.NET Core invests in consistency to accelerate open-source innovation

Overview

The goal of the .NET Foundation is to foster innovation by harnessing the power of the open-source community. However, as the scope of projects under the foundation grew, so did the complexity faced by developers both inside and outside Microsoft, who had to learn different ways of doing things to work on different parts of the software stack. Faced with a broad range of tools, several different CI systems, and lots of confusion, the .NET Engineering Services team decided to invest in greater consistency. Shared tooling across repos and a single, shared CI system are just two examples of how the team is continually chipping away at barriers to developer productivity—so that all developers working on the project can realize their full potential as they work together to write, test, build, and ship high-quality software.

Table of Contents

Visit the Webpage

The Team

The .NET Foundation is an independent organization created to foster innovation through open development and collaboration. Its activities include promoting the broad spectrum of software available to .NET developers, advocating for the needs of .NET open source developers, evangelizing the benefits of the .NET platform, promoting the benefits of the open source model to developers already using .NET, and providing administrative support for a number of .NET open source projects.

Within Microsoft, the .NET Engineering Services team consists of some 20 people who do work for the .NET Foundation as contributors to the project. Excluding GitHub, they're responsible for most of the infrastructure that enables Microsoft developers to work alongside contributors from around the globe to develop, test, build, and ship .NET Core—an open source, cross platform successor to the Microsoft .NET Framework that runs on the Windows, Linux, and macOS operating systems.

The Challenge

The .NET Foundation supports many open source projects, one of which is .NET Core. The project is developed publicly on GitHub, which provides both the necessary scale and a means of 'going where the open source community lives.' It's composed of dozens of isolated git repositories, it targets several platforms instead of just Windows, and its components may ship in more than one 'vehicle'—for example, Roslyn, the C# compiler, ships as a component of Visual Studio and the SDK.

Like most open source projects, the goal of .NET Core—and the .NET Foundation as a whole—is to enlist the developer community at large to "come and help," as they've enthusiastically done from the very first day. Having .NET Core as an open source project also promotes a more transparent development process, as well as an active and engaged community.

But how do you put the infrastructure in place to enable Microsoft and external developers to work together, especially with something as large and complex as .NET? How do you make sure there isn't total chaos? What tools and processes can you put in place to ensure code quality and stability, and how do you scale them across so many projects? Just as important, how do you enable each and every one of those developers to do their best work—and contribute as much as they can?

These are just a few of the challenges the .NET Engineering Services team has worked to address since the project's inception. Like most projects, many initial infrastructure decisions were based on necessity and expediency, such as the decision to use Jenkins for GitHub pull request (PR) and continuous integration (CI) validation because it supported cross-platform, open-source development. However, as the project grew, so did the complexity faced by developers both inside and outside of Microsoft.

It makes it a lot harder for developers to move from one repo to another… and if it's this hard on Microsoft developers, how can we expect most individual contributors to figure it out?

MATT MITCHELL
Principal Software Engineer

A lot of this complexity was due to the isolated, distributed nature of the repo structure, which, as it expanded, adopted several different toolsets and CI systems. Yet these repositories are still dependent on each other to build a coherent product. The net result of all this? It was hard for developers who weren't intimately familiar with the entire software stack—including most individual contributors—to move between repos, such as might be needed to debug the root cause of a problem and propose a fix.

"By .NET Core 2.0, we had about 80 different repos for .NET Core and the related software stack, including ASP.NET Core," explains Matt Mitchell, Principal Software Engineer on the .NET Engineering Services team. "While many independent repos often makes development within a single repo easier, it makes it a lot harder for developers to move from one repo to another, upon which they may have to learn an entirely new way of doing things. And if it's this hard on Microsoft developers, how can we expect most individual contributors to figure it out? As .NET Core 3.0 planning began, it became clear that we could not create a release of the scope that we wanted without significant changes to our infrastructure."

To fully appreciate the magnitude of this challenge, it's worth taking a look at a few numbers. Under .NET Core, the Roslyn (C# & VB.NET compiler) repo has seen more than 18,000 PRs since the first one in 2014. And the CoreFx (libraries) repo has seen some 22,000 PRs—representing more than 800 contributors. In terms of CI, without which none of this would even be possible, the numbers are even more impressive. For Roslyn, more than 700,000 automated tests are run for one iteration of a PR. For the Core CLR (runtime), that number climbs to about 1.6 million. And for the CoreFx libraries, it increases to 7.5 million. "In the past week alone, CoreFx has seen 414 PRs," says Mitchell. "If you estimate three to four iterations for each PR, you're looking at a total of 9.3 to 12.4 billion CI tests. For CI after commit, the numbers climb much higher."

The Journey

Faced with a broad range of tools, several different CI systems, and lots of confusion, Mitchell's team decided to invest in greater consistency—including shared tooling across repos and a single, shared CI system.

For developers working across different parts of the stack, consistency in tooling is a big plus

MATT MITCHELL
Principal Software Engineer

Shared tooling across repos

Prior to .NET Core 3.0, there were three to five different tooling implementations scattered throughout various repos, depending on how you counted. While this allowed each team to customize its tooling and build only what it needed, it made it difficult for developers to efficiently move between repos. For example, what do developers type to build and test? Where are the logs placed? And how is a new project added? "Multiple tooling implementations was a major barrier, making it considerably more difficult for developers to move between different parts of the software stack and help out," says Mitchell. "For example, someone working on the CoreFx libraries would need to re-learn an entirely new way of doing things before they could work on the C# compiler."

For .NET Core 3.0, the team worked to bring all repos under a common directory structure, set of commands, and build and test logic. Striking the right balance between being too prescriptive and not being prescriptive enough, however, was a challenge. The team ended up taking a blended approach by defining a common set of commands and scripts, a common repo layout, and a common set of build targets. Repos that choose to adopt all this have predictable behavior, making it easy to roll out changes. Repos that don't wish to adopt it all can pick and choose from a variety of task packages that provide basic functionality, like signing and packaging, which tend to look the same across all repos.

"For developers working across different parts of the stack, consistency in tooling is a big plus," says Mitchell. "Common tooling also increases the efficiency with which we can deliver new tooling improvements. Today we only need to implement such tooling improvements once, enabling us to deliver further developer productivity gains across all repos all that much faster."

A single, shared CI system

Now that we longer need to worry about the operational aspects of CI, we're free to focus on further improvements that will be felt by all—including Microsoft developers and individual contributors alike

MATT MITCHELL
Principal Software Engineer

Up through the .NET Core 2.2 release, over time and for different considerations, a number of different CI systems were adopted. For example, ASP.NET Core chose AppVeyor and Travis for GitHub PRs, whereas official ASP.NET Core builds used TeamCity. The rest of .NET Core used Jenkins for GitHub PRs and rolling validation, and used 'classic' (pre-YAML) Azure Pipelines workflows for all official builds. "A lot of this diversity was simply from necessity," explains Mitchell. "At the time those decisions were made, there was no single CI system that met every repo's needs." All of this was confusing to developers, who had to learn different CI systems to move between repos—just like they did with all the different tooling implementations. And in some repos, all CI was done serially, on a single system, resulting in developers having to wait up to 2 or 3 hours between iterations on their code—and that's if the CI system was working well, which wasn't always the case.

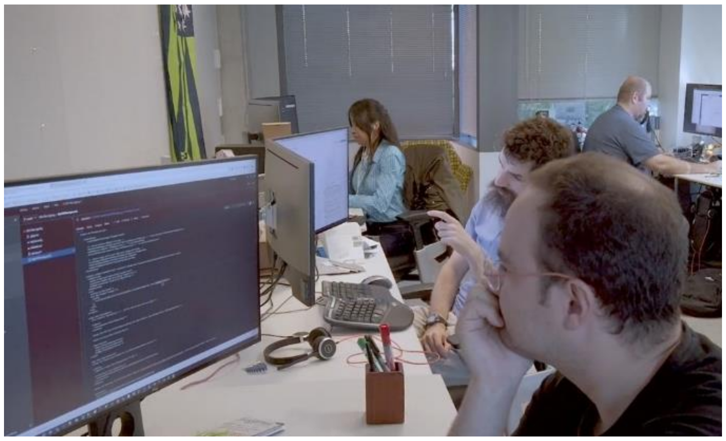
The mix of different CI systems also caused several headaches for Mitchell's team. For example, while Jenkins is flexible, maintaining a stable installation required continual babysitting, taking the majority of Mitchell's own time. Custom orchestration on some CI systems required a lot of compromises, checked in pipeline job descriptions weren't human-readable, and things quickly became over-parameterized as the team attempted to deal the broad variance in build requirements. And with official build vs. nightly validation vs. PR validation processes defined in different systems, sharing logic was difficult, forcing developers to take additional care when making process changes.

When Azure Pipelines began to roll out YAML-based build pipelines and support for public GitHub projects, Mitchell's team recognized that it had a unique opportunity. "With this new support, we could move all our existing workflows out of the separate systems and into 'modern' Azure Pipelines—and also make some changes to how we deal with official CI vs. PR workflows."

Today, all of the primary .NET Core 3.0 repos are on Azure Pipelines for public PRs and official CI. All logic is in YAML code, which is stored in GitHub, and YAML pipelines are used everywhere. Developers no longer need to learn different CI systems to move between repos and can more easily contribute changes to those CI processes. Finally, under Azure Pipelines, all CI pipelines are parallelized, enabling more CI tests to be completed in the same amount of time—and thus enabling faster iterations by developers.

Things are also better for Mitchell's team, which, having offloaded all CI onto Azure, no longer needs to worry about keeping it all up and running. "Now that we longer need to worry about the operational aspects of CI, we're free to focus on further improvements that will be felt by all—including Microsoft developers and individual contributors alike," says Mitchell.

The Results



We've evolved our infrastructure quite a bit over the years, setting ourselves up to develop and ship a more exciting product—and do so more reliably—than ever before. It's been a lot of work, but the results have been worth it.

MATT MITCHELL
Principal Software Engineer

Over the past few years, as the scope of .NET Core has continued to grow, Mitchell's team has made significant strides delivering an infrastructure that fosters innovation through open collaboration. Today, developers can more easily move from repo to repo, track down the root cause of a bug, and propose and test a fix—all without nearly *as* much specialized knowledge. Shared tooling across repos, a shared CI system, and the broad adoption of YAML are just a few examples of how the .NET Engineering Services team is continually chipping away at barriers to developer productivity—so that developers working on .NET Core, whether internal or external to Microsoft, can realize their full potential as they work together to write, test, build, and ship high-quality software.

So far, reactions from developers who've been most directly affected by the new infrastructure—namely, internal Microsoft developers who work on .NET Core all day long— have been highly positive. "A shared infrastructure has really increased my productivity and developer velocity," says Zach Danz, Software Engineer at Microsoft, who works on Windows Forms and the Windows Forms Designer. "It's nice being able to rely on the responsiveness of the .NET Engineering Services team in sticky situations, and sharing processes makes it easy to reach out to sister teams when I'm in a pickle!"

Shared tooling and a shared CI system have enabled the .NET Engineering Services team to deliver other improvements, as well. "Being able to make infrastructure changes across the full stack was probably an equally large win," says Mitchell. "For example, we recently changed how we publish our packages - which would have been very expensive tactically before, when each repo was doing its own thing."

As the work on .NET Core 3.0 winds down, the .NET Engineering Services team plans to build its recent work to deliver new improvements that will be felt by *all* developers working on .NET Core. While planning is still in the early stages, the team expects to make further investments in developer workflows and satisfaction, as well as investments in reducing the time to turn a fix into a shippable, coherent product. The team is also looking at how it can improve its infrastructure telemetry. "If we can better track where we fail, what our resource usage looks like, what our dependency state looks like, and so on, we can better determine where we need to invest to ship a better product," says Mitchell.

Mitchell's team was also involved in the decision to consolidate several foundational .NET Core repos, as a means of easing some of the major pains still faced by individual contributors. "Over the years, we pulled parts out component-by-component, giving each one its own repo," he says. "We've recently accepted that this wasn't the right approach because lots of contributors aren't familiar with our overall software architecture. Roslyn is fairly straightforward, but the runtime and SDKs can be pretty confusing—it's not clear where to start when you want to make a change. With our next release, we'll lower this barrier and make it a lot easier for individual contributors to work the same way we do at Microsoft." Concludes Mitchell, "We've evolved our infrastructure quite a bit over the years, setting ourselves up to develop and ship a more exciting product—and do so more reliably—than ever before. It's been a lot of work, but the results have been worth it. We realize that some things are still harder for individual contributors than for Microsoft developers today, but we're committed to changing that—and the work we've done getting to .NET Core gives us a foundation on which to do so."

Learn more

Story details

.NET Core infrastructure technical deep dive

.NET Core and open source overview

.NET repos on GitHub

Roslyn repo (C# compiler) builds and tests

Tools and technologies

DevOps solutions on Azure

Azure Pipelines

YAML schema for Azure Pipelines

Learn More

DevOps practices online course

Create your first pipeline quickstart

Explore more DevOps Stories >