Available rules for rulesets

In this article

Learn which rules you can add to a ruleset to protect specific branches and tags in a repository.

Who can use this feature
Anyone with read access to a repository can view the repository's rulesets. People with admin access to a repository, or a custom role with the "edit repository rules" permission, can create, edit, and delete rulesets for a repository and view ruleset insights. For more information, see "About custom repository roles."

Rulesets are available in public repositories with GitHub Free and GitHub Free for organizations, and in public and private repositories with GitHub Pro, GitHub Team, and GitHub Enterprise Cloud. For more information, see "GitHub’s plans."

You can create rulesets to control how users can interact with selected branches and tags in a repository. When you create a ruleset, you can choose to enable or disable the rules described in the following sections.

When you create a ruleset, you can allow certain users to bypass the rules in the ruleset. This can be users with certain permissions, specific teams, or GitHub Apps. For more information, see "About rulesets."

For more information on creating rulesets, see "Creating rulesets for repositories in your organization" and "Creating rulesets for a repository."

Restrict creations

If selected, only users with bypass permissions can create branches or tags whose name matches the pattern you specify.

Restrict updates

If selected, only users with bypass permissions can push to branches or tags whose name matches the pattern you specify.

Restrict deletions

If selected, only users with bypass permissions can delete branches or tags whose name matches the pattern you specify. This rule is selected by default.

Require linear history

Enforcing a linear commit history prevents collaborators from pushing merge commits to the targeted branches or tags. This means that any pull requests merged into the branch or tag must use a squash merge or a rebase merge. A strictly linear commit history can help teams reverse changes more easily. For more information about merge methods, see "About pull request merges."

Before you can require a linear commit history, your repository must allow squash merging or rebase merging. For more information, see "Configuring pull request merges."

Require deployments to succeed before merging

Note: This rule is not available for rulesets created at the organization level.

You can require that changes are successfully deployed to specific environments before a branch can be merged. For example, you can use this rule to ensure that changes are successfully deployed to a staging environment before the changes merge to your default branch.

Require signed commits

When you enable required commit signing on a branch, contributors and bots can only push commits that have been signed and verified to the branch. For more information, see "About commit signature verification."

Notes:

- If you have enabled vigilant mode in your account settings, which indicates that your commits will always be signed, any commits that GitHub identifies as "Partially verified" are permitted on branches that require signed commits. For more information about vigilant mode, see "Displaying verification statuses for all of your commits."
- If a collaborator pushes an unsigned commit to a branch that requires commit signatures, the collaborator will need to rebase the commit to include a verified signature, then force push the rewritten commit to the branch.

You can always push local commits to the branch if the commits are signed and verified. You can also merge signed and verified commits into the branch using a pull request on GitHub Enterprise Cloud. However, you cannot squash and merge a pull request into the branch on GitHub Enterprise Cloud unless you are the author of the pull request. You can squash and merge pull requests locally. For more information, see "Checking out pull requests locally."

For more information about merge methods, see "About merge methods on GitHub."

Require a pull request before merging

You can require that all changes to the target branch be associated with a pull request. The pull request doesn't necessarily have to be approved, but it must be opened.

Additional settings

Note: If you select **Dismiss stale pull request approvals when new commits are pushed** and/or **Require approval of the most recent reviewable push**, manually creating the merge commit for a pull request and pushing it directly to a protected branch will fail, unless the contents of the merge exactly match the merge generated by GitHub for the pull request.

In addition, with these settings, approving reviews will be dismissed as stale if the merge base introduces new changes after the review was submitted. The merge base is the commit that is the last common ancestor between the topic branch and the base branch. If the merge base changes, the pull request cannot be merged until someone approves the work again.

Repository administrators or custom roles with the "edit repository rules" permission can require that all pull requests receive a specific number of approving reviews before someone merges the pull request into a protected branch. You can require approving reviews from people with write permissions in the repository or from a designated code owner.

If you enable required reviews, collaborators can only push changes to a branch via a pull request that is approved by the required number of reviewers with write permissions.

If someone chooses the **Request changes** option in a review, then that person must approve the pull request before the pull request can be merged. If a reviewer who requests changes on a pull request isn't available, anyone with write permissions for the repository can dismiss the blocking review.

Even after all required reviewers have approved a pull request, collaborators cannot merge the pull request if there are other open pull requests that have a head branch pointing to the same commit with pending or rejected reviews. Someone with write permissions must approve or dismiss the blocking review on the other pull requests first.

Optionally, you can choose to dismiss stale pull request approvals when commits are pushed that affect the diff in the pull request. GitHub records the state of the diff at the point when a pull request is approved. This state represents the set of changes that the reviewer approved. If the diff changes from this state (for example, because a contributor pushes new changes to the pull request branch or clicks **Update branch**, or because a related pull request is merged into the target branch), the approving review is dismissed as stale, and the pull request cannot be merged until someone approves the work again. For information about the target branch branch, see "About pull requests."

Optionally, you can choose to require reviews from code owners. If you do, any pull request that modifies content with a code owner must be approved by that code owner before the pull request can be merged into the protected branch.

Optionally, you can require an approval from someone other than the last person to push to a branch before a pull request can be merged. This means at least one other authorized reviewer has approved any changes. For example, the "last reviewer" can check that the latest set of changes incorporates feedback from other reviews, and does not add new, unreviewed content.

For complex pull requests that require many reviews, requiring an approval from someone other than the last person to push can be a compromise that avoids the need to dismiss all stale reviews: with this option, "stale" reviews are not dismissed, and the pull request remains approved as long as someone other than the person who made the most recent changes approves it. Users who have already reviewed a pull request can reapprove after the most recent push to meet this requirement. If you are concerned about pull requests being "hijacked" (where unapproved content is added to approved pull requests), it is safer to dismiss stale reviews.

Optionally, you can require all comments on the pull request to be resolved before it can be merged to a branch. This ensures that all comments are addressed or acknowledged before merge.

Require status checks to pass before merging

Required status checks ensure that all required CI tests are passing before collaborators can make changes to a branch or tag targeted by your ruleset. Required status checks can be checks or statuses. For more information, see "About status checks."

You can use the commit status API to allow external services to mark commits with an appropriate status. For more information, see "Commit statuses" in the REST API documentation.

After enabling required status checks, all required status checks must pass before collaborators can merge changes into the branch or tag.

Any person or integration with write permissions to a repository can set the state of any status check in the repository, but in some cases you may only want to accept a status check from a specific GitHub App. When you add a required status check rule, you can select an app as the expected source of status updates. The app must be installed in the repository with the `statuses:write` permission, must have recently submitted a check run, and must be associated with a pre-existing required status check in the ruleset. If the status is set by any other person or integration, merging won't be allowed. If you select "any source", you can still manually verify the author of each status, listed in the merge box.

Note: For organization-level status checks, the app must be installed with the `statuses:write` permission. Only apps with this permission are displayed when configuring rulesets at the organization-level.

You can think of required status checks as being either "loose" or "strict." The type of required status check you choose determines whether your branch is required to be up to date with the base branch before merging.

Type of required status check	Setting	Merge requirements	Considerations
Strict	The **Require branches to be up to date before merging** checkbox is checked.	The topic branch **must** be up to date with the base branch before merging.	This is the default behavior for required status checks. More builds may be required, as you'll need to bring the head branch up to date after other collaborators update the target branch.
Loose	The **Require branches to be up to date before merging** checkbox is **not** checked.	The branch **does not** have to be up to date with the base branch before merging.	You'll have fewer required builds, as you won't need to bring the head branch up to date after other collaborators merge pull requests. Status checks may fail after you merge your branch if there are incompatible changes

			with the base branch.
Disabled	The **Require status checks to pass before merging** checkbox is **not** checked.	The branch has no merge restrictions.	If required status checks aren't enabled, collaborators can merge the branch at any time, regardless of whether it is up to date with the base branch. This increases the possibility of incompatible changes.

For troubleshooting information, see "Troubleshooting required status checks."

Block force pushes

You can prevent users from force pushing to the targeted branches or tags. This rule is enabled by default.

If someone force pushes to a branch or tag, commits that other collaborators have based their work on may be removed from the history of the branch or tag. This may lead to merge conflicts or corrupted pull requests. Force pushing can also be used to delete branches or point a branch to commits that were not approved in a pull request.

Enabling force pushes will not override any other rules. For example, if a branch requires a linear commit history, you cannot force push merge commits to that branch.

Require workflows to pass before merging

Note: This rule is replacing required workflows for GitHub Actions. You can read more about this change on the GitHub blog.

You can require all changes made to a targeted branch to pass specified workflows before they can be merged. This rule can only be configured at the organization level.

To use this rule, you must first create a workflow file. The workflow file needs to be in a repository that matches the visibility of the repositories you want to run it in. Specifically, a public workflow can run on any repository in your organization, an internal workflow can only run on internal and private repositories, and a private workflow can only run on private repositories. For more information, see "About workflows."

If the workflow file is in an internal or private repository and you want to use the workflow in other repositories in the organization, you will need to allow access to the workflow from outside the repository. For more information, see "Allowing access to components in an internal repository" or "Allowing access to components in a private repository."

When you add this rule to a ruleset, you will select the source repository and the workflow you want to enforce. The workflow triggers on the `pull_request` or `merge_group` events.

A workflow can also block someone from creating a repository, since a workflow can't run against a repository that's being initialized. To get around this, the ruleset either needs to have "Evaluate" as the enforcement status, or someone with bypass permissions needs to create the repository and bypass the branch protection.

Metadata restrictions

Notes:

- Adding metadata restrictions can impact the experience of people contributing to your repository. Before you enact a ruleset with metadata restrictions, you can select the "Evaluate" enforcement status for your ruleset to test the effects of any metadata restrictions without impacting contributors. For more information on metadata restrictions, see "Available rules for rulesets."
- Metadata restrictions are intended to increase consistency between commits in your repository. They are not intended to replace security measures such as requiring code review via pull requests.
- If you squash merge a branch, all commits on that branch must meet any metadata requirements for the base branch.

Organizations on a GitHub Enterprise plan can access additional rules to control how commit metadata must be formatted. You can use literal strings or regular expression syntax to define a pattern that the commit metadata must conform to. For example, you can require that commit messages contain a GitHub issue number, or that the committer or author has an email address ending in `@octoorg.com`. You can also control the format of new branch names and tag names. For a selection of useful regular expressions for commit metadata, see "Creating rulesets for a repository."

If a contributor tries to update a branch or tag with a commit that doesn't meet your requirements, the contributor will see an error telling them what was wrong with their commit. This error can appear both in the command line, when the user pushes, and on GitHub.com, when the user tries to make a commit or merge a pull request. Commits are immutable in Git: once a contributor has created a commit, they cannot edit the commit's metadata, so they may need to perform a rebase to rewrite their commit history with new commits before they can successfully contribute their work to the repository.

Metadata restrictions are useful for enforcing consistency between the commits in a branch's history. This can be useful for enforcing adherence to best practices, such as the Conventional Commits specification, or for integrating with tooling that relies on commit metadata. For example, it is easier to run scripts based on the contents of a commit message if each message conforms to a predictable format.

Important considerations for metadata restrictions

Metadata restrictions block "ref updates." If a contributor pushes work that includes a commit that doesn't meet the requirements, the push is not rejected, but the branch or tag they are targeting is not updated. Technically, the commits still enter your repository: the commits will be "retrievable" (you can navigate to them in your repository), but not "reachable" (they are not connected to the history of a branch or tag). If the contributor's push also includes work on other branches or tags, with commits that meet the requirements of those branches or tags, then those references will be successfully updated.

Metadata restrictions can increase friction for people contributing to a repository. Generally, if you impose metadata restrictions, you should do so on a limited set of branches to avoid impacting contributors' daily work. For example, instead of requiring consistent commit messages on any topic branch that a contributor might work on, you should require consistent commit messages on `main` only, then require pull requests into `main`.

If you use squash merges, you should be aware that metadata restrictions are evaluated before the merge, so all commits on the pull request must meet the requirements. For metadata restrictions that apply to committer emails, the pattern must also include `noreply@github.com` for squash merges to satisfy the restriction.

When you add metadata restrictions to an existing branch or tag, the rules are enforced for new commits pushed to the branch or tag from that point forward, but they are not enforced against the existing history of the branch or tag.