

AZ-400: Development for enterprise DevOps

Structure your Git Repo

Introduction

Completed

- 1 minute

As a version control system, Git is easy to get started with but challenging to master.

While there's no one way to implement Git in the right way, many techniques can help you scale the implementation of Git across the organization.

Simple things like structuring your code into micro repos, selecting a lean branching and merging model, and using pull requests for code review can make your teams more productive.

This module examines Git repositories structure, explains the differences between mono versus multiple repos, and helps you create a changelog.

Learning objectives

After completing this module, students and professionals can:

- Understand Git repositories.
- Implement mono repo or multiple repos.
- Explain how to structure Git Repos.
- Implement a changelog.

Prerequisites

- Understanding of what DevOps is and its concepts.
- Familiarity with version control principles is helpful but isn't necessary.
- Beneficial to have experience in an organization that delivers software.

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Explore monorepo versus multiple repos

Completed

- 3 minutes

A repository is where your work history is stored, usually in a git subdirectory.

How should you organize your code repository? Development teams aim to separate concerns in their software and repositories. As time passes, it isn't unusual for code repositories to become cluttered with irrelevant code and artifacts.

When it comes to organizing your repositories, there are two main philosophies: using a single repository (Monorepo) or multiple repositories.

- Monorepos is a source control pattern where all source code is kept in one repository. It's easy to give all employees access to everything at once. Clone it, and you're done.
- Organizing your projects into separate repositories is referred to as multiple repositories.

The fundamental difference between mono repo and multiple repo philosophies is what enables teams to work together most efficiently. In an extreme scenario, the multiple repos view suggests that each subteam can work in its repository. It allows them to work in their respective areas using the libraries, tools, and development workflows that optimize their productivity.

The cost of consuming anything not developed within a given repository is equivalent to using a third-party library or service, even if it was written by someone sitting nearby.

If you come across a bug in your library, you should address it in the corresponding repository. Once you have published a new artifact, you can return to your repository and make the necessary code changes. However, if the bug is in a different code base or involves different libraries, tools, or workflows, you may need to seek assistance from the owner of that system and wait for their response.

When using the mono repo view, managing complex dependency graphs can increase the difficulty of using a single repository. The benefits of allowing different teams to work independently aren't substantial. Some teams may find an efficient way of working, but this may not be true for all groups. Furthermore, other teams may choose a suboptimal approach, negating any benefits gained by others. Consolidating all your work in a mono repo lets you focus on closely monitoring this single repository.

The hassle of making changes in other repos or waiting for teams to make changes for you is avoided in a mono repo where anyone can change anything.

If you discover a bug in a library, fixing it's as easy as finding a bug in your own code.

Note

In Azure DevOps, it's common to use a separate repository for each associated solution within a project.

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Implement a change log

Completed

- 3 minutes

The concept of a changelog is simple enough: It's a file that has a list of changes made to a project, usually in date order. The typical breakdown is to separate a list of versions, and then within each version, show:

- Added features
- Modified/Improved features
- Deleted features

Some teams will post changelogs as blog posts; others will create a CHANGELOG.md file in a GitHub repository.

Automated change log tooling

While changelogs can be created and manually maintained, you might want to consider using an automated changelog creation tool. At least as a starting point.

Using native GitHub commands

The git log command can be useful for automatically creating content. Example: create a new section per version:

```
git log [options] vX.X.X..vX.X.Y | helper-script > projectchangelogs/X.X.Y
```

Git changelog

One standard tool is [gitchangelog](#). This tool is based on Python.

GitHub changelog generator

Another standard tool is called [github-changelog-generator](#).

```
$ github_changelog_generator -u github-changelog-generator -p TimerTrend-3.0
```

This tool is based on Gem.

Should you use autogenerated log-based data?

Preference is always to avoid dumping log entries into a changelog. Logs are "noisy," so it's easy to generate a mess that isn't helpful.

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Knowledge check

Completed

- 6 minutes

Choose the best response for each question. Then select **Check your answers** .

Check your knowledge

1.

Which of the following choices isn't a repository strategy?

☐

Monorepo.

☐

Multiple Repo.

☐

Feature Repo.

2.

Which of the following choices is the fundamental difference between the monorepo and multiple repos philosophies?

☐

Allow teams to have their own repositories to develop.

☐

Allow teams working together on a system to go faster.

☐

Avoid teams losing their changes.

3.

Which of the following choices is a common tool to create a changelog?

☐

gitchangelog.

☐

gitlogchange.

☐

gitloggenerator.

Check your answers

You must answer all questions before checking your work.

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Summary

Completed

- 1 minute

This module examined Git repositories structure, explained the differences between mono versus multiple repos, and helped you create a changelog.

You learned how to describe the benefits and usage of:

- Understand Git repositories.
- Implement mono repo or multiple repos.
- Explain how to structure Git Repos.
- Implement a changelog.

Learn more

- [Understand source control - Azure DevOps.](#)
- [Build Azure Repos Git repositories - Azure Pipelines | Microsoft Docs.](#)
- [Check out multiple repositories in your pipeline - Azure Pipelines | Microsoft Docs.](#)

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Manage Git branches and workflows

Introduction

Completed

- 1 minute

This module explores Git branching types, concepts, and models for the continuous delivery process. It helps companies define their branching strategy and organization.

Learning objectives

After completing this module, students and professionals can:

- Describe Git branching workflows.
- Implement feature branches.
- Fork a repo.

Prerequisites

- Understanding of what DevOps is and its concepts.
- Familiarity with version control principles is helpful but isn't necessary.
- Beneficial to have experience in an organization that delivers software.

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Explore branch workflow types

Completed

- 4 minutes

What is a successful 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 workflows

Most popular Git workflows will have some sort of centralized repo that individual developers will push and pull from.

Below is a list of some popular Git workflows that we'll go into more detail about in the next section.

These comprehensive workflows offer more specialized patterns about managing branches for feature development, hotfixes, and eventual release.

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.

Forking workflow

The Forking Workflow is fundamentally different than the other workflows discussed in this tutorial.

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.

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

[Explore feature branch workflow](#)

Completed

- 4 minutes

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.

The encapsulation makes it easy for multiple developers to work on a particular feature without disturbing the main codebase. It also means the main branch will never contain broken code, a huge advantage for continuous integration environments.

Encapsulating feature development also makes it possible to use pull requests, which are a way to start discussions around a branch. They allow other developers to sign out on a feature before it integrates into the official project. Or, if you get stuck in the middle of a feature, you can open a pull request asking for suggestions from your colleagues.

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.

Trunk-based development workflow

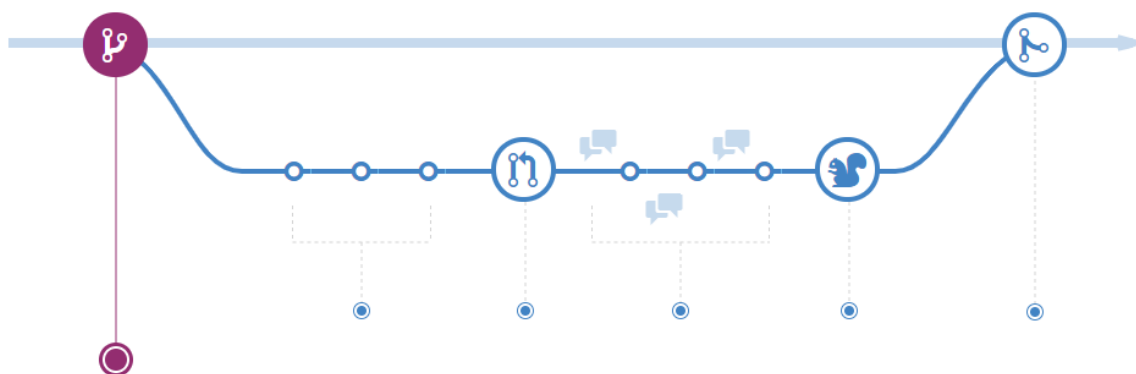
The trunk-based development Workflow assumes a central repository, and the main represents the official project history.

Instead of committing directly to their local main branch, developers create a new branch whenever they start working on a new feature.

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.

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 create an environment where you can try out new ideas.

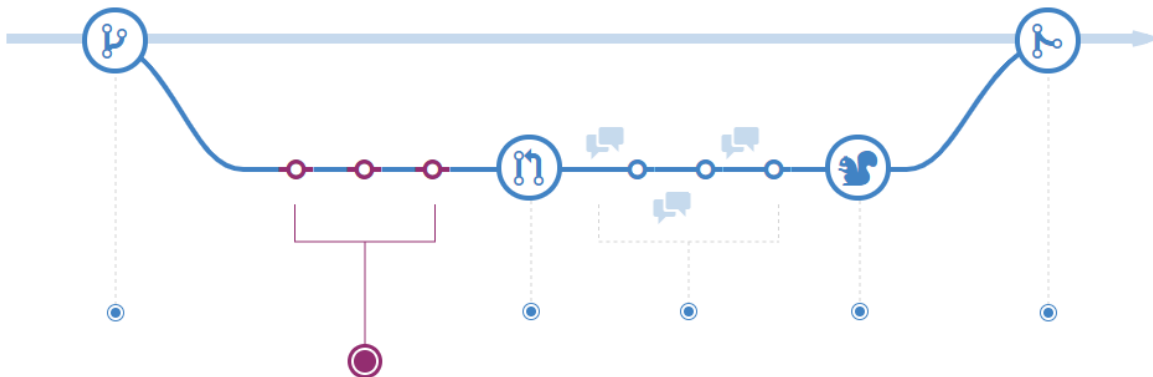
Changes you make on a branch don't affect the main branch, so you're free to experiment and commit changes, safe in the knowledge that your branch won't be merged until it's ready to be reviewed by someone you're collaborating with.

Branching is a core concept in Git; the entire branch flow is based upon it. 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 others can see what is being worked on.

Add commits



Once your branch has been created, it's time to make changes. Whenever you add, edit, or delete a file, you make a commit and add them to your branch.

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 your actions and why.

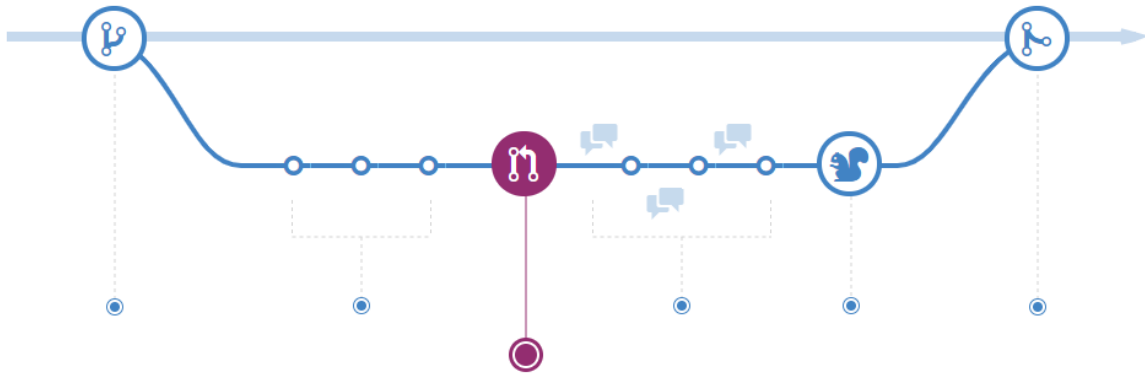
Each commit has an associated commit message explaining why a particular change was made.

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 displays them as commits once pushed to the server.

By writing clear commit messages, you can make it easier for others to follow along and provide feedback.

Open a pull request



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 screenshots or general ideas.
- You're stuck and need help or advice.
- You're ready for someone to review your work.

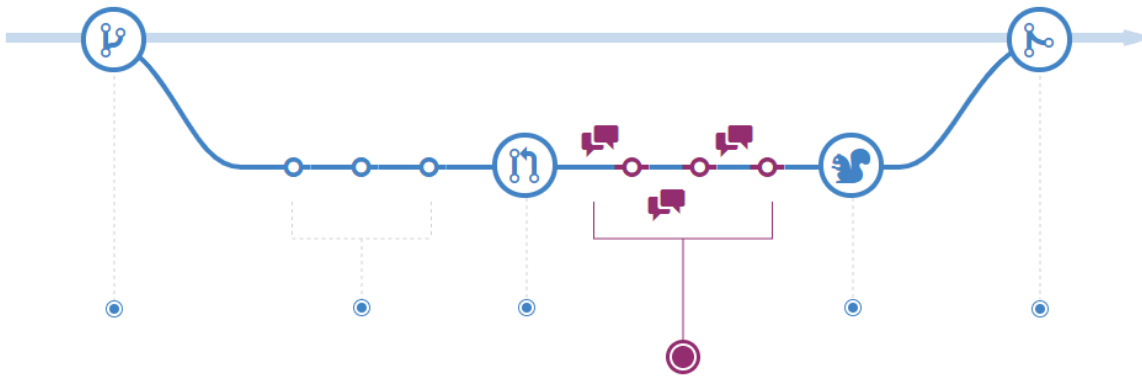
Using the @mention system in your Pull Request message, you can ask for feedback from specific people or teams, whether they're down the hall or 10 time zones away.

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.

Discuss and review your code



Once a Pull Request has been opened, the person or team reviewing your changes may have questions or comments.

Perhaps the coding style doesn't match project guidelines, the change is missing unit tests, everything looks excellent, and the props are in order.

Pull Requests are designed to encourage and capture this type of conversation.

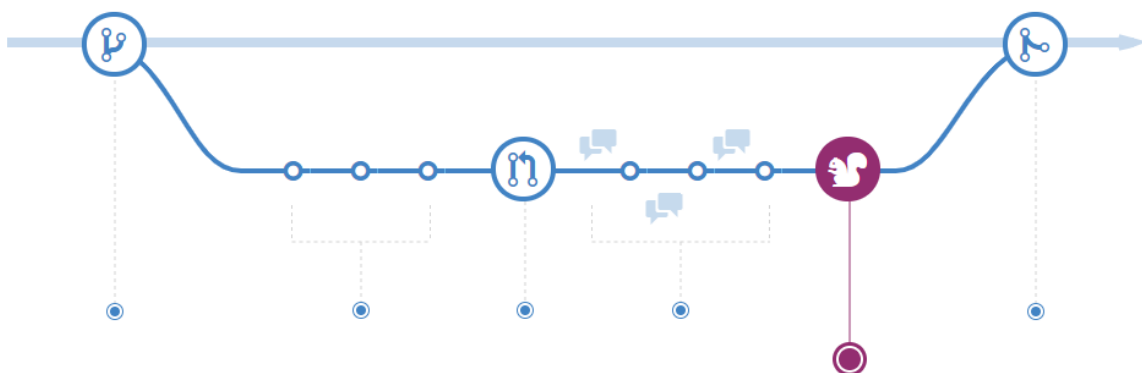
You can also continue to push to your branch, considering discussion and feedback about your commits.

Suppose someone comments that you forgot to do something, or if there's a bug in the code, you can fix it in your branch and push up the change.

Git will show your new commits and any feedback you may receive in the unified Pull Request view.

Pull Request comments are written in Markdown, so you can embed images and emojis, use pre-formatted text blocks, and other lightweight formatting.

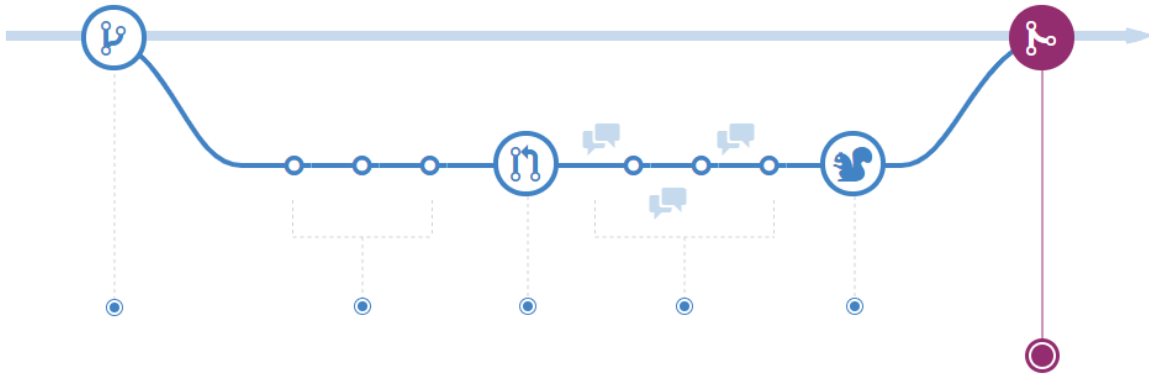
Deploy



With Git, you can deploy from a branch for final testing in an environment before merging to the main.

Once your pull request has been reviewed and the branch passes your tests, you can deploy your changes to verify them. You can roll it back if your branch causes issues by deploying the existing main.

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.

You can associate issues with code by incorporating specific keywords into your Pull Request text. 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.

Other Git workflows, like the Git Forking Workflow and the Gitflow Workflow, are repo-focused and can use the Git Feature Branch Workflow to manage their branching models.

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

[Explore Git branch model for continuous delivery](#)

Completed

- 3 minutes

The purpose of writing code is to ship enhancements to your software.

A branching model that introduces too much process overhead doesn't help increase the speed of getting changes to customers—developing a branching model gives you enough padding not to ship poor-quality changes. But, at the same time doesn't introduce too many processes to slow you down.

The internet is full of branching strategies for Git; while there's no right or wrong, a perfect branching strategy works for your team!

You'll learn always to use the combination of feature branches and pull requests to have a ready-to-ship main branch.

Getting ready

Let's cover the principles of what we suggest:

- The main branch:
 - The main branch is the only way to release anything to production.
 - The main branch should always be in a ready-to-release state.
 - Protect the main branch with branch policies.
 - Any changes to the main branch flow through pull requests only.
 - Tag all releases in the main branch with Git tags.
- The feature branch:
 - Use feature branches for all new features and bug fixes.
 - Use feature flags to manage long-running feature branches.
 - Changes from feature branches to the main only flow through pull requests.
 - Name your feature to reflect its purpose.

List of branches:

```
bugfix/description
features/feature-name
features/feature-area/feature-name
hotfix/description
users/username/description
users/username/workitem
```

- Pull requests:
 - Review and merge code with pull requests.
 - Automate what you inspect and validate as part of pull requests.
 - Tracks pull request completion duration and set goals to reduce the time it takes.

We'll be using the myWebApp created in the previous exercises. See [Describe working with Git locally](#).

In this recipe, we'll be using three trendy extensions from the marketplace:

- [Azure CLI](#): is a command-line interface for Azure.
- [Azure DevOps CLI](#): It's an extension of the Azure CLI for working with Azure DevOps and Azure DevOps Server designed to seamlessly integrate with Git, CI pipelines, and Agile tools. With the Azure DevOps CLI, you can contribute to your projects without leaving the command line. CLI runs on Windows, Linux, and Mac.

- [Git Pull Request Merge Conflict](#) : This open-source extension created by Microsoft DevLabs allows you to review and resolve the pull request merge conflicts on the web. Any conflicts with the target branch must be resolved before a Git pull request can complete. With this extension, you can resolve these conflicts on the web as part of the pull request merge instead of doing the merge and resolving conflicts in a local clone.

The Azure DevOps CLI supports returning the query results in JSON, JSONC, YAML, YAMLC, table, TSV, and none. You can configure your preference by using the configure command.

How to do it

Important

You need to have the project created in the first Learning Path: [Describe working with Git locally](#).

1. After you've cloned the main branch into a local repository, create a new feature branch, myFeature-1:

myWebApp

```
git checkout -b feature/myFeature-1
```

Output:

Switched to a new branch 'feature/myFeature-1'.

2. Run the Git branch command to see all the branches. The branch showing up with an asterisk is the "currently-checked-out" branch:

myWebApp

```
git branch
```

Output:

feature/myFeature-1

main

3. Make a change to the Program.cs file in the feature/myFeature-1 branch:

myWebApp

```
notepad Program.cs
```

4. Stage your changes and commit locally, then publish your branch to remote:

myWebApp

```
git status
```


Output:

On branch feature/myFeature-1 Changes not staged for commit: (use "git add <file>..." to update what will be committed) (use "git checkout -- <file>..." to discard changes in working directory) modified: Program.cs.

myWebApp

```
git add .
git commit -m "Feature 1 added to Program.cs"
```

Output:

[feature/myFeature-1 70f67b2] feature 1 added to program.cs 1 file changed, 1 insertion(+).

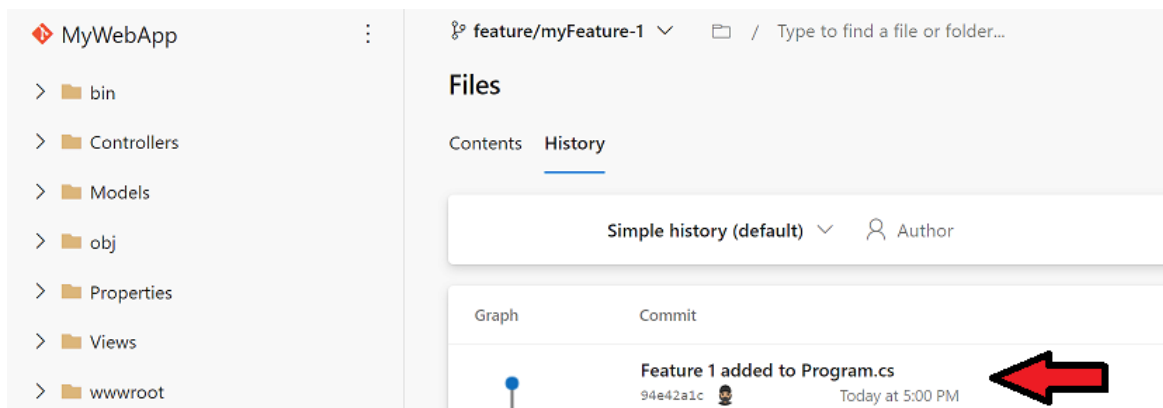
myWebApp

```
git push -u origin feature/myFeature-1
```

Output:

*Delta compression using up to 8 threads. Compressing objects: 100% (3/3), done. Writing objects: 100% (3/3), 348 bytes | 348.00 KiB/s, done. Total 3 (delta 2), reused 0 (delta 0) remote: Analyzing objects... (3/3) (10 ms) remote: Storing packfile... done (44 ms) remote: Storing index... done (62 ms) To https://dev.azure.com/Geeks/PartsUnlimited/_git/MyWebApp * [new branch] feature/myFeature-1 -> feature/myFeature-1 Branch feature/myFeature-1 set up to track remote branch feature/myFeature-1 from origin.*

The remote shows the history of the changes:



5. Configure Azure DevOps CLI for your organization and project. Replace **organization** and **project name** :

```
az devops configure --defaults organization=https://dev.azure.com/organization
project="project name"
```

6. Create a new pull request (using the Azure DevOps CLI) to review the changes in the feature-1 branch:

```
az repos pr create --title "Review Feature-1 before merging to main" --work-items 38 39 `
--description "#Merge feature-1 to main" `
--source-branch feature/myFeature-1 --target-branch main `
--repository myWebApp --open
```

Use the `--open` switch when raising the pull request to open it in a web browser after it has been created. The `--deletesource-branch` switch can be used to delete the branch after the pull request is complete. Also, consider using `--auto-complete` to complete automatically when all policies have passed, and the source branch can be merged into the target branch.

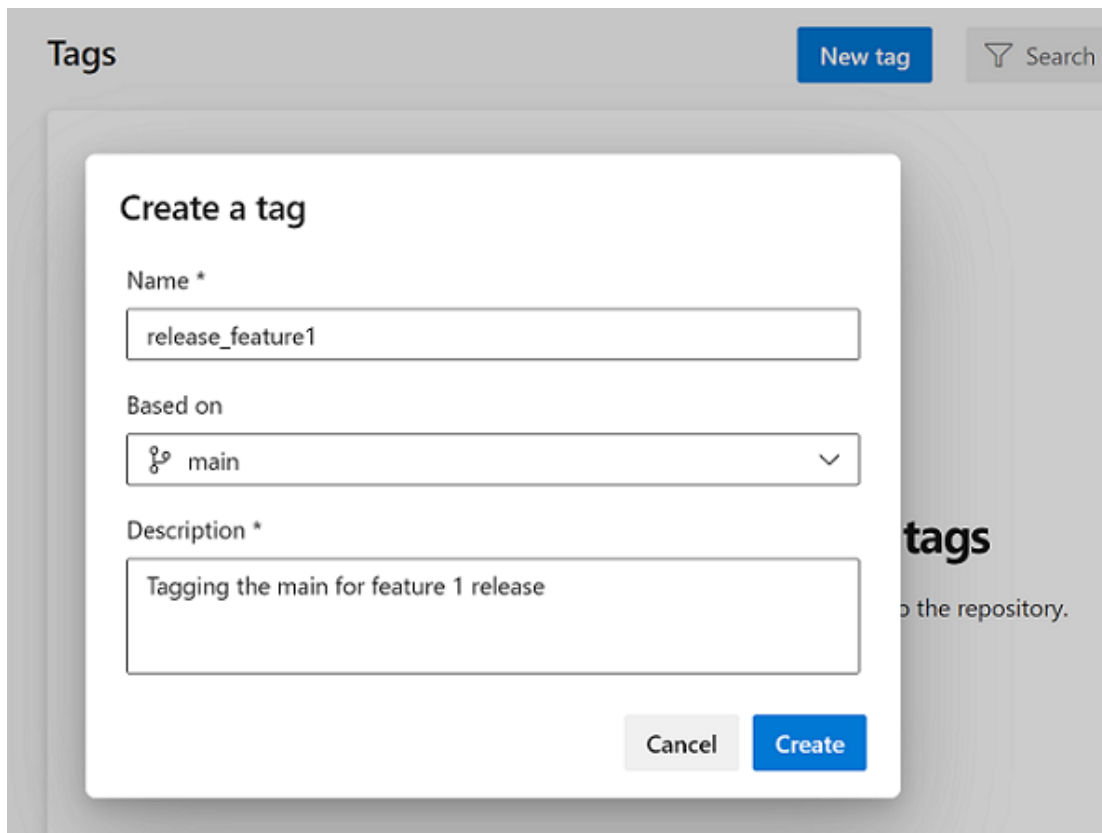
Note

For more information about **az repos pr create** parameter, see [Create a pull request to review and merge code](#).

The team jointly reviews the code changes and approves the pull request:

The screenshot displays the Azure DevOps pull request interface. At the top, the title "Review Feature-1 before merging to main" is shown. Below the title, a green "Completed" badge is followed by the PR number "134" and a user icon. To the right, the text "feature/myFeature-1 into main" is displayed. Below this, there are tabs for "Overview", "Files", "Updates", and "Commits", with "Overview" being the active tab. The main content area shows a commit icon and the text "completed this pull request Just now". Below this, a summary box states "Merged PR 34: Review Feature-1 before merging to main" with a commit hash "0792185c" and a user icon, followed by "Just now". A link "Show details" is provided. At the bottom, a green checkmark icon is next to the text "No merge conflicts" and "Last checked Just now".

The main is ready to release. Team tags the main branch with the release number:



7. Start work on Feature 2. Create a branch on remote from the main branch and do the checkout locally:

myWebApp

```
git push origin main:refs/heads/feature/myFeature-2
```

Output:

Total 0 (delta 0), reused 0 (delta 0) To

https://dev.azure.com/Geeks/PartsUnlimited/_git/MyWebApp * [new branch]
origin/HEAD -> refs/heads/feature/myFeature-2.

myWebApp

```
git checkout feature/myFeature-2
```

Output:

Switched to a new branch 'feature/myFeature-2' Branch feature/myFeature-2 set up to track remote branch feature/myFeature-2 from origin.

8. Modify Program.cs by changing the same comment line in the code changed in feature-1.

```

public class Program
{
    // Editing the same line (file from feature-2 branch)
    public static void Main(string[] args)
    {
        BuildWebHost(args).Run();
    }

    public static IWebHost BuildWebHost(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>()
            .Build();
}

```

9. Commit the changes locally, push them to the remote repository, and then raise a pull request to merge the changes from feature/myFeature-2 to the main branch:

```

az repos pr create --title "Review Feature-2 before merging to main" --work-items 40
42 `
--description "#Merge feature-2 to main" `
--source-branch feature/myFeature-2 --target-branch main `
--repository myWebApp --open

```

A critical bug is reported in production against the feature-1 release with the pull request in flight. To investigate the issue, you need to debug against the version of the code currently deployed in production. To investigate the issue, create a new fof branch using the release_feature1 tag:

myWebApp

```
git checkout -b fof/bug-1 release_feature1
```

Output:

Switched to a new branch, 'fof/bug-1'.

10. Modify Program.cs by changing the same line of code that was changed in the feature-1 release:

```

public class Program
{
    // Editing the same line (file from feature-FOF branch)
    public static void Main(string[] args)
    {
        BuildWebHost(args).Run();
    }

    public static IWebHost BuildWebHost(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>()
            .Build();
}

```

11. Stage and commit the changes locally, then push changes to the remote repository:

myWebApp

```

git add .
git commit -m "Adding FOF changes."
git push -u origin fof/bug-1

```

Output:

To https://dev.azure.com/Geeks/PartsUnlimited/_git/MyWebApp * [new branch] fof/bug-1 - fof/bug-1 Branch fof/bug-1 set up to track remote branch fof/bug-1 from origin.

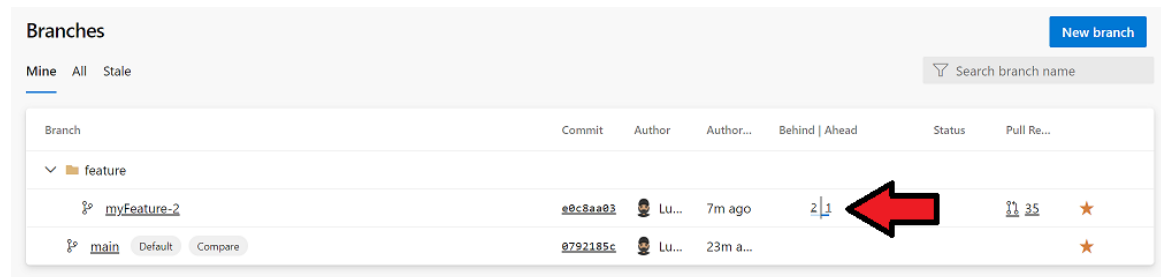
12. Immediately after the changes have been rolled out to production, tag the fof\bug-1 branch with the release_bug-1 tag, then raise a pull request to merge the changes from fof/bug-1 back into the main:

```
az repos pr create --title "Review Bug-1 before merging to main" --work-items 100 `
--description "#Merge Bug-1 to main" `
--source-branch fof/Bug-1 --target-branch main `
--repository myWebApp --open
```

As part of the pull request, the branch is deleted. However, you can still reference the entire history using the tag.

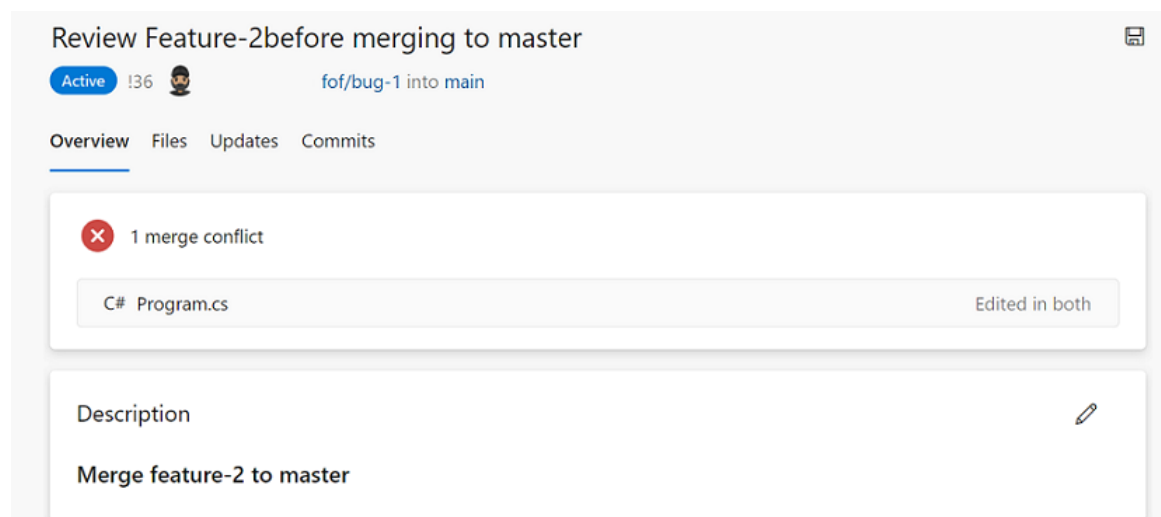
With the critical bug fix out of the way, let's review the feature-2 pull request.

The branches page makes it clear that the feature/myFeature-2 branch is one change ahead of the main and two changes behind the main:



Branch	Commit	Author	Author...	Behind Ahead	Status	Pull Re...
feature						
myFeature-2	ebc8aa03	Lu...	7m ago	2 1	11 35	★
main	9792185c	Lu...	23m a...			★

If you tried to approve the pull request, you'd see an error message informing you of a merge conflict:



Review Feature-2before merging to master

Active 136 fof/bug-1 into main

Overview Files Updates Commits

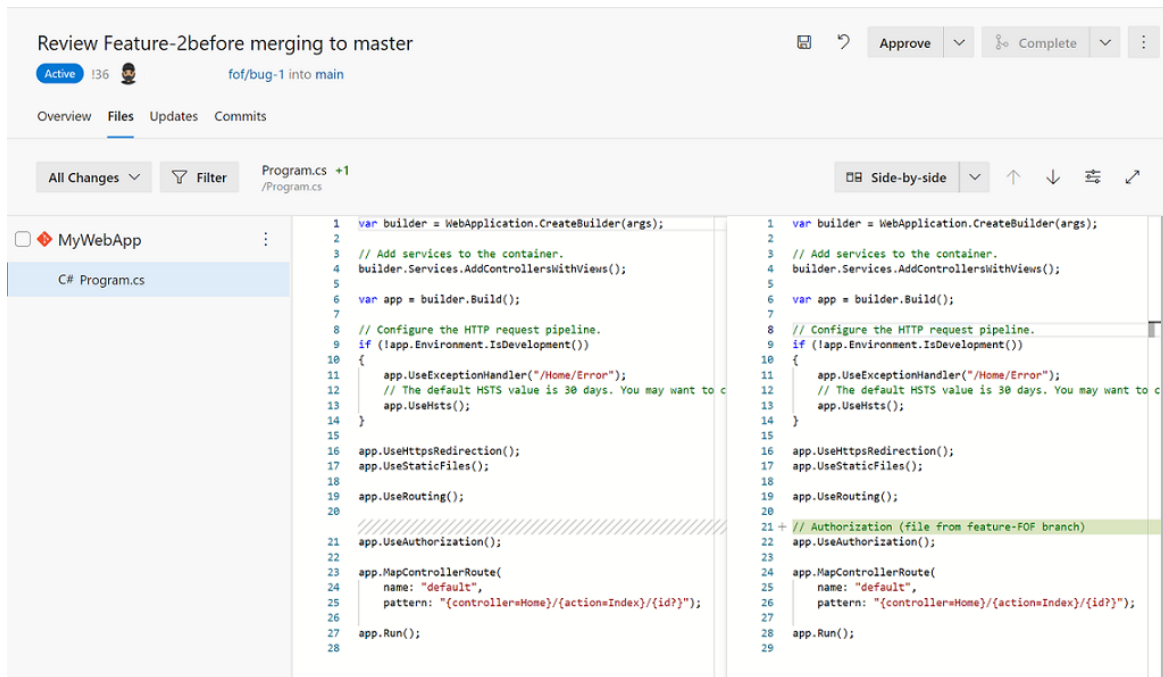
1 merge conflict

C# Program.cs Edited in both

Description

Merge feature-2 to master

13. The Git Pull Request Merge Conflict resolution extension makes it possible to resolve merge conflicts right in the browser. Navigate to the conflicts tab and click on Program.cs to resolve the merge conflicts:



The user interface allows you to take the source, target, add custom changes, review, and submit the merge. With the changes merged, the pull request is completed.

How it works

We learned how the Git branching model gives you the flexibility to work on features in parallel by creating a branch for each feature.

The pull request workflow allows you to review code changes using the branch policies.

Git tags are a great way to record milestones, such as the version of code released; tags give you a way to create branches from tags.

We created a branch from a previous release tag to fix a critical bug in production.

The branches view in the web portal makes it easy to identify branches ahead of the main. Also, it forces a merge conflict if any ongoing pull requests try to merge to the main without resolving the merge conflicts.

A lean branching model allows you to create short-lived branches and push quality changes to production faster.

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

[Explore GitHub flow](#)

Completed

- 3 minutes

GitHub is the best tool to enable collaboration in your projects. GitHub flow is a branch-based workflow suggested for GitHub.

Note

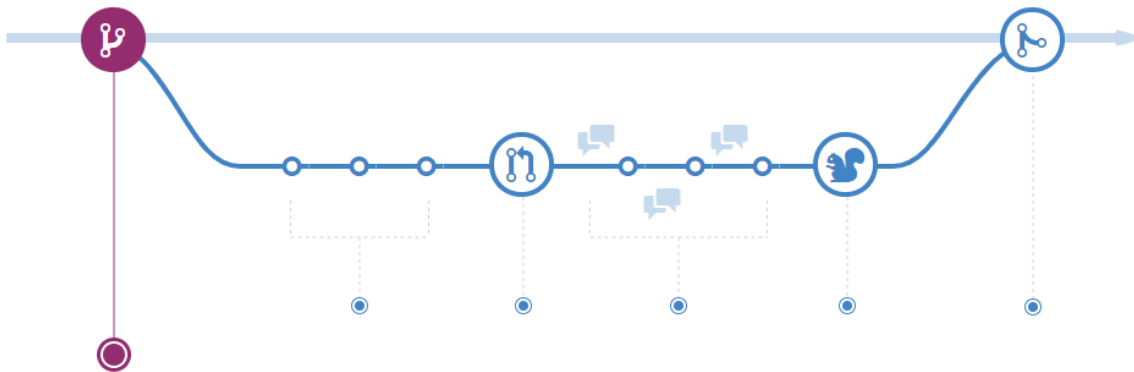
To implement GitHub flow, you'll need a GitHub account and a repository. See "[Signing up for GitHub](#)" and "[Create a repo](#)."

Tip

You can complete all steps of GitHub flow through the GitHub web interface, command line, [GitHub CLI](#), or [GitHub Desktop](#).

The first step is to create a branch in your repository to work without affecting the default branch, and you give collaborators a chance to review your work.

For more information, see "[Creating and deleting branches within your repository](#)."



Make any desired changes to the repository. If you make a mistake, you can revert or push extra changes to fix it.

Commit and push your changes to your branch to back up your work to remote storage, giving each commit a descriptive message. Each commit should contain an isolated, complete change making it easy to revert if you take a different approach.

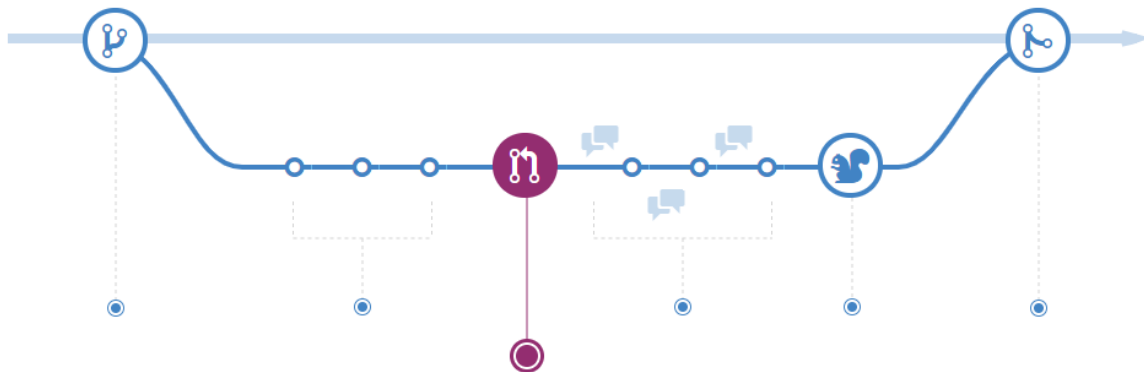
Anyone collaborating with your project can see your work, answer questions, and make suggestions or contributions. Continue to create, commit, and push changes to your branch until you're ready to ask for feedback.

Tip


You can make a separate branch for each change to make it easy for reviewers to give feedback or for you to understand the differences.

Once you're ready, you can create a pull request to ask collaborators for feedback on your changes. See "[Creating a pull request](#)."

Pull request review is one of the most valuable features of collaboration. You can require approval from your peers and team before merging changes. Also, you can mark it as a draft in case you want early feedback or advice before you complete your changes.



Describe the pull request as much as possible with the suggested changes and what problem you're resolving. You can add images, links, related issues, or any information to document your change and help reviewers understand the PR without opening each file. See "[Basic writing and formatting syntax](#)" and "[Linking a pull request to an issue](#)."



John Doe commented on Mar 4

Collaborator

- Overall - Minor typos and word capitalizations corrected.
- Lab numbering and ordering to follow Skillpipe and Slides order.

Related Issue

#213

Checklist

Mark completed with "x" between brackets, "[x]", or checking the box once the PR is created:

☒ Has related GitHub Issue

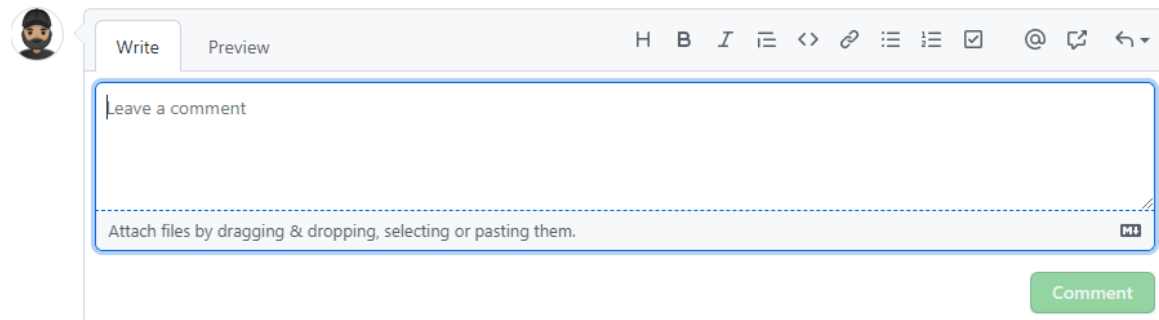
☒ Tested it

☒ Read the PR collaboration guide

Changes proposed in this pull request:

- Lab numbering and ordering to follow Skillpipe and Slides order.

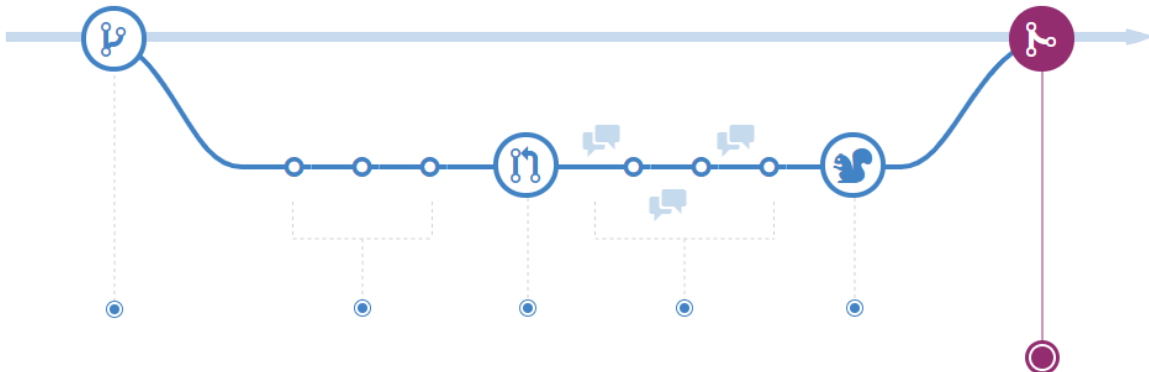
Another way to improve PR quality and documentation and explicitly point something out to the reviewers is to use the comment session area. Also, you can @mention or request a review from specific people or teams.

A screenshot of the GitHub Pull Request comment interface. At the top left is a user profile icon. To its right are tabs for 'Write' and 'Preview'. Further right is a rich text editor toolbar with icons for bold (H), italic (I), underline (≡), code (<>), link, list, checkbox, @mention, share, and undo. Below the toolbar is a large text area with the placeholder text 'Leave a comment'. At the bottom of the text area is a smaller line with the text 'Attach files by dragging & dropping, selecting or pasting them.' and a small 'x' icon. To the right of the text area is a green 'Comment' button.

There are other Pull Requests configurations, such as automatically requesting a review from specific teams or users when a pull request is created or checks to run on pull requests. For more information, see "[About status checks](#)" and "[About protected branches](#)".

After the reviewers' comments and checks validation, the changes should be ready to be merged, and they can approve the Pull Request. See [Merging a pull request](#)."

If you have any conflicts, GitHub will inform you to resolve them. "[Addressing merge conflicts](#)."



After a successful pull request merges, there's no need for the remote branch to stay there. You can delete your branch to prevent others from accidentally using old branches. For more information, see "[Deleting and restoring branches in a pull request](#)".

Note

GitHub keeps the commit and merges history if you need to restore or revert your pull request.

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

[Explore fork workflow](#)

Completed

- 3 minutes

The forking workflow is fundamentally different than other popular Git workflows.

Instead of using a single server-side repository to act as the "central" codebase, it gives every developer their server-side repository.

It means that each contributor has two Git repositories:

- A private local.
- A public server-side.

The forking workflow is most often seen in public open-source projects.

The main advantage of the forking workflow is that contributions can be integrated without the need for everybody to push to a single central repository.

Developers push to their server-side repositories, and only the project maintainer can push to the official repository.

It allows the maintainer to accept commits from any developer without giving them written access to the official codebase.

The forking workflow typically will be intended for merging into the original project maintainer's repository.

The result is a distributed workflow that provides you a flexible way for large, organic teams (including untrusted third parties) to collaborate securely.

This also makes it an ideal workflow for open-source projects.

How it works

As in the other Git workflows, the forking workflow begins with an official public repository stored on a server.

But when a new developer wants to start working on the project, they don't directly clone the official repository.

Instead, they fork the official repository to create a copy of it on the server.

This new copy serves as their personal public repository—no other developers can push to it, but they can pull changes from it (we'll see why this is necessary in a moment).

After they've created their server-side copy, the developer does a git clone to get a copy of it onto their local machine.

It serves as their private development environment, just like in the other workflows.

When they're ready to publish a local commit, they push the commit to their public repository—not the official one.

Then, they file a pull request with the main repository, which lets the project maintainer know that an update is ready to be integrated.

The pull request also serves as a convenient discussion thread if there are issues with the contributed code.

The following is a step-by-step example of this workflow:

- A developer 'forks' an 'official' server-side repository. It creates their server-side copy.
- The new server-side copy is cloned to their local system.
- A Git remote path for the 'official' repository is added to the local clone.
- A new local feature branch is created.
- The developer makes changes to the new branch.
- New commits are created for the changes.
- The branch gets pushed to the developer's server-side copy.
- The developer opens a pull request from the new branch to the 'official' repository.
- The pull request gets approved for merge and is merged into the original server-side repository.

To integrate the feature into the official codebase:

- The maintainer pulls the contributor's changes into their local repository.
- Checks to make sure it doesn't break the project.
- Merges it into their local main branch.
- Pushes the main branch to the official repository on the server.

The contribution is now part of the project, and other developers should pull from the official repository to synchronize their local repositories.

It's essential to understand that the notion of an "official" repository in the forking workflow is merely a convention.

The only thing that makes the official repository, so official is that it's the repository of the project maintainer.

Forking vs. cloning

It's essential to note that "forked" repositories and "forking" aren't special operations.

Forked repositories are created using the standard git clone command. Forked repositories are generally "server-side clones" managed and hosted by a Git service provider such as Azure Repos.

There's no unique Git command to create forked repositories.

A clone operation is essentially a copy of a repository and its history.

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Version Control with Git in Azure Repos

Completed

- 60 minutes

Estimated time: 60 minutes.

Lab files: none.

Scenario

Azure DevOps supports two types of version control, Git and Team Foundation Version Control (TFVC). Here's a quick overview of the two version control systems:

- **Team Foundation Version Control (TFVC):** TFVC is a centralized version control system. Typically, team members have only one version of each file on their dev machines. Historical data is maintained only on the server. Branches are path-based and created on the server.
- **Git:** Git is a distributed version control system. Git repositories can live locally (on a developer's machine). Each developer has a copy of the source repository on their dev machine. Developers can commit each set of changes on their dev machine, perform version control operations such as history, and compare without a network connection.

Git is the default version control provider for new projects. You should use Git for version control in your projects unless you need centralized version control features in TFVC.

In this lab, you'll learn to establish a local Git repository, which can easily be synchronized with a centralized Git repository in Azure DevOps. In addition, you'll learn about Git branching and merging support. You'll use Visual Studio Code, but the same processes apply to using any Git-compatible client.

Objectives

After completing this lab, you'll be able to:

- Clone an existing repository.
- Save work with commits.
- Review the history of changes.
- Work with branches by using Visual Studio Code.

Requirements

- This lab requires **Microsoft Edge** or an [Azure DevOps-supported browser](#).
- **Set up an Azure DevOps organization:** If you don't already have an Azure DevOps organization that you can use for this lab, create one by following the instructions available at [Create an organization or project collection](#).
- If you don't have Git **2.29.2** or later installed, start a web browser, navigate to the [Git for Windows download page](#), and install it.
- If you don't have Visual Studio Code installed yet, navigate to the [Visual Studio Code download page](#) from the web browser window, download it, and install it.
- If you don't have the Visual Studio C# extension installed yet, navigate to the [C# extension installation page](#) in the web browser window and install it.

Exercises

During this lab, you'll complete the following exercises:

- Exercise 0: Configure the lab prerequisites.
- Exercise 1: Clone an existing repository.
- Exercise 2: Save work with commits.
- Exercise 3: Review history.
- Exercise 4: Work with branches.

Launch Exercise

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

[Knowledge check](#)

Completed

- 5 minutes

Choose the best response for each question. Then select **Check your answers** .

Check your knowledge

1.

Which of the following choices isn't a branch workflow type?

☐

Forking workflow.

☐

Trunk-based development.

☐

SmartFlow workflow.

2.

Which of the following choices gives every developer a server-side repository?

☐

Feature branching.

☐

Trunk-based.

☐

Forking.

3.

Which of the following choices creates a feature branch using git commands?

☐

git checkout -b feature_branch.

☐

git checkout -n feature_branch.

☐

git feature start feature_branch.

Check your answers

You must answer all questions before checking your work.

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

[Summary](#)

Completed

- 1 minute

This module explored Git branching types, concepts, and models for the continuous delivery process. It helped companies define their branching strategy and organization.

You learned how to describe the benefits and usage of:

- Describe Git branching workflows.
- Implement feature branches.
- Fork a repo.

Learn more

- [Git branching guidance - Azure Repos | Microsoft Docs](#) .
- [Create a new Git branch from the web - Azure Repos | Microsoft Docs](#) .
- [How Microsoft develops modern software with DevOps - Azure DevOps | Microsoft Docs](#) .
- [Fork your repository - Azure Repos | Microsoft Docs](#) .

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Collaborate with pull requests in Azure Repos

Introduction

Completed

- 1 minute

This module presents pull requests for collaboration and code reviews using Azure DevOps and GitHub mobile for pull request approvals.

It helps understand how pull requests work and how to configure them.

Learning objectives

After completing this module, students and professionals can:

- Use pull requests for collaboration and code reviews.
- Give feedback-using pull requests.
- Configure branch policies.
- Use GitHub mobile for pull requests approvals.

Prerequisites

- Understanding of what DevOps is and its concepts.
- Familiarity with version control principles is helpful but isn't necessary.
- Beneficial to have experience in an organization that delivers software.

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Collaborate with pull requests

Completed

- 3 minutes

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.

Everyone shares a single repository, and topic branches are used to develop features and isolate changes.

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.

Also, start code review and general discussion about a set of changes before being merged into the main branch.

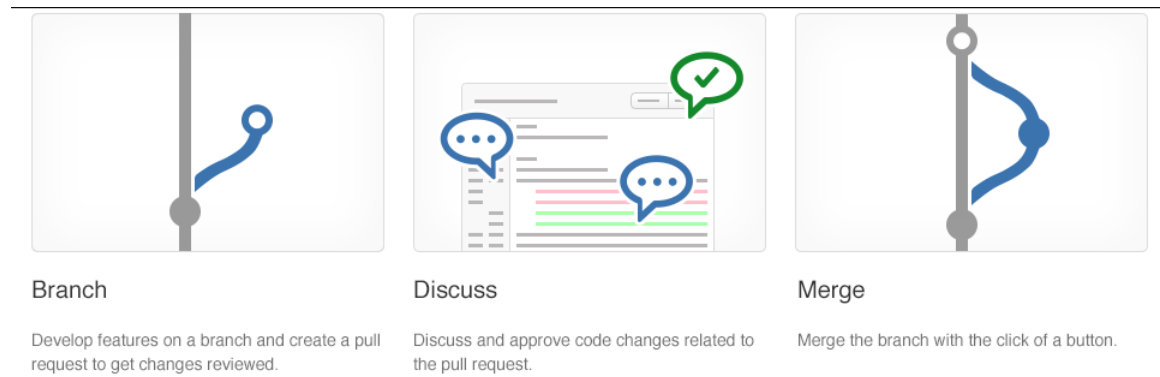
Pull requests combine the review and merge of your code into a single collaborative process.

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.



Get your code reviewed

The code review done in a pull request isn't just to find bugs—that's what your tests are concerning.

A good code review catches less obvious problems that could lead to costly issues later.

Code reviews help protect your team from bad merges and broken builds that sap your team's productivity.

The review catches these problems before the merge, protecting your essential branches from unwanted changes.

Cross-pollinate expertise and spread problem-solving strategies by using a wide range of reviewers in your code reviews.

Diffusing skills and knowledge makes your team more robust and more resilient.

Give great feedback

High-quality reviews start with high-quality feedback. The keys to great feedback in a pull request are:

- Have the right people review the pull request.
- Make sure that reviewers know what the code does.
- Give actionable, constructive feedback.
- Reply to comments promptly.

When assigning reviewers to your pull request, make sure you select the right set of reviewers.

You want reviewers who know how your code works and try to include developers working in other areas to share their ideas.

Also, who can provide a clear description of your changes and build your code that has your fix or feature running in it.

Reviewers should try to provide feedback on changes they disagree with. Identify the issue and give a specific suggestion on what you would do differently.

This feedback has clear intent and is easy for the owner of the pull request to understand.

The pull request owner should reply to the comments, accept the suggestion, or explain why the suggested change isn't ideal.

Sometimes a suggestion is good, but the changes are outside the scope of the pull request.

Take these suggestions and create new work items and feature branches separate from the pull request to make those changes.

Protect branches 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.

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Exercise - Azure Repos collaborating with pull requests

Completed

- 6 minutes

Code issues that are found sooner are both easier and cheaper to fix. So development teams strive to push code quality checks as far left into the development process as possible.

As the name suggests, branch policies give you a set of out-of-the-box policies that can be applied to the branches on the server.

Any changes being pushed to the server branches need to follow these policies before the changes can be accepted.

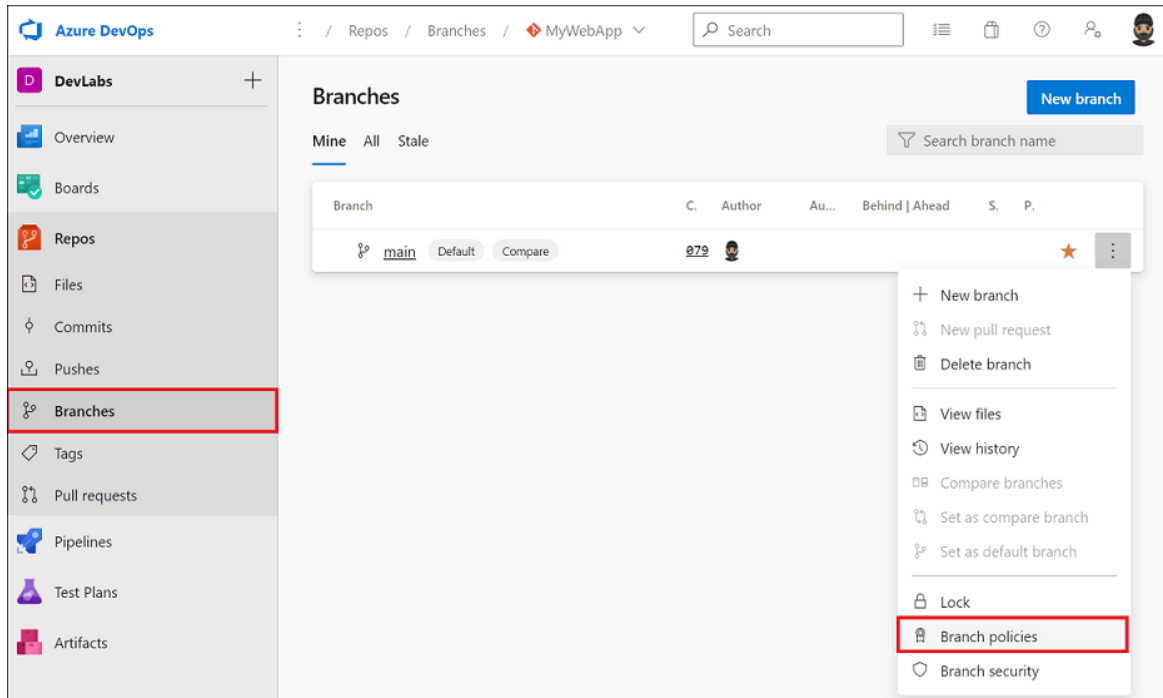
Policies are a great way to enforce your team's code quality and change-management standards. In this recipe, you'll learn how to configure branch policies on your main branch.

Getting ready

The out-of-the-box branch policies include several policies, such as build validation and enforcing a merge strategy. We'll only focus on the branch policies needed to set up a code-review workflow in this recipe.

How to do it

1. Open the branches view for the myWebApp Git repository in the parts-unlimited team portal. Select the main branch, and from the pull-down, context menu choose Branch policies:



2. In the policies view, It presents out-of-the-box policies. Set the minimum number of reviewers to 1:

The screenshot shows the 'Branch Policies' configuration page. At the top, a note states: 'Note: If any required policy is enabled, this branch cannot be deleted and changes must be made via pull request.' Below this, the 'Require a minimum number of reviewers' policy is shown with a toggle switch set to 'On'. The 'Minimum number of reviewers' is set to 1 in a text input field. Below this, several other options are listed with unchecked checkboxes: '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', and 'When new changes are pushed:'.

The Allow requestors to approve their own changes option allows the submitter to self-approve their changes.

It's OK for mature teams, where branch policies are used as a reminder for the checks that need to be performed by the individual.

3. Use the review policy with the comment-resolution policy. It allows you to enforce that the code review comments are resolved before the changes are accepted. The

requester can take the feedback from the comment and create a new work item and resolve the changes. It at least guarantees that code review comments aren't lost with the acceptance of the code into the main branch:

☒ 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.

4. A requirement instigates a code change in the team project. If the work item triggered the work isn't linked to the change, it becomes hard to understand why it was made over time. It's especially useful when reviewing the history of changes. Configure the Check for linked work items policy to block changes that don't have a work item linked to them:

☒ On

Check for linked work items
Encourage traceability by checking for linked work items on pull requests.


☒ **Required**
Block pull requests from being completed unless they have at least one linked work item.

☐ **Optional**
Warn if there are no linked work items, but allow pull requests to be completed.

5. Select the option to automatically include reviewers when a pull request is raised automatically. You can map which reviewers are added based on the area of the code being changed:

Add new reviewer policy

Reviewers

 Add people

Policy requirement

☐ Optional

☒ Required

For pull request affecting these folders

i

Leave blank to include the specified reviewers on all pull requests

Completion options

☒ Allow requestors to approve their own changes

Activity feed message

i

How it works

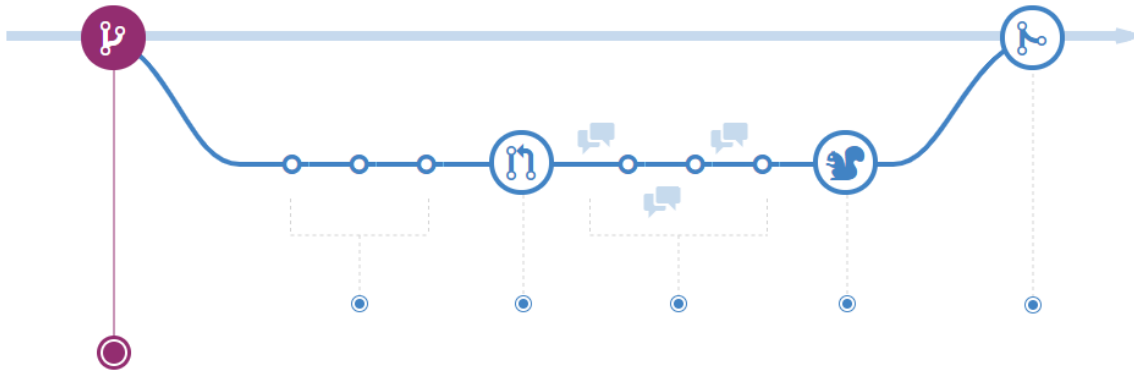
With the branch policies in place, the main branch is now fully protected.

The only way to push changes to the main branch is by first making the changes in another branch and then raising a pull request to trigger the change-acceptance workflow.

Choose to create a new branch from one of the existing user stories in the work item hub.

By creating a new branch from a work item, that work item automatically gets linked to the branch.

You can optionally include more than one work item with a branch as part of the create workflow:



Prefix in the name when creating the branch to make a folder for the branch to go in.

In the preceding example, the branch will go in the folder. It is a great way to organize branches in busy environments.

With the newly created branch selected in the web portal, edit the HomeController.cs file to include the following code snippet and commit the changes to the branch.

In the image below, you'll see that you can directly commit the changes after editing the file by clicking the commit button.

The file path control in the team portal supports search.

Start typing the file path to see all files in your Git repository under that directory, starting with these letters showing up in the file path search results dropdown.

Contents

Highlight changes

Commit...

```

1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics;
4  using System.Linq;
5  using System.Threading.Tasks;
6  using Microsoft.AspNetCore.Mvc;
7  using mywebapp.Models;
8
9  namespace mywebapp.Controllers
10 {
11     public class HomeController : Controller
12     {
13         public IActionResult Index()
14         {
15             return View();
16         }
17
18         private string JoinTwoStrings(string one, string two)
19         {
20             var NewString = string.Concat(one, two);
21             return NewString;
22         }
23     }
24 }

```

The code editor in the web portal has several new features in Azure DevOps Server, such as support for bracket matching and toggle white space.

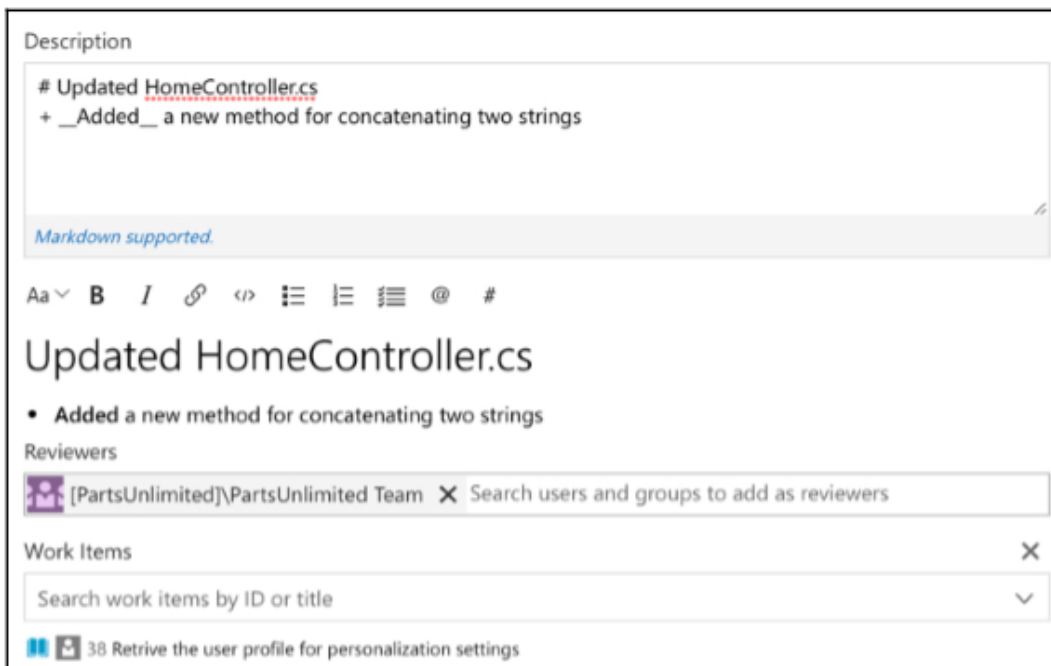
You can load the command palette by pressing it. Among many other new options, you can now toggle the file using a file mini-map, collapse, and expand, and other standard operations.

To push these changes from the new branch into the main branch, create a pull request from the pull request view.

Select the new branch as the source and the main as the target branch.

The new pull request form supports markdown, so you can add the description using the markdown syntax.

The description window also supports @ mentions and # to link work items:

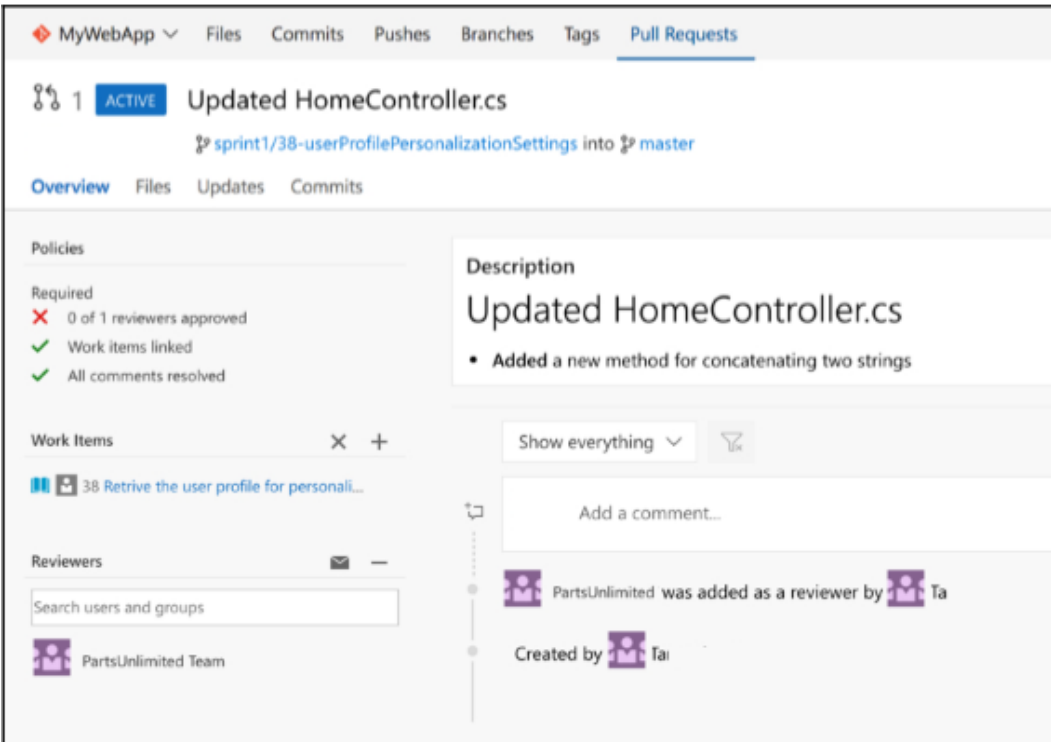


The screenshot shows the 'Description' field of a pull request form. The text entered is: `# Updated HomeController.cs` followed by `+ __Added__ a new method for concatenating two strings`. Below the text area, a blue message states 'Markdown supported.' A rich text toolbar is visible with icons for text color, bold, italic, link, code, bulleted list, numbered list, table, mentions (@), and work items (#). Below the toolbar, the title 'Updated HomeController.cs' is displayed, followed by a bulleted list item: '• Added a new method for concatenating two strings'. The 'Reviewers' section shows a dropdown menu with '[PartsUnlimited]\PartsUnlimited Team' selected and a search bar. The 'Work Items' section has a search bar with the placeholder 'Search work items by ID or title'. At the bottom, there is a status bar with icons and the text '38 Retrieve the user profile for personalization settings'.

The pull request is created; the overview page summarizes the changes and the status of the policies.

The Files tab shows you a list of changes and the difference between the previous and the current versions.

Any updates pushed to the code files will show up in the Updates tab, and a list of all the commits is shown under the Commits tab:



Open the Files tab: this view supports code comments at the line level, file level, and overall.

The comments support both @ for mentions and # to link work items, and the text supports markdown syntax:

The code comments are persisted in the pull request workflow; the code comments support multiple iterations of reviews and work well with nested responses.

The reviewer policy allows for a code review workflow as part of the change acceptance.

It's an excellent way for the team to collaborate on any code changes pushed into the main branch.

When the required number of reviewers approves the pull request, it can be completed.

You can also mark the pull request to autocomplete after your review. It autocompletes the pull requests once all the policies have been successfully compiled.

There's more

Have you ever been in a state where a branch has been accidentally deleted? It can't be easy to figure out what happened.

Azure DevOps Server now supports searching for deleted branches. It helps you understand who deleted it and when. The interface also allows you to recreate the branch.

Deleted branches are only shown if you search for them by their exact name to cut out the noise from the search results.

To search for a deleted branch, enter the full branch name into the branch search box. It will return any existing branches that match that text.

You'll also see an option to search for an exact match in the list of deleted branches.

If a match is found, you'll see who deleted it and when. You can also restore the branch. Restoring the branch will re-create it at the commit to which is last pointed.

However, it won't restore policies and permissions.

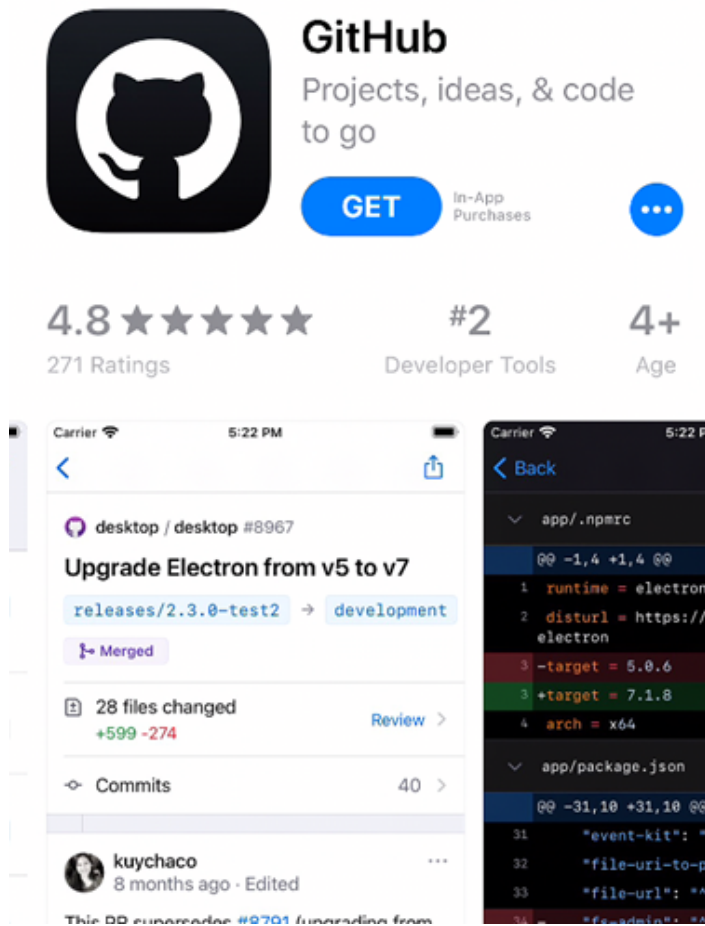
[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

[Examine GitHub mobile for pull request approvals](#)

Completed

- 1 minute



Using a mobile app in combination with Git is a convenient option, particularly when urgent pull request approvals are required.

- The app can render markdown, images, and PDF files directly on the mobile device.
- Pull requests can be managed within the app, along with marking files as viewed, collapsing files.
- Comments can be added.
- Emoji short codes are rendered.

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

[Knowledge check](#)

Completed

- 4 minutes

Choose the best response for each question. Then select **Check your answers** .

Check your knowledge

1.

Which of the following statements best describe a pull request?

☐

Pull requests are commonly used by teams and organizations collaborating, where everyone shares a single repository, and topic branches are used to develop features and isolate changes.

☐

Pull requests are used to publish new changes from feature branches into production branches and update the environments with the latest code version.

☐

Pull request creates reports from the latest branch updates and generates remediation summary for resolving issues during development.

2.

Which of the following choices isn't a benefit of using pull requests?

☐

Protect branches with policies.

☐

Give feedback.

☐

Create reports about code security and quality.

3.

Which of the following choices is the most effective way to avoid direct updates from feature branches into main?

☐

Branch Policies.

☐

Branch Security.



Branch Lock.

Check your answers

You must answer all questions before checking your work.

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Summary

Completed

- 1 minute

This module presented pulls requests for collaboration and code reviews using Azure DevOps and GitHub mobile for pull request approvals.

It helped understanding how pull requests work and how to configure them.

You learned how to describe the benefits and usage of:

- Use pull requests for collaboration and code reviews.
- Give feedback using pull requests.
- Configure branch policies.
- Use GitHub mobile for pull requests approvals.

Learn more

- [About pull requests and permissions - Azure Repos | Microsoft Docs](#) .
- [Create a pull request to review and merge code - Azure Repos | Microsoft Docs](#) .
- [Review and comment on pull requests - Azure Repos | Microsoft Docs](#) .
- [Protect your Git branches with policies - Azure Repos | Microsoft Docs](#) .
- [Creating an issue or pull request - GitHub Docs](#) .

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Identify technical debt

Introduction

Completed

- 1 minute

This module examines technical debt, complexity, quality metrics, and plans for effective code reviews and code quality validation.

Learning objectives

After completing this module, students and professionals can:

- Identify and manage technical debt.
- Integrate code quality tools.
- Plan code reviews.
- Describe complexity and quality metrics.

Prerequisites

- Understanding of what DevOps is and its concepts.
- Familiarity with version control principles is helpful but isn't necessary.
- Beneficial to have experience in an organization that delivers software.

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Examine code quality

Completed

- 2 minutes

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.

A study on "Software Defect Origins and Removal Methods" found that individual programmers are less than 50% efficient at finding bugs in their software. And most forms of testing are only 35% efficient. It makes it difficult to determine quality.

There are five key traits to measure for higher 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.

Some metrics you may consider to improve maintainability are the number of stylistic warnings and Halstead complexity measures.

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.

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Examine complexity and quality metrics

Completed

- 2 minutes

While there are various quality metrics, a few of the most important ones are listed here.

Complexity metrics can help in measuring quality. Cyclomatic complexity measures the number of linearly independent paths through a program's source code. Another way to understand quality is through calculating Halstead complexity measures.

This measure:

- Program vocabulary.
- Program length.
- Calculated program length.
- Volume.
- Difficulty.
- Effort.

Code analysis tools can be used to check for security, performance, interoperability, language usage, and globalization and should be part of every developer's toolbox and

software build process.

Regularly running a static code analysis tool and reading its output is a great way to improve as a developer because the things caught by the software rules can often teach you something.

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.

We've previously talked about some general project metrics and KPIs. 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?

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Introduction to technical debt

Completed

- 4 minutes

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.

The term-technical debt was chosen for its comparison to financial debt. It's common for people in financial debt to make decisions that seem appropriate or the only option at the time, but in so doing, interest increases.

The more interest that accumulates, the harder it is for them in the future and the more minor options available to them later. With financial debt, soon, interest increases on interest, creating a snowball effect. Similarly, 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.

So, how does it happen?

The most common excuse is tight deadlines. 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 I have two copies of the same code that I need to modify in the future instead of one, and I 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.

The organization might not have coding standards at all. So, the developers didn't even know what they should be producing. The developers might not have precise requirements to target. Well, they might be subject to last-minute requirement changes.

Necessary-refactoring work might be delayed. There might not be any code quality testing, manual or automated. In the end, it just makes it harder and harder to deliver value to customers in a reasonable time frame and at a reasonable cost.

Technical debt is one of the main reasons that projects fail to meet their deadlines.

Over time, it increases in much the same way that monetary debt does. Common sources of technical debt are:

- 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.

Note

Over time, the technical debt must be paid back. Otherwise, the team's ability to fix issues and implement new features and enhancements will take longer and eventually become cost-prohibitive.

We have seen that technical debt adds a set of problems during development and makes it much more difficult to add extra customer value.

Having technical debt in a project saps productivity, frustrates development teams, makes code both hard to understand and fragile, increases the time to make changes and validate those changes. Unplanned work frequently gets in the way of planned work.

Longer-term, it also saps the organization's strength. Technical debt tends to creep up on an organization. It starts small and grows over time. Every time a quick hack is made or testing is circumvented because changes need to be rushed through, the problem grows worse and worse. Support costs get higher and higher, and invariably, a serious issue arises.

Eventually, the organization can't respond to its customers' needs in a timely and cost-efficient way.

Automated measurement for monitoring

One key way to minimize the constant acquisition of technical debt is to use automated testing and assessment.

In the demos that follow, we'll look at one of the common tools used to assess the debt: SonarCloud. (The original on-premises version was SonarQube).

There are other tools available, and we'll discuss a few of them.

Later, in the next hands-on lab, you'll see how to configure your Azure Pipelines to use SonarCloud, understand the analysis results, and finally how to configure quality profiles to control the rule sets that are used by SonarCloud when analyzing your projects.

For more information, see [SonarCloud](#).

To review:

Azure DevOps can be integrated with a wide range of existing tooling used to check code quality during builds.

Which code quality tools do you currently use (if any)?

What do you like or don't you like about the tools?

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Measure and manage technical debt

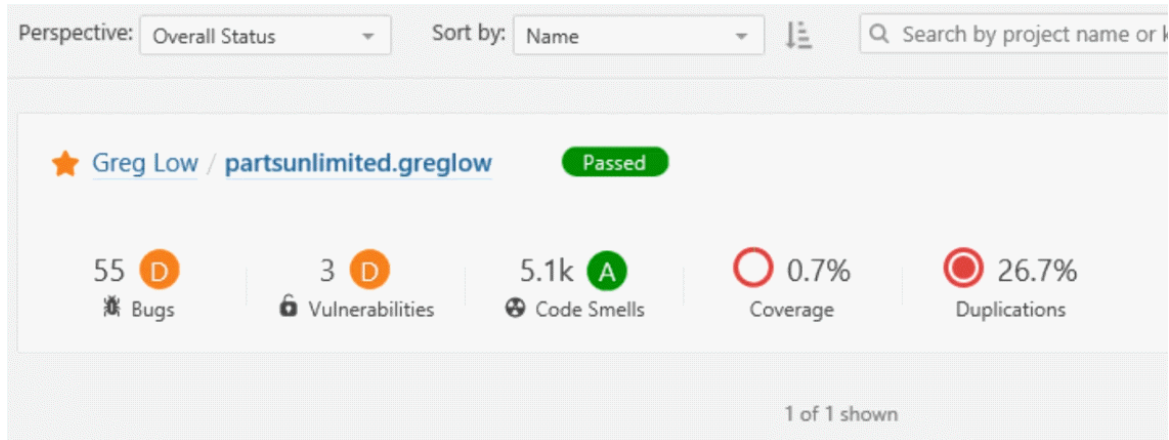
Completed

- 1 minute

Integrating the assessment and measurement of technical debt and code quality overall is essential as part of your Continuous Integration and Deployment pipelines in Azure DevOps.

In the Continuous Integration course in this series, we showed how to add support for SonarCloud into an Azure DevOps pipeline.

After it's added and a build done, you can see an analysis of your code:



If you drill into the issues, you can see what the issues are, along with suggested remedies and estimates of the time required to a remedy.



[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

[Integrate other code quality tools](#)

Completed

- 1 minute

Many tools can be used to assess aspects of your code quality and technical debt.

NDepend

For .NET developers, a common tool is NDepend.

NDepend is a Visual Studio extension that assesses the amount of technical debt a developer has added during a recent development period, typically in the last hour.

With this information, the developer might make the required corrections before ever committing the code.

NDepend lets you create code rules expressed as C# LINQ queries, but it has many built-in rules that detect a wide range of code smells.

Resharper Code Quality Analysis


Resharper can make a code quality analysis from the command line. Also, be set to fail builds when code quality issues are found automatically.

Rules can be configured for enforcement across teams.

Search in Azure DevOps

To find other code quality tools that are easy to integrate with Azure DevOps, search for the word **Quality** when adding a task into your build pipeline.

Add tasks |  Refresh

quality 

Marketplace ^



Resharper Code Quality Analysis

Run the Resharper Command-Line Tool and automatically fail builds when code quality issues are found. Configure team-shared coding rules for seamless code quality standards enforcement on each commit



Build Quality Checks

Breaks a build based on quality metrics like number of warnings or code coverage.



Code Quality NDepend for TFS 2017/TFS 2018 and VSTS

Build tasks and hub helping you to understand the technical debt of built code and it's evolution. Explore code metrics and define quality gates and trends



Quality Gate Widget

Widget to show the SonarQube Quality Gate status for a project



Quality Gate Web Smoke Test

Simple server-side (not agent based) Task for performing a smoke test on a released web application to validate that it is available at the end of a release as a Quality Gate.



SonarCloud quality gate

this extension covers quality gate enforcement using data from SonarCloud



SQL Enlight Code Quality

Build and Release tasks for SQL Enlight Code Quality integration with Visual Studio Team Services and Team Foundation Server 2017/2018.



WhiteSource

Detect & fix security vulnerabilities, problematic open source licenses and quality issues.



SonarQube build breaker

For more information, you can see:

- [NDepend](#)
- Visual Studio marketplace
- Resharper Code Quality Analysis

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

[Plan effective code reviews](#)

Completed

- 1 minute

Most developers would agree that code reviews can improve the quality of the applications they produce, but only if the process for the code reviews is effective. It is essential, upfront, to agree that everyone is trying to achieve better code quality.

Achieving code quality can seem to challenge because there's no single best way to write any piece of code, at least code with any complexity. Everyone wants to do good work and to be proud of what they create.

It means that it's easy for developers to become over-protective of their code. The organizational culture must let all involved feel that the code reviews are more like mentoring sessions where ideas about improving code are shared than interrogation sessions where the aim is to identify problems and blame the author.

The knowledge-sharing that can occur in mentoring-style sessions can be one of the most important outcomes of the code review process. It often happens best in small groups (even two people) rather than in large team meetings. And it's important to highlight what has been done well, not just what needs improvement.

Developers will often learn more in effective code review sessions than they'll in any formal training. Reviewing code should be an opportunity for all involved to learn, not just as a chore that must be completed as part of a formal process.

It's easy to see two or more people working on a problem and think that one person could have completed the task by themselves. That is a superficial view of the longer-term outcomes.

Team management needs to understand that improving the code quality reduces the cost of code, not increases it. Team leaders need to establish and foster an appropriate culture across their teams.

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Knowledge check

Completed

- 4 minutes

Choose the best response for each question. Then select **Check your answers** .

Check your knowledge

1.

Which of the following choices is a characteristic in source code possibly indicating a deeper problem?

☐

Memory Leak.

☐

Bug.

☐

Code smell.

2.

Which of the following choices can measure the probability that a system will run without failure over a specific period of operation?

☐

Maintainability

☐

Reliability

☐

Reusability

3.

Which of the following tools can you use to do code quality checks?

☐

Veracode.

☐

SonarCloud.

☐

Azure Security Center.

Check your answers

You must answer all questions before checking your work.

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

[Summary](#)

Completed

- 1 minute

This module examined technical debt, complexity, quality metrics, and plans for effective code reviews and code quality validation.

You learned how to describe the benefits and usage of:

- Identify and manage technical debt.
- Integrate code quality tools.
- Plan code reviews.
- Describe complexity and quality metrics.

Learn more

- [Technical Debt – The Anti-DevOps Culture - Developer Support \(microsoft.com\)](#).
- [Microsoft Security Code Analysis documentation overview | Microsoft Docs](#).
- [Build Quality Indicators report - Azure DevOps Server | Microsoft Docs](#).

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Explore Git hooks

Introduction

Completed

- 1 minute

This module describes Git hooks and their usage during the development process, implementation, and behavior.

Learning objectives

After completing this module, students and professionals can:

- Understand Git hooks.
- Identify when used Git hooks.
- Implement Git hooks for automation.
- Explain Git hooks' behavior.

Prerequisites

- Understanding of what DevOps is and its concepts.
- Familiarity with version control principles is helpful but is not necessary.
- Beneficial to have experience in an organization that delivers software.

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Introduction to Git hooks

Completed

- 4 minutes

Continuous delivery demands a significant level of automation. You can't be continuously delivering if you don't have a quality codebase. It's where git fares so well.

It lets you automate most of the checks in your codebase. Before committing the code into your local repository, let alone the remote.

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

Practical use cases for using Git hooks

Since Git hooks execute the scripts on the specific event type they're called on, you can do much anything with 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.)

In the next unit, you will see how to implement Git Hooks.

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Implement Git hooks

Completed

- 5 minutes

When pushing quality into the development process, developing code locally, you want developers to identify and catch code quality issues. It's even before raising the pull request to trigger the branch policies.

Git hooks allow you to run custom scripts whenever certain important events occur in the Git life cycle—for example, committing, merging, and pushing.

Git ships with several sample hook scripts in the repo `.git\hooks` directory. Git executes the contents on the particular occasion type they're approached. You can do practically anything with Git hooks.

Here are a few instances of where you can use Git hooks to uphold arrangements, guarantee consistency, and control your environment:

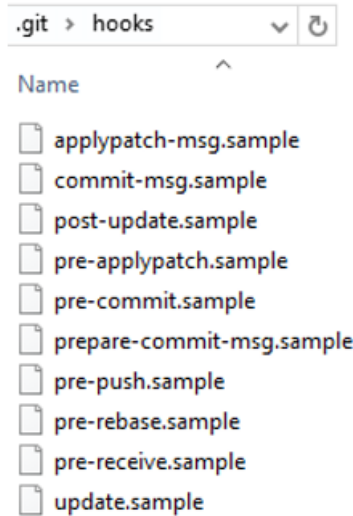
- Enforcing preconditions for merging
- Verifying work Item ID association in your commit message Preventing you and your team from committing faulty code
- Sending notifications to your team's chat room (Teams, Slack, HipChat)

In this recipe, we'll use the pre-commit Git hook to scan the commit for keywords from a predefined list to block the commit if it contains any of these keywords.

Getting ready

Let's start by exploring client-side Git hooks. Navigate to the repo `.git\hooks` directory – you'll find that there are a bunch of samples, but they're disabled by default.

Automation through Source Control...



"Using #Git hooks is like having little robot minions to carry out your every wish..."

Note

If you open that folder, you'll find a file called pre-commit.sample. To enable it, rename it to pre-commit by removing the .sample extension and making the script executable.

The script is found and executed when you attempt to commit using git commit. You commit successfully if your pre-commit script exits with a 0 (zero). Otherwise, the commit fails. If you're using Windows, simply renaming the file won't work.

Git will fail to find the shell in the chosen path specified in the script.

The problem is lurking in the first line of the script, the shebang declaration:

```
#!/bin/sh
```

On Unix-like OSs, the #! Tells the program loader that it's a script to be interpreted, and /bin/sh is the path to the interpreter you want to use, sh in this case.

Windows isn't a Unix-like OS. Git for Windows supports Bash commands and shell scripts via Cygwin.

By default, what does it find when it looks for sh.exe at /bin/sh?

Nothing, nothing at all. Fix it by providing the path to the sh executable on your system. It's using the 64-bit version of Git for Windows, so the baseline looks like this:

```
#!C:/Program\ Files/Git/usr/bin/sh.exe
```

How to do it

How could Git hooks stop you from accidentally leaking Amazon AWS access keys to GitHub?

You can invoke a script at pre-commit.

Using Git hooks to scan the increment of code being committed into your local repository for specific keywords:

Replace the code in this pre-commit shell file with the following code.

```
#!/C:/Program\ Files/Git/usr/bin/sh.exe
matches=$(git diff-index --patch HEAD | grep '^+' | grep -Pi 'password|keyword2|keyword3')
if [ ! -z "$matches" ]
then
    cat <<\EOT
Error: Words from the blocked list were present in the diff:
EOT
    echo $matches
    exit 1
fi
```

You don't have to build the complete keyword scan list in this script.

You can branch off to a different file by referring to it here to encrypt or scramble if you want to.

How it works

The Git diff-index identifies the code increment committed in the script. This increment is then compared against the list of specified keywords. If any matches are found, an error is raised to block the commit; the script returns an error message with the list of matches. The pre-commit script doesn't return 0 (zero), which means the commit fails.

There's more

The repo .git\hooks folder isn't committed into source control. You may wonder how you share the goodness of the automated scripts you create with the team.

The good news is that, from Git version 2.9, you can now map Git hooks to a folder that can be committed into source control.

You could do that by updating the global settings configuration for your Git repository:

```
Git config --global core.hooksPath '~/githubhooks'
```

If you ever need to overwrite the Git hooks you have set up on the client-side, you can do so by using the no-verify switch:

```
Git commit --no-verify
```

Server-side service hooks with Azure Repos

So far, we've looked at the client-side Git Hooks on Windows. Azure Repos also exposes server-side hooks. Azure DevOps uses the exact mechanism itself to create Pull requests. You can read more about it at the [Server hooks event reference](#).

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

[Knowledge check](#)

Completed

- 4 minutes

Choose the best response for each question. Then select **Check your answers** .

Check your knowledge

1.

Which of the following choices isn't a Git hook event?

☐

Post-checkout.

☐

Pre-commit.

☐

After-commit.

2.

Which of the following choices is a practical use case for using Git hooks?

☐

Enforce preconditions for merging applying custom code with the event post-merge.

☐

Send notifications to your team's chat room.

☐

Validate code and merge branches automatically using the commit-msg event.

3.

Which of the following choices is a valid event to verify Work Item ID association in your commit message before commit?

☐

Commit-msg.

☐

Commit-msg-id.

☐

Msg-commit.

Check your answers

You must answer all questions before checking your work.

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

[Summary](#)

Completed

- 1 minute

This module described Git hooks and their usage during the development process, implementation, and behavior.

You learned how to describe the benefits and usage of:

- Understand Git hooks.
- Identify when used Git hooks.
- Implement Git hooks for automation.
- Explain Git hooks' behavior.

Learn more

- [Creating a pre-receive hook script - GitHub Docs](#) .
- [Service hooks event reference - Azure DevOps | Microsoft Docs](#) .

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Plan foster inner source

Introduction

Completed

- 1 minute

This module explains how to use Git to foster inner source and implement Fork and its workflows.

Learning objectives

After completing this module, students and professionals can:

- Use Git to foster inner source across the organization.
- Implement fork workflow.
- Choose between branches and forks.
- Share code between forks.

Prerequisites

- Understanding of what DevOps is and its concepts.
- Familiarity with version control principles is helpful but is not necessary.
- Beneficial to have experience in an organization that delivers software.

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Explore foster inner source

Completed

- 2 minutes

The fork-based pull request workflow is popular with open-source projects because it allows anybody to contribute to a project.

You don't need to be an existing contributor or write access to a project to offer your changes.

This workflow isn't just for open source: forks also help support inner source workflows within your company.

Before forks, you could contribute to a project-using Pull Requests.

The workflow is simple enough: push a new branch up to your repository, open a pull request to get a code review from your team, and have Azure Repos evaluate your branch policies.

You can click one button to merge your pull request into main and deploy when your code is approved.

This workflow is great for working on your projects with your team. But what if you notice a simple bug in a different project within your company and you want to fix it yourself?

What if you're going to add a feature to a project that you use, but another team develops?

It's where forks come in; forks are at the heart of inner source practices.

Inner source

Inner source – sometimes called "internal open source" – brings all the benefits of open-source software development inside your firewall.

It opens your software development processes so that your developers can easily collaborate on projects across your company.

It uses the same processes that are popular throughout the open-source software communities.

But it keeps your code safe and secure within your organization.

Microsoft uses the inner source approach heavily.

As part of the efforts to standardize a one-engineering system throughout the company – backed by Azure Repos – Microsoft has also opened the source code to all our projects to everyone within the company.

Before the move to the inner source, Microsoft was "siloe": only engineers working on Windows could read the Windows source code.

Only developers working on Office could look at the Office source code.

So, if you're an engineer working on Visual Studio and you thought that you found a bug in Windows or Office – or wanted to add a new feature – you're out of luck.

But by moving to offer inner sources throughout the company, powered by Azure Repos, it's easy to fork a repository to contribute back.

As an individual making the change, you don't need to write access to the original repository, just the ability to read it and create a fork.

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Implement the fork workflow

Completed

- 5 minutes

A fork is a copy of a repository. Forking a repository allows you to experiment with changes without affecting the original project freely.

Most commonly, forks are used to propose changes to someone else's project. Or use someone else's project as a starting point for your idea.

A fork is a complete copy of a repository, including all files, commits, and (optionally) branches.

Forks are a great way to support an Inner Source workflow: you can create a fork to suggest changes when you don't have permission to write to the original project directly.

Once you're ready to share those changes, it's easy to contribute them back-using pull requests.

What's in a fork?

A fork starts with all the contents of its upstream (original) repository.

You can include all branches or limit them to only the default branch when you create a fork.

None of the permissions, policies, or build pipelines are applied.

The new fork acts as if someone cloned the original repository, then pushed it to a new, empty repository.

After a fork has been created, new files, folders, and branches aren't shared between the repositories unless a Pull Request (PR) carries them along.

Sharing code between forks

You can create PRs in either direction: from fork to upstream or upstream to fork.

The most common approach will be from fork to upstream.

The destination repository's permissions, policies, builds, and work items will apply to the PR.

Choosing between branches and forks

For a small team (2-5 developers), we recommend working in a single repo.

Everyone should work in a topic branch, and the main should be protected with branch policies.

As your team grows more significant, you may find yourself outgrowing this arrangement and prefer to switch to a forking workflow.

We recommend the forking workflow if your repository has many casual or infrequent contributors (like an open-source project).

Typically, only core contributors to your project have direct commit rights into your repository.

It would help if you asked collaborators from outside this core set of people to work from a fork of the repository.

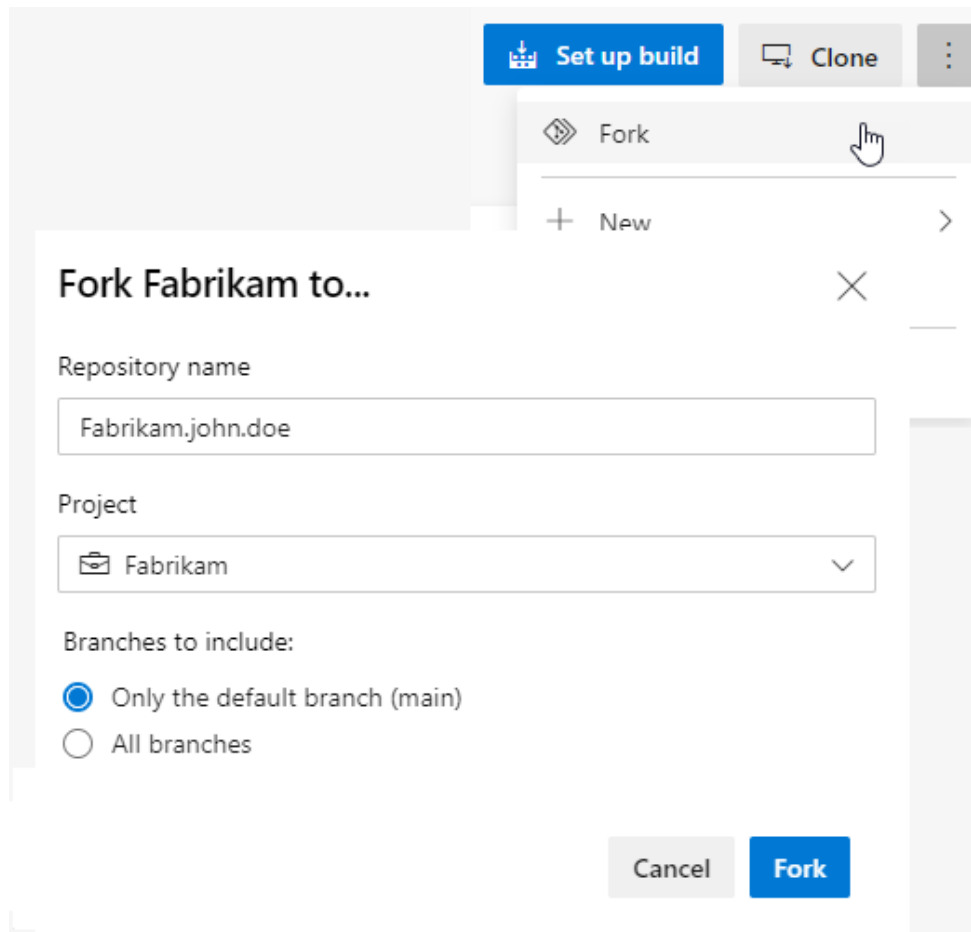
Also, it will isolate their changes from yours until you've had a chance to vet the work.

The forking workflow

- Create a fork.
- Clone it locally.
- Make your changes locally and push them to a branch.
- Create and complete a PR to upstream.
- Sync your fork to the latest from upstream.

Create the Fork

1. Navigate to the repository to fork and choose fork.
2. Specify a name and choose the project where you want the fork to be created. If the repository contains many topic branches, we recommend you fork only the default branch.
3. Choose the ellipsis, then Fork to create the fork.



Note

You must have the Create Repository permission in your chosen project to create a fork. We recommend you create a dedicated project for forks where all contributors have the Create Repository permission. For an example of granting this permission, see [Set Git repository permissions](#).

Clone your fork locally

Once your fork is ready, clone it using the command line or an IDE like Visual Studio. The fork will be your origin remote.

For convenience, after cloning, you'll want to add the upstream repository (where you forked from) as a remote named upstream.

```
git remote add upstream {upstream_url}
```

Make and push changes

It's possible to work directly in main - after all, this fork is your copy of the repo.

We recommend you still work in a topic branch, though.

It allows you to maintain multiple independent workstreams simultaneously.

Also, it reduces confusion later when you want to sync changes into your fork.

Make and commit your changes as you normally would. When you're done with the changes, push them to origin (your fork).

Create and complete a PR

Open a pull request from your fork to the upstream. All the policies required reviewers and builds will be applied in the upstream repo. Once all policies are satisfied, the PR can be completed, and the changes become a permanent part of the upstream repo.

New Pull Request

Source: FabrikamFiber.jamal.fork
Branch: users/jamal/date-fix
into: FabrikamFiber / master

Title *
Added a new option to the settings page

Description
Added a new option to the settings page.
Added a new option for users to manage delivery preferences. Tested the following:
- [] Opt out of email
- [] Opt into email
- [] Specify an alternate address
Markdown supported.

Reviewers
[FabrikamFiber]\FabrikamFiber Team

Work Items

Important

Anyone with the Read permission can open a PR to upstream. If a PR build pipeline is configured, the build will run against the code introduced in the fork.

Sync your fork to the latest

When you've gotten your PR accepted into upstream, you'll want to make sure your fork reflects the latest state of the repo.

We recommend rebasing on upstream's main branch (assuming main is the main development branch).

```
git fetch upstream main
git rebase upstream/main
git push origin
```

The forking workflow lets you isolate changes from the main repository until you're ready to integrate them. When you're ready, integrating code is as easy as completing a pull request.

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Describe inner source with forks

Completed

- 4 minutes

People fork repositories when they want to change the code in a repository they don't have to write access to.

If you don't have write access, you aren't part of the team contributing to that repository, so why would you modify the code repository?

We tend to look for technical reasons to improve something in our work.

You may find a better way to implement the solution or enhance functionality by contributing to or improving an existing feature.

You can fork repositories in the following situations:

- I want to make a change.
- I think the project is exciting and may wish to use it.
- I want to use some code in that repository as a starting point for my project.

Software teams are encouraged to contribute to all projects internally, not just their software projects.

Forks are a great way to foster a culture of inner open source.

Forks are a recent addition to the Azure DevOps Git repositories.

This recipe will teach you to fork an existing repository and contribute changes upstream via a pull request.

Getting ready

A fork starts with all the contents of its upstream (original) repository.

When you create a fork in Azure DevOps, you can include all branches or limit them to only the default branch.

A fork doesn't copy the permissions, policies, or build definitions of the repository being forked.

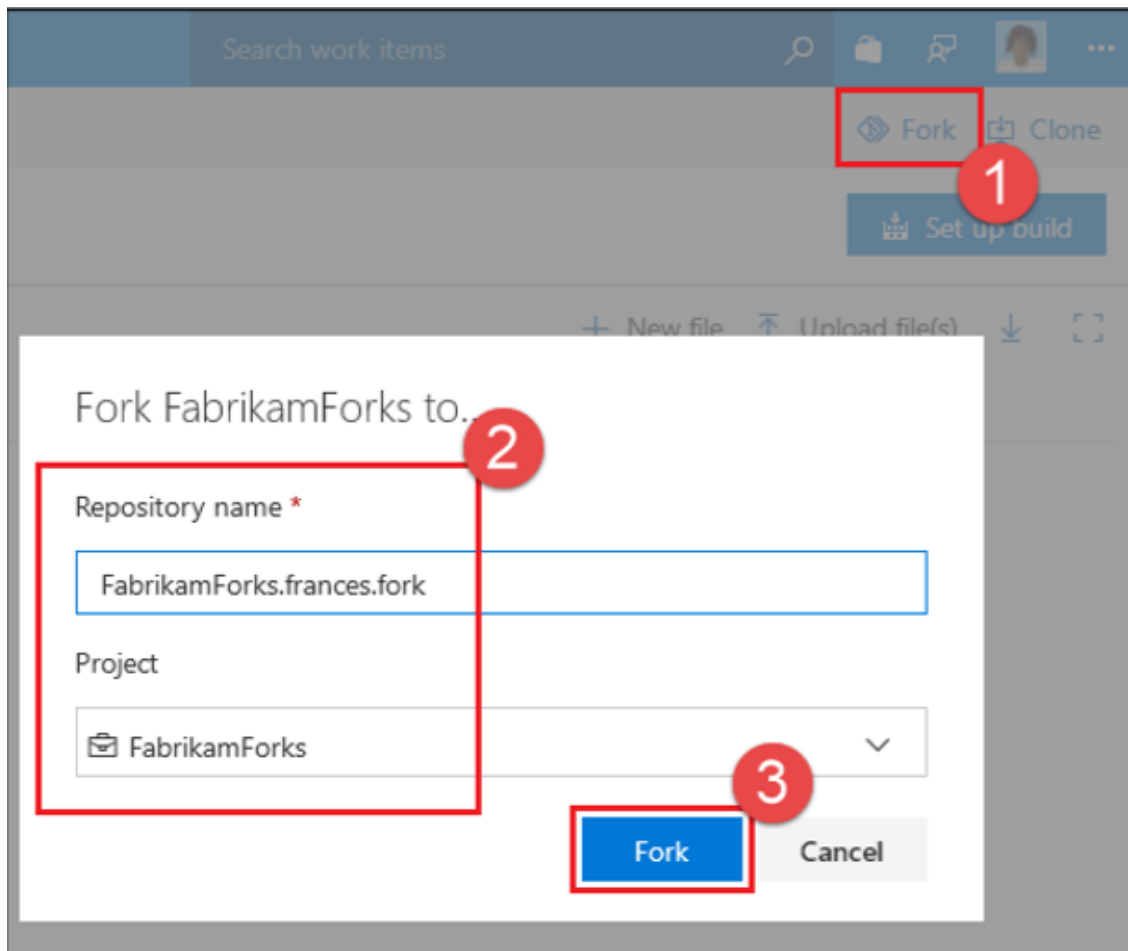
After a fork has been created, the newly created files, folders, and branches aren't shared between the repositories unless you start a pull request.

Pull requests are supported in either direction: from fork to upstream or upstream to fork.

The most common approach for a pull request will be from fork to upstream.

How to do it

1. Choose the Fork button (1), then select the project where you want the fork to be created (2). Give your fork a name and choose the Fork button (3).

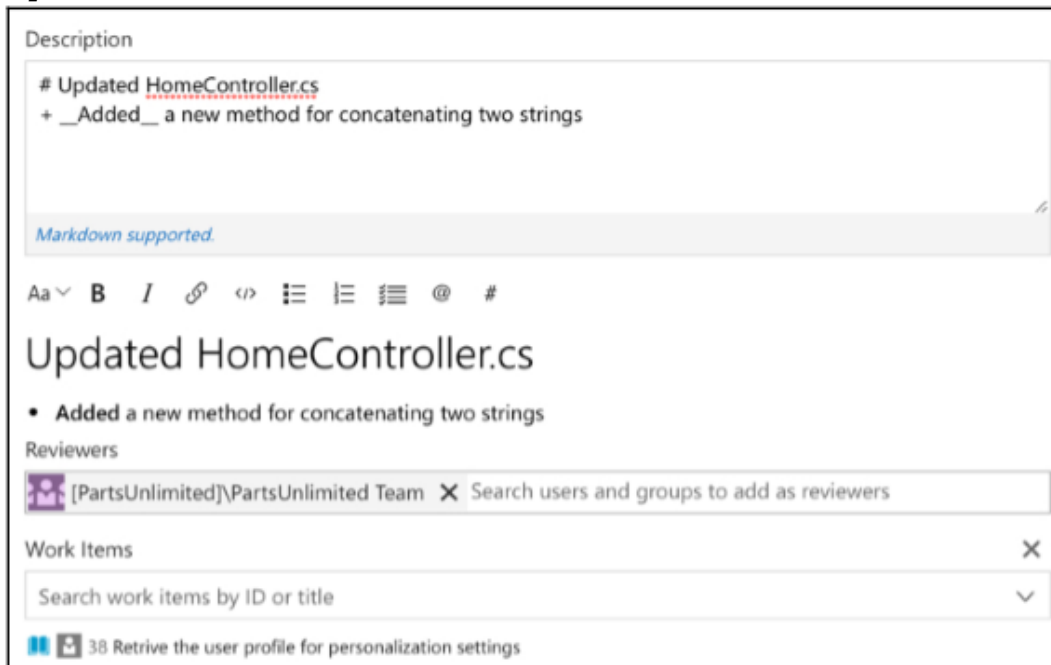


2. Once your fork is ready, clone it using the command line or an IDE, such as Visual Studio. The fork will be your origin remote. For convenience, you'll want to add the upstream repository (where you forked from) as a remote named upstream. On the command line, type:

```
git remote add upstream {upstream_url}
```

3. It's possible to work directly in the main – this fork is your copy of the repo. We recommend you still work in a topic branch, though. It allows you to maintain multiple independent workstreams simultaneously. Also, it reduces confusion later when you want to sync changes into your fork. Make and commit your changes as you normally would. When you finish the modifications, push them to the origin (your fork).
4. Open a pull request from your fork to the upstream. The upstream repo will apply all the policies required for reviewers and builds. Once all the policies are satisfied, the PR can be completed, and the changes become a permanent part of the upstream

repo:



Description

Updated HomeController.cs
+ _Added_ a new method for concatenating two strings

Markdown supported.

Aa B I #

Updated HomeController.cs

- Added a new method for concatenating two strings

Reviewers

[PartsUnlimited]\PartsUnlimited Team X Search users and groups to add as reviewers

Work Items X

Search work items by ID or title v

38 Retrive the user profile for personalization settings

5. When your PR is accepted upstream, you must ensure your fork reflects the latest repo state. We recommend rebasing on the upstream's main branch (assuming the main is the main development branch). On the command line, run:

```
git fetch upstream main
git rebase upstream/main
git push origin
```

For more information about Git, see:

- [Clone an Existing Git repo](#).
- [Azure Repos Git Tutorial](#).

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

[Knowledge check](#)

Completed

- 4 minutes

Choose the best response for each question. Then select **Check your answers** .

Check your knowledge

1.

Which of the following is a correct statement about a fork?

☐

A fork is a copy of a repository.

☐

A fork is a branch for future development.

☐

A fork is a branching strategy when working with centralized version control.

2.

Which of the following choices are the correct-forking workflow steps?

☐

Create a fork, make your changes locally and push them to a branch, create and complete a PR to the upstream.

☐

Create a fork, clone it locally, make your changes locally, push them to a branch, create and complete a PR to the upstream, and sync your fork to the latest from upstream.

☐

Create a fork, clone it locally, make your changes locally, create and complete a PR to the upstream, push the changes to a branch.

3.

Which of the following choices is the recommended team size to use a single repo instead of fork workflow?

☐

Large teams.

☐

10-20 developers.

☐

2-5 developers.

Check your answers

You must answer all questions before checking your work.

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

[Summary](#)

Completed

- 1 minute

This module explained how to use Git to foster inner source and implement Fork and its workflows.

You learned how to describe the benefits and usage of:

- Use Git to foster inner source across the organization.
- Implement fork workflow.
- Choose between branches and forks.
- Share code between forks.

Learn more

- [Fork your repository - Azure Repos | Microsoft Docs](#).
- [Clone an existing Git repo - Azure Repos | Microsoft Docs](#).

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Manage Git repositories

Introduction

Completed

- 1 minute

This module explores how to work with large repositories and purge repository data.

Learning objectives

After completing this module, students and professionals can:

- Understand large Git repositories.
- Explain Git Virtual File System (GVFS).
- Use Git Large File Storage (LFS).
- Purge repository data.

Prerequisites

- Understanding of what DevOps is and its concepts.
- Familiarity with version control principles is helpful but isn't necessary.
- Beneficial to have experience in an organization that delivers software.

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Work with large repositories

Completed

- 3 minutes

Git is a great version control system widely adopted and recommended, but a few concerns should be made and taken care of when working with large repositories.

While having a local copy of repositories in a distributed version control system is functional, that can be a significant problem when large repositories are in place.

For example, Microsoft discovered this issue when migrating a repository with over 300 GB of data from an internal system to Git.

Why repositories become large

There are two primary causes for large repositories:

- Long history
- Large binary files

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]
```

You can also reduce clones by filtering branches or cloning only a single branch.

VFS for Git

VFS for Git helps with large repositories. It requires a Git LFS client.

Typical Git commands are unaffected, but the Git LFS works with the standard filesystem to download necessary files in the background when you need files from the server.

The Git LFS client was released as open-source. The protocol is a straightforward one with four endpoints similar to REST endpoints.

For more information on large repositories, see: [Working with large files](#) and [Virtual File System for Git: Enable Git at Enterprise Scale](#).

Scalar



Scalar is a .NET Core application available for Windows and macOS. With tools and extensions for Git to allow very large repositories to maximize your Git command

performance. Microsoft uses it for Windows and Office repositories.

If Azure Repos hosts your repository, you can clone a repository using the [GVFS protocol](#).

It achieves by enabling some advanced Git features, such as:

- *Partial clone*: reduces time to get a working repository by not downloading all Git objects right away.
- *Background prefetch*: downloads Git object data from all remotes every hour, reducing the time for foreground git fetch calls.
- *Sparse-checkout*: limits the size of your working directory.
- *File system monitor*: tracks the recently modified files and eliminates the need for Git to scan the entire work tree.
- *Commit-graph*: accelerates commit walks and reachability calculations, speeding up commands like git log.
- *Multi-pack-index*: enables fast object lookups across many pack files.
- *Incremental repack*: Repacks the packed Git data into fewer pack files without disrupting concurrent commands using the multi-pack-index.

Note

We update the list of features that Scalar automatically configures as a new Git version is released.

For more information, see:

- [microsoft/scalar: Scalar](#).
- [Introducing Scalar: Git at scale for everyone](#).

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

[Purge repository data](#)

Completed

- 2 minutes

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.

If you commit sensitive data (for example, password, key) to Git, it can be removed from history. Two tools are commonly used:

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.

Note

More details are in the repository [git-filter-repo](#).

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
```

The following bash shows how to find all the places that a file called passwords.txt exists in the repository. Also, to replace all the text in it, you can execute the **--replace-text** option:

```
$ bfg --replace-text passwords.txt
```

For more information, see:

[Quickly rewrite git repository history](#).

[Removing files from Git Large File Storage](#).

[Removing sensitive data from a repository](#).

[BFG Repo Cleaner](#).

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Manage releases with GitHub Repos

Completed

- 3 minutes

Part of the release process starts with your version control. You'll understand how to manage releases in the repository using GitHub.

In the following modules, you'll see details about deploying a piece of software after packaging your code, binary files, release notes, and related tasks.

Releases in GitHub are based on [Git tags](#). You can think of a tag as a photo of your repository's current state. If you need to mark an essential phase of your code or your following deliverable code is done, you can create a tag and use it during the build and release process to package and deploy that specific version. For more information, see [Viewing your repository's releases and tags](#).

When creating new releases with release notes, it's possible to @mentions contributors, add links to binary files and edit or delete existing releases.

Jan 20, 2022

azure-pipelines-bot

v2.198.1

eaf95e8

Compare

v2.198.1 Pre-release

Features

- [macOS] Return the correct minor version on 11 and later OS. (#3605)
- Add Event Log dumping (#3653)
- Add passive validation - list local group memberships (#3673)
- Added RepoType telemetry for checkout task (#3677)
- Enabling validation of checksum for online agent update for ADO OnPrem (#3679)
- Add function to read waagent.conf settings and condition to run it only on Linux (#3680)
- Add passive validation - dumping cloud-init logs (#3681)
- Added masking for environment variables containing credentials in diagnostic logs (#3682)
- Download TEE plugin conditionally during checkout (#3684)
- Added information about user groups into environment file (Linux, MacOS) (#3690)
- Add cloud-init logs to diagnostics archive (#3700)

Bugs

- Porting logic of handling negative patterns for DownloadBuildArtifacts task (#3664)
- Added tests for ported logic of handling negative patterns for DownloadBuildArtifacts task (#3665)
- Adding support of negative patterns for DownloadBuildArtifacts task in scenarios with using file share (#3666)
- Fix permissions setting on MacOS while downloading TEE (#3704)

Agent Downloads

	Package	SHA-256
Windows x64	vsts-agent-win-x64-2.198.1.zip	8c426cc43d23d709e4540b6152c57a08394738ee302e8a92c5f4f763da93feef
Linux x64	vsts-agent-linux-x64-2.198.1.tar.gz	0da3ac2dc74a271dc6718c0aa6057effa58280a8191bf20ad165d57810b42d9f

Assets 3

assets.json 3.46 KB

Source code (zip)

Source code (tar.gz)

Image reference: [Releases · Microsoft/azure-pipelines-agent \(github.com\)](#)

Also, you can:

- Publish an action from a specific release in GitHub Marketplace.
- Choose whether Git LFS objects are included in the ZIP files and tarballs GitHub creates for each release.
- Receive notifications when new releases are published in a repository.

Creating a release

To create a release, use the `gh release create` command. Replace the **tag** with the desired tag name for the release and follow the interactive prompts.

```
gh release create tag
```

To create a prerelease with the specified title and notes.

```
gh release create v1.2.1 --title
```

If you @mention any GitHub users in the notes, the published release on GitHub.com will include a Contributors section with an avatar list of all the mentioned users.

You can check other commands and arguments from the [GitHub CLI manual](#).

Editing a release

You can't edit Releases with GitHub CLI.

To edit, use the Web Browser:

1. Navigate to the main repository page on GitHub.com.
2. Click **Releases** to the right of the list of files.
3. Click on the **edit icon** on the right side of the page, next to the release you want to edit.
4. Edit the details for the release, then click **Update release**.

Deleting a release

To delete a release, use the following command, replace the **tag** with the release tag to delete, and use the `-y` flag to skip confirmation.

```
gh release delete tag -y
```

For more information, see:

- [Managing releases in a repository - GitHub Docs](#) - If you want to perform the same steps from Web Browser instead of GitHub CLI.
- [Publishing an action in the GitHub Marketplace](#).
- [Managing Git LFS objects in archives of your repository](#).

- [Viewing your subscriptions](#).

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

[Automate release notes with GitHub](#)

Completed

- 2 minutes

After learning how to create and manage release tags in your repository, you'll learn how to configure the automatically generated release notes template from your GitHub releases.

You can generate an overview of the contents of a release, and you can also customize your automated release notes.

It's possible to use labels to create custom categories to organize pull requests you want to include or exclude specific labels and users from appearing in the output.

Creating automatically generated release notes

While configuring your release, you'll see the option Auto-generate release notes to include all changes between your tag and the last release. If you never created a release, it will consist of all changes from your repository.

Releases Tags

Choose a tag Target: main

Choose an existing tag, or create a new tag when you publish this release.

Release title

Write Preview

H B I [list icon] <> [link icon] [checkbox icon] [at icon] [share icon] [undo icon]

+ Auto-generate release notes

Describe this release

Attach files by dragging & dropping, selecting or pasting them.

Attach binaries by dropping them here or selecting them.

☐ This is a pre-release
We'll point out that this release is identified as non-production ready.

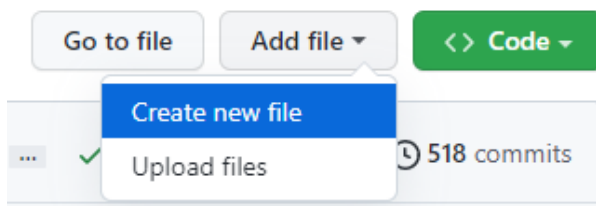
Publish release Save draft

You can choose if you want to customize it or leave it as it is.

Configuring automatically generated release notes template

You can customize the auto-generate release notes template by using the following steps.

1. Navigate to your repository and create a new file.



2. You can use the name **.github/release.yml** to create the **release.yml** file in the **.github** directory.

devops-journey / .github / in main

<> Edit new file

👁 Preview

1


3. Specify in YAML the pull request labels and authors you want to exclude from this release. You can also create new categories and list the pull request labels in each. For more information about configuration options, see [Automatically generated release notes - GitHub Docs](#).

Example configuration:

```
# .github/release.yml

changelog:
  exclude:
    labels:
      - ignore-for-release
    authors:
      - octocat
  categories:
    - title: Breaking Changes
      labels:
        - Semver-Major
        - breaking-change
    - title: Exciting New Features
      labels:
        - Semver-Minor
        - enhancement
    - title: Other Changes
      labels:
        - *
```

4. Commit your new file.



Commit new file

Create release.yml

Add an optional extended description...

☒ Commit directly to the main branch.
☐ Create a new branch for this commit and start a pull request. [Learn more about pull requests](#).

Commit new file Cancel

5. Try to create a new release and click + **Auto-generate release notes** to see the template structure.

For more information, see:

- [About releases - GitHub Docs](#)
- [Linking to releases - GitHub Docs](#)
- [Automation for release forms with query parameters - GitHub Docs](#)

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

[Knowledge check](#)

Completed

- 4 minutes

Choose the best response for each question. Then select **Check your answers** .

Check your knowledge

1.

Which of the following choices is for working with large files in repositories?

☐

Package Management.

☐

Git LFS.

☐

Git.

2.

Which of the following choices isn't a common situation when you need to purge data from repositories?

☐

When you need to reduce the size of a repository significantly by removing history.

☐

When you need to remove a sensitive file that should not have been uploaded.

☐

When you need to hide code and history without removing permissions.

3.

Which of the following choices is the built-in Git command for removing files from the repository?

☐

git remove-branch command.

☐

git filter-branch command.

☐

git purge-branch command.

You must answer all questions before checking your work.

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

[Summary](#)

Completed

- 1 minute

This module explored how to work with large repositories and purge repository data.

You learned how to describe the benefits and usage of:

- Understand large Git repositories.
- Explain Git Virtual File System (GVFS).
- Use Git Large File Storage (LFS).
- Purge repository data.

Learn more

- [Get started with Git and Visual Studio - Azure Repos | Microsoft Docs](#).
- [Using Git LFS and VFS for Git introduction - Code With Engineering Playbook \(microsoft.github.io\)](#).

- [Work with large files in your Git repo - Azure Repos | Microsoft Docs](#).
- [Delete a Git repo from your project - Azure Repos | Microsoft Docs](#).

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .