**Cloud DevOps**

Cloud computing provides DevOps automation with a standard and centralized platform for testing, deployment, and production. Using a cloud platform solves many issues with distributed complexity.

DevOps automation is becoming cloud-centric. Most public and private cloud computing providers support DevOps systemically on their platform, including continuous integration and continuous development tools. DevOps automation technology provides centralized governance and control for a sound DevOps process.

Cloud-based DevOps lessens the need to account for resources leveraged. Clouds leverage usage-based accounting, which tracks the use of resources by application, developer, user, data, etc.

Cloud isn't really driving DevOps; rather, DevOps is driving the interest and the growth of cloud

**Trunk-Based Development**

Trunk-based development is a method of version control branch management that seeks to remove complexity and user error by eliminating long-lived software feature branches in favor of developers merging to a single branch called the “trunk.” Choosing an effective Git branching strategy is an underappreciated requirement for software managers. Ineffective branching strategies create friction within and between software teams. This friction slows down the speed of development and leads to human error. When people make mistakes reconciling code between different git branches, bugs surface as a result.

**Feature branches vs. trunk-based development**

When you’re writing code by yourself, managing your Git branches probably isn’t something you think much about. You likely write a little bit of code, then push it to GitHub. That’s the end of your branch management. Likely, if the thing you’re working on is a small project, you commit directly to the primary branch (note: primary branch is used interchangeably with trunk, master, and mainline).

you’ll start by creating a branch from primary, working on it a little bit, and submitting a pull request (PR). That pull request opens up your code for review from your peers. After they approve it, you merge the code which is pushed to an automated process. That process involves a number of steps: compiling the code, checking it for adherence to style guidelines, running unit tests, etc. Assuming all of those steps pass, the automated systems deploy the new code to some sort of shared environment. There, users interact with the new code.

This process remains the same regardless of whether you’re using traditional feature branching or trunk-based development. Instead, what changes is how many times you take those steps, and how much friction exists at each point.

**How feature branch development works**

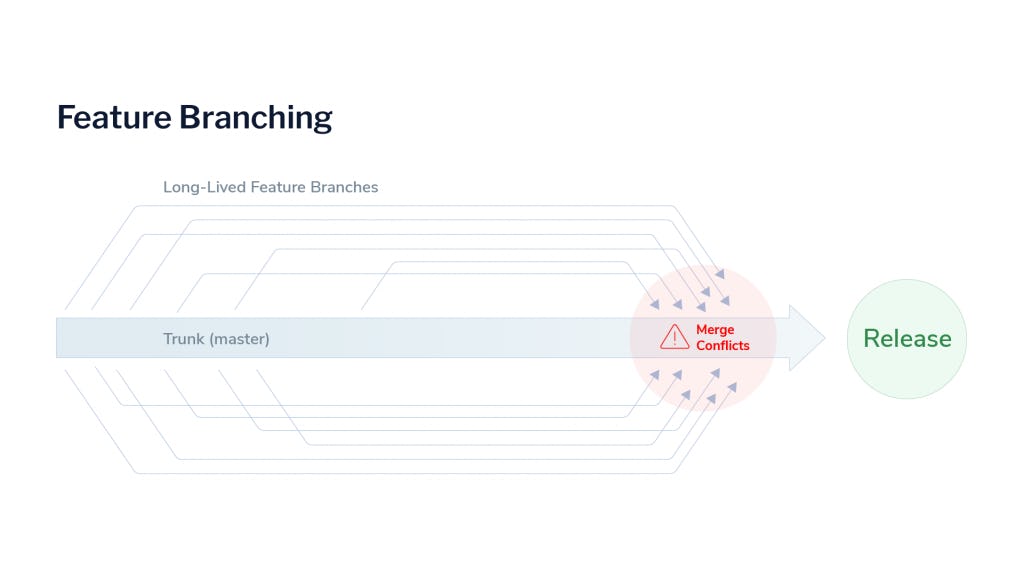
A feature branch or GitFlow branching strategy, their strategy often includes the following branches and processes:

* A primary branch. This is the central branch where everyone starts any new work. Often, this branch is tied to an environment that is used for final user approval before some code graduates to production.
* A release branch. This is the branch a continuous integration pipeline actually deploys to production servers.
* A release-candidate branch. This is the branch where changes between primary and release are reconciled. This branch takes over the user approval environment around the time of a release. Usually, this comes at the end of a sprint, where all the code changes from primary are merged into the release-candidate branch.
* A qa branch. This branch serves the QA team to allow them to test features and bug fixes before heading off for end user approval.
* A dev branch. This branch is the first integration step where developers work on features collaboratively. Often, this branch is also associated with a deployment environment, so that devs can share code and work on something in collaboration with one another in different geographic locations.
* Any number of branches that are tied to the development of new features or bug fixes. These branches are almost always directly tied to a single ticket in an issue tracking database. Usually, these branches do not include an integration environment. Instead, to run the code for one of these branches, a developer would check out the code to their local development environment and run it there. Every change that eventually makes it to the production environment starts here.

**Feature branch development workflow**

The flow for a new feature or bug fix is complicated. A developer will create a new branch with a command like git checkout primary && git pull && git checkout -b ticket-123-bug-fix, then do some work. That command locally checks out the primary branch, pulls the latest changes, then creates a new feature branch for the user to work from.

Since the fix is so simple, Team is able to do code review in under an hour. They assign the ticket to the QA team. The QA team verifies the bug fix on the QA branch, so the developer makes a pull request to the primary branch. Again, the fix is simple, so approval comes swiftly. After perhaps a day of work, the developer merges their changes into primary. Even at this point, the code’s journey isn’t done. It will still need to move to the release-candidate and release branches before making it production. These steps usually happen at a set time. Many of the teams often have a release cadence of two weeks, to match up with sprint windows.



**How trunk-based development works**

A trunk-based development flow is much simpler. There are only two primary sources of branches:

* The trunk branch (hence the name of the branching strategy). This is the branch that’s deployed to any integration environments—the branch that’s deployed to production, the one that’s deployed to user acceptance environments.
* Feature branches for new work or bug fixes. These branches are just like the feature-branch development branches. This is where developers make changes. When they’re done, they create a pull request to the trunk branch.

**Trunk-based development workflow**

The flow for a new feature or bug fix is quite simple. The developer creates a new branch by running git checkout trunk && git pull && git checkout -b ticket-1234-bug-fix. Once again, this checks out the trunk branch, pulls changes, and then creates a new branch to address the bug fix.

Our bug fix is just as simple as it was last time. Easily verified, and just a few lines of code. The developer creates a pull request, to the trunk branch. With this pull request, they include a git tag that identifies their changes. A developer reviews their code and verifies the fix, so they approve the pull request.

The developer merges the code to trunk and verifies that it works in an integration environment. They promote their tag to the QA environment. This tag deploys some new code for the QA team, which they’re able to quickly verify. The QA team promotes that tag to the user acceptance environment. Customers get their hands on it. The operations team verifies it and they promote the tag to the production system. An automated system takes the tag, checks out the code, builds it, and deploys it.



**The benefits of trunk-based development**

The ‘lead time for changes’ is much shorter in a trunk-based development strategy, but that’s not the only advantage. Let’s run through the benefits of adopting a trunk-based strategy.

**Short-lived branches**

Short-lived branches are one of the main advantages of a trunk-based strategy. Go back and look again at the path that a branch takes to get to release in the feature branch flow. No branch in a feature flow is truly short-lived. It had half a dozen touchpoints where the branch needed a pull request into yet another branch.

That’s a lot of overhead for a developer to manage. In the trunk-based development flow, there is a single touchpoint. While the code still progresses through the same acceptance steps, there is only one pull request. By minimizing the number of pull requests as the new changes flow through the system, we minimize the friction inherent in the pull request process and the potential slowdowns between writing code and delivering it to the customer.

**Fewer merge conflicts**

Merge conflicts pose a major challenge in code quality as team size grows. Merge conflicts happen often in GitFlow systems. This is because every branch the team uses is a moving target. When the developer creates an initial branch from the primary branch, they’re taking a snapshot of that code and then changing it. When it comes time to re-integrate their changes with another branch (i.e., submit a merge request), that branch has changed.

In a trunk-based system, developers also code against a moving target. But the potential for conflicts is minimized due to the fact that there is just one branch to send a pull request to.

Merge conflicts need to be resolved by humans. That introduces human error into the equation, which is always risky. This is especially true the later in the process a merge conflict arises. A conflict that surfaces during the merge to a dev branch can be quickly sorted out between two developers. A conflict that arises when trying to move from primary to release is a hairy situation that requires a great deal of caution.

Every time developers manually resolve a release conflict, they risk introducing new bugs into the system. Sometimes, they might delete a feature entirely. The long-lived nature of feature branches heightens the likelihood of encountering merge conflicts along the way. we created made it through the entire process in about a day—until it got to primary. Then it had to sit there for up to a week or more. That’s a lot of opportunity for new conflicts to arise, and when they do they have a high likelihood of introducing bugs.

With trunk-based development, software developers can avoid this “merge hell”, reduce the odds of human error, and increase their productivity.

**Quicker releases: Continuous Integration and Continuous Delivery (CI/CD)**

Trunk-based development enables teams to ship new changes to production much faster and is a prerequisite for Continuous Integration and Continuous Delivery (CI/CD). This is another major benefit of the trunk-based flow. In fact, it’s often the reason businesses decide to adopt the strategy.

Teams that shrink the time between writing and shipping code end up building more features that customers love and shipping fewer bugs. The benefits of accelerating your code release cadence are significant, and trunk-based development is a key part of that acceleration.

Teams that are mired in feature-branch development inevitably have long, slow deployment cycles. Teams that embrace branching methodologies see their velocity skyrocket in ways that GitFlow teams can’t even imagine.

Feature management platform to engage in trunk-based development, software company Coder increased developer velocity by 15%, saved thousands of dollars a month, and shipped new changes to production in minutes, rather than days or weeks.

**What do you need for trunk-based development?**

**An extensive automated testing scheme**

Managers who come from a GitFlow background cringe at the description of trunk-based development. The idea of having no separation between the code you deploy to production and the code that’s still in active development is terrifying, because they don’t have confidence in the code their developers write.

That’s not a personal failing. Every developer ships bugs, and managers don’t want bugs to make it to customers. But it might be a failing of their process. The benefits of writing software tests for code have been understood for decades. Yet, teams persist in not writing tests, or writing ineffective tests. They participate in a false economy, believing that they “don’t have time” to write effective tests. Yet, as we’ve shown, adopting a trunk-based strategy has enormous implications for velocity, and means that teams ship fewer bugs.

So, this is one of the first steps that teams switching from feature branching strategies to trunk-based approaches have to tackle. They need to design and implement high-quality software tests. When you’re using continuous integration to ship code to your production environment multiple times per day, you don’t have the bandwidth for a full regression test before every deployment. You need tests to identify code that breaks before it gets into trunk.

**Feature flags**

Feature flags (a more advanced version of feature toggles) are another non-negotiable requirement for trunk-based development. Think back to our description of the trunk-based flow. Imagine that instead of a bug we were fixing, we were adding some small part of a new feature. It’s common for new feature implementations to be spread out over multiple tickets.

And if you’re using trunk-based development, spreading new features out over multiple tickets is something you’re going to do more of. Again, the goal is to keep your branches short-lived, which means smaller individual units of work. You want to make small changes, so that you can push code quickly.

If you’re working on an in-progress feature and you finish part of the code, you need a mechanism to tell the application to ignore that code in production. After all, your changes are going to deploy rapidly. You won’t have time to stitch the whole feature together before this code gets in front of customers.

That’s where feature flagging comes in. With feature flags, you hide code for those new features behind some logic gates. Feature flags allow developers to deploy incomplete features to production without releasing them to end-users.

Such capabilities enable trunk-based development by reducing the risk of merging incomplete code changes.

It’s worth noting that, if you wrap every new feature in a feature flag, then flags can quickly start filling up your codebase, creating technical debt. That’s why it’s critical to use a feature management platform. team makes it easy to create, see, and manage all the feature flags in your codebase.

while developers work on a new feature, you keep the corresponding feature flag ‘off’ and, in team easy-to-use dashboard, you can toggle it ‘on’ when you’re ready. What’s more, you can easily manage feature flags in different environments. As the new feature coalesces in Dev or QA environments, you can flip the flag on there to see how the new code behaves.

Once you’ve verified that it all works, you enable the feature flag in your production environment. Or, if you’re unsure how much value the new code brings, you can enable it for just segments of your customer base over time. This combines the benefits of quick rollouts that come with trunk-based development with the steadiness and surety that comes from a more conservative GitFlow system.

**Trunk-based development enables DevOps and unlocks team potential**

Remember that one of the key tenets of trunk-based development (and the DevOps workflow that inspired it) is that of incremental improvement.

One of the ways you can do that is by exploring feature flagging for your application, no matter your workflow. You can enable your team to engage in trunk-based development and dramatically increase their productivity.

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