# How We Use Git at Microsoft

02/01/2018 • 8 minutes to read

**In this article**

By: Matt Cooper

## How We Use Git

We've talked a lot about hosting the world's largest Git repository, about how we're moving Microsoft to Git, and about the challenges of Git at scale. We often get asked, "how does Microsoft actually use Git?" and, while we've given a little guidance based on our learnings, we'd like to be more thorough. This will be a whirlwind walkthrough of the standard Git workflow at Microsoft. There are certainly teams which diverge from these principles in big and small ways based on their needs, but by and large, these are the default practices for most teams.

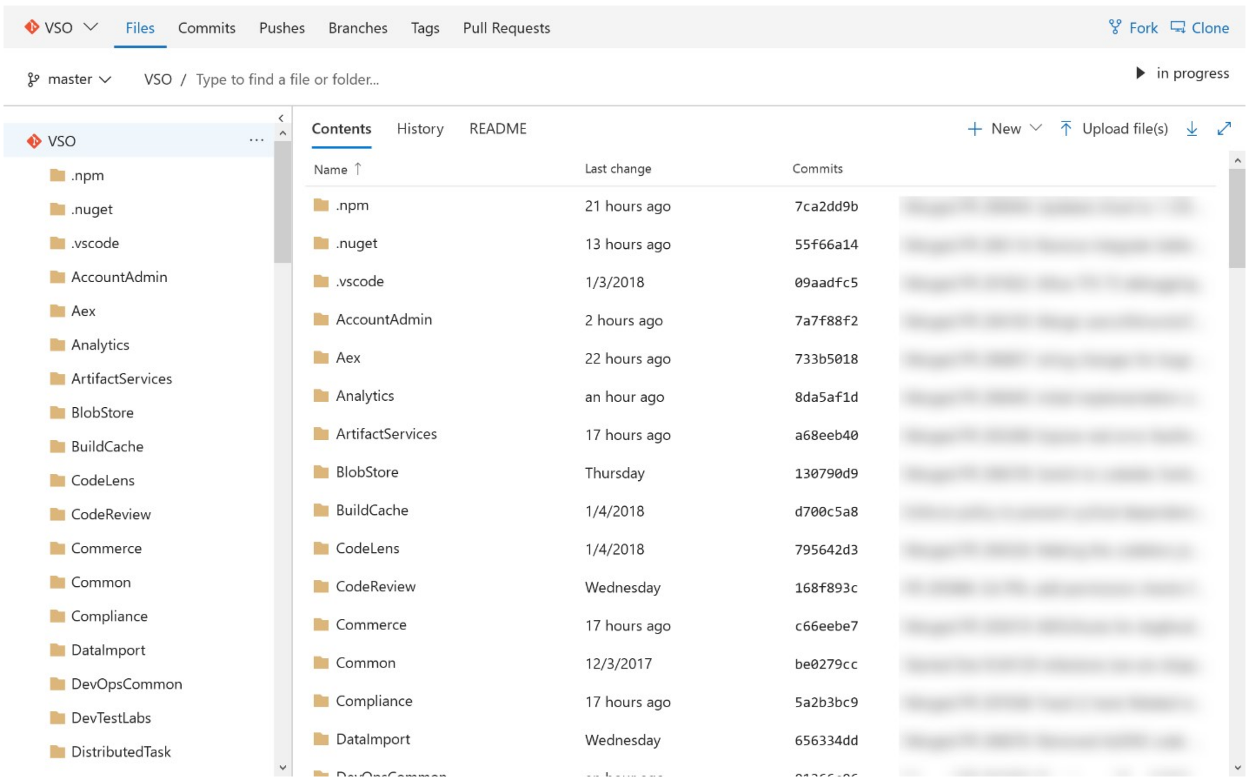
## The Team and Product Hosting Git

Other articles and videos on this site go into more detail about the team and product makeup. As a refresher, there are around 500 engineers in our organization. We're geographically distributed into 3 main sites (Redmond, Raleigh, and Hyderabad) plus dozens of remote workers in cities across the world. We ship multiple products from our codebase: Team Foundation Server comes out yearly and has several point updates per year. Azure DevOps ships every three weeks, plus a daily hotfix train. These two products, though separate, share about 90% of their code. In addition, we have several components built from our repository which ship on a different schedule, such as Team Explorer for Visual Studio. Everything we do with our code structure, branch structure, and workflows is to support a large distributed team pumping out complex software.

## Git Repository Structure

The majority of our code is in one Git repository hosted by Azure DevOps. That repository contains about 8GB worth of history on a fresh clone. We follow a single-trunk model; more on our branch structure later.

Code is broken up into components, which each live in their own root-level folder. Really large components, especially some of the older components, may be made up of multiple subcomponents. Those subcomponents get separate sub-folders within the parent component.



| Name ↑ | Last change | Commits |
|---|---|---|
| .npm | 21 hours ago | 7ca2dd9b |
| .nuget | 13 hours ago | 55f66a14 |
| .vscode | 1/3/2018 | 09aadfc5 |
| AccountAdmin | 2 hours ago | 7a7f88f2 |
| Aex | 22 hours ago | 733b5018 |
| Analytics | an hour ago | 8da5af1d |
| ArtifactServices | 17 hours ago | a68eeb40 |
| BlobStore | Thursday | 130790d9 |
| BuildCache | 1/4/2018 | d700c5a8 |
| CodeLens | 1/4/2018 | 795642d3 |
| CodeReview | Wednesday | 168f893c |
| Commerce | 17 hours ago | c66eebe7 |
| Common | 12/3/2017 | be0279cc |
| Compliance | 17 hours ago | 5a2b3bc9 |
| DataImport | Wednesday | 656334dd |

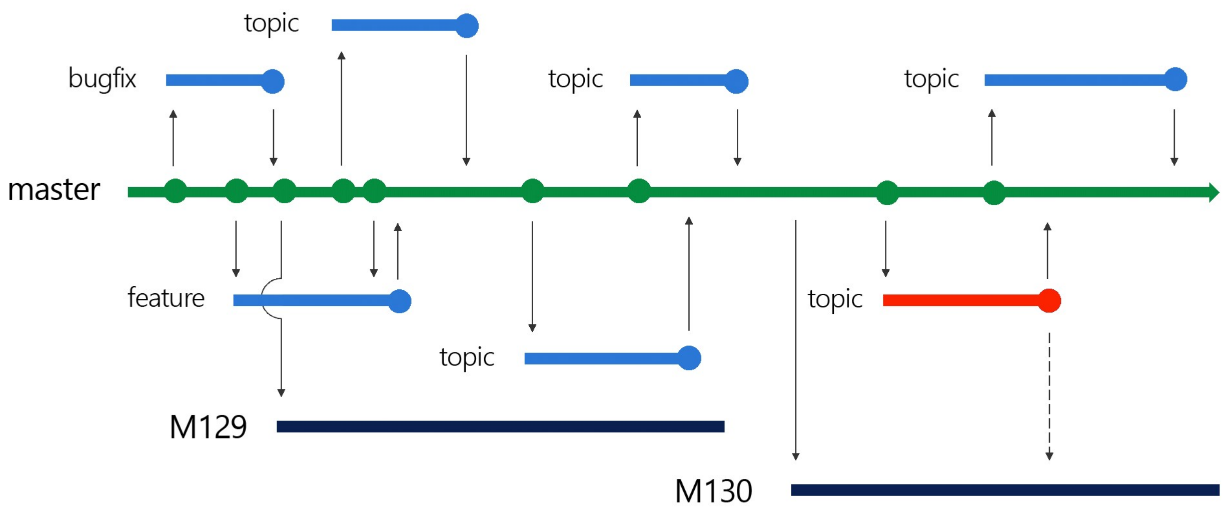
We have a few adjunct repositories, as well. For instance, our build & release agent and tasks, VS Code extension, and more are developed in the open on GitHub. Configuration changes on Visual Studio Team Services are checked into a separate repository. A handful of other packages we depend on come from other places, and we consume them via NuGet.

## Mono Repo or Multi-Repo with Git

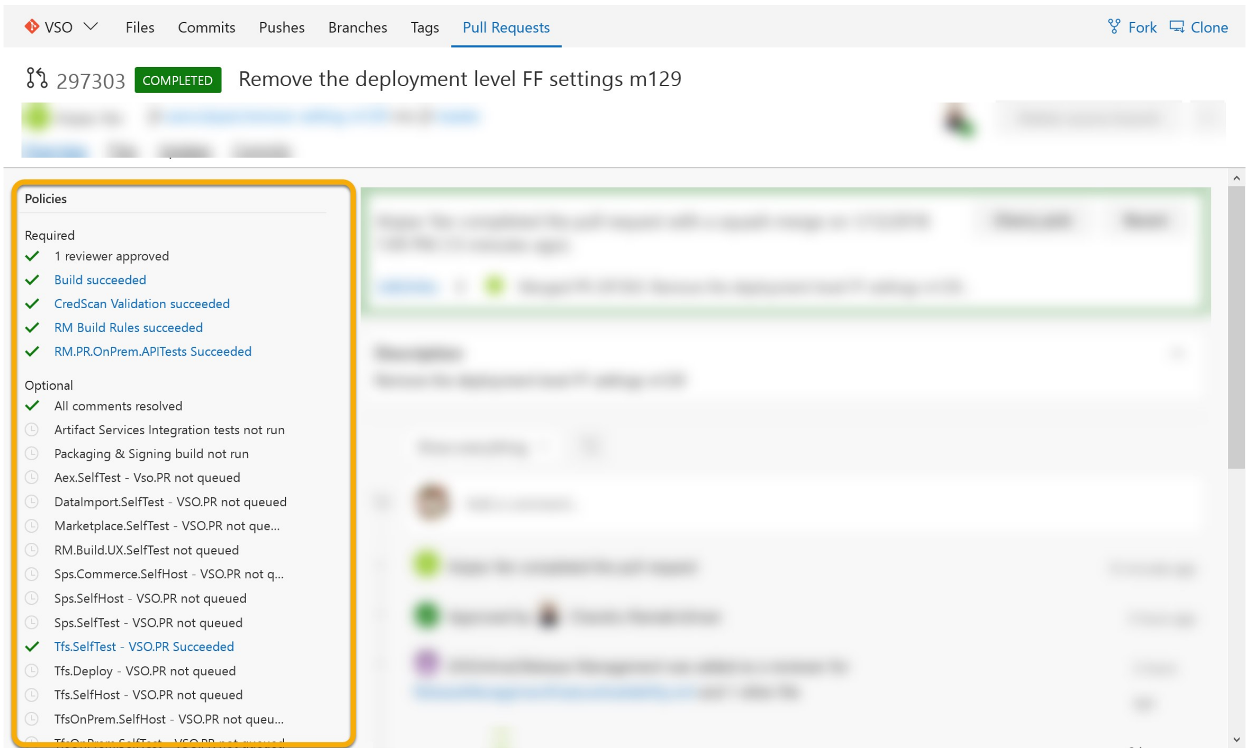
It's worth pausing to reflect that this isn't the only way. While we've elected to have a single monolithic repository (the "mono-repo"), other products at Microsoft use a multi-repo approach. Skype, for instance, has hundreds of small repositories that get stitched together in various combinations to create their many different clients, services, and tools. Especially for teams embracing microservices, multi-repo can be the right approach. TFS/Azure DevOps, like many products at Microsoft, began as a monolith, and its code organization reflects that.

## Git Branch Structure and Policies

We use a branching strategy that we call "Release Flow". It's a trunk-based development model, similar to the ones that we recommend to our customers in our branch strategy guide. Release Flow lets us keep master buildable at all times (more on that later) and work from short-lived topic branches. When we're ready to ship, whether that's a sprint or a major TFS update, we start a new release branch off master. Release branches never merge back to master, so we require cherry-picking important changes. In the diagram below, short-lived branches are shown in light blue and the release branches are shown in dark blue. One branch with a commit that needs cherry-picking is shown in red.



We use a couple of Azure DevOps features to help enforce this structure and keep master clean. Branch policies prevent direct pushes to master. We require a successful build (including passing tests), signoff by the owners of any code that was touched, and a handful of external checks verifying corporate policies before a PR can be completed.



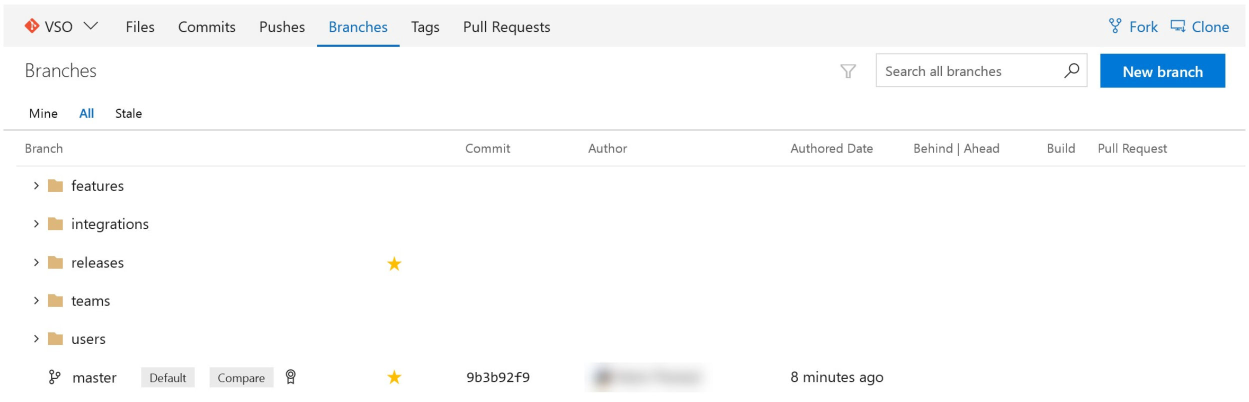
297303 COMPLETED Remove the deployment level FF settings m129

Policies

Required
- ✓ 1 reviewer approved
- ✓ Build succeeded
- ✓ CredScan Validation succeeded
- ✓ RM Build Rules succeeded
- ✓ RM.PR.OnPrem.APITests Succeeded

Optional
- ✓ All comments resolved
- Artifact Services Integration tests not run
- Packaging & Signing build not run
- Aex.SelfTest - Vso.PR not queued
- DataImport.SelfTest - VSO.PR not queued
- Marketplace.SelfTest - VSO.PR not que...
- RM.Build.UX.SelfTest not queued
- Sps.Commerce.SelfHost - VSO.PR not q...
- Sps.SelfHost - VSO.PR not queued
- Sps.SelfTest - VSO.PR not queued
- ✓ Tfs.SelfTest - VSO.PR Succeeded
- Tfs.Deploy - VSO.PR not queued
- Tfs.SelfHost - VSO.PR not queued
- TfsOnPrem.SelfHost - VSO.PR not queu...

We also like to keep our branch hierarchy tidy. We use permissions to block creation of branches at the root level of the hierarchy. Everyone can create branches in folders like users/, features/, and teams/. Only release managers have permission to create branches under releases/, and some automation tools have permission to the integrations/ folder.



| Branch | Commit | Author | Authored Date | Behind \| Ahead | Build | Pull Request |
|---|---|---|---|---|---|---|
| features | | | | | | |
| integrations | | | | | | |
| releases ★ | | | | | | |
| teams | | | | | | |
| users | | | | | | |
| master Default Compare ★ | 9b3b92f9 | [redacted] | 8 minutes ago | | | |

## Working in the Git Repository

Within this structure, how do engineers actually get their daily work done? Obviously the environment's going to vary heavily by team and by individual — some people like the command line, others like Visual Studio, and others work on different platforms. But the structures and policies in place on our repository ensure a solid and consistent foundation. Let's walk through a handful of common tasks.

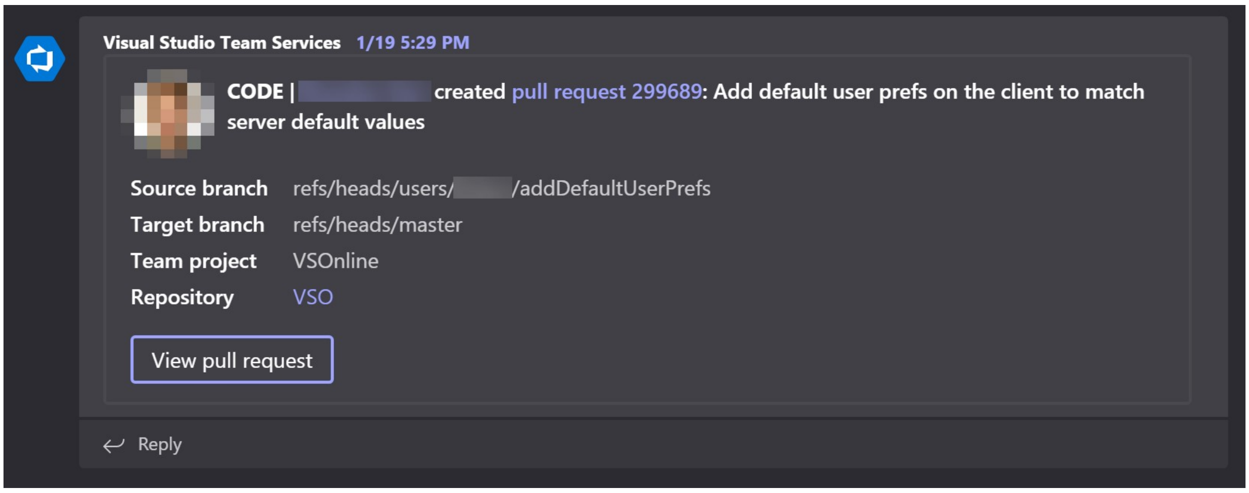
### Git Workflow to Build a New Feature

Our first stop has to be building a new feature. That's the meat of a software engineer's job, right? We'll skip past the non-Git parts like looking at telemetry data, coming up with a design and a spec, and even writing the actual code. Let's jump right in to working with the repository.

First, the engineer syncs to the latest commit on master. We keep master always buildable, so this is virtually guaranteed to be a good starting point. The developer checks out a new feature branch, makes code changes, commits, and pushes to the server. When our engineer starts a pull request, several interesting things happen.

### Using Git Branch Policy

First, automated systems start checking that the new code builds, hasn't broken anything, and hasn't violated any policies (security, compliance, and so on). This doesn't block other work from happening in parallel. Most teams have configured integration with Microsoft Teams, which announces the new PR to the engineer's colleagues. The owners of any code touched are automatically added as reviewers. We make liberal use of optional reviewers for code that many people touch, like REST client generation and shared controls, as a way to get expert eyes on those changes.



**Visual Studio Team Services** 1/19 5:29 PM

**CODE | [redacted] created pull request 299689: Add default user prefs on the client to match server default values**

| | |
|---|---|
| **Source branch** | refs/heads/users/[redacted]/addDefaultUserPrefs |
| **Target branch** | refs/heads/master |
| **Team project** | VSOnline |
| **Repository** | VSO |

Once the people and the automation are satisfied, our engineer completes the pull request. If there's a merge conflict, the engineer is given instructions on how to sync to the conflict, fix it, and re-push the changes. The automation all runs again on the fixed code, but humans don't have to sign off again. Soon, we'll be rolling out an extension built by our friends in Windows which allows conflict resolution right in the browser. That's pretty amazing — by using an extensibility point, the Windows team built something cool enough that we're turning around and including in our own workflow.

Azure DevOps merges the code to master, and it'll deploy in the next sprint or TFS release. Importantly, that doesn't mean the new feature will show up right away. We've talked elsewhere about how we decouple deployment and exposure of new features using feature flags. This means even if the feature needs a little more bake time before it's ready to show off, if the product builds and deploys, it's safe to go to master. Once in master, the code ends up in an official build, where it's (again) tested, confirmed to meet policy, and digitally signed.

### Shipping

Deployment is a complex topic which we've covered elsewhere on the site. The branch model is the main intersection between deployment and Git. At the end of each sprint, one of our release managers sends out an email reminding people of the cut-off time. At the appointed time, the release manager creates a new release branch off a particular commit, then lets everyone know which commits are in and which ones missed the train.

If an engineer misses the cutoff, he or she can (with permission) cherry-pick the changes into the release branch. These cherry-picks go through a PR to land in the release branch.

### Feature Flags Allow Code to Be Deployed with Controlled Exposure

As we deploy out through our rings and stages, controlling exposure with feature flags, maybe we discover an issue in production. Despite all our automation and review, things happen. There's no place like production! Usually, we're alerted by health monitoring and telemetry when something isn't right. A developer can — you guessed it — create a branch off master, make a fix, and PR it into master. Keeping the same general workflow means that developers don't have to context-switch or learn a different process for a (hopefully rare) different code change.

One additional step is required: to cherry-pick the change into the release branch. We run a hotfix deployment out of the current release branch each weekday morning, though we can also do this on demand for urgent fixes. The fix actually hits production out of the release branch first. But because we develop in master first, we know it won't regress the next sprint when a new release branch is created from master.

Releases of the on-premises TFS are largely the same, though without the deployment rings and stages. Also, because we do more manual testing on different configurations and data shapes, there's a longer tail between cutting the release branch and putting the product in the hands of customers.

## Git Facilitates our Shift Left

Working this way with Git gives us a number of benefits. First, we work out of a single master, virtually eliminating merge debt. Second, the pull request flow gives us a common point to force testing, code review, and error detection early in the pipeline. This helps us shorten the feedback cycle to developers — errors are usually detected in minutes, not hours or days. Also, it gives us confidence when we refactor, since all changes are tested all the time.

Currently, we have more than 200 pull requests and 300 continuous integration builds per day. Together, that amounts to 500 test runs every 24 hours, a level that would have been a fantasy without this workflow.

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.