2020

Git branching strategies

Determine a workflow to deliver better products faster

- ✓ Centralized workflow
- ✓ Feature branching
- ✓ Personal branching
- ✓ GitFlow
- ✓ GitLab Flow

GitLab

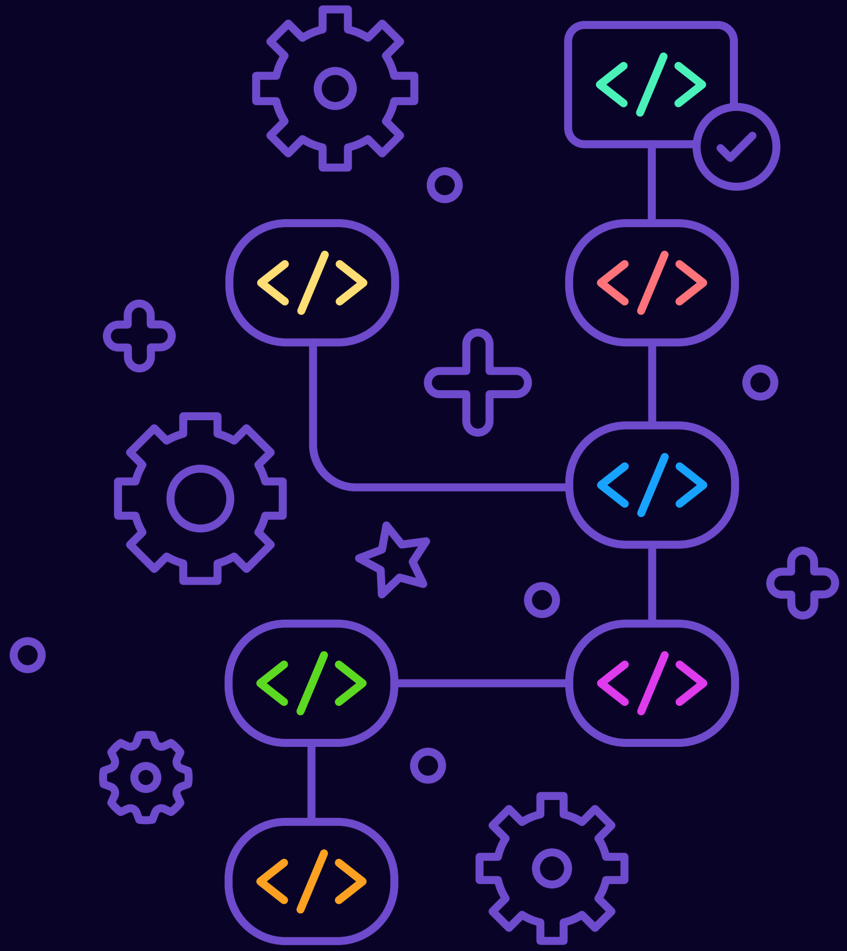


Table of contents

Start your GitLab free trial

Introduction

Software development is a booming business. Teams of various sizes are expected to deliver better quality software at a faster pace. To effectively work as a team, source control is invaluable. With Git, teams can work together using various branching strategies that were not possible before.

Git is a distributed version control system created by Linus Torvalds in 2005. It replaced the centralized version control systems that were prevalent at the time, such as Subversion. By 2014, Git was officially the most used source control management (SCM) tool. By 2018, 87.2% of developers in the yearly Stack Overflow survey said they used Git for version control.

Git is a great tool for tracking source code changes, but it's process agnostic. It doesn't tell you what you should or shouldn't do. And even though many developers use it every day, a lot of them don't get the maximum benefits from it.

They cling to the old ways of Subversion, which was centralized and hard to branch.

This eBook explores the factors impacting the choice of a branching strategy and investigates the benefits and drawbacks of several branching strategies.

Development dilemma

Let's start by briefly discussing what branching is and why you should think about it. Whether you realize it or not, all source control systems have at least one branch — the `master` branch where teams put their code and commit any changes.

Branching allows one team member to make a change and commit that to the server so another can download it and work with the changes. When the changes are approved, a build of the code on the `master` branch can then be deployed to a test or production environment.

Now, let's say a team member is working on a non-trivial change that may take a few days or even weeks.

This change is built over multiple commits, but the moment the change is committed to the `master` branch, it can't be released to production anymore, because it contains an incomplete feature.

One solution to this is to build a feature switch, a simple Boolean that indicates whether the new code should be executed. Only after the entire feature is complete will the Boolean be set to true.

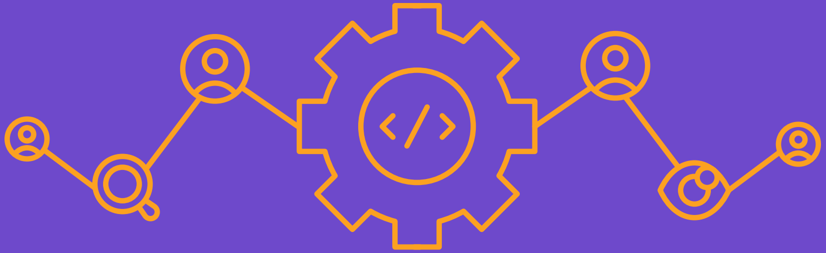
Feature switches have a few problems, though. First, you may have to write duplicate code. Let's say you have a formula that's slightly altered. It would make sense to copy the formula and only execute the new one if the feature switch is set to true. But any current development now has to take the new code branch into account. Second, it's easy to forget a switch, meaning the incomplete feature ends up in production after all. Finally, your code will be littered with if-else branches that are no longer necessary.

You might then consider the use of interfaces in code. This basically means that you switch out one implementation of the code with another. However, this has limited use cases (e.g. when you're going to use a new caching mechanism, a new API, or even a new database). Of course, this means your code has to be completely modular and adhere to object-oriented best practices. This is rarely the case in the real world, especially with older systems.

Branching: The solution to a seamless workflow

With branching, a team member creates a new branch — a `feature` branch — and develops a new feature on this branch. Meanwhile, another team member can continue working on the `master` branch, but doesn't see the teammate's changes.

Team members must make sure to keep up with changes to `master`, because changes to `master` may affect their feature as well. To do this, team members pull the `master` branch into their own `feature` branch on a daily basis. Any conflicts can be resolved as soon as possible. After someone finishes their feature, they can merge it back into the `master` branch so others can see the changes as well and pull them into their own `feature` branches.



By using a branching strategy, a team member is able to work on the `master` branch and release new versions of the software, while someone else can create the new feature. The code base is clean, and there's never any risk of an unfinished feature going into production.

Centralized versus decentralized

Next, let's discuss the difference between centralized and decentralized source control systems to better understand some of the benefits Git has over older systems and, in particular, why it handles branching so much better.

Centralized workflow

In a centralized source control system, everything is stored on a server. The server has the complete history and all branches stored as physical files. All branches are created and maintained on the server. If you want to work in a branch, you first have to create it on the server — which means copying all files from the main branch — and then download it to your own machine. Commits are sent to the server, and any merges are then resolved on the server.

Working with branches in a centralized source control system can be challenging, because you need access to the server, creating branches takes a long time, and merging back can be a difficult task.

Decentralized workflow

In contrast, a decentralized source control system brings a copy of the complete repository to your computer, which means you can commit, branch, and merge locally. It also means the server doesn't have to store a physical file for each branch — it just needs the differences between each commit.

With a decentralized workflow, a developer can go back a few commits, create a new branch, make a few commits, and merge it all back into a main branch. This workflow also enables offline development without loss of functionality. Where branching was best avoided before, it's a best practice now.

A decentralized way of working can be puzzling, especially if you're coming from centralized source code systems, because merge conflicts aren't handled by the server, but have to be resolved locally, which can result in confusing merge commits. Exacerbating the issue is that branches are not directly available to everyone on the team.

All in all, both systems look alike, but behave differently.

Terminology

Before we continue, let's solidify some terminology.

In Git, there are two repositories:

- **one remote** (on the server)
- **one local** (on your computer)

A **commit** is always done on your local repository, and you then **push** it to the remote repository. A commit is only visible to others after it's been pushed to the remote repository.

When you want to get the commits of your coworkers, you **fetch** or **pull** the branch. Fetching downloads the commits, but doesn't apply them to your branch. Ultimately, you have to pull the branch to apply the commits of your coworkers to your local branch. In practice, many developers rarely fetch and always pull straight away.

Branching strategies

Different branching strategies for Git vary between simple and convoluted. Some strategies make sense for small teams, while others work best for larger teams.

Branching strategies may also differ for particular applications. Large systems require different strategies from small systems, and strategies can also depend on how often a system changes.

How teams approach branching is an important part of finding success with source code management. Having the wrong strategy in place can severely limit your capacity and increase your chance of bugs. It's up to development teams to decide on a Git branching strategy that makes sense given team and organization goals and processes.

Start your GitLab free trial

Centralized workflow

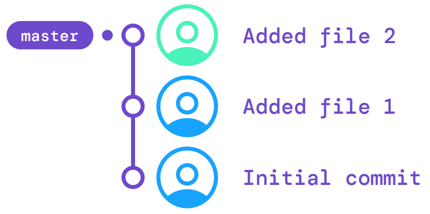
The first strategy we're going to discuss is the easiest, the centralized workflow. It's how Subversion used to work, so it's great for teams transitioning to Git.

Using a centralized workflow strategy, everyone commits to the `master` branch. You don't need any other branches. This strategy can work for small teams where people communicate and no two developers work on the same piece of code simultaneously. Communication is important to ensure this strategy is successful.

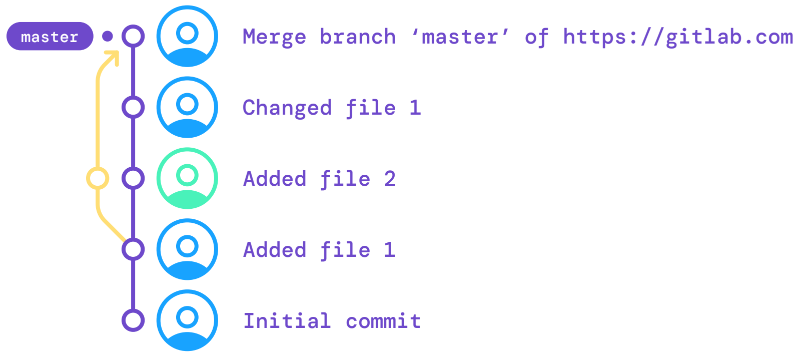
Centralized workflow is easy, but also limited. When larger teams constantly commit to the same branch, it becomes difficult to find a stable moment for releasing your changes. Basically, it means you can't commit any unstable changes and you have to keep them local until they're ready for release.

Let's look at a simple example.

If team member Alice creates a repository and makes a commit on it, teammate Bob then pulls in those changes and makes another commit.



After that, it gets interesting when Alice makes a change and commits, but doesn't pull in Bob's changes first. Git now automatically creates a merge commit.



What happens now is the following: Alice's local repository is at commit x, but the remote repository is at commit y. The missing commits have to be pulled from the remote repository, after which Git creates a separate branch and immediately merges it back with the current branch (`master`).

Bob's change now looks as if it was committed on a branch and then merged back to `master` after Alice's commit.

Alice can't push changes to the remote repository until all the remote commits are merged on their local branch, so any merge conflicts would have to be solved first. This process can be very confusing for new Git users coming from Subversion. Luckily, Git handles most of this automatically.

To avoid automatic merges, it's important for new Git users is to always pull code before you commit. But this poses a new challenge: You can't always pull when you have uncommitted changes. Git is careful about touching your work and when you haven't committed yet.

What you can do, though, is **stash** your changes (similar to shelving in Subversion). This will set your changes aside so you have no open changes. You can then pull to synchronize your branch. After that, you apply your stash to your branch and Git will try to merge where possible or give you merge conflicts that you can then change.

A word of warning here: You might be tempted to think of your stashed code as "mine," but it's actually "theirs," since it isn't on your branch yet. After you've applied your stash and solved any merge conflicts, you can just commit as usual and you won't have automatic merge commits (unless someone pushed their changes while you were applying your stash). Automatic merge commits aren't a bad thing, just possibly confusing to new users.

Because this strategy is simple, it is well-suited for small teams, Git beginners, and projects that don't get a lot of updates.

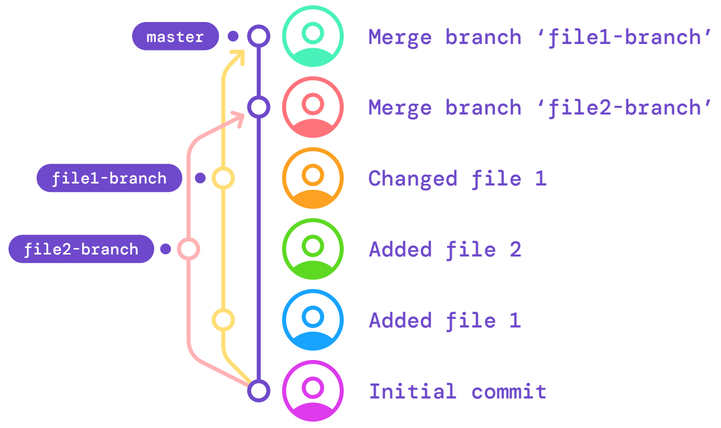
Now that you've seen the simplest of branch strategies, let's look at a more difficult strategy.

Start your GitLab free trial

Feature branching

With feature branching, every feature gets its own branch. There's still the `master` branch, but you don't commit directly to it. Instead, whenever you need to make a change, you create a branch, make your changes there, and then merge it into `master`.

For example, suppose the `master` branch is created and both Alice and Bob create a separate branch from it. Alice and John are going to work on file 1, while Bob is going to work on file 2. First, Alice creates file 1 and commits to the `file1` branch. Then Bob creates file 2, commits this to the `file2` branch, and then merges the branch back to `master`. Meanwhile, John changes file 1, commits this to the `file1` branch, and then merges the branch back to `master`.



In this example, additional merge commits are created for clarity. Alternatively, Bob's branch could've been appended to the `master` like a regular commit. Also, John updated his `master` branch before merging the `file2` branch. If John didn't, he'd get an automatic merge commit, just as you saw in the centralized workflow example. Once you're done with a feature, you can remove the branches if you like.

This branching strategy is slightly more complicated than the centralized workflow, but it allows multiple people to work on the same feature. Also, the `master` branch isn't "polluted" with features that aren't finished yet.

Teams of any size can adopt this strategy after they've familiarized themselves with Git using the centralized workflow. Because it allows multiple people to work on the same software and features simultaneously, this strategy is well-suited for software that is still in development, but can be used beyond that stage as well.

Start your GitLab free trial

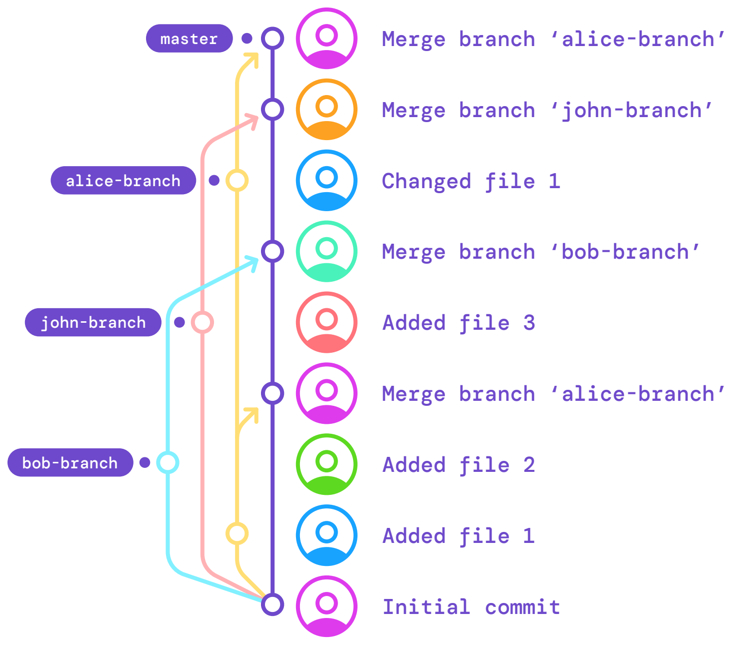
Personal branching

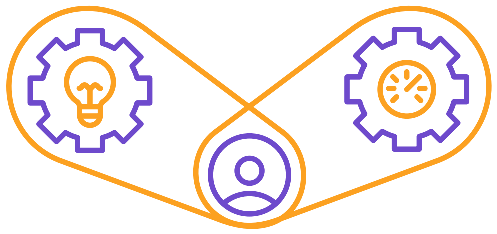
Personal branching is a lot like feature branching, except you don't branch per feature, but per developer. This can work well if people work on different features and bugs. Every user can merge back to the `master` branch whenever their work is done.

This strategy has the same benefits as feature branching except that it's harder for two developers to work on the same feature. Also, if you're working on feature A and you're assigned to feature B, there's no way to easily merge back feature A without polluting the `master` branch with an incomplete feature. On the plus side, branch management is slightly easier because there are simply fewer branches.

In the following example, we see Alice working on file 1, Bob working on file 2, and John working on file 3.

(Please note: merge commits have been removed for clarity.)


master
alice-branch
john-branch
bob-branch
Merge branch 'alice-branch'
Merge branch 'john-branch'
Changed file 1
Merge branch 'bob-branch'
Added file 3
Merge branch 'alice-branch'
Added file 2
Added file 1
Initial commit



Combining personal branching with feature branching can help work around some problems. You could create a feature branch A from your personal branch. Now, if feature B gets priority, you can easily create a new branch for B and work on that. When you're done, merge it back to your personal branch and then back to the `master` branch.

Your feature branches can stay local or push them to the remote. Meanwhile, you can use your personal branch for bug fixes and other small changes.

This strategy can work well for small teams in which every team member develops their own part of the application.

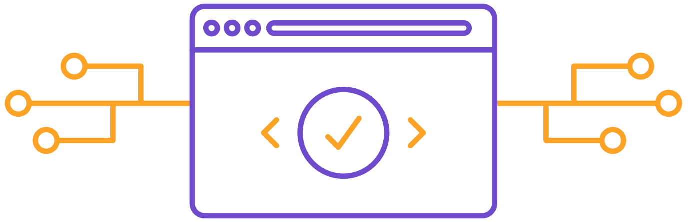
GitFlow

GitFlow is by far the most complicated branching strategy we'll discuss (and one of the most complex in general). The main takeaway is that your `master` branch should always be releasable to production.

There can be no untested or incomplete code on your `master` branch. Basically, your `master` branch should always be a reflection of your production environment, except right before a release.

With GitFlow, no one ever commits to the `master` branch. Instead, you have what's known as a `develop` branch. On the `develop` branch, you use `feature` branches. Whenever the develop branch is ready to go to production, you create a new release branch. This `release` branch is meant for testing and bug fixing. Bug fixes can be merged back to the `develop` branch. Once you're satisfied with the `release` branch, it can be merged back to `master`. With this strategy, the `master` branch always reflects production.

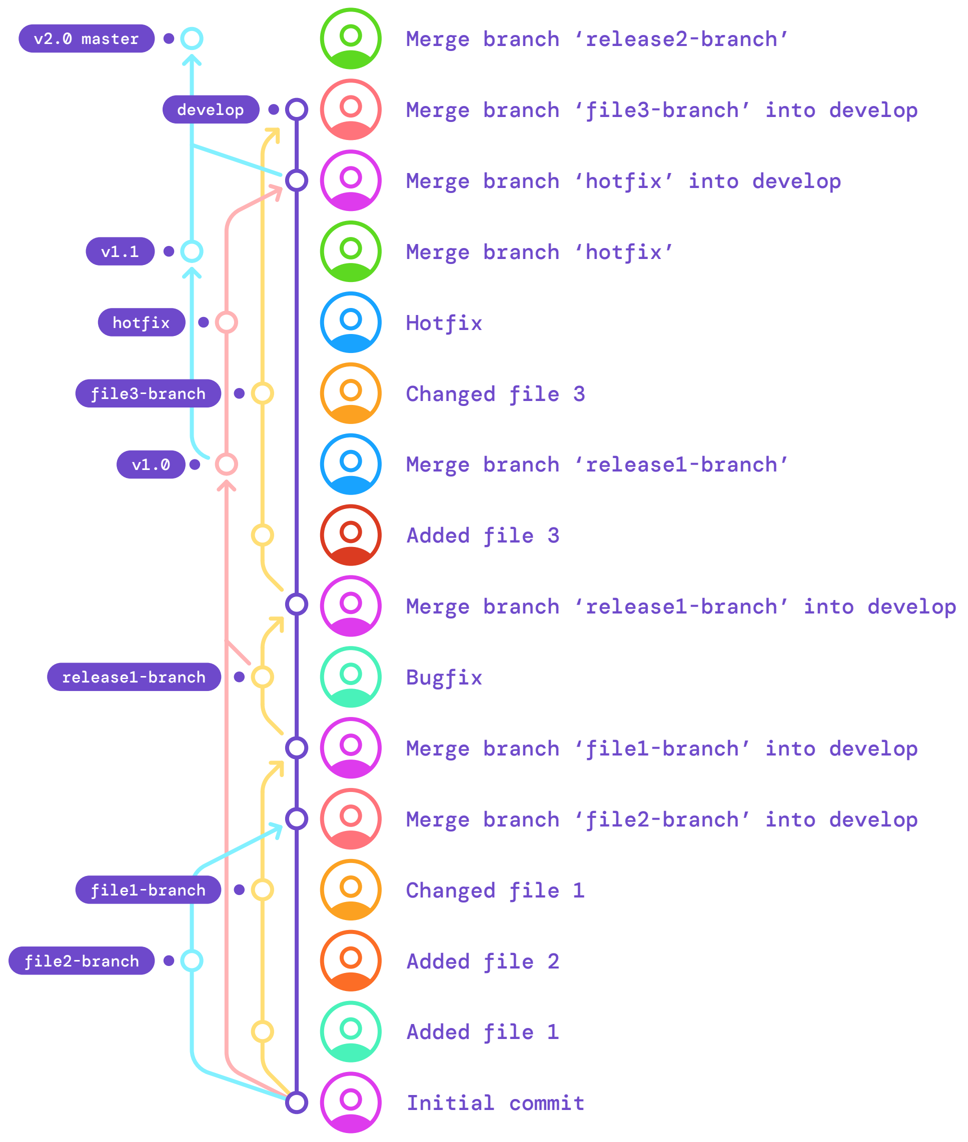
There's one exception to the above flow. When there's a blocking bug on production, you're allowed to branch from `master`, fix the bug, and merge it back to both `master` (directly) and `develop` (so it's not overwritten with the next release). You can use tags on the `master` branch to see which commit is what version.



Let's again look at an example.

We're going to work on a project and release v1.0, 1.1 (hotfix), and 2.0. First, Alice and John work on file 1, while Bob works on file 2. When both features are merged back to the `develop` branch, Alice creates a `release1` branch. Unfortunately, she finds a bug, so she creates a `bugfix` branch and merges it back to the `release1` branch and the `develop` branch. Meanwhile, Bob and John continue working on file 3. Alice merges the `release1` branch back to `master` and deploys the software as v1.0.

In the following graph, the `master` branch isn't visible until the v1.0 tag, where it seems to branch from the develop branch.



```
Merge branch 'release2-branch'
Merge branch 'file3-branch' into develop
Merge branch 'hotfix' into develop
Merge branch 'hotfix'
Hotfix
Changed file 3
Merge branch 'release1-branch'
Added file 3
Merge branch 'release1-branch' into develop
Bugfix
Merge branch 'file1-branch' into develop
Merge branch 'file2-branch' into develop
Changed file 1
Added file 2
Added file 1
Initial commit
```

After the release of v1.0, Alice finds a bug in production and creates a hotfix branch from `master`, pushes a fix, and merges it back into `develop` and `master`, creating a v1.1 release. Meanwhile, Bob and John have finished working on file 3 and merged it back into `develop`. Alice creates a `release2` branch and merges it into `master`, making a v2.0 release.

By now, it should be clear that GitFlow is a complicated strategy. It requires multiple branch rules, like not pushing to `master`, for it to work. Everyone on the team needs a thorough understanding of Git and branching and merging.

The benefit of GitFlow is that it allows larger teams to work on big, complex software while still being able to quickly fix bugs in production.In addition, the `release` branch allows for a staging period where testers and users can test the software before it's released, which doesn't hinder development in any way.

Teams of any size can reap the benefits of this approach, but because it is also quite complex, smaller teams may find one of the other strategies easier to use. If you are dealing with multiple environments and regular deployments, GitFlow may offer the flexibility you need.

Start your GitLab free trial

GitLab Flow

GitLab offers a simpler alternative to GitFlow: GitLab Flow

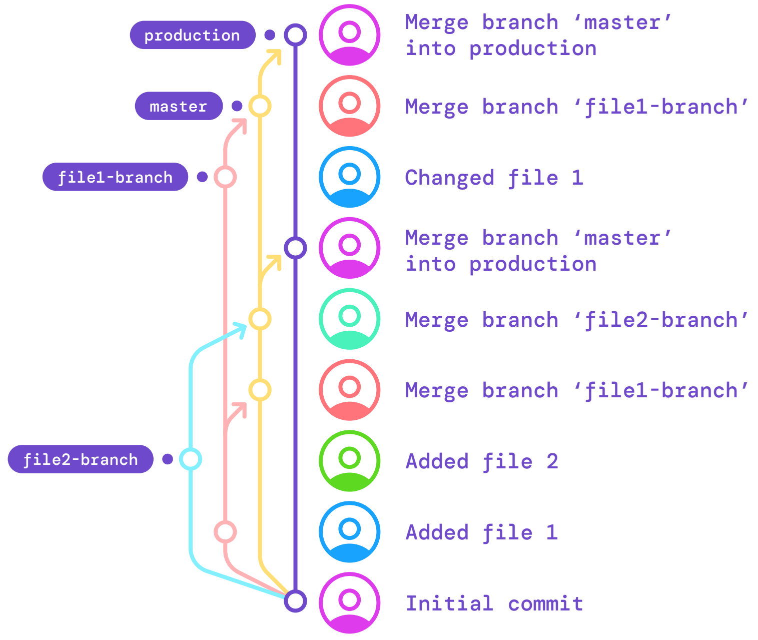
First of all, GitLab Flow assumes your `master` branch is your default branch, because it almost always is. Where GitFlow requires you to create a develop branch and make that your default, GitLab Flow works with the `master` branch right away.

The idea, then, is to practice feature branching, but also keep a separate production branch. Whenever your `master` branch is ready to be deployed, you merge it into the production branch and release that.

In this example, Alice creates a `file1` branch and Bob a `file2` branch. Once both branches are merged to `master`, the branch is merged to the production branch and it can be released.

After that, Alice makes an additional change to file 1, merges that back into `master`, and then into production, after which it can be released again.

This is fairly simple, but it has a downside: no release branch as in GitFlow. This branch allows us to test in a test or acceptance environment. With GitLab Flow, however, we can add a pre-production branch.

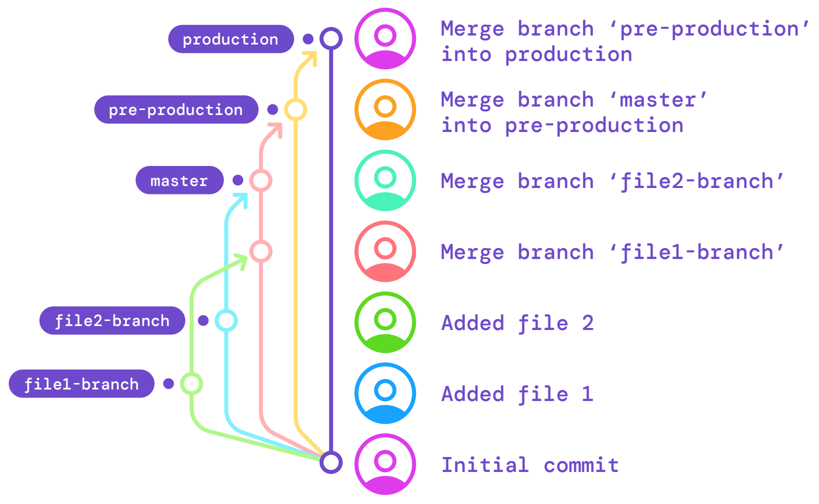

production
master
file1-branch
file2-branch
Merge branch 'master' into production
Merge branch 'file1-branch'
Changed file 1
Merge branch 'master' into production
Merge branch 'file2-branch'
Merge branch 'file1-branch'
Added file 2
Added file 1
Initial commit

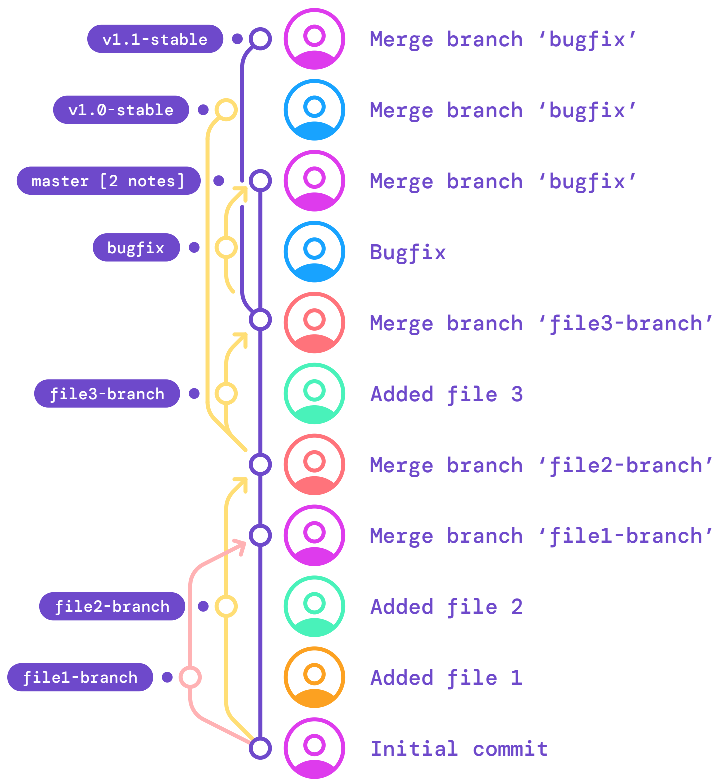
As you can see, we now have our release branch from GitFlow back. We can again use this to make bug fixes and merge those back to `master` before going to production. You can add as many pre-production branches as you like — for example, from `master` to test, from test to acceptance, and from acceptance to production. However, each new branch creates more work and complexity.

Finally, you can use GitLab Flow to work with release branches. For instance, when you have a public API, you might need to maintain different versions. Using GitLab Flow, you can simply make a `v1` branch and a `v2` branch, for example.

In the example, you see a `v1.0-stable` branch being created after the `file1` and `file2` branches are merged to `master`. After that, we create a `file3` branch, merge it to `master` and create a `v1.1-stable` branch. We now have two versions of the software we can maintain individually, which can be helpful if we find a bug that goes back to `v1.0`.

In this case, we create a bugfix branch from `master` and merge it back. After that, we don't merge the bugfix branch into `v1.0-stable` and `v1.1-stable`, but instead do a cherry-pick, meaning the bugfix commit is applied on `v1.0-stable` and `v1.1-stable`. That way, we don't have to merge back, taking the complete history into the other branches.


production
pre-production
master
file2-branch
file1-branch
Merge branch 'pre-production' into production
Merge branch 'master' into pre-production
Merge branch 'file2-branch'
Merge branch 'file1-branch'
Added file 2
Added file 1
Initial commit



Start your GitLab free trial

Using GitLab Flow, teams can cooperate and maintain various versions of software in different environments. GitLab Flow prevents the overhead of releasing, tagging, and merging (a challenge encountered with GitFlow) to create a more seamless way to deploy code.

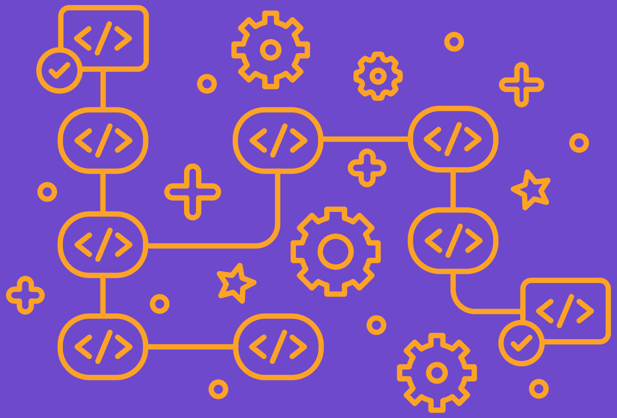
With GitLab Flow, commits only flow downstream, ensuring that every line of code is tested in all environments. It's suitable for teams of any size and has the flexibility to adapt to unique needs and challenges.

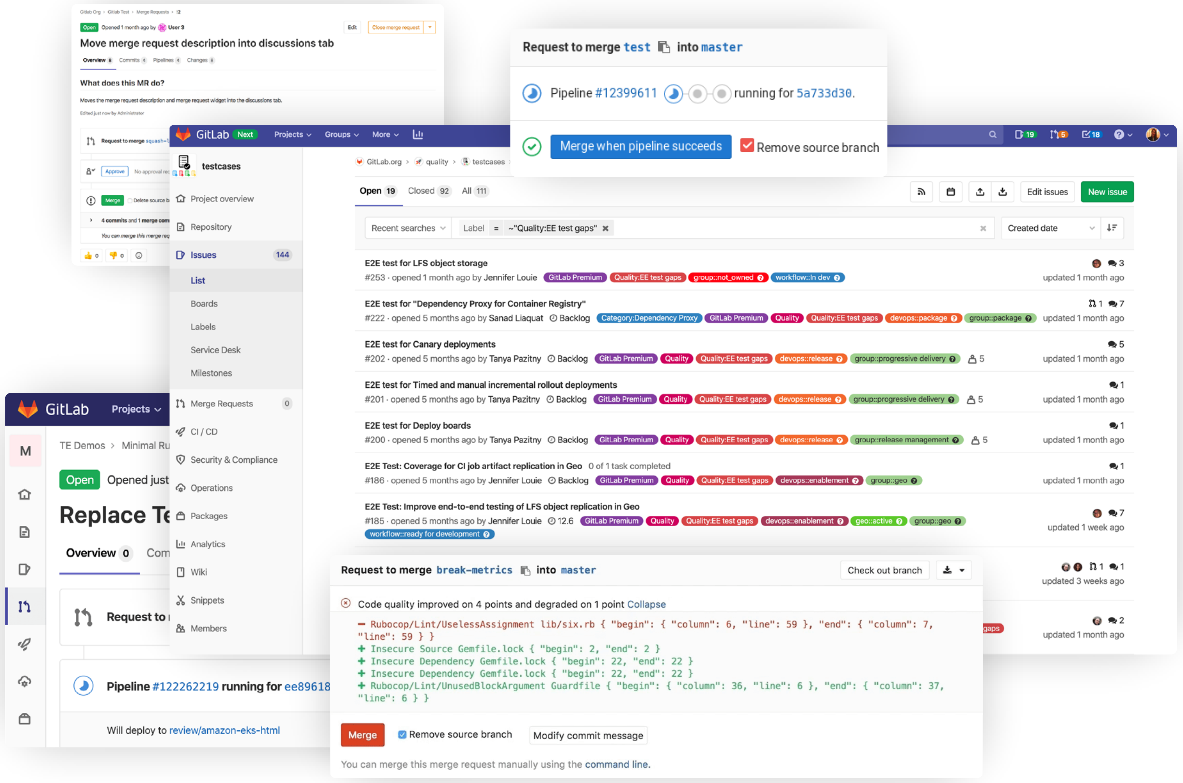
Start your GitLab free trial

Conclusion

We've looked at various Git branching strategies, and examined the pros and cons of some popular strategies.

Developers and teams can benefit from using GitLab Flow, because its simplicity and flexibility can boost productivity. Selecting the right branching strategy can propel development teams to faster delivery, stronger collaboration, and higher velocity.





E2E test for LFS object storage
#253 · opened 1 month ago by Jennifer Louie — GitLab Premium, Quality:EE test gaps, group::not_owned, workflow::In dev — updated 1 month ago

E2E test for "Dependency Proxy for Container Registry"
#222 · opened 5 months ago by Sanad Liaquat — Backlog — Category:Dependency Proxy, GitLab Premium, Quality, Quality:EE test gaps, devops::package, group::package — updated 1 month ago

E2E test for Canary deployments
#202 · opened 5 months ago by Tanya Pazitny — Backlog — GitLab Premium, Quality, Quality:EE test gaps, devops::release, group::progressive delivery — updated 1 month ago

E2E test for Timed and manual incremental rollout deployments
#201 · opened 5 months ago by Tanya Pazitny — Backlog — GitLab Premium, Quality, Quality:EE test gaps, devops::release, group::progressive delivery — updated 1 month ago

E2E test for Deploy boards
#200 · opened 5 months ago by Tanya Pazitny — Backlog — GitLab Premium, Quality, Quality:EE test gaps, devops::release, group::release management — updated 1 month ago

E2E Test: Coverage for CI job artifact replication in Geo 0 of 1 task completed
#186 · opened 5 months ago by Jennifer Louie — Backlog — GitLab Premium, Quality, Quality:EE test gaps, devops::enablement, group::geo — updated 1 month ago

E2E Test: Improve end-to-end testing of LFS object replication in Geo
#185 · opened 5 months ago by Jennifer Louie — 12.6 — GitLab Premium, Quality, Quality:EE test gaps, devops::enablement, geo::active, group::geo, workflow::ready for development — updated 1 week ago

Request to merge break-metrics into master

Code quality improved on 4 points and degraded on 1 point Collapse

```
− Rubocop/Lint/UselessAssignment lib/six.rb { "begin": { "column": 6, "line": 59 }, "end": { "column": 7,
"line": 59 } }
+ Insecure Source Gemfile.lock { "begin": 2, "end": 2 }
+ Insecure Dependency Gemfile.lock { "begin": 22, "end": 22 }
+ Insecure Dependency Gemfile.lock { "begin": 22, "end": 22 }
+ Rubocop/Lint/UnusedBlockArgument Guardfile { "begin": { "column": 36, "line": 6 }, "end": { "column": 37,
"line": 6 } }
```

Merge — Remove source branch — Modify commit message

You can merge this merge request manually using the command line.

Pipeline #122262219 running for ee89618

Will deploy to review/amazon-eks-html

Try GitLab free

All GitLab features — free for 30 days.
GitLab is more than just branching strategies. It is a full software development lifecycle & DevOps tool in a single application.

Start your free trial

About GitLab

GitLab is a DevOps platform built from the ground up as a single application for all stages of the DevOps lifecycle enabling Product, Development, QA, Security, and Operations teams to work concurrently on the same project.

GitLab provides a single data store, one user interface, and one permission model across the DevOps lifecycle. This allows teams to significantly reduce cycle time through more efficient collaboration and enhanced focus.

Built on Open Source, GitLab leverages the community contributions of thousands of developers and millions of users to continuously deliver new DevOps innovations.

More than 100,000 organizations from startups to global enterprises, including Ticketmaster, Jaguar Land Rover, NASDAQ, Dish Network, and Comcast trust GitLab to deliver great software faster.

GitLab is the world's largest all-remote company, with more than 1,250 team members in more than 65 countries and regions.

Learn more at GitLab.com