Nour El-Deen Abou El-Kassem

38 Followers About Follow

Get git-flows differences under your belt

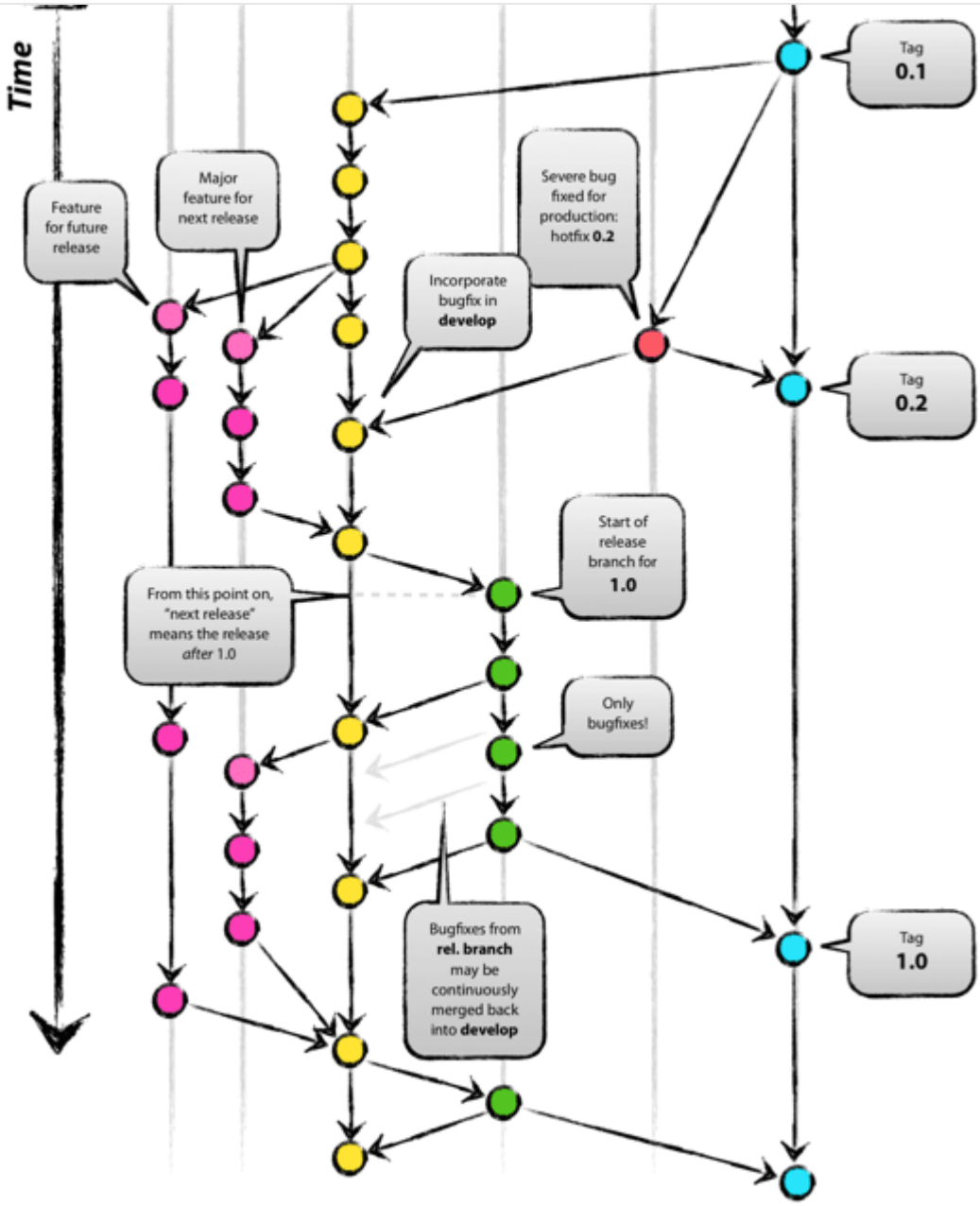
Nour El-Deen Abou El-Kassem 2 hours ago · 10 min read

A Git-flow is meant to track, organize your contribution, keeps a clean history of your contribution, and ease the process of adding new features, maintaining and publishing our software.

This is a subjective target since not all of our projects are the same, even the environment and the process that starts with a feature on your Jira board till the production stage varies from a company to another.

In this article, I will explain the differences between GitFlow, GitHub flow, Trunk-based, and GitLab flow. In terms of contribution, branching models and management, reviewing, releasing, pros & cons, and when should we use this git-flow?

So let’s start with the well-known GitFlow.



Gitflow

Gitflow

Contributing in Gitflow

- You have 2 main long-lived branches which represent the contribution *master*, *development*.
- *development* is responsible for contributions in terms of adding new features and fixes.

short-lived branches then merge them into the long-lived ones.

- For a new feature, developers will create a feature branch of the *development* then start their work which will be represented in a series of commits. After this, they will open a pull request to merge their work into *development*.
- For a fix, that is not for a release and it is not a hotfix, a normal fix that will keep the *development* branch stable. Developers will follow the same approach we mentioned above for adding new features.
- For a hotfix, you will create a branch from the release that has the issue on *master*, then add the commit that will fix that release. This branch should be merged on both *development* and *master*. So you will ensure that this bug won't raise again since both of them are now have the fix.

Reviewing in Gitflow

- As we mentioned above, you will always review your code whenever you have a new feature, fix, hotfix, or even a release. All the kinds of contributions are done on short-lived branches that will be merged on the long-lived ones.
- Reviewing will be done through pull requests.
- Short-lived branches should always be reviewed before they got merged.
- Reviewing does not necessarily mean code review it can also be a QA/QC review.

Releasing in Gitflow

- You will create a release branch starting from development. Developers will contribute to the release branch, for instance, they can add a fix after the QA/QC team reviewed the release.
- When the release branch is stable and has been reviewed by QA/QC team as well as your teammates, you can merge it back into *development* and then into the master to be released.

Short-lived branches

- Release branches
- Fix & Hotfix branches
- Feature branches

Long-lived branches

- *development*
- *master*

Pros

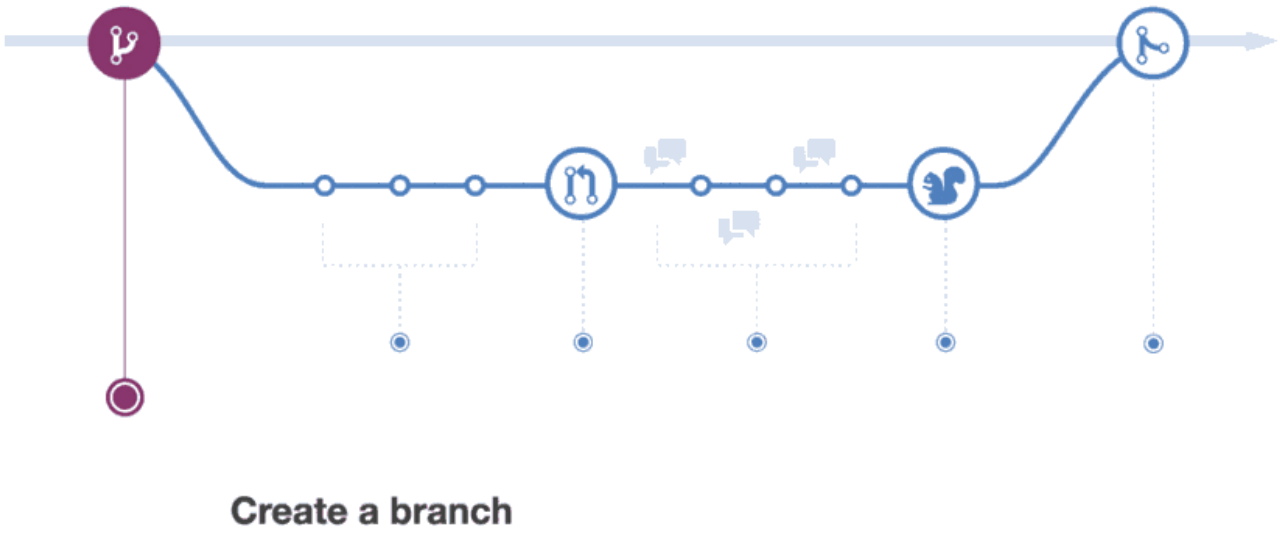
- Gitflow is common and many software engineers have experience with it.
- Since it is common it has many tools that ease the use of it like Gerrit.
- You can easily link the master branch with any CI/CD tool since a merge into master means a new release.

Cons

- Doesn't help in CI/CD concept because all of the software engineers' contribution is done on short-lived branches, not the long-lived ones.
- *development* may don't have a recent activity for a couple of days.
- *master* won't have a clear history of your released features however it will have a clear history of your releases by tags.
- Too many short-lived branches put pressure on garbage collecting the outdated branches.
- Some sort of duplication between the *master* and *development* branches.
- Developers should contribute to the *development* branch, not the *master*.

When should we use it?

- For a team that most of them are juniors.
- For companies that don't rely on quick features delivery.



GitHub flow

GitHub flow

Contributing in GitHub flow

- We have our *main* branch aka *master* we create a branch from the main branch for whatever contribution we need to do, for instance, feature fix, hotfix.
- We start our series of commits then we open a pull request then have our discussion and cycle or review and comments resolving from your teammates and QA/QC team.
- Once you have an approved contribution from QA/QC team and your teammates then you will deploy your work from that branch before merging it.
- Once you deployed your branch you are able to merge it. For hotfixes, you will create another branch from main add your fix to it, review it, then merge it.

- The *main* branch always has the latest work and a clean history of your contributions.

Reviewing in GitHub flow

- You will review your commit by opening a pull request from your branch into *main*.
- You will have a cycle of comments and commits that resolving them until your contribution gets approved by both of QA/QC team and your teammates.

Releasing in GitHub flow

- You will release your work from the branch you have created from the *main* branch and before merging it.

Short-lived branches

- Any branch you will create from the *main*.

Long-lived branches

- *main* branch.

Pros

- Simple and easy to learn and follow.
- Speed the process of development.
- Enables CI/CD.
- *main* always deployable.

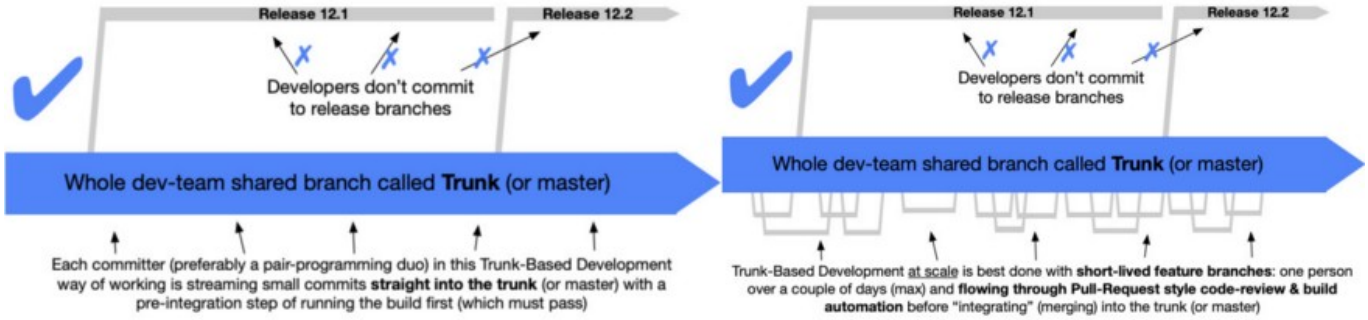
Cons

- Not suitable for mobile application development.
- Not well-organized. Its speed comes at the cost of comprehensiveness and may make it harder to manage the overall development process.

- The master branch can become cluttered more easily since it functions as both the production *and* development branches.

When should we use it?

- Suitable for extreme programming which can deploy new features many times a day.
- It is highly used for front-end teams since they need to deploy their work as soon as possible.



Trunk-based development

Trunk-based Development

Contributing in trunk-based have two cases.

- For small trunk-based systems, developers contribute in a single branch which is called *trunk* aka *master*.
- For scaled trunk-based systems, developers contribute in a feature branch then merge it to the *trunk*.

Reviewing in trunk-based have two cases

For a small trunk-based system, the review process can be done in three different ways or any similar way that applies the same concept.

- Through the code portal, which means by looking into the series of commits on GitHub, check the changes that this commit did, and then the review will be

pushing it.

- Through a plugin in your IDE that shows the changes in your commit.
- In conclusion, the point is to use *trunk* only for your contribution and to review as soon and fast as possible.

For large scaled tunk-based system, since we have a feature branch so the developers will review it by opening a pull request.

For a small trunk-based system, the QA/QC team will always review the *trunk*. Whenever you have a new feature added to it you will notify them and they will review it. Their feedback will be considered in the continuous integration.

For a scaled trunk-based system, the QA/QC team can review the *trunk* as same as we do for small systems or they can review the feature branches before they got merged, which means that the feature branch won't be merged unless both of QA/QC team and your teammates review it. Since we have a rapid contribution so the reviewing cycle won't take much time.

Releasing in trunk-based have two cases.

You can release from a tag in the *trunk* without having a release branch.

You can create a release branch with the following policies.

- The release branches should be created at a late time before the scheduled release date.
- Developers should never commit to a release branch you will cherry-pick needed commits from *trunk* only.
- Release branches are never merged back into the *trunk*.
- If you have a bug fix this should be done on the *trunk* then cherry-pick it to the release branch.

which means you won't delete the release branch if you released a new version of your code because of a bug fix.

- Since release branches will be deleted in the next planned release so you have to tag the last commit that was picked for the current release with a tag.

Short-lived branches

- Release branches
- Feature branches for large-scaled teams

Long-lived branches

- Trunk (Master/Mainline)

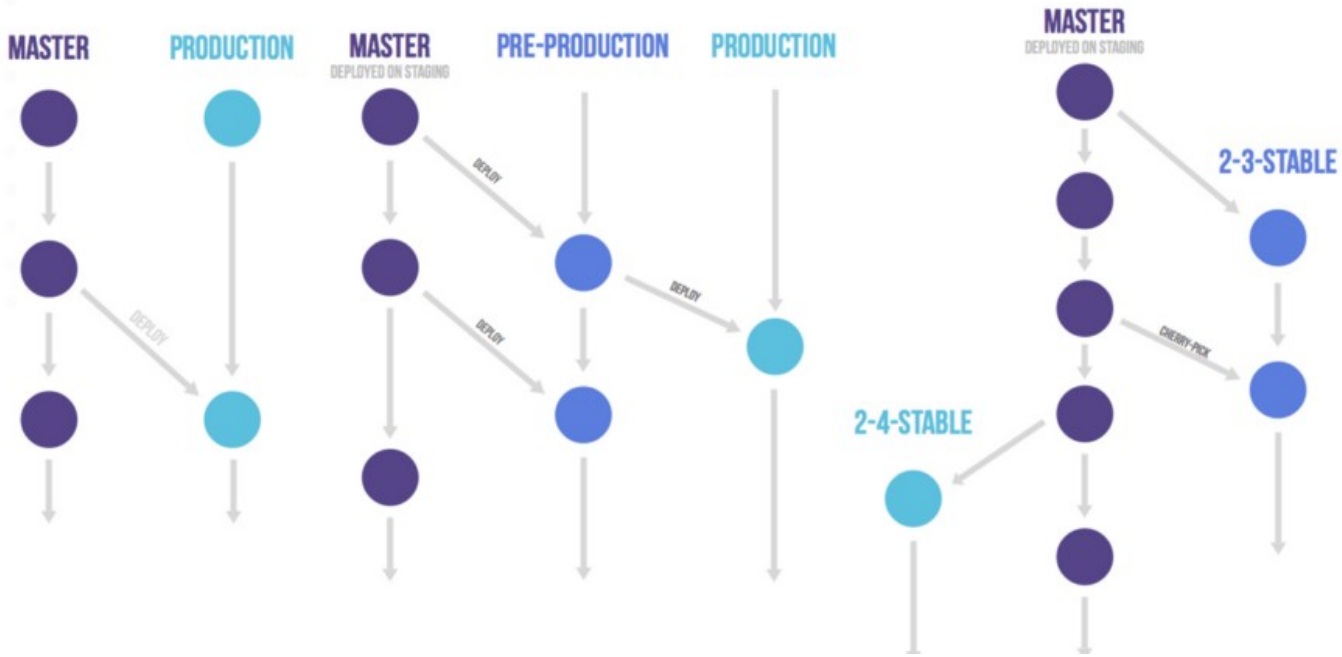
Pros

- Enable continuous integration since all of the team members to commit to the *trunk* multiple times a day and continuous delivery since you always have a stable *trunk* with the latest features so you can generate a release at any time or for your planned releases.
- Avoid merging hell.
- Increase team collaboration.
- You will always have the recent features in the *trunk*.

Cons

- Code review for small teams may be a little risky since you do this on the code portal itself. So if a software engineer pushed a commit that causes a bug it will harm your *trunk*. Which should not be done.
- You can't have a fixed branch for releasing so this will not give you the flexibility to integrate with any CI/CD tool for automated deployments.

- When you need to go faster, for instance, if you are a start-up and you are racing with other competitors.
- When most of your team are experienced seniors.



GitLab flow

Contributing in GitLab flow

- Developers will contribute to the *master*, they will create a feature, fix, release-preparing, or even a hotfix branch from *master* to add their commits and then merge it back to *master* after a cycle of reviewing and comments resolving between you, teammates, and QA/QC team until your contribution got approved.
- The *master* branch is concerned with your contribution. It should show a clean history of your different types of contributions, for instance, new features, bug fixes, hotfix, or new release preparation.
- In GitLab flow, the branching model for releasing a new version of your software varies based on your system type. Different system types are explained in the releasing section below. The main point to be mentioned here that is the

Reviewing in GitLab flow

- When you create a branch from *master* to add your commits, you should go into a cycle of reviewing and comments resolving between you, teammates, and the QA/QC team until your contribution got approved. Then you are able to merge it into *master*.

Releasing in GitLab flow

- For a single-environment system, you will have a *production* branch and a *master*. The production branch is concerned only with the deployments/releases. So whenever you need to release a new version of your software you will merge *master* into *production*.
- For a multi-environment system, for instance, if you have *staging* and *pre-production* environments and the *production* one. Developers will merge the *master* into *staging* to deploy on a staging environment. When they need to deploy in a *pre-production* environment they will merge the *master* into the *pre-production* environment to deploy on it. If you are good to go production then you will merge *pre-production* into *production* to follow the concept of downstream commits. If there is a bug in your deployment then you will fix it on *master*, you will create a hotfix/fix branch from *master*, add the commit that fixes this bug then you will merge *master* into *staging* to test it on a staging environment, then into *pre-production* to test on it, and then merge the *pre-production* into *production* to go live.
- For a multi-version system, for instance, if you have 2 different versions of your software 2.x (a new version of your software that has a new architecture or a completely new UI) and 1.x. (the old version of your software but you still support it with new versions and bug fixes). Then you will create 2 release branches for both of your supported versions and the *master* one. If you have more than 2 you will create a release branch for each of them. Whenever you need to release a new version for any of your software versions, you will cherry-pick the commits that are concerned for this version and add them to its release branch. If you think that you may have conflicts for the picked commits you can create a branch from master cherry-pick

pick the commit that prepares for your version's release as well as the other commits into the release branch. You should know that release branches are concerned only with your release history.

Short-lived branches

- Any branch that is created from master.

Long-lived branches

- *master* branch.
- Different Production branches / Release branches

Pros

- Enables CI/CD
- Simpler than Gitflow and covers many scenarios & different system types than GitHub
- Easy to learn
- Keeps the *master* branch always stable since you take the cycle of review before merging
- Keeps a clean history on *master* for your contribution and for production/release branches for your releases.
- Eliminates the merge hell since you merge all of your contributions to *master*.
- Suitable for any kind of software development.
- You can integrate easily with any CI/CD tool since you have a separate long-lived branch for your releases.

Cons

When should we use it?

- As we mentioned this flow fits in many systems and for different kinds of software development even it is not the best to be used here, for instance, GitHub is better than GitLab flow for SaaS applications. However, GitLab still fits in such a case like this.

Finally

After getting the differences between the mentioned git-flows. You are now able to pick the most convenient and suitable one for your environment and consider those best practices as well.

Git Github Gitlab Gitflow Flow