Release Flow

07/01/2018 • 8 minutes to read

In this article

How We Deliver Changes to Production
Development
Releases at Sprint Milestones
Releasing Hotfixes
Moving On
In Conclusion

By: Edward Thomson

How We Deliver Changes to Production

Microsoft is actively pursuing a strategy of using "one engineering system" throughout the company: a modern system to build all our products based on Git and Azure DevOps. And one of the questions that we're often asked is how we use version control and branching to deliver changes safely to production.

The Azure DevOps team uses a trunk-based branching strategy to help us develop Azure DevOps quickly and deploy it regularly. This strategy needs to be able to scale to our development needs: a single repository that contains the entire Azure DevOps product, hundreds of developers split across three main offices, and deployment in multiple Azure data centers around the world.

We call it "Release Flow", and it encompasses the entire DevOps process: from development to release.

Development

1. **Branch**
   The first step when a developer wants to fix a bug or implement a feature is to create a new branch off of our main integration branch, `master`. Thanks to Git's lightweight branching model, we create these short-lived "topic" branches any and every time we want to write some code. Developers are encouraged to commit early and to avoid long-running feature branches by using feature flags.

2. **Push**
   When the developer is ready to get their changes integrated and ship their changes to the rest of the team, they push their local branch to a branch on the server, and open a pull request. Since we have several hundred developers working in our repository, each with many branches, we use a naming convention for branches on the server to help alleviate confusion and what we call "branch proliferation". Generally developers create a local branch named `users/<username>/feature`, where `<username>` is (of course) replaced with their account name. For example, I create branches inside the `users/ethomson` folder.

3. **Pull Request**
   We use Azure DevOps Pull Requests to control how developers topic branches are merged into master. Pull Requests ensure that our branch policies are satisfied: first, we build the proposed changes and run a quick test pass. We run about 60,000 tests — our "level 0" and "level 1" test suites — in just under five minutes. This isn't our complete test matrix, but it's enough to quickly give us a high confidence in pull request.

   Next, we require that other members of the Azure DevOps team review the code and approve the changes. Code review picks up where the automated tests left off, and are particularly good at spotting architectural problems. Manual code reviews ensure that more engineers on the team have visibility into the changes and that code quality remains high.

4. **Merge**
   Once all the build policies are satisfied and reviewers have signed off, then the pull request is completed. This means that the topic branch is merged into the main integration branch, `master`.

   After merge, we run additional acceptance tests that take more time to run. These are more like a traditional post-checkin tests and we use them to perform an even more thorough validation. This gives us a good balance between having fast tests during the pull request review yet still having complete test coverage before release.

This is how developers get changes into our code base -- and at this point, our branching strategy looks like a typical trunk-based development model. But unlike some other trunk-based strategies, like GitHub Flow, we do not deploy these changes to production to test them before merging the pull request, nor do we deploy to production when the pull request is merged.

An often overlooked part of GitHub Flow is that pull requests are actually delivered directly to production — to test them — *before* they're merged into master. This means that developers need to wait in the "deployment queue" to test their changes before they can merge their pull requests.

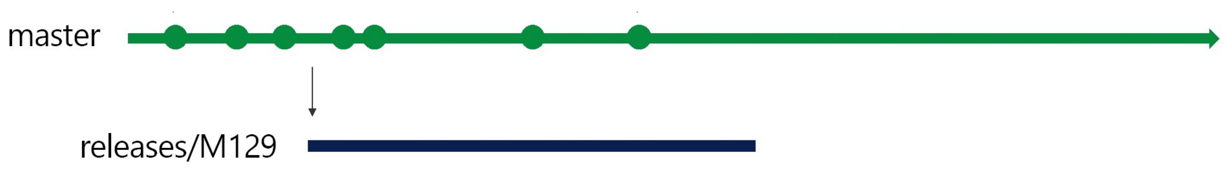
The Azure DevOps team has several hundred developers working constantly in our repository, and we complete over 200 pull requests into master per day. If each of those pull requests required a deployment to multiple Azure data centers across the globe, our developers would waste time waiting in the queue to deploy their branches instead of writing software.

Instead, we continue developing in our master branch and batch up deployments into three week blocks, aligned with our sprint cadence.

Releases at Sprint Milestones

At the end of a sprint, we create a deployment branch from the master branch: for example, at the end of sprint 129, we create a new branch `releases/M129`. We then put the sprint 129 branch into production.

Once we've branched to our deployment branch, the `master` branch remains open for developers to merge changes. These changes, of course, do not get deployed to production - they'll be deployed within the next three weeks, during the next sprint deployment.



Releasing Hotfixes

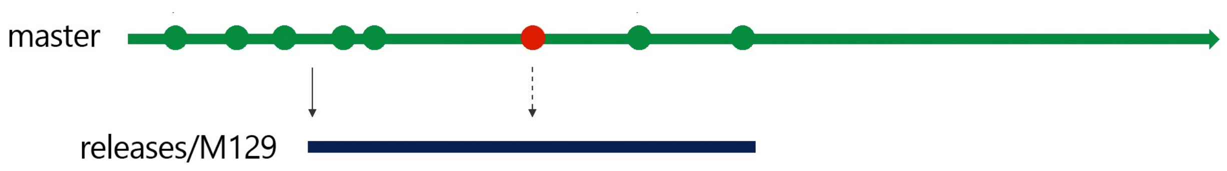
Obviously, some changes need to go to production more quickly. We generally won't add big new features in the middle of a sprint, but sometimes we want to bring a bug fix into Azure DevOps quickly to unblock users. Sometimes we have embarassing typos that we want to correct. And sometimes we have a bug that causes an availability issue, which we call a "live site incident".

When this happens, we start with our normal workflow: we create a branch *from* `master`, get it code reviewed, and complete the pull request to merge it. We always start by making the change in `master` first: this allows us to create the fix quickly, and validate it locally without having to switch to the release branch locally.

More importantly, by following this process, we're guaranteed that our change goes into `master`. This is critical for us: if we were to fix a bug in the release branch first, and accidentally forget to bring the change back to `master`, we would have a recurrence of the bug during the next deploy - when we create our sprint 130 release branch from `master` in three weeks.

It's particularly easy to forget to do this during the confusion and stress that can arise during an outage. So by always bringing our changes to `master` first, we know that we'll always have our changes in both the `master` branch and our release branch.

Azure DevOps actually has unique functionality to enable this workflow: from the Pull Request page, you can cherry-pick a pull request onto a different branch. To bring changes immediately into production, once we have merged the pull request into master, we cherry-pick the change into the release branch. This creates a new pull request that targets the release branch, backporting the contents that were just merged into `master` .



By opening a new pull request, we get traceability and reliability from branch policies. And using the Azure DevOps cherry-pick functionality allows us to do it quickly: we don't need to download the release branch to a local computer to cherry-pick the changes, it's all handled efficiently on the server. And if we need to make changes, to fix merge conflicts or make minor changes due to differences between the two branches, we can do *that* on the server, too. We can edit changes directly from the text editor built-in to Azure DevOps, or we can use the [Pull Request Merge Conflict Extension]() for a more advanced experience.

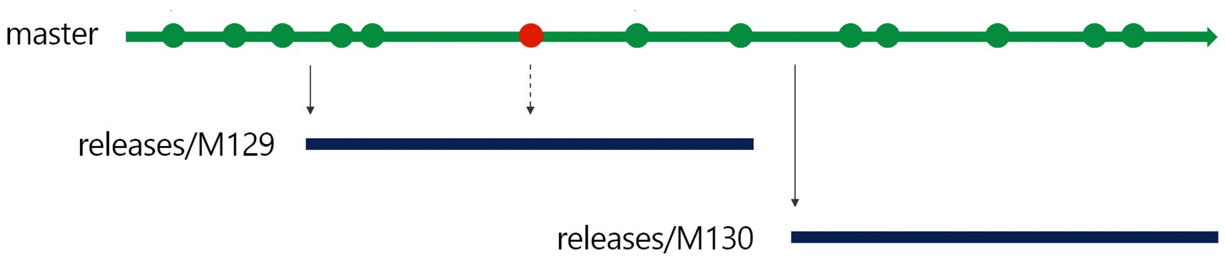
Once we have a pull request targeting our release branch, we'll code review it again, evaluate the branch policies, and test it. Once it's merged, it will get deployed to our first "ring" of servers in minutes. From there, we'll progressively deploy it to more Azure DevOps accounts using deployment rings. As more users are exposed to the changes, we'll monitor its success and ensure that our change has fixed the bug while not introducing any new deficiencies or slowdowns as the fix is deployed to the rest of our data centers.

Moving On

After three weeks, we'll finish adding features to sprint 130, and we'll be ready to deploy those changes. To deploy, we'll create the new release branch, `releases/M130` from master, and deploy that.

At this point, we'll actually have *two* branches in production: since we use a ring-based deployment to bring changes to production safely, our fast ring will get the sprint 130 changes, while our slow ring servers will stay on sprint 129 while the new changes are validated in production. (This raises an interesting problem: if we need to hotfix a change in the middle of a deployment, we may need to hotfix two different releases: the sprint 129 release *and* the sprint 130 release.)

Once all the rings have been deployed, our old branch from sprint 129 is completely abandoned. We'll never need it again, since we were very careful to ensure that any changes that we brought into the sprint 129 branch as a hotfix was also made in `master`. So those changes will also be in the `releases/M130` branch that we create.



In Conclusion

The Release Flow model is at the heart of the Azure DevOps team's development methodology. It allows us to use a simple, trunk-based branching strategy for our online service. But instead of keeping our developers stuck in a deployment queue, waiting to be able to merge their changes, our developers can keep working.

Release Flow lets us deploy new features across all our Azure data centers at a regular cadence, and despite the size of our codebase and the number of developers working in it, we can bring hotfixes into production quickly and efficiently.

Edward Thomson is the Git Community program manager for Azure DevOps. Previously, he was a software engineer working on version control at Microsoft, GitHub and SourceGear.