# Descriptions of Git Workflows

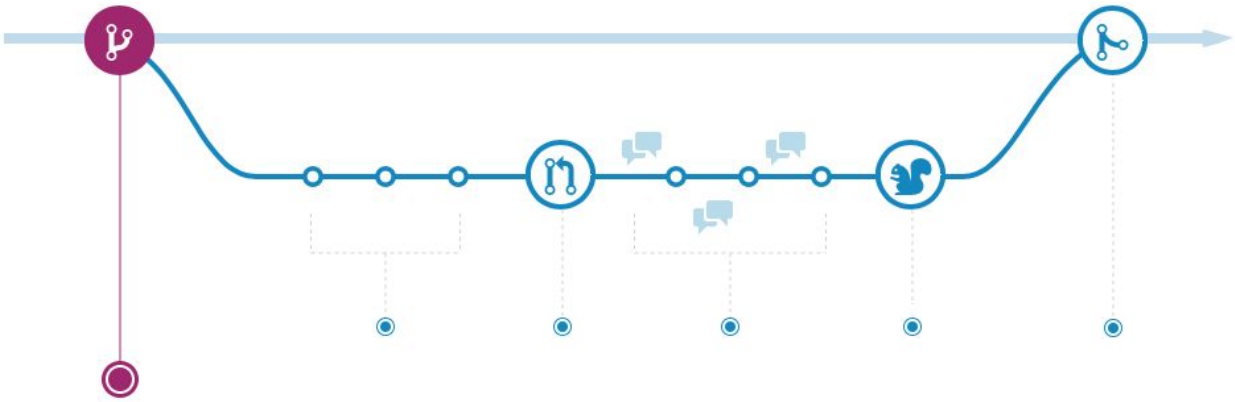
## GitHub Flow

---

**Description 1 (https://guides.github.com/introduction/flow/):**

GitHub flow is a lightweight, branch-based workflow that supports teams and projects where deployments are made regularly. This guide explains how and why GitHub flow works.

Create a branch



When you're working on a project, you're going to have a bunch of different features or ideas in progress at any given time – some of which are ready to go, and others which are not. Branching exists to help you manage this workflow.

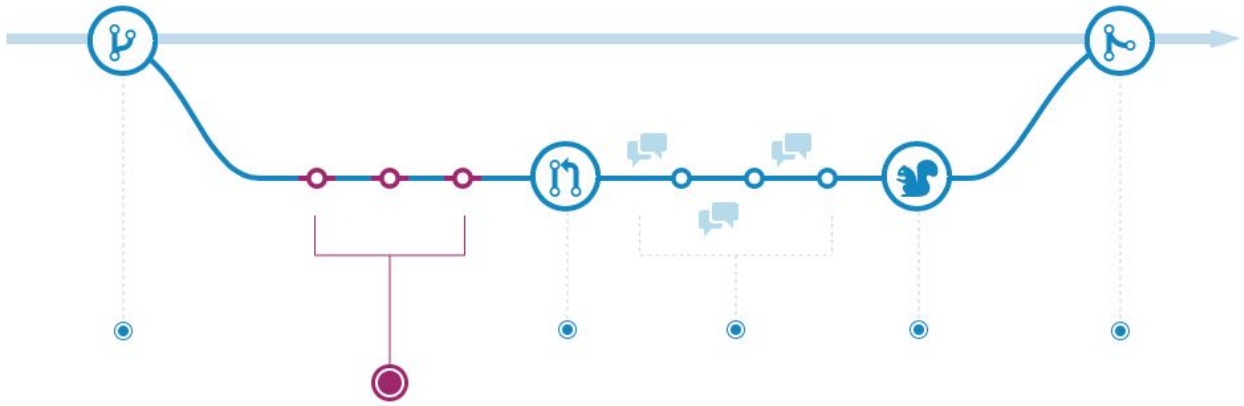
When you create a branch in your project, you're creating an environment where you can try out new ideas. Changes you make on a branch don't affect the master branch, so you're free to **experiment** and commit changes, safe in the knowledge that your branch won't be merged until it's ready to be reviewed by someone you're collaborating with.

ProTip

Branching is a core concept in Git, and the entire GitHub flow is based upon it. There's only one rule: anything in the master branch is always deployable.

Because of this, it's extremely important that your new branch is created off of master when working on a feature or a fix. Your branch name should be descriptive (e.g., refactor-authentication, user-content-cache-key, make-retina-avatars), so that others can see what is being worked on.

Add commits



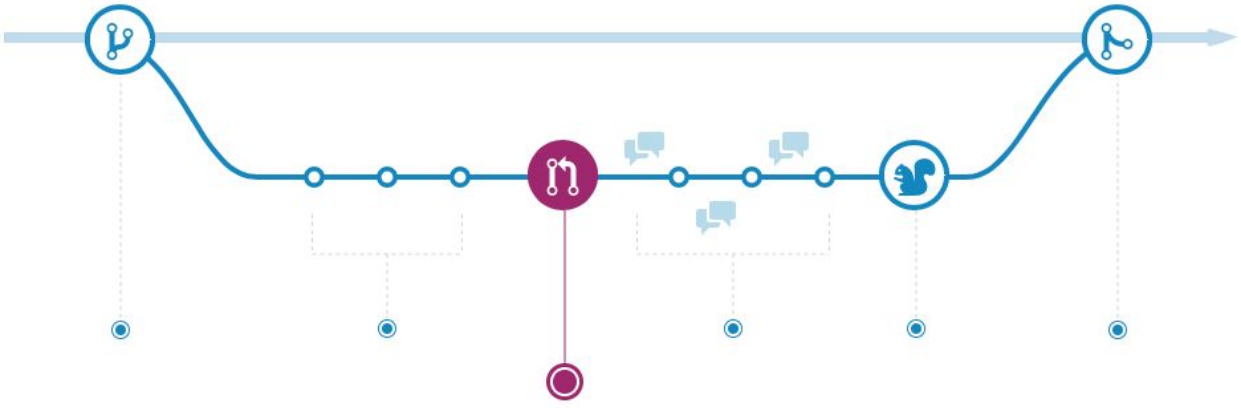
Once your branch has been created, it's time to start making changes. Whenever you add, edit, or delete a file, you're making a commit, and adding them to your branch. This process of adding commits keeps track of your progress as you work on a feature branch.

Commits also create a transparent history of your work that others can follow to understand what you've done and why. Each commit has an associated commit message, which is a description explaining why a particular change was made. Furthermore, each commit is considered a separate unit of change. This lets you roll back changes if a bug is found, or if you decide to head in a different direction.

ProTip

Commit messages are important, especially since Git tracks your changes and then displays them as commits once they're pushed to the server. By writing clear commit messages, you can make it easier for other people to follow along and provide feedback.

Open a Pull Request



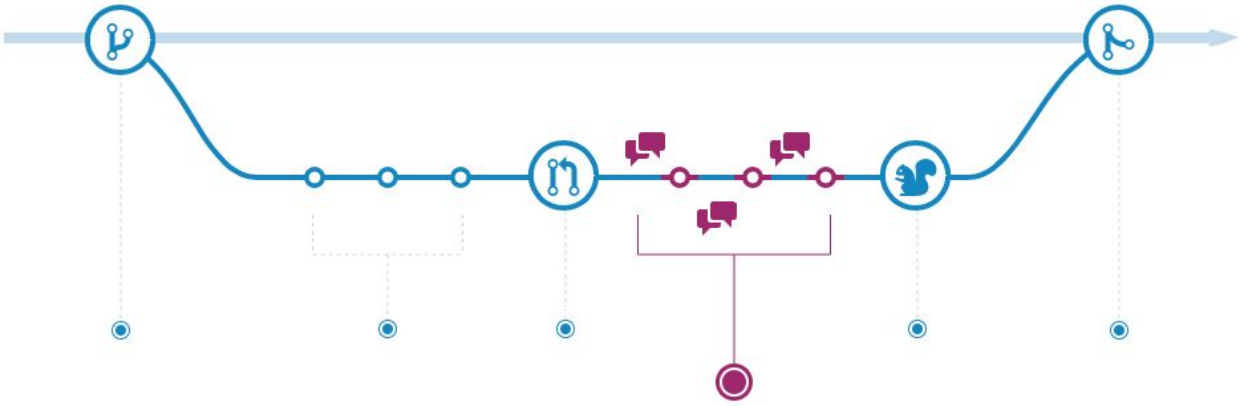
Pull Requests initiate discussion about your commits. Because they're tightly integrated with the underlying Git repository, anyone can see exactly what changes would be merged if they accept your request.

You can open a Pull Request at any point during the development process: when you have little or no code but want to share some screenshots or general ideas, when you're stuck and need help or advice, or when you're ready for someone to review your work. By using GitHub's @mention system in your Pull Request message, you can ask for feedback from specific people or teams, whether they're down the hall or ten time zones away.

ProTip

Pull Requests are useful for contributing to open source 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 master branch.

Discuss and review your code



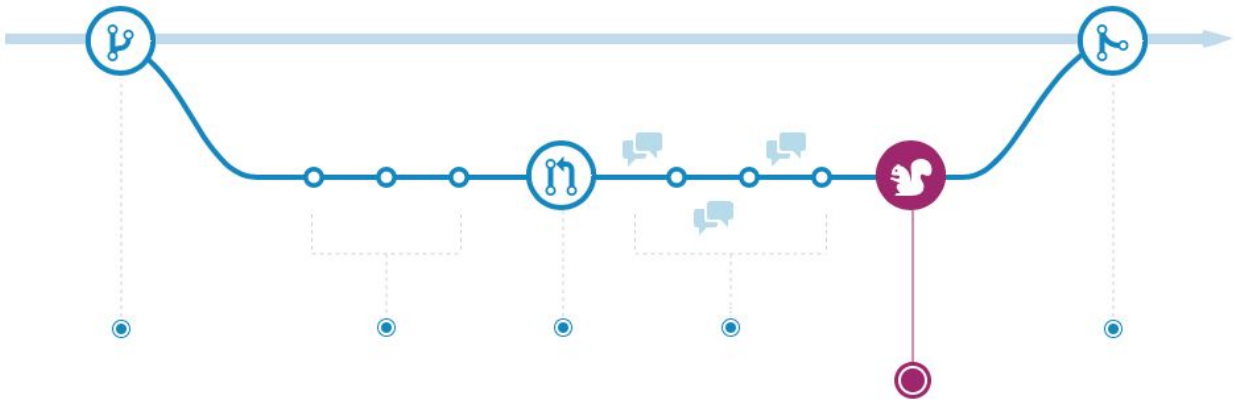
Once a Pull Request has been opened, the person or team reviewing your changes may have questions or comments. Perhaps the coding style doesn't match project guidelines, the change is missing unit tests, or maybe everything looks great and props are in order. Pull Requests are designed to encourage and capture this type of conversation.

You can also continue to push to your branch in light of discussion and feedback about your commits. If someone comments that you forgot to do something or if there is a bug in the code, you can fix it in your branch and push up the change. GitHub will show your new commits and any additional feedback you may receive in the unified Pull Request view.

ProTip

Pull Request comments are written in Markdown, so you can embed images and emoji, use preformatted text blocks, and other lightweight formatting.

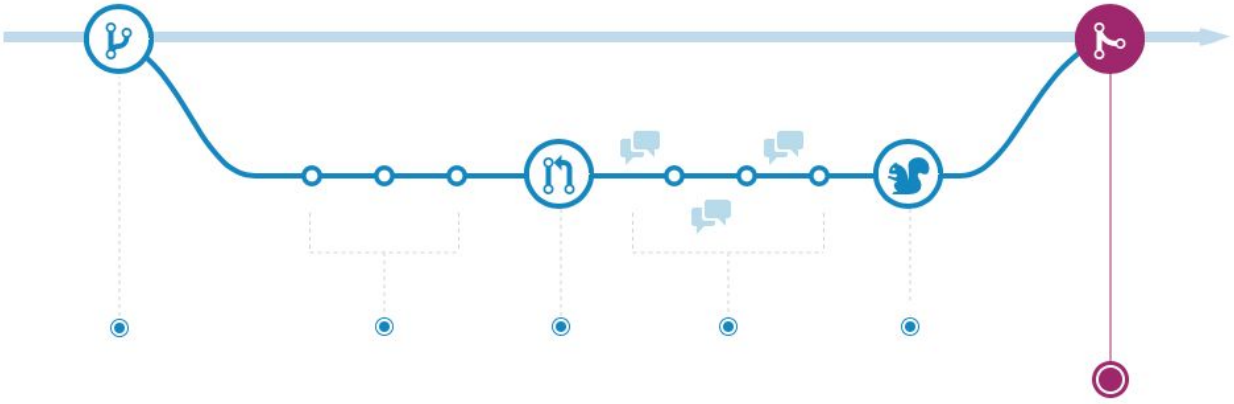
Deploy



With GitHub, you can deploy from a branch for final testing in production before merging to master.

Once your pull request has been reviewed and the branch passes your tests, you can deploy your changes to verify them in production. If your branch causes issues, you can roll it back by deploying the existing master into production.

Merge



Now that your changes have been verified in production, it is time to merge your code into the master 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.

ProTip

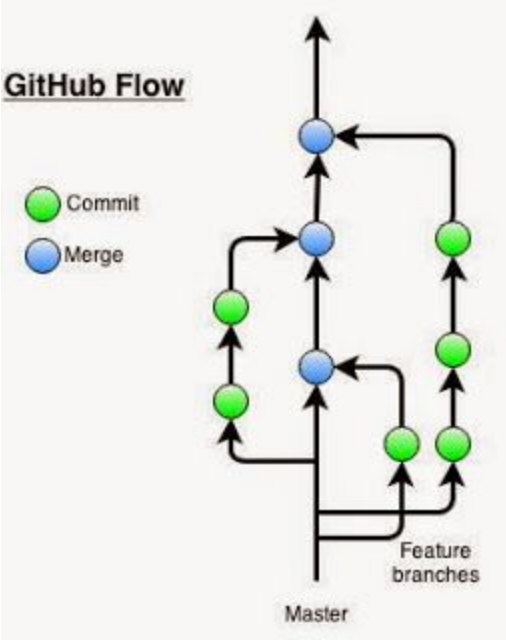
By incorporating certain keywords into the text of your Pull Request, you can associate issues with code. When your Pull Request is merged, the related issues are also closed. For example, entering the phrase `Closes #32` would close issue number 32 in the repository. For more information, check out our help article.

---

**Description 2 (https://www.endpoint.com/blog/2014/05/02/git-workflows-that-work):**

GitHub's own workflow, their internal workflow, is quite different from what everyone else does who uses GitHub. It is based on a set of simple business choices:

- Anything in the master branch is deployable
- To work on something new, create a descriptively named branch off of master (ie: new-oauth2-scopes)
- Commit to that branch locally and regularly push your work to the same named branch on the server
- When you need feedback or help, or you think the branch is ready for merging, open a pull request
- After someone else has reviewed and signed off on the feature, you can merge it into master
- Once it is merged and pushed to 'master' on the origin, you can and should deploy immediately




They release many times per day to production using this model. They branch off master for every change they make, hot fixes and features are treated the same. Then they merge back into master and release. They even have automated their releases using an irc bot.

---

**Description 3 (https://medium.com/@patrickporto/4-branching-workflows-for-git-30d0aaee7bf):**

The GitHub Flow is a lightweight workflow. It was created by GitHub in 2011 and respects the following 6 principles:

1. Anything in the master branch is deployable
2. To work on something new, create a branch off from master and given a descriptively name(ie: new-oauth2-scopes)
3. Commit to that branch locally and regularly push your work to the same named branch on the server
4. When you need feedback or help, or you think the branch is ready for merging, open a pull request
5. After someone else has reviewed and signed off on the feature, you can merge it into master
6. Once it is merged and pushed to master, you can and should deploy immediately

Advantages

- It is friendly for the Continuous Delivery and Continuous Integration
- A simpler alternative to Git Flow
- It is ideal when it needs to maintain single version in production

Disadvantages

- The production code can become unstable most easily
- Are not adequate when it needs the release plans
- It doesn’t resolve anything about deploy, environments, releases, and issues
- It isn’t recommended when multiple versions in production are needed

---

## Feature Branch

---

**Description 1 (https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow):**

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

Encapsulating feature development also makes it possible to leverage pull requests, which are a way to initiate discussions around a branch. They give other developers the opportunity to sign off on a feature before it gets integrated 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. The point is, pull requests make it incredibly easy for your team to comment on each other's work.

The Git Feature Branch Workflow is a composable workflow that can be leveraged by other high-level Git workflows. We discussed other Git workflows on the Git workflow overview page. Git Feature Branch Workflow is branching model focused, meaning that it is a guiding framework for managing and creating branches. Other workflows are more repo focused. The Git Feature Branch Workflow can be incorporated into other workflows. The Gitflow, and Git Forking Workflows traditionally use a Git Feature Branch Workflow in regards to their branching models.

How it works

The Feature Branch Workflow assumes a central repository, and master represents the official project history. Instead of committing directly on their local master branch, developers create a new branch every time they start work on a new feature. Feature branches should have descriptive names, like animated-menu-items or issue-#1061. The idea is to give a clear, highly-focused purpose to each branch. Git makes no technical distinction between the master branch and feature branches, so developers can edit, stage, and commit changes to a feature branch.

In addition, feature branches can (and should) be pushed to the central repository. This makes it possible to share a feature with other developers without touching any official code. Since master is the only "special" branch, storing several feature branches on the central repository doesn't pose any problems. Of course, this is also a convenient way to back up everybody's local commits. The following is a walk-through of the life-cycle of a feature branch.

Start with the master branch

All feature branches are created off the latest code state of a project. This guide assumes this is maintained and updated in the master branch.

```
git checkout master
git fetch origin
git reset --hard origin/master
```

This switches the repo to the master branch, pulls the latest commits and resets the repo's local copy of master to match the latest version.

Create a new-branch

Use a separate branch for each feature or issue you work on. After creating a branch, check it out locally so that any changes you make will be on that branch.

```
git checkout -b new-feature
```

This checks out a branch called new-feature based on master, and the -b flag tells Git to create the branch if it doesn’t already exist.

Update, add, commit, and push changes

On this branch, edit, stage, and commit changes in the usual fashion, building up the feature with as many commits as necessary. Work on the feature and make commits like you would any time you use Git. When ready, push your commits, updating the feature branch on Bitbucket.

```
git status
git add <some-file>
git commit
```

Push feature branch to remote

It’s a good idea to push the feature branch up to the central repository. This serves as a convenient backup, when collaborating with other developers, this would give them access to view commits to the new branch.

```
git push -u origin new-feature
```

This command pushes new-feature to the central repository (origin), and the -u flag adds it as a remote tracking branch. After setting up the tracking branch, git push can be invoked without any parameters to automatically push the new-feature branch to the central repository. To get feedback on the new feature branch, create a pull request in a repository management solution like Bitbucket Cloud or Bitbucket Server. From there, you can add reviewers and make sure everything is good to go before merging.

Resolve feedback

Now teammates comment and approve the pushed commits. Resolve their comments locally, commit, and push the suggested changes to Bitbucket. Your updates appear in the pull request.

Merge your pull request

Before you merge, you may have to resolve merge conflicts if others have made changes to the repo. When your pull request is approved and conflict-free, you can add your code to the master branch. Merge from the pull request in Bitbucket.

Pull requests

Aside from isolating feature development, branches make it possible to discuss changes via pull requests. Once someone completes a feature, they don’t immediately merge it into master. Instead, they push the feature branch to the central server and file a pull request asking to merge their additions into master. This gives other developers an opportunity to review the changes before they become a part of the main codebase.

Code review is a major benefit of pull requests, but they’re actually designed to be a generic way to talk about code. You can think of pull requests as a discussion dedicated to a particular branch. This means that they can also be used much earlier in the development process. For example, if a developer needs help with a particular feature, all they have to do is file a pull request. Interested parties will be notified automatically, and they’ll be able to see the question right next to the relevant commits.

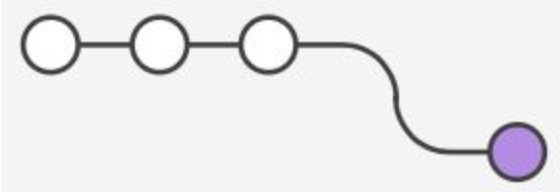
Once a pull request is accepted, the actual act of publishing a feature is much the same as in the Centralized Workflow. First, you need to make sure your local master is synchronized with the upstream master. Then, you merge the feature branch into master and push the updated master back to the central repository.

Pull requests can be facilitated by product repository management solutions like Bitbucket Cloud or Bitbucket Server. View the Bitbucket Server pull requests documentation for an example.

Example

The following is an example of the type of scenario in which a feature branching workflow is used. The scenario is that of a team doing code review around on a new feature pull request. This is one example of the many purposes this model can be used for.

Mary begins a new feature



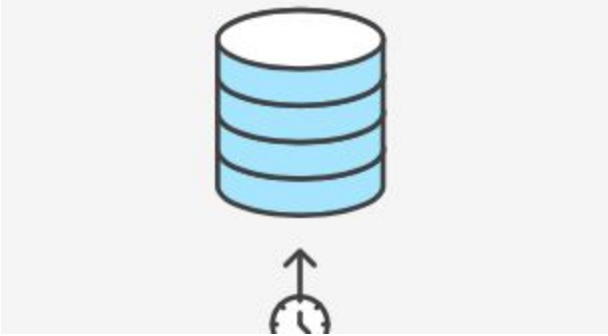
Before she starts developing a feature, Mary needs an isolated branch to work on. She can request a new branch with the following command:

```
git checkout -b marys-feature master
```

This checks out a branch called marys-feature based on master, and the -b flag tells Git to create the branch if it doesn't already exist. On this branch, Mary edits, stages, and commits changes in the usual fashion, building up her feature with as many commits as necessary:

```
git status
git add <some-file>
git commit
```

Mary goes to lunch



Mary adds a few commits to her feature over the course of the morning. Before she leaves for lunch, it's a good idea to push her feature

branch up to the central repository. This serves as a convenient backup, but if Mary was collaborating with other developers, this would also give them access to her initial commits.

```
git push -u origin marys-feature
```

This command pushes marys-feature to the central repository (origin), and the -u flag adds it as a remote tracking branch. After setting up the tracking branch, Mary can call git push without any parameters to push her feature.

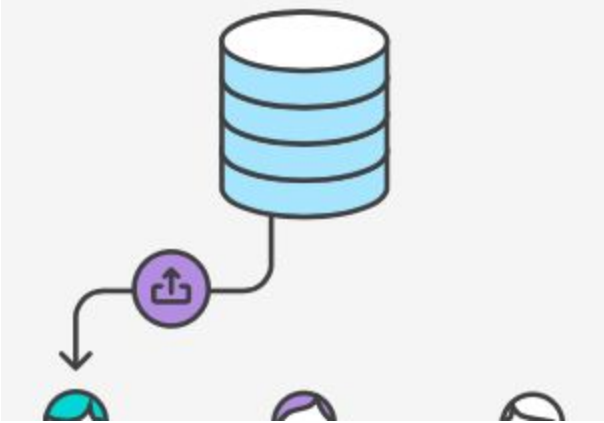
Mary finishes her feature

When Mary gets back from lunch, she completes her feature. Before merging it into master, she needs to file a pull request letting the rest of the team know she's done. But first, she should make sure the central repository has her most recent commits:

```
git push
```

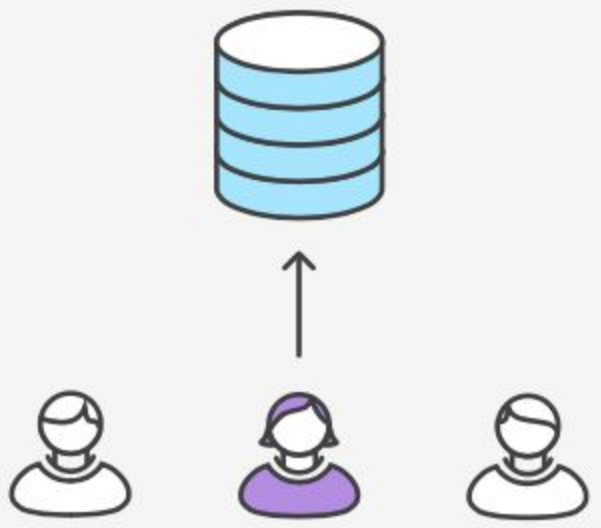
Then, she files the pull request in her Git GUI asking to merge marys-feature into master, and team members will be notified automatically. The great thing about pull requests is that they show comments right next to their related commits, so it's easy to ask questions about specific changesets.

Bill receives the pull request



Bill gets the pull request and takes a look at marys-feature. He decides he wants to make a few changes before integrating it into the official project, and he and Mary have some back-and-forth via the pull request.

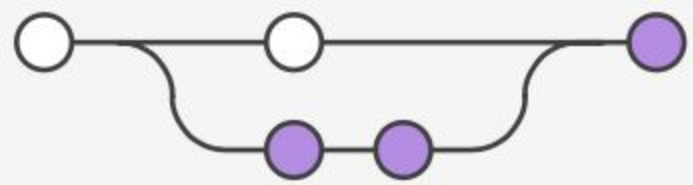
Mary makes the changes



To make the changes, Mary uses the exact same process as she did to create the first iteration of her feature. She edits, stages, commits, and pushes updates to the central repository. All her activity shows up in the pull request, and Bill can still make comments along the way.

If he wanted, Bill could pull `marys-feature` into his local repository and work on it on his own. Any commits he added would also show up in the pull request.

Mary publishes her feature



Once Bill is ready to accept the pull request, someone needs to merge the feature into the stable project (this can be done by either Bill or Mary):

```
git checkout master
git pull
git pull origin marys-feature
git push
```

This process often results in a merge commit. Some developers like this because it's like a symbolic joining of the feature with the rest of the code base. But, if you're partial to a linear history, it's possible to rebase the feature onto the tip of master before executing the merge, resulting in a fast-forward merge.

Some GUI's will automate the pull request acceptance process by running all of these commands just by clicking an "Accept" button. If yours doesn't, it should at least be able to automatically close the pull request when the feature branch gets merged into master.

Meanwhile, John is doing the exact same thing

While Mary and Bill are working on `marys-feature` and discussing it in her pull request, John is doing the exact same thing with his own feature branch. By isolating features into separate branches, everybody can work independently, yet it's still trivial to share changes with other developers when necessary.

Summary

In this document, we discussed the Git Feature Branch Workflow. This workflow helps organize and track branches that are focused on business domain feature sets. Other Git workflows like the Git Forking Workflow and the Gitflow Workflow are repo focused and can leverage the Git Feature Branch Workflow to manage their branching models. This document demonstrated a high-level code example and fictional example for implementing the Git Feature Branch Workflow. Some key associations to make with the Feature Branch Workflow are:

- focused on branching patterns
- can be leveraged by other repo oriented workflows
- promotes collaboration with team members through pull requests and merge reviews

Utilizing git rebase during the review and merge stages of a feature branch will create enforce a cohesive Git history of feature merges. A feature branching model is a great tool to promote collaboration within a team environment.

---

**Description 2 (https://www.codingblocks.net/podcast/comparing-git-workflows/):**

How it works

- All development for a new feature is performed in a dedicated feature branch.
- This allows multiple developers the ability to iterate on a feature without modifying master.
    - _*Hopefully*_ this means that master never gets incomplete code, making it advantageous for continuous integration/deployed environments.
- This feature encapsulation allows teams to utilize pull requests.
    - This allows teams to discuss and review code before it's merged into master.
- Some workflows build on top of this workflow, such as GitFlow and Forking Workflow.

Example

- Start from master (i.e. git checkout master).
- Create a new branch off of master (i.e. git checkout -b MyNewFeature).
- Similar to before, make, stage, and commit changes (i.e. git add, git commit).

- Push your new branch to the remote (or centralized) repo (i.e. git push origin MyNewFeature).
- Use the tooling of your repo management system to create a PR (i.e. GitHub, GitLab, VSTS, Bitbucket, etc.)
    - Note that if there are conflicts, those will need to be corrected locally (i.e. git pull –rebase again) and the feature branch re-pushed.
- Prior to the pull request, if another developer wanted to contribute to the feature branch, they could pull it locally to contribute to it (i.e. git pull MyNewFeature) assuming the feature branch has already been pushed to the centralized repo.

Pros

- Promotes code review and team collaboration.
- Keeps master stable.

Cons

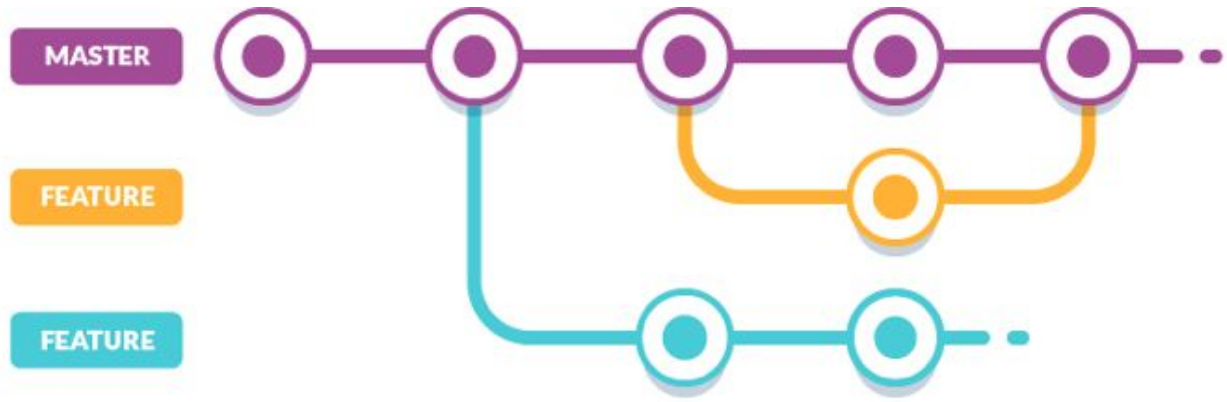
- Long lived branches have a higher risk of merge conflicts, best to keep your features small.

When should the seasoned Git guru use it?

- Always? /8)
    - This workflow is likely the building block to another workflow you need.
- Definitely best used for large teams and/or projects.

---

**Description 3 (https://buddy.works/blog/5-types-of-git-workflows):**




The basic workflow is great for developing a simple website. However, once two developers start working on two separate functionalities within one project, problems begin to appear.

Let’s say one of the developers has finished their functionality and wants to release it. However, they cannot do that because the second feature isn’t ready. Making a release at this moment could result in messing everything up, to say the least.

This is where branches - the core feature of Git - come in handy. Branches are independent “tracks” of developing a project. For each new functionality, a new branch should be created, where the new feature is developed and tested. Once the feature is ready, the branch can be merged to the master branch so it can be released to LIVE.

Feature Branch and Merge requests

The feature branch workflow assumes that all developers on the team have equal knowledge and position. In bigger teams, however, there’s always some form of hierarchy (Junior vs. Senior.)

In this case, you can employ merge requests and push permissions.

Before a branch is merged to master, it needs to be verified and checked for errors. Junior developers can create a merge request and assign it to one of the Seniors, so they can review the code and leave comments. If everything’s okay, the request is accepted and the branch is merged.

Merge requests can be combined with push permissions that allow to restrict pushing changes to specific branches in the repository, so you can keep full control over the code.

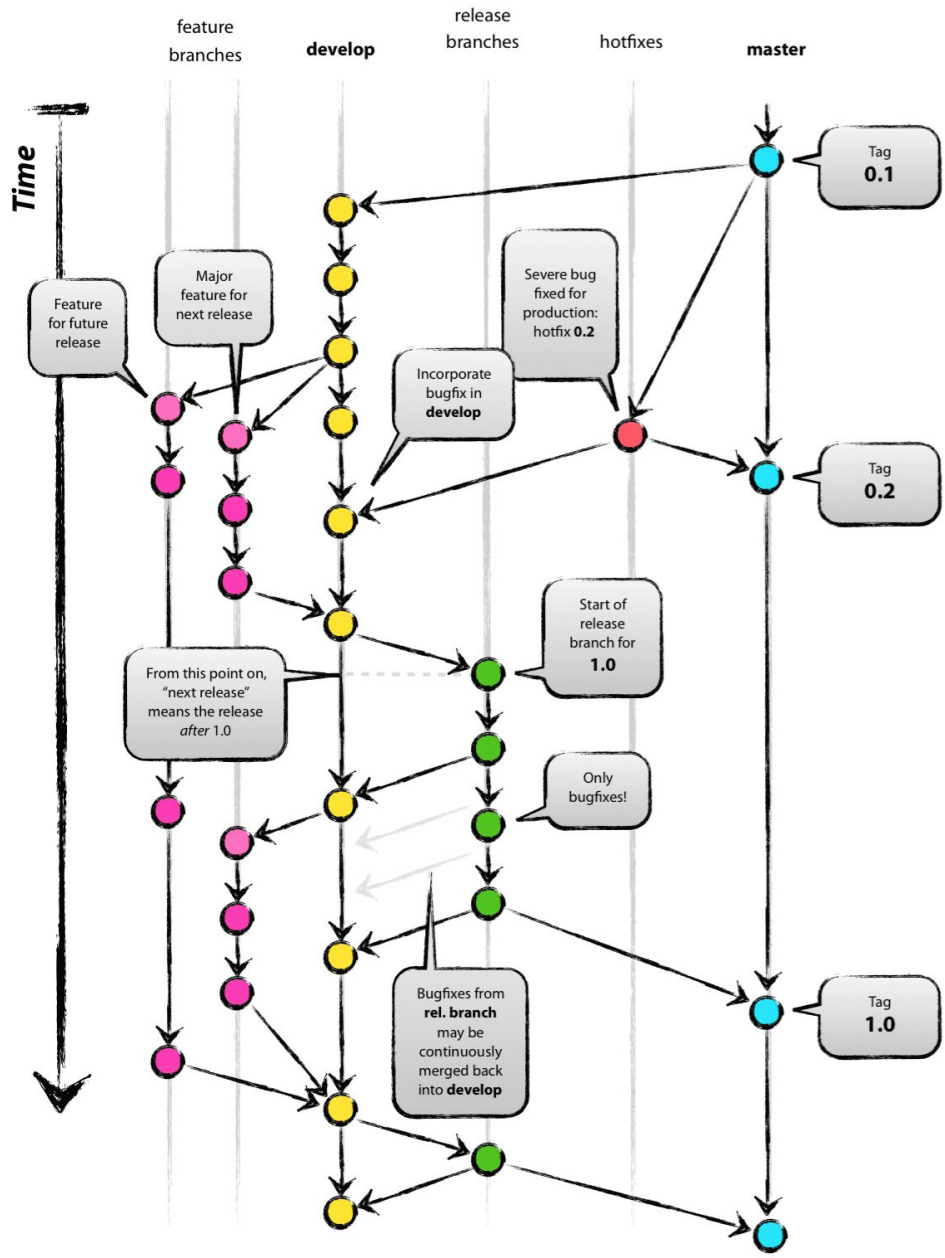
---

## GitFlow

---

**Description 1 (https://nvie.com/posts/a-successful-git-branching-model/):**

In this post I present the development model that I've introduced for some of my projects (both at work and private) about a year ago, and which has turned out to be very successful. I've been meaning to write about it for a while now, but I've never really found the time to do so thoroughly, until now. I won't talk about any of the projects' details, merely about the branching strategy and release management.



Why git?

For a thorough discussion on the pros and cons of Git compared to centralized source code control systems, see the web. There are plenty of flame wars going on there. As a developer, I prefer Git above all other tools around today. Git really changed the way developers think of merging and branching. From the classic CVS/Subversion world I came from, merging/branching has always been considered a bit scary ("beware of merge conflicts, they bite you!") and something you only do every once in a while.

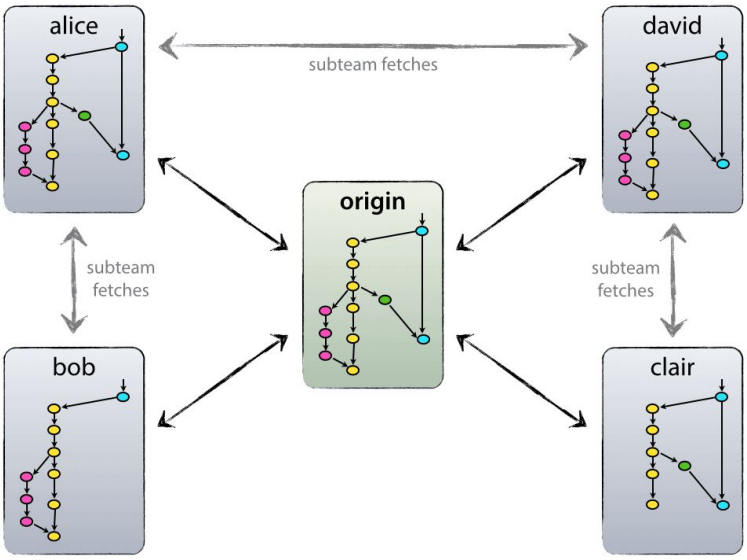
But with Git, these actions are extremely cheap and simple, and they are considered one of the core parts of your daily workflow, really. For example, in CVS/Subversion books, branching and merging is first discussed in the later chapters (for advanced users), while in every Git book, it's already covered in chapter 3 (basics).

As a consequence of its simplicity and repetitive nature, branching and merging are no longer something to be afraid of. Version control tools are supposed to assist in branching/merging more than anything else.

Enough about the tools, let's head onto the development model. The model that I'm going to present here is essentially no more than a set of procedures that every team member has to follow in order to come to a managed software development process.

Decentralized but centralized

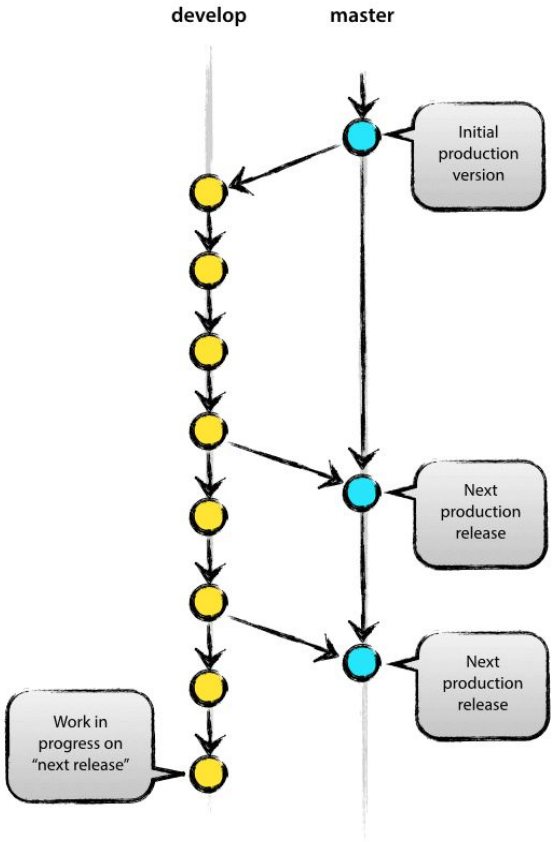
The repository setup that we use and that works well with this branching model, is that with a central "truth" repo. Note that this repo is only considered to be the central one (since Git is a DVCS, there is no such thing as a central repo at a technical level). We will refer to this repo as origin, since this name is familiar to all Git users.



Each developer pulls and pushes to origin. But besides the centralized push-pull relationships, each developer may also pull changes from other peers to form sub teams. For example, this might be useful to work together with two or more developers on a big new feature, before pushing the work in progress to origin prematurely. In the figure above, there are subteams of Alice and Bob, Alice and David, and Clair and David.

Technically, this means nothing more than that Alice has defined a Git remote, named bob, pointing to Bob's repository, and vice versa.

The main branches




At the core, the development model is greatly inspired by existing models out there. The central repo holds two main branches with an infinite lifetime:

- master
- develop

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

We consider `origin/master` to be the main branch where the source code of HEAD always reflects a production-ready state.

We consider `origin/develop` to be the main branch where the source code of HEAD always reflects a state with the latest delivered development changes for the next release. Some would call this the "integration branch". This is where any automatic nightly builds are built from.

When the source code in the develop branch reaches a stable point and is ready to be released, all of the changes should be merged back into master somehow and then tagged with a release number. How this is done in detail will be discussed further on.

Therefore, each time when changes are merged back into master, this is a new production release by definition. We tend to be very strict at this, so that theoretically, we could use a Git hook script to automatically build and roll-out our software to our production servers every time there was a commit on master.

Supporting branches

Next to the main branches master and develop, our development model uses a variety of supporting branches to aid parallel development between team members, ease tracking of features, prepare for production releases and to assist in quickly fixing live production problems. Unlike the main branches, these branches always have a limited life time, since they will be removed eventually.

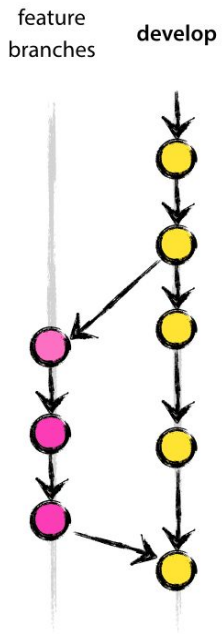
The different types of branches we may use are:

- Feature branches
- Release branches
- Hotfix branches

Each of these branches have a specific purpose and are bound to strict rules as to which branches may be their originating branch and which branches must be their merge targets. We will walk through them in a minute.

By no means are these branches “special” from a technical perspective. The branch types are categorized by how we use them. They are of course plain old Git branches.

Feature branches




May branch off from:
  develop
Must merge back into:
  develop
Branch naming convention:
  anything except master, develop, release-*, or hotfix-*

Feature branches (or sometimes called topic branches) are used to develop new features for the upcoming or a distant future release. When starting development of a feature, the target release in which this feature will be incorporated may well be unknown at that point. The essence of a feature branch is that it exists as long as the feature is in development, but will eventually be merged back into develop (to definitely add a new feature to the upcoming release) or discarded (in case of a disappointing experiment).

Feature branches typically exist in developer repos only, not in origin.

Creating a feature branch

When starting work on a new feature, branch off from the develop branch.

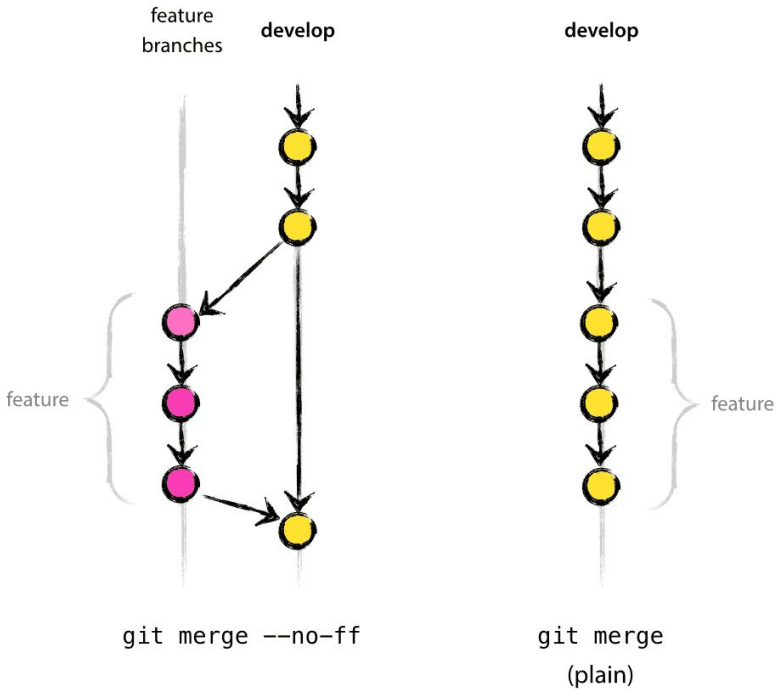
```
$ git checkout -b myfeature develop
Switched to a new branch "myfeature"
```

Incorporating a finished feature on develop

Finished features may be merged into the develop branch to definitely add them to the upcoming release:

```
$ git checkout develop
Switched to branch 'develop'
$ git merge --no-ff myfeature
Updating ea1b82a..05e9557
(Summary of changes)
$ git branch -d myfeature
Deleted branch myfeature (was 05e9557).
$ git push origin develop
```

The `--no-ff` flag causes the merge to always create a new commit object, even if the merge could be performed with a fast-forward. This avoids losing information about the historical existence of a feature branch and groups together all commits that together added the feature. Compare:



In the latter case, it is impossible to see from the Git history which of the commit objects together have implemented a feature—you would have to manually read all the log messages. Reverting a whole feature (i.e. a group of commits), is a true headache in the latter situation, whereas it is easily done if the `--no-ff` flag was used.

Yes, it will create a few more (empty) commit objects, but the gain is much bigger than the cost.

Release branches

May branch off from:
  develop
Must merge back into:
  develop and master
Branch naming convention:
  release-*

Release branches support preparation of a new production release. They allow for last-minute dotting of the i's and crossing the t's. Furthermore, they allow for minor bug fixes and preparing meta-data for a release (version number, build dates, etc.). By doing all of this work on a release branch, the develop branch is cleared to receive features for the next big release.

The key moment to branch off a new release branch from develop is when develop (almost) reflects the desired state of the new release. At least all the features that are targeted for the release-to-be-built must be merged in to develop at this point in time. All features targeted at future releases may not—they must wait until after the release branch is branched off.

It is exactly at the start of a release branch that the upcoming release gets assigned a version number—not any earlier. Up until that moment, the develop branch reflected changes for the “next release”, but it is unclear whether that “next release” will eventually become 0.3 or 1.0, until the release branch is started. That decision is made on the start of the release branch and is carried out by the project’s rules on version number bumping.

Creating a release branch

Release branches are created from the develop branch. For example, say version 1.1.5 is the current production release and we have a big release coming up. The state of develop is ready for the “next release” and we have decided that this will become version 1.2 (rather than 1.1.6 or 2.0). So we branch off and give the release branch a name reflecting the new version number:

```
$ git checkout -b release-1.2 develop
Switched to a new branch "release-1.2"
$ ./bump-version.sh 1.2
```

```
Files modified successfully, version bumped to 1.2.
$ git commit -a -m "Bumped version number to 1.2"
[release-1.2 74d9424] Bumped version number to 1.2
1 files changed, 1 insertions(+), 1 deletions(-)
```

After creating a new branch and switching to it, we bump the version number. Here, bump-version.sh is a fictional shell script that changes some files in the working copy to reflect the new version. (This can of course be a manual change—the point being that some files change.) Then, the bumped version number is committed.

This new branch may exist there for a while, until the release may be rolled out definitely. During that time, bug fixes may be applied in this branch (rather than on the develop branch). Adding large new features here is strictly prohibited. They must be merged into develop, and therefore, wait for the next big release.

Finishing a release branch

When the state of the release branch is ready to become a real release, some actions need to be carried out. First, the release branch is merged into master (since every commit on master is a new release by definition, remember). Next, that commit on master must be tagged for easy future reference to this historical version. Finally, the changes made on the release branch need to be merged back into develop, so that future releases also contain these bug fixes.

The first two steps in Git:

```
$ git checkout master
Switched to branch 'master'
$ git merge --no-ff release-1.2
Merge made by recursive.
(Summary of changes)
$ git tag -a 1.2
```

The release is now done, and tagged for future reference.

Edit: You might as well want to use the -s or -u <key> flags to sign your tag cryptographically.

To keep the changes made in the release branch, we need to merge those back into develop, though. In Git:

```
$ git checkout develop
Switched to branch 'develop'
$ git merge --no-ff release-1.2
Merge made by recursive.
```

```
(Summary of changes)
```

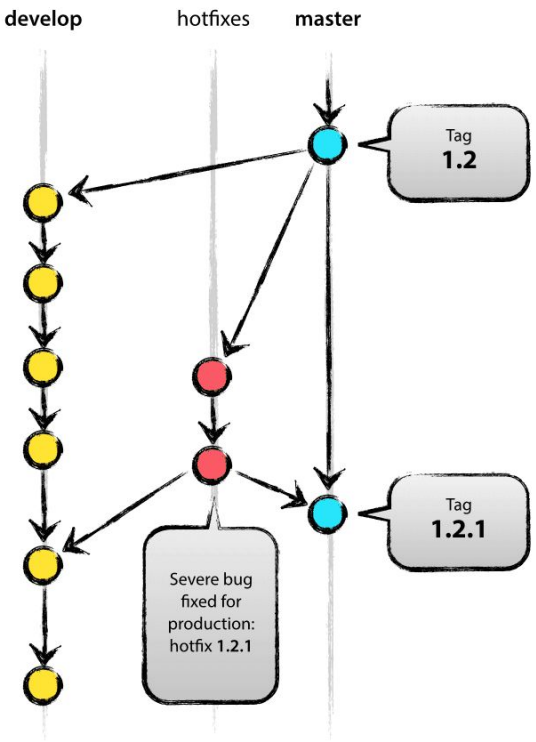
This step may well lead to a merge conflict (probably even, since we have changed the version number). If so, fix it and commit.

Now we are really done and the release branch may be removed, since we don't need it anymore:

```
$ git branch -d release-1.2
Deleted branch release-1.2 (was ff452fe).
```

Hotfix branches

May branch off from:
  master
Must merge back into:
  develop and master
Branch naming convention:
  hotfix-*




Hotfix branches are very much like release branches in that they are also meant to prepare for a new production release, albeit unplanned. They arise from the necessity to act immediately upon an undesired state of a live production version. When a critical bug in a production version must be resolved immediately, a hotfix branch may be branched off from the corresponding tag on the master branch that marks the production version.

The essence is that work of team members (on the develop branch) can continue, while another person is preparing a quick production fix.

Creating the hotfix branch

Hotfix branches are created from the master branch. For example, say version 1.2 is the current production release running live and causing troubles due to a severe bug. But changes on develop are yet unstable. We may then branch off a hotfix branch and start fixing the problem:

```
$ git checkout -b hotfix-1.2.1 master
Switched to a new branch "hotfix-1.2.1"
$ ./bump-version.sh 1.2.1
Files modified successfully, version bumped to 1.2.1.
```

```
$ git commit -a -m "Bumped version number to 1.2.1"
[hotfix-1.2.1 41e61bb] Bumped version number to 1.2.1
1 files changed, 1 insertions(+), 1 deletions(-)
```

Don’t forget to bump the version number after branching off!

Then, fix the bug and commit the fix in one or more separate commits.

```
$ git commit -m "Fixed severe production problem"
[hotfix-1.2.1 abbe5d6] Fixed severe production problem
5 files changed, 32 insertions(+), 17 deletions(-)
```

Finishing a hotfix branch

When finished, the bugfix needs to be merged back into master, but also needs to be merged back into develop, in order to safeguard that the bugfix is included in the next release as well. This is completely similar to how release branches are finished.

First, update master and tag the release.

```
$ git checkout master
Switched to branch 'master'
$ git merge --no-ff hotfix-1.2.1
Merge made by recursive.
(Summary of changes)
$ git tag -a 1.2.1
```

Edit: You might as well want to use the -s or -u <key> flags to sign your tag cryptographically.

Next, include the bugfix in develop, too:

```
$ git checkout develop
Switched to branch 'develop'
$ git merge --no-ff hotfix-1.2.1
Merge made by recursive.
(Summary of changes)
```

The one exception to the rule here is that, when a release branch currently exists, the hotfix changes need to be merged into that release branch, instead of develop. Back-merging the bugfix into the release branch will eventually result in the bugfix being merged into develop too, when the release branch is finished. (If work in develop immediately requires this bugfix and cannot wait for the release branch to be finished, you may safely merge the bugfix into develop now already as well.)

Finally, remove the temporary branch:

```
$ git branch -d hotfix-1.2.1
Deleted branch hotfix-1.2.1 (was abbe5d6).
```

Summary

While there is nothing really shocking new to this branching model, the “big picture” figure that this post began with has turned out to be tremendously useful in our projects. It forms an elegant mental model that is easy to comprehend and allows team members to develop a shared understanding of the branching and releasing processes.

A high-quality PDF version of the figure is provided here. Go ahead and hang it on the wall for quick reference at any time.

Update: And for anyone who requested it: here’s the gitflow-model.src.key of the main diagram image (Apple Keynote).

---

**Description 2 (https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow):**

Gitflow Workflow is a Git workflow design that was first published and made popular by Vincent Driessen at nvie. The Gitflow Workflow defines a strict branching model designed around the project release. This provides a robust framework for managing larger projects.

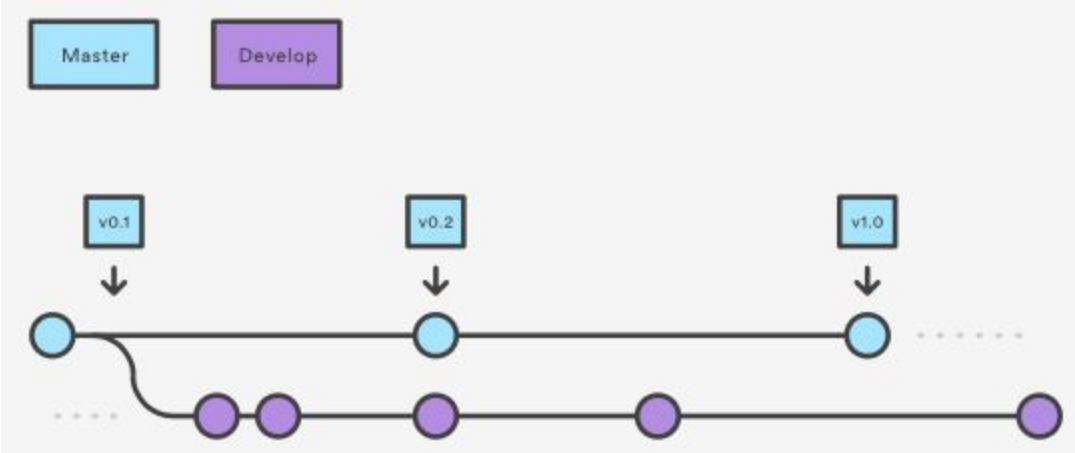
Gitflow is ideally suited for projects that have a scheduled release cycle. This workflow doesn’t add any new concepts or commands beyond what’s required for the Feature Branch Workflow. Instead, it assigns very specific roles to different branches and defines how and when they should interact. In addition to feature branches, it uses individual branches for preparing, maintaining, and recording releases. Of course, you also get to leverage all the benefits of the Feature Branch Workflow: pull requests, isolated experiments, and more efficient collaboration.

Getting Started

Gitflow is really just an abstract idea of a Git workflow. This means it dictates what kind of branches to set up and how to merge them together. We will touch on the purposes of the branches below. The git-flow toolset is an actual command line tool that has an installation process. The installation process for git-flow is straightforward. Packages for git-flow are available on multiple operating systems. On OSX systems, you can execute `brew install git-flow`. On windows you will need to download and install git-flow. After installing git-flow you can use it in your project by executing `git flow init`. Git-flow is a wrapper around Git.

The `git flow init` command is an extension of the default git init command and doesn't change anything in your repository other than creating branches for you.

How it works




Develop and Master Branches

Instead of a single master branch, this workflow uses two branches to record the history of the project. The master branch stores the official release history, and the develop branch serves as an integration branch for features. It's also convenient to tag all commits in the master branch with a version number.

The first step is to complement the default master with a develop branch. A simple way to do this is for one developer to create an empty develop branch locally and push it to the server:

```
git branch develop
git push -u origin develop
```

This branch will contain the complete history of the project, whereas master will contain an abridged version. Other developers should now clone the central repository and create a tracking branch for develop.

When using the git-flow extension library, executing `git flow init` on an existing repo will create the develop branch:

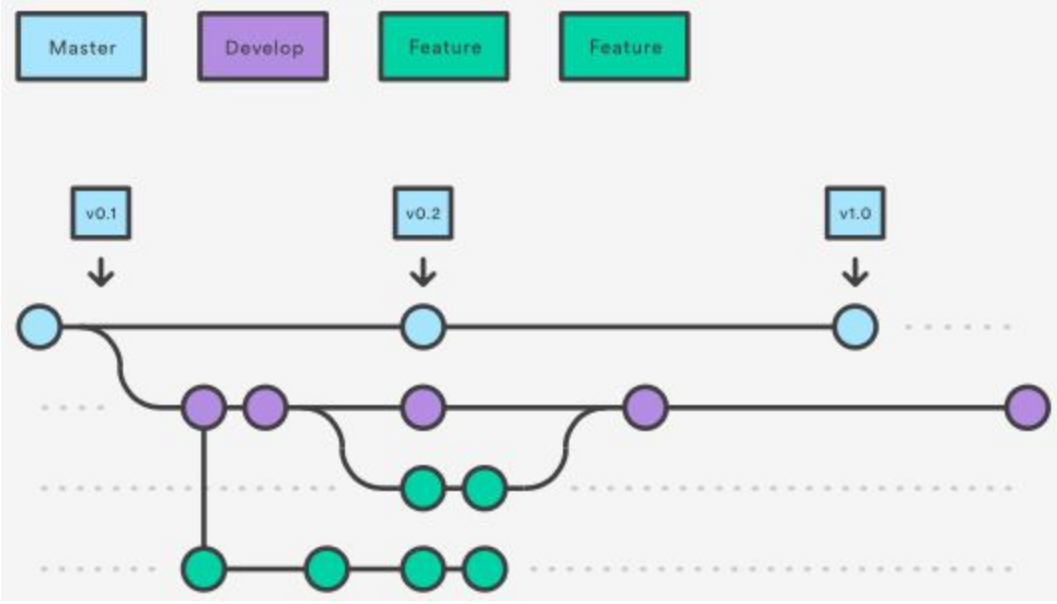
```
$ git flow init
Initialized empty Git repository in ~/project/.git/
No branches exist yet. Base branches must be created now.
Branch name for production releases: [master]
Branch name for "next release" development: [develop]

How to name your supporting branch prefixes?
Feature branches? [feature/]
Release branches? [release/]
Hotfix branches? [hotfix/]
Support branches? [support/]
Version tag prefix? []

$ git branch
* develop
  master
```

Feature Branches

Each new feature should reside in its own branch, which can be pushed to the central repository for backup/collaboration. But, instead of branching off of master, feature branches use develop as their parent branch. When a feature is complete, it gets merged back into develop. Features should never interact directly with master.




Note that feature branches combined with the develop branch is, for all intents and purposes, the Feature Branch Workflow. But, the Gitflow Workflow doesn't stop there.

Feature branches are generally created off to the latest develop branch.

Creating a feature branch

Without the git-flow extensions:

```
git checkout develop
git checkout -b feature_branch
```

When using the git-flow extension:

```
git flow feature start feature_branch
```

Continue your work and use Git like you normally would.

Finishing a feature branch

When you're done with the development work on the feature, the next step is to merge the feature_branch into develop.

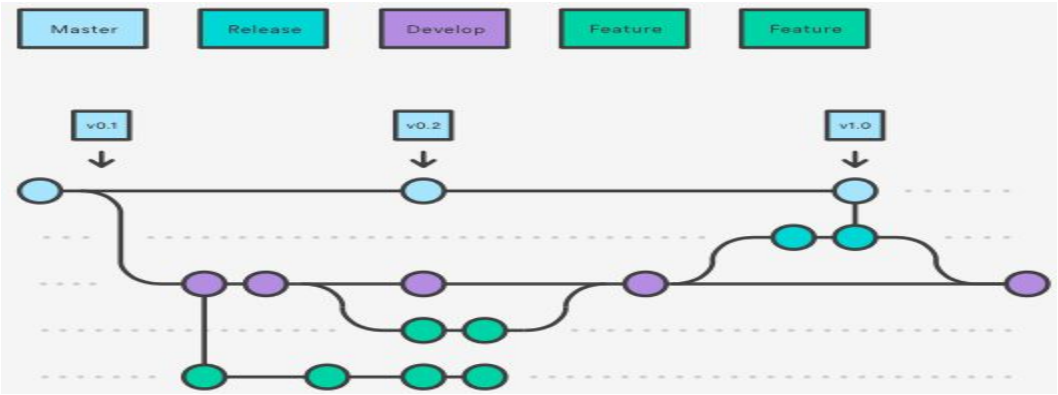
Without the git-flow extensions:

```
git checkout develop
git merge feature_branch
```

Using the git-flow extensions:

```
git flow feature finish feature_branch
```

Release Branches



Once develop has acquired enough features for a release (or a predetermined release date is approaching), you fork a release branch off of develop. Creating this branch starts the next release cycle, so no new features can be added after this point—only bug fixes, documentation generation, and other release-oriented tasks should go in this branch. Once it's ready to ship, the release branch gets merged into master and tagged with a version number. In addition, it should be merged back into develop, which may have progressed since the release was initiated.

Using a dedicated branch to prepare releases makes it possible for one team to polish the current release while another team continues working on features for the next release. It also creates well-defined phases of development (e.g., it's easy to say, “This week we're preparing for version 4.0,” and to actually see it in the structure of the repository).

Making release branches is another straightforward branching operation. Like feature branches, release branches are based on the develop branch. A new release branch can be created using the following methods.

Without the git-flow extensions:

```
git checkout develop
git checkout -b release/0.1.0
```

When using the git-flow extensions:

```
$ git flow release start 0.1.0
Switched to a new branch 'release/0.1.0'
```

Once the release is ready to ship, it will get merged it into master and develop, then the release branch will be deleted. It’s important to merge back into develop because critical updates may have been added to the release branch and they need to be accessible to new features. If your organization stresses code review, this would be an ideal place for a pull request.

To finish a release branch, use the following methods:

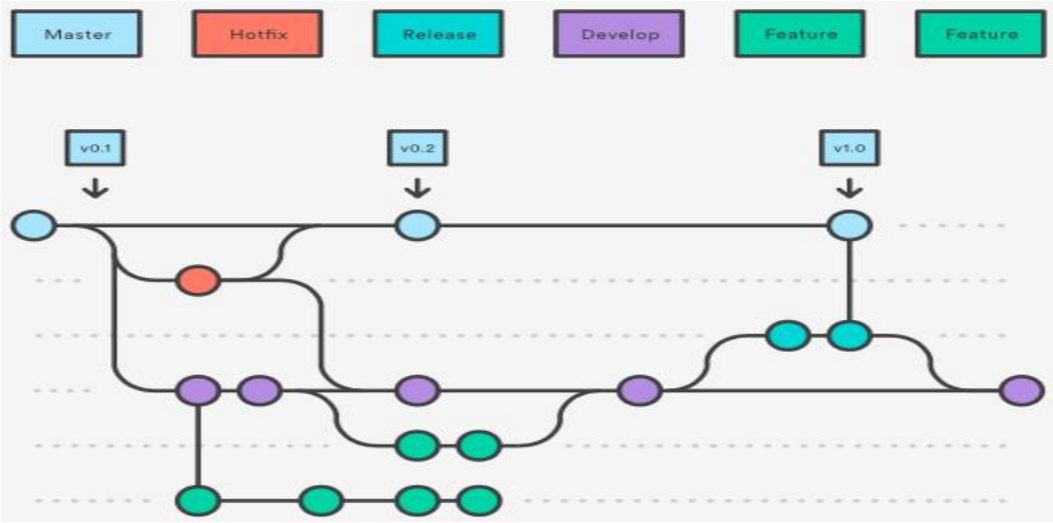
Without the git-flow extensions:

```
git checkout master
git merge release/0.1.0
```

Or with the git-flow extension:

```
git flow release finish '0.1.0'
```

Hotfix Branches




Maintenance or “hotfix” branches are used to quickly patch production releases. Hotfix branches are a lot like release branches and feature branches except they're based on master instead of develop. This is the only branch that should fork directly off of master. As soon as the fix is complete, it should be merged into both master and develop (or the current release branch), and master should be tagged with an updated version number.

Having a dedicated line of development for bug fixes lets your team address issues without interrupting the rest of the workflow or waiting for the next release cycle. You can think of maintenance branches as ad hoc release branches that work directly with master. A hotfix branch can be created using the following methods:

Without the git-flow extensions:

```
git checkout master
git checkout -b hotfix_branch
```

When using the git-flow extensions:

```
$ git flow hotfix start hotfix_branch
```

Similar to finishing a release branch, a hotfix branch gets merged into both master and develop.

```
git checkout master
git merge hotfix_branch
git checkout develop
git merge hotfix_branch
git branch -D hotfix_branch
```

```
$ git flow hotfix finish hotfix_branch
```

Example

A complete example demonstrating a Feature Branch Flow is as follows. Assuming we have a repo setup with a master branch.

```
git checkout master
git checkout -b develop
git checkout -b feature_branch
# work happens on feature branch
git checkout develop
git merge feature_branch
git checkout master
git merge develop
git branch -d feature_branch
```

In addition to the feature and release flow, a hotfix example is as follows:

```
git checkout master
git checkout -b hotfix_branch
# work is done commits are added to the hotfix_branch
git checkout develop
git merge hotfix_branch
git checkout master
git merge hotfix_branch
```

Summary

Here we discussed the Gitflow Workflow. Gitflow is one of many styles of Git workflows you and your team can utilize.

Some key takeaways to know about GitFlow are:

- The workflow is great for a release-based software workflow.
- Gitflow offers a dedicated channel for hotfixes to production.

The overall flow of Gitflow is:

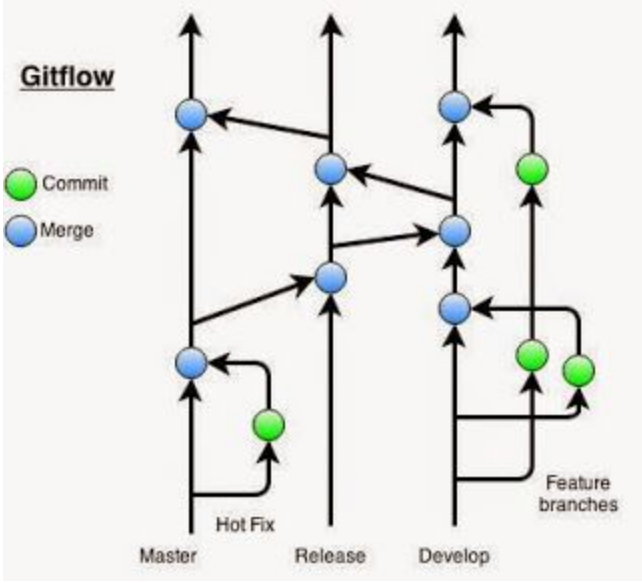
1. A develop branch is created from master
2. A release branch is created from develop
3. Feature branches are created from develop
4. When a feature is complete it is merged into the develop branch
5. When the release branch is done it is merged into develop and master
6. If an issue in master is detected a hotfix branch is created from master
7. Once the hotfix is complete it is merged to both develop and master

---

**Description 3 (https://www.endpoint.com/blog/2014/05/02/git-workflows-that-work):**

On the other end of the spectrum from a master only workflow, is Gitflow. Here there are at least three main branches: develop (or development), release, and master. There are other branches as well for features and hot fixes. Many of these are long running. For example, you merge develop into the release branch but then you continue working on develop and add more commits. The workflow looks like this:

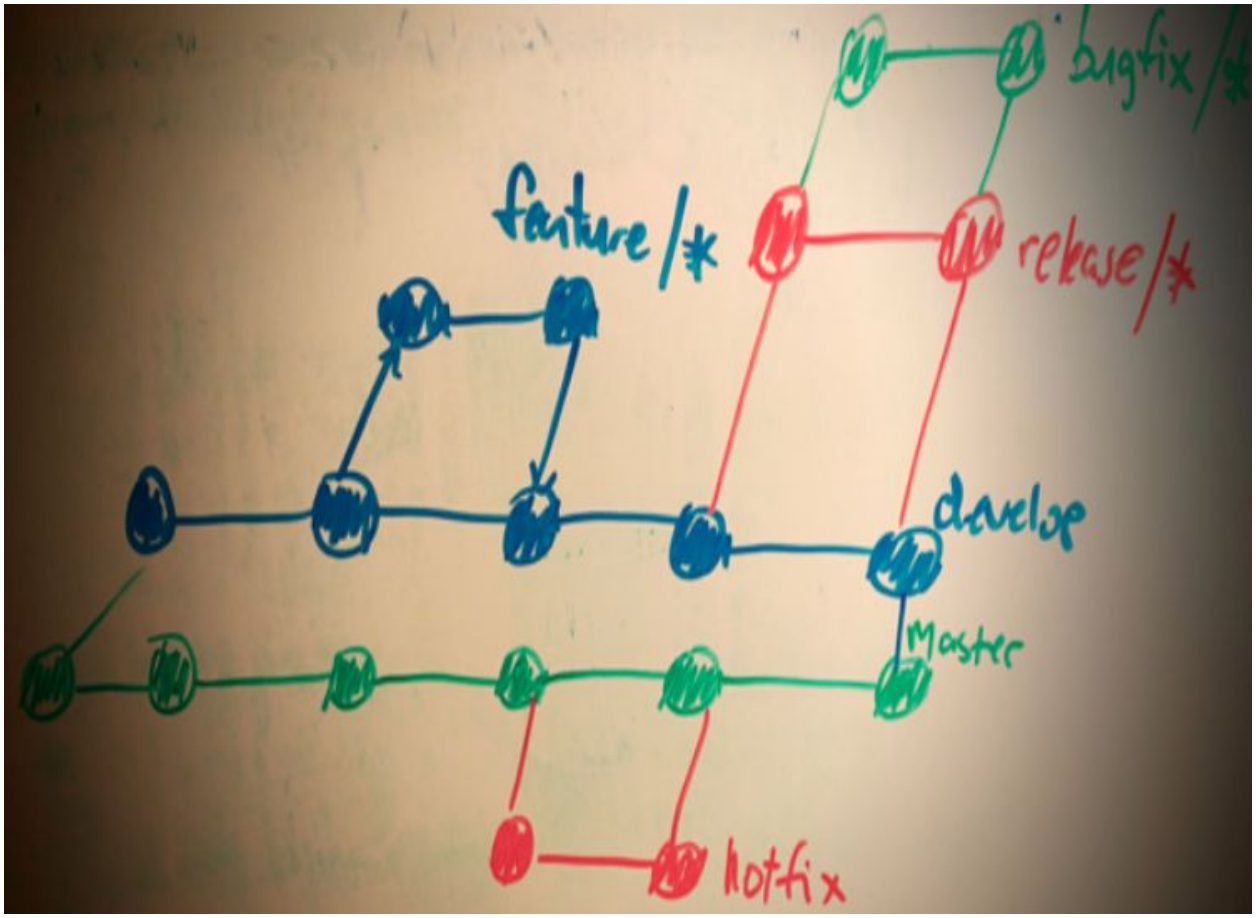
- All work is done in a branch. Features are branched off develop. Hot fixes are treated different and are branched off master.
- Features are merged back into develop after approval.
- Develop is merged into a release branch.
- Hot fixes are merged back into master, but also must be merged into develop and the release branch.
- The release branch is merged into master.



- Master is deployed to production.

---

**Description 4 (http://drincruz.github.io/slides/git-workflow-comparison/#/types-of-workflows):**




Gitflow notes

- master mirrors production
- feature branches branch off of develop
- release branches branch off of develop with features not yet in master
- hotfix branches branch off of master
- release branches merge back into master and develop
- hotfix branches also merge back into master and develop

Gitflow in a nutshell [feature]

- clone.
- create feature/{xyz} branch off of develop.
- code & commit.
- pull request to merge into develop.

Gitflow in a nutshell [release]

- create release/{version_number} branch off of develop.
- qa & bugfix
- merge into develop AND master.

Takeaways

- use what works best for _your_ development cycle
- these workflows are guidelines, not strict rules
- one does not simply merge into master

---

**Description 5 (https://datasift.github.io/gitflow/IntroducingGitFlow.html):**

What Is GitFlow?

GitFlow is a branching model for Git, created by Vincent Driessen. It has attracted a lot of attention because it is very well suited to collaboration and scaling the development team.

Key Benefits

Parallel Development
One of the great things about GitFlow is that it makes parallel development very easy, by isolating new development from finished work. New development (such as features and non-emergency bug fixes) is done in feature branches, and is only merged back into the main body of code when the developer(s) is happy that the code is ready for release.

Although interruptions are a BadThing(tm), if you are asked to switch from one task to another, all you need to do is commit your changes and then create a new feature branch for your new task. When that task is done, just checkout your original feature branch and you can continue where you left off.

Collaboration
Feature branches also make it easier for two or more developers to collaborate on the same feature, because each feature branch is a sandbox where the only changes are the changes necessary to get the new feature working. That makes it very easy to see and follow what each collaborator is doing.

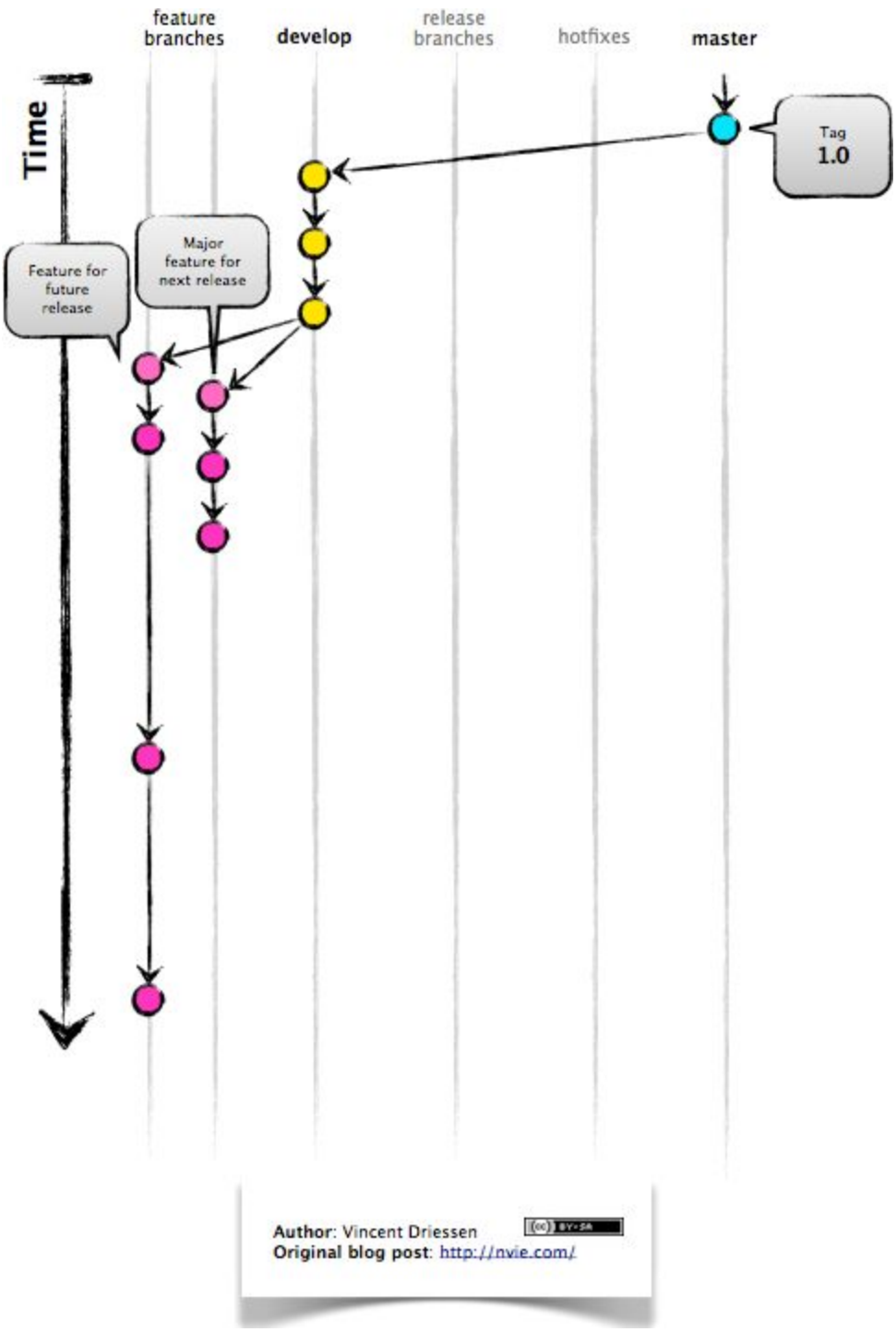
Release Staging Area
As new development is completed, it gets merged back into the develop branch, which is a staging area for all completed features that haven't yet been released. So when the next release is branched off of develop, it will automatically contain all of the new stuff that has been finished.

Support For Emergency Fixes

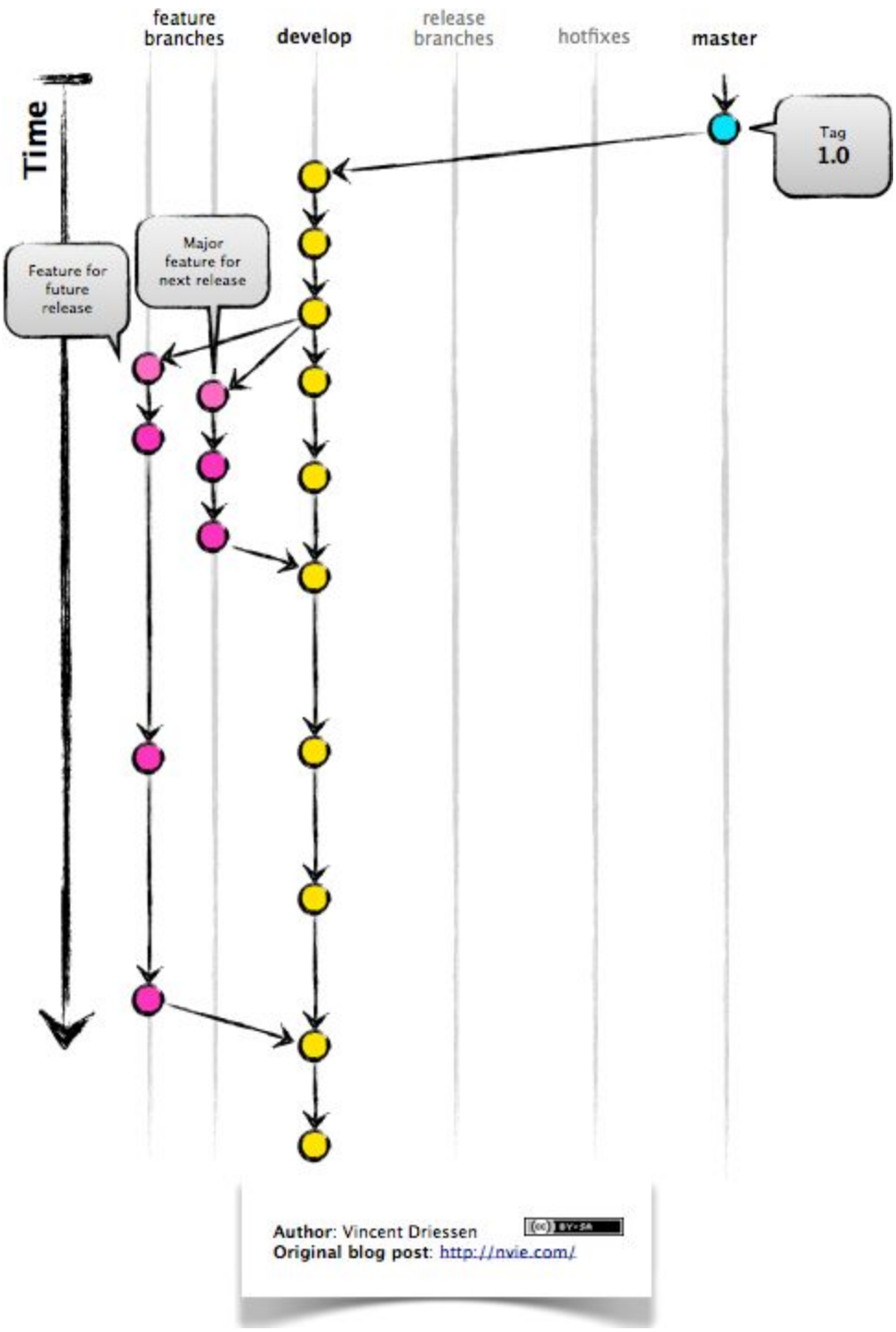
GitFlow supports hotfix branches - branches made from a tagged release. You can use these to make an emergency change, safe in the knowledge that the hotfix will only contain your emergency fix. There's no risk that you'll accidentally merge in new development at the same time.

How It Works

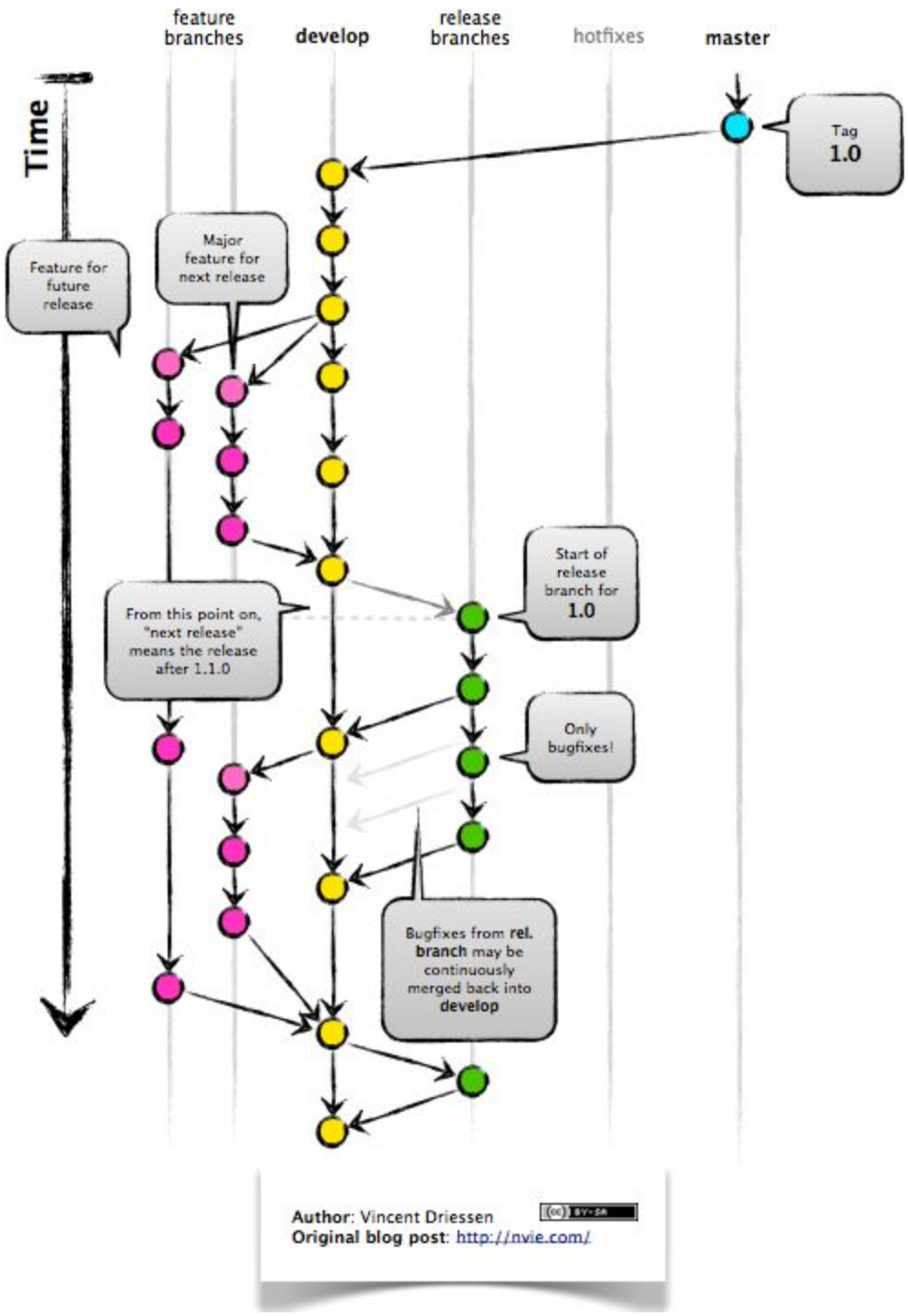
New development (new features, non-emergency bug fixes) are built in feature branches:




Feature branches are branched off of the develop branch, and finished features and fixes are merged back into the develop branch when they're ready for release:

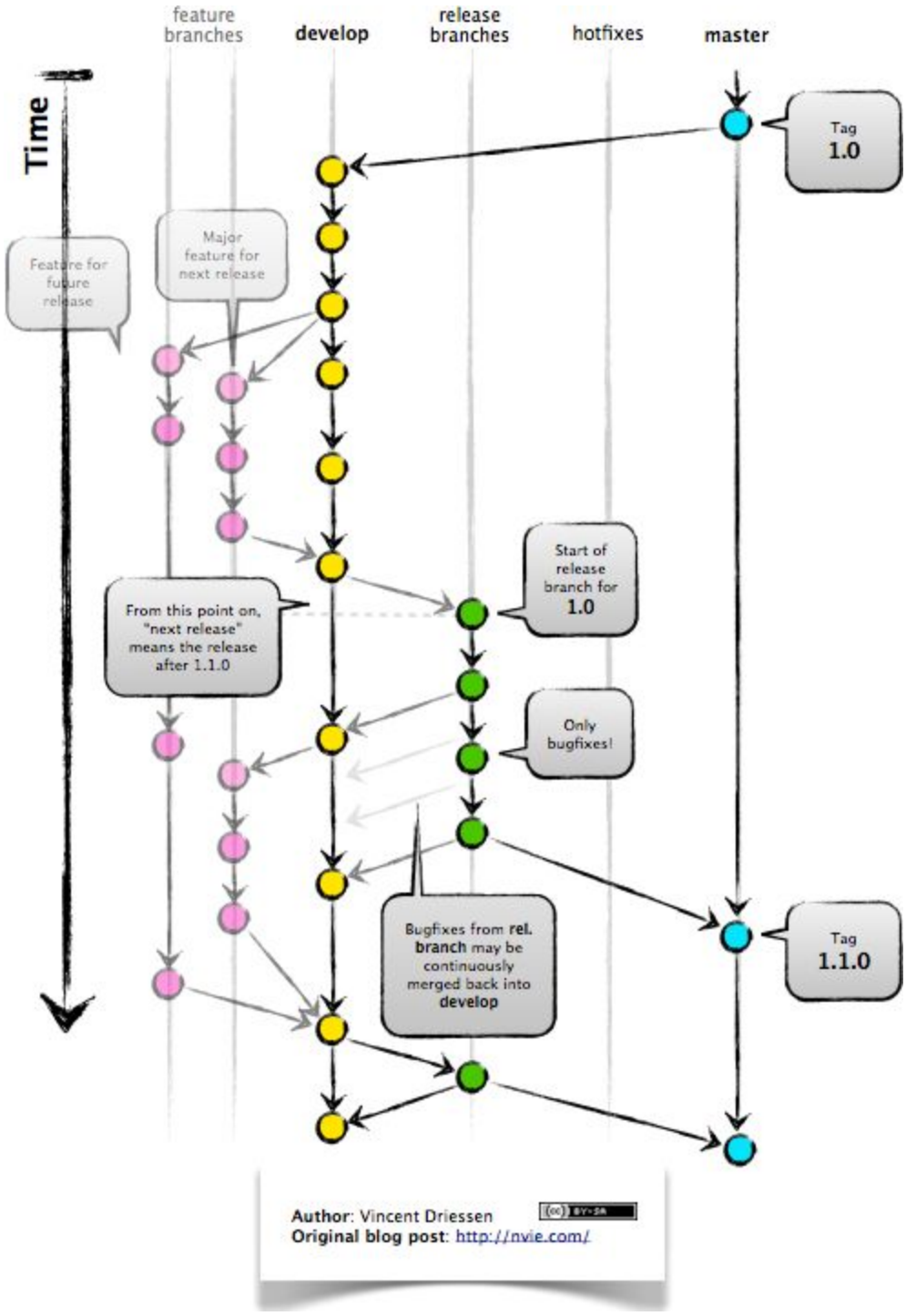



When it is time to make a release, a release branch is created off of develop:



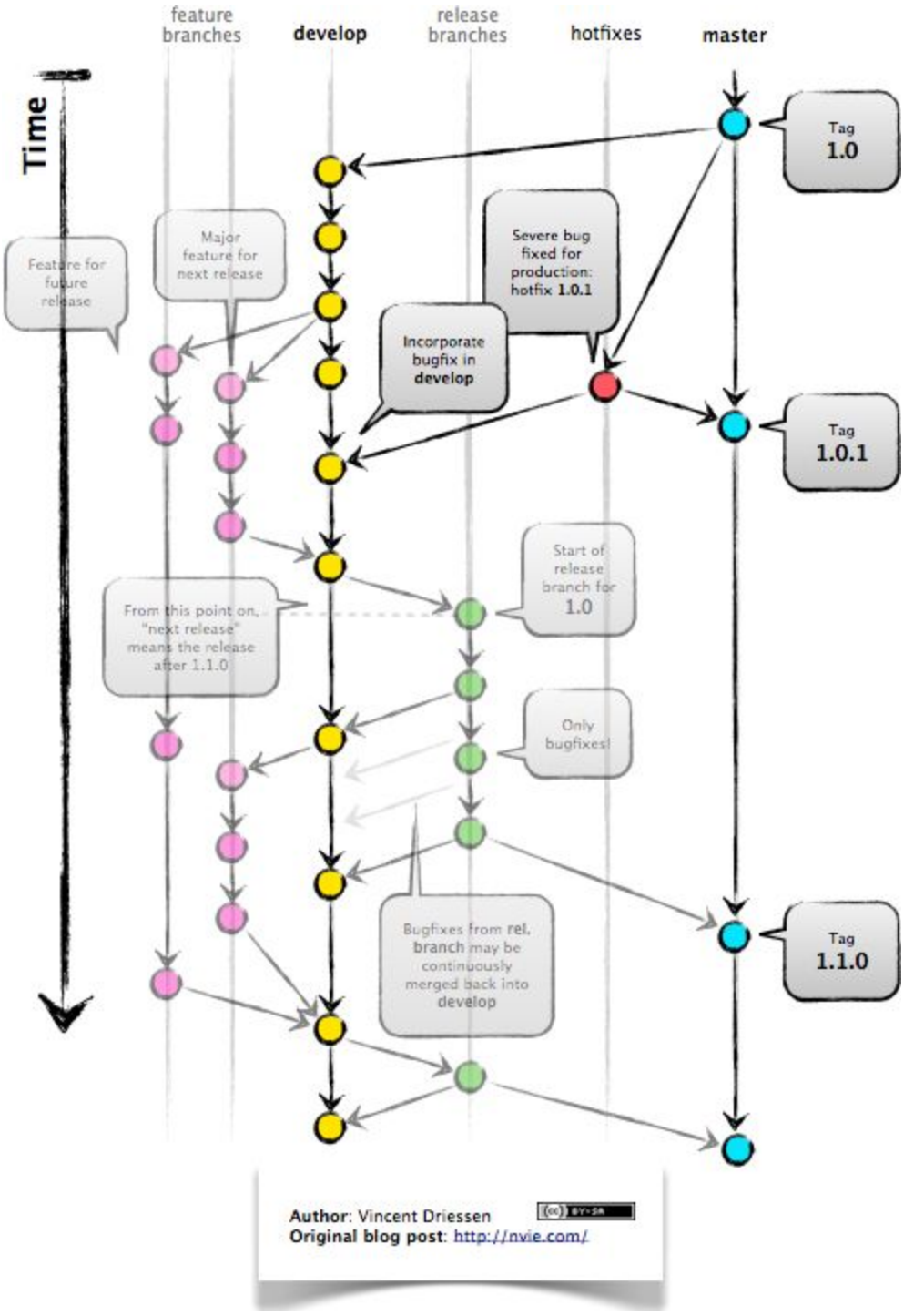

The code in the release branch is deployed onto a suitable test environment, tested, and any problems are fixed directly in the release branch. This deploy -> test -> fix -> redeploy -> retest cycle continues until you're happy that the release is good enough to release to customers.

When the release is finished, the release branch is merged into master and into develop too, to make sure that any changes made in the release branch aren't accidentally lost by new development.




The master branch tracks released code only. The only commits to master are merges from release branches and hotfix branches.

Hotfix branches are used to create emergency fixes:




They are branched directly from a tagged release in the master branch, and when finished are merged back into both master and develop to make sure that the hotfix isn't accidentally lost when the next regular release occurs.

---

**Description 6 (https://www.codingblocks.net/podcast/comparing-git-workflows/):**

How it works

- Builds upon the Feature Branch workflow.
- However instead of just a master branch, you create additional branches with very specific purposes.
    - As well as specific times when those branches should interact.
- The master branch represents the officially released versions.
    - Create version tags off of master.
- A develop branch is used to iterate on the next version.
    - All new feature branches are based on develop instead of master (unlike the Feature Branch workflow).
- When it's time to release, a new release branch is based off of develop.
    - Fixes to the release are made directly to the release branch.
- When the release has been tested and ready for production (i.e. release to manufacture, etc). the release branch is merged into both master and develop.
    - This is when you tag master with the version number.
- Hot fixes are made in separate branches based off of master.
    - These are the only branches that are based off of master.
    - Once the hot fix is complete, the hot fix branch is merged into both master and develop
    - Tag the version in master.

Example

- So, so, so many commands. See Atlassian's write up for a complete walk through of this workflow with the necessary commands.

Pros

- Ideal for projects with a scheduled release cycle.
- Great for projects where you only need to support one version, for example, your company's website.
- All of the benefits of the Feature Branch workflow, i.e. pull requests, isolated feature development, collaboration.
- Tooling exists to help streamline the process.
- Master only has stable code.
- Allows one team to polish the release while another(s) can work on features for the next release.
- Hot fix branches allow teams to address issues without disrupting other teams working to polish a build or develop new features for the next version.

    - Atlassian refers to this benefit as a "*dedicated channel for hot fixes to production*."
- No commits left behind.

Cons

- Doesn't work well with projects that need to support multiple versions concurrently, for example, supporting code for all currently supported versions of Windows.
    - Welcome to Merge Hell. Population: You.

When should the seasoned Git guru use it?

- Works great for projects that tend to roll forward.

---

## Trunk-based Development

---

**Description 1 (https://trunkbaseddevelopment.com/5-min-overview/):**

Distance

Branches create distance between developers and we do not want that

— Frank Compagner, Guerrilla Games

Assuming any network-accessible source control, physical distance is mitigated by AV technologies including screen sharing. So we will not worry about that so much these days.

Frank's 'distance' is about the distance to the integration of code from multiple components/modules/sub-teams for a binary that could be deployed or shipped. The problematic distance is to code not yet in the single shared branch, that might:

- break something unexpected once merged
- be difficult to merge in.
- not show that work was duplicated until it is merged
- not show problems of incompatibility/undesirability that does not break the build

Trunk-Based Development is a branching model that reduces this distance to the minimum.\

What it is

Notes

Use of "Developers" throughout this site, means "QA-automators" for the same buildable thing, too.
When we say 'the trunk' on this site, it is just a branch in a single repository that developers in a team are focusing on for development. It may be called 'master'. That hints at the fact that the branch in question may literally not be called 'trunk' at all.

There are many deciding factors before a development team settles on Trunk-Based Development, but here is a short overview of the practices if they do:

**Releasability of work in progress**

Trunk-Based Development will always be **release ready**

If an executive manager visited the development team and commanded "Competitor X has launched feature Y, go live now with what we have", the worst response would be "give us one hour". The

development team might have been very busy with tricky or even time-consuming tasks (therefore partially complete), but in an hour, they are able to go live with something just stabilized from the trunk. Perhaps they can do it in less than an hour. The rule, though, is to **never break the build**, and **always be release ready** because the CIO or the business may surprise you.

*Where releases happen*

A key facilitating rule is that Trunk-Based Development teams exclusively **either** release directly from the trunk - see release from trunk, **or** they make a branch from the trunk specifically for the actual release. See Branch for release. Teams with a higher release cadence do the former, and those with a lower release cadence to the latter.

**Checking out / cloning**

All developers in a team working on an application/service, clone and checkout from the trunk. They will update/pull/sync from that branch many times a day, **knowing** that the build passes. Their fast source-control system means that their delays are a matter of a few seconds for this operation. They are now integrating their teammates' commits on an hour-by-hour basis.

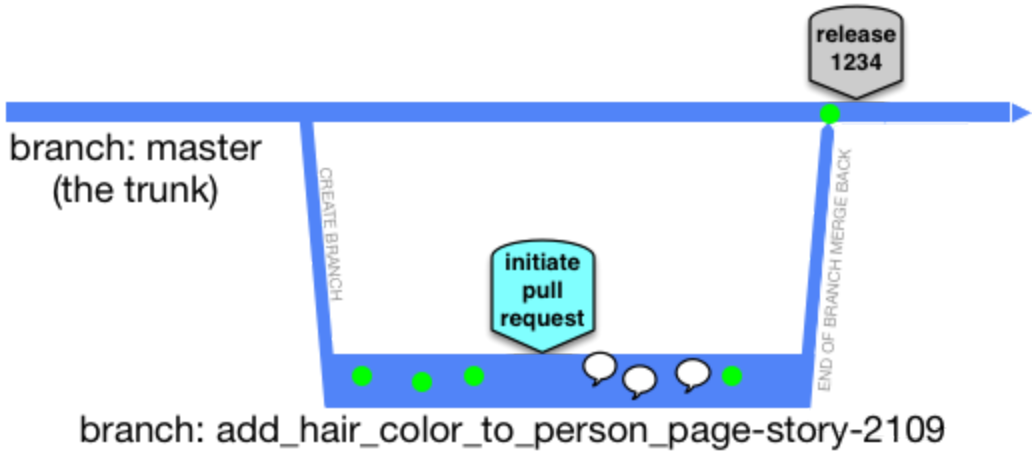
**Committing**

Similarly, developers completing a piece of development work (changes to source code), that does not break the build, will commit it back to the trunk. That it does not break the build should be provable. The granularity of that commit (how many a developer would implicitly do a day) can vary and is learned through experience, but commits are typically small.

The developer needs to run the build, to prove that they did not break anything with the commit **before** the commit is pushed anywhere. They might have to do an update/pull/sync before they commit/push the changes back to the team's version control server, and additional builds too. There's a risk of a race condition there, but let us assume that is not going to happen for most teams.

**Code Reviews**

The developer needs to get the commit reviewed. Some teams will count the fact that the code was 'pair programmed' as an automatic review. Other teams will follow a conventional design where the commit is marshaled for review before landing in the trunk. In modern portal solutions, marshaled nearly always means a branch/fork (Pull Request) that is visible to the team.



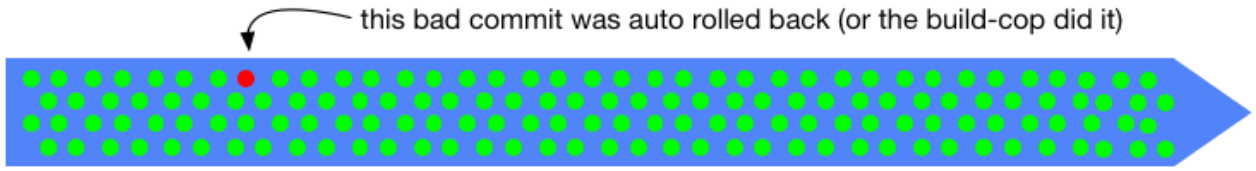

(key)

^ the speech bubbles are stylized code review comments

Code review branches can (and should) be deleted after the code review is complete and be very short-lived. This is tricky for teams new to Trunk Based Development.
Note: You want to keep the commentary/approval/rejection that is part of the review for historical and auditing purposes, but you do not want to keep the branch. Specifically, you do not want the developers to focus on the branch after the code review and merge back to the trunk.

A safety net

Continuous Integration (CI) daemons are set up to watch the trunk (and the short-lived feature branches used in review), and as quickly and completely as possible loudly/visibly inform the team that the trunk is broken. Some teams will lock the trunk and roll-back changes. Others will allow the CI server to do that automatically.




(key)
The high bar is verifying the commit before it lands in the trunk. Short-lived Pull Request branches are the modern place for that.

Developer team commitments

As stated, developers are pledging to be rigorous and not break the build. They're also going to need to consider the impact of their potentially larger commits, especially where renames or

moves were wholesale, and adopt techniques to allow those changes to be more easily consumed by teammates.

Drilling into ‘Distance’

Problematic ‘distance’ has a few tangible examples:

- Late merges of development that happened more than a couple of days ago.
    - Difficult merges in particular
- A breaking build that lowers development team throughput, and diverts resources while it is being fixed

---

**Description 2 (https://www.toptal.com/software/trunk-based-development-git-flow):**

In order to develop quality software, we need to be able to track all changes and reverse them if necessary. Version control systems fill that role by tracking project history and helping to merge changes made by multiple people. They greatly speed up work and give us the ability to find bugs more easily.

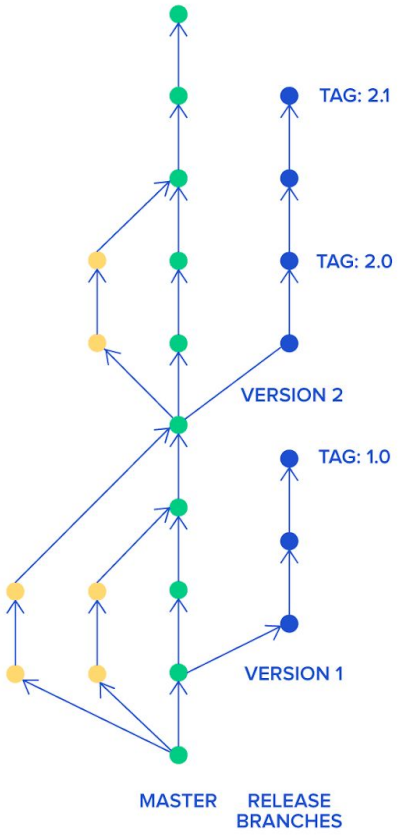
Moreover, working in distributed teams is possible mainly thanks to these tools. They enable several people to work on different parts of a project at the same time and later join their results into a single product.

Git is merely a tool. You can use it in many different ways. Currently, the two most popular development styles you can encounter are Git flow and trunk-based development. Quite often, people are familiar with one of those styles and they might neglect the other one.

In the trunk-based development model, all developers work on a single branch with open access to it. Often it’s simply the `master` branch. They commit code to it and run it. It’s super simple.

In some cases, they create short-lived feature branches. Once code on their branch compiles and passess all tests, they merge it straight to `master`. It ensures that development is truly continuous and prevents developers from creating merge conflicts that are difficult to resolve.

Let’s have a look at trunk-based development workflow.




The only way to review code in such an approach is to do full source code review. Usually, lengthy discussions are limited. No one has strict control over what is being modified in the source code base—that is why it's important to have enforceable code style in place. Developers that work in such style should be experienced so that you know they won't lower source code quality.

This style of work can be great when you work with a team of [seasoned software developers]. It enables them to introduce new improvements quickly and without unnecessary bureaucracy. It also shows them that you trust them, since they can introduce code straight into the `master` branch. Developers in this workflow are very autonomous—they are delivering directly and are checked on final results in the working product. There is definitely much less micromanagement and possibility for office politics in this method.

If, on the other hand, you do not have a seasoned team or you don't trust them for some reason, you shouldn't go with this method—you should choose Git flow instead. It will save you unnecessary worries.

Pros and Cons of Trunk-based Development

Let's take a closer look at both sides of the cost—the very best and very worst scenarios.
When Does Trunk-based Development Work Best?

- When you are just starting up.
  If you are working on your minimum viable product, then this style is perfect for you. It offers maximum development speed with minimum formality. Since there are no pull requests, developers can deliver new functionality at the speed of light. Just be sure to hire experienced programmers.
- When you need to iterate quickly.
  Once you reached the first version of your product and you noticed that your customers want something different, then don't think twice and use this style to pivot into a new direction. You are still in the exploration phase and you need to be able to change your product as fast as possible.
- When you work mostly with senior developers.
  If your team consists mainly of senior developers, then you should trust them and let them do their job. This workflow gives them the autonomy that they need and enables them to wield their mastery of their profession. Just give them purpose (tasks to accomplish) and watch how your product grows.

When Can Trunk-based Development Cause Problems?

- When you run an open-source project.
  If you are running an open-source project, then Git flow is the better option. You need very strict control over changes and you can't trust contributors. After all, anyone can contribute. Including online trolls.
- When you have a lot of junior developers.
  If you hire mostly junior developers, then it's a better idea to tightly control what they are doing. Strict pull requests will help them to improve their skills and will find potential bugs more quickly.
- When you have established product or manage large teams.
  If you already have a prosperous product or manage large teams at a huge enterprise, then Git flow might be a better idea. You want to have strict control over what is happening with a well-established product worth millions of dollars. Probably, application performance and load capabilities are the most important things. That kind of optimization requires very precise changes.

Use the Right Tool for the Right Job

As I said before, Git is just a tool. Like every other tool, it needs to be used appropriately.

Git flow manages all changes through pull requests. It provides strict access control to all changes. It's great for open-source projects, large enterprises, companies with established products, or a team of inexperienced junior developers. You can safely check what is being introduced into the source code. On the other hand, it might lead to extensive micromanagement, disputes involving office politics, and significantly slower development.

Trunk-based development gives programmers full autonomy and expresses more faith in them and their judgement. Access to source code is free, so you really need to be able to trust your team. It provides excellent software development speed and reduces processes. These factors make it perfect when creating new products or pivoting an existing application in an all-new direction. It works wonders if you work mostly with experienced developers.

**Still, if you work with junior programmers or people you don't fully trust, Git flow is a much better alternative.**

Equipped with this knowledge, I hope you will be able to choose the workflow that perfectly matches your project.

Understanding the basics

- What is a trunk in software?
  In the world of software development, "trunk" means main development branch under a version control system. It's the base of a project, where all improvements are being merged together.
- What is code branching and merging?
  **Branches are created from the base project, in order to develop a new feature, fix a bug, or simply do some refactoring.** They prevent software developers from disturbing each other and enable them to work in parallel. **Once a change is finished and tested, the branch is merged back to the trunk.**
- What is a branch in version control?
  **A branch in a version control system is a duplicate of the base project.** It's created so that changes can happen in parallel across other branches. It essentially solves the problem of working on the same files at the same time.
- What is a feature branch?
  **The feature branch has a clear and highly-focused purpose—to add specific functionality to a project. It shouldn't contain any other changes that fix bugs, introduce other features, or are part of a refactoring**.
- What is a version control system?
  Version control systems trace changes occurring in files in a project. They can be recalled or reviewed later. It's extremely useful for bringing back previous versions as well. This allows developers to find bugs with less effort, as they can see and track all the changes as they occurred.

---

## Cactus Model

---

**Description 1 (https://barro.github.io/2016/02/a-succesful-git-branching-model-considered-harmful/):**

When people start to use git and get introduced to branches and to the ease of branching, they may do a couple of Google searches and very often end up on a blog post about A successful Git branching model. The biggest issue with this article is that it comes up as one of the first ones in many git branching related searches when it should serve as a warning on how not to use branches in software development.

What is wrong with “A successful Git branching model”?

To put it bluntly, this type of development approach where you use shared remote branches for everything and merge them back as they are is much more complicated than it should be. The basic principle in making usable systems is to have sane defaults. This branching model makes that mistake from the very beginning by not using the master branch for something that a developer who clones the repository would expect it to be used, development.

Using individual (long lived) branches for features also make it harder to ensure that everything works together when changes are merged back together. This is especially pronounced in today’s world where continuous integration should be the default practice of software development regardless how big the project is. By integrating all changes together regularly you’ll avoid big integration issues that waste a lot of time to resolve, especially for bigger projects with hundreds or thousands of developers. This type of development practice where every feature is developed in its own shared remote branch drives the process naturally towards big integration issues instead of avoiding them.

Also in “A successful Git branching model” merge commits are encouraged as the main method for integrating changes. I will explain next why merge commits are bad and what you will lose by using them.

What is wrong with merge commits?

“A successful Git branching model” talks how non-fast-forward merge commits can be thought of as a way to keep all commits related to a certain feature nicely in one group. Then if you decide that a feature is not for you, you can just revert that one commit and have the whole feature removed. I would argue that this is a really rare situation that you revert a feature or that you even get it done completely right on the first try.

Merges in git very often create additional commits that begin with the message that looks like the following: “`Merge branch 'some-branch' of`

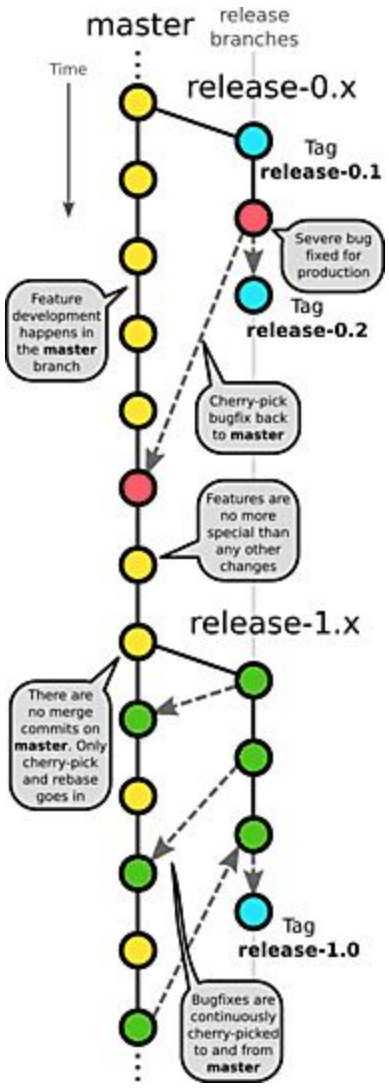
`git://git.some.domain/repository/`". That does not provide any value when you want to see what has actually changed. You need go to the commit message and read what happens there, probably in the second paragraph. Not to mention going back in history to the branch and trying to see what happens in that branch.

Having non-linear history also makes git bisect harder to do when issues are only revealed during integration. You may have both of the branches good individually but then the merge commit fails because your changes don't conflict. This is not even that hard to encounter when one developer changes some internal interface and other developer builds something new based on the old interface definition. These kind of can be easy or hard to figure out, but having the history linear without any merge commits could immediately point out the commit that causes issues.

Something more simple

The cactus model

Let me show a much more simple alternative, that we can call the cactus model. It gets the name from the fact that all branches branch out from a wide trunk (master branch) and never get merged back. Cactus model should reflect much better the way that comes up naturally when working with git and making sure that continuous integration principles are used.




In figure 1 you can see the principle how the cactus branching model works and following sections explain the reasoning behind it. Some principles shown here may need Gerrit or similar integrated code review and repository management system to be fully usable.
All development happens on the master branch

master branch is the default that is checked out after git clone. So why not also have all development also happen there? No need to guess or needlessly document the development branch when it is the default one. This only applies to the central repository that is cloned and kept up to date by everyone. Individual developers are encouraged to use local branches for development but avoid shared remote branches.

Developers should git rebase their changes regularly so that their local branches would follow the latest origin/master. This is to make sure that we do not develop on an outdated baseline.

Using local branches

Cactus model does not to discourage using branches when they are useful. Especially an individual developer should use short lived feature branches in their local repository and integrate them with the `origin/master` whenever there is something that can be shared with everyone else. Local branches are just to make it more easy to move between features while commits are tested or under code review.

Figures 2 and 3 show the basic principle of local branches and rebases by visualizing a tree state. In figure 2 we have a situation with two active local development branches (fuchsia circles) and one branch that is under code review (blue circles) and ready to be integrated to origin/master (yellow circles). In figure 3 we have updated the origin/master with two new commits (yellow-blue circles) and submitted two commits for code review (blue circle) and consider them to be ready for integration. As branches don't automatically disappear from the repository, the integrated commits are still in the local repository (gray circles), but hopefully forgotten and ready to be garbage collected.

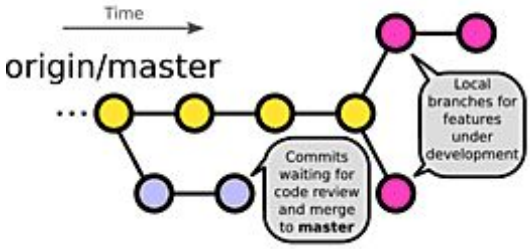



Figure 2: Local development branches

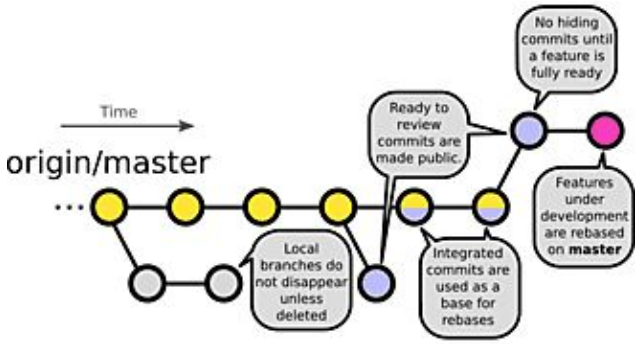



Figure 3: Local branches after integration and rebase

Shared remote branches

As a main principle, shared remote branches should be avoided. All changes should be made available on `origin/master` and other developers should build their changes on top of that by continuously updating their working copies. This ensures that we do not end up in integration hell that will happen when many feature branches need to be combined at once.

If you use staged code review system, like Gerrit or github, then you can just `git fetch` the commit chain and build on top of that. Then `git push` your changes to your own repository to some specific branch, that you have hopefully rebased on top of the `origin/master` before pushing.

Releases are branched out from origin/master

Releases get their own tags or branches that are branched out from `origin/master`. In case we need a hotfix, just add that to the release branch and cherry-pick it to the master branch, if applicable. By using some specific tagging and branching naming scheme should enable for automatic releases but this should be completely invisible to developers in their daily work.

There are only fast-forward merges

`git merge` is not used. Changes go to `origin/master` by using `git rebase` or `git cherry-pick`. This avoids cluttering the repository with merge commits that do not really provide any real value and avoids christmas tree look on the repository. Rebasing also makes the history linear, so that `git bisect` is really easy to use for finding regressions.

If you are using Gerrit, you can also use cherry-pick submit strategy. This simply enables putting a collection of commits to `origin/master` at any desired order instead of having to settle for the order decided when commits were first put for a code review.

Concluding remarks

Git is really cool as a version control system. You can do all kinds of nifty stuff with it really easily that was hard or impossible to do. Branches are just pointers to certain commits and that way you can create a branch really cheaply from anything. Also you can do all kinds of fancy merges and this makes using and mixing branches very easy. But as with all tools, branches should be used appropriately to make it easier for developers to their daily development tasks, not harder by default.

I have also seen these kind of scary development practices to be used in projects with hundreds of developers when moving to git from some other version control systems. Some organizations even take this as far as they fully remove the `master` branch from the central repository and create all kinds of questions and obstacles by not having a sane default that is expected by anyone who has used git anywhere else.

---