Adopting modern engineering processes and tools at Microsoft IT

To support our customers and partners, Microsoft IT must respond more quickly to evolving business needs. Taking six months to deliver an application or an update is no longer an option. We need to deliver value faster and more efficiently. To increase our responsiveness, we started the journey to modern engineering. Our goal is to release new functionality every day in a process called continuous integration, continuous delivery.

An important part of this journey is adopting agile development processes and tools. We’re moving to an iterative design process and using agile methods to develop new services and features. And to help our engineers understand how to apply agile methods in their day-to-day work, we’ve embedded a set of practices, called Engineering Fundamentals, into our development environment. We’ve also set up tools and processes to run a project using agile methodology. This includes:

- Moving to Visual Studio Team Services to manage our backlog and work items.
- Reducing the Dev/Test environment footprint using virtualized environments hosted by Microsoft Azure.
- Improving build and test automation.
- Establishing a pipeline for continuous integration and continuous delivery using Visual Studio Team Services.

While we’re still working toward our goal of releasing every day, we’ve already shortened our release tempo from as long as six months to as little as two weeks.

![Continuous Integration Diagram](image)

*Figure 1. The continuous integration, continuous delivery process*

Using agile methodology for planning and design

In the modern engineering model, design and architecture are iterative processes. Instead of using the traditional, slow process of creating detailed plans up front, we create sketches of the future state design at a relative level of detail and fidelity. Then we use rapid prototyping to validate and refine our design and get feedback about how it works. Everyone on the team applies deep and shared customer knowledge in every design decision. Throughout the design process, we solicit ideas from a diverse audience and draw upon the work of other teams.

Applying the Engineering Fundamentals

Each sprint team applies them to each project. A checklist defines every step that an engineer should take in the modern engineering approach to do each development task. This encourages engineers to think not only about what they’re delivering, but also about how they’re delivering it.

Engineering Fundamentals serve as requirements, and each fundamental has acceptance criteria that features inherit. Sprint teams make sure that the service they deliver adheres to the fundamentals. New employees use them as
guiding principles to learn how to do engineering in our organization. And the Engineering Fundamentals are embedded in Visual Studio Team Services—where the engineers do their daily work—so they’re easy to find and use, as shown here.

Figure 2. Engineering Fundamentals embedded in Visual Studio Team Services

The Engineering Fundamentals can be summarized as follows.

Continuous delivery
Dynamic and on-demand environment provisioning. Engineers can, at any time, provision an environment that includes all of the prerequisite components and dependency services, so that when code is deployed, it can run the designated functions.

Continuous integration. Engineers can, at any time, start an unattended build for a check-in that produces a functional build in minutes, which is deployable to any environment without competing for resources.

Continuous validation. Engineers can, at any time, start an automated validation process for a deployment that increases release readiness without manual effort, in minutes.

Continuous service deployment. Engineers can, at any time, start an unattended deployment process for a build that takes minutes in a functional environment without competing for resources.

Customer focus
Safe testing in production. Engineers can, at any time, start tests to learn from experimentation or prove service health.

Know the stakeholders deriving value from the service. Engineers keep a relentless focus on stakeholders: users, customers, and partners. This helps them communicate requirements and translate them into features, user stories, and tasks that produce an output that delights the customer.
Core engineering practices

**Ready to code.** Engineers can, at any time, check out code to any development environment (local or remote) to compile, run, debug—all within minutes.

**Ready to build.** Engineers can, at any time, provision a development environment with all of the prerequisite components, so that code can be checked out, compiled, and run.

**Componentization.** Engineers use componentization to make the codebase easier to build, compose, and deploy.

**Security and privacy compliance.** All services are highly secure and comply with security and privacy standards. Engineers integrate security infrastructure and tools into the continuous delivery process/pipeline.

**Service health, analytics, and monitoring.** Engineers use telemetry and data to form insights about the user experience, system health, the business value of the service, and to support automation.

Using Azure to improve agility

On-premises deployment takes a lot of manual work to configure and manage servers. Also, automated deployment in this scenario requires custom PowerShell scripts. To support continuous integration and agile release processes, we moved projects from Microsoft Team Foundation Server running on-premises to Visual Studio Team Services on Azure. Dev/Test environments are now virtualized. We deploy a complete virtual environment with a click. And we use Visual Studio Team Services to create automation pipelines to promote code, create environments, and to kick off tests.

Using Visual Studio Team Services agile and scrum templates, we set up projects in a few minutes. Templates also support uniform processes across teams, so it’s easier for engineers to move between projects. And because release manager includes a library of scripts, we need less custom code to seamlessly promote a build through development, staging, and production.

To increase speed and agility, we incorporate other Azure services into our solutions. When solution components must remain on premises for security reasons, we use Microsoft Azure Hybrid Connection Manager to integrate them with cloud services. New applications are entirely Azure-based.

Automating build and test

The biggest challenge in moving to an agile, continuous delivery model is ensuring the quality of the code. Pushing code into production without enough testing could break the build. Handling this issue in an agile manner requires evaluating risks and mitigating them proactively. Safety is important, but so is knowing which risks are acceptable.

To manage risks, we put modern engineering tools and processes in place that help prevent production problems and that mitigate problems before customers experience them. Test automation is an important part of this. Previously, code was handed off to the test team, and they executed one or more tests against each component of a feature or solution. For example, on a Bing site, the ability to navigate to a search page might be one test case. Another might be the ability to see the search box and another the ability to type in the search box. Another might be to get a set of results. Each test succeeds or fails. Testing each component separately in this manner is time-consuming.

Today we use a different approach. Engineers write code for functional tests based on end-to-end scenarios. Scenario-based testing puts the customer experience first. It addresses these questions: What is a customer really going to do? What are the key scenarios? What *must* work? What are the critical elements to test? How can we automate the tests? We no longer have explicit tests for each component, but rather use functional tests of outcomes. For example, a functional test might get search results for a query. To get faster turnaround, the team determines what minimum level of quality is required to ship the code. The goal is to test and correct failures faster.

Production-ready check-ins

We use build definitions in Visual Studio Team Services to support gated, production-ready check-ins. Code must pass tests to be promoted to the next phase of testing or deployment. If code fails to pass a test in any phase, the engineer must fix the code and then check it back in for retesting.
The phases are as follows:

1. An engineer tests code on a local computer.
2. The code is checked into a branch and incorporated into the daily build that includes all check-ins from the last 24 hours in an integrated environment with other systems that the new code has dependencies on. First, an automated build verification test runs, then automated tests run against the environment.
3. We manually test the build against scenarios that weren’t covered in automated testing.
4. Stakeholders/customers test the feature or application to verify that it meets their requirements.
5. The code is deployed into production. It’s deployed to a limited set of users first, and then if there are no issues, released to additional sets of users.

**Improving business value and customer satisfaction**

We still have a long way to go on our journey to modern engineering, but our efforts are already yielding benefits. We’re able to maintain a consistent and predictable release cycle and deliver updates and enhancements frequently. This way our development efforts yield business value faster. Using single source control within Visual Studio Team Services has helped us increase code reuse for more efficient development. Also, applications are “production ready” at release, and rework is reduced. Finally, breaking down releases into smaller chunks has reduced risk because all features represent two weeks of effort rather than several months.

**Best practices**

We found some practices that worked well when adopting the processes and tools for modern engineering.

**To run fast, first slow down**

Balancing the need to make infrastructure improvements with ongoing business needs for new functionality was a challenge. Stakeholders agreed on priorities and timelines to make sure that we would lay a good foundation. We made the right investments early, such as in test automation. Changing the infrastructure and moving from six-month to two-week release cycles took about a year. Before we could run fast, we first had to slow down.

**Let one team manage migration to Visual Studio Team Services**

A single team was responsible for moving projects from Team Foundation Server to Visual Studio Team Services. Team members became familiar with the steps and tools involved and handled the process more efficiently as they gained experience with multiple migration projects. This process saved time and headaches.

**For more information**

A set of three case studies describes the Microsoft IT transformation. Moving to modern engineering gives an overview. Creating a culture of modern engineering within Microsoft IT describes how we’re working to change the development culture so we’re more cooperative and responsive to our customers.

**Microsoft IT Showcase**

microsoft.com/itshowcase