DEVOPS BRANCHING STRATEGIES EXPLAINED

In any DevOps environment, version control is one of the primary components of the DevOps pipeline. It effectively allows the following:

- Managing all the source code changes
- Tracking all code changes
- Enabling multiple developers to work on the same project simultaneously

However, if these repositories are not managed properly, they can quickly become bloated and unwieldy, defeating the essential purpose of source control. One of the best ways to keep everything organized is by adhering to a proper branching strategy for all your development needs.

In this article, we will discuss different branching strategies that can be used to streamline your development experience.

(This article is part of our DevOps Guide. Use the right-hand menu to go deeper into individual practices and concepts.)

What is a branching strategy?

Simply put, a branching strategy is something a software development team uses when interacting with a version control system for writing and managing code. As the name suggests, the branching strategy focuses on how branches are used in the development process.

One major purpose of a version control system is to enable a collaborative development environment without overlapping or affecting the codebase. There, each team member modifying the same source code will inevitably be making conflicting code changes. However, we can avoid

such conflicts with a version control system by using branches when writing and merging code to a master branch to create the end product.

Interested in Enterprise DevOps? Learn more about DevOps Solutions and Tools with BMC. ›

Why you need a branching strategy in DevOps

A properly implemented branching strategy will be the key to creating an efficient DevOps process. DevOps is focused on creating a fast, streamlined, and efficient workflow without compromising the quality of the end product.

A branching strategy helps define how the delivery team functions and how each feature, improvement, or bug fix is handled. It also reduces the complexity of the delivery pipeline by allowing developers to focus on developments and deployments only on the relevant branches—without affecting the entire product.

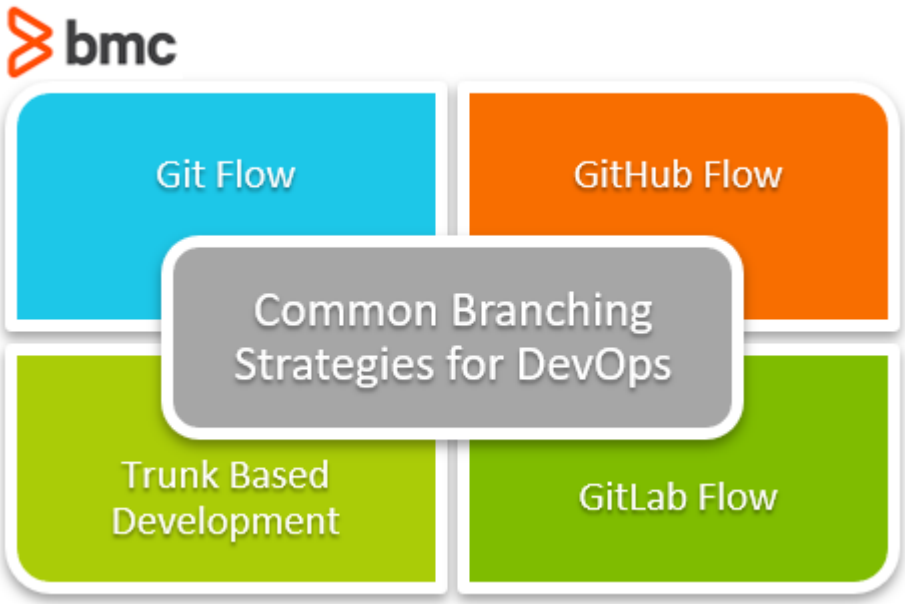
Selecting a branching strategy

The selection process for a branching strategy depends entirely on the users and the project requirements. Factors like the development method, scale, user preferences highly impact this selection. Additionally, other factors like CI/CD tools decide what branching strategies can be used in your DevOps pipeline.

Branching strategies that do not align or make it more difficult to implement Continuous Integration and Continuous Delivery in DevOps pipelines should not be used in a DevOps environment.

A good branching strategy should have the following characteristics:

- Provides a clear path for the development process from initial changes to production
- Allows users to create workflows that lead to structured releases
- Enables parallel development
- Optimizes developer workflow without adding any overhead.
- Enables faster release cycles
- Efficiently integrates with all DevOps practices and tools such as different version control systems
- Offers the ability to enable GitOps (if you require it)



Common DevOps branching strategies

Now we have a better understanding of what a branching strategy is and what we are trying to achieve from it. Let's look at some popular branching strategies currently used in the industry.

(Compare popular version control systems.)

Git Flow

Git Flow is the most widely known branching strategy that takes a multi-branch approach to manage the source code. This approach consists of two main branches that live throughout the development lifecycle.

Primary Branches

- **master.** The primary branch where all the production code is stored. Once the code in the "develop" branch is ready to be released, the changes are merged to the master branch and used in the deployment.
- **develop.** This is where all the actual development happens. All the pre-production code is stored here, and the completed code of all the supporting branches is merged directly to the develop branch.

Support Branches

During the development, developers create different branches for specific use cases using the develop branch as the base. The following are some branches created like that:

- **feature-*** feature branches are used to develop new features and branches off exclusively from the develop branch.
- **hotfix-*** This is to deal with production issues where quick fixes are required. They can branch off from the master itself, but need to be merged to both master and develop branches.
- **release-*** This branch is used to aggregate fixes and improvements and prepare for the production release. It will be branched from the develop branch and merged to both develop

and master.

Advantages of Git Flow

- Straightforward and separate branches for specific purposes with a proper naming convention
- Ideal when handling multiple versions of the production code
- Great for enterprise customers who need to adhere to release plans and workflows
- Clearly defined branches that help define the test scope and test only the specific branches
- Widespread support by most git tools

Disadvantages of Git Flow

- Git history becomes unreadable
- master/develop split can be redundant in most development scenarios
- Can be complicated to integrate with CI/CD tools
- Not recommended when users need to maintain a single production version
- This strategy can overcomplicate the source control depending on the scope of the project

GitHub Flow

As the name suggests, this strategy was introduced by GitHub, aiming to provide a simple and lightweight approach to manage the development. It adheres to the following guidelines when managing the source control with a single primary branch.

- **master.** The primary branch where code is branched off from and merged to. Anything in the master branch is deployable.
- Any change (feature/bug) is made in a new branch derived from the master with a descriptive branch name describing the development.
    - Commit to the development branch locally and regularly push to the branch.
- Create a pull request once the development is done so that the code can be reviewed.
- Once the code is reviewed and approved, it must be tested in the branch before merging to the master branch.
- From this point, users can immediately deploy the master branch with the new changes.

Advantages of GitHub Flow

- Relatively simpler approach with a simple workflow
- Clean and easily readable Git history
- Ability to easily integrate into CI/CD pipelines
- Ideal when you need to maintain a single production version

Disadvantages of GitHub Flow

- An oversimplified approach that is not suitable when dealing with release-based developments
- Not suitable when maintaining multiple versions of the code
- Can lead to unstable production code if branches are not properly tested before merging with the master

Trunk Based Development (TBD)

The Trunk Based Development strategy involves developers integrating their changes directly into a shared trunk (master) at least once a day. This shared trunk is always in a releasable state. Developers can pull from this trunk, create a local repository, and then push the code to the shared trunk.

This regular integration enables developers to view each other's changes quickly and immediately react if there are any conflicts.

Scaled Trunk Based Development

Smaller teams can commit directly to the shared trunk after build and functionality tests. However, for larger teams, development can be broken down into feature/bug-fix branches. Then, developers will push code to specific branches continuously, and this code can be verified via pull requests and tested before finally merging into the shared trunk. This approach enables development teams to both:

- Scale seamlessly without overburdening the shared trunk
- Maintain all the changes in a more organized and manageable manner

When it comes to deployment, TBD uses feature flags to manage the developments in the shared trunk. Using these feature flags, teams can toggle portions of code on or off for the build process and deploy only the necessary code in production environments.

Advantages of Trunk Based Development

- True continuous integration as developers constantly keeps the trunk updated
- Excellent choice for CI/CD pipelines with simpler workflows for automated testing
- Shorter feedback loops for developers as code changes are quickly visible
- Lead to faster release cycles
- Smaller iterations allow teams to keep track of all the changes while reducing code conflicts and improving overall code quality

Disadvantages of Trunk Based Development

- Non-experienced developers might find this approach daunting as they are directly interacting with the shared trunk (master)
- Improperly managed feature flags can lead to issues
- Shifting from more traditional methods such as Git Flow can be difficult

GitLab Flow

The GitLab strategy combines feature-driven development and feature branches with issue tracking. This strategy is similar to GitHub flow yet includes environmental branches such as development, pre-production, and production.

In GitLab Flow, development happens in one of these environmental branches, and verified and tested code is merged to other branches until they reach the production branch. Let's assume that

we have the three environmental branches mentioned above. In that case, the development workflow will be:

1. **development.** This is where all the development happens. Developers will create separate branches for the feature/bug-fix they are working on and merge them to this branch. Then, it will get reviewed and tested.
2. **pre-production.** Once the developed features and fixes are ready to be released, the source code up to that point will be merged to a pre-production branch. Then this code will go through additional testing and finally be merged with the production branch to be deployed.
3. **production.** Once the production-ready code is merged, this branch can be directly deployed in the production environment. This environment branch will only contain production-ready code.

Advantages of GitLab Flow

- Provides proper isolation between environments and ensures a clean state in the branches
- Easily integrates into CI/CD pipelines
- Improves GitHub Flow while streamlining the process for a DevOps environment
- Easier, cleaner to read the git history

Disadvantages of GitLab Flow

- Can be complex to implement with the additional overhead of managing environmental branches
- Development branches can get complicated and messy if not properly managed

How to choose your branching strategy

All the branching strategies mentioned above are already tried and tested branching strategies that can be used to manage your source code. However, each has its own strengths and weaknesses.

- The traditional Git Flow will not be ideal for rapidly evolving DevOps environments.
- The other strategies described here try to improve Git Flow and modernize it to fit an agile DevOps process.

So, as always, you have to select the best strategy that satisfies all your requirements and suits your organizational practices.

Related reading

- BMC DevOps Blog
- Ignoring in Git: How To Use .gitignore Files
- What is CI/CD? Intro to Continuous Integration and Continuous Delivery
- How To Set Up a Continuous Integration & Delivery (CI/CD) Pipeline
- Software Project Management Phases & Best Practices
- DevOps Engineer Roles & Responsibilities