Reusing workflows

In this article

Learn how to avoid duplication when creating a workflow by reusing existing workflows.

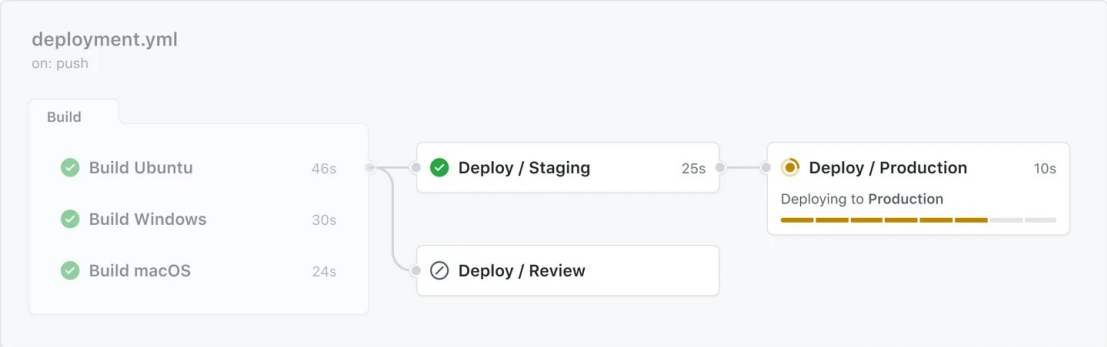
Overview

Rather than copying and pasting from one workflow to another, you can make workflows reusable. You and anyone with access to the reusable workflow can then call the reusable workflow from another workflow.

Reusing workflows avoids duplication. This makes workflows easier to maintain and allows you to create new workflows more quickly by building on the work of others, just as you do with actions. Workflow reuse also promotes best practice by helping you to use workflows that are well designed, have already been tested, and have been proven to be effective. Your organization can build up a library of reusable workflows that can be centrally maintained.

The diagram below shows an in-progress workflow run that uses a reusable workflow.

- After each of three build jobs on the left of the diagram completes successfully, a dependent job called "Deploy" is run.
- The "Deploy" job calls a reusable workflow that contains three jobs: "Staging", "Review", and "Production."
- The "Production" deployment job only runs after the "Staging" job has completed successfully.
- When a job targets an environment, the workflow run displays a progress bar that shows the number of steps in the job. In the diagram below, the "Production" job contains 8 steps, with step 6 currently being processed.
- Using a reusable workflow to run deployment jobs allows you to run those jobs for each build without duplicating code in workflows.



A workflow that uses another workflow is referred to as a "caller" workflow. The reusable workflow is a "called" workflow. One caller workflow can use multiple called workflows. Each called workflow is referenced in a single line. The result is that the caller workflow file may contain just a few lines of YAML, but may perform a large number of tasks when it's run. When you reuse a workflow, the entire called workflow is used, just as if it was part of the caller workflow.

If you reuse a workflow from a different repository, any actions in the called workflow run as if they were part of the caller workflow. For example, if the called workflow uses `actions/checkout`, the action checks out the contents of the repository that hosts the caller workflow, not the called workflow.

When a reusable workflow is triggered by a caller workflow, the `github` context is always associated with the caller workflow. The called workflow is automatically granted access to `github.token` and `secrets.GITHUB_TOKEN`. For more information about the `github` context, see "Contexts."

You can view the reused workflows referenced in your GitHub Actions workflows as dependencies in the dependency graph of the repository containing your workflows. For more information, see “About the dependency graph.”

Reusable workflows and starter workflows

Starter workflows allow everyone in your organization who has permission to create workflows to do so more quickly and easily. When people create a new workflow, they can choose a starter workflow and some or all of the work of writing the workflow will be done for them. Within a starter workflow, you can also reference reusable workflows to make it easy for people to benefit from reusing centrally managed workflow code. If you use a commit SHA when referencing the reusable workflow, you can ensure that everyone who reuses that workflow will always be using the same YAML code. However, if you reference a reusable workflow by a tag or branch, be sure that you can trust that version of the workflow. For more information, see "Security hardening for GitHub Actions."

For more information, see "Creating starter workflows for your organization."

Access to reusable workflows

A reusable workflow can be used by another workflow if any of the following is true:

- Both workflows are in the same repository.
- The called workflow is stored in a public repository, and your enterprise allows you to use public reusable workflows.
- The called workflow is stored in an internal repository and the settings for that repository allow it to be accessed. For more information, see "Sharing actions and workflows with your enterprise."
- The called workflow is stored in a private repository and the settings for that repository allow it to be accessed. For more information, see "Sharing actions and workflows with your enterprise."

Note: To enhance security, GitHub Actions does not support redirects for actions or reusable workflows. This means that when the owner, name of an action's repository, or name of an action is changed, any workflows using that action with the previous name will fail.

Using runners

Using GitHub-hosted runners

The assignment of GitHub-hosted runners is always evaluated using only the caller's context. Billing for GitHub-hosted runners is always associated with the caller. The caller workflow cannot use GitHub-hosted runners from the called repository. For more information, see "Using GitHub-hosted runners."

Using self-hosted runners

Called workflows that are owned by the same user or organization or enterprise as the caller workflow can access self-hosted runners from the caller's context. This means that a called workflow can access self-hosted runners that are:

- In the caller repository
- In the caller repository's organization or enterprise, provided that the runner has been made available to the caller repository

Limitations

- You can connect up to four levels of workflows. For more information, see "Nesting reusable workflows."
- You can call a maximum of 20 reusable workflows from a single workflow file. This limit includes any trees of nested reusable workflows that may be called starting from your top-level caller workflow file.

  For example, *top-level-caller-workflow.yml* → *called-workflow-1.yml* → *called-workflow-2.yml* counts as 2 reusable workflows.
- Any environment variables set in an `env` context defined at the workflow level in the caller workflow are not propagated to the called workflow. For more information, see "Variables" and "Contexts."
- Similarly, environment variables set in the `env` context, defined in the called workflow, are not accessible in the `env` context of the caller workflow. Instead, you must use outputs of the reusable workflow. For more information, see "Using outputs from a reusable workflow."
- To reuse variables in multiple workflows, set them at the organization, repository, or environment levels and reference them using the `vars` context. For more information see "Variables" and "Contexts."
- Reusable workflows are called directly within a job, and not from within a job step. You cannot, therefore, use `GITHUB_ENV` to pass values to job steps in the caller workflow.

Creating a reusable workflow

Reusable workflows are YAML-formatted files, very similar to any other workflow file. As with other workflow files, you locate reusable workflows in the `.github/workflows`

directory of a repository. Subdirectories of the `workflows` directory are not supported.

For a workflow to be reusable, the values for `on` must include `workflow_call`:

```
on:
  workflow_call:
```

Using inputs and secrets in a reusable workflow

You can define inputs and secrets, which can be passed from the caller workflow and then used within the called workflow. There are three stages to using an input or a secret in a reusable workflow.

1. In the reusable workflow, use the `inputs` and `secrets` keywords to define inputs or secrets that will be passed from a caller workflow.

   ```
   on:
     workflow_call:
       inputs:
         config-path:
           required: true
           type: string
       secrets:
         envPAT:
           required: true
   ```

   For details of the syntax for defining inputs and secrets, see `on.workflow_call.inputs` and `on.workflow_call.secrets`.

2. In the reusable workflow, reference the input or secret that you defined in the `on` key in the previous step.

   > **Note**: If the secrets are inherited by using `secrets: inherit` in the calling workflow, you can reference them even if they are not explicitly defined in the `on` key. For more information, see "Workflow syntax for GitHub Actions."

   ```
   jobs:
     reusable_workflow_job:
       runs-on: ubuntu-latest
       environment: production
       steps:
       - uses: actions/labeler@v4
         with:
           repo-token: ${{ secrets.envPAT }}
           configuration-path: ${{ inputs.config-path }}
   ```

   In the example above, `envPAT` is an environment secret that's been added to the `production` environment. This environment is therefore referenced within the job.

   > **Note**: Environment secrets are encrypted strings that are stored in an environment that you've defined for a repository. Environment secrets are only available to workflow jobs that reference the appropriate environment. For more information, see "Using environments for deployment."

3. Pass the input or secret from the caller workflow.

   To pass named inputs to a called workflow, use the `with` keyword in a job. Use the `secrets` keyword to pass named secrets. For inputs, the data type of the input value must match the type specified in the called workflow (either boolean, number,

or string).

```yaml
jobs:
  call-workflow-passing-data:
    uses: octo-org/example-repo/.github/workflows/reusable-workflow.yml@main
    with:
      config-path: .github/labeler.yml
    secrets:
      envPAT: ${{ secrets.envPAT }}
```

Workflows that call reusable workflows in the same organization or enterprise can use the `inherit` keyword to implicitly pass the secrets.

```yaml
jobs:
  call-workflow-passing-data:
    uses: octo-org/example-repo/.github/workflows/reusable-workflow.yml@main
    with:
      config-path: .github/labeler.yml
    secrets: inherit
```

Example reusable workflow

This reusable workflow file named `workflow-B.yml` (we'll refer to this later in the example caller workflow) takes an input string and a secret from the caller workflow and uses them in an action.

YAML

```yaml
name: Reusable workflow example

on:
  workflow_call:
    inputs:
      config-path:
        required: true
        type: string
    secrets:
      token:
        required: true

jobs:
  triage:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/labeler@v4
      with:
        repo-token: ${{ secrets.token }}
        configuration-path: ${{ inputs.config-path }}
```

Calling a reusable workflow

You call a reusable workflow by using the `uses` keyword. Unlike when you are using actions within a workflow, you call reusable workflows directly within a job, and not from within job steps.

`jobs.<job_id>.uses`

You reference reusable workflow files using 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.

You can call multiple workflows, referencing each in a separate job.

```
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
```

Passing inputs and secrets to a reusable workflow

To pass named inputs to a called workflow, use the `with` keyword in a job. Use the `secrets` keyword to pass named secrets. For inputs, the data type of the input value must match the type specified in the called workflow (either boolean, number, or string).

```
jobs:
  call-workflow-passing-data:
    uses: octo-org/example-repo/.github/workflows/reusable-workflow.yml@main
    with:
      config-path: .github/labeler.yml
    secrets:
      envPAT: ${{ secrets.envPAT }}
```

Workflows that call reusable workflows in the same organization or enterprise can use the `inherit` keyword to implicitly pass the secrets.

```
jobs:
  call-workflow-passing-data:
    uses: octo-org/example-repo/.github/workflows/reusable-workflow.yml@main
    with:
      config-path: .github/labeler.yml
    secrets: inherit
```

Using a matrix strategy with a reusable workflow

Jobs using the matrix strategy can call a reusable workflow.

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 pass different inputs to a reusable workflow. For more information about matrices, see "Using a matrix for your jobs."

This example job below calls a reusable workflow and references the matrix context by defining the variable `target` with the values `[dev, stage, prod]`. It will run three jobs, one for each value in the variable.

YAML

```yaml
jobs:
  ReuseableMatrixJobForDeployment:
    strategy:
      matrix:
        target: [dev, stage, prod]
    uses: octocat/octo-repo/.github/workflows/deployment.yml@main
    with:
      target: ${{ matrix.target }}
```

Supported keywords for jobs that call a reusable workflow

When you call a reusable workflow, you can only use the following keywords in the job containing the call:

- `jobs.<job_id>.name`
- `jobs.<job_id>.uses`
- `jobs.<job_id>.with`
- `jobs.<job_id>.with.<input_id>`
- `jobs.<job_id>.secrets`
- `jobs.<job_id>.secrets.<secret_id>`
- `jobs.<job_id>.secrets.inherit`
- `jobs.<job_id>.strategy`
- `jobs.<job_id>.needs`
- `jobs.<job_id>.if`
- `jobs.<job_id>.concurrency`
- `jobs.<job_id>.permissions`

Note:

- If `jobs.<job_id>.permissions` is not specified in the calling job, the called workflow will have the default permissions for the `GITHUB_TOKEN`. For more information, see "Automatic token authentication."
- The `GITHUB_TOKEN` permissions passed from the caller workflow can be only downgraded (not elevated) by the called workflow.
- If you use `jobs.<job_id>.concurrency.cancel-in-progress: true`, don't use the same value for `jobs.<job_id>.concurrency.group` in the called and caller workflows as this will cause the workflow that's already running to be cancelled. A called workflow uses the name of its caller workflow in `${{ github.workflow }}`, so using this context as the value of `jobs.<job_id>.concurrency.group` in both caller and called workflows will cause the caller workflow to be cancelled when the called workflow runs.

Example caller workflow

This workflow file calls two workflow files. The second of these, `workflow-B.yml` (shown in the example reusable workflow), is passed an input (`config-path`) and a secret (`token`).

YAML

```yaml
name: Call a reusable workflow
```

```yaml
on:
  pull_request:
    branches:
      - main

jobs:
  call-workflow:
    uses: octo-org/example-repo/.github/workflows/workflow-A.yml@v1

  call-workflow-passing-data:
    permissions:
      contents: read
      pull-requests: write
    uses: octo-org/example-repo/.github/workflows/workflow-B.yml@main
    with:
      config-path: .github/labeler.yml
    secrets:
      token: ${{ secrets.GITHUB_TOKEN }}
```

Nesting reusable workflows

You can connect a maximum of four levels of workflows - that is, the top-level caller workflow and up to three levels of reusable workflows. For example: *caller-workflow.yml* → *called-workflow-1.yml* → *called-workflow-2.yml* → *called-workflow-3.yml*. Loops in the workflow tree are not permitted.

From within a reusable workflow you can call another reusable workflow.

YAML

```yaml
name: Reusable workflow

on:
  workflow_call:

jobs:
  call-another-reusable:
    uses: octo-org/example-repo/.github/workflows/another-reusable.yml@v1
```

Passing secrets to nested workflows

You can use `jobs.<job_id>.secrets` in a calling workflow to pass named secrets to a directly called workflow. Alternatively, you can use `jobs.<job_id>.secrets.inherit` to pass all of the calling workflow's secrets to a directly called workflow. For more information, see the section "Reusing workflows" above, and the reference article "Workflow syntax for GitHub Actions." Secrets are only passed to directly called workflow, so in the workflow chain A > B > C, workflow C will only receive secrets from A if they have been passed from A to B, and then from B to C.

In the following example, workflow A passes all of its secrets to workflow B, by using the `inherit` keyword, but workflow B only passes one secret to workflow C. Any of the other secrets passed to workflow B are not available to workflow C.

```yaml
jobs:
  workflowA-calls-workflowB:
    uses: octo-org/example-repo/.github/workflows/B.yml@main
    secrets: inherit # pass all secrets
```

```yaml
jobs:
```

```yaml
  workflowB-calls-workflowC:
    uses: different-org/example-repo/.github/workflows/C.yml@main
    secrets:
      envPAT: ${{ secrets.envPAT }} # pass just this secret
```

Access and permissions

A workflow that contains nested reusable workflows will fail if any of the nested workflows is inaccessible to the initial caller workflow. For more information, see "Reusing workflows."

`GITHUB_TOKEN` permissions can only be the same or more restrictive in nested workflows. For example, in the workflow chain A > B > C, if workflow A has `package: read` token permission, then B and C cannot have `package: write` permission. For more information, see "Automatic token authentication."

For information on how to use the API to determine which workflow files were involved in a particular workflow run, see "Monitoring which workflows are being used."

Using outputs from a reusable workflow

A reusable workflow may generate data that you want to use in the caller workflow. To use these outputs, you must specify them as the outputs of the reusable workflow.

If a reusable workflow that sets an output is executed with a matrix strategy, the output will be the output set by the last successful completing reusable workflow of the matrix which actually sets a value. That means if the last successful completing reusable workflow sets an empty string for its output, and the second last successful completing reusable workflow sets an actual value for its output, the output will contain the value of the second last completing reusable workflow.

The following reusable workflow has a single job containing two steps. In each of these steps we set a single word as the output: "hello" and "world." In the `outputs` section of the job, we map these step outputs to job outputs called: `output1` and `output2`. In the `on.workflow_call.outputs` section we then define two outputs for the workflow itself, one called `firstword` which we map to `output1`, and one called `secondword` which we map to `output2`.

YAML

```yaml
name: Reusable workflow

on:
  workflow_call:
    # Map the workflow outputs to job outputs
    outputs:
      firstword:
        description: "The first output string"
        value: ${{ jobs.example_job.outputs.output1 }}
      secondword:
        description: "The second output string"
        value: ${{ jobs.example_job.outputs.output2 }}

jobs:
  example_job:
    name: Generate output
    runs-on: ubuntu-latest
    # Map the job outputs to step outputs
    outputs:
      output1: ${{ steps.step1.outputs.firstword }}
      output2: ${{ steps.step2.outputs.secondword }}
    steps:
```

```
      - id: step1
        run: echo "firstword=hello" >> $GITHUB_OUTPUT
      - id: step2
        run: echo "secondword=world" >> $GITHUB_OUTPUT
```

We can now use the outputs in the caller workflow, in the same way you would use the outputs from a job within the same workflow. We reference the outputs using the names defined at the workflow level in the reusable workflow: `firstword` and `secondword`. In this workflow, `job1` calls the reusable workflow and `job2` prints the outputs from the reusable workflow ("hello world") to standard output in the workflow log.

YAML

```yaml
name: Call a reusable workflow and use its outputs

on:
  workflow_dispatch:

jobs:
  job1:
    uses: octo-org/example-repo/.github/workflows/called-workflow.yml@v1

  job2:
    runs-on: ubuntu-latest
    needs: job1
    steps:
      - run: echo ${{ needs.job1.outputs.firstword }} ${{
needs.job1.outputs.secondword }}
```

For more information on using job outputs, see "Workflow syntax for GitHub Actions." If you want to share something other than a variable (e.g. a build artifact) between workflows, see "Storing workflow data as artifacts."

Monitoring which workflows are being used

You can use the GitHub REST API to monitor how reusable workflows are being used. The `prepared_workflow_job` audit log action is triggered when a workflow job is started. Included in the data recorded are:

- `repo` - the organization/repository where the workflow job is located. For a job that calls another workflow, this is the organization/repository of the caller workflow.
- `@timestamp` - the date and time that the job was started, in Unix epoch format.
- `job_name` - the name of the job that was run.
- `calling_workflow_refs` - an array of file paths for all the caller workflows involved in this workflow job. The items in the array are in the reverse order that they were called in. For example, in a chain of workflows A > B > C, when viewing the logs for a job in workflow C, the array would be `["octo-org/octo-repo/.github/workflows/B.yml", "octo-org/octo-repo/.github/workflows/A.yml"]`.
- `calling_workflow_shas` - an array of SHAs for all the caller workflows involved in this workflow job. The array contains the same number of items, in the same order, as the `calling_workflow_refs` array.
- `job_workflow_ref` - the workflow file that was used, in the form `{owner}/{repo}/{path}/{filename}@{ref}`. For a job that calls another workflow, this identifies the called workflow.

For information about using the REST API to query the audit log for an organization, see "Organizations."

Note: Audit data for `prepared_workflow_job` can only be viewed using the REST API. It is not visible in the GitHub web interface, or included in JSON/CSV exported audit data.

Re-running workflows and jobs with reusable workflows

Reusable workflows from public repositories can be referenced using a SHA, a release tag, or a branch name. For more information, see "Reusing workflows."

When you re-run a workflow that uses a reusable workflow and the reference is not a SHA, there are some behaviors to be aware of:

- Re-running all jobs in a workflow will use the reusable workflow from the specified reference. For more information about re-running all jobs in a workflow, see "Re-running workflows and jobs."
- Re-running failed jobs or a specific job in a workflow will use the reusable workflow from the same commit SHA of the first attempt. For more information about re-running failed jobs in a workflow, see "Re-running workflows and jobs." For more information about re-running a specific job in a workflow, see "Re-running workflows and jobs."

Next steps

To continue learning about GitHub Actions, see "Events that trigger workflows."

You can standardize deployments by creating a self-hosted runner group that can only execute a specific reusable workflow. For more information, see "Managing access to self-hosted runners using groups."