circleci

HOME DOCS ORBS IMAGES

Workflow orchestration

4 days ago 13 min read

Cloud Server v4.x Server v3.x

On This Page

Ask AI

Workflows in CircleCI are used to orchestrate jobs. Workflows have options to control run order, scheduling, and access to resources. This page explains how to configure workflows to suit your project. Optimizing your workflows can increase the speed of your software development through faster feedback, shorter reruns, and more efficient use of resources.

Overview

A **workflow** is a set of rules for defining a collection of jobs and their run order. Create workflows to orchestrate your jobs using the options described on this page.

With workflows, you can:

- Run and troubleshoot jobs independently with real-time status feedback.
- Schedule workflows for jobs that should only run periodically.
- Fan-out to run multiple jobs concurrently for efficient version testing.
- Fan-in to deploy to multiple platforms.
- Catch failures in real-time and rerun only failed jobs.

Workflows configuration examples

> For a full specification of the `workflows` key, see the Workflows section of the configuration reference.

Concurrent job execution

The example in this section shows the default workflow orchestration model of concurrent jobs. Concurrent jobs are defined as follows:

- Use the `workflows` key.
- Name the workflow, in this case, `build_and_test`.
- Nest the `jobs` key with a list of job names that are defined in the configuration file. In this example the jobs have no dependencies defined, so they run concurrently.

> **Using Docker?** Authenticating Docker pulls from image registries is recommended when using the Docker execution environment. Authenticated pulls allow access to private Docker images, and may also grant higher rate limits, depending on your registry provider. For further information see Using Docker authenticated pulls.

```
jobs:
  build:
    docker:
      - image: cimg/base:2023.06
    steps:
      - checkout
      - run: <command>
  test:
    docker:
      - image: cimg/base:2023.06
    steps:
      - checkout
      - run: <command>
workflows:
  build_and_test:
    jobs:
      - build
      - test
```

See the Sample concurrent workflow config ↗ for a full example.

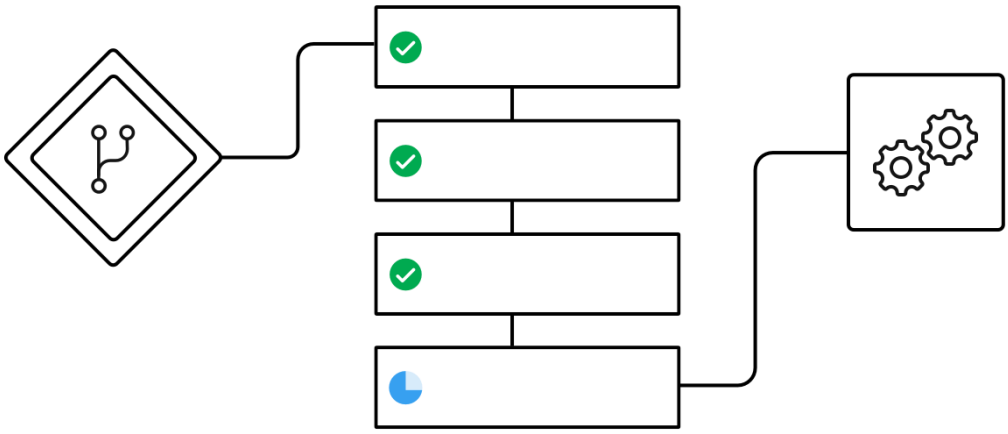
When using workflows, note the following best practices:

- Move the quickest jobs up to the start of your workflow. For example, lint or syntax checking should happen before longer-running, more computationally expensive jobs.
- Using a "setup" job at the *start* of a workflow can be helpful to do some preflight checks and populate a workspace for all the following jobs.

> Refer to the Optimization reference for tips to improve your configuration.

Sequential job execution

This example shows a workflow with four sequential jobs. Each job waits to start until the "required" job finishes successfully, as illustrated in the following diagram:



This configuration snippet is an example of a workflow configured for sequential jobs:

```
workflows:
  build-test-and-deploy:
    jobs:
      - build
      - test1:
          requires:
            - build
      - test2:
          requires:
            - test1
      - deploy:
          requires:
            - test2
```

Define job dependencies using the `requires` key. A job must wait until all upstream jobs in the dependency graph have run. In this example, the `deploy` job runs when the `build`, `test1` and `test2` jobs complete successfully:

- The `deploy` job waits for the `test2` job
- The `test2` job waits for the `test1` job

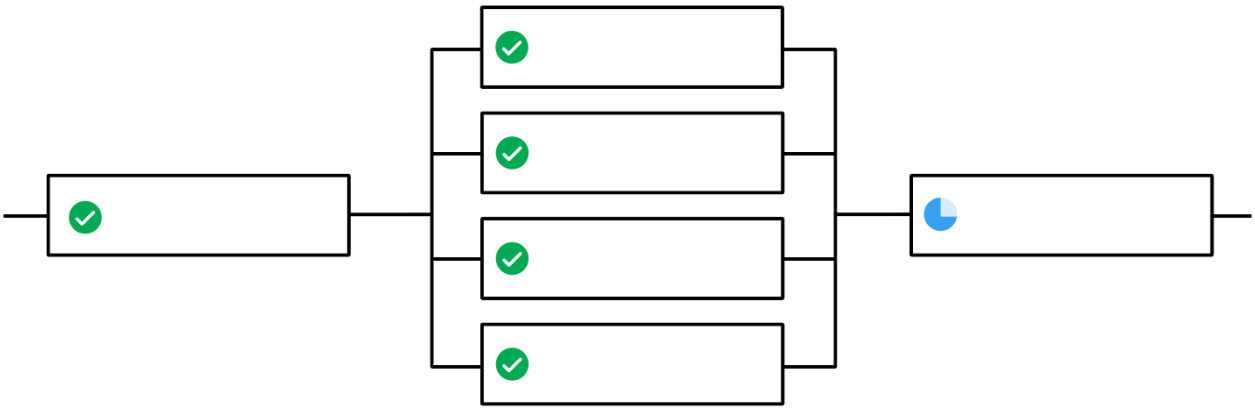
- The `test1` job waits for the `build` job

See the Sample Sequential Workflow config ↗ for a full example.

Fan-out/fan-in workflow

This example workflow has a fan-out/fan-in structure, as follows:

- A common build job is run.
- The workflow fans-out to run a set of acceptance test jobs concurrently.
- The workflow fans-in to run a common deploy job.



This configuration snippet is an example of a workflow configured for fan-out/fan-in job execution:

```
workflows:
  build_accept_deploy:
    jobs:
      - build
      - acceptance_test_1:
          requires:
            - build
      - acceptance_test_2:
```

```
            requires:
              - build
        - acceptance_test_3:
            requires:
              - build
        - acceptance_test_4:
            requires:
              - build
        - deploy:
            requires:
              - acceptance_test_1
              - acceptance_test_2
              - acceptance_test_3
              - acceptance_test_4
```

In this example, as soon as the `build` job finishes successfully, all four acceptance test jobs start. The `deploy` job waits for all four acceptance test jobs to succeed before it starts.

See the Sample Fan-in/Fan-out Workflow config ↗ for a full example.

Hold a workflow for a manual approval

Use an `approval` job to configure a workflow to wait for manual approval before continuing. Anyone who has push access to the repository can approve the job to continue the workflow or cancel to end the workflow. Approve or Cancel either by using the buttons in the CircleCI web app, or via the API.

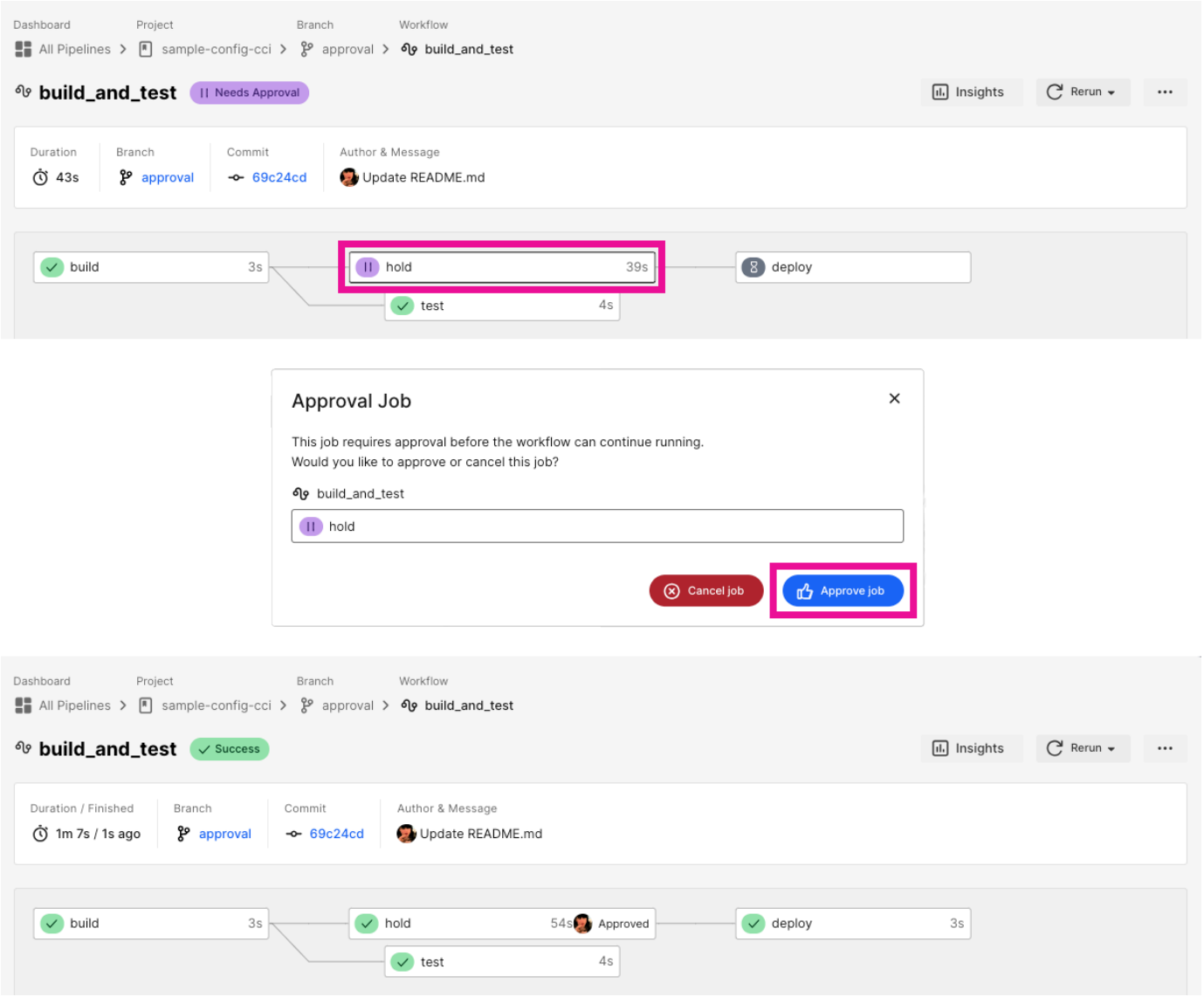
Some things to keep in mind when using manual approval in a workflow:

- `approval` is a special job type that is configured when listing jobs under the `workflows` key. You do not need to define an `approval` type job in the `jobs` section of your configuration. If you do configure steps for a job that is given the `approval` type in the workflows section, the steps for that job will not be run. An `approval` job is *only* used to *hold* the workflow for approval, not to run any work.
- The `approval` job name must be unique and not used by any other job in your configuration.
- The name of the approval job is arbitrary. For example, an approval job can be named `hold`, `wait`, `pause`, etc.
- All jobs that run *after* a manual approval job **must** `require` the name of the `approval` job.
- Jobs run in the order defined in the workflow.
- When the workflow encounters a job with `type: approval`, the workflow pauses until action is taken to approve or cancel.

- If approval is granted the workflow continues to process jobs in the order defined in the configuration file.
- If cancel is granted the downstream jobs are not run.
- Jobs downstream of an `approval` job can be restricted by adding a **restricted context** to those downstream jobs.

The following screenshot demonstrates:

- A workflow that needs approval.
- The approval popup.
- The workflow map after approval.



By clicking on the `approval` job's name (`hold`, in the screenshot above), an approval dialog box appears. You can approve, cancel, or close the popup without approving.

Configure an approval job

To set up a manual approval workflow, add a job to the `jobs` list in your workflow with `type: approval`. For example:

```
# ...
# << your config for the build, test1, test2, and deploy jobs >>
# ...

workflows:
  build-test-and-approval-deploy:
    jobs:
      - build  # your custom job from your config, that builds your code
      - test1: # your custom job; runs test suite 1
          requires: # test1 will not run until the `build` job is completed.
            - build
      - test2: # another custom job; runs test suite 2,
          requires: # test2 is dependent on the success of job `test1`
            - test1
      - hold: # <<< A job that will require manual approval in the CircleCI web application.
          type: approval # This key-value pair will set your workflow to a status of "Needs Approval"
          requires: # We only run the "hold" job when test2 has succeeded
           - test2
      # On approval of the `hold` job, any successive job that requires the `hold` job will run.
      # In this case, a user is manually triggering the deploy job.
      - deploy:
          requires:
            - hold
```

In this example, the `deploy` job will not run until the `hold` job is approved.

Approve a job

To approve a job follow these steps:

CIRCLECI WEB APP API

1. Select the `hold` job in the **Workflows** page of the CircleCI web app.
2. Select **Approve**.

Cancel a job

To Cancel a job follow these steps:

CIRCLECI WEB APP API

1. Select the `hold` job in the **Workflows** page of the CircleCI web app.
2. Select **Cancel**.

In this example, the purpose of the `hold` job is to wait for approval to begin deployment. A job can be approved for up to 90 days after it starts.

Scheduling a workflow

> Scheduled workflows are not available for projects integrated through the GitHub App, GitLab or Bitbucket Data Center.

> **The deprecation of the scheduled workflows feature is postponed**. Since the deprecation announcement went live, your feedback and feature requests have been monitored and it is clear there is more work for us to do to improve the existing scheduled pipelines experience, and also make migration easier for all. Updates on a new deprecation timeline will be announced here and on CircleCI Discuss ↗ .

By default, a workflow runs on every `git push`. To trigger a workflow on a schedule, add the `triggers` key to the workflow and specify a `schedule`. Scheduled workflows use the `cron` syntax to represent Coordinated Universal Time (UTC).

Running a workflow for every commit for every branch can be inefficient and expensive. Scheduling a workflow is an alternative to building on every commit. You can *schedule* a workflow to run at a certain time for a specific branch or branches. Consider scheduling workflows that are resource-intensive or that generate reports on a schedule rather than on every commit.

> A scheduled workflow will run on a schedule only. A scheduled workflow will **not** be run on commits to your code.

If you do not configure any workflows in your `.circleci/config.yml`, an implicit workflow is used. If you configure a scheduled workflow the implicit workflow is no longer run. If you want to build on every commit you must add your workflow to your configuration file.

> When you schedule a workflow, the workflow will be counted as an individual user seat.

Build every night

In the example below, the `nightly` workflow is configured to run every day at 12:00am UTC. The `cron` key is specified using POSIX `crontab` syntax. See the crontab man page for `cron` syntax basics. The workflow runs on the `main` and `beta` branches.

> Scheduled workflows may be delayed by up to 15 minutes. This delay is to maintain reliability during busy times, such as 12:00am UTC. Do not assume that scheduled workflows start with to-the-minute accuracy.

```yaml
workflows:
  commit:
    jobs:
      - test
      - deploy
  nightly:
    triggers:
      - schedule:
          cron: "0 0 * * *"
          filters:
            branches:
              only:
                - main
                - /^release\/.*/
    jobs:
      - coverage
```

In the above example:

- The `commit` workflow has no `triggers` key and runs on every `git push`.
- The `nightly` workflow has a `triggers` key and runs on the specified `schedule`, which is daily, and only runs on the `main` branch, as well as any branch that starts `release/`.

Specifying a valid schedule

A valid `schedule` requires:

- A `cron` key
- A `filters` key
- The `branches` filter must be present

The value of the `cron` key must be a valid crontab entry ↗ .

The following are **not** supported:

- Cron step syntax (for example, `*/1`, `*/20`).
- Range elements within comma-separated lists of elements.
- Range elements for days (for example, `Tue-Sat`).

Use comma-separated digits instead.

Example **invalid** cron range syntax:

```
1      triggers:
2        - schedule:
3            cron: "5 4 * * 1,3-5,6" # < the range separator with `-` is
4  invalid
5            filters:
6              branches:
7                only:
                   - main
```

Example **valid** cron range syntax:

```
1      triggers:
2        - schedule:
3            cron: "5 4 * * 1,3,4,5,6"
4            filters:
5              branches:
6                only:
7                  - main
```

The value of the `filters` key must be a map that defines rules for execution on specific branches.

For more details, see the `branches` section of the CircleCI configuration reference.

For a full configuration example, see the Sample Scheduled Workflows configuration ↗ .

Using contexts to share and secure environment variables

In a workflow, you can use a context to securely provide environment variables to specific jobs. Contexts allow you to define environment variables at the organization level and control access to them through security restrictions. Using contexts, sensitive data like API keys or credentials are securely shared with only the jobs that require them. Sensitive data in contexts will not be exposed in your config file.

The following example shows a workflow with four sequential jobs that each use a context to access environment variables. See the **Contexts** page for detailed instructions on this setting in the application.

The following `config.yml` snippet is an example of a sequential job workflow configured to use the environment variables defined in the `org-global` context:

```
workflows:
  build-test-and-deploy:
    jobs:
      - build
      - test1:
          requires:
            - build
          context: org-global
      - test2:
          requires:
            - test1
          context: org-global
      - deploy:
          requires:
            - test2
```

The `test1` and `test2` jobs have access to environment variables stored in the `org-global` context if the pipeline meets the restrictions set for the context, for example:

- Was the pipeline triggered by a user who **has access** (is in the relevant org/security group etc.)?
- Does the **project have access** to the context? By default all projects in an organization have access to contexts set for that organization, but restrictions on project access can be configured.
- Does the pipeline meet the requirements of any **expression restrictions** set up for the context?

Use conditional logic in workflows

You may use a `when` clause (the inverse clause `unless` is also supported) under a workflow declaration with a **[logic-statements]** to determine whether or not to run that workflow.

The example configuration below uses a pipeline parameter, `run_integration_tests` to drive the `integration_tests` workflow.

```
version: 2.1

parameters:
  run_integration_tests:
    type: boolean
    default: false

workflows:
  integration_tests:
    when: << pipeline.parameters.run_integration_tests >>
    jobs:
      - mytestjob

jobs:
```

This example prevents the workflow `integration_tests` from running unless the `run_integration_tests` pipeline parameter is `true`. For example, when the pipeline is triggered with the following in the `POST` body:

```
{
    "parameters": {
        "run_integration_tests": true
    }
}
```

Using filters in your workflows

The following sections provide examples for using filters in your workflows to manage job execution.

You can filter workflows by branch, git tag, or neither. Workflow filters for branches and tags have the keys `only` and `ignore`:

- Any branches/tags that match `only` will run the job.
- Any branches/tags that match `ignore` will not run the job.
- If neither `only` nor `ignore` are specified then the job is skipped for all branches/tags.
- If both `only` and `ignore` are specified the `only` is considered before `ignore`.

If **both branch and tag** filtering is configured and a push to your code includes both branch and tag information, the **branch** filters take precedence. In this scenario, if there are no branch filters configured, tag `ignore` filters are used, if they exist.

Branch-level job execution

The following example has one workflow that is configured to run different sets of jobs for different branches:

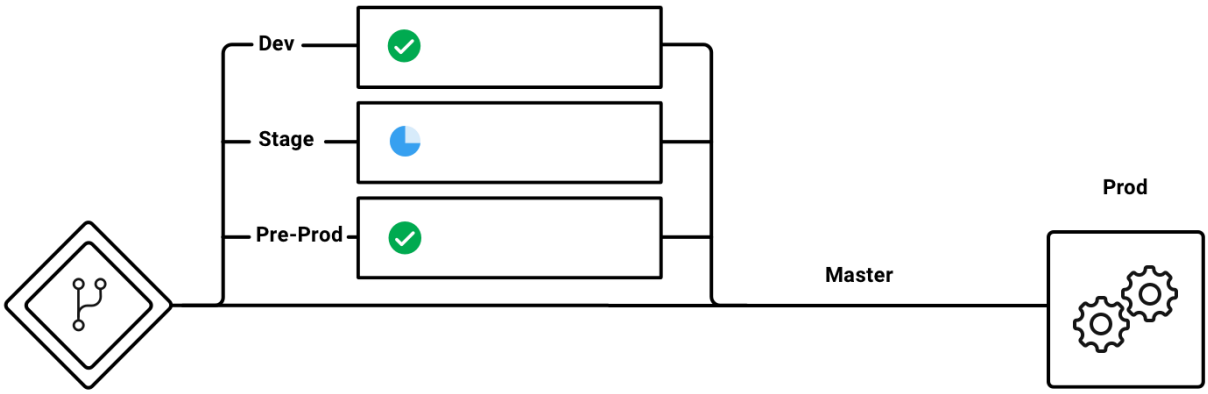
- The `test_dev` job is run on the `dev` branch and any branch that begins `user-`
- The `test_stage` job is run on the `stage` branch
- The `test_pre-prod` job is run on any branch starting `pre-prod` including any suffix added to the branch name using a hyphen.

> Workflows ignore `branches` keys used in the `jobs` declaration. If you use the deprecated job-level branches key, replace them with workflow filters.

> This example shows how to provide strings and lists of strings when configuring workflow filters.

```
workflows:
  dev_stage_pre-prod:
    jobs:
      - test_dev:
          filters:  # using regex filters requires the entire branch to
match
            branches:
              only:  # only branches matching the below regex filters
will run
                - dev
                - /user-.*/
      - test_stage:
          filters:
            branches:
              only: stage
      - test_pre-prod:
          filters:
            branches:
              only: /pre-prod(?:-.+)?$/
```

This setup can be illustrated as follows:



For more information on regular expressions, see the **Using Regular Expressions to Filter Tags And Branches** section below.

For a full example of workflows, see the **configuration file** for the Sample Sequential Workflow With Branching project.

Executing workflows for a git tag

> Webhook payloads are capped at 25 MB and for some events a maximum of 3 tags. If you push several tags at once, CircleCI may not receive them all.

CircleCI does not run workflows for tags unless you explicitly specify tag filters using regular expressions. Both lightweight and annotated tags are supported.

If you have configured a job to run on a git tag you must also specify tag filters for any dependent jobs. Use **regular expressions** to specify tag filters for a job.

In the example below, two workflows are defined:

- `untagged-build` runs the `build` job for all branches.
- `tagged-build` runs `build` for all branches **and** all tags starting with `v`.

```yaml
workflows:
  untagged-build:
    jobs:
      - build
  tagged-build:
    jobs:
      - build:
          filters:
            tags:
              only: /^v.*/
```

In the example below, two jobs are configured within the `build-deploy` workflow:

- The `build` job runs for all branches and all tags.
- The `deploy` job runs for all branches and only for tags starting with 'v'.

```yaml
workflows:
  build-deploy:
    jobs:
      - build:
          filters:  # required since `deploy` has tag filters AND requires `build`
            tags:
              only: /.*/
      - deploy:
          requires:
            - build
          filters:
            tags:
              only: /^v.*/
```

In the example below, three jobs are configured for the `build-test-deploy` workflow:

- The `build` job runs for all branches and only tags starting with 'config-test'.
- The `test` job runs once the `build` job completes for all branches and only tags starting with 'config-test'.
- The `deploy` job runs once the `test` job completes for no branches and only tags st g with 'config-test'.

```
workflows:
  build-test-deploy:
    jobs:
      - build:
          filters:  # required since `test` has tag filters AND requires
`build`
            tags:
              only: /^config-test.*/
      - test:
          requires:
            - build
          filters:  # required since `deploy` has tag filters AND
requires `test`
            tags:
              only: /^config-test.*/
      - deploy:
          requires:
            - test
          filters:
            tags:
              only: /^config-test.*/
            branches:
              ignore: /.*/
```

In the example below, two jobs are defined (`test` and `deploy`) and three workflows use those jobs:

- The `build` workflow runs for all branches except `main` and is not run on tags.
- The `staging` workflow will only run on the `main` branch and is not run on tags.
- The `production` workflow runs for no branches and only for tags starting with `v.`.

```
workflows:
  build: # This workflow will run on all branches except 'main' and will
not run on tags
    jobs:
      - test:
          filters:
            branches:
              ignore: main
  staging: # This workflow will only run on 'main' and will not run on
tags
```

```
    jobs:
      - test:
          filters: &filters-staging # this yaml anchor is setting these values to "filters-staging"
            branches:
              only: main
      - deploy:
          requires:
            - test
          filters:
            <<: *filters-staging # this is calling the previously set yaml anchor
  production: # This workflow will only run on tags (specifically starting with 'v.') and will not run on branches
    jobs:
      - test:
          filters: &filters-production # this yaml anchor is setting these values to "filters-production"
            branches:
              ignore: /.*/
            tags:
              only: /^v.*/
      - deploy:
          requires:
            - test
          filters:
            <<: *filters-production # this is calling the previously set yaml anchor
```

Using regular expressions to filter tags and branches

CircleCI branch and tag filters support the Java variant of regex pattern matching. When writing filters, CircleCI matches exact regular expressions.

For example, `only: /^config-test/` only matches the `config-test` tag. To match all tags starting with `config-test`, use `only: /^config-test.*/` instead.

Using tags for semantic versioning is a common use case. To match patch versions 3-7 of a 2.1 release, you can write `/^version-2\.1\.[3-7]/`.

For full details on pattern-matching rules, see the `java.util.regex` documentation↗ .

Using workspaces to share data between jobs

Each workflow has an associated **workspace** for transferring files to downstream jobs as a workflow progresses.

Configuration options are available to:

- **persist files to the workspace**

```
- persist_to_workspace:
    root: /tmp/workspace
    paths:
      - target/application.jar
      - build/*
```

- **attach a workflow's workspace** to a container.

```
- attach_workspace:
    at: /tmp/workspace
```

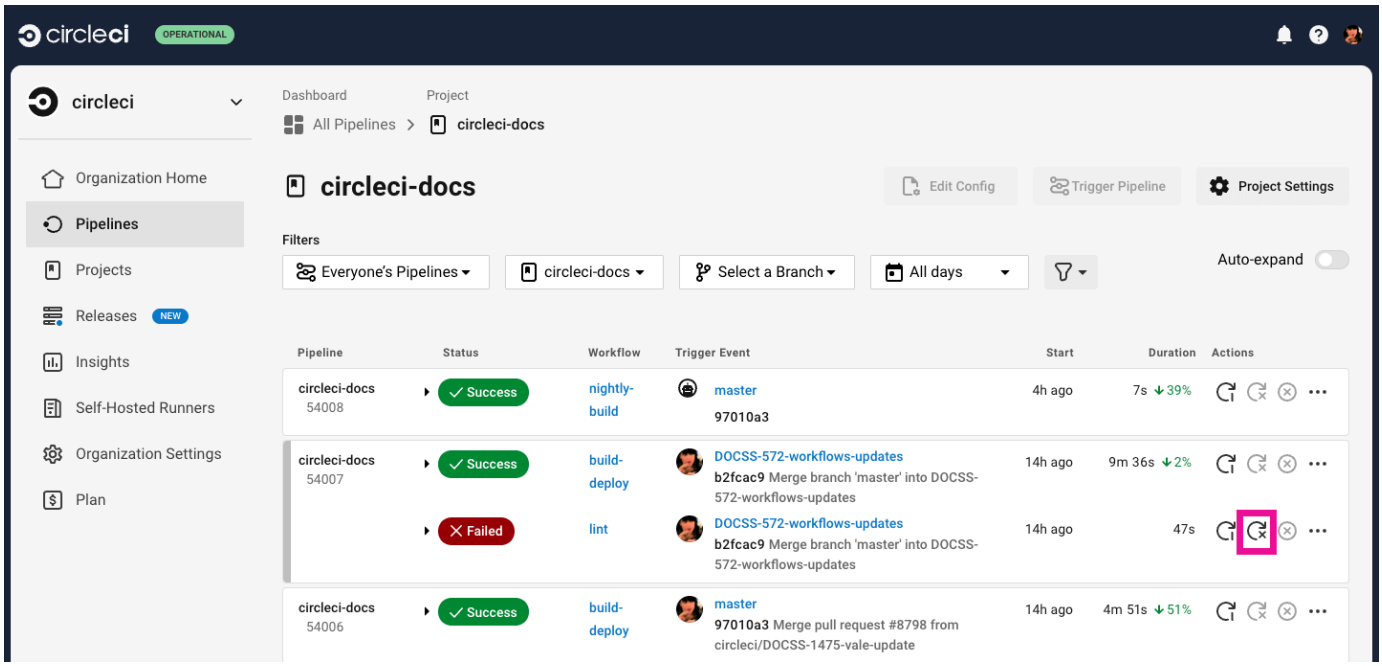
For further information on workspaces and their configuration see the **Using Workspaces to Share Data Between Jobs** doc.

Rerunning a workflow's failed jobs

Workflows help to speed up your ability to respond to failures. One way to do this is to only rerun failed jobs rather than a whole workflow. To rerun only a workflow's *failed* jobs, follow these steps:

1. In the **CircleCI web app** ↗ select your organization.
2. Select **Pipelines** in the sidebar.
3. Use the filters to find your project and pipeline.
4. Find the row in the pipeline view for the workflow you would like to rerun from failed and select the **Rerun from failed** icon. This option is also available in the workflow view using the rerun dropdown menu, which you can access by clicking on the workflow name or badge.

RERUN FROM THE PIPELINES PAGE RERUN FROM THE WORKFLOWS PAGE



> If you rerun a workflow containing a job that was previously re-run with SSH, the new workflow runs with SSH enabled for that job, even after SSH capability is disabled at the project level.

Workflow states

Workflows may have one of the following states:

State	Description	Terminal state
RUNNING	Workflow is in progress	No
NOT RUN	Workflow never started	Yes
CANCELED	Workflow canceled before it finished	Yes
FAILING	A job in the workflow failed, but others are still running or yet to be approved	No
FAILED	One or more jobs in the workflow failed	Yes
SUCCESS	All jobs in the workflow completed successfully	Yes

State	Description	Terminal state
NEEDS APPROVAL (UI) / ON HOLD	A job in the workflow is waiting for approval	No
ERROR	We experienced an internal error starting a job in the workflow	Yes
UNAUTHORIZED	One or more of the jobs terminated with a `unauthorized` job status. The user who triggered the pipeline or approved an approval job does not have access to a required restricted context.	Yes

> After 90 days non-terminal workflows are automatically by CircleCI.

Troubleshooting

This section describes common problems and solutions for workflows.

Workflow and subsequent jobs do not trigger

If you do not see your workflows running, check for configuration errors that may be preventing the workflow from starting. Navigate to your **project's pipelines** and find your workflow name to locate the failure.

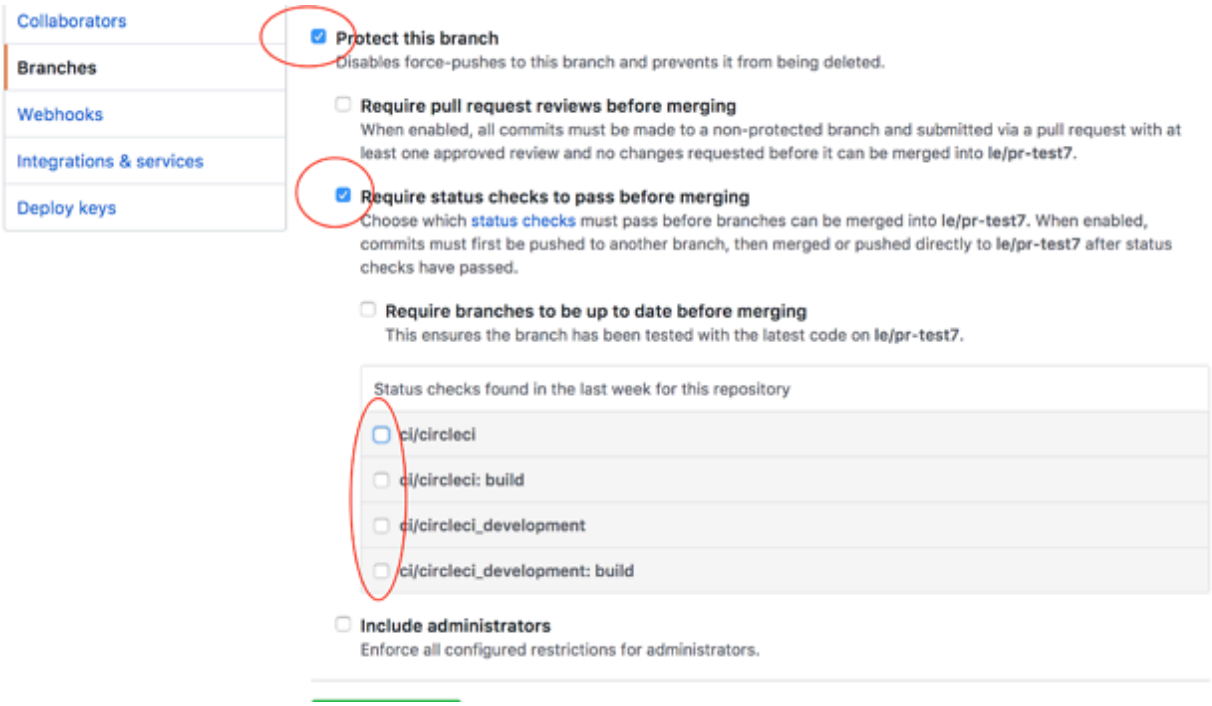
Rerunning workflows fails

Failures may happen before a workflow runs during pipeline processing. Re-running the in this case workflow will fail. Push a change to the project repository or use the trigger pipeline option to rerun the pipeline.

> You cannot rerun jobs and workflows that are >= 90 days.

Workflows waiting for status in GitHub

If you have workflows configured on a protected branch and the status check never completes, check the `ci/circleci` status key. `ci/circleci` is related to a deprecated check and sh be and deselected.



Go to **Settings** > **Branches** in GitHub and select **Edit** on the protected branch to deselect the settings, for example: `https://github.com/your-org/project/settings/branches`.

See also

- See the workflows section of the FAQ.
- For workflow configuration examples, see the CircleCI Demo Workflows page on GitHub.

Suggest an edit to this page

Make a contribution

Learn how to contribute

Still need help?

Ask the CircleCI community

Join the research community

Visit our Support site

circleci Docs

Terms of Use Privacy Policy Cookie Policy Security