Trunk-based development

Trunk-based development is a version control management practice where developers merge small, frequent updates to a core “trunk” or main branch. Since it streamlines merging and integration phases, it helps achieve CI/CD and increases software delivery and organizational performance.

In the early days of software development, programmers didn’t have the luxury of modern version control systems. Rather, they developed two versions of their software concurrently as a means of tracking changes and reversing them if necessary. Over time, this process proved to be labor-intensive, costly, and inefficient.

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.

What is trunk-based development?

Trunk-based development is a version control management practice where developers merge small, frequent updates to a core “trunk” or main branch. It’s a common practice among DevOps teams and part of the DevOps lifecycle since it streamlines merging and integration phases. In fact, trunk-based development is a required practice of CI/CD. Developers can create short-lived branches with a few small commits compared to other long-lived feature branching strategies. As codebase complexity and team size grow, trunk-based development helps keep production releases flowing.

Gitflow vs. trunk-based development

Gitflow is an alternative Git branching model that uses long-lived feature branches and multiple primary branches. Gitflow has more, longer-lived branches and larger commits than trunk-based development. Under this model, developers create a feature branch and delay merging it to the

main trunk branch until the feature is complete. These long-lived feature branches require more collaboration to merge as they have a higher risk of deviating from the trunk branch and introducing conflicting updates.

See solution

Build and operate software with Open DevOps

Learn more

Related material

How to get to Continuous Integration

Read more

Gitflow also has separate primary branch lines for development, hotfixes, features, and releases. There are different strategies for merging commits between these branches. Since there are more branches to juggle and manage, there is often more complexity that requires additional planning sessions and review from the team.

Trunk-based development is far more simplified since it focuses on the main branch as the source of fixes and releases. In trunk-based development the main branch is assumed to always be stable, without issues, and ready to deploy.

Benefits of trunk-based development

Trunk-based development is a required practice for continuous integration. If build and test processes are automated but developers work on isolated, lengthy feature branches that are infrequently integrated into a shared branch, continuous integration is not living up to its potential.

Trunk-based development eases the friction of code integration. When developers finish new work, they must merge the new code into the main branch. Yet they should not merge changes to the truck until they have verified that they can build successfully. During this phase, conflicts may arise if modifications have been made since the new work began. In particular, these conflicts are increasingly complex as development teams grow and the code base scales. This happens when developers create separate branches that deviate from the source branch and other developers are simultaneously merging overlapping code. Luckily, the trunk-based development model reduces these conflicts.

Allows continuous code integration

In the trunk-based development model, there is a repository with a steady stream of commits flowing into the main branch. Adding an automated test suite and code coverage monitoring for this stream of commits enables continuous integration. When new code is merged into the trunk, automated integration and code coverage tests run to validate the code quality.

Ensures continuous code review

The rapid, small commits of trunk-based development make code review a more efficient process. With small branches, developers can quickly see and review small changes. This is far easier compared to a long-lived feature branch where a reviewer reads pages of code or manually inspects a large surface area of code changes.

Enables consecutive production code releases

Teams should make frequent, daily merges to the main branch. Trunk-based development strives to keep the trunk branch “green”, meaning it's ready to deploy at any commit. Automated tests, code converge, and code reviews provides a trunk-based development project with the assurances it’s ready to deploy to production at any time. This gives team agility to frequently deploy to production and set further goals of daily production releases.

Trunk-based development and CI/CD

As CI/CD grew in popularity, branching models were refined and optimized, leading to the rise of trunk-based development. Now, trunk-based development is a requirement of continuous integration. With continuous integration, developers perform trunk-based development in conjunction with automated tests that run after each committee to a trunk. This ensures the project works at all times.

Trunk-based development best practices

Trunk-based development ensures teams release code quickly and consistently. The following is a list of exercises and practices that will help refine your team's cadence and develop an optimized release schedule.

Develop in small batches

Trunk-based development follows a quick rhythm to deliver code to production. If trunk-based development was like music it would be a rapid staccato -- short, succinct notes in rapid succession, with the repository commits being the notes. Keeping commits and branches small allows for a more rapid tempo of merges and deployments.

Small changes of a couple of commits or modification of a few lines of code minimize cognitive overhead. It's much easier for teams to have meaningful conversations and make quick decisions when reviewing a limited area of code versus a sprawling set of changes.

Feature flags

Feature flags nicely complement trunk-based development by enabling developers to wrap new changes in an inactive code path and activate it at a later time. This allows developers to forgo creating a separate repository feature branch and instead commit new feature code directly to the main branch within a feature flag path.

Feature flags directly encourage small batch updates. Instead of creating a feature branch and waiting to build out the complete specification, developers can instead create a trunk commit that introduces the feature flag and pushes new trunk commits that build out the feature specification within the flag.

Implement comprehensive automated testing

Automated testing is necessary for any modern software project intending to achieve CI/CD. There are multiple types of automated tests that run at different stages of the release pipeline. Short running unit and integration tests are executed during development and upon code merge. Longer running, full stack, end-to-end tests are run in later pipeline phases against a full staging or production environment.

Automated tests help trunk-based development by maintaining a small batch rhythm as developers merge new commits. The automated test suite reviews the code for any issues and automatically approves or denies it. This helps developers rapidly create commits and run them through automated tests to see if they introduce any new issues.

Perform asynchronous code reviews

In trunk-based development, code review should be performed immediately and not put into an asynchronous system for later review. Automated tests provide a layer of preemptive code review. When developers are ready to review a team member's pull request, they can first check that the automated tests passed and the code coverage has increased. This gives the reviewer immediate reassurance that the new code meets certain specifications. The reviewer can then focus on optimizations.

Have three or fewer active branches in the application's code repository

Once a branch merges, it is best practice to delete it. A repository with a large amount of active branches has some unfortunate side effects. While it can be beneficial for teams to see what work is in progress by examining active branches, this benefit is lost if there are stale and inactive branches still around. Some developers use Git user interfaces that may become unwieldy to work with when loading a large number of remote branches.

Merge branches to the trunk at least once a day

High-performing, trunk-based development teams should close out and merge any open and merge-ready branches at least on a daily basis. This exercise helps keep rhythm and sets a cadence for release tracking. The team can then tag the main trunk at the end of day as a release commit, which has the helpful side effect of generating a daily agile release increment.

Reduced number of code freezes and integration phases

Agile CI/CD teams shouldn't need planned code freezes or pauses for integration phases -- although an organization may need them for other reasons. The "continuous" in CI/CD implies that updates are constantly flowing. Trunk-based development teams should try to avoid blocking code freezes and plan accordingly to ensure the release pipeline is not stalled.

Build fast and execute immediately

In order to maintain a quick release cadence, build and test execution times should be optimized. CI/CD build tools should use caching layers where appropriate to avoid expensive computations for static. Tests should be optimized to use appropriate stubs for third-party services.