: one for the issue opened event and two for the two issue labeled events.

```yaml
on:
  issues:
    types:
      - opened
      - labeled
```

For more information about each event and their activity types, see "Events that trigger workflows."

Using filters

Some events have filters that give you more control over when your workflow should run.

For example, the `push` event has a `branches` filter that causes your workflow to run only when a push to a branch that matches the `branches` filter occurs, instead of when any push occurs.

```yaml
on:
  push:
    branches:
      - main
      - 'releases/**'
```

Using activity types and filters with multiple events

If you specify activity types or filters for an event and your workflow triggers on multiple events, you must configure each event separately. You must append a colon (`:`) to all events, including events without configuration.

For example, a workflow with the following `on` value will run when:

- A label is created
- A push is made to the `main` branch in the repository
- A push is made to a GitHub Pages-enabled branch

```yaml
on:
  label:
    types:
      - created
  push:
    branches:
      - main
  page_build:
```

on.<event_name>.types

Use `on.<event_name>.types` to define the type of activity that will trigger a workflow run. Most GitHub events are triggered by more than one type of activity. For example, the `label` is triggered when a label is `created`, `edited`, or `deleted`. The `types` keyword enables you to narrow down activity that causes the workflow to run. When only one activity type triggers a webhook event, the `types` keyword is unnecessary.

You can use an array of event `types`. For more information about each event and their activity types, see "Events that trigger workflows."

```yaml
on:
  label:
    types: [created, edited]
```

on.<pull_request|pull_request_target>.<branches|branches-ignore>

When using the `pull_request` and `pull_request_target` events, you can configure a workflow to run only for pull requests that target specific branches.

Use the `branches` filter when you want to include branch name patterns or when you want to both include and exclude branch names patterns. Use the `branches-ignore` filter when you only want to exclude branch name patterns. You cannot use both the `branches` and `branches-ignore` filters for the same event in a workflow.

If you define both `branches`/`branches-ignore` and `paths`/`paths-ignore`, the workflow will only run when both filters are satisfied.

The `branches` and `branches-ignore` keywords accept glob patterns that use characters like `*`, `**`, `+`, `?`, `!` and others to match more than one branch name. If a name contains any of these characters and you want a literal match, you need to escape each of these special characters with `\`. For more information about glob patterns, see the "Workflow syntax for GitHub Actions."

Example: Including branches

The patterns defined in `branches` are evaluated against the Git ref's name. For example, the following workflow would run whenever there is a `pull_request` event for a pull request targeting:

- A branch named `main` (`refs/heads/main`)
- A branch named `mona/octocat` (`refs/heads/mona/octocat`)
- A branch whose name starts with `releases/`, like `releases/10` (`refs/heads/releases/10`)

```yaml
on:
  pull_request:
```

```
    # Sequence of patterns matched against refs/heads
    branches:
      - main
      - 'mona/octocat'
      - 'releases/**'
```

Note: You should not use branch filtering to skip workflow runs if the workflow has been configured to be required. For more information, see "Required workflows."

If a workflow is skipped due to branch filtering, path filtering, or a commit message, then checks associated with that workflow will remain in a "Pending" state. A pull request that requires those checks to be successful will be blocked from merging.

Example: Excluding branches

When a pattern matches the `branches-ignore` pattern, the workflow will not run. The patterns defined in `branches-ignore` are evaluated against the Git ref's name. For example, the following workflow would run whenever there is a `pull_request` event unless the pull request is targeting:

- A branch named `mona/octocat` (`refs/heads/mona/octocat`)
- A branch whose name matches `releases/**-alpha`, like `releases/beta/3-alpha` (`refs/heads/releases/beta/3-alpha`)

```
on:
  pull_request:
    # Sequence of patterns matched against refs/heads
    branches-ignore:
      - 'mona/octocat'
      - 'releases/**-alpha'
```

Example: Including and excluding branches

You cannot use `branches` and `branches-ignore` to filter the same event in a single workflow. If you want to both include and exclude branch patterns for a single event, use the `branches` filter along with the `!` character to indicate which branches should be excluded.

If you define a branch with the `!` character, you must also define at least one branch without the `!` character. If you only want to exclude branches, use `branches-ignore` instead.

The order that you define patterns matters.

- A matching negative pattern (prefixed with `!`) after a positive match will exclude the Git ref.
- A matching positive pattern after a negative match will include the Git ref again.

The following workflow will run on `pull_request` events for pull requests that target `releases/10` or `releases/beta/mona`, but not for pull requests that target `releases/10-alpha` or `releases/beta/3-alpha` because the negative pattern `!releases/**-alpha` follows the positive pattern.

```
on:
  pull_request:
    branches:
      - 'releases/**'
      - '!releases/**-alpha'
```

`on.push.<branches|tags|branches-ignore|tags-ignore>`

When using the `push` event, you can configure a workflow to run on specific branches or tags.

Use the `branches` filter when you want to include branch name patterns or when you want to both include and exclude branch names patterns. Use the `branches-ignore` filter when you only want to exclude branch name patterns. You cannot use both the `branches` and `branches-ignore` filters for the same event in a workflow.

Use the `tags` filter when you want to include tag name patterns or when you want to both include and exclude tag names patterns. Use the `tags-ignore` filter when you only want to exclude tag name patterns. You cannot use both the `tags` and `tags-ignore` filters for the same event in a workflow.

If you define only `tags` / `tags-ignore` or only `branches` / `branches-ignore`, the workflow won't run for events affecting the undefined Git ref. If you define neither `tags` / `tags-ignore` or `branches` / `branches-ignore`, the workflow will run for events affecting either branches or tags. If you define both `branches` / `branches-ignore` and `paths` / `paths-ignore`, the workflow will only run when both filters are satisfied.

The `branches`, `branches-ignore`, `tags`, and `tags-ignore` keywords accept glob patterns that use characters like `*`, `**`, `+`, `?`, `!` and others to match more than one branch or tag name. If a name contains any of these characters and you want a literal match, you need to *escape* each of these special characters with `\`. For more information about glob patterns, see the "Workflow syntax for GitHub Actions."

Example: Including branches and tags

The patterns defined in `branches` and `tags` are evaluated against the Git ref's name. For example, the following workflow would run whenever there is a `push` event to:

- A branch named `main` (`refs/heads/main`)
- A branch named `mona/octocat` (`refs/heads/mona/octocat`)
- A branch whose name starts with `releases/`, like `releases/10` (`refs/heads/releases/10`)
- A tag named `v2` (`refs/tags/v2`)
- A tag whose name starts with `v1.`, like `v1.9.1` (`refs/tags/v1.9.1`)

```yaml
on:
  push:
    # Sequence of patterns matched against refs/heads
    branches:
      - main
      - 'mona/octocat'
      - 'releases/**'
    # Sequence of patterns matched against refs/tags
    tags:
      - v2
      - v1.*
```

Example: Excluding branches and tags

When a pattern matches the `branches-ignore` or `tags-ignore` pattern, the workflow will not run. The patterns defined in `branches` and `tags` are evaluated against the Git ref's name. For example, the following workflow would run whenever there is a `push` event, unless the `push` event is to:

- A branch named `mona/octocat` (`refs/heads/mona/octocat`)
- A branch whose name matches `releases/**-alpha` , like `releases/beta/3-alpha` (`refs/heads/releases/beta/3-alpha`)
- A tag named `v2` (`refs/tags/v2`)
- A tag whose name starts with `v1.` , like `v1.9` (`refs/tags/v1.9`)

```yaml
on:
  push:
    # Sequence of patterns matched against refs/heads
    branches-ignore:
      - 'mona/octocat'
      - 'releases/**-alpha'
    # Sequence of patterns matched against refs/tags
    tags-ignore:
      - v2
      - v1.*
```

Example: Including and excluding branches and tags

You can't use `branches` and `branches-ignore` to filter the same event in a single workflow. Similarly, you can't use `tags` and `tags-ignore` to filter the same event in a single workflow. If you want to both include and exclude branch or tag patterns for a single event, use the `branches` or `tags` filter along with the `!` character to indicate which branches or tags should be excluded.

If you define a branch with the `!` character, you must also define at least one branch without the `!` character. If you only want to exclude branches, use `branches-ignore` instead. Similarly, if you define a tag with the `!` character, you must also define at least one tag without the `!` character. If you only want to exclude tags, use `tags-ignore` instead.

The order that you define patterns matters.

- A matching negative pattern (prefixed with `!`) after a positive match will exclude the Git ref.
- A matching positive pattern after a negative match will include the Git ref again.

The following workflow will run on pushes to `releases/10` or `releases/beta/mona` , but not on `releases/10-alpha` or `releases/beta/3-alpha` because the negative pattern `!releases/**-alpha` follows the positive pattern.

```yaml
on:
  push:
    branches:
      - 'releases/**'
      - '!releases/**-alpha'
```

`on.<push|pull_request|pull_request_target>.<paths|paths-ignore>`

When using the `push` and `pull_request` events, you can configure a workflow to run based on what file paths are changed. Path filters are not evaluated for pushes of tags.

Use the `paths` filter when you want to include file path patterns or when you want to both include and exclude file path patterns. Use the `paths-ignore` filter when you only want to exclude file path patterns. You cannot use both the `paths` and `paths-ignore` filters for the same event in a workflow. If you want to both include and exclude path patterns for a single event, use the `paths` filter prefixed with the `!` character to indicate which paths should be excluded.

Note: The order that you define `paths` patterns matters:

- A matching negative pattern (prefixed with `!`) after a positive match will exclude the path.
- A matching positive pattern after a negative match will include the path again.

If you define both `branches` / `branches-ignore` and `paths` / `paths-ignore`, the workflow will only run when both filters are satisfied.

The `paths` and `paths-ignore` keywords accept glob patterns that use the `*` and `**` wildcard characters to match more than one path name. For more information, see the "Workflow syntax for GitHub Actions."

Example: Including paths

If at least one path matches a pattern in the `paths` filter, the workflow runs. For example, the following workflow would run anytime you push a JavaScript file (`.js`).

```yaml
on:
  push:
    paths:
      - '**.js'
```

Note: You should not use path filtering to skip workflow runs if the workflow has been configured to be required. For more information, see "Required workflows."

If a workflow is skipped due to path filtering, branch filtering, or a commit message, then checks associated with that workflow will remain in a "Pending" state. A pull request that requires those checks to be successful will be blocked from merging.

Example: Excluding paths

When all the path names match patterns in `paths-ignore`, the workflow will not run. If any path names do not match patterns in `paths-ignore`, even if some path names match the patterns, the workflow will run.

A workflow with the following path filter will only run on `push` events that include at least one file outside the `docs` directory at the root of the repository.

```yaml
on:
  push:
    paths-ignore:
      - 'docs/**'
```

Example: Including and excluding paths

You can not use `paths` and `paths-ignore` to filter the same event in a single workflow. If you want to both include and exclude path patterns for a single event, use the `paths` filter prefixed with the `!` character to indicate which paths should be excluded.

If you define a path with the `!` character, you must also define at least one path without the `!` character. If you only want to exclude paths, use `paths-ignore` instead.

The order that you define `paths` patterns matters:

- A matching negative pattern (prefixed with `!`) after a positive match will exclude the path.
- A matching positive pattern after a negative match will include the path again.

This example runs anytime the `push` event includes a file in the `sub-project` directory

or its subdirectories, unless the file is in the `sub-project/docs` directory. For example, a push that changed `sub-project/index.js` or `sub-project/src/index.js` will trigger a workflow run, but a push changing only `sub-project/docs/readme.md` will not.

```yaml
on:
  push:
    paths:
      - 'sub-project/**'
      - '!sub-project/docs/**'
```

Git diff comparisons

> **Note:** If you push more than 1,000 commits, or if GitHub does not generate the diff due to a timeout, the workflow will always run.

The filter determines if a workflow should run by evaluating the changed files and running them against the `paths-ignore` or `paths` list. If there are no files changed, the workflow will not run.

GitHub generates the list of changed files using two-dot diffs for pushes and three-dot diffs for pull requests:

- **Pull requests:** Three-dot diffs are a comparison between the most recent version of the topic branch and the commit where the topic branch was last synced with the base branch.
- **Pushes to existing branches:** A two-dot diff compares the head and base SHAs directly with each other.
- **Pushes to new branches:** A two-dot diff against the parent of the ancestor of the deepest commit pushed.

Diffs are limited to 300 files. If there are files changed that aren't matched in the first 300 files returned by the filter, the workflow will not run. You may need to create more specific filters so that the workflow will run automatically.

For more information, see "About comparing branches in pull requests."

`on.schedule`

You can use `on.schedule` to define a time schedule for your workflows. You can schedule a workflow to run at specific UTC times using POSIX cron syntax. Scheduled workflows run on the latest commit on the default or base branch. The shortest interval you can run scheduled workflows is once every 5 minutes.

This example triggers the workflow every day at 5:30 and 17:30 UTC:

```yaml
on:
  schedule:
    # * is a special character in YAML so you have to quote this string
    - cron:  '30 5,17 * * *'
```

A single workflow can be triggered by multiple `schedule` events. You can access the schedule event that triggered the workflow through the `github.event.schedule` context. This example triggers the workflow to run at 5:30 UTC every Monday-Thursday, but skips the `Not on Monday or Wednesday` step on Monday and Wednesday.

```yaml
on:
  schedule:
    - cron: '30 5 * * 1,3'
```

```yaml
    - cron: '30 5 * * 2,4'

jobs:
  test_schedule:
    runs-on: ubuntu-latest
    steps:
      - name: Not on Monday or Wednesday
        if: github.event.schedule != '30 5 * * 1,3'
        run: echo "This step will be skipped on Monday and Wednesday"
      - name: Every time
        run: echo "This step will always run"
```

For more information about cron syntax, see "Events that trigger workflows."

`on.workflow_call`

Use `on.workflow_call` to define the inputs and outputs for a reusable workflow. You can also map the secrets that are available to the called workflow. For more information on reusable workflows, see "Reusing workflows."

`on.workflow_call.inputs`

When using the `workflow_call` keyword, you can optionally specify inputs that are passed to the called workflow from the caller workflow. For more information about the `workflow_call` keyword, see "Events that trigger workflows."

In addition to the standard input parameters that are available, `on.workflow_call.inputs` requires a `type` parameter. For more information, see `on.workflow_call.inputs.<input_id>.type`.

If a `default` parameter is not set, the default value of the input is `false` for a boolean, `0` for a number, and `""` for a string.

Within the called workflow, you can use the `inputs` context to refer to an input. For more information, see "Contexts."

If a caller workflow passes an input that is not specified in the called workflow, this results in an error.

Example of `on.workflow_call.inputs`

```yaml
on:
  workflow_call:
    inputs:
      username:
        description: 'A username passed from the caller workflow'
        default: 'john-doe'
        required: false
        type: string

jobs:
  print-username:
    runs-on: ubuntu-latest

    steps:
      - name: Print the input name to STDOUT
        run: echo The username is ${{ inputs.username }}
```

For more information, see "Reusing workflows."

`on.workflow_call.inputs.<input_id>.type`

Required if input is defined for the `on.workflow_call` keyword. The value of this parameter is a string specifying the data type of the input. This must be one of: `boolean`, `number`, or `string`.

`on.workflow_call.outputs`

A map of outputs for a called workflow. Called workflow outputs are available to all downstream jobs in the caller workflow. Each output has an identifier, an optional `description`, and a `value`. The `value` must be set to the value of an output from a job within the called workflow.

In the example below, two outputs are defined for this reusable workflow: `workflow_output1` and `workflow_output2`. These are mapped to outputs called `job_output1` and `job_output2`, both from a job called `my_job`.

Example of `on.workflow_call.outputs`

```yaml
on:
  workflow_call:
    # Map the workflow outputs to job outputs
    outputs:
      workflow_output1:
        description: "The first job output"
        value: ${{ jobs.my_job.outputs.job_output1 }}
      workflow_output2:
        description: "The second job output"
        value: ${{ jobs.my_job.outputs.job_output2 }}
```

For information on how to reference a job output, see `jobs.<job_id>.outputs`. For more information, see "Reusing workflows."

`on.workflow_call.secrets`

A map of the secrets that can be used in the called workflow.

Within the called workflow, you can use the `secrets` context to refer to a secret.

> **Note:** If you are passing the secret to a nested reusable workflow, then you must use `jobs.<job_id>.secrets` again to pass the secret. For more information, see "Reusing workflows."

If a caller workflow passes a secret that is not specified in the called workflow, this results in an error.

Example of `on.workflow_call.secrets`

```yaml
on:
  workflow_call:
    secrets:
      access-token:
        description: 'A token passed from the caller workflow'
        required: false

jobs:

  pass-secret-to-action:
```

```
    runs-on: ubuntu-latest
    steps:
    # passing the secret to an action
      - name: Pass the received secret to an action
        uses: ./.github/actions/my-action
        with:
          token: ${{ secrets.access-token }}

  # passing the secret to a nested reusable workflow
  pass-secret-to-workflow:
    uses: ./.github/workflows/my-workflow
    secrets:
       token: ${{ secrets.access-token }}
```

`on.workflow_call.secrets.<secret_id>`

A string identifier to associate with the secret.

`on.workflow_call.secrets.<secret_id>.required`

A boolean specifying whether the secret must be supplied.

`on.workflow_run.<branches|branches-ignore>`

When using the `workflow_run` event, you can specify what branches the triggering workflow must run on in order to trigger your workflow.

The `branches` and `branches-ignore` filters accept glob patterns that use characters like `*`, `**`, `+`, `?`, `!` and others to match more than one branch name. If a name contains any of these characters and you want a literal match, you need to *escape* each of these special characters with `\`. For more information about glob patterns, see the "Workflow syntax for GitHub Actions."

For example, a workflow with the following trigger will only run when the workflow named `Build` runs on a branch whose name starts with `releases/`:

```
on:
  workflow_run:
    workflows: ["Build"]
    types: [requested]
    branches:
      - 'releases/**'
```

A workflow with the following trigger will only run when the workflow named `Build` runs on a branch that is not named `canary`:

```
on:
  workflow_run:
    workflows: ["Build"]
    types: [requested]
    branches-ignore:
      - "canary"
```

You cannot use both the `branches` and `branches-ignore` filters for the same event in a workflow. If you want to both include and exclude branch patterns for a single event, use the `branches` filter along with the `!` character to indicate which branches should be excluded.

The order that you define patterns matters.

- A matching negative pattern (prefixed with `!`) after a positive match will exclude the branch.
- A matching positive pattern after a negative match will include the branch again.

For example, a workflow with the following trigger will run when the workflow named `Build` runs on a branch that is named `releases/10` or `releases/beta/mona` but will not `releases/10-alpha`, `releases/beta/3-alpha`, or `main`.

```yaml
on:
  workflow_run:
    workflows: ["Build"]
    types: [requested]
    branches:
      - 'releases/**'
      - '!releases/**-alpha'
```

`on.workflow_dispatch`

When using the `workflow_dispatch` event, you can optionally specify inputs that are passed to the workflow.

`on.workflow_dispatch.inputs`

The triggered workflow receives the inputs in the `inputs` context. For more information, see "Contexts."

Notes:

- The workflow will also receive the inputs in the `github.event.inputs` context. The information in the `inputs` context and `github.event.inputs` context is identical except that the `inputs` context preserves Boolean values as Booleans instead of converting them to strings. The `choice` type resolves to a string and is a single selectable option.
- The maximum number of top-level properties for `inputs` is 10.
- The maximum payload for `inputs` is 65,535 characters.

Example of `on.workflow_dispatch.inputs`

```yaml
on:
  workflow_dispatch:
    inputs:
      logLevel:
        description: 'Log level'
        required: true
        default: 'warning'
        type: choice
        options:
          - info
          - warning
          - debug
      print_tags:
        description: 'True to print to STDOUT'
        required: true
        type: boolean
      tags:
        description: 'Test scenario tags'
        required: true
        type: string
      environment:
        description: 'Environment to run tests against'
        type: environment
```

```
        required: true

jobs:
  print-tag:
    runs-on: ubuntu-latest
    if:  ${{ inputs.print_tags }}
    steps:
      - name: Print the input tag to STDOUT
        run: echo  The tags are ${{ inputs.tags }}
```

`on.workflow_dispatch.inputs.<input_id>.required`

A boolean specifying whether the input must be supplied.

`on.workflow_dispatch.inputs.<input_id>.type`

The value of this parameter is a string specifying the data type of the input. This must be one of: `boolean`, `choice`, `number`, or `string`.

`permissions`

You can use `permissions` to modify the default permissions granted to the `GITHUB_TOKEN`, adding or removing access as required, so that you only allow the minimum required access. For more information, see "Automatic token authentication."

You can use `permissions` either as a top-level key, to apply to all jobs in the workflow, or within specific jobs. When you add the `permissions` key within a specific job, all actions and run commands within that job that use the `GITHUB_TOKEN` gain the access rights you specify. For more information, see `jobs.<job_id>.permissions`.

For each of the available scopes, shown in the table below, you can assign one of the permissions: `read`, `write`, or `none`. If you specify the access for any of these scopes, all of those that are not specified are set to `none`.

Available scopes and details of what each allows an action to do:

Scope	Allows an action using `GITHUB_TOKEN` to
`actions`	Work with GitHub Actions. For example, `actions: write` permits an action to cancel a workflow run. For more information, see "Permissions required for GitHub Apps."
`checks`	Work with check runs and check suites. For example, `checks: write` permits an action to create a check run. For more information, see "Permissions required for GitHub Apps."
`contents`	Work with the contents of the repository. For example, `contents: read` permits an action to list the commits, and `contents:write` allows the action to create a release. For more information, see "Permissions required for GitHub Apps."
`deployments`	Work with deployments. For example, `deployments: write` permits an action to create a new deployment. For more information, see "Permissions required for GitHub Apps."
`discussions`	Work with GitHub Discussions. For example, `discussions: write` permits an action to close

	`discussions: write` permits an action to close or delete a discussion. For more information, see "Using the GraphQL API for Discussions."
`issues`	Work with issues. For example, `issues: write` permits an action to add a comment to an issue. For more information, see "Permissions required for GitHub Apps."
`packages`	Work with GitHub Packages. For example, `packages: write` permits an action to upload and publish packages on GitHub Packages. For more information, see "About permissions for GitHub Packages."
`pages`	Work with GitHub Pages. For example, `pages: write` permits an action to request a GitHub Pages build. For more information, see "Permissions required for GitHub Apps."
`pull-requests`	Work with pull requests. For example, `pull-requests: write` permits an action to add a label to a pull request. For more information, see "Permissions required for GitHub Apps."
`repository-projects`	Work with GitHub projects (classic). For example, `repository-projects: write` permits an action to add a column to a project (classic). For more information, see "Permissions required for GitHub Apps."
`security-events`	Work with GitHub code scanning and Dependabot alerts. For example, `security-events: read` permits an action to list the Dependabot alerts for the repository, and `security-events: write` allows an action to update the status of a code scanning alert. For more information, see "Repository permissions for 'Code scanning alerts'" and "Repository permissions for 'Dependabot alerts'" in "Permissions required for GitHub Apps."
`statuses`	Work with commit statuses. For example, `statuses:read` permits an action to list the commit statuses for a given reference. For more information, see "Permissions required for GitHub Apps."

Defining access for the `GITHUB_TOKEN` scopes

You can define the access that the `GITHUB_TOKEN` will permit by specifying `read`, `write`, or `none` as the value of the available scopes within the `permissions` key.

```yaml
permissions:
  actions: read|write|none
  checks: read|write|none
  contents: read|write|none
  deployments: read|write|none
  issues: read|write|none
  discussions: read|write|none
  packages: read|write|none
  pages: read|write|none
  pull-requests: read|write|none
  repository-projects: read|write|none
  security-events: read|write|none
```

```
  statuses: read|write|none
```

If you specify the access for any of these scopes, all of those that are not specified are set to `none`.

You can use the following syntax to define one of `read-all` or `write-all` access for all of the available scopes:

```yaml
permissions: read-all
```

```yaml
permissions: write-all
```

You can use the following syntax to disable permissions for all of the available scopes:

```yaml
permissions: {}
```

Changing the permissions in a forked repository

You can use the `permissions` key to add and remove read permissions for forked repositories, but typically you can't grant write access. The exception to this behavior is where an admin user has selected the **Send write tokens to workflows from pull requests** option in the GitHub Actions settings. For more information, see "Managing GitHub Actions settings for a repository."

Setting the `GITHUB_TOKEN` permissions for all jobs in a workflow

You can specify `permissions` at the top level of a workflow, so that the setting applies to all jobs in the workflow.

Example: Setting the `GITHUB_TOKEN` permissions for an entire workflow

This example shows permissions being set for the `GITHUB_TOKEN` that will apply to all jobs in the workflow. All permissions are granted read access.

```yaml
name: "My workflow"

on: [ push ]

permissions: read-all

jobs:
  ...
```

`env`

A `map` of variables that are available to the steps of all jobs in the workflow. You can also set variables that are only available to the steps of a single job or to a single step. For more information, see `jobs.<job_id>.env` and `jobs.<job_id>.steps[*].env`.

Variables in the `env` map cannot be defined in terms of other variables in the map.

When more than one environment variable is defined with the same name, GitHub uses the most specific variable. For example, an environment variable defined in a step will override job and workflow environment variables with the same name, while the step executes. An environment variable defined for a job will override a workflow variable

with the same name, while the job executes.

Example of `env`

```yaml
env:
  SERVER: production
```

`defaults`

Use `defaults` to create a `map` of default settings that will apply to all jobs in the workflow. You can also set default settings that are only available to a job. For more information, see `jobs.<job_id>.defaults`.

When more than one default setting is defined with the same name, GitHub uses the most specific default setting. For example, a default setting defined in a job will override a default setting that has the same name defined in a workflow.

`defaults.run`

You can use `defaults.run` to provide default `shell` and `working-directory` options for all `run` steps in a workflow. You can also set default settings for `run` that are only available to a job. For more information, see `jobs.<job_id>.defaults.run`. You cannot use contexts or expressions in this keyword.

When more than one default setting is defined with the same name, GitHub uses the most specific default setting. For example, a default setting defined in a job will override a default setting that has the same name defined in a workflow.

Example: Set the default shell and working directory

```yaml
defaults:
  run:
    shell: bash
    working-directory: ./scripts
```

`concurrency`

Use `concurrency` to ensure that only a single job or workflow using the same concurrency group will run at a time. A concurrency group can be any string or expression. The expression can only use `github`, `inputs` and `vars` contexts. For more information about expressions, see "Expressions."

You can also specify `concurrency` at the job level. For more information, see `jobs.<job_id>.concurrency`.

When a concurrent job or workflow is queued, if another job or workflow using the same concurrency group in the repository is in progress, the queued job or workflow will be `pending`. Any previously pending job or workflow in the concurrency group will be canceled. To also cancel any currently running job or workflow in the same concurrency group, specify `cancel-in-progress: true`.

Notes:

- The concurrency group name is case insensitive. For example, `prod` and `Prod` will be treated as the same concurrency group.

- Ordering is not guaranteed for jobs or runs using concurrency groups, they are handled in the order that they are processed.

Examples: Using concurrency and the default behavior

```yaml
concurrency: staging_environment
```

```yaml
concurrency: ci-${{ github.ref }}
```

Example: Using concurrency to cancel any in-progress job or run

```yaml
concurrency:
  group: ${{ github.ref }}
  cancel-in-progress: true
```

Example: Using a fallback value

If you build the group name with a property that is only defined for specific events, you can use a fallback value. For example, `github.head_ref` is only defined on `pull_request` events. If your workflow responds to other events in addition to `pull_request` events, you will need to provide a fallback to avoid a syntax error. The following concurrency group cancels in-progress jobs or runs on `pull_request` events only; if `github.head_ref` is undefined, the concurrency group will fallback to the run ID, which is guaranteed to be both unique and defined for the run.

```yaml
concurrency:
  group: ${{ github.head_ref || github.run_id }}
  cancel-in-progress: true
```

Example: Only cancel in-progress jobs or runs for the current workflow

If you have multiple workflows in the same repository, concurrency group names must be unique across workflows to avoid canceling in-progress jobs or runs from other workflows. Otherwise, any previously in-progress or pending job will be canceled, regardless of the workflow.

To only cancel in-progress runs of the same workflow, you can use the `github.workflow` property to build the concurrency group:

```yaml
concurrency:
  group: ${{ github.workflow }}-${{ github.ref }}
  cancel-in-progress: true
```

`jobs`

A workflow run is made up of one or more `jobs`, which run in parallel by default. To run jobs sequentially, you can define dependencies on other jobs using the `jobs.<job_id>.needs` keyword.

Each job runs in a runner environment specified by `runs-on`.

You can run an unlimited number of jobs as long as you are within the workflow usage limits. For more information, see "Usage limits, billing, and administration" for GitHub-hosted runners and "About self-hosted runners" for self-hosted runner usage limits.

If you need to find the unique identifier of a job running in a workflow run, you can use the GitHub Enterprise Server API. For more information, see "Actions."

`jobs.<job_id>`

Use `jobs.<job_id>` to give your job a unique identifier. The key `job_id` is a string and its value is a map of the job's configuration data. You must replace `<job_id>` with a string that is unique to the `jobs` object. The `<job_id>` must start with a letter or `_` and contain only alphanumeric characters, `-`, or `_`.

Example: Creating jobs

In this example, two jobs have been created, and their `job_id` values are `my_first_job` and `my_second_job`.

```yaml
jobs:
  my_first_job:
    name: My first job
  my_second_job:
    name: My second job
```

`jobs.<job_id>.name`

Use `jobs.<job_id>.name` to set a name for the job, which is displayed in the GitHub UI.

`jobs.<job_id>.permissions`

For a specific job, you can use `jobs.<job_id>.permissions` to modify the default permissions granted to the `GITHUB_TOKEN`, adding or removing access as required, so that you only allow the minimum required access. For more information, see "Automatic token authentication."

By specifying the permission within a job definition, you can configure a different set of permissions for the `GITHUB_TOKEN` for each job, if required. Alternatively, you can specify the permissions for all jobs in the workflow. For information on defining permissions at the workflow level, see `permissions`.

For each of the available scopes, shown in the table below, you can assign one of the permissions: `read`, `write`, or `none`. If you specify the access for any of these scopes, all of those that are not specified are set to `none`.

Available scopes and details of what each allows an action to do:

Scope	Allows an action using `GITHUB_TOKEN` to
`actions`	Work with GitHub Actions. For example, `actions: write` permits an action to cancel a workflow run. For more information, see "Permissions required for GitHub Apps."
`checks`	Work with check runs and check suites. For example, `checks: write` permits an action to create a check run. For more information, see "Permissions required for GitHub Apps."

`contents`	Work with the contents of the repository. For example, `contents: read` permits an action to list the commits, and `contents:write` allows the action to create a release. For more information, see "Permissions required for GitHub Apps."
`deployments`	Work with deployments. For example, `deployments: write` permits an action to create a new deployment. For more information, see "Permissions required for GitHub Apps."
`discussions`	Work with GitHub Discussions. For example, `discussions: write` permits an action to close or delete a discussion. For more information, see "Using the GraphQL API for Discussions."
`issues`	Work with issues. For example, `issues: write` permits an action to add a comment to an issue. For more information, see "Permissions required for GitHub Apps."
`packages`	Work with GitHub Packages. For example, `packages: write` permits an action to upload and publish packages on GitHub Packages. For more information, see "About permissions for GitHub Packages."
`pages`	Work with GitHub Pages. For example, `pages: write` permits an action to request a GitHub Pages build. For more information, see "Permissions required for GitHub Apps."
`pull-requests`	Work with pull requests. For example, `pull-requests: write` permits an action to add a label to a pull request. For more information, see "Permissions required for GitHub Apps."
`repository-projects`	Work with GitHub projects (classic). For example, `repository-projects: write` permits an action to add a column to a project (classic). For more information, see "Permissions required for GitHub Apps."
`security-events`	Work with GitHub code scanning and Dependabot alerts. For example, `security-events: read` permits an action to list the Dependabot alerts for the repository, and `security-events: write` allows an action to update the status of a code scanning alert. For more information, see "Repository permissions for 'Code scanning alerts'" and "Repository permissions for 'Dependabot alerts'" in "Permissions required for GitHub Apps."
`statuses`	Work with commit statuses. For example, `statuses:read` permits an action to list the commit statuses for a given reference. For more information, see "Permissions required for GitHub Apps."

Defining access for the `GITHUB_TOKEN` scopes

You can define the access that the `GITHUB_TOKEN` will permit by specifying `read`, `write`,

or `none` as the value of the available scopes within the `permissions` key.

```yaml
permissions:
  actions: read|write|none
  checks: read|write|none
  contents: read|write|none
  deployments: read|write|none
  issues: read|write|none
  discussions: read|write|none
  packages: read|write|none
  pages: read|write|none
  pull-requests: read|write|none
  repository-projects: read|write|none
  security-events: read|write|none
  statuses: read|write|none
```

If you specify the access for any of these scopes, all of those that are not specified are set to `none`.

You can use the following syntax to define one of `read-all` or `write-all` access for all of the available scopes:

```yaml
permissions: read-all
```

```yaml
permissions: write-all
```

You can use the following syntax to disable permissions for all of the available scopes:

```yaml
permissions: {}
```

Changing the permissions in a forked repository

You can use the `permissions` key to add and remove read permissions for forked repositories, but typically you can't grant write access. The exception to this behavior is where an admin user has selected the **Send write tokens to workflows from pull requests** option in the GitHub Actions settings. For more information, see "Managing GitHub Actions settings for a repository."

Example: Setting the `GITHUB_TOKEN` permissions for one job in a workflow

This example shows permissions being set for the `GITHUB_TOKEN` that will only apply to the job named `stale`. Write access is granted for the `issues` and `pull-requests` scopes. All other scopes will have no access.

```yaml
jobs:
  stale:
    runs-on: ubuntu-latest

    permissions:
      issues: write
      pull-requests: write

    steps:
      - uses: actions/stale@v5
```

`jobs.<job_id>.needs`

Use `jobs.<job_id>.needs` to identify any jobs that must complete successfully before this job will run. It can be a string or array of strings. If a job fails or is skipped, all jobs that need it are skipped unless the jobs use a conditional expression that causes the job to continue. If a run contains a series of jobs that need each other, a failure or skip applies to all jobs in the dependency chain from the point of failure or skip onwards. If you would like a job to run even if a job it is dependent on did not succeed, use the `always()` conditional expression in `jobs.<job_id>.if`.

Example: Requiring successful dependent jobs

```yaml
jobs:
  job1:
  job2:
    needs: job1
  job3:
    needs: [job1, job2]
```

In this example, `job1` must complete successfully before `job2` begins, and `job3` waits for both `job1` and `job2` to complete.

The jobs in this example run sequentially:

1. `job1`
2. `job2`
3. `job3`

Example: Not requiring successful dependent jobs

```yaml
jobs:
  job1:
  job2:
    needs: job1
  job3:
    if: ${{ always() }}
    needs: [job1, job2]
```

In this example, `job3` uses the `always()` conditional expression so that it always runs after `job1` and `job2` have completed, regardless of whether they were successful. For more information, see "Expressions."

`jobs.<job_id>.if`

You can use the `jobs.<job_id>.if` conditional to prevent a job from running unless a condition is met. You can use any supported context and expression to create a conditional. For more information on which contexts are supported in this key, see "Contexts."

> **Note:** The `jobs.<job_id>.if` condition is evaluated before `jobs.<job_id>.strategy.matrix` is applied.

When you use expressions in an `if` conditional, you can, optionally, omit the `${{ }}` expression syntax because GitHub Actions automatically evaluates the `if` conditional as an expression. However, this exception does not apply everywhere.

You must always use the `${{ }}` expression syntax or escape with `''`, `""`, or `()` when

the expression starts with `!`, since `!` is reserved notation in YAML format. For example:

```
if: ${{ ! startsWith(github.ref, 'refs/tags/') }}
```

For more information, see "Expressions."

Example: Only run job for specific repository

This example uses `if` to control when the `production-deploy` job can run. It will only run if the repository is named `octo-repo-prod` and is within the `octo-org` organization. Otherwise, the job will be marked as *skipped*.

YAML

```yaml
name: example-workflow
on: [push]
jobs:
  production-deploy:
    if: github.repository == 'octo-org/octo-repo-prod'
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v3
        with:
          node-version: '14'
      - run: npm install -g bats
```

`jobs.<job_id>.runs-on`

Use `jobs.<job_id>.runs-on` to define the type of machine to run the job on.

- The destination machine can be a self-hosted runner.
- You can target runners based on the labels assigned to them, or their group membership, or a combination of these.
- You can provide `runs-on` as:
  - a single string
  - a single variable containing a string
  - an array of strings, variables containing strings, or a combination of both
  - a `key: value` pair using the `group` or `label` keys
- If you specify an array of strings or variables, your workflow will execute on any runner that matches all of the specified `runs-on` values. For example, here the job will only run on a self-hosted runner that has the labels `linux`, `x64`, and `gpu`:

  ```yaml
  runs-on: [self-hosted, linux, x64, gpu]
  ```

  For more information, see "Choosing self-hosted runners."
- You can mix strings and variables in an array. For example:

  ```yaml
  on:
    workflow_dispatch:
      inputs:
        chosen-os:
          required: true
          type: choice
  ```

```
        options:
        - Ubuntu
        - macOS

jobs:
  test:
    runs-on: [self-hosted, "${{ inputs.chosen-os }}"]
    steps:
    - run: echo Hello world!
```

- If you would like to run your workflow on multiple machines, use `jobs.<job_id>.strategy`.

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

Choosing GitHub-hosted runners

If you use a GitHub-hosted runner, each job runs in a fresh instance of a runner image specified by `runs-on`.

Available GitHub-hosted runner types are:

Virtual Machine	Processor (CPU)	Memory (RAM)	Storage (SSD)	OS (YAML workflow label)	Notes
Linux	2	7 GB	14 GB	`ubuntu-latest`, `ubuntu-22.04`, `ubuntu-20.04`	The `ubuntu-latest` label currently uses the Ubuntu 22.04 runner image.
Windows	2	7 GB	14 GB	`windows-latest`, `windows-2022`, `windows-2019`	The `windows-latest` label currently uses the Windows 2022 runner image.
macOS	3	14 GB	14 GB	`macos-latest`, `macos-12`, `macos-11`	The `macos-latest` workflow label currently uses the macOS 12 runner image.
macOS	4	14 GB	14 GB	`macos-13` [Beta]	N/A

Note: The `-latest` runner images are the latest stable images that GitHub provides, and might not be the most recent version of the operating system available from the operating system vendor.

Warning: Beta and Deprecated Images are provided "as-is", "with all faults" and "as available" and are excluded from the service level agreement and warranty. Beta Images may not be covered by customer support.

Example: Specifying an operating system

```
runs-on: ubuntu-latest
```

For more information, see "Using GitHub-hosted runners."

Choosing self-hosted runners

To specify a self-hosted runner for your job, configure `runs-on` in your workflow file with self-hosted runner labels.

All self-hosted runners have the `self-hosted` label. Using only this label will select any self-hosted runner. To select runners that meet certain criteria, such as operating system or architecture, we recommend providing an array of labels that begins with `self-hosted` (this must be listed first) and then includes additional labels as needed. When you specify an array of labels, jobs will be queued on runners that have all the labels that you specify.

Although the `self-hosted` label is not required, we strongly recommend specifying it when using self-hosted runners to ensure that your job does not unintentionally specify any current or future GitHub-hosted runners.

Example: Using labels for runner selection

```
runs-on: [self-hosted, linux]
```

For more information, see "About self-hosted runners" and "Using self-hosted runners in a workflow."

Choosing runners in a group

You can use `runs-on` to target runner groups, so that the job will execute on any runner that is a member of that group. For more granular control, you can also combine runner groups with labels.

Example: Using groups to control where jobs are run

In this example, Ubuntu runners have been added to a group called `ubuntu-runners`. The `runs-on` key sends the job to any available runner in the `ubuntu-runners` group:

```
name: learn-github-actions
on: [push]
jobs:
  check-bats-version:
    runs-on:
      group: ubuntu-runners
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v3
        with:
          node-version: '14'
      - run: npm install -g bats
      - run: bats -v
```

Example: Combining groups and labels

When you combine groups and labels, the runner must meet both requirements to be eligible to run the job.

In this example, a runner group called `ubuntu-runners` is populated with Ubuntu runners, which have also been assigned the label `ubuntu-20.04-16core`. The `runs-on` key combines `group` and `labels` so that the job is routed to any available runner within the group that also has a matching label:

```
name: learn-github-actions
on: [push]
jobs:
```

```yaml
  check-bats-version:
    runs-on:
      group: ubuntu-runners
      labels: ubuntu-20.04-16core
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v3
        with:
          node-version: '14'
      - run: npm install -g bats
      - run: bats -v
```

`jobs.<job_id>.environment`

Use `jobs.<job_id>.environment` to define the environment that the job references. All deployment protection rules must pass before a job referencing the environment is sent to a runner. For more information, see "Using environments for deployment."

You can provide the environment as only the environment `name`, or as an environment object with the `name` and `url`. The URL maps to `environment_url` in the deployments API. For more information about the deployments API, see "Repositories."

Example: Using a single environment name

```yaml
environment: staging_environment
```

Example: Using environment name and URL

```yaml
environment:
  name: production_environment
  url: https://github.com
```

The value of `url` can be an expression. Allowed expression contexts: `github`, `inputs`, `vars`, `needs`, `strategy`, `matrix`, `job`, `runner`, and `env`. For more information about expressions, see "Expressions."

Example: Using output as URL

```yaml
environment:
  name: production_environment
  url: ${{ steps.step_id.outputs.url_output }}
```

The value of `name` can be an expression. Allowed expression contexts: `github`, `inputs`, `vars`, `needs`, `strategy`, and `matrix`. For more information about expressions, see "Expressions."

Example: Using an expression as environment name

```yaml
environment:
  name: ${{ github.ref_name }}
```

`jobs.<job_id>.concurrency`

You can use `jobs.<job_id>.concurrency` to ensure that only a single job or workflow

using the same concurrency group will run at a time. A concurrency group can be any string or expression. Allowed expression contexts: `github`, `inputs`, `vars`, `needs`, `strategy`, and `matrix`. For more information about expressions, see "Expressions."

You can also specify `concurrency` at the workflow level. For more information, see `concurrency`.

When a concurrent job or workflow is queued, if another job or workflow using the same concurrency group in the repository is in progress, the queued job or workflow will be `pending`. Any previously pending job or workflow in the concurrency group will be canceled. To also cancel any currently running job or workflow in the same concurrency group, specify `cancel-in-progress: true`.

Notes:

- The concurrency group name is case insensitive. For example, `prod` and `Prod` will be treated as the same concurrency group.
- Ordering is not guaranteed for jobs or runs using concurrency groups, they are handled in the order that they are processed.

Examples: Using concurrency and the default behavior

```yaml
concurrency: staging_environment
```

```yaml
concurrency: ci-${{ github.ref }}
```

Example: Using concurrency to cancel any in-progress job or run

```yaml
concurrency:
  group: ${{ github.ref }}
  cancel-in-progress: true
```

Example: Using a fallback value

If you build the group name with a property that is only defined for specific events, you can use a fallback value. For example, `github.head_ref` is only defined on `pull_request` events. If your workflow responds to other events in addition to `pull_request` events, you will need to provide a fallback to avoid a syntax error. The following concurrency group cancels in-progress jobs or runs on `pull_request` events only; if `github.head_ref` is undefined, the concurrency group will fallback to the run ID, which is guaranteed to be both unique and defined for the run.

```yaml
concurrency:
  group: ${{ github.head_ref || github.run_id }}
  cancel-in-progress: true
```

Example: Only cancel in-progress jobs or runs for the current workflow

If you have multiple workflows in the same repository, concurrency group names must be unique across workflows to avoid canceling in-progress jobs or runs from other workflows. Otherwise, any previously in-progress or pending job will be canceled, regardless of the workflow.

To only cancel in-progress runs of the same workflow, you can use the `github.workflow` property to build the concurrency group:

```yaml
concurrency:
  group: ${{ github.workflow }}-${{ github.ref }}
  cancel-in-progress: true
```

`jobs.<job_id>.outputs`

You can use `jobs.<job_id>.outputs` to create a `map` of outputs for a job. Job outputs are available to all downstream jobs that depend on this job. For more information on defining job dependencies, see `jobs.<job_id>.needs`.

Outputs are Unicode strings, and can be a maximum of 1 MB. The total of all outputs in a workflow run can be a maximum of 50 MB.

Job outputs containing expressions are evaluated on the runner at the end of each job. Outputs containing secrets are redacted on the runner and not sent to GitHub Actions.

To use job outputs in a dependent job, you can use the `needs` context. For more information, see "Contexts."

Example: Defining outputs for a job

```yaml
jobs:
  job1:
    runs-on: ubuntu-latest
    # Map a step output to a job output
    outputs:
      output1: ${{ steps.step1.outputs.test }}
      output2: ${{ steps.step2.outputs.test }}
    steps:
      - id: step1
        run: echo "test=hello" >> "$GITHUB_OUTPUT"
      - id: step2
        run: echo "test=world" >> "$GITHUB_OUTPUT"
  job2:
    runs-on: ubuntu-latest
    needs: job1
    steps:
      - env:
          OUTPUT1: ${{needs.job1.outputs.output1}}
          OUTPUT2: ${{needs.job1.outputs.output2}}
        run: echo "$OUTPUT1 $OUTPUT2"
```

`jobs.<job_id>.env`

A `map` of variables that are available to all steps in the job. You can set variables for the entire workflow or an individual step. For more information, see `env` and `jobs.<job_id>.steps[*].env`.

When more than one environment variable is defined with the same name, GitHub uses the most specific variable. For example, an environment variable defined in a step will override job and workflow environment variables with the same name, while the step executes. An environment variable defined for a job will override a workflow variable with the same name, while the job executes.

Example of `jobs.<job_id>.env`

```yaml
jobs:
  job1:
    env:
      FIRST_NAME: Mona
```

`jobs.<job_id>.defaults`

Use `jobs.<job_id>.defaults` to create a `map` of default settings that will apply to all steps in the job. You can also set default settings for the entire workflow. For more information, see `defaults`.

When more than one default setting is defined with the same name, GitHub uses the most specific default setting. For example, a default setting defined in a job will override a default setting that has the same name defined in a workflow.

`jobs.<job_id>.defaults.run`

Use `jobs.<job_id>.defaults.run` to provide default `shell` and `working-directory` to all `run` steps in the job. This keyword can reference several contexts. For more information, see "Contexts."

You can provide default `shell` and `working-directory` options for all `run` steps in a job. You can also set default settings for `run` for the entire workflow. For more information, see `defaults.run`.

When more than one default setting is defined with the same name, GitHub uses the most specific default setting. For example, a default setting defined in a job will override a default setting that has the same name defined in a workflow.

Example: Setting default `run` step options for a job

```yaml
jobs:
  job1:
    runs-on: ubuntu-latest
    defaults:
      run:
        shell: bash
        working-directory: ./scripts
```

`jobs.<job_id>.steps`

A job contains a sequence of tasks called `steps`. Steps can run commands, run setup tasks, or run an action in your repository, a public repository, or an action published in a Docker registry. Not all steps run actions, but all actions run as a step. Each step runs in its own process in the runner environment and has access to the workspace and filesystem. Because steps run in their own process, changes to environment variables are not preserved between steps. GitHub provides built-in steps to set up and complete a job.

GitHub only displays the first 1,000 checks, however, you can run an unlimited number of steps as long as you are within the workflow usage limits. For more information, see "Usage limits, billing, and administration" for GitHub-hosted runners and "About self-hosted runners" for self-hosted runner usage limits.

Example of `jobs.<job_id>.steps`

```yaml
name: Greeting from Mona

on: push

jobs:
  my-job:
    name: My Job
    runs-on: ubuntu-latest
    steps:
      - name: Print a greeting
        env:
          MY_VAR: Hi there! My name is
          FIRST_NAME: Mona
          MIDDLE_NAME: The
          LAST_NAME: Octocat
        run: |
          echo $MY_VAR $FIRST_NAME $MIDDLE_NAME $LAST_NAME.
```

`jobs.<job_id>.steps[*].id`

A unique identifier for the step. You can use the `id` to reference the step in contexts. For more information, see "Contexts."

`jobs.<job_id>.steps[*].if`

You can use the `if` conditional to prevent a step from running unless a condition is met. You can use any supported context and expression to create a conditional. For more information on which contexts are supported in this key, see "Contexts."

When you use expressions in an `if` conditional, you can, optionally, omit the `${{ }}` expression syntax because GitHub Actions automatically evaluates the `if` conditional as an expression. However, this exception does not apply everywhere.

You must always use the `${{ }}` expression syntax or escape with `''`, `""`, or `()` when the expression starts with `!`, since `!` is reserved notation in YAML format. For example:

```yaml
if: ${{ ! startsWith(github.ref, 'refs/tags/') }}
```

For more information, see "Expressions."

Example: Using contexts

This step only runs when the event type is a `pull_request` and the event action is `unassigned`.

```yaml
steps:
  - name: My first step
    if: ${{ github.event_name == 'pull_request' && github.event.action ==
'unassigned' }}
    run: echo This event is a pull request that had an assignee removed.
```

Example: Using status check functions

The `my backup step` only runs when the previous step of a job fails. For more information, see "Expressions."

```yaml
steps:
  - name: My first step
```

```
    uses: octo-org/action-name@main
  - name: My backup step
    if: ${{ failure() }}
    uses: actions/heroku@1.0.0
```

Example: Using secrets

Secrets cannot be directly referenced in `if:` conditionals. Instead, consider setting secrets as job-level environment variables, then referencing the environment variables to conditionally run steps in the job.

If a secret has not been set, the return value of an expression referencing the secret (such as `${{ secrets.SuperSecret }}` in the example) will be an empty string.

```
name: Run a step if a secret has been set
on: push
jobs:
  my-jobname:
    runs-on: ubuntu-latest
    env:
      super_secret: ${{ secrets.SuperSecret }}
    steps:
      - if: ${{ env.super_secret != '' }}
        run: echo 'This step will only run if the secret has a value set.'
      - if: ${{ env.super_secret == '' }}
        run: echo 'This step will only run if the secret does not have a value
set.'
```

For more information, see "Contexts" and "Using secrets in GitHub Actions."

`jobs.<job_id>.steps[*].name`

A name for your step to display on GitHub.

`jobs.<job_id>.steps[*].uses`

Selects an action to run as part of a step in your job. An action is a reusable unit of code. You can use an action defined in the same repository as the workflow, a public repository, or in a published Docker container image.

We strongly recommend that you include the version of the action you are using by specifying a Git ref, SHA, or Docker tag. If you don't specify a version, it could break your workflows or cause unexpected behavior when the action owner publishes an update.

- Using the commit SHA of a released action version is the safest for stability and security.
- If the action publishes major version tags, you should expect to receive critical fixes and security patches while still retaining compatibility. Note that this behavior is at the discretion of the action's author.
- Using the default branch of an action may be convenient, but if someone releases a new major version with a breaking change, your workflow could break.

Some actions require inputs that you must set using the `with` keyword. Review the action's README file to determine the inputs required.

Actions are either JavaScript files or Docker containers. If the action you're using is a Docker container you must run the job in a Linux environment. For more details, see `runs-on`.

Example: Using versioned actions

```yaml
steps:
  # Reference a specific commit
  - uses: actions/checkout@8f4b7f84864484a7bf31766abe9204da3cbe65b3
  # Reference the major version of a release
  - uses: actions/checkout@v4
  # Reference a specific version
  - uses: actions/checkout@v4.2.0
  # Reference a branch
  - uses: actions/checkout@main
```

Example: Using a public action

`{owner}/{repo}@{ref}`

You can specify a branch, ref, or SHA in a public GitHub repository.

```yaml
jobs:
  my_first_job:
    steps:
      - name: My first step
        # Uses the default branch of a public repository
        uses: actions/heroku@main
      - name: My second step
        # Uses a specific version tag of a public repository
        uses: actions/aws@v2.0.1
```

Example: Using a public action in a subdirectory

`{owner}/{repo}/{path}@{ref}`

A subdirectory in a public GitHub repository at a specific branch, ref, or SHA.

```yaml
jobs:
  my_first_job:
    steps:
      - name: My first step
        uses: actions/aws/ec2@main
```

Example: Using an action in the same repository as the workflow

`./path/to/dir`

The path to the directory that contains the action in your workflow's repository. You must check out your repository before using the action.

```yaml
jobs:
  my_first_job:
    steps:
      - name: Check out repository
        uses: actions/checkout@v4
      - name: Use local my-action
        uses: ./.github/actions/my-action
```

Example: Using a Docker Hub action

`docker://{image}:{tag}`

A Docker image published on Docker Hub.

```yaml
jobs:
  my_first_job:
    steps:
      - name: My first step
        uses: docker://alpine:3.8
```

Example: Using a Docker public registry action

`docker://{host}/{image}:{tag}`

A Docker image in a public registry. This example uses the Google Container Registry at `gcr.io`.

```yaml
jobs:
  my_first_job:
    steps:
      - name: My first step
        uses: docker://gcr.io/cloud-builders/gradle
```

Example: Using an action inside a different private repository than the workflow

Your workflow must checkout the private repository and reference the action locally. Generate a personal access token and add the token as a secret. For more information, see "Managing your personal access tokens" and "Using secrets in GitHub Actions."

Replace `PERSONAL_ACCESS_TOKEN` in the example with the name of your secret.

```yaml
jobs:
  my_first_job:
    steps:
      - name: Check out repository
        uses: actions/checkout@v4
        with:
          repository: octocat/my-private-repo
          ref: v1.0
          token: ${{ secrets.PERSONAL_ACCESS_TOKEN }}
          path: ./.github/actions/my-private-repo
      - name: Run my action
        uses: ./.github/actions/my-private-repo/my-action
```

Alternatively, use a GitHub App instead of a personal access token in order to ensure your workflow continues to run even if the personal access token owner leaves. For more information, see "Making authenticated API requests with a GitHub App in a GitHub Actions workflow."

`jobs.<job_id>.steps[*].run`

Runs command-line programs using the operating system's shell. If you do not provide a `name`, the step name will default to the text specified in the `run` command.

Commands run using non-login shells by default. You can choose a different shell and customize the shell used to run commands. For more information, see `jobs.<job_id>.steps[*].shell`.

Each `run` keyword represents a new process and shell in the runner environment. When

you provide multi-line commands, each line runs in the same shell. For example:

- A single-line command:

```yaml
- name: Install Dependencies
  run: npm install
```

- A multi-line command:

```yaml
- name: Clean install dependencies and build
  run: |
    npm ci
    npm run build
```

`jobs.<job_id>.steps[*].working-directory`

Using the `working-directory` keyword, you can specify the working directory of where to run the command.

```yaml
- name: Clean temp directory
  run: rm -rf *
  working-directory: ./temp
```

Alternatively, you can specify a default working directory for all `run` steps in a job, or for all `run` steps in the entire workflow. For more information, see "`defaults.run`" and "`jobs.<job_id>.defaults.run`."

You can also use a `run` step to run a script. For more information, see "Essential features of GitHub Actions."

`jobs.<job_id>.steps[*].shell`

You can override the default shell settings in the runner's operating system using the `shell` keyword. You can use built-in `shell` keywords, or you can define a custom set of shell options. The shell command that is run internally executes a temporary file that contains the commands specified in the `run` keyword.

Supported platform	`shell` parameter	Description	Command run internally
Linux / macOS	unspecified	The default shell on non-Windows platforms. Note that this runs a different command to when `bash` is specified explicitly. If `bash` is not found in the path, this is treated as `sh`.	`bash -e {0}`
All	`bash`	The default shell on non-Windows platforms with a fallback to `sh`. When specifying a bash shell on Windows, the bash shell included with Git for Windows is used.	`bash --noprofile --norc -eo pipefail {0}`

All	`pwsh`	The PowerShell Core. GitHub appends the extension `.ps1` to your script name.	`pwsh -command ". '{0}'"`
All	`python`	Executes the python command.	`python {0}`
Linux / macOS	`sh`	The fallback behavior for non-Windows platforms if no shell is provided and `bash` is not found in the path.	`sh -e {0}`
Windows	`cmd`	GitHub appends the extension `.cmd` to your script name and substitutes for `{0}`.	`%ComSpec% /D /E:ON /V:OFF /S /C "CALL "{0}""`.
Windows	`pwsh`	This is the default shell used on Windows. The PowerShell Core. GitHub appends the extension `.ps1` to your script name. If your self-hosted Windows runner does not have *PowerShell Core* installed, then *PowerShell Desktop* is used instead.	`pwsh -command ". '{0}'"`.
Windows	`powershell`	The PowerShell Desktop. GitHub appends the extension `.ps1` to your script name.	`powershell -command ". '{0}'"`.

Example: Running a command using Bash

```yaml
steps:
  - name: Display the path
    shell: bash
    run: echo $PATH
```

Example: Running a command using Windows `cmd`

```yaml
steps:
  - name: Display the path
    shell: cmd
    run: echo %PATH%
```

Example: Running a command using PowerShell Core

```yaml
steps:
  - name: Display the path
    shell: pwsh
    run: echo ${env:PATH}
```

Example: Using PowerShell Desktop to run a command

```
steps:
  - name: Display the path
    shell: powershell
    run: echo ${env:PATH}
```

Example: Running an inline Python script

```
steps:
  - name: Display the path
    shell: python
    run: |
      import os
      print(os.environ['PATH'])
```

Custom shell

You can set the `shell` value to a template string using `command [options] {0} [more_options]`. GitHub interprets the first whitespace-delimited word of the string as the command, and inserts the file name for the temporary script at `{0}`.

For example:

```
steps:
  - name: Display the environment variables and their values
    shell: perl {0}
    run: |
      print %ENV
```

The command used, `perl` in this example, must be installed on the runner.

Exit codes and error action preference

For built-in shell keywords, we provide the following defaults that are executed by GitHub-hosted runners. You should use these guidelines when running shell scripts.

- `bash` / `sh`:
  - By default, fail-fast behavior is enforced using `set -e` for both `sh` and `bash`. When `shell: bash` is specified, `-o pipefail` is also applied to enforce early exit from pipelines that generate a non-zero exit status.
  - You can take full control over shell parameters by providing a template string to the shell options. For example, `bash {0}`.
  - `sh`-like shells exit with the exit code of the last command executed in a script, which is also the default behavior for actions. The runner will report the status of the step as fail/succeed based on this exit code.
- `powershell` / `pwsh`
  - Fail-fast behavior when possible. For `pwsh` and `powershell` built-in shell, we will prepend `$ErrorActionPreference = 'stop'` to script contents.
  - We append `if ((Test-Path -LiteralPath variable:\LASTEXITCODE)) { exit $LASTEXITCODE }` to powershell scripts so action statuses reflect the script's last exit code.
  - Users can always opt out by not using the built-in shell, and providing a custom shell option like: `pwsh -File {0}`, or `powershell -Command "& '{0}'"`, depending

on need.

- `cmd`
  - There doesn't seem to be a way to fully opt into fail-fast behavior other than writing your script to check each error code and respond accordingly. Because we can't actually provide that behavior by default, you need to write this behavior into your script.
  - `cmd.exe` will exit with the error level of the last program it executed, and it will return the error code to the runner. This behavior is internally consistent with the previous `sh` and `pwsh` default behavior and is the `cmd.exe` default, so this behavior remains intact.

`jobs.<job_id>.steps[*].with`

A `map` of the input parameters defined by the action. Each input parameter is a key/value pair. Input parameters are set as environment variables. The variable is prefixed with `INPUT_` and converted to upper case.

Input parameters defined for a Docker container must use `args`. For more information, see "`jobs.<job_id>.steps[*].with.args`."

Example of `jobs.<job_id>.steps[*].with`

Defines the three input parameters (`first_name`, `middle_name`, and `last_name`) defined by the `hello_world` action. These input variables will be accessible to the `hello-world` action as `INPUT_FIRST_NAME`, `INPUT_MIDDLE_NAME`, and `INPUT_LAST_NAME` environment variables.

```yaml
jobs:
  my_first_job:
    steps:
      - name: My first step
        uses: actions/hello_world@main
        with:
          first_name: Mona
          middle_name: The
          last_name: Octocat
```

`jobs.<job_id>.steps[*].with.args`

A `string` that defines the inputs for a Docker container. GitHub passes the `args` to the container's `ENTRYPOINT` when the container starts up. An `array of strings` is not supported by this parameter. A single argument that includes spaces should be surrounded by double quotes `""`.

Example of `jobs.<job_id>.steps[*].with.args`

```yaml
steps:
  - name: Explain why this job ran
    uses: octo-org/action-name@main
    with:
      entrypoint: /bin/echo
      args: The ${{ github.event_name }} event triggered this step.
```

The `args` are used in place of the `CMD` instruction in a `Dockerfile`. If you use `CMD` in your `Dockerfile`, use the guidelines ordered by preference:

1. Document required arguments in the action's README and omit them from the `CMD` instruction.
2. Use defaults that allow using the action without specifying any `args`.
3. If the action exposes a `--help` flag, or something similar, use that as the default to make your action self-documenting.

`jobs.<job_id>.steps[*].with.entrypoint`

Overrides the Docker `ENTRYPOINT` in the `Dockerfile`, or sets it if one wasn't already specified. Unlike the Docker `ENTRYPOINT` instruction which has a shell and exec form, `entrypoint` keyword accepts only a single string defining the executable to be run.

Example of `jobs.<job_id>.steps[*].with.entrypoint`

```yaml
steps:
  - name: Run a custom command
    uses: octo-org/action-name@main
    with:
      entrypoint: /a/different/executable
```

The `entrypoint` keyword is meant to be used with Docker container actions, but you can also use it with JavaScript actions that don't define any inputs.

`jobs.<job_id>.steps[*].env`

Sets variables for steps to use in the runner environment. You can also set variables for the entire workflow or a job. For more information, see `env` and `jobs.<job_id>.env`.

When more than one environment variable is defined with the same name, GitHub uses the most specific variable. For example, an environment variable defined in a step will override job and workflow environment variables with the same name, while the step executes. An environment variable defined for a job will override a workflow variable with the same name, while the job executes.

Public actions may specify expected variables in the README file. If you are setting a secret or sensitive value, such as a password or token, you must set secrets using the `secrets` context. For more information, see "Contexts."

Example of `jobs.<job_id>.steps[*].env`

```yaml
steps:
  - name: My first action
    env:
      GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
      FIRST_NAME: Mona
      LAST_NAME: Octocat
```

`jobs.<job_id>.steps[*].continue-on-error`

Prevents a job from failing when a step fails. Set to `true` to allow a job to pass when this step fails.

`jobs.<job_id>.steps[*].timeout-minutes`

The maximum number of minutes to run the step before killing the process.

`jobs.<job_id>.timeout-minutes`

The maximum number of minutes to let a job run before GitHub automatically cancels it. Default: 360

If the timeout exceeds the job execution time limit for the runner, the job will be canceled when the execution time limit is met instead. For more information about job execution time limits, see "Usage limits, billing, and administration" for GitHub-hosted runners and "About self-hosted runners" for self-hosted runner usage limits.

> **Note:** The `GITHUB_TOKEN` expires when a job finishes or after a maximum of 24 hours. For self-hosted runners, the token may be the limiting factor if the job timeout is greater than 24 hours. For more information on the `GITHUB_TOKEN`, see "Automatic token authentication."

`jobs.<job_id>.strategy`

Use `jobs.<job_id>.strategy` to use a matrix strategy for your jobs. A matrix strategy lets you use variables in a single job definition to automatically create multiple job runs that are based on the combinations of the variables. For example, you can use a matrix strategy to test your code in multiple versions of a language or on multiple operating systems. For more information, see "Using a matrix for your jobs."

`jobs.<job_id>.strategy.matrix`

Use `jobs.<job_id>.strategy.matrix` to define a matrix of different job configurations. Within your matrix, define one or more variables followed by an array of values. For example, the following matrix has a variable called `version` with the value `[10, 12, 14]` and a variable called `os` with the value `[ubuntu-latest, windows-latest]`:

```yaml
jobs:
  example_matrix:
    strategy:
      matrix:
        version: [10, 12, 14]
        os: [ubuntu-latest, windows-latest]
```

A job will run for each possible combination of the variables. In this example, the workflow will run six jobs, one for each combination of the `os` and `version` variables.

By default, GitHub Enterprise Server will maximize the number of jobs run in parallel depending on runner availability. The order of the variables in the matrix determines the order in which the jobs are created. The first variable you define will be the first job that is created in your workflow run. For example, the above matrix will create the jobs in the following order:

- `{version: 10, os: ubuntu-latest}`
- `{version: 10, os: windows-latest}`
- `{version: 12, os: ubuntu-latest}`
- `{version: 12, os: windows-latest}`
- `{version: 14, os: ubuntu-latest}`
- `{version: 14, os: windows-latest}`

A matrix will generate a maximum of 256 jobs per workflow run. This limit applies to both GitHub Enterprise Server-hosted and self-hosted runners.

The variables that you define become properties in the `matrix` context, and you can reference the property in other areas of your workflow file. In this example, you can use `matrix.version` and `matrix.os` to access the current value of `version` and `os` that the job is using. For more information, see "Contexts."

Example: Using a single-dimension matrix

You can specify a single variable to create a single-dimension matrix.

For example, the following workflow defines the variable `version` with the values `[10, 12, 14]`. The workflow will run three jobs, one for each value in the variable. Each job will access the `version` value through the `matrix.version` context and pass the value as `node-version` to the `actions/setup-node` action.

```yaml
jobs:
  example_matrix:
    strategy:
      matrix:
        version: [10, 12, 14]
    steps:
      - uses: actions/setup-node@v3
        with:
          node-version: ${{ matrix.version }}
```

Example: Using a multi-dimension matrix

You can specify multiple variables to create a multi-dimensional matrix. A job will run for each possible combination of the variables.

For example, the following workflow specifies two variables:

- Two operating systems specified in the `os` variable
- Three Node.js versions specified in the `version` variable

The workflow will run six jobs, one for each combination of the `os` and `version` variables. Each job will set the `runs-on` value to the current `os` value and will pass the current `version` value to the `actions/setup-node` action.

```yaml
jobs:
  example_matrix:
    strategy:
      matrix:
        os: [ubuntu-22.04, ubuntu-20.04]
        version: [10, 12, 14]
    runs-on: ${{ matrix.os }}
    steps:
      - uses: actions/setup-node@v3
        with:
          node-version: ${{ matrix.version }}
```

Example: Using contexts to create matrices

You can use contexts to create matrices. For more information about contexts, see "Contexts."

For example, the following workflow triggers on the `repository_dispatch` event and uses information from the event payload to build the matrix. When a repository dispatch event is created with a payload like the one below, the matrix `version` variable will have

a value of `[12, 14, 16]`. For more information about the `repository_dispatch` trigger, see "Events that trigger workflows."

```
{
  "event_type": "test",
  "client_payload": {
    "versions": [12, 14, 16]
  }
}
```

```
on:
  repository_dispatch:
    types:
      - test

jobs:
  example_matrix:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        version: ${{ github.event.client_payload.versions }}
    steps:
      - uses: actions/setup-node@v3
        with:
          node-version: ${{ matrix.version }}
```

`jobs.<job_id>.strategy.matrix.include`

Use `jobs.<job_id>.strategy.matrix.include` to expand existing matrix configurations or to add new configurations. The value of `include` is a list of objects.

For each object in the `include` list, the key:value pairs in the object will be added to each of the matrix combinations if none of the key:value pairs overwrite any of the original matrix values. If the object cannot be added to any of the matrix combinations, a new matrix combination will be created instead. Note that the original matrix values will not be overwritten, but added matrix values can be overwritten.

For example, this matrix:

```
strategy:
  matrix:
    fruit: [apple, pear]
    animal: [cat, dog]
    include:
      - color: green
      - color: pink
        animal: cat
      - fruit: apple
        shape: circle
      - fruit: banana
      - fruit: banana
        animal: cat
```

will result in six jobs with the following matrix combinations:

- `{fruit: apple, animal: cat, color: pink, shape: circle}`
- `{fruit: apple, animal: dog, color: green, shape: circle}`
- `{fruit: pear, animal: cat, color: pink}`
- `{fruit: pear, animal: dog, color: green}`
- `{fruit: banana}`
- `{fruit: banana, animal: cat}`

following this logic:

- `{color: green}` is added to all of the original matrix combinations because it can be added without overwriting any part of the original combinations.
- `{color: pink, animal: cat}` adds `color:pink` only to the original matrix combinations that include `animal: cat`. This overwrites the `color: green` that was added by the previous `include` entry.
- `{fruit: apple, shape: circle}` adds `shape: circle` only to the original matrix combinations that include `fruit: apple`.
- `{fruit: banana}` cannot be added to any original matrix combination without overwriting a value, so it is added as an additional matrix combination.
- `{fruit: banana, animal: cat}` cannot be added to any original matrix combination without overwriting a value, so it is added as an additional matrix combination. It does not add to the `{fruit: banana}` matrix combination because that combination was not one of the original matrix combinations.

Example: Expanding configurations

For example, the following workflow will run four jobs, one for each combination of `os` and `node`. When the job for the `os` value of `windows-latest` and `node` value of `16` runs, an additional variable called `npm` with the value of `6` will be included in the job.

```yaml
jobs:
  example_matrix:
    strategy:
      matrix:
        os: [windows-latest, ubuntu-latest]
        node: [14, 16]
        include:
          - os: windows-latest
            node: 16
            npm: 6
    runs-on: ${{ matrix.os }}
    steps:
      - uses: actions/setup-node@v3
        with:
          node-version: ${{ matrix.node }}
      - if: ${{ matrix.npm }}
        run: npm install -g npm@${{ matrix.npm }}
      - run: npm --version
```

Example: Adding configurations

For example, this matrix will run 10 jobs, one for each combination of `os` and `version` in the matrix, plus a job for the `os` value of `windows-latest` and `version` value of `17`.

```yaml
jobs:
  example_matrix:
    strategy:
      matrix:
        os: [macos-latest, windows-latest, ubuntu-latest]
        version: [12, 14, 16]
        include:
          - os: windows-latest
            version: 17
```

If you don't specify any matrix variables, all configurations under `include` will run. For example, the following workflow would run two jobs, one for each `include` entry. This lets you take advantage of the matrix strategy without having a fully populated matrix.

```yaml
jobs:
  includes_only:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        include:
          - site: "production"
            datacenter: "site-a"
          - site: "staging"
            datacenter: "site-b"
```

`jobs.<job_id>.strategy.matrix.exclude`

To remove specific configurations defined in the matrix, use `jobs.<job_id>.strategy.matrix.exclude`. An excluded configuration only has to be a partial match for it to be excluded. For example, the following workflow will run nine jobs: one job for each of the 12 configurations, minus the one excluded job that matches `{os: macos-latest, version: 12, environment: production}`, and the two excluded jobs that match `{os: windows-latest, version: 16}`.

```yaml
strategy:
  matrix:
    os: [macos-latest, windows-latest]
    version: [12, 14, 16]
    environment: [staging, production]
    exclude:
      - os: macos-latest
        version: 12
        environment: production
      - os: windows-latest
        version: 16
runs-on: ${{ matrix.os }}
```

Note: All `include` combinations are processed after `exclude`. This allows you to use `include` to add back combinations that were previously excluded.

`jobs.<job_id>.strategy.fail-fast`

You can control how job failures are handled with `jobs.<job_id>.strategy.fail-fast` and `jobs.<job_id>.continue-on-error`.

`jobs.<job_id>.strategy.fail-fast` applies to the entire matrix. If `jobs.<job_id>.strategy.fail-fast` is set to `true` or its expression evaluates to `true`, GitHub Enterprise Server will cancel all in-progress and queued jobs in the matrix if any job in the matrix fails. This property defaults to `true`.

`jobs.<job_id>.continue-on-error` applies to a single job. If `jobs.<job_id>.continue-on-error` is `true`, other jobs in the matrix will continue running even if the job with `jobs.<job_id>.continue-on-error: true` fails.

You can use `jobs.<job_id>.strategy.fail-fast` and `jobs.<job_id>.continue-on-error` together. For example, the following workflow will start four jobs. For each job, `continue-on-error` is determined by the value of `matrix.experimental`. If any of the jobs with `continue-on-error: false` fail, all jobs that are in progress or queued will be cancelled. If the job with `continue-on-error: true` fails, the other jobs will not be affected.

```yaml
jobs:
  test:
    runs-on: ubuntu-latest
```

```
    continue-on-error: ${{ matrix.experimental }}
    strategy:
      fail-fast: true
      matrix:
        version: [6, 7, 8]
        experimental: [false]
        include:
          - version: 9
            experimental: true
```

`jobs.<job_id>.strategy.max-parallel`

By default, GitHub Enterprise Server will maximize the number of jobs run in parallel depending on runner availability. To set the maximum number of jobs that can run simultaneously when using a `matrix` job strategy, use `jobs.<job_id>.strategy.max-parallel`.

For example, the following workflow will run a maximum of two jobs at a time, even if there are runners available to run all six jobs at once.

```
jobs:
  example_matrix:
    strategy:
      max-parallel: 2
      matrix:
        version: [10, 12, 14]
        os: [ubuntu-latest, windows-latest]
```

`jobs.<job_id>.continue-on-error`

Prevents a workflow run from failing when a job fails. Set to `true` to allow a workflow run to pass when this job fails.

Example: Preventing a specific failing matrix job from failing a workflow run

You can allow specific jobs in a job matrix to fail without failing the workflow run. For example, if you wanted to only allow an experimental job with `node` set to `15` to fail without failing the workflow run.

```
runs-on: ${{ matrix.os }}
continue-on-error: ${{ matrix.experimental }}
strategy:
  fail-fast: false
  matrix:
    node: [13, 14]
    os: [macos-latest, ubuntu-latest]
    experimental: [false]
    include:
      - node: 15
        os: ubuntu-latest
        experimental: true
```

`jobs.<job_id>.container`

Note: If your workflows use Docker container actions, job containers, or service containers, then you must use a Linux runner:

- If you are using GitHub-hosted runners, you must use an Ubuntu runner.
- If you are using self-hosted runners, you must use a Linux machine as your runner and Docker must be installed.

Use `jobs.<job_id>.container` to create a container to run any steps in a job that don't already specify a container. If you have steps that use both script and container actions, the container actions will run as sibling containers on the same network with the same volume mounts.

If you do not set a `container`, all steps will run directly on the host specified by `runs-on` unless a step refers to an action configured to run in a container.

Note: The default shell for `run` steps inside a container is `sh` instead of `bash`. This can be overridden with `jobs.<job_id>.defaults.run` or `jobs.<job_id>.steps[*].shell`.

Example: Running a job within a container

YAML

```yaml
name: CI
on:
  push:
    branches: [ main ]
jobs:
  container-test-job:
    runs-on: ubuntu-latest
    container:
      image: node:18
      env:
        NODE_ENV: development
      ports:
        - 80
      volumes:
        - my_docker_volume:/volume_mount
      options: --cpus 1
    steps:
      - name: Check for dockerenv file
        run: (ls /.dockerenv && echo Found dockerenv) || (echo No dockerenv)
```

When you only specify a container image, you can omit the `image` keyword.

```yaml
jobs:
  container-test-job:
    runs-on: ubuntu-latest
    container: node:18
```

`jobs.<job_id>.container.image`

Use `jobs.<job_id>.container.image` to define the Docker image to use as the container to run the action. The value can be the Docker Hub image name or a registry name.

`jobs.<job_id>.container.credentials`

If the image's container registry requires authentication to pull the image, you can use `jobs.<job_id>.container.credentials` to set a `map` of the `username` and `password`. The credentials are the same values that you would provide to the `docker login` command.

Example: Defining credentials for a container registry

```yaml
container:
  image: ghcr.io/owner/image
  credentials:
     username: ${{ github.actor }}
     password: ${{ secrets.github_token }}
```

`jobs.<job_id>.container.env`

Use `jobs.<job_id>.container.env` to set a `map` of environment variables in the container.

`jobs.<job_id>.container.ports`

Use `jobs.<job_id>.container.ports` to set an `array` of ports to expose on the container.

`jobs.<job_id>.container.volumes`

Use `jobs.<job_id>.container.volumes` to set an `array` of volumes for the container to use. You can use volumes to share data between services or other steps in a job. You can specify named Docker volumes, anonymous Docker volumes, or bind mounts on the host.

To specify a volume, you specify the source and destination path:

`<source>:<destinationPath>`.

The `<source>` is a volume name or an absolute path on the host machine, and `<destinationPath>` is an absolute path in the container.

Example: Mounting volumes in a container

```yaml
volumes:
  - my_docker_volume:/volume_mount
  - /data/my_data
  - /source/directory:/destination/directory
```

`jobs.<job_id>.container.options`

Use `jobs.<job_id>.container.options` to configure additional Docker container resource options. For a list of options, see "`docker create` options."

Warning: The `--network` and `--entrypoint` options are not supported.

`jobs.<job_id>.services`

Note: If your workflows use Docker container actions, job containers, or service containers, then you must use a Linux runner:

- If you are using GitHub-hosted runners, you must use an Ubuntu runner.
- If you are using self-hosted runners, you must use a Linux machine as your runner and Docker must be installed.

Used to host service containers for a job in a workflow. Service containers are useful for creating databases or cache services like Redis. The runner automatically creates a Docker network and manages the life cycle of the service containers.

If you configure your job to run in a container, or your step uses container actions, you don't need to map ports to access the service or action. Docker automatically exposes all ports between containers on the same Docker user-defined bridge network. You can directly reference the service container by its hostname. The hostname is automatically mapped to the label name you configure for the service in the workflow.

If you configure the job to run directly on the runner machine and your step doesn't use a container action, you must map any required Docker service container ports to the Docker host (the runner machine). You can access the service container using localhost and the mapped port.

For more information about the differences between networking service containers, see "About service containers."

Example: Using localhost

This example creates two services: nginx and redis. When you specify the Docker host port but not the container port, the container port is randomly assigned to a free port. GitHub sets the assigned container port in the `${{job.services.<service_name>.ports}}` context. In this example, you can access the service container ports using the `${{ job.services.nginx.ports['8080'] }}` and `${{ job.services.redis.ports['6379'] }}` contexts.

```
services:
  nginx:
    image: nginx
    # Map port 8080 on the Docker host to port 80 on the nginx container
    ports:
      - 8080:80
  redis:
    image: redis
    # Map TCP port 6379 on Docker host to a random free port on the Redis
container
    ports:
      - 6379/tcp
```

`jobs.<job_id>.services.<service_id>.image`

The Docker image to use as the service container to run the action. The value can be the Docker Hub image name or a registry name.

`jobs.<job_id>.services.<service_id>.credentials`

If the image's container registry requires authentication to pull the image, you can use `jobs.<job_id>.container.credentials` to set a `map` of the `username` and `password`. The credentials are the same values that you would provide to the `docker login` command.

Example of `jobs.<job_id>.services.<service_id>.credentials`

```
services:
  myservice1:
```

```
    image: ghcr.io/owner/myservice1
    credentials:
      username: ${{ github.actor }}
      password: ${{ secrets.github_token }}
  myservice2:
    image: dockerhub_org/myservice2
    credentials:
      username: ${{ secrets.DOCKER_USER }}
      password: ${{ secrets.DOCKER_PASSWORD }}
```

`jobs.<job_id>.services.<service_id>.env`

Sets a `map` of environment variables in the service container.

`jobs.<job_id>.services.<service_id>.ports`

Sets an `array` of ports to expose on the service container.

`jobs.<job_id>.services.<service_id>.volumes`

Sets an `array` of volumes for the service container to use. You can use volumes to share data between services or other steps in a job. You can specify named Docker volumes, anonymous Docker volumes, or bind mounts on the host.

To specify a volume, you specify the source and destination path:

`<source>:<destinationPath>`.

The `<source>` is a volume name or an absolute path on the host machine, and `<destinationPath>` is an absolute path in the container.

Example of `jobs.<job_id>.services.<service_id>.volumes`

```
volumes:
  - my_docker_volume:/volume_mount
  - /data/my_data
  - /source/directory:/destination/directory
```

`jobs.<job_id>.services.<service_id>.options`

Additional Docker container resource options. For a list of options, see "`docker create` options."

> **Warning:** The `--network` option is not supported.

`jobs.<job_id>.uses`

The location and version of a reusable workflow file to run as a job. Use one of the following syntaxes:

- `{owner}/{repo}/.github/workflows/{filename}@{ref}` for reusable workflows in public, internal and private repositories.
- `./.github/workflows/{filename}` for reusable workflows in the same repository.

In the first option, `{ref}` can be a SHA, a release tag, or a branch name. If a release tag

and a branch have the same name, the release tag takes precedence over the branch name. Using the commit SHA is the safest option for stability and security. For more information, see "Security hardening for GitHub Actions."

If you use the second syntax option (without `{owner}/{repo}` and `@{ref}`) the called workflow is from the same commit as the caller workflow. Ref prefixes such as `refs/heads` and `refs/tags` are not allowed.

Example of `jobs.<job_id>.uses`

```yaml
jobs:
  call-workflow-1-in-local-repo:
    uses: octo-org/this-repo/.github/workflows/workflow-
1.yml@172239021f7ba04fe7327647b213799853a9eb89
  call-workflow-2-in-local-repo:
    uses: ./.github/workflows/workflow-2.yml
  call-workflow-in-another-repo:
    uses: octo-org/another-repo/.github/workflows/workflow.yml@v1
```

For more information, see "Reusing workflows."

`jobs.<job_id>.with`

When a job is used to call a reusable workflow, you can use `with` to provide a map of inputs that are passed to the called workflow.

Any inputs that you pass must match the input specifications defined in the called workflow.

Unlike `jobs.<job_id>.steps[*].with`, the inputs you pass with `jobs.<job_id>.with` are not available as environment variables in the called workflow. Instead, you can reference the inputs by using the `inputs` context.

Example of `jobs.<job_id>.with`

```yaml
jobs:
  call-workflow:
    uses: octo-org/example-repo/.github/workflows/called-workflow.yml@main
    with:
      username: mona
```

`jobs.<job_id>.with.<input_id>`

A pair consisting of a string identifier for the input and the value of the input. The identifier must match the name of an input defined by `on.workflow_call.inputs.<inputs_id>` in the called workflow. The data type of the value must match the type defined by `on.workflow_call.inputs.<input_id>.type` in the called workflow.

Allowed expression contexts: `github`, and `needs`.

`jobs.<job_id>.secrets`

When a job is used to call a reusable workflow, you can use `secrets` to provide a map of secrets that are passed to the called workflow.

Any secrets that you pass must match the names defined in the called workflow.

Example of `jobs.<job_id>.secrets`

```yaml
jobs:
  call-workflow:
    uses: octo-org/example-repo/.github/workflows/called-workflow.yml@main
    secrets:
      access-token: ${{ secrets.PERSONAL_ACCESS_TOKEN }}
```

`jobs.<job_id>.secrets.inherit`

Use the `inherit` keyword to pass all the calling workflow's secrets to the called workflow. This includes all secrets the calling workflow has access to, namely organization, repository, and environment secrets. The `inherit` keyword can be used to pass secrets across repositories within the same organization, or across organizations within the same enterprise.

Example of `jobs.<job_id>.secrets.inherit`

```yaml
on:
  workflow_dispatch:

jobs:
  pass-secrets-to-workflow:
    uses: ./.github/workflows/called-workflow.yml
    secrets: inherit
```

```yaml
on:
  workflow_call:

jobs:
  pass-secret-to-action:
    runs-on: ubuntu-latest
    steps:
      - name: Use a repo or org secret from the calling workflow.
        run: echo ${{ secrets.CALLING_WORKFLOW_SECRET }}
```

`jobs.<job_id>.secrets.<secret_id>`

A pair consisting of a string identifier for the secret and the value of the secret. The identifier must match the name of a secret defined by `on.workflow_call.secrets.<secret_id>` in the called workflow.

Allowed expression contexts: `github`, `needs`, and `secrets`.

Filter pattern cheat sheet

You can use special characters in path, branch, and tag filters.

- `*`: Matches zero or more characters, but does not match the `/` character. For example, `Octo*` matches `Octocat`.
- `**`: Matches zero or more of any character.
- `?`: Matches zero or one of the preceding character.
- `+`: Matches one or more of the preceding character.
- `[]` Matches one character listed in the brackets or included in ranges. Ranges can only include `a-z`, `A-Z`, and `0-9`. For example, the range `[0-9a-z]` matches any

digit or lowercase letter. For example, `[CB]at` matches `Cat` or `Bat` and `[1-2]00` matches `100` and `200`.
- `!`: At the start of a pattern makes it negate previous positive patterns. It has no special meaning if not the first character.

The characters `*`, `[`, and `!` are special characters in YAML. If you start a pattern with `*`, `[`, or `!`, you must enclose the pattern in quotes. Also, if you use a flow sequence with a pattern containing `[` and/or `]`, the pattern must be enclosed in quotes.

```yaml
# Valid
paths:
  - '**/README.md'

# Invalid - creates a parse error that
# prevents your workflow from running.
paths:
  - **/README.md

# Valid
branches: [ main, 'release/v[0-9].[0-9]' ]

# Invalid - creates a parse error
branches: [ main, release/v[0-9].[0-9] ]
```

For more information about branch, tag, and path filter syntax, see "`on.<push>.<branches|tags>`", "`on.<pull_request>.<branches|tags>`", and "`on.<push|pull_request>.paths`."

Patterns to match branches and tags

Pattern	Description	Example matches
`feature/*`	The `*` wildcard matches any character, but does not match slash (`/`).	`feature/my-branch` `feature/your-branch`

`feature/**`	The `**` wildcard matches any character including slash (`/`) in branch and tag names.	`feature/beta-a/my-branch`
		`feature/your-branch`
		`feature/mona/the/octocat`

`main` `releases/mona-the-octocat`	Matches the exact name of a branch or tag name.	`main` `releases/mona-the-octocat`
`'*'`	Matches all branch and tag names that don't contain a slash (`/`). The `*` character is a special character in YAML. When you start a pattern with `*`, you must use quotes.	`main` `releases`
`'**'`	Matches all branch and tag names. This is the default behavior when you don't use a `branches` or `tags` filter.	`all/the/branches` `every/tag`

`'*feature'`	The `*` character is a special character in YAML. When you start a pattern with `*`, you must use quotes.	`mona-feature`
		`feature`
		`ver-10-feature`

`v2*`	Matches branch and tag names that start with `v2`.	`v2`
		`v2.0`
		`v2.9`

`v[12].[0-9]+.[0-9]+`	Matches all semantic versioning branches and tags with major version 1 or 2.	`v1.10.1` `v2.0.0`

Patterns to match file paths

Path patterns must match the whole path, and start from the repository's root.

Pattern	Description of matches	Example matches
`'*'`	The `*` wildcard matches any character, but does not match slash (`/`). The `*` character is a special character in YAML. When you start a pattern with `*`, you must use quotes.	`README.md` `server.rb`
`'*.jsx?'`	The `?` character matches zero or one of the preceding character.	`page.js` `page.jsx`
`'**'`	The `**` wildcard matches any character including slash (`/`). This is the default behavior when you don't use a `path` filter.	`all/the/files.md`
`'*.js'`	The `*` wildcard matches any character, but does not match slash (`/`). Matches all `.js` files at the root of the repository.	`app.js` `index.js`

`'**.js'`	Matches all `.js` files in the repository.	`index.js`
		`js/index.js`
		`src/js/app.js`

`docs/*`	All files within the root of the `docs` directory, at the root of the repository.	`docs/README.md` `docs/file.txt`
`docs/**`	Any files in the `/docs` directory at the root of the repository.	`docs/README.md` `docs/mona/octocat.txt`

`docs/**/*.md`	A file with a `.md` suffix anywhere in the `docs` directory.	`docs/README.md`
		`docs/mona/hello-world.md`
		`docs/a/markdown/file.md`

`'**/docs/**'`	Any files in a `docs` directory anywhere in the repository.	`docs/hello.md` `dir/docs/my-file.txt`

		`space/docs/plan/space.doc`

`'**/README.md'`	A README.md file anywhere in the repository.	`README.md`
		
		`js/README.md`

`'**/*src/**'`	Any file in a folder with a `src` suffix anywhere in the repository.	`a/src/app.js`
		
		`my-src/code/js/app.js`

`'**/*-post.md'`	A file with the suffix `-post.md` anywhere in the repository.	`my-post.md`
		
		`path/their-post.md`

`'**/migrate-*.sql'`	A file with the prefix `migrate-` and suffix `.sql` anywhere in the repository.	`migrate-10909.sql`
		
		`db/migrate-v1.0.sql`
		
		`db/sept/migrate-v1.sql`

`'*.md'`
 	
`'!README.md'`
	Using an exclamation mark (`!`) in front of a pattern negates it. When a file matches a pattern and also matches a negative pattern defined later in the file, the file will not be included.	`hello.md`
		
		Does not match
		
		`README.md`
		
		`docs/hello.md`

`'*.md'`
 	
`'!README.md'`
 	
`README*`
	Patterns are checked sequentially. A pattern that negates a previous pattern will re-include file paths.	`hello.md`
		
		`README.md`
		
		`README.doc`