In today's software development, speed and flexibility are vital. When multiple developers work together, managing code branches can quickly become messy. That's why teams need an effective branching strategy to handle multiple changes efficiently.

**What is a branching strategy?**

*Branches are primarily used as a means 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 (and any bug and bug fixes) are kept apart from each other allowing you to fix mistakes more easily.This means that branches protect the mainline of code and any changes made to any given branch don’t affect other developers.

*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 essentially a set of rules that developers can follow to stipulate how they interact with a shared codebase.*

* branching strategy will help solve this issue so that developers can work together without stepping on each other’s toes. In other words, it enables teams to work in parallel to achieve faster releases and fewer conflicts by creating a clear process when making changes to source control.
* When we talk 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.

**Why you need a branching strategy**

* A branching strategy is necessary to avoid conflicts when merging and to allow for the easier integration of changes into the master trunk.
* Enhance productivity by ensuring proper coordination among developers
* Enable parallel development
* Help organize a series of planned, structured release
* Map a clear path when making changes to software through to production
* Maintain a bug-free code where developers can quickly fix issues and get these changes back to production without disrupting the development workflow

## Git branching

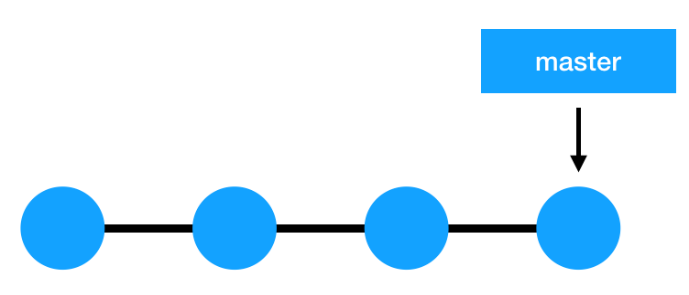
Branches are not just exclusive to Git. Git actually handles branches and why it’s a standout among other VCS tools. (*If you are working on a big software development project that requires cross-team collaborations and frequent changes then you will need to use a version control system. Whichever system you choose will ultimately depend on your organization’s needs. Nonetheless, whether you’re working in a small team or collaborating on a large software development project across teams, version or source control is an important element of the development process*.)

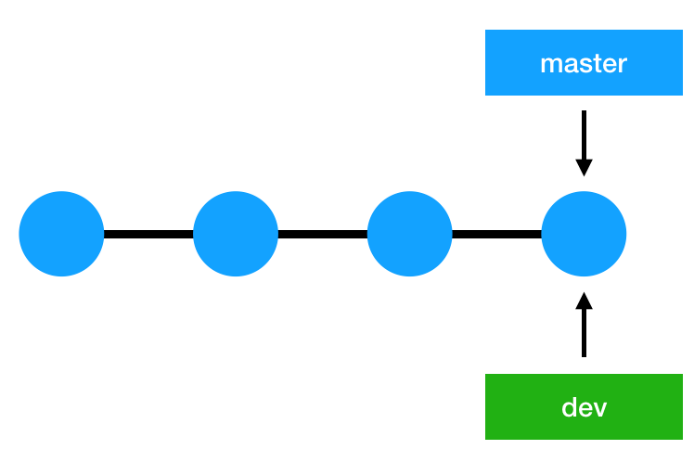
The biggest advantage of a Git branch is that it’s ‘**lightweight’**, meaning that data consists of a series of snapshots so with every **commit** you make, Git takes a picture of what your files look like at that moment and stores a reference to that snapshot. This means that these branches aren’t just storing the copies of our file system like it is done traditionally but it is just creating a pointer to the latest commit that we had made after doing a change and will not save all the files in the branch but a reference to the snapshot taken which is stored with a picture of how all the data or files looks like.

Meanwhile, other VCS tools store information as a list of file-based changes which may slow things down and use up a lot of space.

As you create new commits in the new branch, Git creates new pointers to track the changes. Git branches, then, can be seen as a pointer to a snapshot of your changes.

*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.*





**What is a branch?**

A branch in Git represents an independent line of development having its own working directory, stagging area and committed project history.

**The master branch?**

The first branch that we get when we create a Git repository is called the master branch. Commonly, the master branch holds the production code, the one running in the production environment and your users are interacting with it. Development work is generally done in a separate dev branch which is later merged with the master branch when they are ready for production server.

## What are some common Git branching strategies?

## GitFlow

GitFlow enables parallel development where developers can work separately from the master/main branch on several other branches whicha are created from the master branch.

Afterwards, when changes are complete, the developer merges these changes back to the master branch for release.

This branching strategy consists of the following branches:

* **Master:** The main branch from where we have to take a start
* **Develop/Development**: This branch can be used by developers to do all the development work of the project.
* **Feature**- to develop new features we create/branches off a new branch from the development branch called as feature.
* **Release-** It is used to prepare a new production release. usually branched from the develop branch and must be merged back to both develop and master
* **Hotfix-** hotfix branches arise from a bug that has been discovered and must be resolved. it enables 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, with an infinite lifetime, while the rest are supporting branches that are meant to aid parallel development among developers, usually short-lived*.

**Working:**

We start the project with Master branch and create a new branch name as develop for the purpose to make all the development of the project here in this branch.

The original code is present in the master branch. As in the development process we are supposed to add new features to the project so a new branch feature is branched off from the develop branch. Now the Major feature for the next release or fro future releases are done in this branch and then it is merged to the develop branch.

Now the new versions are supposed to be released ith the new dfeatures added or bugs being fixed by the develop branch, a new branch Release is branched off from the develop branch.

Now the development process is underway with the new features being added by the feature branch by merging and these new features and bug fixing and a new version of it is released and hence release branch consists of the all the new developments so it is merged to Master branch time to time.

If there in case in Master branch there arises a severe bug then a new branch is branched off from Master branch named as Hotfix which work is to fix severe bugs in the Master branch

#### **GitFlow pros and cons**

* 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
* It is also ideal when handling multiple versions of the production code.
* 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.
* GitFlow is not an efficient approach for teams wanting to implement continuous integration and continuous delivery.



## GitHub Flow

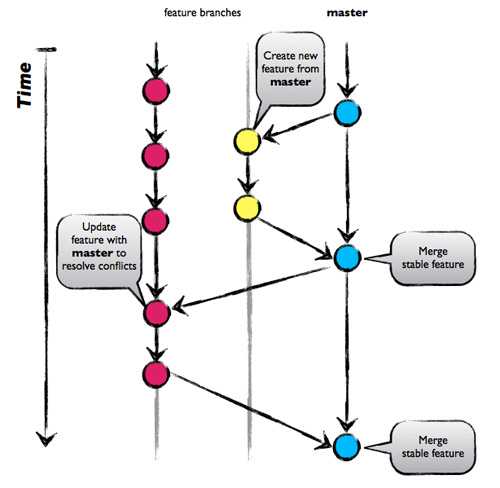
* GitHub Flow is a simpler alternative to GitFlow
* ideal for smaller teams as they don’t need to manage multiple versions**.**

Unlike GitFlow, 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 isolate their work which are then merged back into Master. ***The feature branch is then deleted.***

Working:

It is started from the Master/main branch, then develop branch is created for development but not used all over the project then develop branch further creates new branch features, which is directly linked to the Master branch.

It adds up further features and directly merges it to Master branch without involving Develop branch features are added and then directly merged into the main branch.



**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.

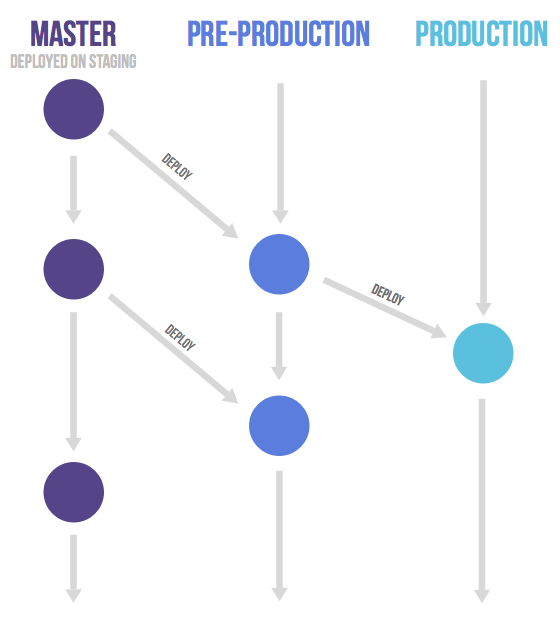
## GitLab Flow

GitLab Flow 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 ***(Master/Pre-Production/Production)*** and when you prefer to have a staging environment separate from the production environment (where all the development occurs I.e, Master Branch). 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.



## Trunk-based development

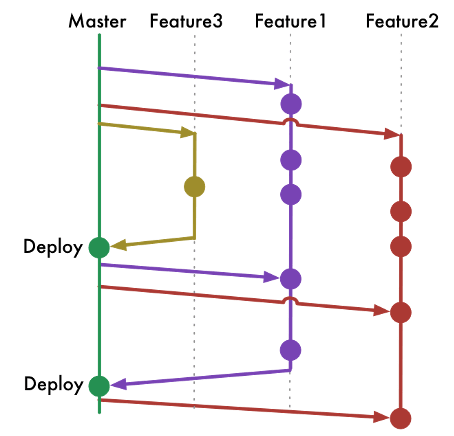
Trunk-based development is a branching strategy that in fact requires **no branches** but instead, developers integrate their changes into a shared trunk at least once a day. This shared trunk should be ready for release anytime.

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.***

Consequently, trunk-based development is a key enabler of continuous integration (CI) and continuous delivery (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. As ***the trunk is always kept ready for release***,

***It only consists of a single branch master which is always open to accept any change/feature\_added made by developer and delivered to Master branch directly and it is released at that particular time. There is no limit to accept any or feature addition by the developer and doesn’t require any branch to reach Master branch.***



* 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.

## CONTINUOUS INTEGRATION AND CONTINUOUS DEPLOYMENT

CI and CD come in to play usually in the middle phase, so before a software product is released into the market but after the testing phase during which the application is designed and trialed

* CONTINUOUS INTEGRATION refers to the process by which new features are integrated automatically into the original app’s existing codebase after errors and tweaks have been identified.
* CONTINUOUS DEPLOYMENT, which involves packaging and deploying applications from the development server to the production server. When the software has been deployed, operations teams move in to do things like configure servers and monitor any issues

# GIT BRANCHING STRATEGIES:

1. **Master Branch:** The master branch is like the main version of your project that is ready for release. It contains the stable and tested code that is used in the production environment.
2. **Integration Branch:** The integration branch is where different changes from developers or teams come together. It ensures that all the code changes work well together before being merged into the master branch. Continuous Integration (CI) tools help automate the process of merging and testing these changes.
3. **Staging Branch:** The staging branch acts as a staging area for final testing. It provides a separate environment where the integrated changes can be thoroughly tested, allowing Quality Assurance (QA) teams to check if everything works as expected before releasing it to the production environment.
4. **Dev/Deploy Branch:** The dev/deploy branch is where active development takes place. Developers create separate branches off this branch to work on specific features or fixes. Once the changes are completed and tested, they are merged back into the dev/deploy branch. Continuous Deployment (CD) tools automate the process of deploying the changes from the dev/deploy branch to the staging or production environment.

The CI/CD (Continuous Integration/Continuous Deployment) process involves using automated tools and practices to integrate code changes frequently, run tests, and deploy the software to different environments. QA teams play a crucial role in testing the changes in the staging branch to ensure high-quality software before it goes into production, which is the live environment where end-users can access and use the software.

GIT BRANCHING STRATEGIES SEQUENCE OF STEPS:

1. Developers create separate feature branches off the dev/deploy branch to work on specific features or fixes. These branches are usually short-lived and dedicated to a single task.
2. Once the development is complete, developers merge their feature branches back into the dev/deploy branch. This helps consolidate the changes from different developers or teams into one central branch.
3. The CI/CD process is triggered when changes are merged into the dev/deploy branch. Continuous Integration (CI) tools automatically build and test the code to ensure it meets quality standards. This helps catch any integration issues or conflicts early on.
4. If the CI tests are successful, the changes are then merged into the integration branch. This branch serves as a central point where all changes come together. It allows for further integration testing and ensures that all the code changes work well together
5. .
6. After successful integration testing, the changes are merged into the staging branch. This provides a separate environment where QA teams can thoroughly test the integrated changes and perform additional quality assurance checks. They ensure that the software functions correctly and meets the desired quality standards before moving forward.
7. Once the changes are approved in the staging branch, they are merged into the master branch. The master branch represents the stable and production-ready code. The changes are now ready for deployment to the production environment.
8. Continuous Deployment (CD) tools are triggered when changes are merged into the master branch. These tools automate the process of deploying the changes to the production environment, making the updated software available to end-users.

This sequence ensures a controlled and structured workflow, with thorough testing and quality checks at each stage before moving changes into production. It allows for collaboration, integration testing, and quality assurance, reducing the risk of introducing errors or issues into the production environment.