Plan your migration to Git

11/05/2018 • 9 minutes to read

By: Matt Cooper

Moving your team from a centralized version control system to Git requires more than just learning new commands. To support distributed development, Git stores information about file history and branches in a way that is fundamentally different from a centralized version control system. A successful migration requires that you understand these differences and plan your migration accordingly.

The Azure DevOps team has helped numerous customers, including teams within Microsoft, migrate from centralized version control systems to Git. The Azure DevOps team has developed this guidance based on their experience.

We recommend that you:

- Evaluate the tools and processes you're using
- Select a branching strategy for Git
- Decide how to migrate history – or if you even want to
- Maintain your old version control system
- Remove binaries and executables from source control
- Train your team in the concepts and practice of Git
- Perform the actual migration to Git

Evaluate your tools and processes

Changing version control systems will naturally disrupt your development workflow as developers begin using new tools and practices. This disruption can be an opportunity to improve other aspects of the development process.

You may wish to consider:

- Build and Test
  When do you perform builds, and when do you run tests? Adopting continuous integration, so that every check-in performs a build and a test pass, will help you identify defects early and provides a strong safety net for your project.
- Code reviews
  Is your team performing regular code reviews? Are code reviews required, and are they happening before the code is checked in? Git's branching model makes a pull request-based code review workflow a natural part of your development process, and pull requests complement a continuous integration workflow nicely.
- Release Management
  Are you performing continuous delivery? Moving to different version control tools will require you to make changes to your deployment processes, so a migration is a good time to adopt a modern release pipeline and automate your deployment processes.

Select a branching strategy

Before migrating your code you should select a branching strategy. Using long-lived, isolated feature branches is discouraged; this tends to delay merges until integration becomes very challenging. By using modern continuous delivery techniques like "feature flags", you can integrate code into the main branch quickly, but still keep in-progress features hidden from users until they're complete.

Short-lived "topic" branches allow developers to work close to the main branch and integrate quickly, avoiding merge problems. Two common topic branch strategies are GitFlow and a simpler variation, GitHub Flow, which you can adopt or look to for inspiration.

If you're currently using a long-lived feature branch strategy, it may be easiest to begin adopting feature flags before migrating to Git, so that you need to migrate as few branches as possible. Be sure to document the mapping between legacy branches and the new branches in Git so that your team understands where they should commit their new work.

Migrating history

You may be tempted to migrate your existing source code's history to Git. There are numerous tools that claim to migrate a complete history of all branches from a centralized tool to Git, and at first glance this may seem like a good solution. A Git commit appears to map relatively well to the changeset or check-in model that your previous version control tool used, but there are some serious limitations with this translation:

- In most centralized version control systems, all branches exist as folders inside the repository. For example: the main branch may be a folder named `/trunk` while other branches exist as folders like `/branch/one` and `/branch/two`. In a Git repository, branches instead apply to the entire repository and a 1:1 translation may be difficult.
- In some version control systems, a "tag" or a "label" is a collection that can contain various files in the tree, perhaps even files at different versions. In Git, a tag is a snapshot of the entire repository at a specific point in time and it cannot represent a subset of the repository or combine files at different versions.
- Most version control systems store details about the way files change between versions, recording fine-grained change types like rename, undelete and rollback. Git stores versions as a snapshot of the entire repository, and the metadata about how files change is not available.

These differences mean that a full history migration will be lossy, at best, and possibly even misleading. Given this lossiness, the effort involved in performing a migration with history and the relative rarity that this history is used, we suggest that you avoid importing history and instead perform a "tip migration", bringing only a snapshot of the most recent version of a branch into Git.

For most development teams, the time spent trying to migrate history is typically better spent on other areas of the migration that have a higher return on investment, especially improving processes. The Azure DevOps team has observed this while migrating numerous customers from a centralized system to Git, including many customers at Microsoft. In almost all cases, we recommend the tip migration.

Maintaining your old version control system

During and after a migration, developers will still need access to the history in your old version control system, so you can't simply shut the old system down once you've migrated. This is especially true if you've only done a tip migration: although the old version control history becomes less relevant over time, it is still important to be able to refer back to it, and highly regulated environments may have specific legal and auditing requirements about version control history. Instead, set the old version control system read-only once you have performed a migration.

For large development teams and regulated environments, we recommend placing "breadcrumbs" in Git that point users back to the old version control system. A simple example is a text file at the root of a Git repository, added as the first Git commit, before the tip migration, pointing to the URL of the old version control server. If you migrate many branches, a text file in each should explain how the Git branches were migrated from the old system.

Adding breadcrumbs is especially helpful for developers who start working on a project long after it's been converted to Git and who don't have familiarity with the old version control system.

Binary files and tools

Due to the way Git stores history, you should avoid adding binary files to a repository, especially binaries that are very large or that change regularly. Git's storage model is optimized for versioning text files like source code, which are compact and highly compressible. Binary files typically are neither, and once they've been added to a repository, they will remain in the repository history and in every future clone.

Migrating to Git provides an opportunity to remove these binaries from the your codebase. We recommend removing checked-in libraries, tools and build output from your repository and instead using a package management system like NuGet to manage dependencies.

Assets like icons and artwork may need to align with a specific version of your source code. Small, infrequently-changed assets like icons can be included directly in a repository; since they do not change often, they will not bloat the history. But large or frequently changing assets should be stored using the Git LFS (Large File Storage) extension. Learn more about managing large files with Git.

Training

Perhaps the biggest challenge in migrating to Git is helping developers understand how Git stores changes and how commits form a history of development. It's not enough to just prepare a "cheat sheet" that maps commands in the old system to Git commands: your developers need to stop thinking about version control history in terms of a centralized, linear model and need to understand Git's history model and the commit graph. Since people learn in different ways, plan on making several types of training material available. Live, lab-based training with an expert instructor works well for some people. The Pro Git book is available for free online and is an excellent starting point. Microsoft also offers a Git walkthrough designed to rapidly get someone up to speed.

Identify key members of the team as Git experts, empower them to help others, and make sure that the rest of the team is encouraged to ask them questions.

Migrating code

Once you've updated your processes, analyzed your code and started training your team, you can finally do a migration of the source code. Whether you do a tip migration or are trying to migrate history, we recommend that you do one or more test migrations into a test repository. Before you do the real migration, you'll want to make sure:

- All code files have migrated and there are no stray binaries in the repository
- You can push the repository successfully to Azure DevOps
- Users have the appropriate permissions to fetch and push
- All branches are available
- Builds are successful, and all your tests are passing

We recommend you do the final migration at a time when few people are working, ideally between milestones when there is some natural "downtime" instead of at the end of a sprint when developers are rushing to finish work. Aim to migrate over a weekend when nobody needs to check-in.

Plan to make a firm cutover from your old version control system to Git; trying to keep multiple systems operating in parallel is confusing since developers may not know how, or where, to check-in and there is no "single source of truth". Set the old version control system read-only to avoid this, otherwise you may have to try to do a *second* migration to include any changes that were made in the old system during the migration.

The actual process you take will vary based on the system you're migrating from. We have a detailed article about migrating from Team Foundation Version Control, and guidance for migrating from other systems as well.

Migration checklist

Area	Task
Team workflows	Determine how builds will run
	Determine when tests will run
	Develop a release management process
	Move your code reviews to pull requests
Branching strategy	Pick a Git branching strategy
	Document the branching strategy, including why it was selected and how legacy branches map
History	Decide how long to keep legacy VC running
	Identify branches which need to migrate
	If needed, create "breadcrumbs" to help engineers navigate back to the legacy system
Binaries and tools	Identify which binaries and undiffable files to remove from the repo
	Decide on an approach for large files, such as Git-LFS
	Decide on an approach for delivering tools and libraries, such as NuGet
Training	Identify training materials
	Plan training: events, written material, videos, etc.
	Identify members of the team to serve as local Git experts
Code migration	Run multiple test runs to ensure the migration will go smoothly
	Identify and communicate a time to make the cutover
	Create the new Git repo on Azure DevOps
	Migrate the mainline branch first, followed by any additional branches needed

Learn more about migrating to Azure DevOps from Team Foundation Server and get an overview of how your TFVC commands and workflow map to Git.

Get started with unlimited free private Git repos in Azure Repos.

Matt Cooper is currently a program manager for Azure DevOps, focused on Azure Pipelines. Previously he's worked on package management, Xbox, and Dynamics CRM.