***Following could be the Strategy although there is no right or wrong approach, but this could be one of the approaches.***

Committing changes to a branch will not affect other branches, and you can share branches with others without having to merge the changes into the main project. Create new branches to isolate changes for a feature or a bug fix from your master branch and other work.

Branch policies are an important part of the Git workflow and enable you to:

*· Isolate work in progress from the completed work in your master branch*

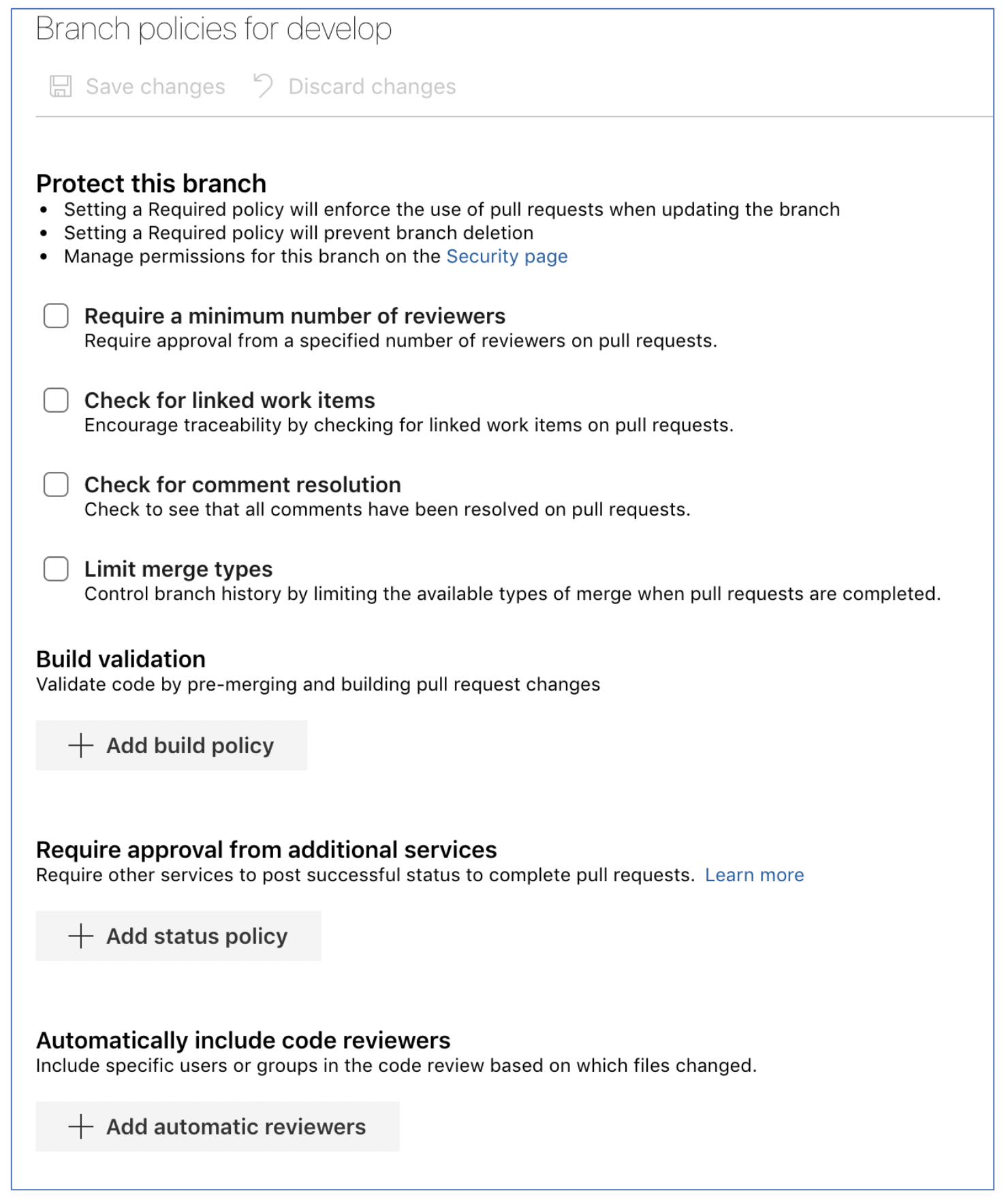
*· Guarantee changes build before they get to master*

*· Limit who can contribute to specific branches*

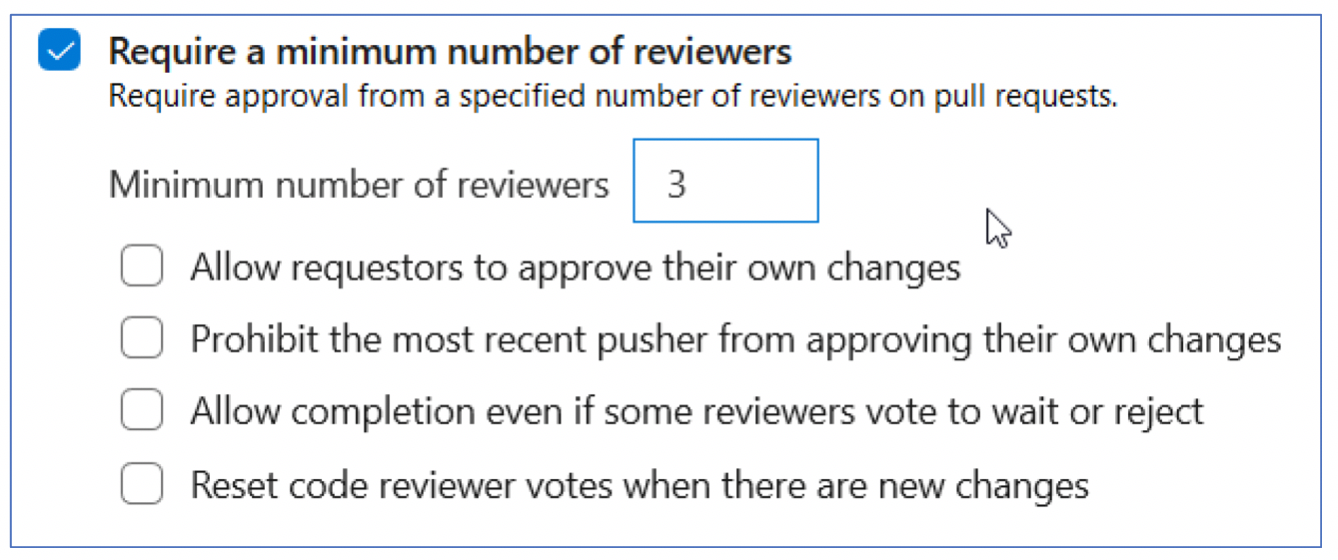
*· Enforce who can create branches and the naming guidelines for the branches*

*· Automatically include the right reviewers for every code change*

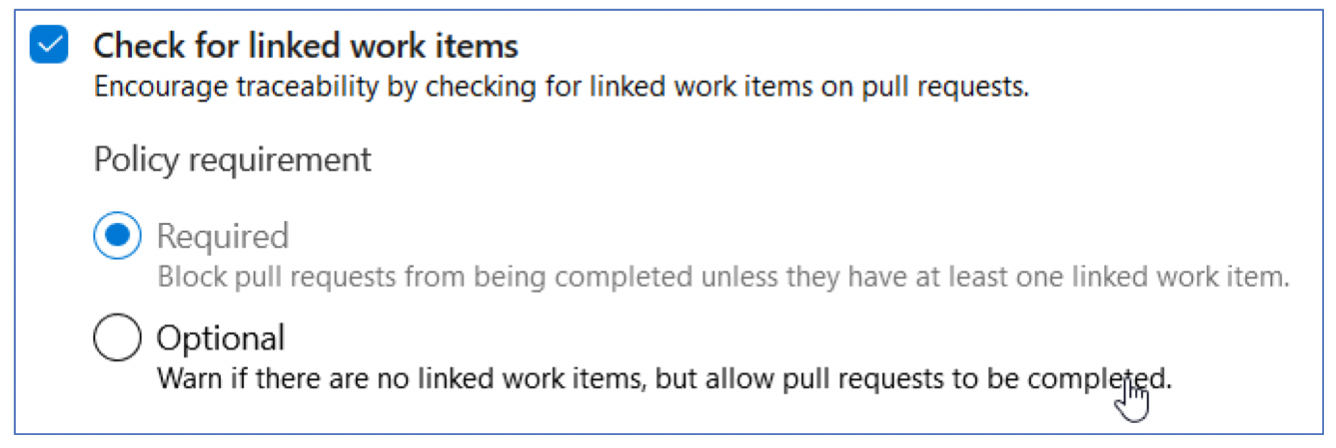
*· Enforce best practices with required code reviewers*



Code reviews are a best practice for most software development projects. To require teams to review their changes before completing a pull request, select Require a minimum number of reviewers.



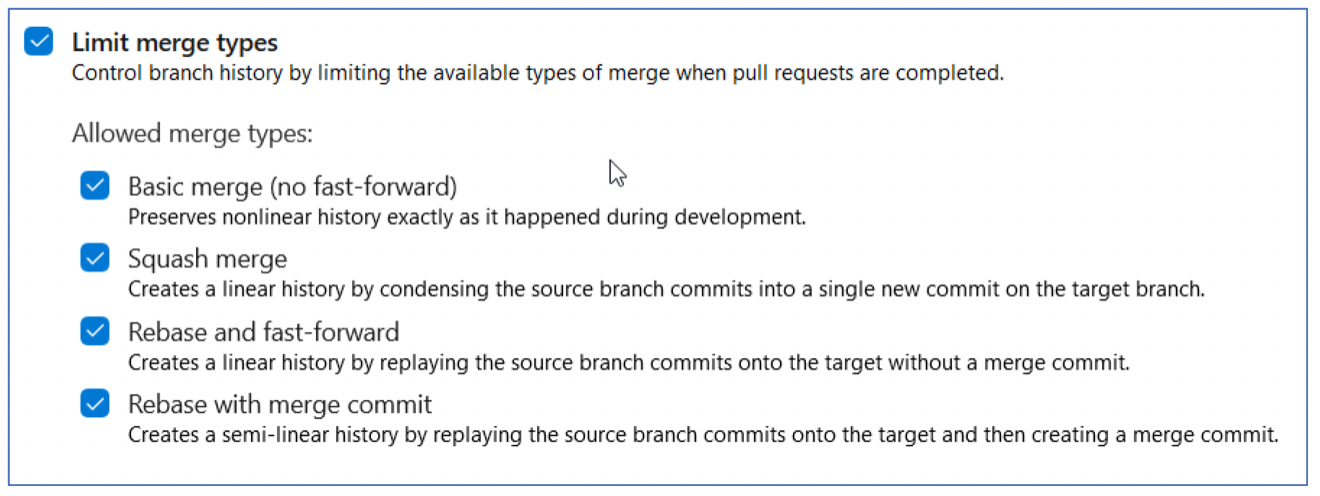
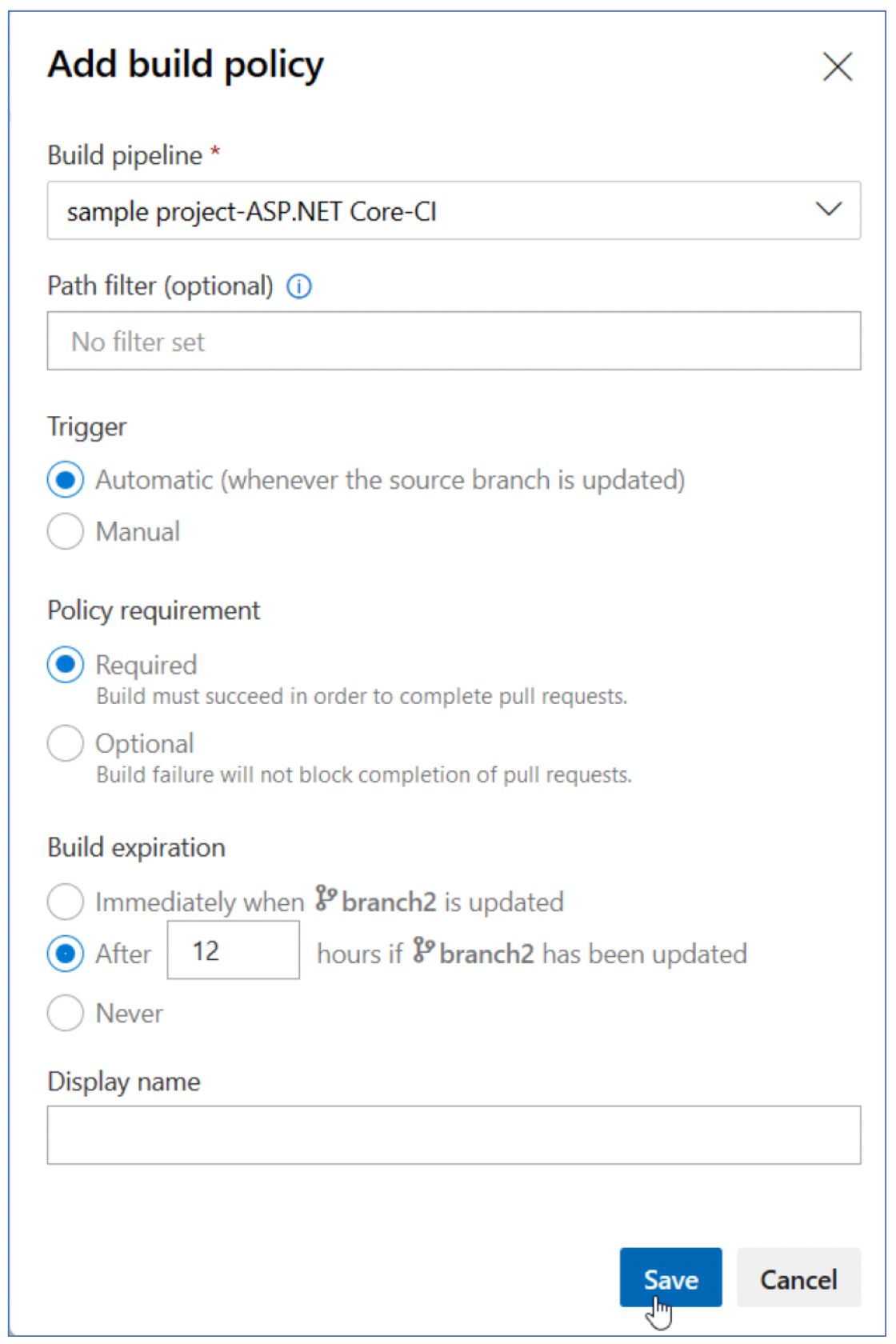
Require associations between pull requests and a work item to ensure that changes to your branch have work item management tracking. Linking work items provide additional context for your changes and ensures that updates go through your work item tracking process.



Configure a comment resolution policy for your branch by selecting Check for comment resolution.



Maintain a consistent branch history by enforcing a merge strategy when a pull request finishes. Select Enforce a merge strategy and pick an option to require that pull requests merge using that strategy.

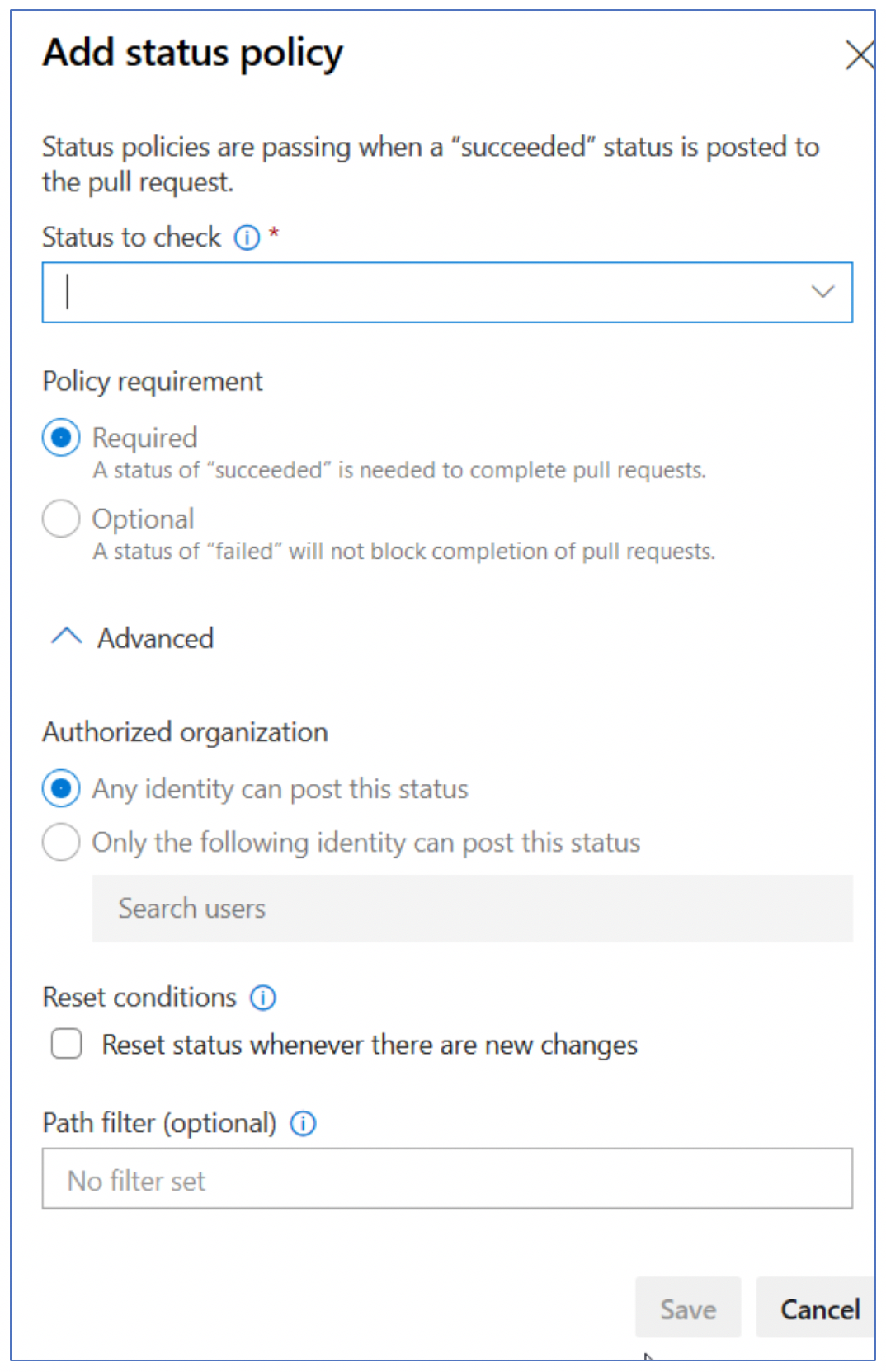
Set a policy requiring changes in a pull request to build successfully with the protected branch before the pull request can be completed.

Build policies reduce breaks and keep your test results passing.

Build policies help even if you’re using continuous integration (CI) on your development branches to catch problems early.

External services can use the PR Status API to post detailed status to your PRs.

The branch policy for additional services brings the ability for those third-party services to participate in the PR workflow and establish policy requirements.

Azure

Select reviewers for specific directories and files in your repo.

These reviewers are automatically added to pull requests that change files along those paths. You can also specify an Activity feed message.

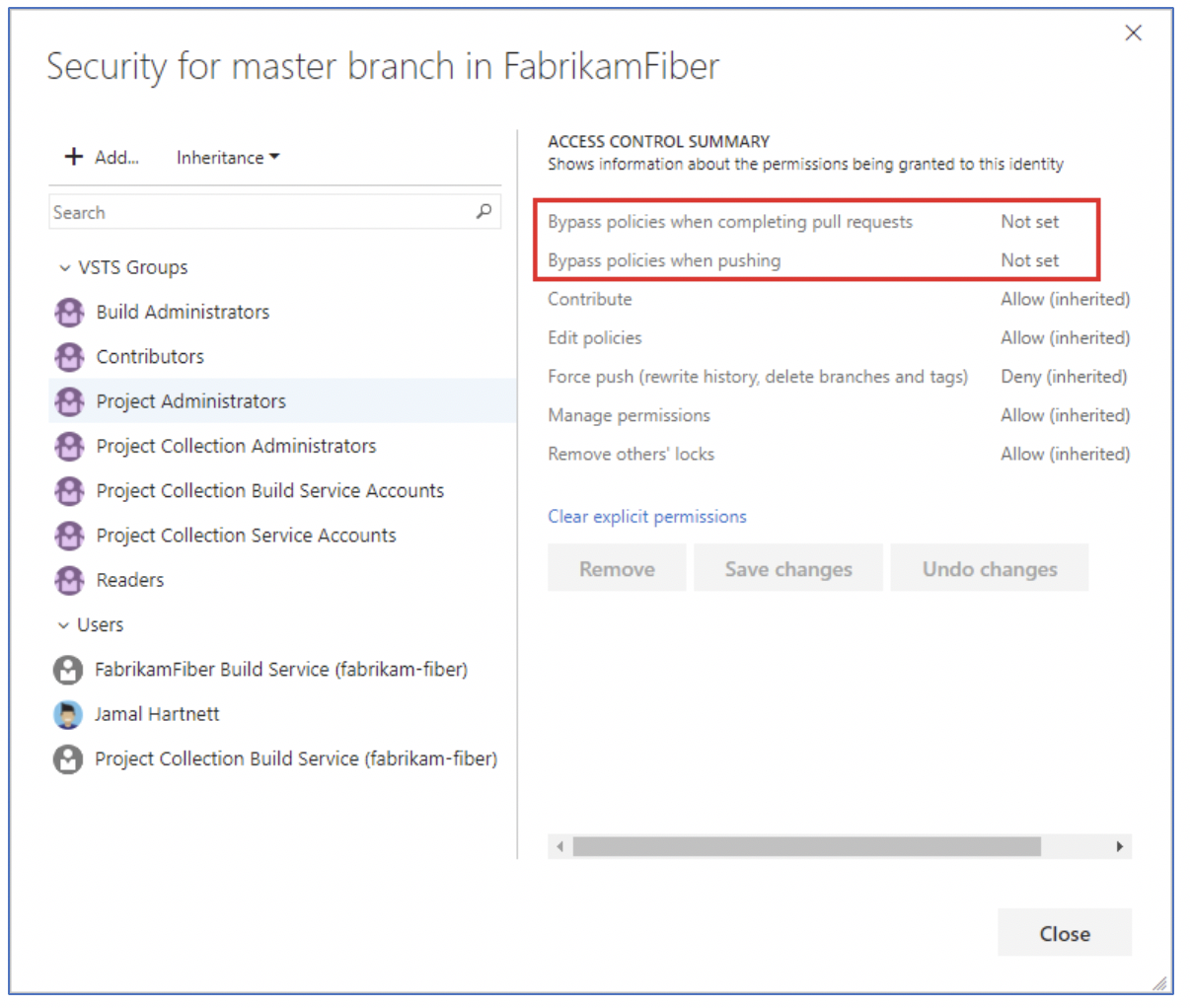
Bypass branch policies:

There are several permissions that allow users to bypass branch policy.

In Azure DevOps Services, the Exempt from policy enforcement permission does not exist. Instead, there are two new permissions:

*· Bypass policies when completing pull requests*

*· Bypass policies when pushing*



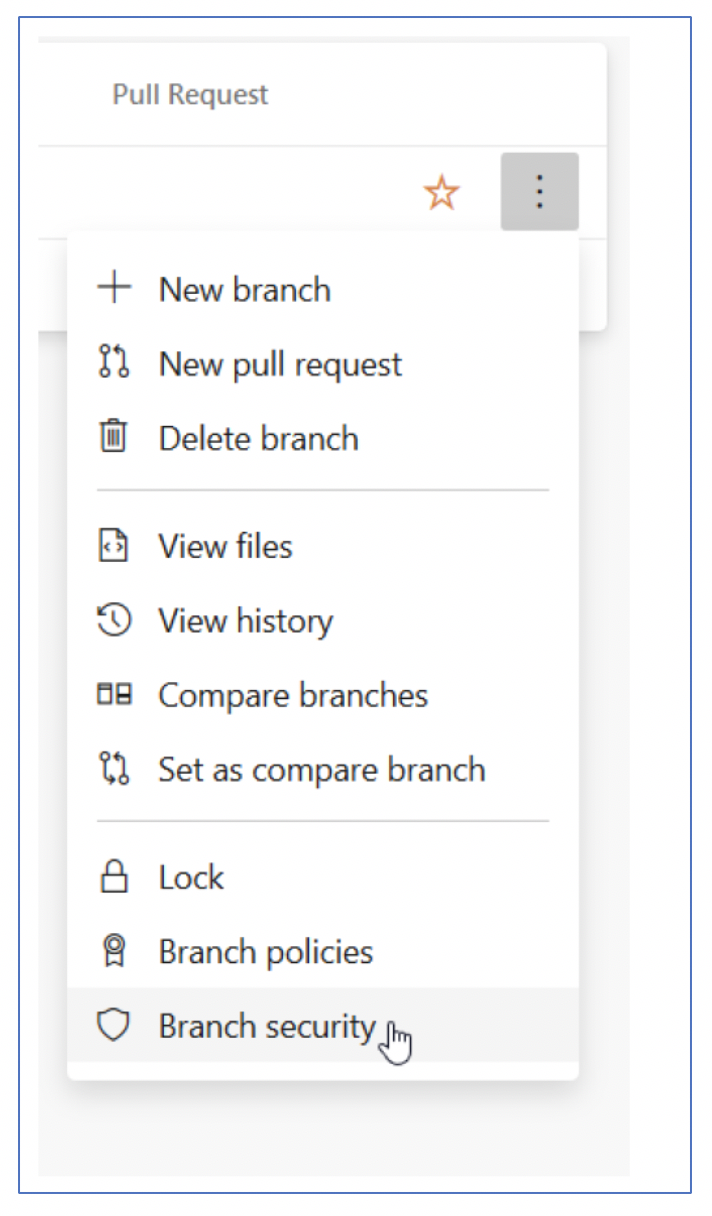
Users that previously had Exempt from policy enforcement enabled now have the two new permissions enabled instead.

In some cases, you need to bypass policy requirements. Bypassing lets you push changes to the branch directly or complete a pull request even if branch policies aren’t satisfied.

Branch Securities:

· Set up permissions to control who can read and update the code in a branch on your GlT repo.

· You can set permissions for individual users and groups and inherit and override permissions as needed from your repo permissions.



· Remove permissions for a user or group by selecting the user or Azure DevOps group, then selecting Remove.

· The user or group will still exist in your Project and this change will not affect other permissions for the user or group.

Branch Lock:

· Prevent updates to a Git branch by locking the branch. Locking a branch prevents other users from changing the existing commit history.

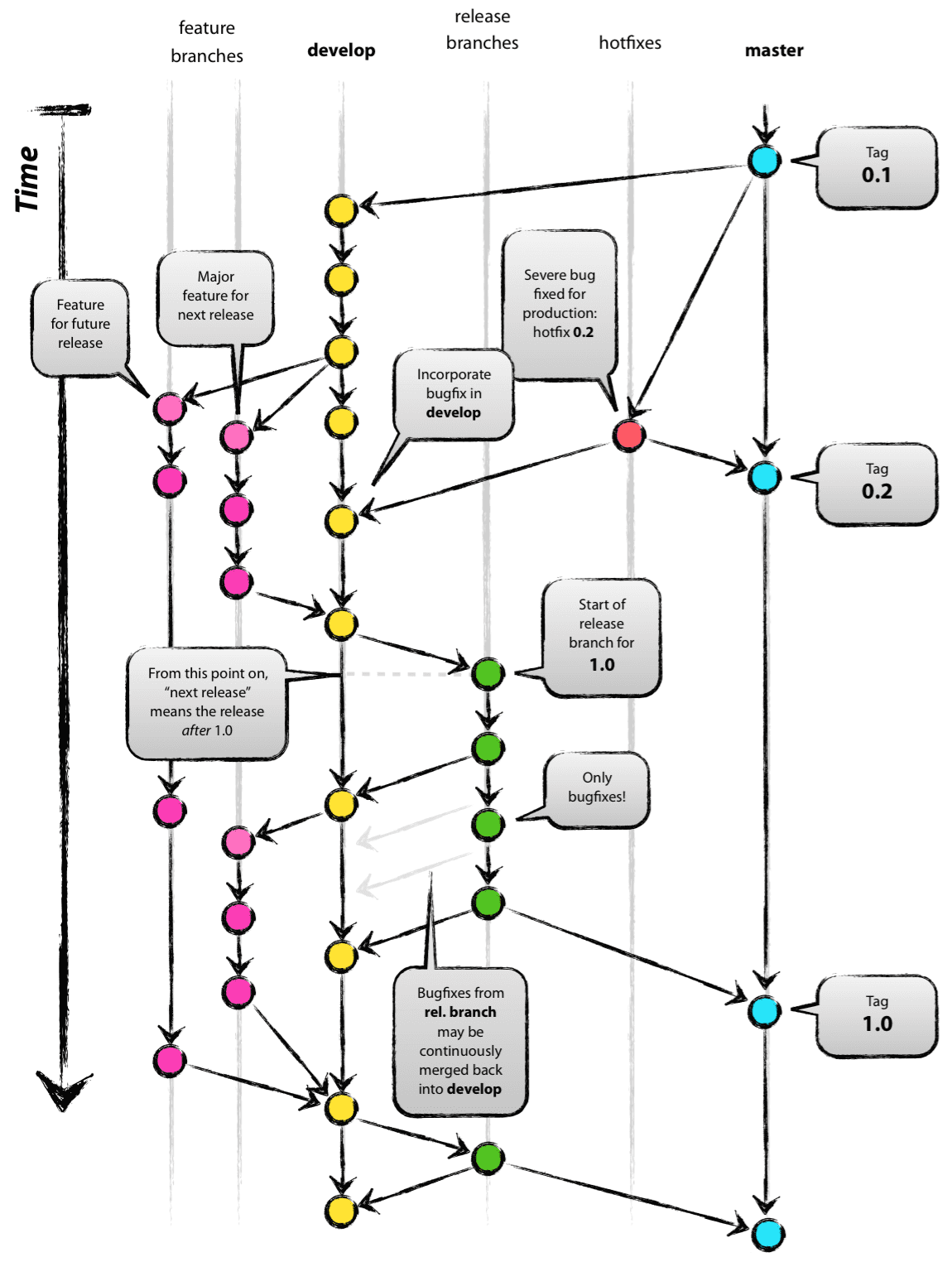
· Locking also blocks any new commits from being added to the branch by others.

· Locking is ideal for preventing new changes that might conflict with an important merge or to place a branch into a read-only state.

· Locking does not prevent cloning of a repo or fetching updates made in the branch into your local repo.

Compare Branches :

· The ahead of and behind numbers listed for each branch are in comparison with the branch currently labeled Compare on the Branches page. Update your compare branch to see how far ahead or behind your branches shown on the page are to another branch in your repo.

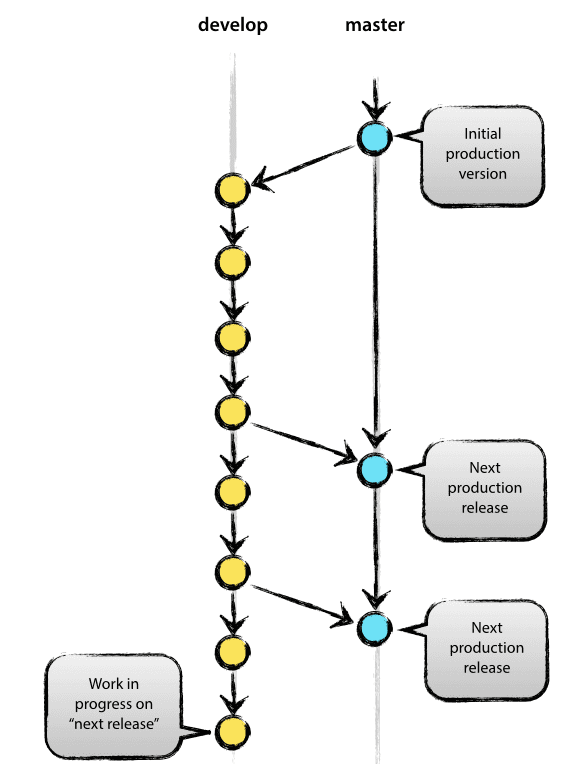
Successful Git branching model:

The main branches :

The central repo holds two main branches with an infinite lifetime:

* master
* develop

The master branch at origin should be familiar to every Git user. Parallel to the master branch, another branch exists called develop.



Supporting branches:

After, the main branches master and develop, our development model uses a variety of supporting branches for parallel development between team members, and tracking of features, production releases and to assist in quickly fixing live production problems. Unlike the main branches, these branches always have a limited lifetime(*short-lived*), since they will be removed after the work is completed.

The different types of branches we may use are:

* *Feature branches*
* *Release branches*
* *Hotfix branches*

