In Present day software development, speed and ability to rapid change are important when it comes to developing Software and releasing software. However, when you have a big team working simultaneously, branching and merging code can become a disordered very fast. Therefore, teams need to have a process in place to implement multiple changes at once. This is where having an efficient branching strategy comes up as a mandate priority for these teams.

Coming to Branching Strategy,

**Branching Strategy:**

Branches are used for teams to develop features giving them a separate workspace for their code. These branches are usually merged back to a master branch upon completion of work. In this way, features (bugs and bug fixes) are kept apart from each other allowing you to fix mistakes more easily.

A branching strategy, therefore, is the strategy that software development teams adopt when writing, merging and deploying code when using a version control system. It is essential to have the set of rules that developers can follow and how they interact with a shared codebase.

Such a strategy is necessary as it helps keep repositories organized to avoid errors in the application and worry about merge hell when multiple developers are working simultaneously and are all adding their changes at the same time. Such merge conflicts would eventually put off shipping code quickly and thus delay from creating and maintaining an efficient DevOps process. he whole purpose of [DevOps](https://www.flagship.io/devops/) is creating a fast workflow that would allow for the release of small batches of code.

Thus, adhering to a branching strategy will help resolve this issue it enables teams to work in parallel to achieve releases faster and fewer conflicts by creating a clear process when making changes to source control.

When we discuss about branches, we are referring to independent lines of code that branch off the master branch, allowing developers to work independently before merging their changes back to the code base.

**Git Branching**

Branches are not just exclusive to Git. Consequently, before we go through into the various branching strategies, including Git branching strategies.

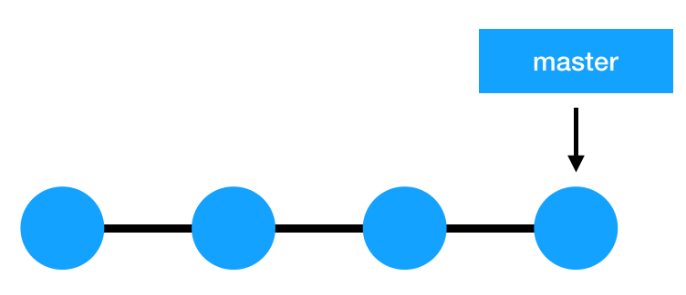
In simple understanding,  [Git and other version control tools allow developers to track, manage and organize their code](https://www.flagship.io/glossary/version-control/). Git branches allow developers to diverge from the main branch by creating separate branches to isolate code changes. The default branch in Git is the master branch.

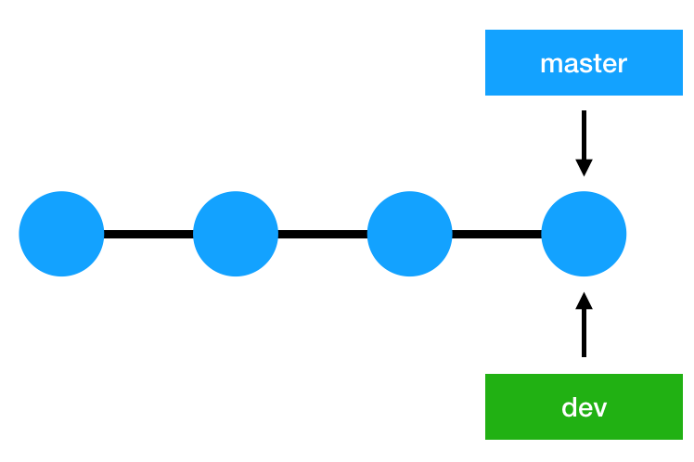
The biggest advantage of a Git branch is that data consists of a series of snapshots so with every commit you make, Git takes a picture of what your files look like at that moment and stores a reference to that snapshot. This means that these branches aren’t just copies of the file system but simply a pointer to the latest commit. In other VCS tools store information as a list of file-based changes which may slow things down and use up significant space.

**PRO’s**

**In Git, a branch is essentially a reference or a pointer to the latest commit in a given context; it’s not a container for commits. In the new branch as you create new branches, Git track the changes by creating new pointers. Git branches, then, can be seen as a pointer to a snapshot of our changes.**

The image below explains, where here the top image shows the master branch and a pointer pointing to the last commit and the image right below it shows what happens when you create a new branch called ‘dev’- a new pointer now points to the latest commit.





To sum up, **the Git branching model is easy to go when compared to other version control systems; this is why it’s so easy and cheap to create branches in Git,**as the whole code doesn’t need to be copied to the branch creating a large number of duplicate files, as opposed to other VCS tools.

**Common Git branching strategies are:**

* [Git Flow](https://www.flagship.io/git-branching-strategies/#gitflow)
* [GitLab Flow](https://www.flagship.io/git-branching-strategies/#github-flow)
* [Github Flow](https://www.flagship.io/git-branching-strategies/#gitlab-flow)
* [Trunk-based development](https://www.flagship.io/git-branching-strategies/#trunk-based-development)
  1. **Git Flow**

[Git Flow](https://nvie.com/posts/a-successful-git-branching-model/)  enables parallel development where developers can work separately from the master branch on features where a feature branch is created from the master branch. Afterwards, when changes are complete, the developer merges these changes to the master branch for release.

This branching strategy consists of the following branches:

* **Master**
* **Develop**
* **Feature**- to develop new features that branches off the develop branch
* **Release**- help prepare a new production release; usually branched from the develop branch and must be merged back to both develop and master
* **Hotfix**- also helps prepare for a release but unlike release branches, hotfix branches arise from a bug that has been discovered and must be resolved; it gives developers to keep working on their own changes on the develop branch while the bug is being fixed.

The main and develop branches are considered to be the main branches, while the rest are supporting branches that are meant to support parallel development among developers.

**Pros and Cons**

* Benefit of this model is that it allows for parallel development to protect the production code so the main branch remains stable for release while developers work on separate branches.
* This strategy contains separate and straightforward branches for specific purposes though for that reason it may become complicated for many use cases.
* It is also ideal when handling multiple versions of the production code.
* Moreover, the various types of branches make it easier for developers to organize their work.
* However, as more branches are added, they may become difficult to manage as developers merge their changes from the development branch to the main. Developers will first need to create the release branch then make sure any final work is also merged back into the development branch and then that release branch will need to be merged into the main branch.
* In the event that changes are tested and the test fails, it would become increasingly difficult to figure out where the issue is exactly as developers are lost in a sea of commits.

It is not an efficient approach for teams wanting to implement continuous integration and continuous delivery. Thus, in that case a much simpler workflow such as GitHub Flow is recommended.

**2.GitHub Flow**

[GitHub Flow](https://docs.github.com/en/get-started/quickstart/github-flow)is a simpler alternative to Git Flow ideal for smaller teams as they don’t need to manage multiple versions.

The main idea behind this model is keeping the master code in a constant deployable state and hence can support continuous integration and continuous delivery processes.

This model doesn’t have release branches. You start off with the main branch then developers create branches, feature branches that stem directly from the master, to divide their work which are then merged back into main. The feature branch is then deleted.

**GitHub Flow pros and cons**

Github Flow focuses on Agile principles and so it is a fast and streamlined branching strategy with short production cycles and frequent releases.

This strategy also allows for fast feedback loops so that teams can quickly identify issues and resolve them.

Since there is no development branch as you are testing and automating changes to one branch which allows for quick and continuous deployment.

This strategy is particularly suited for small teams and web applications and it is ideal when you need to maintain a single production version.

Thus, this strategy is not suitable for handling multiple versions of the code.

Furthermore, the lack of development branches makes this strategy more susceptible to bugs and so can lead to an unstable production code if branches are not properly tested before merging with the master-release preparation and bug fixes happen in this branch. The master branch, as a result, can become cluttered more easily as it serves as both a production and development branch.

A further disadvantage is as this model is more suited to small teams and hence, as teams grow merge conflicts can occur as everyone is merging to the same branch and there is a lack of transparency meaning developers cannot see what other developers are working on.

**GitLab Flow**

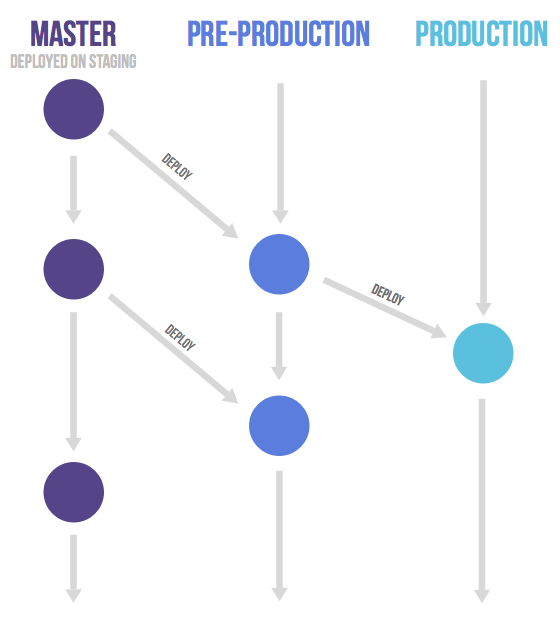
[GitLab Flow](https://docs.gitlab.com/ee/topics/gitlab_flow.html)is a simpler alternative to GitFlow that combines feature-driven development and feature branching with issue tracking.

With GitFlow, developers create a develop branch and make that the default while GitLab Flow works with the main branch right away.

GitLab Flow is great when you want to maintain multiple environments and when you prefer to have a [staging environment](https://www.flagship.io/test-environment/) separate from the production environment. Then, whenever the main branch is ready to be deployed, you can merge back into the production branch and release it.

Thus, this strategy offers proper isolation between environments allowing developers to maintain several versions of software in different environments.

While GitHub Flow assumes that you can deploy into production whenever you merge a feature branch into the master, GitLab Flow seeks to resolve that issue by allowing the code to pass through internal environments before it reaches production, as seen in the image below.



Therefore, this method is suited for situations where you don’t control the timing of the release, such as an iOS app that needs to go through the App store validation first or when you have specific deployment windows.

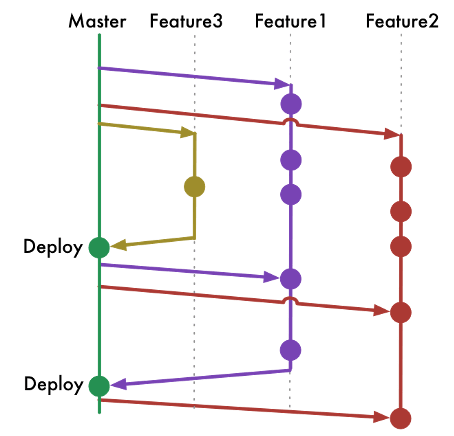
**Trunk-based development**

[Trunk-based development](https://www.flagship.io/glossary/trunk-based-development/) is a branching strategy that in fact requires no branches but instead, developers integrate their changes into a shared trunk at least once a day. This shared trunk should be ready for release anytime.

The main idea behind this strategy is that developers make smaller changes more frequently and thus the goal is to limit long-lasting branches and avoid merge conflicts as all developers work on the same branch. In other words, developers commit directly into the trunk without the use of branches.

Such that Trunk-based development is a key enabler of[continuous integration (CI) and continuous delivery (CD)](https://www.flagship.io/ci-cd/)  since changes are done more frequently to the trunk, often multiple times a day (CI) which allows features to be released much faster (CD).

This strategy is often combined with [feature flags](https://www.flagship.io/feature-flags/). As the trunk is always kept ready for release, feature flags help decouple deployment from release so any changes that are not ready can be wrapped in a feature flag and kept hidden while features that are complete can be released to end-users without delay.

[](https://dev.to/arbitrarybytes/comparing-git-branching-strategies-dl4)

As version control systems matured, various development styles emerged, enabling programmers to find bugs more easily, code in parallel with their colleagues, and accelerate release cadence. Today, most programmers leverage one of two development models to deliver quality software -- Gitflow and trunk-based development.

Gitflow, which was popularized first, is a stricter development model where only certain individuals can approve changes to the main code. This maintains code quality and minimizes the number of bugs. Trunk-based development is a more open model since all developers have access to the main code. This enables teams to iterate quickly and implement [CI/CD](https://www.atlassian.com/continuous-delivery).

As version control systems matured, various development styles emerged, enabling programmers to find bugs more easily, code in parallel with their colleagues, and accelerate release cadence. Today, most programmers leverage one of two development models to deliver quality software -- Gitflow and trunk-based development.

Gitflow, which was popularized first, is a stricter development model where only certain individuals can approve changes to the main code. This maintains code quality and minimizes the number of bugs. Trunk-based development is a more open model since all developers have access to the main code. This enables teams to iterate quickly and implement CI/CD.

**Trunk-based development pros and cons**

As we’ve seen, trunk-based development paves the way for continuous integration as the trunk is kept constantly updated.

It also enhances collaboration as developers have better visibility over what changes other developers are making as commits are made directly into the trunk without the need for branches.

* This is no like the other branching methods where each developer works independently in their own branch and any changes that occur in that branch can only be seen after merging into the main branch.
* Because trunk-based development does not require branches, this eliminates the stress of long-lived branches and hence, merge conflicts or the so-called ‘merge hell’ as developers are pushing small changes much more often. This also makes it easier to resolve any conflicts that may arise.

Finally, this strategy allows for quicker releases as the shared trunk is kept in a constant releasable state with a continuous stream of work being integrated into the trunk which results in a more stable release.

However, this strategy is suited to more senior developers as this strategy offers a great amount of freedom which non-experienced developers might find daunting as they are interacting directly with the shared trunk. Thus, for a more junior team whose work you may need to monitor closely, you may opt for a Git branching strategy.

**How to choose the best branching strategy for your team**

When first starting out, it’s best to keep things simple and so initially GitHub Flow or Trunk-based development may work best. They are also ideal for smaller teams requiring only a single version of a release to be maintained. GitFlow is great for open-source projects that require strict access control to changes. This is especially important as open-source projects allow anyone to contribute and so with Git Flow, you can check what is being introduced into the source code.

However, GitFlow, as previously mentioned, is not suitable when we want to implement a DevOps environment. In this case, the other strategies discussed are a better fit for an Agile DevOps process and to [**support** your CI and CD pipeline](https://www.flagship.io/glossary/ci-cd-pipeline/).

The following [table](https://www.alibabacloud.com/blog/how-to-select-a-git-branch-mode_597255) summarizes the strategies and which strategy is appropriate in which context:

| Product type and its release method | Team size | Collaboration maturity | Applicable mainstream branch mode |
| --- | --- | --- | --- |
| All | Small team | High | Trunk-Based Development (TBD) |
| Products that support continuous deployment and release, such as SaaS products | Middle | Moderate | GitHub-Flow and TBD |
| Products with a definite release window and a periodic version release cadence, such as iOS apps | Middle | Moderate | Git-Flow and GitLab-Flow with release branch |
| Products that are demanding for product quality and support continuous deployment and release, such as basic platform products | Middle | Moderate | GitLab-Flow |
| Products that are demanding for product quality and have a long maintenance cycle for released versions, such as 2B basic platform products | Large | Moderate | Git-Flow |

To sum up, there is no such thing as the perfect strategy. The strategy you choose will depend on your team and the nature and complexity of your project and so this should be evaluated on a case-by-case basis.

It’s also fine to start off with one strategy and adapt it over time according to your needs. Needless to say, whatever strategy you end up choosing should aim to increase your team’s productivity by giving them a clear and consistent strategy to organize their work.