DevOps

Week 03

Murtaza Munawar Fazal

Git Branch Workflow

- When evaluating a workflow for your team, you must consider your team's culture. You want the workflow to enhance your team's effectiveness and not be a burden that limits productivity. Some things to consider when evaluating a Git workflow are:
  - Does this workflow scale with team size?
  - Is it easy to undo mistakes and errors with this workflow?
  - Does this workflow impose any new unnecessary cognitive overhead on the team?

Common Branch Workflow

- Most popular Git workflows will have some sort of centralized repo that individual developers will push and pull from.
- These comprehensive workflows offer more specialized patterns about managing branches for feature development, hotfixes, and eventual release.
- Following are two popular Git workflows.
  - Trunk-based development
  - Forking workflow

Trunk Based Development

- Trunk-based development is a logical extension of Centralized Workflow.
- The core idea behind the Feature Branch Workflow is that all feature development should take place in a dedicated branch instead of the main branch.
- This encapsulation makes it easy for multiple developers to work on a particular feature without disturbing the main codebase.
- It also means the main branch should never contain broken code, which is a huge advantage for continuous integration environments.

Trunk Based Development

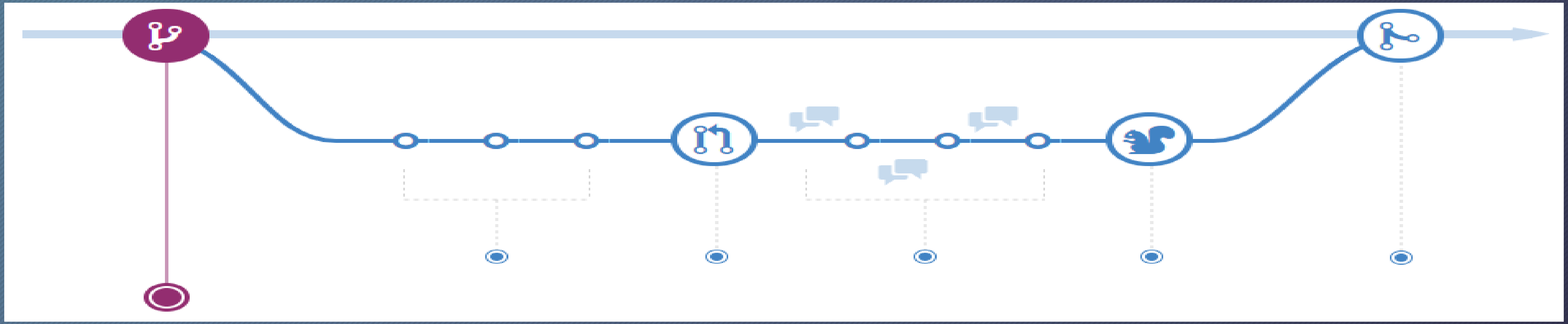
- Pull requests make it incredibly easy for your team to comment on each other's work. Also, feature branches can (and should) be pushed to the central repository. It allows sharing a feature with other developers without touching any official code.
- Since the main is the only "special" branch, storing several feature branches on the central repository doesn't pose any problems. It's also a convenient way to back up everybody's local commits.
- Feature branches should have descriptive names, like new-banner-images or bug-91. The idea is to give each branch a clear, highly focused purpose.
- Git makes no technical distinction between the main and feature branches, so developers can edit, stage, and commit changes to a feature branch.

Forking Workflow

- The Forking Workflow is fundamentally different than Trunk based development.
- Instead of using a single server-side repository to act as the "central" codebase, it gives every developer a server-side repository.
- It means that each contributor has two Git repositories:
  - A private local one.
  - A public server-side one.

Create a Branch

- When you're working on a project, you will have many different features or ideas in progress at any given time – some of which are ready to go and others that aren’t. Branching exists to help you manage this workflow.
- When you create a branch in your project, you're creating an environment where you can try out new ideas.
- Changes you make on a branch don't affect the main branch, so you're free to experiment and commit changes, safe in the knowledge that your branch won't be merged until it's ready to be reviewed by someone you're collaborating with.

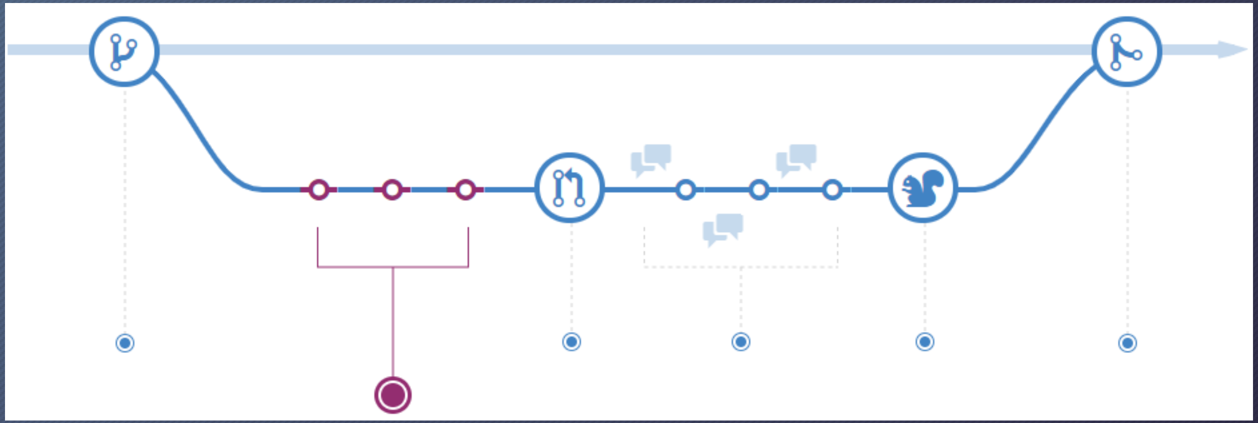


Create a Branch

- Branching is a core concept in Git, and the entire branch flow is based upon it. There's only one rule: anything in the main branch is always deployable.
- Because of this, your new branch must be created off the main when working on a feature or a fix.
- Your branch name should be descriptive (for example, refactor-authentication, user-content-cache-key, make-retina-avatars) so that others can see what is being worked on.

Add Commits

- Once your branch has been created, it's time to start making changes. Whenever you add, edit, or delete a file, you're making a commit and adding them to your branch. This process of adding commits keeps track of your progress as you work on a feature branch.
- Commits also create a transparent history of your work that others can follow to understand what you've done and why.
- Each commit has an associated commit message, which explains why a particular change was made.
- Furthermore, each commit is considered a separate unit of change. It lets you roll back changes if a bug is found or you decide to head in a different direction.
- Commit messages are essential, especially since Git tracks your changes and then displays them as commits once pushed to the server.
- By writing clear commit messages, you can make it easier for other people to follow along and provide feedback.



Open a Pull Request (PR)

- The Pull Requests start a discussion about your commits. Because they're tightly integrated with the underlying Git repository, anyone can see exactly what changes would be merged if they accept your request.
- You can open a Pull Request at any point during the development process when:
    - You've little or no code but want to share some screenshots or general ideas.
    - You're stuck and need help or advice.
    - You're ready for someone to review your work.
- In your Pull Request message, you can ask for feedback from specific people or teams, whether they're down the hall or 10 time zones away.
- Pull Requests help contribute to projects and for managing changes to shared repositories.
- If you're using a Fork & Pull Model, Pull Requests provide a way to notify project maintainers about the changes you'd like them to consider.
- If you're using a Shared Repository Model, Pull Requests help start code review and conversation about proposed changes before they're merged into the main branch.

Merge

- Once your changes have been verified, it's time to merge your code into the main branch.
- Once merged, Pull Requests preserve a record of the historical changes to your code. Because they're searchable, they let anyone go back in time to understand why and how a decision was made.
- By incorporating specific keywords into the text of your Pull Request, you can associate issues with code. When your Pull Request is merged, the related issues can also close.
- This workflow helps organize and track branches focused on business domain feature sets.

Collaborate with Pull Request

- Pull requests let you tell others about changes you've pushed to a GitHub repository.
- Once a pull request is sent, interested parties can review the set of changes, discuss potential modifications, and even push follow-up commits if necessary.
- Pull requests are commonly used by teams and organizations collaborating using the Shared Repository Model.
- Many open-source projects on GitHub use pull requests to manage changes from contributors.
- They help provide a way to notify project maintainers about changes one has made.

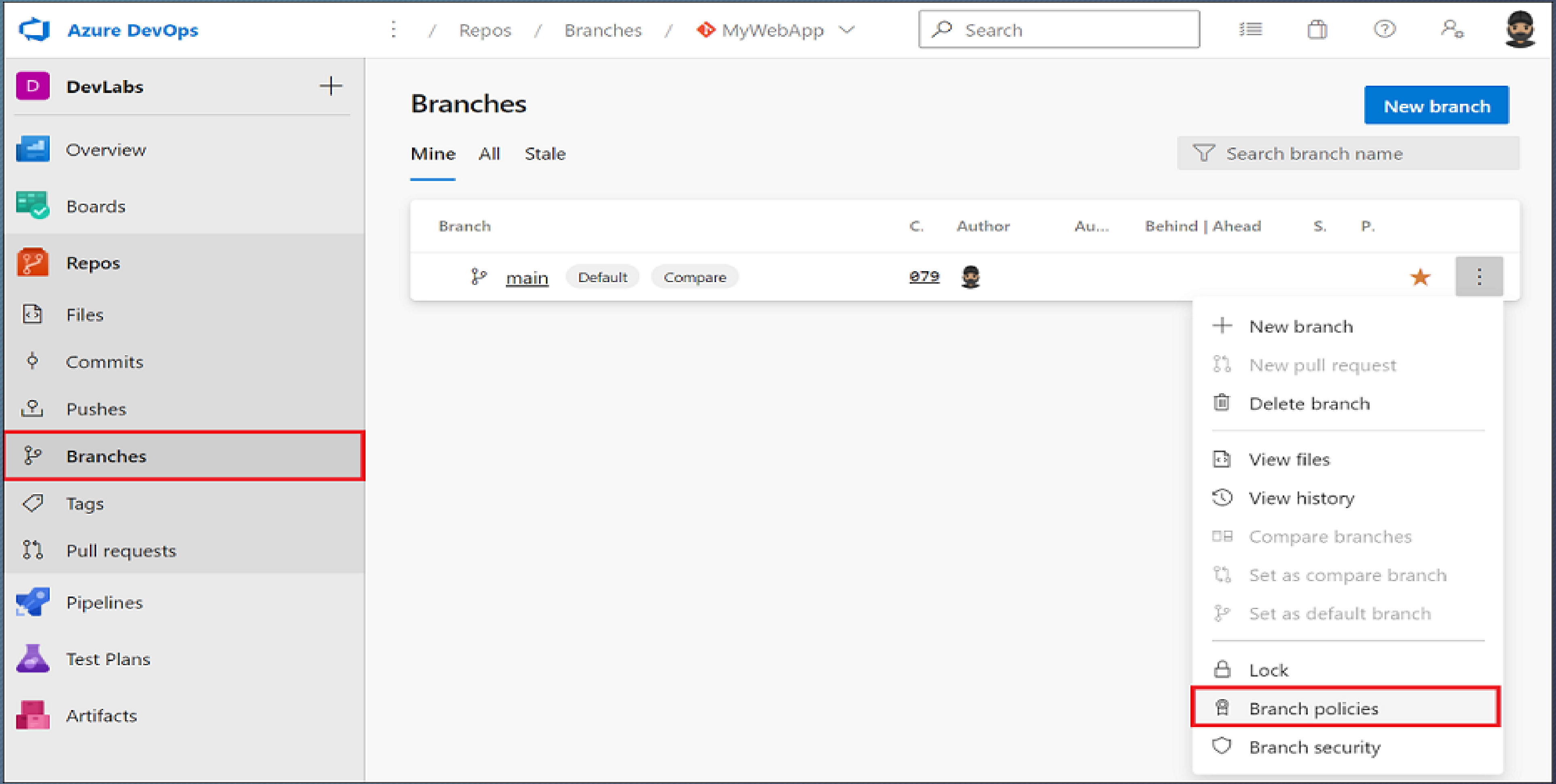
Collaborate with Pull Request

- Also, start code review and general discussion about a set of changes before being merged into the main branch.
- Pull requests combine the review and merge of your code into a single collaborative process.
- Once you're done fixing a bug or new feature in a branch, create a new pull request.
- Add the team members to the pull request so they can review and vote on your changes.
- Use pull requests to review works in progress and get early feedback on changes.
- There's no commitment to merge the changes as the owner can abandon the pull request at any time.

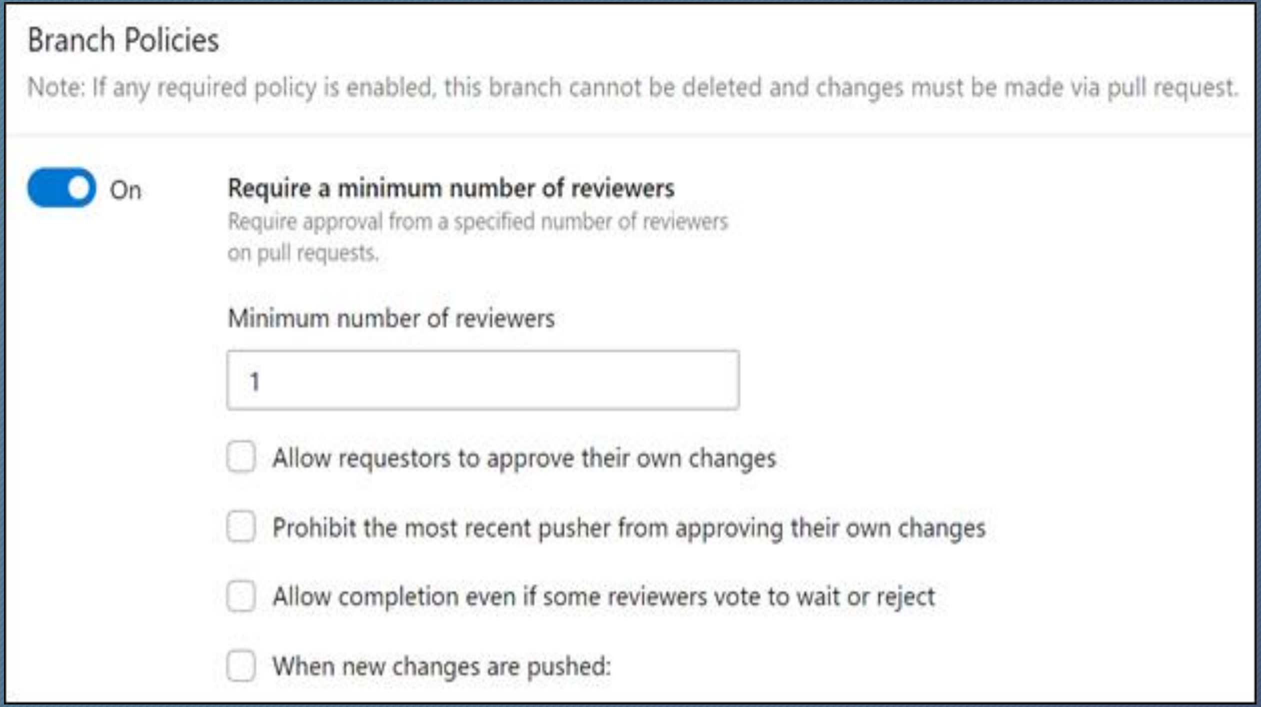
Protect Branch with policies

- There are a few critical branches in your repo that the team relies on always in suitable shapes, such as your main branch.
- Require pull requests to make any changes on these branches. Developers pushing changes directly to the protected branches will have their pushes rejected.
- Add more conditions to your pull requests to enforce a higher level of code quality in your key branches.
- A clean build of the merged code and approval from multiple reviewers are extra requirements you can set to protect your key branches.

Branch Policies in Azure DevOps



Branch Policies in Azure DevOps


Branch Policies

Note: If any required policy is enabled, this branch cannot be deleted and changes must be made via pull request.

On

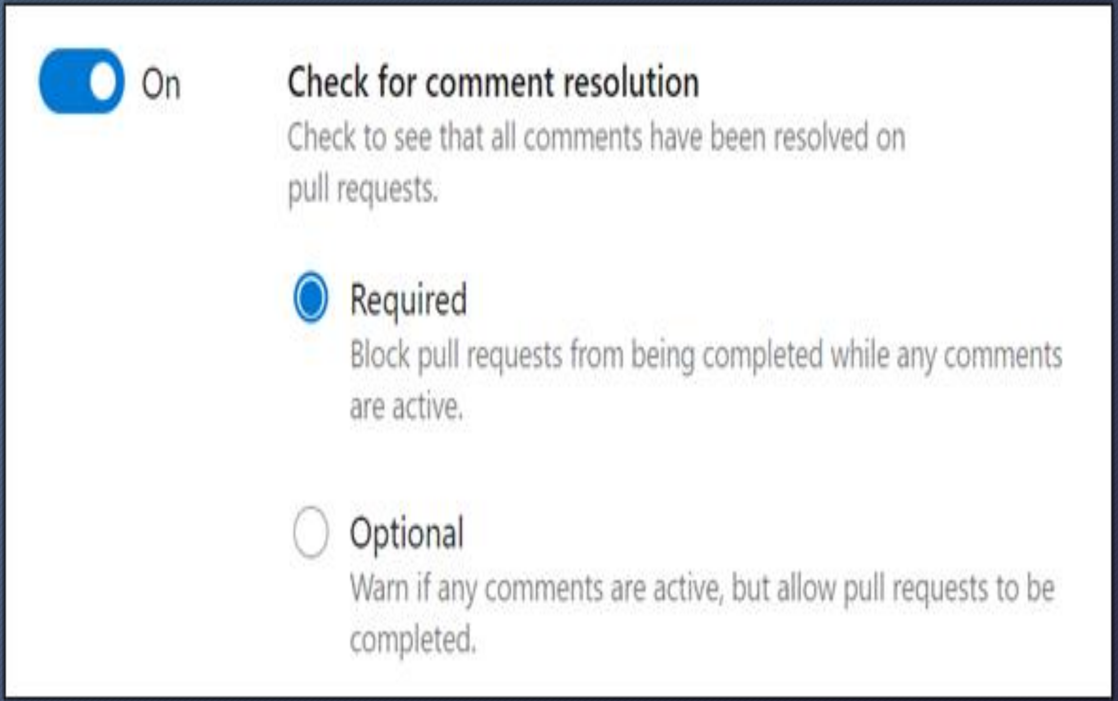
Require a minimum number of reviewers

Require approval from a specified number of reviewers on pull requests.

Minimum number of reviewers

1

- [ ] Allow requestors to approve their own changes
- [ ] Prohibit the most recent pusher from approving their own changes
- [ ] Allow completion even if some reviewers vote to wait or reject
- [ ] When new changes are pushed:


On

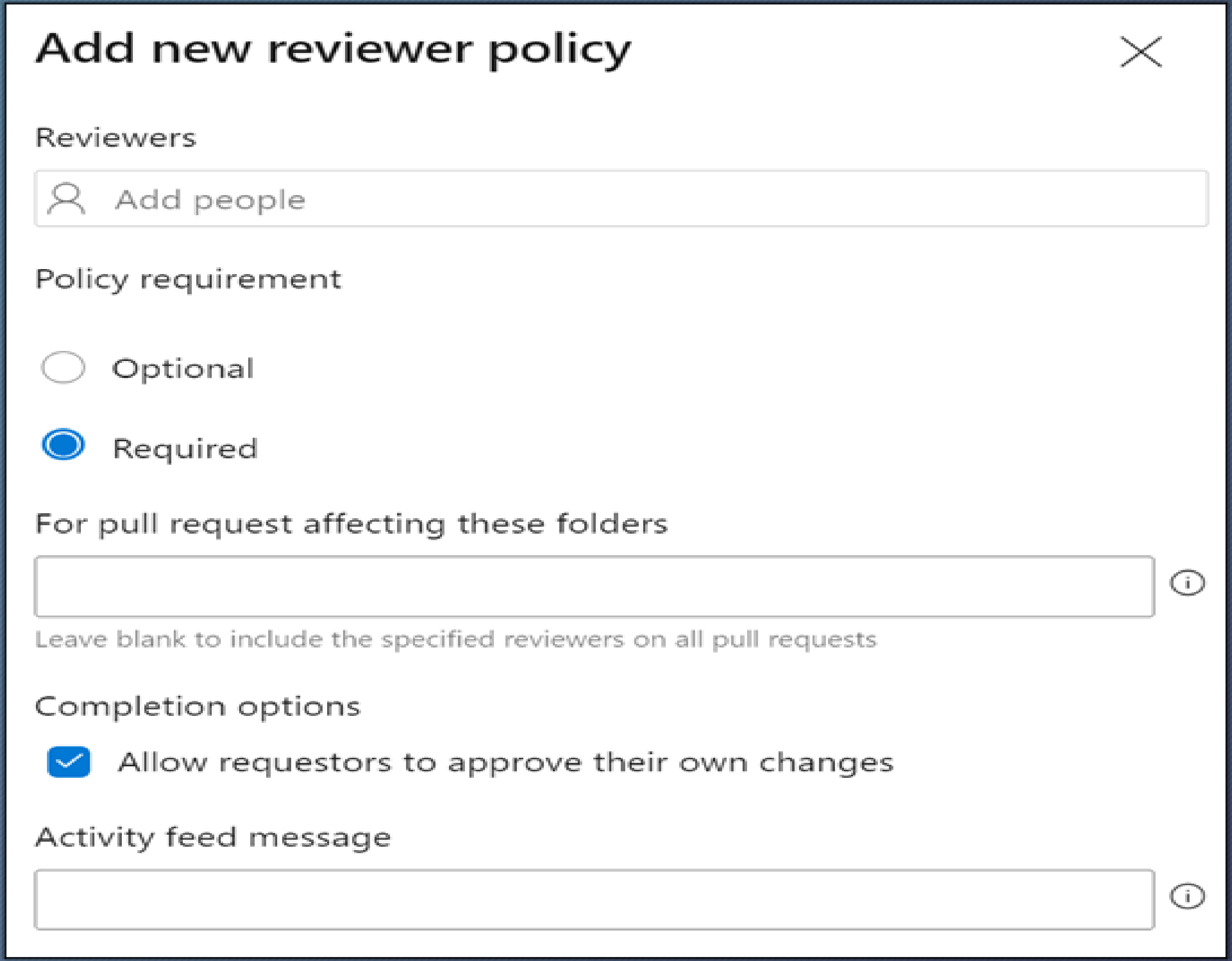
Check for comment resolution

Check to see that all comments have been resolved on pull requests.

[✓ Required]
Block pull requests from being completed while any comments are active.

Optional
Warn if any comments are active, but allow pull requests to be completed.

Branch Policies in Azure DevOps



Add new reviewer policy

Reviewers

Add people

Policy requirement

( ) Optional

(●) Required

For pull request affecting these folders

Leave blank to include the specified reviewers on all pull requests

Completion options

[✓] Allow requestors to approve their own changes

Activity feed message

Shallow clone

- If developers don't need all the available history in their local repositories, a good option is to implement a shallow clone.
- It saves both space on local development systems and the time it takes to sync.
- You can specify the depth of the clone that you want to execute:
  - git clone --depth [depth] [clone-url]

Purge Repository data

- While one of the benefits of Git is its ability to hold long histories for repositories efficiently, there are times when you need to purge data.
- The most common situations are where you want to:
  - Significantly reduce the size of a repository by removing history.
  - Remove a large file that was accidentally uploaded.
  - Remove a sensitive file that shouldn't have been uploaded.

Git filter-repo tool

- The git filter-repo is a tool for rewriting history.
- Its core filter-repo contains a library for creating history rewriting tools. Users with specialized needs can quickly create entirely new history rewriting tools.

BFG Repo-Cleaner

- BFG Repo-Cleaner is a commonly used open-source tool for deleting or "fixing" content in repositories. It's easier to use than the git filter-branch command. For a single file or set of files, use the --delete-files option:
    - $ bfg --delete-files file_I_should_not_have_committed

Git Hooks

- Git hooks are a mechanism that allows code to be run before or after certain Git lifecycle events.
- For example, one could hook into the commit-msg event to validate that the commit message structure follows the recommended format.
- The hooks can be any executable code, including shell, PowerShell, Python, or other scripts. Or they may be a binary executable. Anything goes!
- The only criteria are that hooks must be stored in the .git/hooks folder in the repo root. Also, they must be named to match the related events (Git 2.x):

- applypatch-msg
- pre-applypatch
- post-applypatch
- pre-commit
- prepare-commit-msg
- commit-msg
- post-commit
- pre-rebase
- post-checkout
- post-merge
- pre-receive
- update
- post-receive
- post-update
- pre-auto-gc
- post-rewrite
- pre-push

Examples for Git Hooks

- Some examples of where you can use hooks to enforce policies, ensure consistency, and control your environment:
  - In Enforcing preconditions for merging
  - Verifying work Item ID association in your commit message
  - Preventing you & your team from committing faulty code
  - Sending notifications to your team's chat room (Teams, Slack, HipChat, etc.)

Server-side Hooks

- Azure Repos also exposes server-side hooks. Azure DevOps uses the exact mechanism itself to create Pull requests

Examine Code Quality

- How do we measure code quality?
- The quality of code shouldn't be measured subjectively. A developer-writing code would rate the quality of their code high, but that isn't a great way to measure code quality. Different teams may use different definitions based on context.
- Code that is considered high quality may mean one thing for an automotive developer. And it may mean another for a web application developer.
- The quality of the code is essential, as it impacts the overall software quality.

Reliability

- Reliability measures the probability that a system will run without failure over a specific period of operation. It relates to the number of defects and availability of the software. Several defects can be measured by running a static analysis tool.
- Software availability can be measured using the mean time between failures (MTBF).
- Low defect counts are crucial for developing a reliable codebase.

Maintainability

- Maintainability measures how easily software can be maintained. It relates to the codebase's size, consistency, structure, and complexity. And ensuring maintainable source code relies on several factors, such as testability and understandability.
- You can't use a single metric to ensure maintainability.
- Both automation and human reviewers are essential for developing maintainable codebases.

Testability

- Testability measures how well the software supports testing efforts. It relies on how well you can control, observe, isolate, and automate testing, among other factors.
- Testability can be measured based on how many test cases you need to find potential faults in the system.
- The size and complexity of the software can impact testability.
- So, applying methods at the code level—such as cyclomatic complexity—can help you improve the testability of the component.

Portability

- Portability measures how usable the same software is in different environments. It relates to platform independence.
- There isn't a specific measure of portability. But there are several ways you can ensure portable code.
- It's essential to regularly test code on different platforms rather than waiting until the end of development.
- It's also good to set your compiler warning levels as high as possible and use at least two compilers.
- Enforcing a coding standard also helps with portability.

Reusability

- Reusability measures whether existing assets—such as code—can be used again.
- Assets are more easily reused if they have modularity or loose coupling characteristics.
- The number of interdependencies can measure reusability.
- Running a static analyzer can help you identify these interdependencies.

Common Quality Related Metrics

- One of the promises of DevOps is to deliver software both faster and with higher quality. Previously, these two metrics have been almost opposites. The more quickly you went, the lower the quality. The higher the quality, the longer it took. But DevOps processes can help you find problems earlier, which usually means that they take less time to fix.
- The following is a list of metrics that directly relate to the quality of the code being produced and the build and deployment processes.
  - **Failed builds percentage** - Overall, what percentage of builds are failing?
  - **Failed deployments percentage** - Overall, what percentage of deployments are failing?
  - **Ticket volume** - What is the overall volume of customer or bug tickets?
  - **Bug bounce percentage** - What percentage of customer or bug tickets are reopened?
  - **Unplanned work percentage** - What percentage of the overall work is unplanned?

Technical Debt

- **Technical debt** is a term that describes the future cost that will be incurred by choosing an easy solution today instead of using better practices because they would take longer to complete.
- Technical debt can build up to the point where developers spend almost all their time sorting out problems and doing rework, either planned or unplanned, rather than adding value.
- Example:
  - When developers are forced to create code quickly, they'll often take shortcuts. For example, instead of refactoring a method to include new functionality, let us copy to create a new version. Then I only test my new code and can avoid the level of testing required if I change the original method because other parts of the code use it.
  - Now we have two copies of the same code that we need to modify in the future instead of one, and we run the risk of the logic diverging. There are many causes. For example, there might be a lack of technical skills and maturity among the developers or no clear product ownership or direction.

Reasons for Technical Debt

- Lack of coding style and standards.
- Lack of or poor design of unit test cases.
- Ignoring or not understanding object-oriented design principles.
- Monolithic classes and code libraries.
- Poorly envisioned the use of technology, architecture, and approach. (Forgetting that all system attributes, affecting maintenance, user experience, scalability, and others, need to be considered).
- Over-engineering code (adding or creating code that isn't required, adding custom code when existing libraries are sufficient, or creating layers or components that aren't needed).
- Insufficient comments and documentation.
- Not writing self-documenting code (including class, method, and variable names that are descriptive or indicate intent).
- Taking shortcuts to meet deadlines.
- Leaving dead code in place.