DevOps resource center documentation

Learn about DevOps practices, Agile methods, Git version control, DevOps at Microsoft, and how to assess your organization's DevOps progress.

About DevOps

OVERVIEW

What is DevOps?

Start planning efficient workloads

Start developing modern software

Start delivering quality services

Start operating reliable systems

Plan efficient workloads

CONCEPT

What is Agile?

What is Agile development?

What is Scrum?

What is Kanban?

Adopt an Agile culture

How Microsoft plans with DevOps

Develop modern software

CONCEPT

Select a development environment

Use continuous integration

Understand Git history

Shift left with unit tests

How Microsoft develops with DevOps

Deliver quality services

CONCEPT

Use continuous delivery

What is infrastructure as code?

Shift right to test in production

How Microsoft delivers with DevOps

Operate reliable systems

CONCEPT

What is monitoring?

Safe deployment practices

Progressive experimentation with feature flags

Eliminate downtime through versioned service updates

How Microsoft operates with DevOps

DevSecOps

CONCEPT

Security in DevOps

Enable DevSecOps with Azure and GitHub

What is DevOps?

Article • 01/25/2023

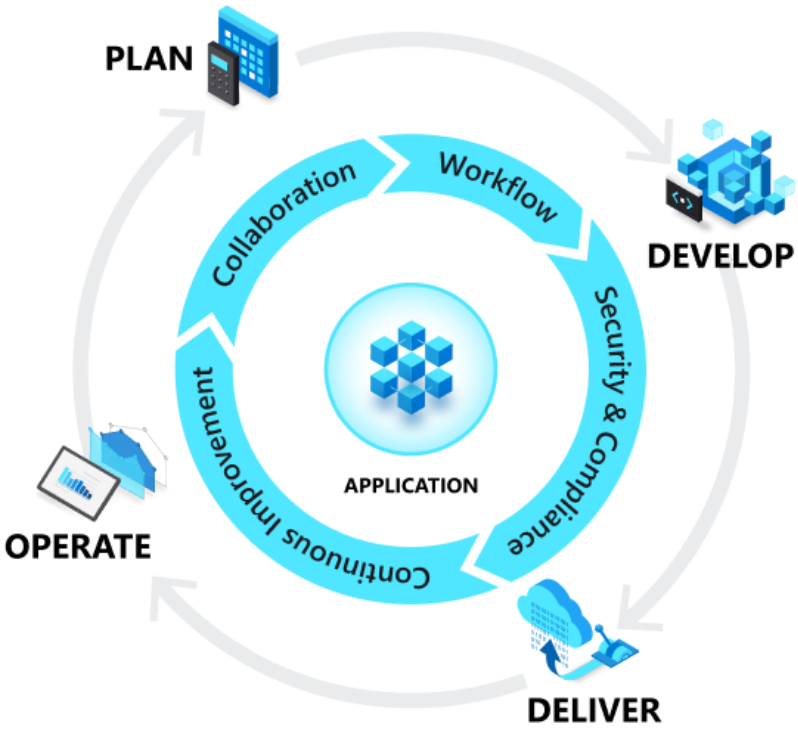
DevOps combines development (Dev) and operations (Ops) to unite people, process, and technology in application planning, development, delivery, and operations. DevOps enables coordination and collaboration between formerly siloed roles like development, IT operations, quality engineering, and security.

Teams adopt DevOps culture, practices, and tools to increase confidence in the applications they build, respond better to customer needs, and achieve business goals faster. DevOps helps teams continually provide value to customers by producing better, more reliable products.

DevOps and the application lifecycle

DevOps influences the application lifecycle throughout its planning, development, delivery, and operations phases. Each phase relies on the other phases, and the phases aren't role-specific. A DevOps culture involves all roles in each phase to some extent.

The following diagram illustrates the phases of the DevOps application lifestyle:

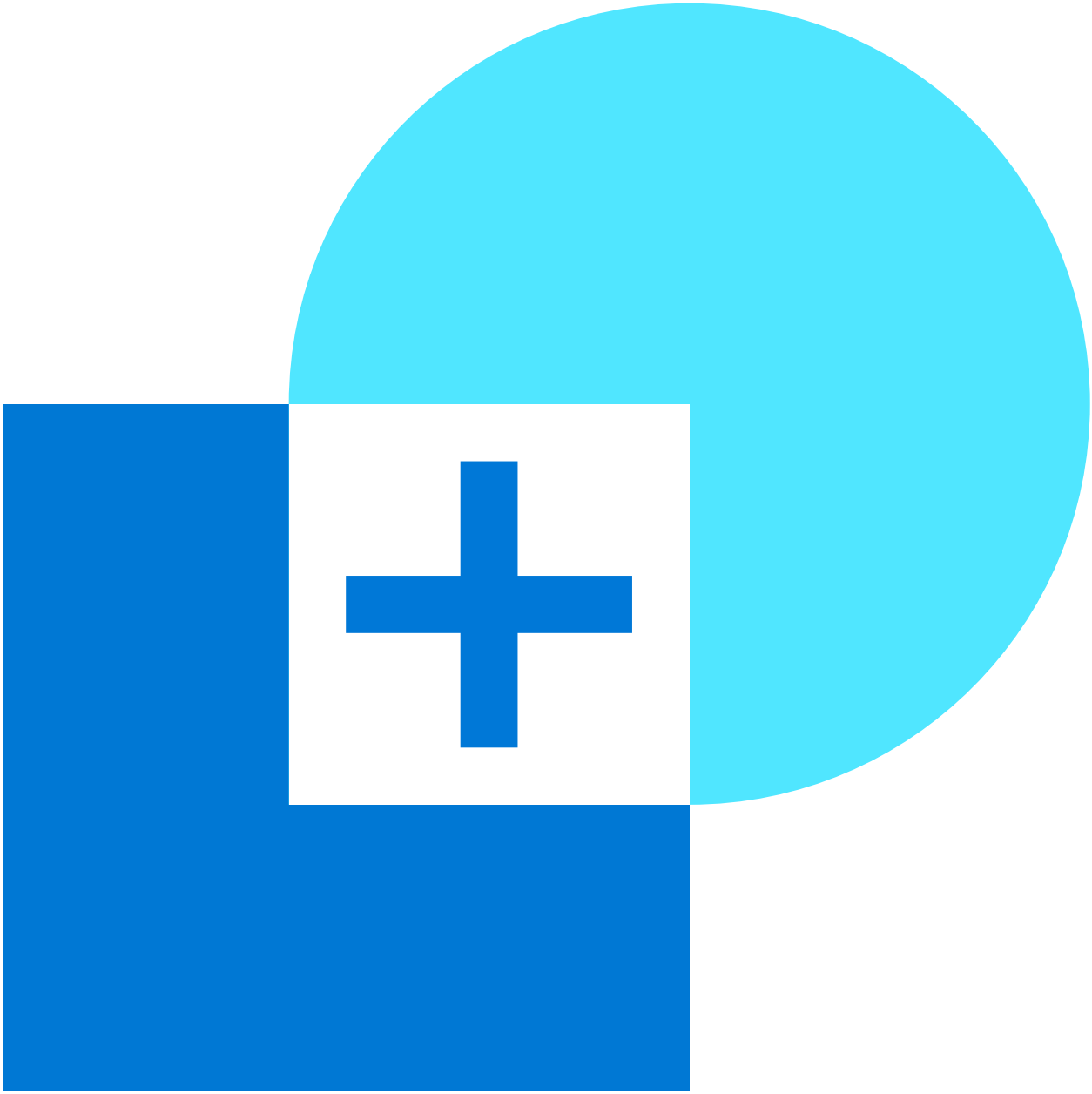


DevOps goals and benefits

When a team adopts DevOps culture, practices, and tools, they can achieve amazing things:

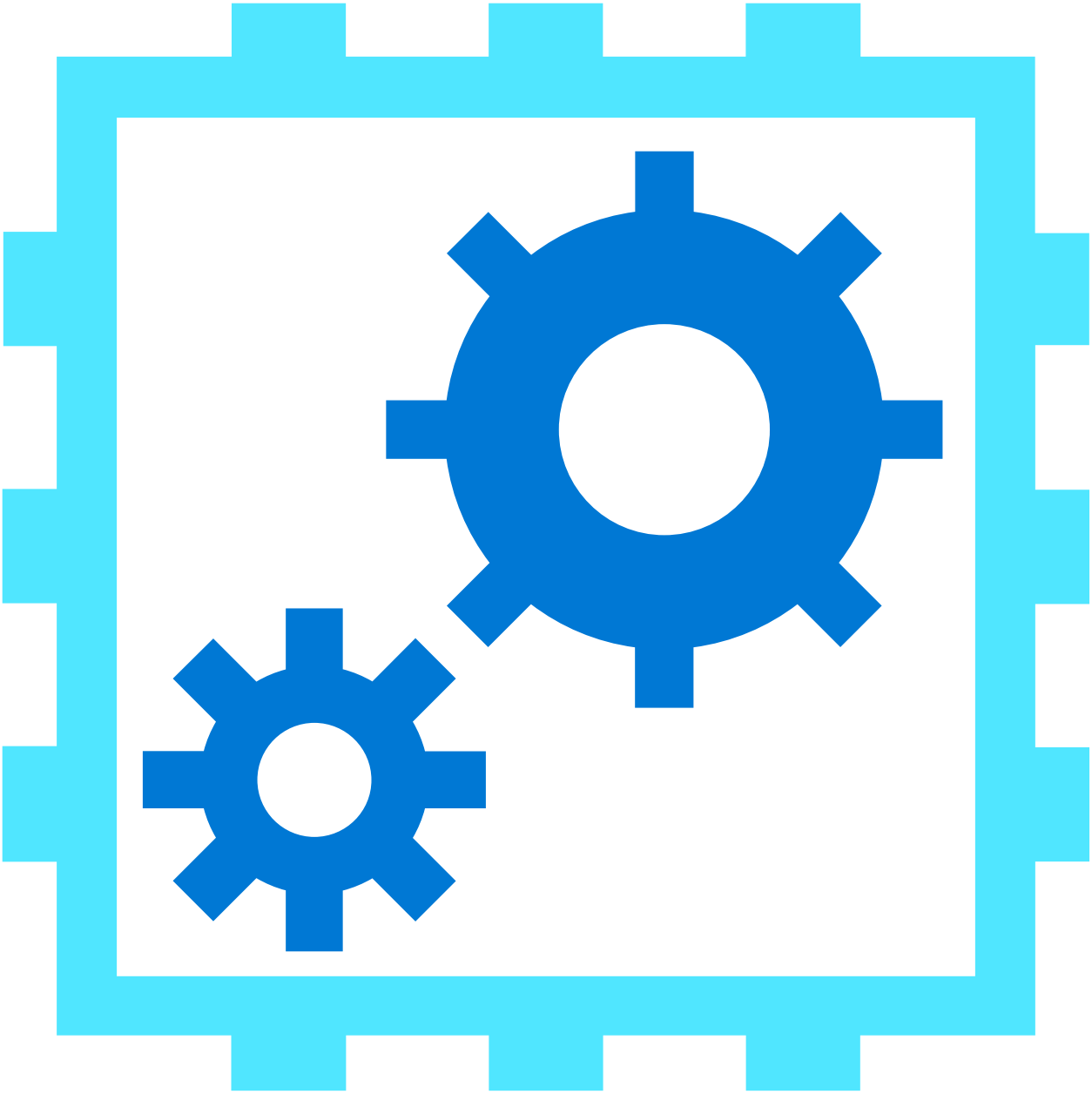
Accelerate time to market

Through increased efficiencies, improved team collaboration, automation tools, and continuous deployment--teams are able to rapidly reduce the time from product inception to market launch.



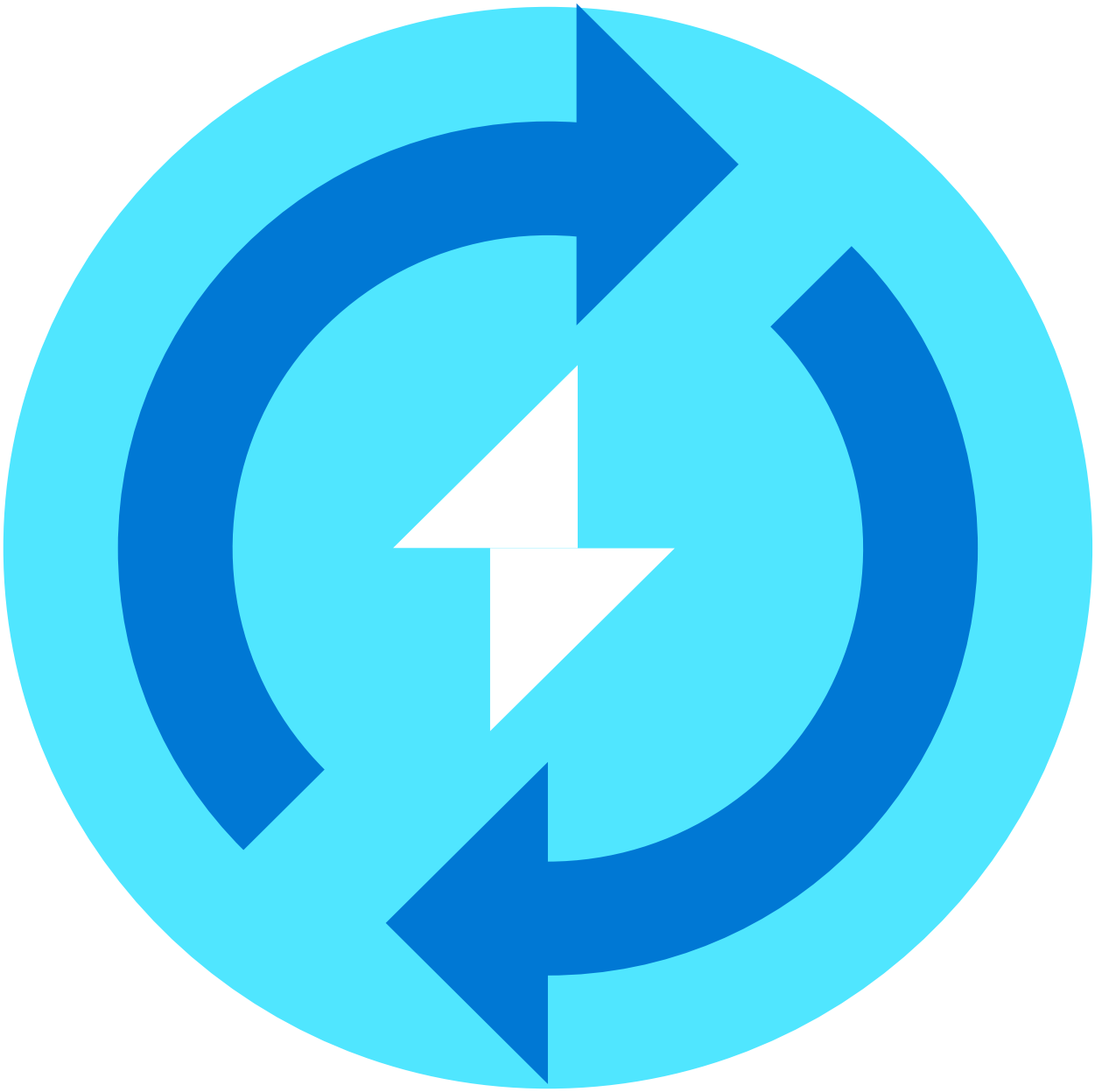
Adapt to the market and competition

A DevOps culture demands teams have a customer-first focus. By marrying agility, team collaboration, and focus on the customer experience, teams can continuously deliver value to their customers and increase their competitiveness in the marketplace.



Maintain system stability and reliability

By adopting continuous improvement practices, teams are able to build in increased stability and reliability of the products and services they deploy. These practices help reduce failures and risk.



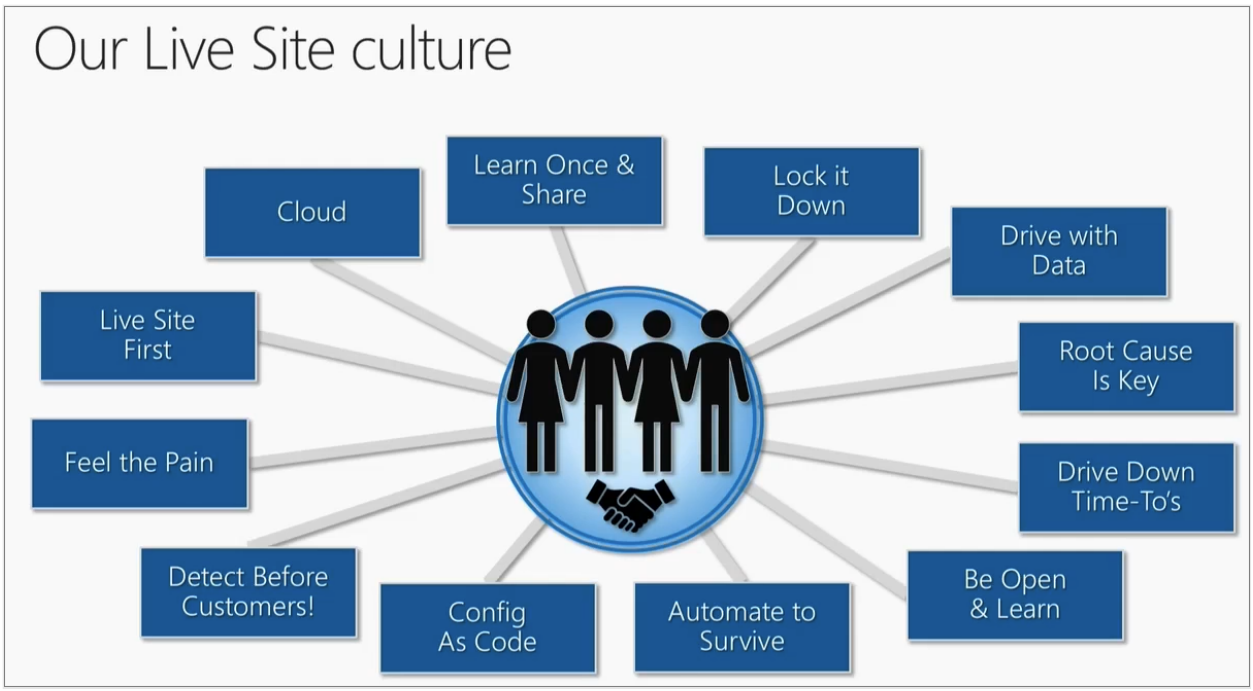
Improve the mean time to recovery

The *mean time to recovery* metric indicates how long it takes to to recover from a failure or breach. To manage software failures, security breaches, and continuous improvement plans, teams should measure and work to improve this metric.

Adopt a DevOps culture

To fully implement DevOps, you must adopt a DevOps culture. Cultivating a DevOps culture requires deep changes in the way people work and collaborate. When organizations commit to a DevOps culture, they create an environment for high-performing teams to evolve. While adopting DevOps practices automates and optimizes processes through technology, without a shift to a DevOps culture within the organization and its people, you won't gain the full benefits of DevOps.

The following image captures key aspects of Microsoft's live site culture.



The following practices are key components of a DevOps culture:

- **Collaboration, visibility, and alignment**: A hallmark of a healthy DevOps culture is collaboration between teams. Collaboration starts with visibility. Development, IT, and other teams should share their DevOps processes, priorities, and concerns with each other. By planning their work together, they are better positioned to align on goals and measures of success as they relate to the business.
- **Shifts in scope and accountability**: As teams align, they take ownership and become involved in other lifecycle phases—not just the ones central to their roles. For example, developers become accountable not only to the innovation and quality established in the develop phase, but also to the performance and stability their changes bring in the operate phase. At the same time, IT operators are sure to include governance, security, and compliance in the plan and develop phase.
- **Shorter release cycles**: DevOps teams remain agile by releasing software in short cycles. Shorter release cycles make planning and risk management easier since progress is incremental, which also reduces the impact on system stability. Shortening the release cycle also allows organizations to adapt and react to evolving customer needs and competitive pressure.
- **Continuous learning**: High-performing DevOps teams establish a growth mindset. They fail fast and incorporate learnings into their processes. They strive to continually improve, increase customer satisfaction, and accelerate innovation and market adaptability.

Implement DevOps practices

You implement DevOps by following DevOps practices (described in the sections that follow) throughout the application lifecycle. Some of these practices help accelerate, automate, and improve a specific phase. Others span several phases, helping teams create seamless processes that help improve productivity.

Continuous integration and continuous delivery (CI/CD)

Continuous Integration (CI) is the practice used by development teams to automate, merge, and test code. CI helps to catch bugs early in the development cycle, which makes them less expensive to fix. Automated tests execute as part of the CI process to ensure quality. CI systems produce artifacts and feed them to release processes to drive frequent deployments.

Continuous Delivery (CD) is a process by which code is built, tested, and deployed to one or more test and production environments. Deploying and testing in multiple environments increases quality. CD systems produce deployable artifacts, including infrastructure and apps. Automated release processes consume these artifacts to release new versions and fixes to existing systems. Systems that monitor and send alerts run continually to drive visibility into the entire CD process.

Version Control

Version control is the practice of managing code in versions—tracking revisions and change history to make code easy to review and recover. This practice is usually implemented using version control systems such as Git, which allow multiple developers to collaborate in authoring code. These systems provide a clear process to merge code changes that happen in the same files, handle conflicts, and roll back changes to earlier states.

The use of version control is a fundamental DevOps practice, helping development teams work together, divide coding tasks between team members, and store all code for easy recovery if needed. Version control is also a necessary element in other practices such as continuous integration and infrastructure as code.

Agile software development

Agile is a software development approach that emphasizes team collaboration, customer and user feedback, and high adaptability to change through short release cycles. Teams that practice Agile provide continual changes and improvements to customers, collect their feedback, then learn and adjust based on customer wants and needs. Agile is substantially different from other more traditional frameworks such as

waterfall, which includes long release cycles defined by sequential phases. Kanban and Scrum are two popular frameworks associated with Agile.

Infrastructure as code

Infrastructure as code defines system resources and topologies in a descriptive manner that allows teams to manage those resources as they would code. Those definitions can also be stored and versioned in version control systems, where they can be reviewed and reverted—again like code.

Practicing infrastructure as code helps teams deploy system resources in a reliable, repeatable, and controlled way. Infrastructure as code also helps automate deployment and reduces the risk of human error, especially for complex large environments. This repeatable, reliable solution for environment deployment lets teams maintain development and testing environments that are identical to production. Duplicating environments to different data centers and cloud platforms likewise becomes simpler and more efficient.

Configuration management

Configuration management refers to managing the state of resources in a system including servers, virtual machines, and databases. Using configuration management tools, teams can roll out changes in a controlled, systematic way, reducing the risks of modifying system configuration. Teams use configuration management tools to track system state and help avoid configuration drift, which is how a system resource's configuration deviates over time from the desired state defined for it.

Along with infrastructure as code, it's easy to templatize and automate system definition and configuration, which help teams operate complex environments at scale.

Continuous monitoring

Continuous monitoring means having full, real-time visibility into the performance and health of the entire application stack. This visibility ranges from the underlying infrastructure running the application to higher-level software components. Visibility is accomplished through the collection of telemetry and metadata and setting of alerts for predefined conditions that warrant attention from an operator. Telemetry comprises event data and logs collected from various parts of the system, which are stored where they can be analyzed and queried.

High-performing DevOps teams ensure they set actionable, meaningful alerts and collect rich telemetry so they can draw insights from vast amounts of data. These insights help the team mitigate issues in real time and see how to improve the application in future development cycles.

Planning

In the planning phase, DevOps teams ideate, define, and describe the features and capabilities of the applications and systems they plan to build. Teams track task progress at low and high levels of granularity, from single products to multiple product portfolios. Teams use the following DevOps practices to plan with agility and visibility:

- Create backlogs.
- Track bugs.
- Manage Agile software development with Scrum.
- Use Kanban boards.
- Visualize progress with dashboards.

For an overview of the several lessons learned and practices Microsoft adopted to support DevOps planning across the company's software teams, see How Microsoft plans with DevOps.

Development

The development phase includes all aspects of developing software code. In this phase, DevOps teams do the following tasks:

- Select a development environment.
- Write, test, review, and integrate the code.
- Build the code into artifacts to deploy into various environments.
- Use version control, usually Git, to collaborate on code and work in parallel.

To innovate rapidly without sacrificing quality, stability, and productivity, DevOps teams:

- Use highly productive tools.
- Automate mundane and manual steps.
- Iterate in small increments through automated testing and continuous integration (CI).

For an overview of the development practices Microsoft adopted to support their shift to DevOps, see How Microsoft develops with DevOps.

Deliver

Delivery is the process of consistently and reliably deploying applications into production environments, ideally via continuous delivery (CD).

In the delivery phase, DevOps teams:

- Define a release management process with clear manual approval stages.
- Set automated gates to move applications between stages until final release to customers.
- Automate delivery processes to make them scalable, repeatable, controlled, and well-tested.

Delivery also includes deploying and configuring the delivery environment's foundational infrastructure. DevOps teams use technologies like infrastructure as code (IaC), containers, and microservices to deliver fully governed infrastructure environments.

Safe deployment practices can identify issues before they affect the customer experience. These practices help DevOps teams deliver frequently with ease, confidence, and peace of mind.

Core DevOps principles and processes Microsoft evolved to provide efficient delivery systems are described in How Microsoft delivers software with DevOps.

Operations

The operations phase involves maintaining, monitoring, and troubleshooting applications in production environments, including hybrid or public clouds like Azure. DevOps teams aim for system reliability, high availability, strong security, and zero downtime.

Automated delivery and safe deployment practices help teams identify and mitigate issues quickly when they occur. Maintaining vigilance requires rich telemetry, actionable alerting, and full visibility into applications and underlying systems.

Practices Microsoft uses to operate complex online platforms are described in How Microsoft operates reliable systems with DevOps.

Next steps

- Plan efficient workloads with DevOps

- Develop modern software with DevOps
- Deliver quality services with DevOps
- Operate reliable systems with DevOps

Other resources

- DevOps solutions on Azure
- The DevOps journey at Microsoft
- Start doing DevOps with Azure
- Security in DevOps (DevSecOps)
- What is platform engineering?

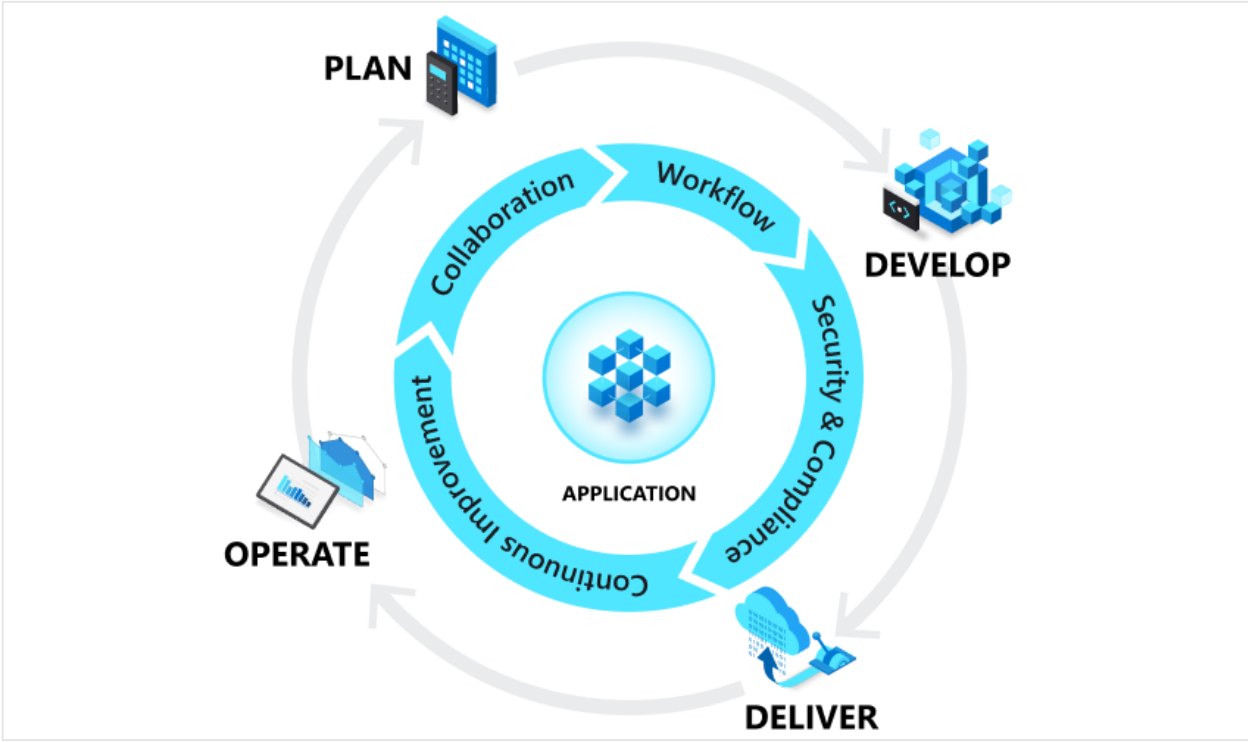
Training and Certifications

- Get started with Azure DevOps
- Introduce DevOps Dojo: Create efficiencies that support your business
- AZ-400: Get started on a DevOps transformation journey
- Facilitate communication and collaboration
- Exam AZ-400: Designing and Implementing Microsoft DevOps Solutions
- AZ-400: Implement security and validate code bases for compliance

Plan efficient workloads with DevOps

Article • 12/11/2023

The planning phase of DevOps is often seen as the first stage of DevOps, which isn't quite accurate. In practice, modern software teams work in tight cycles where each phase continuously informs the others through lessons that are learned.



Sometimes those lessons are positive. Sometimes they're negative. And sometimes they're neutral information that the team needs so that it can make strategic decisions for the future. The industry has coalesced around a single adjective to describe the ability to quickly adapt to the changing circumstances that these lessons create: *Agile*. The term has become so ubiquitous that it's now a synonym for most forms of DevOps planning.

What is Agile?

Agile describes a pragmatic approach to software development that emphasizes incremental delivery, team collaboration, continual planning, and continual learning. It's not a specific set of tools or practices, but rather a planning mindset that's always open to change and compromise.

Teams that employ Agile development practices shorten their development life cycle in order to produce usable software on a consistent schedule. The continuous focus on delivering quality to end users makes it possible for the overall project to rapidly adapt

to evolving needs. To start seeing these kinds of returns, teams need to establish some procedures along the way.

Adopt an Agile culture

Building and nurturing an Agile culture within an organization is a key investment toward effective DevOps. While the end result might be a specific set of software and services, the human resources that are required to produce and maintain those assets deserve special consideration. Teams see the best results when they invest the time to adapt their culture to match the values of the Agile mindset.

Select an Agile method

Agile methods, which are often called frameworks, are comprehensive approaches to phases of the software development life cycle. They prescribe a method for accomplishing work with clear guidance and principles. One of the most popular Agile frameworks is Scrum. Most teams that are new to Agile start with Scrum, due to its mature community and ecosystem. But there are many alternatives, so it's worth taking the time to review different options before settling.

Embrace Agile tools

There's a substantial industry that's built around tools for DevOps planning. These tools generally integrate with various Agile methods and platforms that are used in software development. One common tool is Kanban, which helps organizations and their teams visualize work in order to better plan delivery.

Build Agile teams

Teams work best when everyone has clear direction. Adopting an Agile method can greatly help in this area because Agile improves transparency in DevOps. But there are also other effective techniques that you can apply to improve the function of teams across project milestones. Any organization can benefit from building productive, customer focused teams.

Scale Agile as your organization grows

As Agile has gained popularity, many stereotypes and misinterpretations have cast a negative shadow on its effectiveness. It's easy to say "Yes, we're doing Agile" without any accountability. As time goes on, it's common for bad habits to form for various

reasons, including misunderstandings about the purpose of Agile. Small organizations might find it easy to ignore some of these misconceptions. But in larger operations, these issues can become real headaches if you don't address them. Fortunately, there are helpful guidelines for scaling Agile to large teams.

Next steps

Microsoft was one of the first major companies to adopt DevOps for planning large-scale software projects. Learn about how Microsoft plans in DevOps.

Looking for a hands-on DevOps experience? Check out the Evolve your DevOps practices learning path. It primarily features Azure DevOps, but the concepts and experience apply equally to planning in other DevOps platforms, such as GitHub.

Learn more about platform engineering, where you can use building blocks from Microsoft and other vendors to create deeply personalized, optimized, and secure developer experiences.

What is Agile?

Article • 11/28/2022

Agile is a term that describes approaches to software development that emphasize incremental delivery, team collaboration, continual planning, and continual learning. The term *Agile* was coined in 2001 in the Agile Manifesto. The manifesto set out to establish principles to guide a better approach to software development. At its core, the manifesto declares four value statements that represent the foundation of the Agile movement. As written, the manifesto states:

We have come to value:

- Individuals and interactions over processes and tools.
- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation.
- Responding to change over following a plan.

The manifesto doesn't imply that the items on the right side of these statements aren't important or needed. Rather, items on the left are simply more valued.

Agile methods and practices

It's important to understand that Agile isn't a *thing*. You don't *do Agile*. Rather, Agile is a mindset that drives an approach to software development. Because there's no single approach that works for all situations, the term *Agile* has come to represent various methods and practices that align with the value statements in the manifesto.

Agile methods, which are often called frameworks, are comprehensive approaches to phases of the DevOps lifecycle: planning, development, delivery, and operations. They prescribe a method for accomplishing work, with clear guidance and principles.

Scrum is the most common Agile framework, and the one that most people start with. Agile practices, on the other hand, are techniques that are applied during phases of the software development lifecycle.

- Planning Poker is a collaborative estimation practice that's designed to encourage team members to share their understanding of what *done* means. Many people find the process fun, and it has proven to help foster teamwork and better estimates.
- Continuous integration (CI) is a common Agile engineering practice that involves integrating code changes into the main branch frequently. An automated build verifies changes. As a result, there's a reduction in integration debt and a continually shippable main branch.

These practices, like all Agile practices, carry the *Agile* label, because they're consistent with the principles in the Agile manifesto.

What Agile isn't

As Agile has gained popularity, many stereotypes and misinterpretations have cast a negative shadow on its effectiveness. It's easy to say "*Yes, we're doing Agile*," without any accountability. Keeping this point in mind, consider a few things that Agile isn't.

- Agile isn't cowboy coding. Agile shouldn't be confused with a "we'll figure it out as we go" approach to software development. Such an idea couldn't be further from the truth. Agile requires both a definition of done and explicit value that's delivered to customers in every sprint. While Agile values autonomy for individuals and teams, it emphasizes aligned autonomy to ensure that the increased autonomy produces increased value.
- Agile isn't without rigor and planning. On the contrary, Agile methodologies and practices typically emphasize discipline in planning. The key is continual planning throughout the project, not just planning up front. Continual planning ensures that the team can learn from the work that they execute. Through this approach, they maximize the return on investment (ROI) of planning.

> "Plans are worthless, but planning is everything." — Dwight D. Eisenhower

- Agile isn't an excuse for the lack of a roadmap. This misconception has probably done the most harm to the Agile movement overall. Organizations and teams that

follow an Agile approach absolutely know where they're going and the results that they want to achieve. Recognizing change as part of the process is different from pivoting in a new direction every week, sprint, or month.

- Agile isn't development without specifications. It's necessary in any project to keep your team aligned on *why* and *how* work happens. An Agile approach to specs includes ensuring that specs are *right-sized*, and that they reflect appropriately how the team sequences and delivers work.
- Agile isn't incapable of accommodating unplanned work and other interruptions. It's important to complete sprints on schedule. But just because an issue comes up that sidetracks development doesn't mean that a sprint has to fail. Teams can plan for interruptions by designating resources ahead of time for problems and unexpected issues. Then they can address those issues but stay on track with development.
- Agile isn't inappropriate for large organizations. A common complaint is that collaboration, a key component of Agile methodologies, is difficult in large teams. Another gripe is that scalable approaches to Agile introduce structure and methods that compromise flexibility. In spite of these misconceptions, it's possible to scale Agile principles successfully. For information about overcoming these difficulties, see Scaling Agile to large teams.
- Agile isn't inefficient. To adapt to customers' changing needs, developers invest time each iteration to demonstrate a working product and collect feedback. It's true that these efforts reduce the time that they spend on development. But incorporating customer requests early on saves significant time later. When features stay aligned with the customer's vision, developers avoid major overhauls down the line.
- Agile isn't a poor fit for today's applications, which often center on data streaming. Such projects typically involve more data modeling and extract-transform-load (ETL) workloads than user interfaces. This fact makes it hard to demonstrate usable software on a consistent, tight schedule. But by adjusting goals, developers can still use an Agile approach. Instead of working to accomplish tasks each iteration, developers can focus on running data experiments. Instead of presenting a working product every few weeks, they can aim to better understand the data.

Why Agile?

So why would anyone consider an Agile approach? It's clear that the rules of engagement around building software have fundamentally changed in the last 10-15 years. Many of the activities look similar, but the landscape and environments where we apply them are noticeably different.

- Compare what it's like to purchase software today with the early 2000s. How often do people drive to the store to buy business software?
- Consider how feedback is collected from customers about products. How did a team understand what people thought about their software before social media?
- Consider how often a team desires to update and improve the software that they deliver. Annual updates are no longer feasible against modern competition.

Forrester's Diego Lo Guidice says it best in his blog, *Transforming Application Delivery* (October, 2020).

> "Everything has dramatically changed. Sustainability, besides green and clean, means that what we build today has to be easily and quickly changed tomorrow. Strategic plans are short-term, and planning and change are continuous." — Diego Lo Guidice, Forrester

The rules have changed, and organizations around the world now adapt their approach to software development accordingly. Agile methods and practices don't promise to solve every problem. But they do promise to establish a culture and environment where solutions emerge through collaboration, continual planning and learning, and a desire to ship high-quality software more often.

Next steps

Deciding to take the Agile route to software development can introduce some interesting opportunities for enhancing your DevOps process. One key set of considerations focuses on how Agile development compares and contrasts with an organization's current approach.

What is Agile development?

Article • 11/28/2022

Agile development is a term that's used to describe iterative software development. Iterative software development shortens the DevOps life cycle by completing work in short increments, usually called *sprints*. Sprints are typically one to four weeks long. Agile development is often contrasted with traditional or waterfall development, which plans larger projects up front and completes them according to the plan.

Delivering production quality code every sprint requires the Agile development team to account for an accelerated pace. All coding, testing, and quality verification must be done each and every sprint. Unless a team is properly set up, the results can fall short of expectations. While these disappointments offer great learning opportunities, it's helpful to learn some key lessons before getting started.

This article lays out a few key success factors for Agile development teams:

- Diligent backlog refinement
- Integrating early and often
- Minimizing technical debt

Diligent backlog refinement

An Agile development team works off a backlog of requirements, which are often called *user stories*. The backlog is prioritized, with the most important user stories at the top. The product owner owns the backlog and adds, changes, and reprioritizes user stories based on the customer's needs.

One of the biggest drags on an Agile team's productivity is a poorly defined backlog. A team can't be expected to consistently deliver high quality software each sprint unless they have clearly defined requirements.

The product owner's job is to ensure that every sprint, the engineers have clearly defined user stories to work with. The user stories at the top of the backlog should always be ready for the team to start on. This notion is called backlog refinement. Keeping a backlog ready for an Agile development team requires effort and discipline. Fortunately, it's well worth the investment.

When you refine a backlog, remember the following key considerations.

1. **Refining user stories is often a long-lead activity.** Elegant user interfaces, beautiful screen designs, and customer-delighting solutions all take time and energy to create. Diligent product owners refine user stories two to three sprints in advance. They account for design iterations and customer reviews. They work to ensure every user story is something the Agile team is proud to deliver to the customer.

2. **A user story isn't refined unless the team says it is.** The team needs to review the user story and agree it's ready to work on. If a team doesn't see the user story until day one of a sprint, problems can likely result.

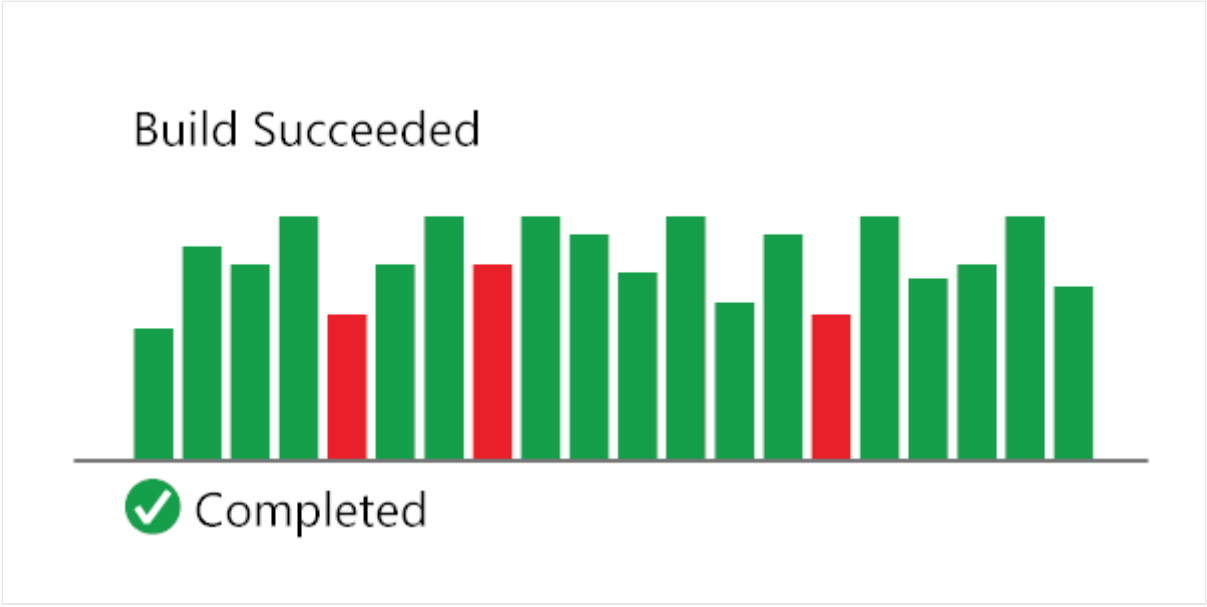
3. **User stories further down the backlog can remain ambiguous.** Don't waste time refining lower-priority items. Focus on the top of the backlog.

Integrate early and often

Continuous integration and continuous delivery (CI/CD) set up your team for the fast pace of Agile development. As soon as possible, automate the build, test, and deployment pipelines. Set up that automation as one of the first tasks your team tackles when you start a new project.

With automation, the team avoids slow, error-prone, and time-intensive manual deployment processes. Since teams release every sprint, there isn't time to do these tasks manually.

CI/CD also influences your software architecture. It ensures you deliver buildable and deployable software. When teams implement a difficult-to-deploy feature, they become aware immediately if the build and deployments fail. CI/CD forces a team to fix deployment issues as they occur. The product is then always ready to ship.



There are some key CI/CD activities that are critically important for effective Agile development.

1. **Unit testing.** Unit tests are the first defense against human error. Consider unit tests a part of coding. Check tests in with the code. Make unit testing a part of every build. Failed unit tests mean a failed build.

2. **Build automation.** The build system should automatically pull code and tests directly from source control when builds run.

3. **Branch and build policies.** Configure branch and build policies to build automatically as the team checks code in to a specific branch.

4. **Deploy to an environment.** Set up a release pipeline that automatically deploys built projects to an environment that mimics production.

Minimize technical debt

With personal finances, it's easier to stay out of debt than to dig out from under it. The same rule applies with technical debt. Technical debt includes anything that the team must address because of shortcuts that were taken earlier. For instance, if you're on a tight schedule, you might sacrifice quality to meet a deadline. Technical debt is the price you pay later, when you have to refactor code to make up for that lack of quality. Examples include fixes to address poor design, bugs, performance issues, operational issues, accessibility concerns, and other issues.

Keeping on top of technical debt requires courage. There are many pressures to delay reworking code. It feels good to work on features and ignore debt. Unfortunately, somebody must pay off the technical debt sooner or later. Just like financial debt,

technical debt becomes harder to pay off the longer it exists. A smart product owner works with their team to ensure there's time to pay off technical debt every sprint. Balancing technical debt reduction with feature development is a difficult task. Fortunately, there are some straightforward techniques for creating productive, customer-focused teams.

Always be Agile

Being Agile means learning from experience and continually improving. Agile development provides more learning cycles than traditional project planning due to the tighter process loops. Each sprint provides something new for the team to learn.

For example:

- A team delivers value to the customer, gets feedback, and then modifies their backlog based on that feedback.
- They learn that their automated builds are missing key tests. They include work in their next sprint to address this issue.
- They find that certain features perform poorly in production, so they make plans to improve performance.
- Someone on the team hears of a new practice. The team decides to try it out for a few sprints.

Teams that are just starting with Agile development should expect more learning opportunities. They're an invaluable part of the process because they lead to growth and improvement.

Next steps

There are many ways to settle on an Agile development process that's right for a team. Azure DevOps provides various process templates. Teams that are looking for different baseline structures to their planning can use these templates as starting points. For information about selecting a process template that best fits a team's culture and goals, see Choose a process flow or process template to work in Azure Boards.

As organizations grow, it can be a challenge to stay disciplined. Learn more about scaling Agile to large teams.

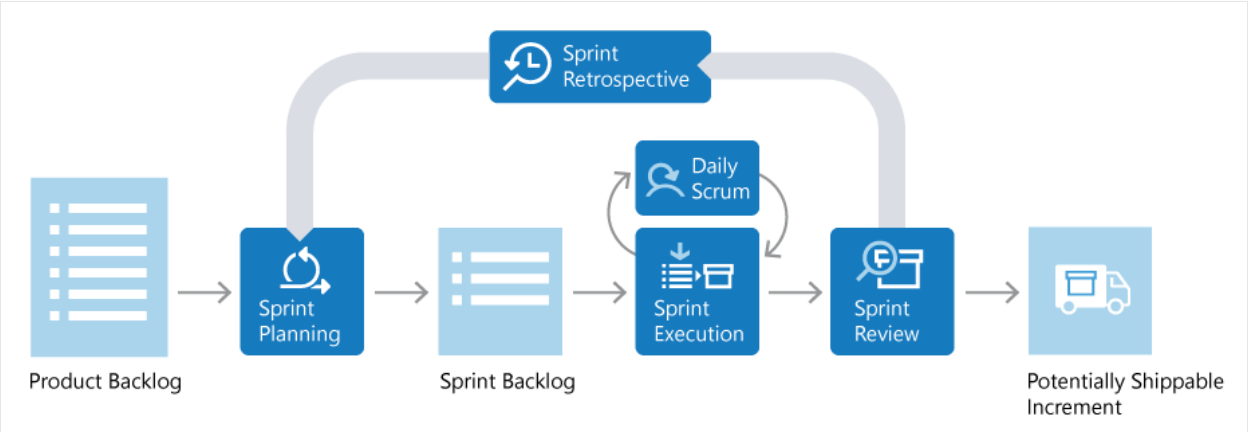
What is Scrum?

Article • 11/28/2022

Scrum is a framework used by teams to manage work and solve problems collaboratively in short cycles. Scrum implements the principles of Agile as a concrete set of artifacts, practices, and roles.

The Scrum lifecycle

The diagram below details the iterative Scrum lifecycle. The entire lifecycle is completed in fixed time periods called *sprints*. A sprint is typically one-to-four weeks long.



Scrum team roles

There are three key roles in Scrum: the *product owner*, the *Scrum master*, and the *development team*.

Product owner

The product owner is responsible for what the team builds, and why they build it. The product owner is responsible for keeping the backlog of work up to date and in priority order.

Scrum master

The Scrum master ensures that the Scrum process is followed by the team. Scrum masters are continually on the lookout for how the team can improve, while also resolving impediments and other blocking issues that arise during the sprint. Scrum masters are part coach, part team member, and part cheerleader.

Development team

The members of the development team actually build the product. The team owns the engineering of the product, and the quality that goes with it.

Product backlog

The *product backlog* is a prioritized list of work the team can deliver. The product owner is responsible for adding, changing, and reprioritizing the backlog as needed. The items at the top of the backlog should always be ready for the team to execute on.

Plan the sprint

In sprint planning, the team chooses backlog items to work on in the upcoming sprint. The team chooses backlog items based on priority and what they believe they can complete in the sprint. The *sprint backlog* is the list of items the team plans to deliver in the sprint. Often, each item on the sprint backlog is broken down into tasks. Once all members agree the sprint backlog is achievable, the sprint starts.

Execute the sprint

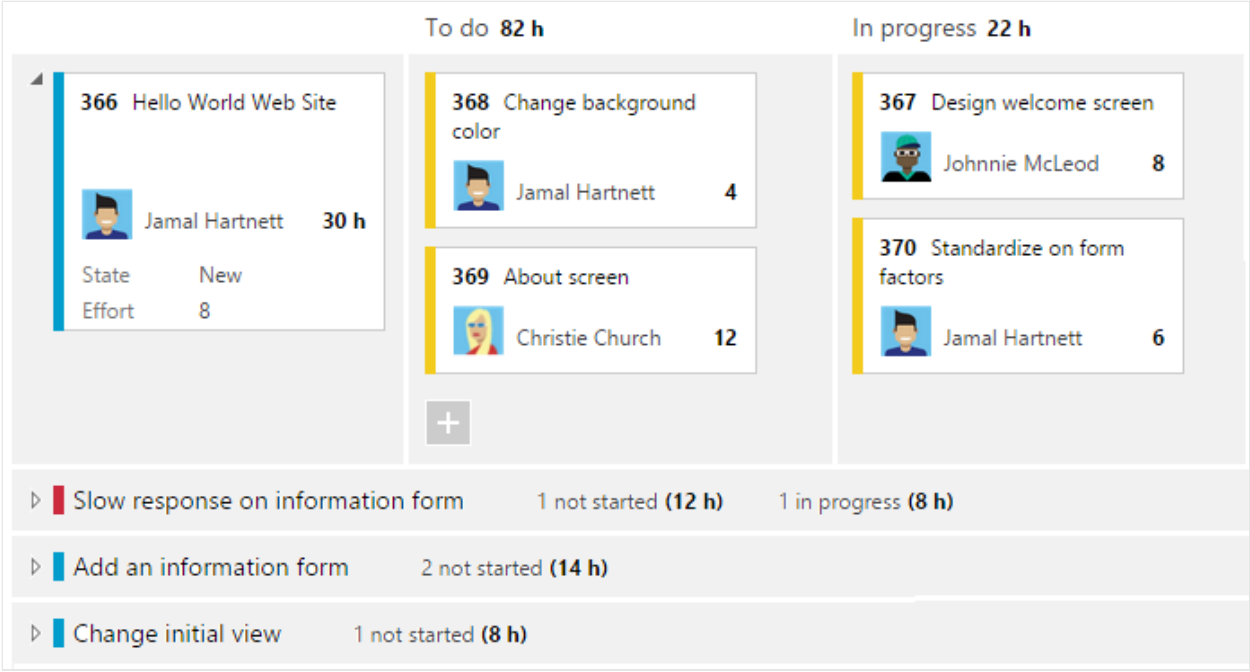
Once the sprint starts, the team executes on the sprint backlog. Scrum does not specify how the team should execute. The team decides how to manage its own work.

Scrum defines a practice called a *daily Scrum*, often called the *daily standup*. The daily Scrum is a daily meeting limited to fifteen minutes. Team members often stand during the meeting to ensure it stays brief. Each team member briefly reports their progress since yesterday, the plans for today, and anything impeding their progress.

To aid the daily Scrum, teams often review two artifacts:

Task board

The task board lists each backlog item the team is working on, broken down into the tasks required to complete it. Tasks are placed in **To do**, **In progress**, and **Done** columns based on their status. The board provides a visual way to track the progress of each backlog item.



Learn more about Kanban task boards.

Sprint burndown chart

The sprint burndown is a graph that plots the daily total of remaining work, typically shown in hours. The burndown chart provides a visual way of showing whether the team is on track to complete all the work by the end of the sprint.

Sprint review and sprint retrospective

At the end of the sprint, the team performs two practices:

Sprint review

The team demonstrates what they've accomplished to stakeholders. They demo the software and show its value.

Sprint retrospective

The team takes time to reflect on what went well and which areas need improvement. The outcome of the retrospective are actions for the next sprint.

Increment

The product of a sprint is called the *increment* or *potentially shippable increment*. Regardless of the term, a sprint's output should be of shippable quality, even if it's part

of something bigger and can't ship by itself. It should meet all the quality criteria set by the team and product owner.

Repeat, learn, improve

The entire cycle is repeated for the next sprint. Sprint planning selects the next items on the product backlog and the cycle repeats. While the team executes the sprint, the product owner ensures the items at the top of the backlog are ready to execute in the following sprint.

This shorter, iterative cycle provides the team with lots of opportunities to learn and improve. A traditional project often has a long lifecycle, say 6-12 months. While a team can learn from a traditional project, the opportunities are far less than a team who executes in two-week sprints, for example.

This iterative cycle is, in many ways, the essence of Agile.

Scrum is very popular because it provides just enough framework to guide teams while giving them flexibility in how they execute. Its concepts are simple and easy to learn. Teams can get started quickly and learn as they go. All of this makes Scrum a great choice for teams just starting to implement Agile principles.

Next steps

Find more information about Scrum resources, training, and certification:

- Scrum.org
- ScrumAlliance.org

Learn how to manage your Scrum process.

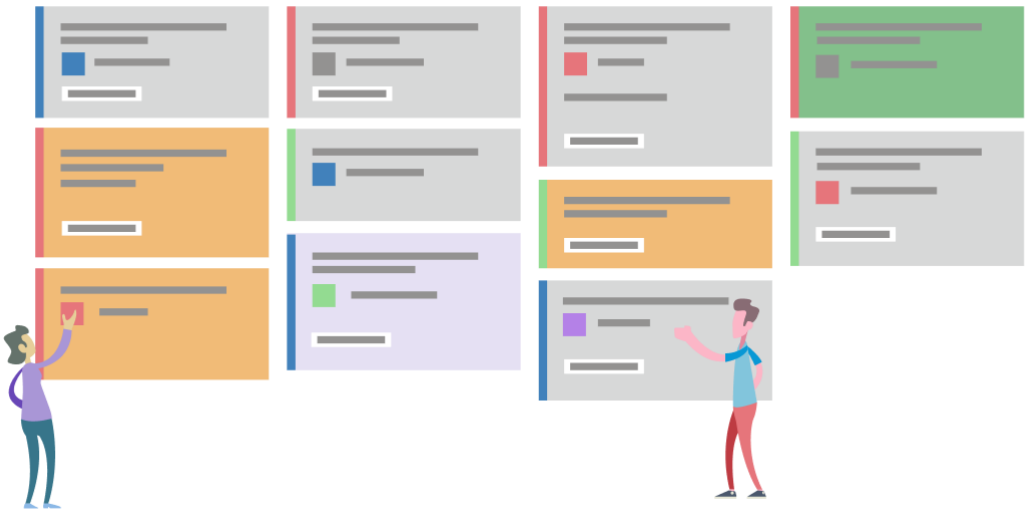
Larger, more complex organizations may find that Scrum doesn't quite fit their needs. For those cases, check out Scaled Agile Framework.

What is Kanban?

Article • 11/28/2022

Kanban is a Japanese term that means signboard or billboard. An industrial engineer named Taiichi Ohno developed Kanban at Toyota Motor Corporation to improve manufacturing efficiency.

Although Kanban was created for manufacturing, software development shares many of the same goals, such as increasing flow and throughput. Software development teams can improve their efficiency and deliver value to users faster by using Kanban guiding principles and methods.



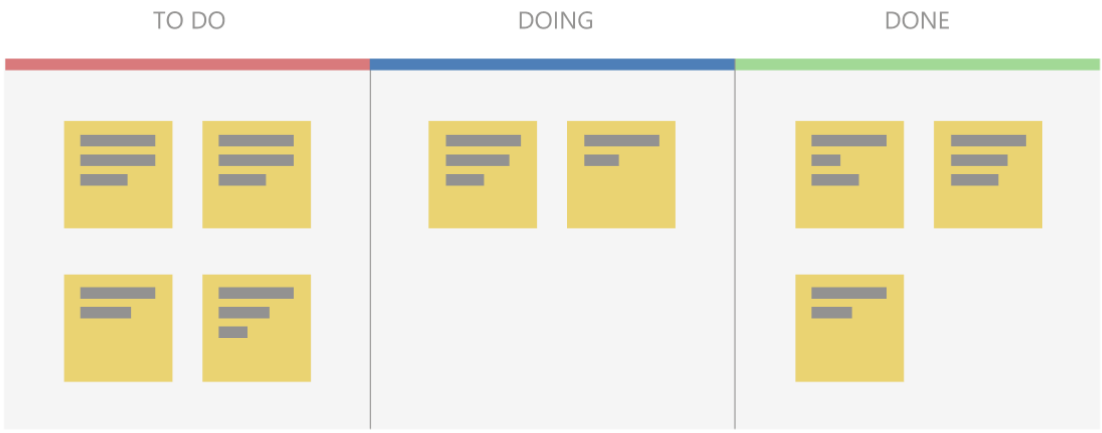
Kanban principles

Adopting Kanban requires adherence to some fundamental practices that might vary from teams' previous methods.

Visualize work

Understanding development team status and work progress can be challenging. Work progress and current state is easier to understand when presented visually rather than as a list of work items or a document.

Visualization of work is a key principle that Kanban addresses primarily through *Kanban boards*. These boards use cards organized by progress to communicate overall status. Visualizing work as cards in different states on a board helps to easily see the big picture of where a project currently stands, as well as identify potential bottlenecks that could affect productivity.



Use a pull model

Historically, stakeholders requested functionality by pushing work onto development teams, often with tight deadlines. Quality suffered if teams had to take shortcuts to deliver the functionality within the timeframe.

Kanban focuses on maintaining an agreed-upon level of quality that must be met before considering work done. To support this model, stakeholders don't push work on teams that are already working at capacity. Instead, stakeholders add requests to a backlog that a team pulls into their workflow as capacity becomes available.

Impose a WIP limit

Teams that try to work on too many things at once can suffer from reduced productivity due to frequent and costly context switching. The team is busy, but work doesn't get done, resulting in unacceptably high lead times. Limiting the number of backlog items a team can work on at a time helps increase focus while reducing context switching. The items the team is currently working on are called *work in progress (WIP)*.

Teams decide on a *WIP limit*, or maximum number of items they can work on at one time. A well-disciplined team makes sure not to exceed their WIP limit. If teams exceed their WIP limits, they investigate the reason and work to address the root cause.

Measure continuous improvement

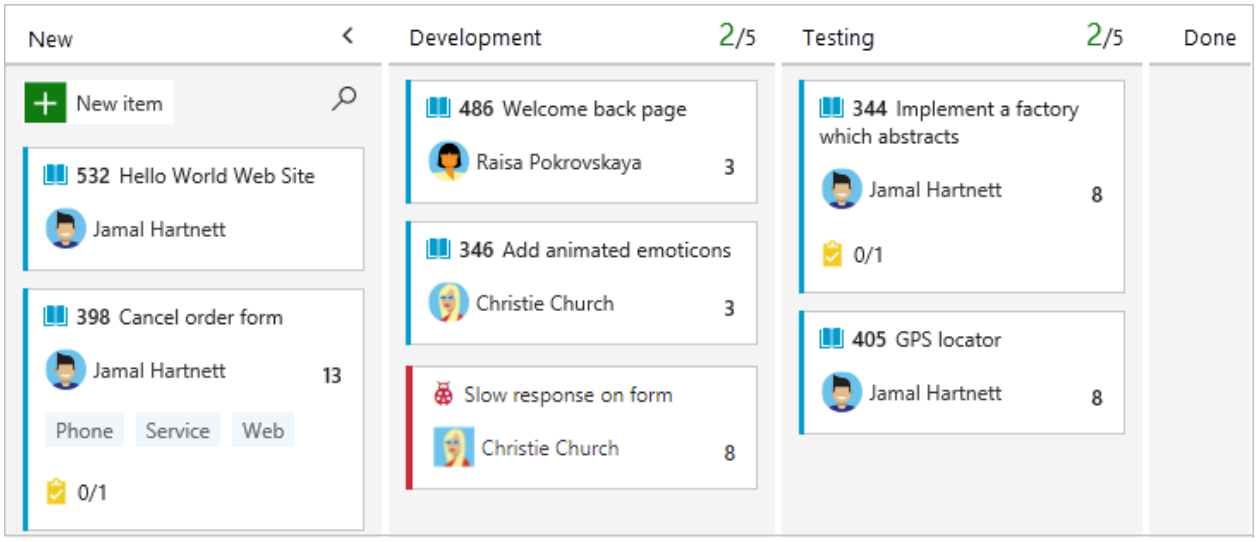
To practice continuous improvement, development teams need a way to measure effectiveness and throughput. Kanban boards provide a dynamic view of the states of work in a workflow, so teams can experiment with processes and more easily evaluate

impact on workflows. Teams that embrace Kanban for continuous improvement use measurements like *lead time* and *cycle time*.

Kanban boards

The *Kanban board* is one of the tools teams use to implement Kanban practices. A Kanban board can be a physical board or a software application that shows cards arranged into columns. Typical column names are **To-do**, **Doing**, and **Done**, but teams can customize the names to match their workflow states. For example, a team might prefer to use **New**, **Development**, **Testing**, **UAT**, and **Done**.

Software development-based Kanban boards display cards that correspond to product backlog items. The cards include links to other items, such as tasks and test cases. Teams can customize the cards to include information relevant to their process.

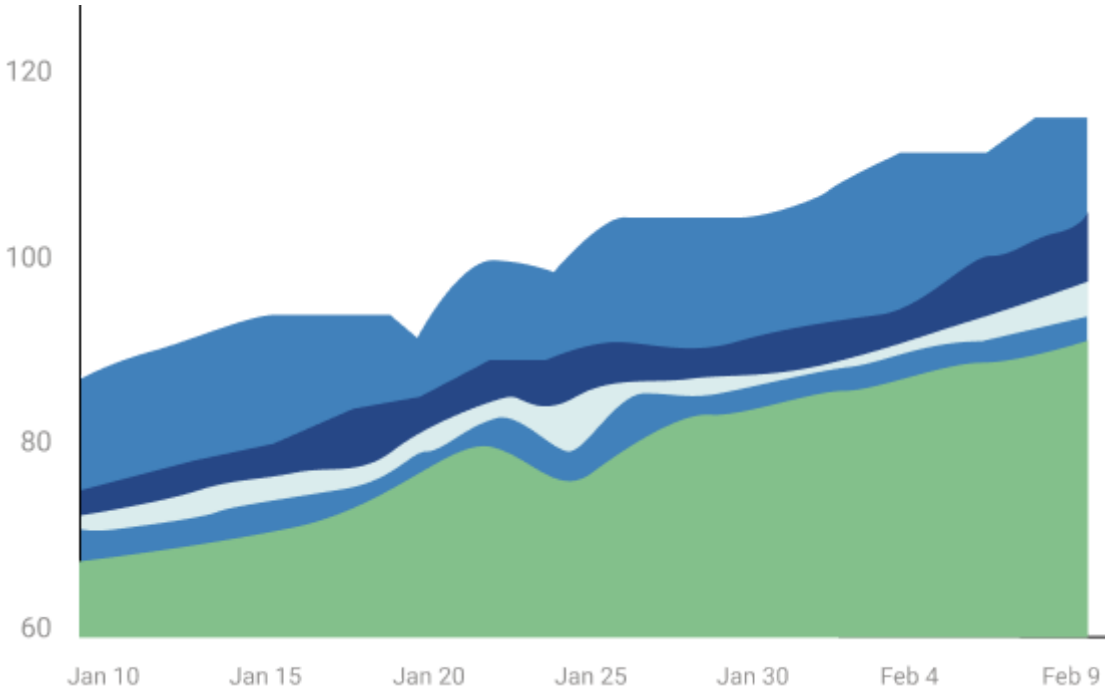


On a Kanban board, the WIP limit applies to all in-progress columns. WIP limits don't apply to the first and last columns, because those columns represent work that hasn't started or is completed. Kanban boards help teams stay within WIP limits by drawing attention to columns that exceed the limits. Teams can then determine a course of action to remove the bottleneck.

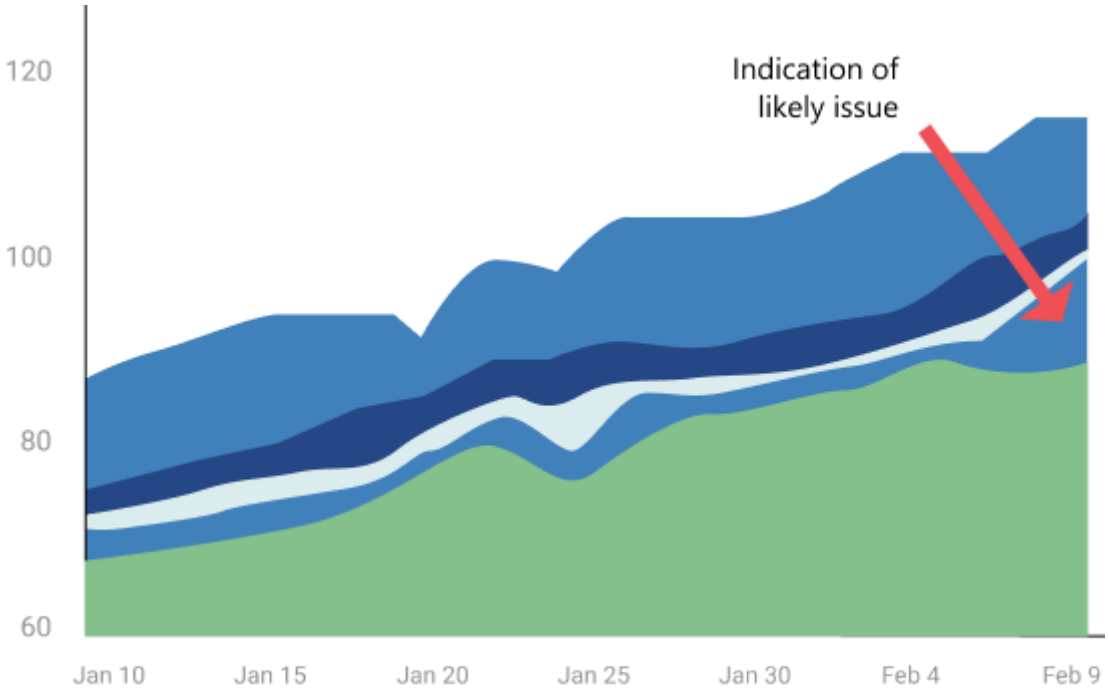
Cumulative flow diagrams

A common addition to software development-based Kanban boards is a chart called a *cumulative flow diagram (CFD)*. The CFD illustrates the number of items in each state over time, typically across several weeks. The horizontal axis shows the timeline, while the vertical axis shows the number of product backlog items. Colored areas indicate the states or columns the cards are currently in.

The CFD is particularly useful for identifying trends over time, including bottlenecks and other disruptions to progress velocity. A good CFD shows a consistent upward trend while a team is working on a project. The colored areas across the chart should be roughly parallel if the team is working within their WIP limits.



A bulge in one or more of the colored areas usually indicates a bottleneck or impediment in the team's flow. In the following CFD, the completed work in green is flat, while the testing state in blue is growing, probably due to a bottleneck.



Kanban and Scrum in Agile development

While broadly fitting under the umbrella of Agile development, Scrum and Kanban are quite different.

- Scrum focuses on fixed length sprints, while Kanban is a continuous flow model.
- Scrum has defined roles, while Kanban doesn't define any team roles.
- Scrum uses velocity as a key metric, while Kanban uses cycle time.

Teams commonly adopt aspects of both Scrum and Kanban to help them work most effectively. Regardless of which characteristics they choose, teams can always review and adapt until they find the best fit. Teams should start simple and not lose sight of the importance of delivering value regularly to users.

Kanban with GitHub

GitHub offers a Kanban experience through project boards (classic). These boards help you organize and prioritize work for specific feature development, comprehensive roadmaps, or release checklists. You can automate project boards (classic) to sync card status with associated issues and pull requests.

Kanban with Azure Boards

Azure Boards provides a comprehensive Kanban solution for DevOps planning. Azure Boards has deep integration across Azure DevOps, and can also be part of Azure Boards-GitHub integration.

- For more information, see Reasons to use Azure Boards to plan and track your work.
- The Learn module Choose an Agile approach to software development provides hands-on Kanban experience in Azure Boards.

Adopt an Agile culture

Article • 11/28/2022

If there's one lesson to be learned from the last decade of "Agile transformations," it's that there's no one-size-fits-all solution to adopting or implementing an Agile approach. Every organization has different needs, constraints, and requirements. Blindly following a prescription won't lead to success.

The Agile movement is about continually finding ways to improve the practice of building software. It's not about a perfect daily standup or retrospective. Instead, it's about creating a culture where the right thing happens more often than not. Activities like standups and retrospectives have their place, but they won't change an organization's culture.

This article details foundational elements that every organization needs to create an Agile mindset and culture. The recommendations shouldn't be followed blindly. Each organization should apply what makes sense in a given environment.

Schedule and rhythm

There's no perfect sprint length. Teams have been successful with sprint lengths that range from one to four weeks. What matters most is consistency.

Select a sprint length that works for your organization's culture, product, and desire to provide updates. For example, the Developer Tools division at Microsoft (roughly 6,000 people) works in three-week sprints. The leadership team didn't choose this sprint length; it came from direct feedback from the engineering teams. The entire division

operates on this three-week sprint schedule. The sprints have since become the *heartbeat* of the organization. Now every team marches to the beat of the same drum.

It's important to pick a sprint length and stick with it. If there are multiple Agile teams, they should all use the same sprint length. If feedback drives a change, then be receptive. It will become clear when the right term is in place.

A culture of shipping

Peter Provost, Principal Group Program Manager at Microsoft, said "You can't cheat shipping." The simplicity and truth of that statement is a cornerstone of Agile culture. What Peter means is that shipping your software will teach you things that you can't and won't understand unless you're actually shipping your software.

Human nature is to delay or avoid doing things until absolutely necessary. This couldn't be more true when it comes to software development. Teams punt bugs to the end of the cycle, don't think about setup or upgrade until they're forced to, and typically avoid things like localization and accessibility wherever possible. When this pattern emerges, teams build up technical debt that will need to be paid at a later time. Shipping demands all debt be paid. You can't cheat shipping. To establish an Agile culture, start by trying to ship the product at the end of every sprint. It won't be easy at first, but when a team attempts it, they quickly discover all the things that should be happening, but aren't.

Healthy teams

There's no recipe for the perfect Agile team. However, a few key characteristics make success much easier to achieve.

Co-locate teams whenever possible

Can a team find success with people spread out across different geographies? Yes, but it's more difficult. When people are co-located and sitting in the same room, the right conversations just tend to happen. It's still possible to succeed with teams located across globe and different time zones. But wouldn't that same team have an advantage without all of those obstacles?

Keep teams intact for a reasonable length of time

Allow teams to master the art of building software together. When teams are scrambled, any chemistry they've developed gets disrupted. Sometimes it's appropriate to reorganize, but teams are typically more productive when they're given time to learn how to work together. As a guideline, try to keep teams intact for at least 12 months.

Load balance work, not people

Sometimes teams fall behind and need help. A common tactic to address this is to lend a person from one team to another. However, that can be counterproductive. A better solution is to load balance work to another team, rather than load-balancing people between them. Pulling a person from one team to help another disrupts both teams, and it can frustrate the person being moved, even if temporary. All of this affects team productivity and, more likely than not, negatively impacts the ability to get back on schedule.

Load balancing work instead of people allows a team that's already established to step in and help out. It becomes a conversation about *priorities*, not a conversation about *people*.

Let teams own feature areas, not layers of architecture

Strive to build vertical teams that own feature areas. These teams are responsible for all the work required to add features to their area, from database to user interface changes. The team is empowered to deliver and own an end-to-end experience.

When horizontal teams own layers of architecture, no single team is responsible for the end-to-end experience. Adding a feature requires multiple teams to coordinate and requires a higher level of dependency management. Resolving bugs requires multiple teams to investigate whether they own the code required to fix the bug. Bugs are batted around as teams determine it's *not their bug* and assign it to another team.

Feature teams don't have these issues. Ownership and accountability are clear. There may be a place for some architectural-based teams. However, vertically focused teams are more effective.

Next steps

As teams embark on their own Agile transformation, keep these foundational principles in mind. Remember, there's no single recipe that will work for every organization. Agile

transformations are a journey. Make changes and learn from them. Over time the organization will develop the Agile culture it needs.

Microsoft is one of the world's largest Agile companies. Learn more about how Microsoft adopted an Agile culture for DevOps planning.

Learn about how Azure DevOps enables teams to adopt and scale an Agile culture.

Building productive teams

Article • 11/28/2022

Engineers thrive in environments where they can focus and *get in the zone*. Teams often face distractions and competing priorities that force engineers to shift context and divide their attention. They struggle to balance *heads down* time with *heads up* time. Adding new features requires team members to be heads down and focused. Responding to customer issues and addressing live site issues requires the team to be heads up and aware of what's going on.

To mitigate distractions, a team can divide itself into two *crews*: one for features and one for live site health.

The two-crew approach yields greater productivity and predictability. Successful implementation relies on these key elements:

- Clearly defined crew roles
- A well-defined crew rotation process
- Frequent adjustments to crew size

Feature crew

The feature crew, or *F-crew*, focuses on the *future*. They work as an effective unit with a clear mission and goal: to build and ship high-quality features.

The F-crew is shielded from the day-to-day chaos of the live service to ensure they have time to design, build, and test their work. They can rely on minimal distractions and

freedom from having to fix issues that arise at random. They're encouraged to seldom check their email and avoid getting pulled into other issues unless they're critical.

When an F-crew member joins a conversation or occasionally gets sucked into an email thread, other team members should chide them: "*You're on the F-crew, what are you doing?*" If an F-crew member needs to address a critical issue, they're encouraged to delegate it to the customer crew and return to feature work.

The F-crew operates as a tight-knit team that swarms on a small set of features. A good work-in-progress (WIP) limit is two features in flight for 4-6 people. By working closely together, they build deep shared context and find critical bugs or design issues that a cursory code review would miss. A dedicated crew allows for a more predictable throughput rate and lead time. Team members often refer to the F-crew as serene and focused. They find it peaceful and rejuvenating to focus deeply on a feature, to dedicate full attention to it. People leave their time on the F-crew feeling refreshed and accomplished.

Customer crew

The customer crew, or *C-crew*, focuses on the *now* and provides frontline support for customer and live-site issues, bugs, telemetry, and monitoring. The C-crew often huddles around a computer, debugging a critical live-site issue. Their number one priority is live-site health. Laser-focused on this environment, they build expert debugging and analysis skills. The customer crew is often referred to as the *shield* team, because it shields the rest of the team from distractions. Rather than work on upcoming features, the C-crew is the bridge between customers and the current product. Crew members are active on email, Twitter, and other feedback channels. Customers want to know they're heard, and the C-crew's job is to hear them. The C-crew triages customer-reported issues immediately and quickly engages and assists blocked customers.

With a deluge of incoming tasks, working on a fast-paced C-crew can, at times, be exhilarating. In a busy week, they address multiple emails, live-site investigations, and bugs. As operations quiet down, they work to improve telemetry and reporting, investing their time to make service upkeep easier.

C-crews allow the team to address issues without pulling team members off other priorities, and ensure customers and partners are heard. Responsiveness to questions and issues becomes a point of pride for C-crews. However, this pace can be draining, necessitating a frequent rotation between crews.

Crew rotation

A well-defined rotation process makes the two-crew system work. You could simply swap the crews (F-crew becomes C-crew and vice versa), but this limits knowledge sharing between and within the crews. Instead, opt for a weekly rotation.

At the end of each week, conduct a short *swap meet* where the team decides who swaps between crews. You can use a whiteboard chart to track who is currently on each crew and when they were swapped. The longest tenured people on each crew should typically swap with each other. However, in any given week, someone may want to remain to complete work on a live-site investigation or feature. While there's flexibility, the longer someone is on a crew, the more likely they should be swapped.

Weekly rotations help prevent silos of knowledge in the team and ensure a constant flow of information and perspective between crews. The frequent movement of engineers creates shared knowledge of the team's work, which helps the C-crew to resolve issues without the help of others. Often, new F-crew members will quickly find a previously overlooked design or code flaw.

Crew size

Crew size varies to maintain the health of the team. If a team has a high incoming rate of live-site issues or has a lot of technical debt, the C-crew gets larger, and vice versa. Adjusting sizes weekly increases predictability in the team's deliverables and dependencies. In some weeks, a team may move everyone to the C-crew to address the feedback from a big release.

This strategy simplifies communication with management. Without a two-crew system, engineers often work on multiple things simultaneously. When several distractions occur in a single week, in-progress features are often delayed. As a result, a team may be unable to confidently give timelines for future feature work.

A dedicated F-crew leads to predictable throughput and lead time. Splitting resources between crews increases accountability within the team and with management about what the team can accomplish each week and each sprint.

Next steps

The two-crew system can help teams understand where engineers should spend their time and to make progress on many competing priorities.

In addition to improving productivity and predictability, the two-crew system can increase team morale. Engineers on each team clearly understand their roles and responsibilities and function more independently and with much greater accountability.

This approach is ideal for DevOps teams, those responsible for both development and operations. However, this approach can be applied to nearly any Agile team dealing with competing priorities.

Microsoft is one of the world's largest Agile companies. Learn how Microsoft organizes teams in DevOps planning.

Scaling Agile to large teams

Article • 11/28/2022

The words *big* and *Agile* aren't often used in the same sentence. Large organizations have earned the reputation of being slow moving. However, that's changing. Many large software organizations are successfully making the transformation to Agile. They're learning to scale Agile principles with or without popular frameworks such as SAFe, LeSS, or Nexus.

At Microsoft, one such organization uses Agile to build products and services shipped under the Azure DevOps brand. This group has 35 feature teams that release to production every three weeks.

Every team within Azure DevOps owns features from start to finish and beyond. They own customer relationships. They manage their own product backlog. They write and check code into the production branch. Every three weeks, the production branch is deployed and the release becomes public. The teams then monitor system health and fix live-site issues.

According to Agile principles, autonomous teams are more productive. An Agile organization wants their teams to have control over their day-to-day execution. But autonomy without alignment would result in chaos. Dozens of teams working independently wouldn't produce a unified, high-quality product. Alignment gives teams their purpose and ensures that they meet organizational goals. Without alignment, even the best performing teams would fail.

To scale Agile, you must enable autonomy for the team while ensuring alignment with the organization.

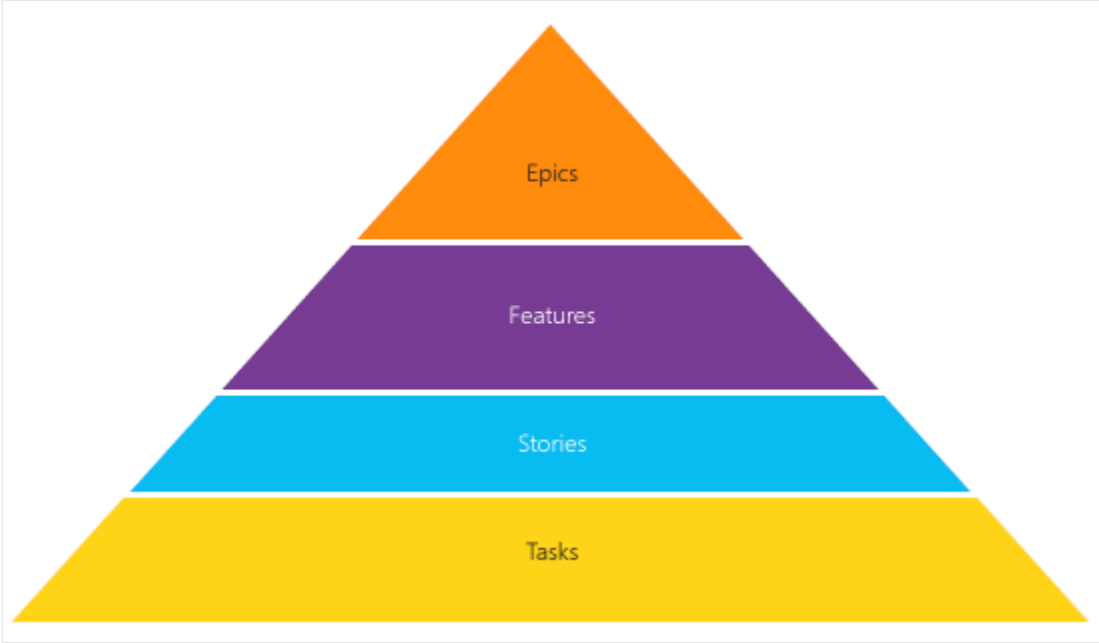
To manage the delicate balance between alignment and autonomy, DevOps leaders need to define a taxonomy, define a planning process, and use feature chats.

Define a taxonomy

An Agile team, and the larger Agile organization it belongs to, need a clearly defined backlog to be successful. Teams will struggle to succeed if organizational goals are unclear.

In order to set clear goals and state how each team contributes to them, the organization needs to define a taxonomy. A clearly defined taxonomy creates the nomenclature for an organization.

A common taxonomy is **epics**, **features**, **stories**, and **tasks**.



Epics

Epics describe initiatives important to the organization's success. Epics may take several teams and several sprints to accomplish, but they aren't without an end. Epics have a clearly defined goal. Once attained, the epic is closed. The number of epics in progress should be manageable in order to keep the organization focused. Epics are broken down into features.

Features

Features define new functionality required to realize an epic's goal. Features are the release-unit; they represent what is released to the customer. Published release notes can be built based on the list of features recently completed. Features can take multiple sprints to complete, but should be sized to ensure a consistent flow of value to the customer. Features are broken down into stories.

Stories

Stories define incremental value the team must deliver to create a feature. The team breaks the feature down into incremental pieces. A single completed story may not provide meaningful value to the customer. However, a completed story represents production-quality software. Stories are the team's work unit. The team defines the stories required to complete a feature. Stories optionally break down into tasks.

Tasks

Tasks define the work required to complete a story.

Initiatives

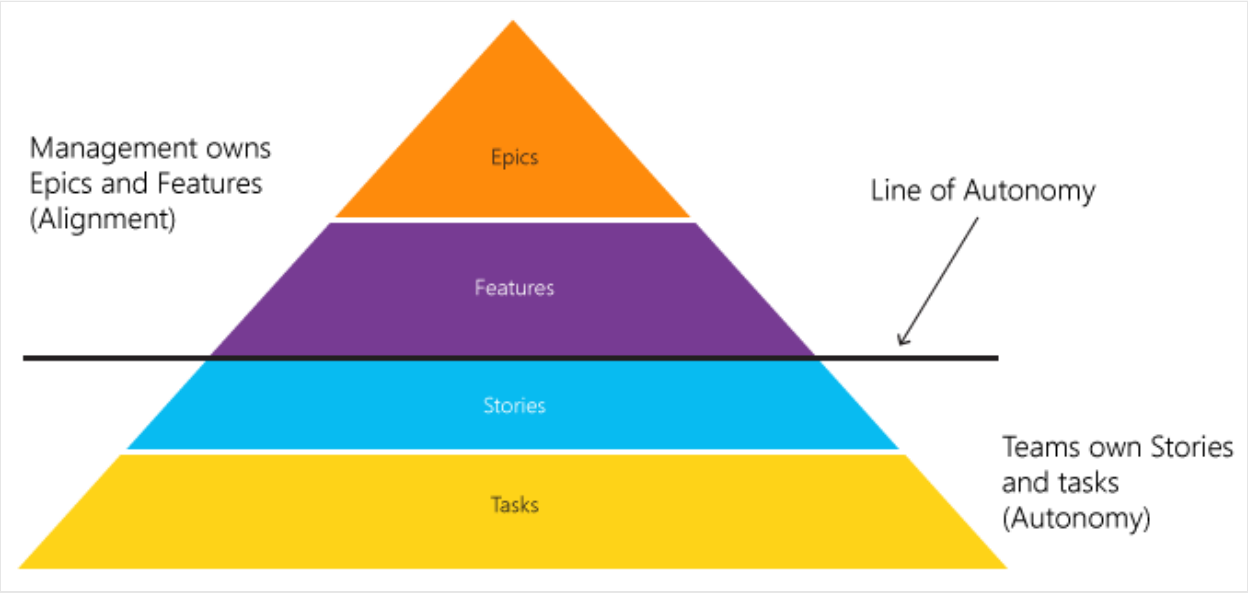
This taxonomy isn't a one-size-fits-all system. Many organizations introduce a level above epics called **initiatives**.

The names of each level can be tailored to your organization. However, the names defined above (epics, features, stories) are widely used in the industry.

Line of autonomy

Once a taxonomy is set, the organization needs to draw a *line of autonomy*. The line of autonomy is the point at which ownership of the level passes from management to the team. Management doesn't interfere with levels that are owned by the team.

The following example shows the line of autonomy drawn below features. Management owns epics and features, which provide alignment. Teams own stories and tasks, and have autonomy over how they execute.



In this example, management doesn't tell the team how to decompose stories, plan sprints, or execute work.

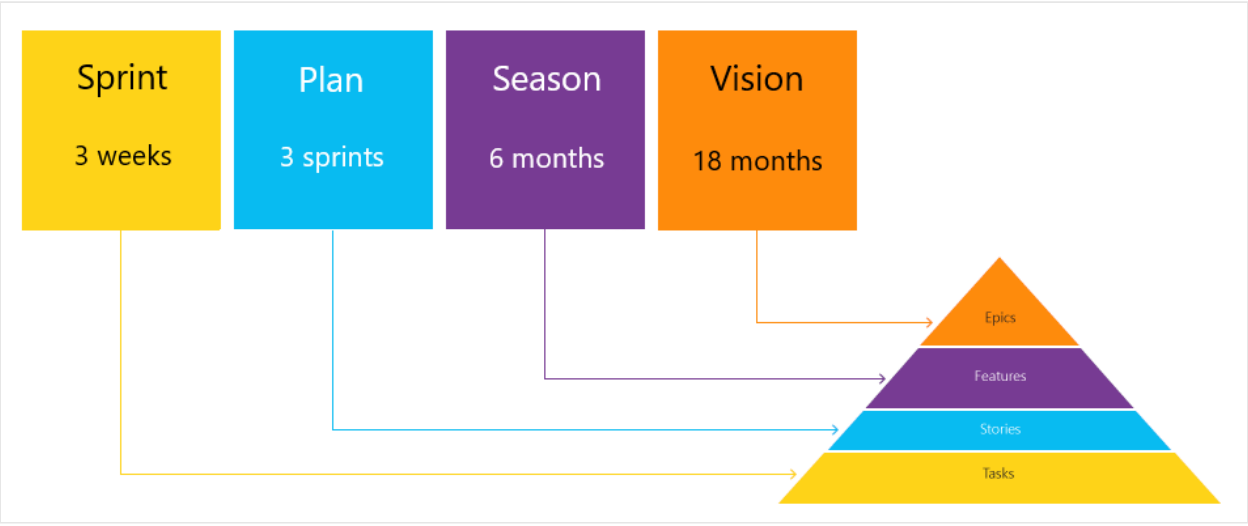
The team, however, must ensure their execution aligns with management's goals. While a team owns their backlog of stories, they must align their backlog with the features assigned to them.

Planning

To scale Agile planning, a team needs a plan for each level of the taxonomy. However, it's important to iterate and update the plan. This process is called rolling wave planning.

The plan provides direction for a fixed period of time with expected calibration at regular intervals. For example, an 18-month plan could be calibrated every six months.

Here's an example of planning methods for each level of a taxonomy: epics, features, stories, tasks.



Vision

The **vision** is expressed through epics and sets the long-term direction of the organization. Epics define what the organization wants to complete in the next 18 months. Management owns the plan and calibrates it every six months.

The vision is presented at an all-hands meeting. As the vision is intended to be ambitious and a lot can change over that period of time, expect to accomplish about 60% of the vision.

Season

A **season** is described through features and sets the strategy for the next six months. Features determine what the organization wants to light up for its customers. Management owns the seasonal plan and presents the vision and seasonal plans at an all-hands meeting. All team plans must align with management's seasonal plan. Expect to accomplish about 80% of the seasonal plan.

3-sprint plan

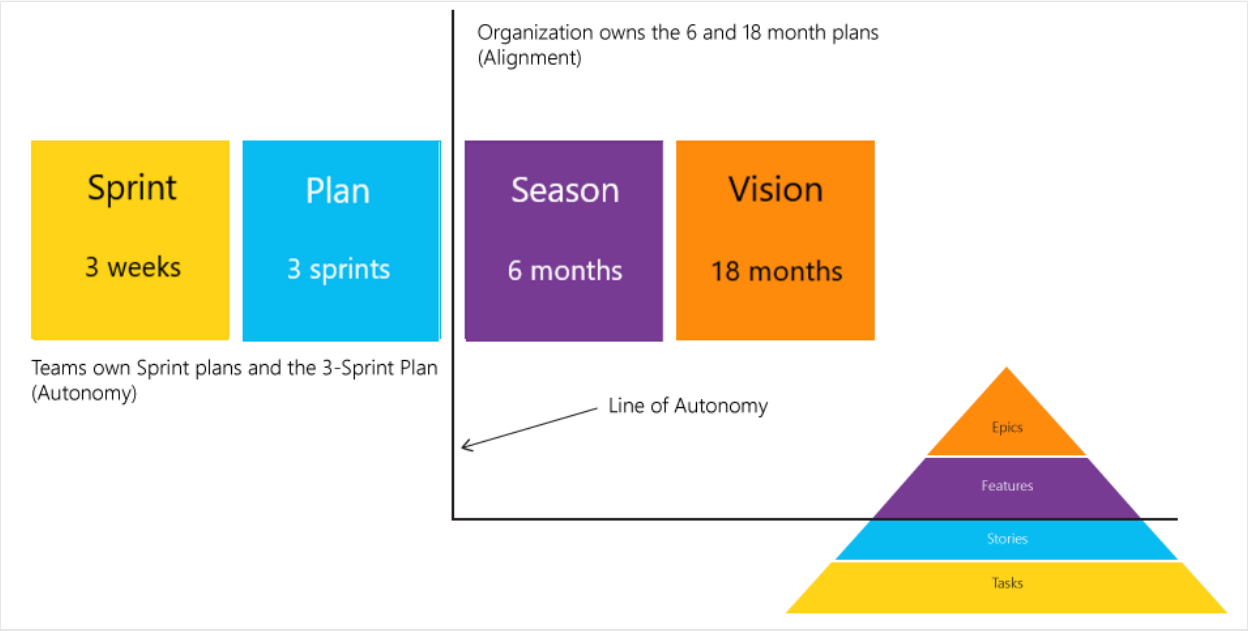
The **3-sprint plan** defines the stories and features the team will finish over the next three sprints. The team owns the plan and calibrates it every sprint. Each team presents their plan to management via the feature chat (see below). The plan specifies how the team's execution aligns with the 6-month seasonal plan. Expect to accomplish about 90% of the 3-sprint plan.

Sprint plan

The **sprint plan** defines the stories and features the team will finish in the next sprint. The team owns the sprint plan and emails it to the entire organization for full transparency. The plan includes what the team accomplished in the past sprint and their focus for the next sprint. Expect to accomplish about 95% of the sprint plan.

Line of autonomy

In this example, the line of autonomy is drawn to show where teams have planning autonomy.



As stated above, management doesn't extend ownership below the line of autonomy. Management provides guidance using the vision and season plans, and then gives the teams autonomy to create 3-sprint and sprint plans.

Feature chats: Where autonomy meets alignment

A **feature chat** is a low-ceremony meeting where each team presents their 3-sprint plan to management. This meeting ensures that team plans align with organizational goals. It also helps management stay informed about what the team is doing. While the 3-sprint plan is calibrated every sprint, feature chats are held as-needed, usually every one-to-three sprints.

A feature chat meeting allocates 15 minutes to each team. With 12 feature teams, these meetings can be scheduled to last about three hours. Each team prepares a 3-slide deck, with the following slides:

Features

The first slide outlines the features that the team will light up in the next three sprints.

Debt

The next slide describes how the team manages technical debt. Debt is anything that doesn't meet management's quality bars. The director of engineering sets the quality bars, which are the same for all teams (alignment). Example quality bars include number of bugs per engineer, percentage of unit tests passing, and performance goals.

Issues and dependencies

The issues and dependencies listed on the final slide include anything that impacts team progress, such as issues the team can't resolve or dependencies on other teams that need escalation.

Each team presents their slides directly to management. The team presents how their 3-sprint plan aligns with the 6-month seasonal plan. Leadership asks clarifying questions and suggests changes in direction. They can also request follow-up meetings to resolve deeper issues.

If a team's plan fails to align with management's expectations, management may request a re-plan. In this rare event, the team will re-plan and schedule a second feature chat to review it.

Trust: The glue that holds alignment and autonomy together

When practicing Agile at scale, trust is a two-way street:

- Management must trust teams to do the right thing. If management doesn't trust the teams, they won't give teams autonomy.
- A team earns trust by consistently delivering high-quality code. If teams aren't trustworthy, management won't give them autonomy.

Management must provide clear plans for teams to align with and then trust their teams to execute. Teams must align their plans with the organization and execute in a trustworthy manner.

As organizations look to scale Agile to larger scenarios, the key is to give teams autonomy while ensuring they're aligned with organizational goals. The critical building blocks are clearly defined ownership and a culture of trust. Once an organization has this foundation in place, they'll find that Agile can scale very well.

Next steps

There are many ways for a team of any size to start seeing benefits today. Check out some of these practices that scale.

Learn about Azure DevOps features for portfolio management and visibility across teams.

Microsoft is one of the world's largest Agile companies. Learn more about how Microsoft scales DevOps planning.

How Microsoft plans with DevOps

Article • 11/28/2022

Microsoft is one of the largest companies in the world to use Agile methodologies. Over years of experience, Microsoft has developed a DevOps planning process that scales from the smallest projects up through massive efforts like Windows. This article describes many of the lessons learned and practices Microsoft implements when planning software projects across the company.

Instrumental changes

The following key changes help make development and shipping cycles healthier and more efficient:

- Promote cultural alignment and autonomy.
- Change focus from individuals to teams.
- Create new planning and learning strategies.
- Implement a multi-crew model.
- Improve code health practices.
- Foster transparency and accountability.

Promote cultural alignment and autonomy

Peter Drucker said, "Culture eats strategy for breakfast." Autonomy, mastery, and purpose are key human motivations. Microsoft tries to provide these motivators to PMs, developers, and designers so they feel empowered to build successful products.

Two important contributors to this approach are *alignment* and *autonomy*.

- Alignment comes from the top down, to ensure that individuals and teams understand how their responsibilities align with broader business goals.
- Autonomy happens from the bottom up, to ensure that individuals and teams have an impact on day-to-day activities and decisions.

There is a delicate balance between alignment and autonomy. Too much alignment can create a negative culture where people perform only as they're told. Too much autonomy can cause a lack of structure or direction, inefficient decision-making, and poor planning.

Change focus from individuals to teams

Microsoft organizes people and teams into three groups: PM, design, and engineering.

- PM defines what Microsoft builds, and why.
- Design is responsible for designing what Microsoft builds.
- Engineering builds the products and ensures their quality.

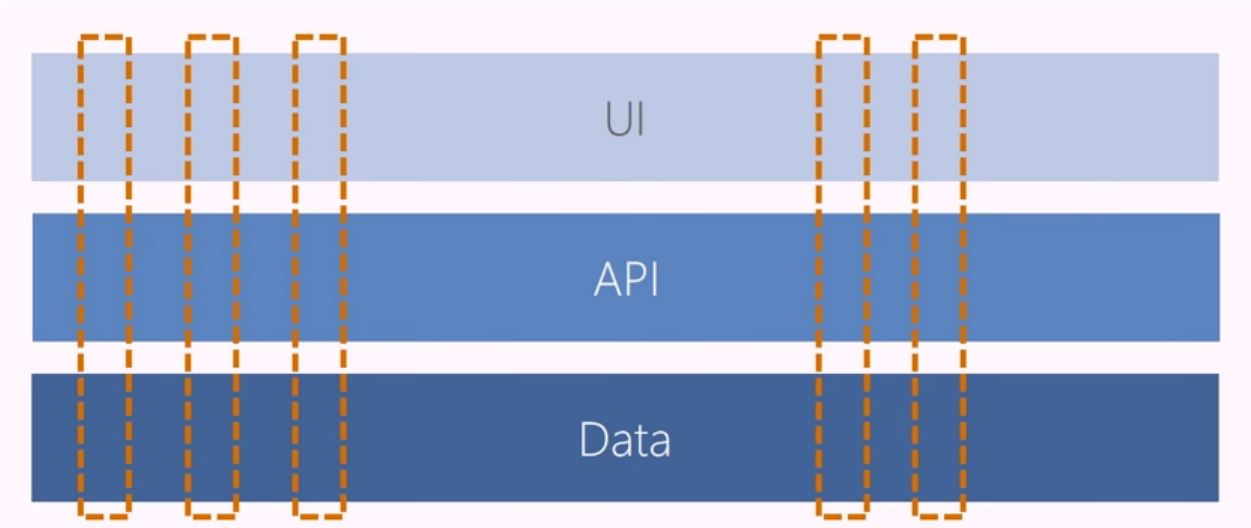
Microsoft teams have the following key characteristics:

- Cross-disciplinary
- 10-12 people
- Self-managing
- Clear charter and goals for 12-18 months
- Physical team rooms
- Own feature deployment
- Own features in production

Strive for vertical teams

Microsoft teams used to be horizontal, covering all UI, all data, or all APIs. Now, Microsoft strives for *vertical teams*. Teams own their areas of the product end-to-end. Strict guidelines in certain tiers ensure uniformity among teams across the product.

The following diagram conceptualizes the difference between horizontal and vertical teams:



Allow self-selecting teams

About every 18 months, Microsoft runs a "yellow sticky exercise," where developers can choose which areas of the product they want to work on for the next couple of planning periods. This exercise provides autonomy, as teams can choose what to work on, and organizational alignment, as it promotes balance among the teams. About 80% of the

people in this exercise remain on their current teams, but they feel empowered because they had a choice.

Create new planning and learning strategies

Dwight Eisenhower said, "Plans are worthless, but planning is everything." Microsoft planning breaks down into the following structure:

- Sprints (3 weeks)
- Plans (3 sprints)
- Seasons (6 months)
- Strategies (12 months)

Engineers and teams are mostly responsible for sprints and plans. Leadership is primarily responsible for seasons and strategies.

The following diagram illustrates Microsoft planning strategy:

This planning structure also helps maximize *learning* while doing planning. Teams are able to get feedback, find out what customers want, and implement customer requests quickly and efficiently.

Implement a multi-crew model

Previous methods fostered an "interrupt culture" of bugs and live site incidents. Microsoft teams came up with their own way to provide focus and avoid distractions. Teams self-organize for each sprint into two distinct crews: *Features (F-crew)* and *Customer (C-crew)*.

The F-crew works on committed features, and the C-crew deals with live site issues and interruptions. The team establishes a rotating cadence that lets members plan activities more easily. For more information about the multi-crew model, see Build productive, customer focused teams.

Improve code health practices

Before switching to Agile methodologies, teams used to let code bugs build up until code was complete at the end of the development phase. Teams then discovered bugs and worked on fixing them. This practice created a roller coaster of bugs, which affected team morale and productivity when teams had to work on bug fixes instead of implementing new features.

Teams now implement a *bug cap*, calculated by the formula `# of engineers x 5 = bug cap`. If a team's bug count exceeds the bug cap at the end of a sprint, they must stop working on new features and fix bugs until they are under their cap. Teams now pay down bug debt as they go.

Foster transparency and accountability

At the end of each sprint, every team sends a mail reporting what they've accomplished in the previous sprint, and what they plan to do in the next sprint.

Objectives and key results (OKRs)

Teams are most effective when they're clear on the goals the organization is trying to achieve. Microsoft provides clarity for teams through *objectives and key results (OKRs)*.

- *Objectives* define the goals to achieve. Objectives are significant, concrete, action oriented, and ideally inspirational statements of intent. Objectives represent big ideas, not actual numbers.
- *Key results* define steps to achieve the objectives. Key results are quantifiable outcomes that evaluate progress and indicate success against objectives in a specific time period.

OKRs reflect the best possible results, not just the most probable results. Leaders try to be ambitious and not cautious. Pushing teams to pursue challenging key results drives acceleration against objectives and prioritizes work that moves towards larger goals.

Adopting an OKR framework can help teams perform better for the following reasons:

- Every team is *aligned* on the plan.

- Teams focus on *achieving outcomes* rather than completing activities.
- Every team is *accountable* for efforts on a regular basis.

OKRs might exist at different levels of a product. For example, there can be top-level product OKRs, component-level OKRs, and team-level OKRs. Keeping OKRs aligned is relatively easy, especially if objectives are set top-down. Any conflicts that arise are valuable early indicators of organizational misalignment.

OKR example

Objective: Grow a strong and happy customer base.

Key results:

- Increase external net promoter score (NPS) from 21 to 35.
- Increase docs satisfaction from 55 to 65.
- New pipeline flow has an Apdex score of 0.9.
- Queue time for jobs is 5 seconds or less.

For more information about OKRs, see Measure business outcomes using objectives and key results.

Select the right metrics

Key results are only as useful as the metrics they're based on. Microsoft uses leading indicators that focus on change. Over time, these metrics build a working picture of product acceleration or deceleration. Microsoft often uses the following metrics:

- Change in monthly growth rate of adoption
- Change in performance
- Change in time to learn
- Change in frequency of incidents

Teams avoid metrics that don't accrue value toward objectives. While they may have certain uses, the following metrics aren't helpful for tracking progress toward objectives:

- Accuracy of original estimates
- Completed hours
- Lines of code
- Team capacity
- Team burndown
- Team velocity
- Number of bugs found

- Code coverage

Before Agile and after Agile

The following table summarizes the changes Microsoft development teams made as they adopted Agile practices.

Before	After
4-6 month milestones	3-week sprints
Horizontal teams	Vertical teams
Personal offices	Team rooms and remote work
Long planning cycles	Continual planning and learning
PM, Dev, and Test	PM, Design, and Engineering
Yearly customer engagement	Continual customer engagement
Feature branches	Everyone works in the main branch
20+ person teams	8-12 person teams
Secret roadmap	Publicly shared roadmap
Bug debt	Zero debt
100-page spec documents	PowerPoint specs
Private repositories	Open source or InnerSource
Deep organization hierarchy	Flattened organization hierarchy
Install numbers define success	User satisfaction defines success
Features ship once a year	Features ship every sprint

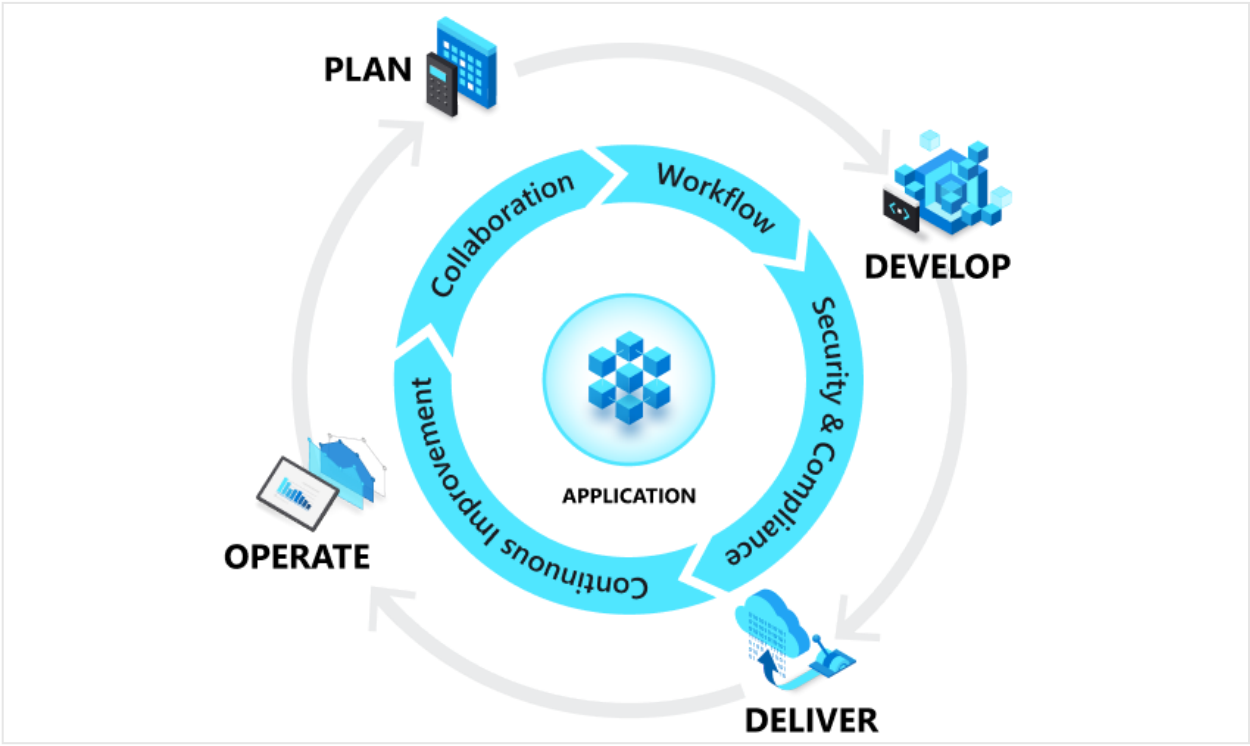
Key takeaways

- Take Agile science seriously, but don't be overly prescriptive. Agile can become too strict. Let the Agile mindset and culture grow.
- Celebrate results, not activity. Deploying functionality outweighs lines of code.
- Ship at every sprint to establish a rhythm and cadence and find all the work that needs to be done.
- Build the culture you want to get the behavior you're looking for.

Develop modern software with DevOps

Article • 11/28/2022

The development phase of DevOps is where all the core software development work happens. As input, it takes in plans for the current iteration, usually in the form of task assignments. Then it produces software artifacts that express the updated functionality. Development requires not only the tools that are used to write code, such as Visual Studio, but also supporting services like version control, issue management, and automated testing.



Select a development environment

Developers ideally spend most of their time in core development tasks, such as editing and debugging code. Having the right toolchain in place can make the difference between peak productivity and suboptimal performance. Integrated development environments (IDEs) have evolved beyond their humble beginnings as places to edit and compile code. Today, developers have the ability to perform nearly all their DevOps tasks from within a single user experience when they select the right development environment.

Manage code through version control and Git

As teams scale, the number of stakeholders who depend on and contribute to codebases can grow quickly. Without a strategy in place to manage changes to source code, development teams put themselves at significant risk of ongoing confusion, errors, and lost productivity. Implementing even the most basic version control can safeguard against those pitfalls. Most teams opt to use Git, the most popular version control system, to manage their code.

Automate processes

The real value of the development stage comes from the implementation of features. Unfortunately, there are many other tasks that sap time from the development team. Compiling code, running tests, and preparing output for deployment are a few examples. To minimize the impact, DevOps emphasizes automating these types of tasks through the practice of continuous integration.

Another time-consuming task in the development lifecycle is fixing bugs. While bugs are often seen as an inevitable part of software development, there are valuable steps any team can take to reduce them. Learn how to shift left to make testing faster and more reliable.

Next steps

Microsoft has been one of the world's largest software development companies for decades. Learn about how Microsoft develops in DevOps.

For a hands-on DevOps experience with continuous integration, see the following learning paths:

- Manage source control with GitHub
- Define and implement continuous integration with Azure DevOps
- Manage the lifecycle of your projects on GitHub

Select a development environment

Article • 11/28/2022

Select the right development environment to support DevOps adoption and performance. A DevOps development environment should not only edit and debug code, but integrate with the rest of the DevOps cycle, including testing, version control, and production monitoring. Microsoft provides two major development environments to support DevOps, Visual Studio and Visual Studio Code.

Use Visual Studio

Visual Studio is a full-featured integrated development environment (IDE). If you can use it, Visual Studio is ideal for working in Windows to build software for various platforms, including .NET or .NET Core, iOS, Android via Xamarin, and targets that support C++.

Visual Studio historically offers DevOps productivity and integration benefits. Visual Studio natively integrates with GitHub and Azure DevOps, and has a robust ecosystem of extensions for every industry DevOps provider.

Use Visual Studio Code

Visual Studio Code is a free, streamlined code editor that offers unlimited customization through tens of thousands of commercial and community extensions. These extensions add support for virtually any language, platform, and DevOps service. Developers can be productive on Windows, Mac, or Linux. Visual Studio Code is the ideal option for developers who can't use Visual Studio.

Develop for Azure

There's no particular preferred development environment for Azure solutions. Thanks to broad support for all major application platforms, you can use virtually any tool to build Azure solutions, and select the deployment model that works best for you. The best way to deploy solutions to production is usually through automation hosted in GitHub Actions or Azure Pipelines.

Both Visual Studio and Visual Studio Code have native features and first-party extensions that simplify working with DevOps processes in Azure, GitHub, and Azure DevOps.

Next steps

Learn to prepare Visual Studio, Visual Studio Code, Eclipse for Java, and IntelliJ IDEA for Azure development in the hands-on learning module Prepare your development environment for Azure development.

What is version control?

Article • 11/28/2022

Version control systems are software that help track changes make in code over time. As a developer edits code, the version control system takes a snapshot of the files. It then saves that snapshot permanently so it can be recalled later if needed.

Without version control, developers are tempted to keep multiple copies of code on their computer. This is dangerous because it's easy to change or delete a file in the wrong copy of code, potentially losing work. Version control systems solve this problem by managing all versions of the code, but presenting the team with a single version at a time.

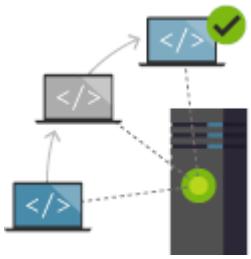
Why version control matters

There are plenty of things that can take up time as a developer. Reproducing bugs, learning new tools, and adding new features or content are just a few examples. As the demands of users scale up, version control helps teams work together and ship on time.

Benefits of version control

Version control benefits many aspects of production.

Create workflows



Version control workflows prevent the chaos of everyone using their own development process with different and incompatible tools. Version control systems provide process enforcement and permissions so everyone stays on the same page.

Work with versions

Every version has a description for what the changes in the version do, such as fix a bug or add a feature. These descriptions help the team follow changes in code by version instead of by individual file changes. Code stored in versions can be viewed and restored from version control at any time as needed. Versions make it easy to base new work off any version of code.

Code together

Version control synchronizes versions and makes sure that changes don't conflict with changes from others. The team relies on version control to help resolve and prevent conflicts, even when people make changes at the same time.

Keep a history

Version control keeps a history of changes as the team saves new versions of code. Team members can review history to find out who, why, and when changes were made. History gives teams the confidence to experiment since it's easy to roll back to a previous good version at any time. History lets anyone base work from any version of code, such as to fix a bug in a previous release.

Automate tasks

Version control automation features save time and generate consistent results. Automate testing, code analysis, and deployment when new versions are saved to version control are three examples.

Next steps

Learn more about the worldwide standard in version control, Git.

What is Git?

Article • 11/28/2022

Git has become the worldwide standard for version control. So what exactly is it?

Git is a distributed version control system, which means that a local clone of the project is a complete version control repository. These fully functional local repositories make it easy to work offline or remotely. Developers commit their work locally, and then sync their copy of the repository with the copy on the server. This paradigm differs from centralized version control where clients must synchronize code with a server before creating new versions of code.

Git's flexibility and popularity make it a great choice for any team. Many developers and college graduates already know how to use Git. Git's user community has created resources to train developers and Git's popularity make it easy to get help when needed. Nearly every development environment has Git support and Git command line tools implemented on every major operating system.

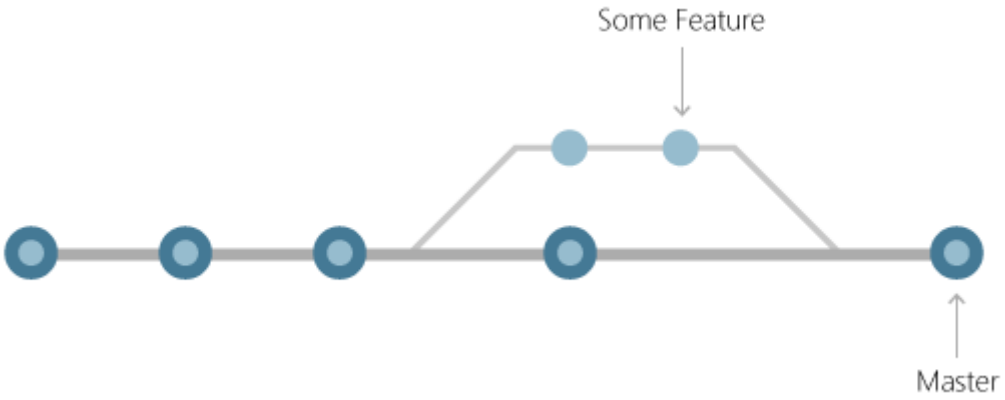
Git basics

Every time work is saved, Git creates a commit. A commit is a snapshot of all files at a point in time. If a file hasn't changed from one commit to the next, Git uses the previously stored file. This design differs from other systems that store an initial version of a file and keep a record of deltas over time.

Commits create links to other commits, forming a graph of the development history. It's possible to revert code to a previous commit, inspect how files changed from one commit to the next, and review information such as where and when changes were made. Commits are identified in Git by a unique cryptographic hash of the contents of the commit. Because everything is hashed, it's impossible to make changes, lose information, or corrupt files without Git detecting it.

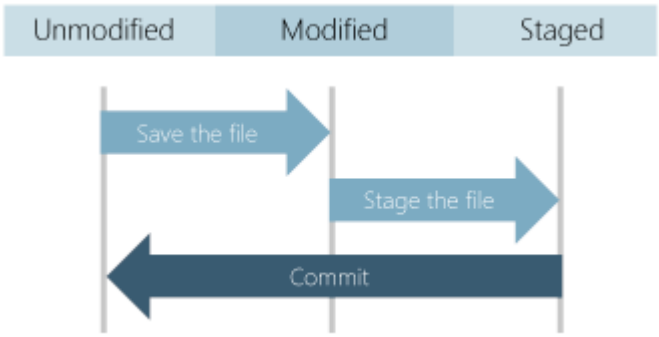
Branches

Each developer saves changes to their own local code repository. As a result, there can be many different changes based off the same commit. Git provides tools for isolating changes and later merging them back together. Branches, which are lightweight pointers to work in progress, manage this separation. Once work created in a branch is finished, it can be merged back into the team's main (or trunk) branch.



Files and commits

Files in Git are in one of three states: modified, staged, or committed. When a file is first modified, the changes exist only in the working directory. They aren't yet part of a commit or the development history. The developer must *stage* the changed files to be included in the commit. The staging area contains all changes to include in the next commit. Once the developer is happy with the staged files, the files are packaged as a *commit* with a message describing what changed. This commit becomes part of the development history.



Staging lets developers pick which file changes to save in a commit in order to break down large changes into a series of smaller commits. By reducing the scope of commits, it's easier to review the commit history to find specific file changes.

Benefits of Git

The benefits of Git are many.

Simultaneous development

Everyone has their own local copy of code and can work simultaneously on their own branches. Git works offline since almost every operation is local.

Faster releases

Branches allow for flexible and simultaneous development. The main branch contains stable, high-quality code from which you release. Feature branches contain work in progress, which are merged into the main branch upon completion. By separating the release branch from development in progress, it's easier to manage stable code and ship updates more quickly.

Built-in integration

Due to its popularity, Git integrates into most tools and products. Every major IDE has built-in Git support, and many tools support continuous integration, continuous deployment, automated testing, work item tracking, metrics, and reporting feature integration with Git. This integration simplifies the day-to-day workflow.

Strong community support

Git is open-source and has become the de facto standard for version control. There is no shortage of tools and resources available for teams to leverage. The volume of community support for Git compared to other version control systems makes it easy to get help when needed.

Git works with any team

Using Git with a source code management tool increases a team's productivity by encouraging collaboration, enforcing policies, automating processes, and improving visibility and traceability of work. The team can settle on individual tools for version control, work item tracking, and continuous integration and deployment. Or, they can choose a solution like GitHub or Azure DevOps that supports all of these tasks in one place.

Pull requests

Use pull requests to discuss code changes with the team before merging them into the main branch. The discussions in pull requests are invaluable to ensuring code quality

and increase knowledge across your team. Platforms like GitHub and Azure DevOps offer a rich pull request experience where developers can browse file changes, leave comments, inspect commits, view builds, and vote to approve the code.

Branch policies

Teams can configure GitHub and Azure DevOps to enforce consistent workflows and process across the team. They can set up branch policies to ensure that pull requests meet requirements before completion. Branch policies protect important branches by preventing direct pushes, requiring reviewers, and ensuring clean builds.

Next steps

Install and set up Git

Install and set up Git

Article • 11/28/2022

Git isn't *yet* a default option on computers, so it must be manually installed and configured. And like other software, it's important to keep Git up to date. Updates protect from security vulnerabilities, fix bugs, and provide access to new features.

The following sections describe how to install and maintain Git for the three major platforms.

Install Git for Windows

Download and install Git for Windows. Once installed, Git is available from the command prompt or PowerShell. It's recommended that you select the defaults during installation unless there's good reason to change them.

Git for Windows doesn't automatically update. To update Git for Windows, download the new version of the installer, which updates Git for Windows in place and retains all settings.

Install Git for macOS

macOS 10.9 (Mavericks) and higher installs Git the first time you attempt to run it from the Terminal. While this method is an easy way to get Git on a system, it doesn't allow for control over how often updates or security fixes are applied.

Instead, it's recommended that you install Git through Homebrew and that you use Homebrew tools to keep Git up to date. Homebrew is a great way to install and manage open source development tools on a Mac from the command line.

Install Homebrew and run the following to install the latest version of Git on a Mac:

```
> brew install git
```

To update the Git install, use Homebrew's upgrade option:

```
> brew upgrade git
```

A graphical installer for Git on macOS is also available from the official Git website.

Install Git for Linux

Use the Linux distribution's native package management system to install and update Git. For example, on Ubuntu:

```
> sudo apt-get install git
```

Configure Git on Linux

Set up the name and email address before starting to work with Git. Git attaches this information to changes and lets others identify which changes belong to which authors.

Run the following commands from the command prompt after installing Git to configure this information:

```
> git config --global user.name "<First_name> <Last_name>"
```

```
> git config --global user.email "<user_email_address>"
```

Visual Studio offers a great out-of-the-box Git experience without any extra tooling. Learn more in this Visual Studio Git tutorial.

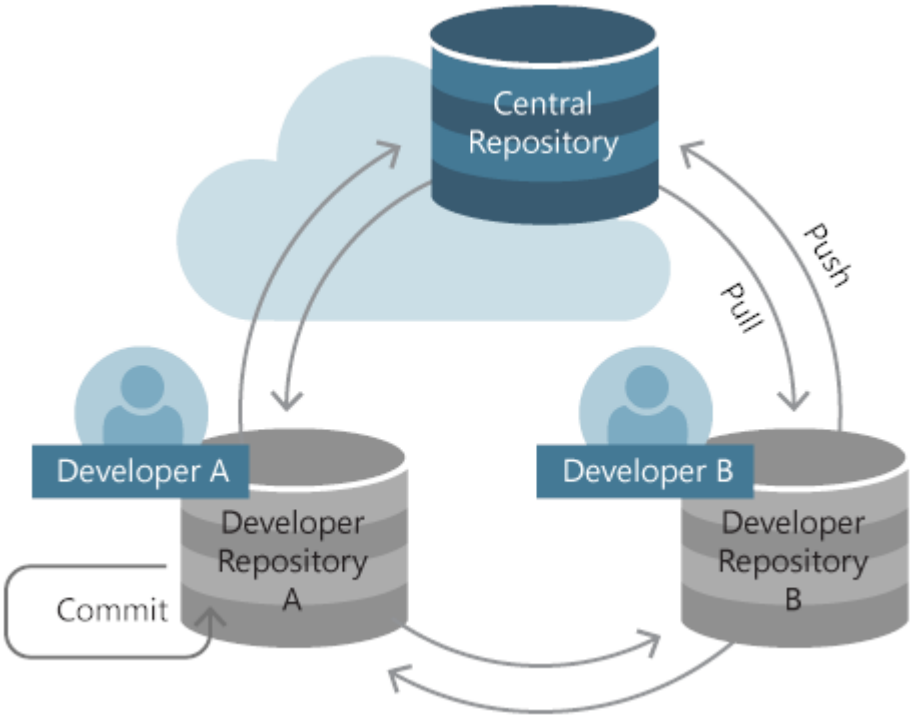
Set up a Git repository

Article • 11/28/2022

A Git repository, or repo, is a folder that Git tracks changes in. There can be any number of repos on a computer, each stored in their own folder. Each Git repo on a system is independent, so changes saved in one Git repo don't affect the contents of another.

A Git repo contains every version of every file saved in the repo. This is different than other version control systems that store only the differences between files. Git stores the file versions in a hidden .git folder alongside other information it needs to manage code. Git saves these files very efficiently, so having a large number of versions doesn't mean that it uses a lot of disk space. Storing each version of a file helps Git merge code better and makes working with multiple versions of code quick and easy.

Developers work with Git through commands issued while working in a local repo on the computer. Even when sharing code or getting updates from the team, it's done from commands that update the local repo. This local-focused design is what makes Git a distributed version control system. Every repo is self-contained, and the owner of the repo is responsible for keeping it up to date with the changes from others.



Most teams use a central repo hosted on a server that everyone can access to coordinate their changes. The central repo is usually hosted in a source control management solution, like GitHub or Azure DevOps. A source control management solution adds features and makes working together easier.

Create a new Git repo

You have two options to create a Git repo. You can create one from the code in a folder on a computer, or clone one from an existing repo. If working with code that's just on the local computer, create a local repo using the code in that folder. But most of the time the code is already shared in a Git repo, so cloning the existing repo to the local computer is the recommended way to go.

Create a new repo from existing code

Use the `git init` command to create a new repo from an existing folder on the computer. From the command line, navigate to the root folder containing the code and run:

```
> git init
```

to create the repo. Next, add any files in the folder to the first commit using the following commands:

```
> git add --all
```

```
> git commit -m "Initial commit"
```

Create a new repo from a remote repository

Use the `git clone` command to copy the contents of an existing repo to a folder on the computer. From the command line, navigate to the folder to contain the cloned repo, then run:

```
> git clone
https://<fabrikam.visualstudio.com/DefaultCollection/Fabrikam/_git/FabrikamProject>
```

Be sure to use the actual URL to the existing repo instead of the placeholder URL shown in this example. This URL, called the clone URL, points to a server where the team coordinates changes. Get this URL from the team, or from the clone button on the site where the repo is hosted.

It's not necessary to add files or create an initial commit when the repo is cloned since it was all copied, along with history, from the existing repo during the clone operation.

Next steps

GitHub and Azure Repos provide unlimited free public and private Git repos.

Visual Studio user? Learn more about how to create and clone repos from Visual Studio in this Git tutorial.

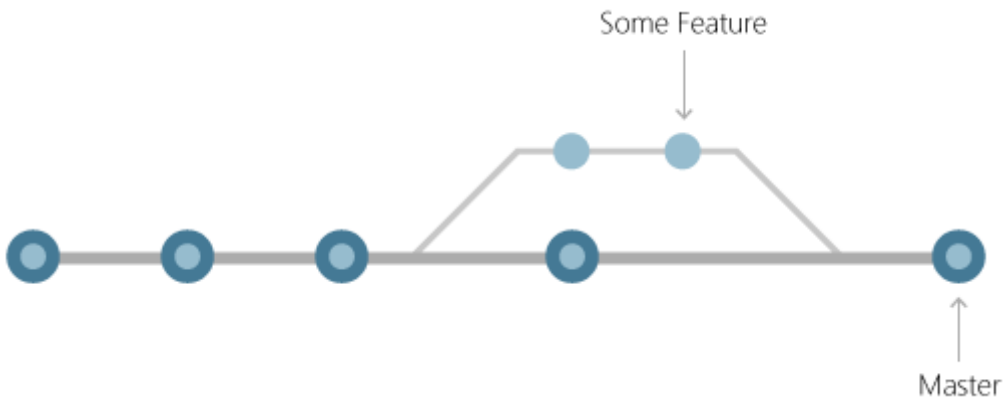
Save and share code with Git

Article • 11/28/2022

Saving and sharing versions of code with a team are the most common things done when using version control. Git has an easy three-step workflow for these tasks:

1. Create a new branch for work
2. Commit changes
3. Push the branch to share it with the team

Git makes it easy to manage work using branches. Every bugfix, new feature, added test, and updated configuration starts with a new branch. Branches are lightweight and local to the development machine, so you don't have to worry about using resources or coordinating changes with others until it's time to push the branch.



Branches enable you to code in isolation from other changes in development. Once everything's working, the branch and its changes are shared with your team. Others can experiment with the code in their own copy of the branch without it affecting work in progress in their own branches.

Create a branch

Create a branch based off the code in a current branch, such as `main`, when starting new work. It's a good practice to check which branch is selected using `git status` before creating a new branch.

Create branches in Git using the `git branch` command:

```
> git branch <branchname>
```

The command to swap between branches in the repo is `git checkout`. After creating the branch, switch to it before saving changes.

```
> git checkout <branchname>
```

Git has a shorthand command to both create the branch and switch to it at the same time:

```
> git checkout -b <branchname>
```

Learn more about working with Git branches in GitHub or Azure DevOps.

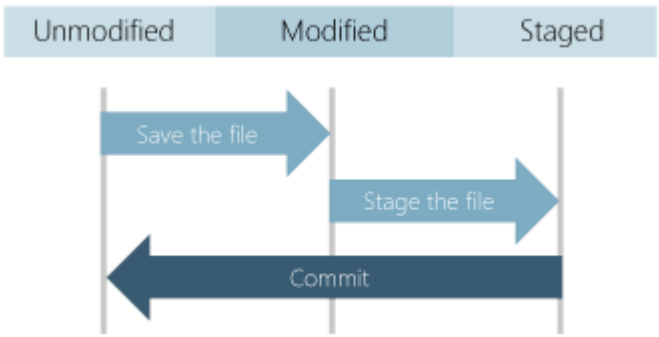
Save changes

Git doesn't automatically snapshot code as edits are made. Git must be told exactly which changes to add to the next snapshot. This is called *staging*. After staging your changes, create a *commit* to save the snapshot permanently.

Stage changes

Git tracks file changes made in the repo as they happen. It separates these changes into three categories:

- *Unmodified files* haven't been changed since the last commit.
- *Modified files* have changes since the last commit, but haven't been staged for the next commit.
- *Staged files* have changes that will be added to the next commit.



When you create a commit, only the staged changes and unchanged files are used for the snapshot. Unstaged changes are kept on the filesystem, but the commit uses the unmodified file in its snapshot.

Commit changes

Save changes in Git by creating a commit. Each commit stores the full file contents of the repo in each commit, not just individual file changes. This behavior is different than other version control systems that store the file-level differences from the last version of the code. Full file histories let Git make better decisions when merging changes and it makes switching between branches of code lightning fast.

Stage changes with `git add` to add changed files, `git rm` to remove files, and `git mv` to move files. Then, use `git commit` command to create the commit.

Usually, developers want to stage all changed files in the repo:

```
> git add –all
```

Then, commit the changes with a short description:

```
> git commit -m "Short description of changes."
```

Every commit has a message that describes its changes. A good commit message helps the developer remember the changes they made in a commit. Good commit messages also makes it easier for others to review the commit.

Learn more about staging files and committing changes in Visual Studio or Visual Studio Code.

Share changes

Whether working on a team or just wanting to back up their own code, developers need to share commits with a repo on another computer. Use the `git push` command to take commits from the local repo and write them into a remote repo. Git is set up in cloned repos to connect to the source of the clone, also known as `origin`. Run `git push` to write the local commits on your current branch to another branch (*branchname*) on this *origin* repository. Git creates *branchname* on the remote repo if it doesn't exist.

```
> git push origin
```

If working in a repo created on the local system with `git init`, you'll need to set up a connection to the team's Git server before changes can be pushed. Learn more about setting up remotes and pushing changes in Visual Studio or Visual Studio Code.

Share branches

Pushing a local branch to the team's shared repo makes its changes accessible to the rest of the team. The first time `git push` is run, adding the `-u` option tells Git to start tracking the local branch to *branchname* from the `origin` repo. After this one-time setup of tracking information, team members can use `git push` directly to share updates quickly and easily.

```
> git push origin <branchname>
```

Next steps

Learn more about branches in GitHub or Azure DevOps.

Learn more about pushing commits and branches in Visual Studio or Visual Studio Code.

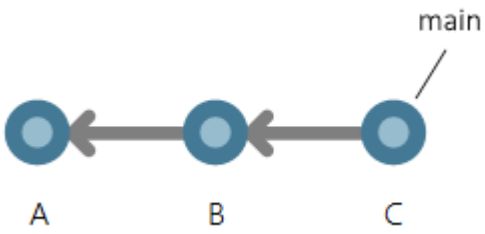
Understand Git history

Article • 11/28/2022

Git represents history in a fundamentally different way than centralized version controls systems (CVCS) such as Team Foundation Version Control, Perforce, or Subversion. Centralized systems store a separate history for each file in a repository. Git stores history as a graph of snapshots of the entire repository. These snapshots, called *commits* in Git, can have multiple parents, creating a history that looks like a graph instead of a straight line. This difference in history is incredibly important and is the main reason users familiar with CVCS find Git confusing.

Commit history basics

Start with a simple history example: a repo with three linear commits.

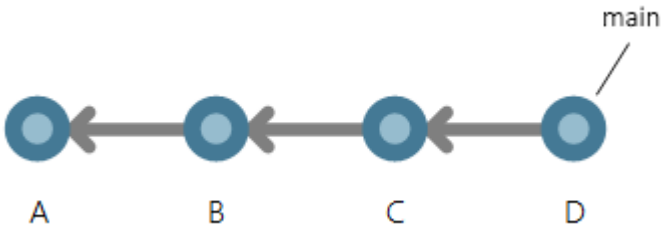


Commit A is the parent of commit B, and commit B is the parent of commit C.
This history looks very similar to a CVCS. The arrow pointing to commit C is a branch. Branches are pointers to specific commits, which is why branching is so lightweight and easy in Git.

A key difference in Git compared to CVCS is that the developer has their own full copy of the repo. They need to keep their local repository in sync with the remote repository by getting the latest commits from the remote repository. To do this, they pull the main branch with the following command:

```
git pull origin main
```

This merges all changes from the main branch in the remote repository, which Git names `origin` by default. This pull brought one new commit and the main branch in the local repo moves to that commit.



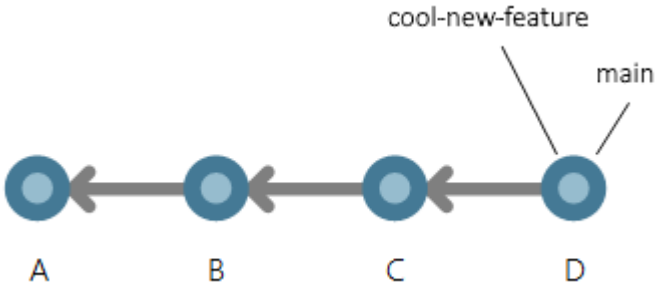
Understand branch history

Now it's time to make a change to the code. It's common to have multiple active branches when working on different features in parallel. This is in stark contrast to CVCS where new branches are heavy and rarely created. The first step is to checkout to a new branch using the following command:

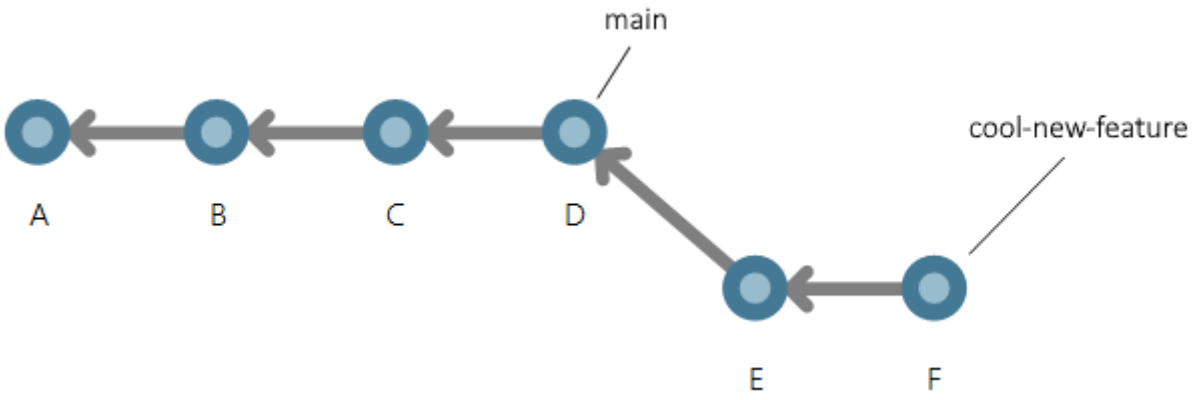
```
git checkout -b cool-new-feature
```

This is a shortcut combining two commands:

- `git branch cool-new-feature` to create the branch
- `git checkout cool-new-feature` to begin working in the branch



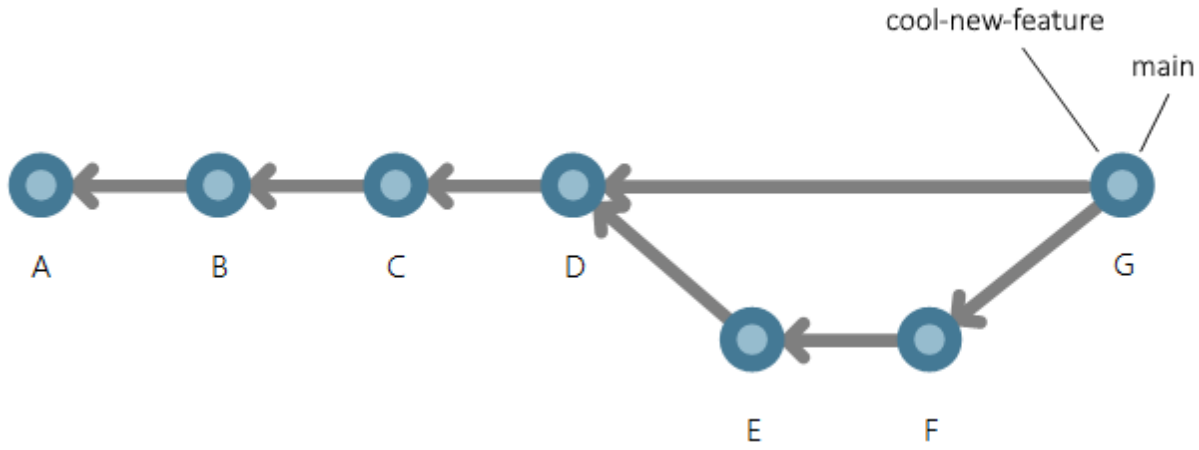
Two branches now point to the same commit. Suppose there are a few changes on the `cool-new-feature` branch in two new commits, E and F.



The commits are reachable by the `cool-new-feature` branch since they were committed to that branch. Now that the feature is done, it needs to be merged into the main

branch. To do that, use the following command:

```
git merge cool-feature main
```



The graph structure of history becomes visible when there's a merge. Git creates a new commit when the branch is merged into another branch. This is a merge commit. There aren't any changes included this merge commit since there were no conflicts. If there were conflicts, the merge commit would include the changes needed to resolve them.

History in the real world

Here's an example of Git history that more closely resembles code in active development on a team. There are three people who merge commits from their own branches into the `main` branch around the same time.

```
λ git log --oneline --graph --color --all --decorate
*   04b26ba (HEAD -> main) Merge feature3
|\
| * ae59408 (feature3) Commit G
| * 854dc3e Commit F
* |   1da0602 Merge feature1
|\ \
| |/
|/|
| * f0525d5 (feature1) Commit B
| * d6237f5 Commit A
* | 1c2bf32 (feature2) Commit E
* | 9ab6898 Commit D
* | fc6a971 Commit C
|/
* 729eccd Initial commit
```

Next steps

Learn more about working with Git history in GitHub and Azure Repos or Git log history simplification.

Get feedback with pull requests

Article • 11/28/2022

Pull requests support reviewing and merging code into a single collaborative process. Once a developer adds a feature or a bug fix, they create a pull request to begin the process of merging the changes into the upstream branch. Other team members are then given a chance to review and approve the code before it's finalized. Use pull requests to review works in progress and get early feedback on changes. But there's no commitment to merge the changes. An owner can abandon a pull request at any time.

Get code reviewed

The code review done as part of a pull request isn't just to find obvious bugs; that's what tests are for. A good code review catches less-obvious problems that could lead to costly issues later.

Code reviews help protect the team from bad merges and broken builds that sap the team's productivity. Reviews catch problems before the merge, protecting important branches from unwanted changes.

Code reviews also encourage and strengthen collaboration and communication between developers. And the team gains a clear history of all changes made between the main branch and the feature branches.

Cross-pollinate expertise and spread problem-solving strategies by using a wide range of reviewers in code reviews. Diffusing skills and knowledge makes the team stronger and more resilient.

Give great feedback

High-quality reviews start with high-quality feedback. The keys to great feedback in a pull request are:

- Have the right people review the pull request.
- Make sure that reviewers know what the code does.
- Give actionable, constructive feedback.
- Reply to comments in a timely manner.

When you assign reviewers to a pull request, be sure to select the right set of reviewers. Reviewers should know how the code works, but also include developers working in other areas so they can share their ideas.

Provide a clear description of the changes and provide a build of the code that has the fix or feature working in it. Reviewers should make an effort to provide feedback on changes they don't agree with. Identify the issue and give specific suggestions on what could be done differently. This feedback has clear intent and is easy for the owner of the pull request to understand.

The pull request owner should reply to comments, accept suggestions, or explain why they decline to apply them. Some suggestions are good, but might be outside the scope of the pull request. Take these suggestions and create new work items and feature branches separate from the pull request to make those changes.

Protect branches with policies

There are a few critical branches in a repo that teams rely on always being in good shape, such as the `main` branch. Teams can require pull requests to make any changes on these branches with platforms like GitHub and Azure DevOps. Developers pushing changes directly to the protected branches will have their pushes rejected.

Add additional conditions to pull requests to enforce a higher level of code quality in key branches. A clean build of the merged code and approval from multiple reviewers are some extra requirements often employed to protect key branches.

Learn more

GitHub has extensive documentation on how to propose changes to your work with pull requests.

Read more about giving great feedback in code reviews and using pull request templates to provide guidance to your reviewers. Azure DevOps also offers a rich pull request experience that's easy to use and scales as needed.

Hosting Git repositories

Article • 11/28/2022

Git has quickly become the worldwide standard for version control. Millions of projects rely on Git for everyday collaboration needs. While the decentralized nature of Git provides substantial benefits, it's still necessary for teams to push their changes to a centralized Git repo in order to merge branches and provide a hub for other DevOps activities.

GitHub

By far, the world's leading host for Git projects is GitHub. GitHub provides much more than just Git hosting. GitHub has features that span the whole DevOps process, including a marketplace of partner products and services.

Learn the basics of GitHub in this hands-on lab.

Self-hosting GitHub

Some organizations might have regulatory or other requirements that prevent them from hosting their source code and other assets outside of their own infrastructure. For these users, GitHub Enterprise Server is available. GitHub Enterprise Server includes the familiar features and user experience, but can be entirely hosted within a company's own infrastructure.

Set up a trial of GitHub Enterprise Server.

Azure Repos

Users already on Azure DevOps or earlier versions of Team Foundation Server have a first-class option in migrating to Azure Repos. Azure Repos provides all the benefits of Git, combined with a familiar user experience and integration points.

Learn the basics of working with Git on Azure Repos.

Self-hosting Azure Repos

Teams that need to keep their source code and other assets within their own infrastructure can use Azure DevOps Server to enjoy all the benefits of Azure Repos.

Download the latest release of Azure DevOps Server.

Migrate to Git from centralized version control

Article • 11/28/2022

Migrating a team to Git from centralized version control requires more than just learning new commands. To support distributed development, Git stores file history and branch information differently than a centralized version control system. Planning and implementing a successful migration to Git from a centralized version control system requires understanding these fundamental differences.

Microsoft has helped migrate many internal teams and customers from centralized version control systems to Git. This experience has produced the following guidance based on practices that consistently succeed.

Steps for successful migration

For a successful migration, teams should:

- Evaluate current tools and processes.
- Select a Git branching strategy.
- Decide whether and how to migrate history.
- Maintain the previous version control system.
- Remove binary files, executables, and tools from source control.
- Train teams in Git concepts and practices.
- Test the migration to Git.

Evaluate current tools and processes

Changing version control systems naturally disrupts the development workflow with new tools and practices. This disruption can be an opportunity to improve other aspects of the DevOps process.

Teams should consider adopting the following practices as they migrate to the new system:

- Continuous integration (CI), where every check-in triggers a build and test pass. CI helps identify defects early and provides a strong safety net for projects.
- *Required code reviews* before checking in code. In the Git branching model, pull request code review is part of the development process. Code reviews complement

the CI workflow.

- Continuous delivery (CD) to automate deployment processes. Changing version control tools requires deployment process changes, so a migration is a good time to adopt a modern release pipeline.

Select a Git branching strategy

Before migrating code, the team should select a branching strategy.

In Git, 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.

Git discourages long-lived, isolated feature branches, which tend to delay merges until integration becomes difficult. By using modern CD techniques like feature flags, teams can integrate code into the main branch quickly, but still keep in-progress features hidden from users until they're complete.

Teams that currently use a long-lived feature branch strategy can adopt feature flags before migrating to Git. Using feature flags simplifies migration by minimizing the number of branches to migrate. Whether they use feature branches or feature flags, teams should document the mapping between legacy branches and new Git branches, so everyone understands where to commit their new work.

Decide whether to migrate history

Teams might be tempted to migrate their existing source code history to Git. Several tools claim to migrate a complete history of all branches from a centralized tool to Git. A Git commit appears to map relatively well to the changeset or check-in model that the previous version control tool used.

However, this mapping has some serious limitations.

- In most centralized version control systems, branches exist as folders in the repository. For example, the main branch might be a folder named */trunk*, and other branches are folders like */branch/one* and */branch/two*. In a Git repository, branches include the entire repository, so a 1:1 translation is difficult.

- In some version control systems, a *tag* or *label* is a collection that can contain various files in the tree, even files at different versions. In Git, a *tag* is a snapshot of

the entire repository at a specific point in time. A tag can't 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, including fine-grained change types like rename, undelete, and rollback. Git stores versions as snapshots of the entire repository, and metadata about the way files changed isn't available.

These differences mean that a full history migration will be lossy at best, and possibly misleading. Given the lossiness, the effort involved, and the relative rarity of using history, it's recommended that most teams avoid importing history. Instead, teams should do a *tip migration*, bringing only a snapshot of the most recent branch version into Git. For most teams, time is best spent on areas of the migration that have a higher return on investment, such as improving processes.

Maintain the old version control system

During and after a migration, developers might still need access to the previous version control history. Although the previous version control history becomes less relevant over time, it's still important to be able to refer to it. Highly regulated environments might have specific legal and auditing requirements for version control history.

Especially for teams that do only a tip migration, it's highly recommended to maintain the previous system indefinitely. Set the old version control system to read-only after you migrate.

Large development teams and regulated environments can place *breadcrumbs* in Git that point back to the old version control system. A simple example is a text file added as the first commit at the root of a Git repository, before the tip migration, that points to the URL of the old version control server. If many branches migrate, a text file in each branch should explain how the branches migrated from the old system. Breadcrumbs are also helpful for developers who start working on a project after it's been migrated and aren't familiar with the old version control system.

Remove binary files and tools

Git's storage model is optimized for versioning text files and source code, which are compact and highly compressible. Binary files are usually large, and once they're added to a repository, they remain in the repository history and in every future clone. Because of the way Git stores history, developers should avoid adding binary files to repositories,

especially binaries that are very large or that change often. Migrating to Git is an opportunity to remove these binaries from the codebase.

It's also recommended to exclude libraries, tools, and build output from repositories. Instead, use package management systems like NuGet to manage dependencies.

Assets like icons and artwork might need to align with a specific version of source code. Small, infrequently-changed assets like icons won't bloat history, and you can include them directly in a repository. To store large or frequently-changing assets, use the Git Large File Storage (LFS) extension. For more information about managing large files in GitHub, see Managing large files. For Azure Repos, see Manage and store large files in Git.

Provide training

One of the biggest challenges in migrating to Git is helping developers understand how Git stores changes and how commits form development history. It's not enough to prepare a cheat sheet that maps old commands to Git commands. Developers need to stop thinking about version control history in terms of a centralized, linear model, and understand Git's history model and the commit graph.

People learn in different ways, so you should provide several types of training materials. Live, lab-based training with an expert instructor works well for some people. The Pro Git book is an excellent starting point that is available free online.

Available free hands-on training courses include:

- Introduction to version control with Git learning path.
- The Get started with Git in Azure Repos quickstart.
- GitHub's Git and GitHub learning resources.

Organizations should work to identify Git experts on teams, empower them to help others, and encourage other team members to ask them questions.

Test the migration

Once teams update their processes, analyze their code, and train their members, it's time to migrate the source code. Whether you do a tip migration or migrate history, it's important to do one or more test migrations into a test repository. Before you do a final migration, make sure:

- All code files migrate.

- All branches are available.
- There are no stray binaries in the repository.
- Users have the appropriate permissions to fetch and push.
- Builds are successful, and all tests pass.

Migrate the code

Do the final migration during nonwork hours, ideally between milestones when there's natural downtime. Migrating at the end of a sprint might cause issues while developers are trying to finish work. Try to migrate over a weekend, when nobody needs to check in.

Plan for a firm cutover from the old version control system to Git. Trying to operate multiple systems in parallel means developers might not know where or how to check in. Set the old version control system to read-only to help avoid confusion. Without this safeguard, a second migration that includes interim changes might be necessary.

The actual migration process varies depending on the system you're migrating from. For information about migrating from Team Foundation Version Control, see Migrate from TFVC to Git.

Migration checklist

Team workflows:

- ✓ Determine how builds will run.
- ✓ Decide when tests will run.
- ✓ Develop a release management process.
- ✓ Move code reviews to pull requests.

Branching strategy:

- ✓ Pick a Git branching strategy.
- ✓ Document the branching strategy, why it was selected, and how legacy branches map.

History:

- ✓ Decide how long to keep legacy version control running.
- ✓ Identify branches that 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 materials, and videos.
- ✓ Identify members of the team to serve as local Git experts.

Code migration:

- ✓ Do multiple test runs to ensure the migration will go smoothly.
- ✓ Identify and communicate a cutover time.
- ✓ Create the new Git repo.
- ✓ Set the old system to read-only.
- ✓ Migrate the main branch first, then any other needed branches.

Next steps

- Migrate to Azure DevOps from Team Foundation Server
- How TFVC commands and workflow map to Git

Import and migrate repositories from TFVC to Git

Article • 03/25/2024

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019

You can migrate code from an existing TFVC repository to a new Git repository within the same organization. Migrating to Git is an involved process for large TFVC repositories and teams. Centralized version control systems, like TFVC, behave differently from Git in fundamental ways. The switch involves a lot more than learning new commands. It is a disruptive change that requires careful planning. You need to think about:

- Revising tools and processes
- Removing binaries and executables
- Training your team

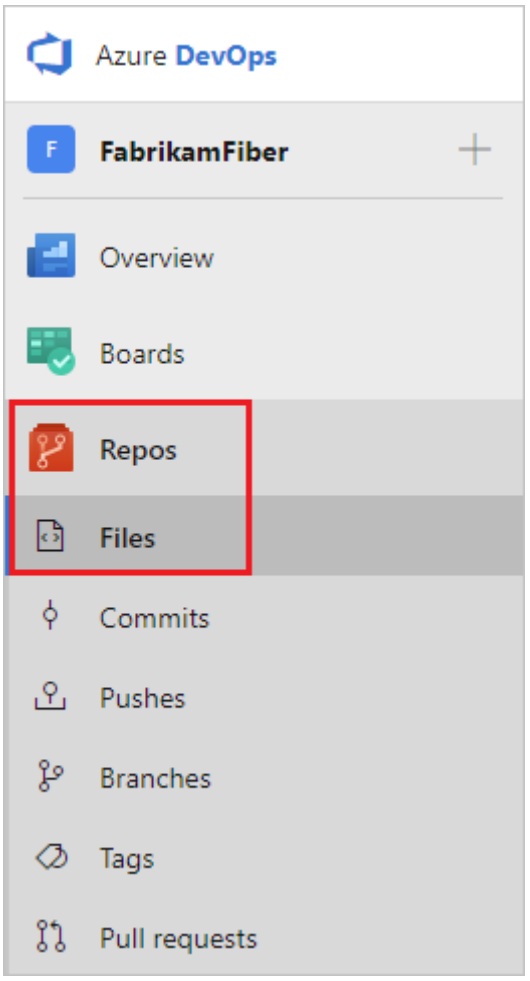
We strongly recommend reading Centralized version control to Git and the following Migrate from TFVC to Git section before starting the migration.

The import experience is great for small simple TFVC repositories. It's also good for repositories that have already been "cleaned up" as outlined in Centralized version control to Git and the following Migrate from TFVC to Git section. These sections also recommend other tools for more advanced TFVC repository configurations.

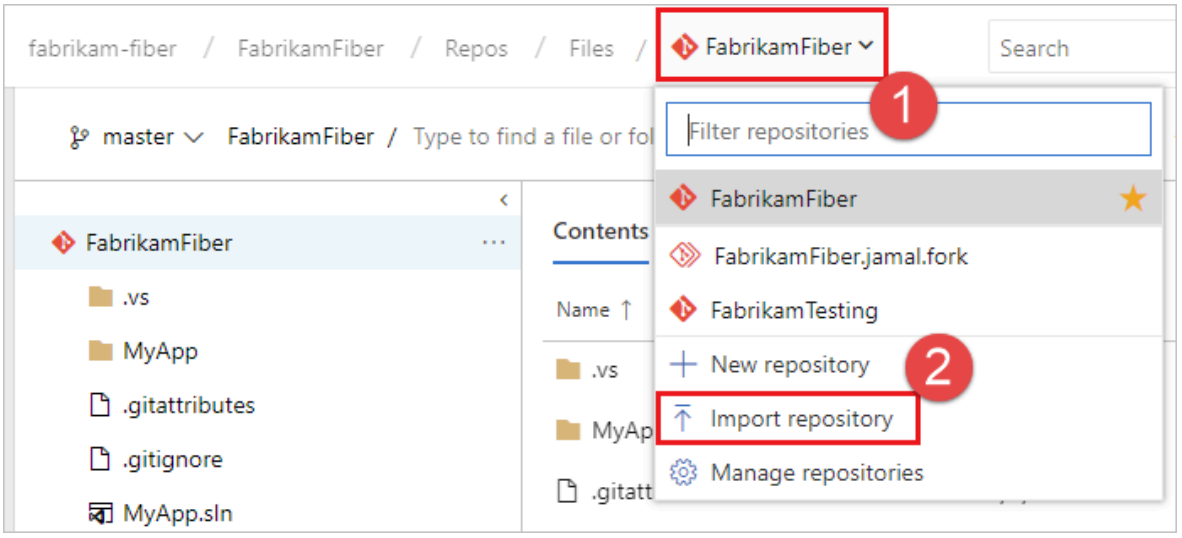
> ⓘ **Important**
>
> Due to the differences in how TFVC and Git store version control history, we recommend that you don't migrate your history. This is the approach that Microsoft took when it migrated Windows and other products from centralized version control to Git.

Importing the repository

1. Select **Repos**, **Files**.



2. From the repo drop-down, select **Import repository**.



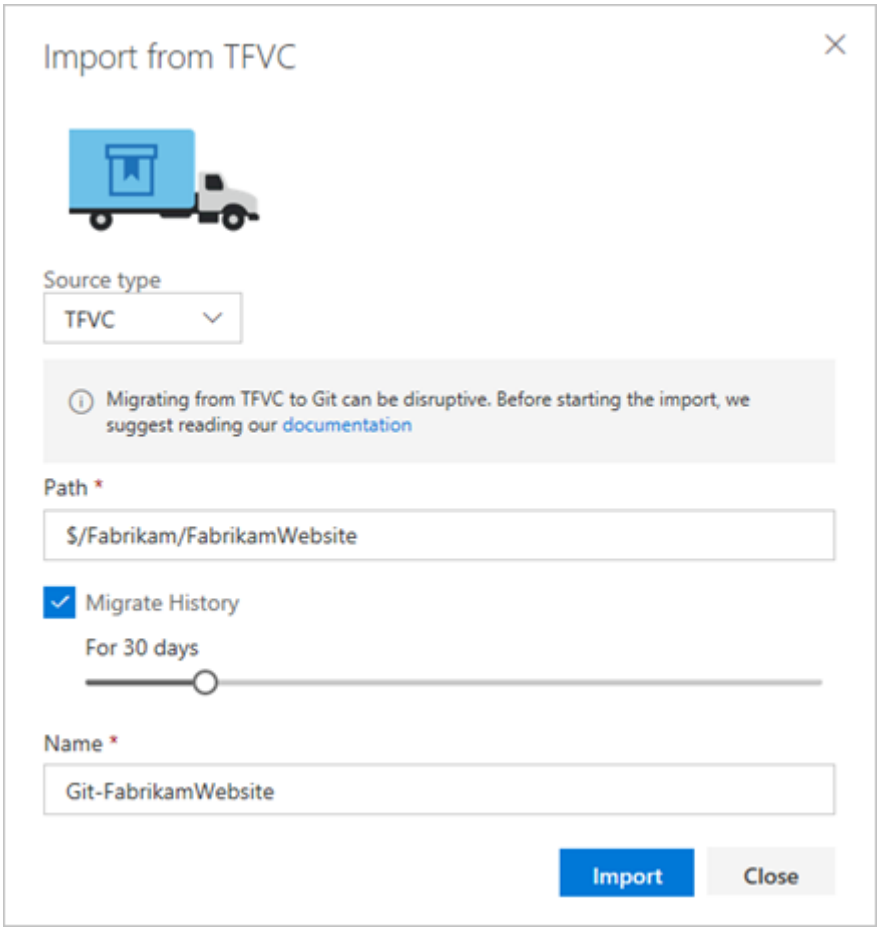
3. Select TFVC from the **Source type** dropdown

4. Type the path to the repository / branch / folder that you want to import to the Git repository. For example, `$/Fabrikam/FabrikamWebsite`

5. If you want to migrate history from the TFVC repository, click **Migrate history** and select the number of days. You can migrate up to 180 days of history starting from the most recent changeset. A link to the TFVC repository is added in the commit

message of the 1st changeset that is migrated to Git. This makes it easy to find older history when needed.

6. Give a name to the new Git repository and click **Import**. Depending on the size of the import, your Git repository would be ready in a few minutes.



Troubleshooting

This experience is optimized for small, simple TFVC repositories or repositories that have been prepared for a migration. This means it has a few limitations.

1. It only migrates the contents of root or a branch. For example, if you have a TFVC project at `$/Fabrikam` which has 1 branch and 1 folder under it, a path to import `$/Fabrikam` would import the folder while `$/Fabrikam/<branch>` would only import the branch.
2. The imported repository and associated history (if imported) cannot exceed 1GB in size.
3. You can import up to 180 days of history.

If any of the above is a blocker for your import, we recommend you try external tools like Git-TFS for importing and reading our whitepapers - Centralized version control to Git and the following Migrate from TFVC to Git section.

> **Important**
>
> The usage of external tools like Git-TFS with Microsoft products, services, or platforms is entirely the responsibility of the user. Microsoft does not endorse, support, or guarantee the functionality, reliability, or security of such third-party extensions.

Migrate from TFVC to Git

Before migrating source code from a centralized version control system to Git, understand the differences between the two and prepare for the migration.

- Requirements
- Steps to migrate
  - Check out the latest version
  - Remove binaries and build tools
  - Convert version control-specific configuration
  - Check in changes and perform the migration
  - Advanced migrations
- Update the workflow

Requirements

In order to make migrations easier, there are a number of requirements before following the importing the repository procedure in the previous section of this article.

- Migrate only a single branch. When planning the migration, choose a new branching strategy for Git. Migrating only the main branch supports a topic-branch based workflow like GitFlow or GitHub Flow.
- Do a tip migration, as in, import only the latest version of the source code. If TFVC history is simple, there's an option to migrate some history, up to 180 days, so that the team can work only out of Git. For more information, see Plan your migration to Git.
- Exclude binary assets like images, scientific data sets, or game models from the repository. These assets should use the Git LFS (Large File Storage) extension, which the import tool doesn't configure.
- Keep the imported repository below 1GB in size.

If the repository doesn't meet these requirements, use the Git-TFS tool to do your migration instead.

> **Important**
>
> The usage of external tools like Git-TFS with Microsoft products, services, or platforms is entirely the responsibility of the user. Microsoft does not endorse, support, or guarantee the functionality, reliability, or security of such third-party extensions.

Steps to migrate

The process to migrate from TFVC is generally straightforward:

1. Check out the latest version of the branch from TFVC on your local disk.
2. Remove binaries and build tools from the repository and set up a package management system like NuGet.
3. Convert version control-specific configuration directives. For example, convert `.tfignore` files to .gitignore, and convert `.tpattributes` files to .gitattributes.
4. Check in changes and perform the migration to Git.

Steps 1-3 are optional. If there aren't binaries in the repository and there's no need to set up a `.gitignore` or a `.gitattributes`, you can proceed directly to the Check in changes and perform the migration step.

Check out the latest version

Create a new TFS workspace and map a working folder for the server directory being migrated to Git. This doesn't require a full working folder mapping. Only map folders that contain binaries to be removed from the repository and folders that contain version control system-specific configuration files like `.tfignore`.

Once mappings are set up, get the folder locally:

```
prettyprint
```

```
tf get /version:T /recursive
```

Remove binaries and build tools

Due to the way Git stores the history of changed files by providing a copy of every file in history to every developer, checking in binary files directly to the repository causes the repo to grow quickly and can cause performance issues.

For build tools and dependencies like libraries, adopt a packaging solution with versioning support, such as NuGet. Many open source tools and libraries are already available on the NuGet Gallery, but for proprietary dependencies, create new NuGet packages.

Once dependencies are moved into NuGet, be sure that they aren't included in the Git repository by adding them to .gitignore.

Convert version control-specific configuration

Team Foundation Version Control provides a `.tfignore` file, which ensures that certain files aren't added to the TFVC repository. You can use the `.tfignore` file for automatically generated files like build output so that they aren't accidentally checked in.

If the project relies on this behavior, convert the `.tfignore` file to a .gitignore file.

Cross-platform TFVC clients also provide support for a `.tpattributes` file that controls how files are placed on the local disk or checked into the repository. If a `.tpattributes` file is in use, convert it to a .gitattributes file.

Check in changes and perform the migration

Check in any changes that remove binaries, migrate to package management, or convert version control-specific configuration. Once you make this final change in TFVC, you can do the import.

Follow the Importing the repository procedure to do the import.

Advanced migrations

The Git-TFS tool is a two-way bridge between Team Foundation Version Control and Git, and you can use it to perform a migration. Git-TFS is appropriate for a migration with full history, more than the 180 days that the Import tool supports. Or you can use Git-TFS to attempt a migration that includes multiple branches and merge relationships.

Before attempting a migration with Git-TFS, note that there are fundamental differences between the way TFVC and Git store history:

- Git stores history as a snapshot of the repository in time, while TFVC records the discrete operations that occurred on a file. Change types in TFVC like rename, undelete, and rollback can't be expressed in Git. Instead of seeing that file `A` was

renamed to file `B`, it only tracks that file `A` was deleted and file `B` was added in the same commit.

- Git doesn't have a direct analog of a TFVC label. Labels can contain any number of files at any specific version and can reflect files at different versions. Although conceptually similar, the Git tags point to a snapshot of the whole repository at a point in time. If the project relies on TFVC labels to know what was delivered, Git tags might not provide this information.
- Merges in TFVC occur at the file level, not at the entire repository. Only a subset of changed files can be merged from one branch to another. Remaining changed files might then be merged in a subsequent changeset. In Git, a merge affects the entire repository, and both sets of individual changes can't be seen as a merge.

Because of these differences, it's recommended that you do a tip migration and keep your TFVC repository online, but read-only, in order to view history.

To attempt an advanced migration with Git-TFS, see clone a single branch with history or clone all branches with merge history.

> **Important**
>
> The usage of external tools like **Git-TFS** with Microsoft products, services, or platforms is entirely the responsibility of the user. Microsoft does not endorse, support, or guarantee the functionality, reliability, or security of such third-party extensions.

Update the workflow

Moving from a centralized version control system to Git is more than just migrating code. The team needs training to understand how Git is different from the existing version control system and how these differences affect day-to-day work.

Learn more about how to migrate from centralized version control to Git.

Feedback

Use continuous integration

Article • 11/28/2022

Continuous integration (CI) is the process of automatically building and testing code every time a team member commits code changes to version control. A code commit to the main or trunk branch of a shared repository triggers the automated build system to build, test, and validate the full branch. CI encourages developers to share their code and unit tests by merging their changes into the shared version control repository every time they complete a task.

Software developers often work in isolation, and then need to integrate their changes with the rest of a team's code base. Waiting days or weeks to integrate code can create many merge conflicts, hard to fix bugs, diverging code strategies, and duplicated efforts. CI avoids these problems because it requires the development team's code to continuously merge to the shared version control branch.

CI keeps the main branch up-to-date. Developers can use modern version control systems like Git to isolate their work in short-lived feature branches. When the feature is complete, the developer submits a pull request from the feature branch to the main branch. On approval of the pull request, the changes merge into the main branch, and the feature branch can be deleted.

Development teams repeat this process for each work item. Teams can establish branch policies to ensure the main branch maintains desired quality criteria.

Build definitions specify that every commit to the main branch triggers the automated build and testing process. Automated tests verify that every build maintains consistent quality. CI catches bugs earlier in the development cycle, making them less expensive to fix.

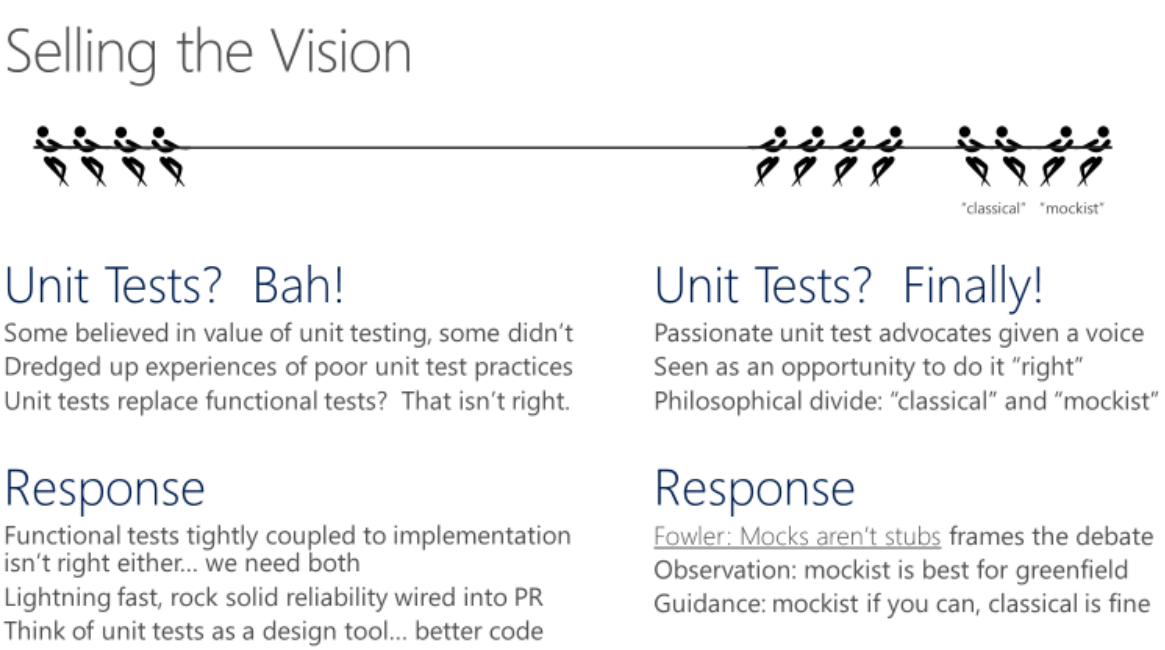
CI is a standard feature in modern DevOps platforms. GitHub users can implement CI through GitHub Actions. Azure DevOps users can use Azure Pipelines.

Shift testing left with unit tests

Article • 11/28/2022

Testing helps ensure that code performs as expected, but the time and effort to build tests takes time away from other tasks such as feature development. With this cost, it's important to extract maximum value from testing. This article discusses DevOps test principles, focusing on the value of unit testing and a shift-left test strategy.

Dedicated testers used to write most tests, and many product developers didn't learn to write unit tests. Writing tests can seem too difficult or like too much work. There can be skepticism about whether a unit test strategy works, bad experiences with poorly-written unit tests, or fear that unit tests will replace functional tests.



To implement a DevOps test strategy, be pragmatic and focus on building momentum. Although you can insist on unit tests for new code or existing code that can be cleanly refactored, it might make sense for a legacy codebase to allow some dependency. If significant parts of product code use SQL, allowing unit tests to take dependency on the SQL resource provider instead of *mocking* that layer could be a short-term approach to progress.

As DevOps organizations mature, it becomes easier for leadership to improve processes. While there might be some resistance to change, Agile organizations value changes that clearly pay dividends. It should be easy to sell the vision of faster test runs with fewer failures, because it means more time to invest in generating new value through feature development.

DevOps test taxonomy

Defining a test taxonomy is an important aspect of the DevOps testing process. A DevOps test taxonomy classifies individual tests by their dependencies and the time they take to run. Developers should understand the right types of tests to use in different scenarios, and which tests different parts of the process require. Most organizations categorize tests across four levels:

- **L0** and **L1** tests are *unit tests*, or tests that depend on code in the assembly under test and nothing else. L0 is a broad class of fast, in-memory unit tests.
- **L2** are *functional tests* that might require the assembly plus other dependencies, like SQL or the file system.
- **L3** functional tests run against testable service deployments. This test category requires a service deployment, but might use *stubs* for key service dependencies.
- **L4** tests are a restricted class of *integration tests* that run against production. L4 tests require a full product deployment.

While it would be ideal for all tests to run at all times, it's not feasible. Teams can select where in the DevOps process to run each test, and use *shift-left* or *shift-right* strategies to move different test types earlier or later in the process.

For example, the expectation might be that developers always run through L2 tests before committing, a pull request automatically fails if the L3 test run fails, and the deployment might be blocked if L4 tests fail. The specific rules may vary from organization to organization, but enforcing the expectations for all teams within an organization moves everyone toward the same quality vision goals.

Unit test guidelines

Set strict guidelines for L0 and L1 unit tests. These tests need to be very fast and reliable. For example, average execution time per L0 test in an assembly should be less than 60 milliseconds. The average execution time per L1 test in an assembly should be less than 400 milliseconds. No test at this level should exceed 2 seconds.

One Microsoft team runs over 60,000 unit tests in parallel in less than six minutes. Their goal is to reduce this time to less than a minute. The team tracks unit test execution time with tools like the following chart, and files bugs against tests that exceed the allowed time.

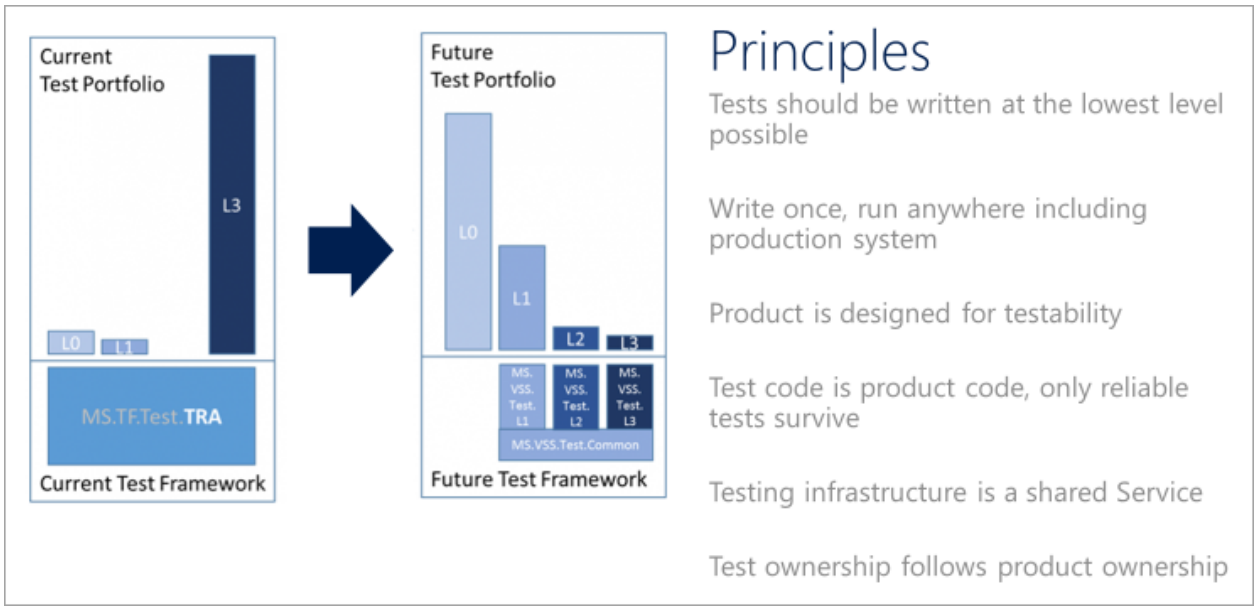
Functional test guidelines

Functional tests must be independent. The key concept for L2 tests is isolation. Properly isolated tests can run reliably in any sequence, because they have complete control over the environment they run in. The state must be known at the beginning of the test. If one test created data and left it in the database, it could corrupt the run of another test that relies on a different database state.

Legacy tests that need a user identity might have called external authentication providers to get the identity. This practice introduces several challenges. The external dependency could be unreliable or unavailable momentarily, breaking the test. This practice also violates the test isolation principle, because a test could change the state of an identity, such as permission, resulting in an unexpected default state for other tests. Consider preventing these issues by investing in identity support within the test framework.

DevOps test principles

To help transition a test portfolio to modern DevOps processes, articulate a quality vision. Teams should adhere to the following test principles when defining and implementing a DevOps testing strategy.

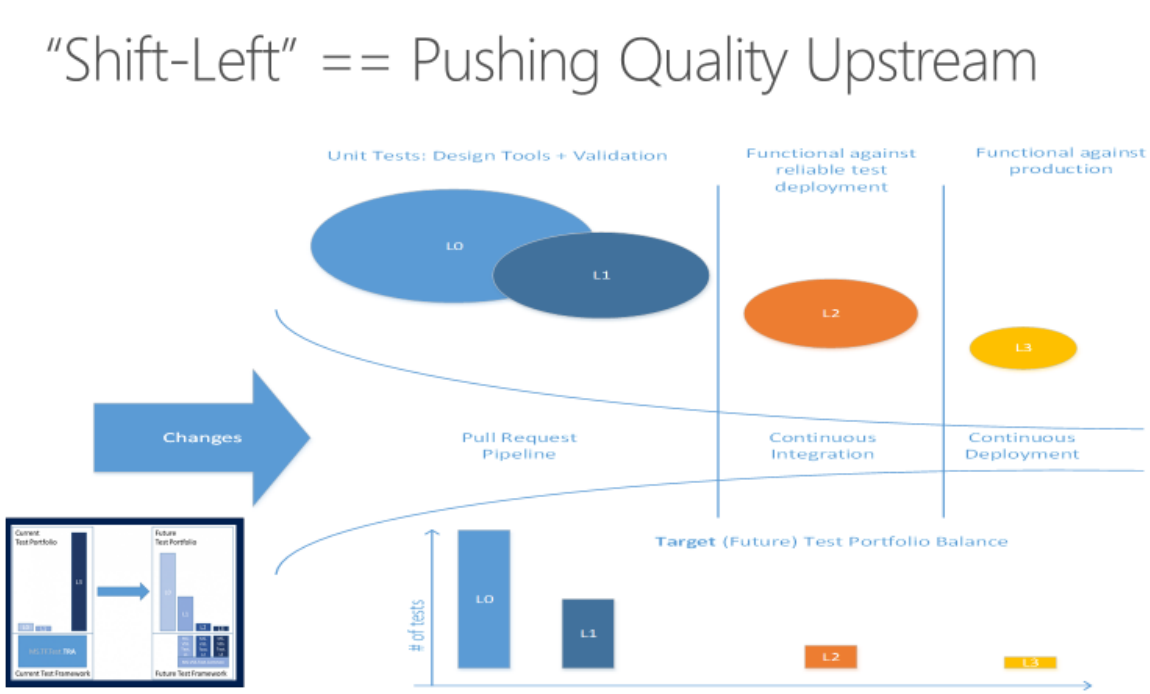


Shift left to test earlier

Tests can take a long time to run. As projects scale, test numbers and types grow substantially. When test suites grow to take hours or days to complete, they can push farther out until they run at the last moment. The code quality benefits of testing aren't realized until long after the code is committed.

Long-running tests might also produce failures that are time-consuming to investigate. Teams can build a tolerance for failures, especially early in sprints. This tolerance undermines the value of testing as insight into codebase quality. Long-running, last-minute tests also add unpredictability to end-of-sprint expectations, because an unknown amount of technical debt must be paid to get the code shippable.

The goal for shifting testing left is to move quality upstream by performing testing tasks earlier in the pipeline. Through a combination of test and process improvements, shifting left reduces both the time it takes for tests to run, and the impact of failures later in the cycle. Shifting left ensures that most testing is completed before a change merges into the main branch.



In addition to shifting certain testing responsibilities left to improve code quality, teams can shift other test aspects right, or later in the DevOps cycle, to improve the final product. For more information, see Shift right to test in production.

Write tests at the lowest possible level

Write more unit tests. Favor tests with the fewest external dependencies, and focus on running most tests as part of the build. Consider a parallel build system that can run unit tests for an assembly as soon as the assembly and associated tests drop. It's not feasible to test every aspect of a service at this level, but the principle is to use lighter unit tests if they can produce the same results as heavier functional tests.

Aim for test reliability

An unreliable test is organizationally expensive to maintain. Such a test works directly against the engineering efficiency goal by making it hard to make changes with confidence. Developers should be able to make changes anywhere and quickly gain confidence that nothing has been broken. Maintain a high bar for reliability. Discourage the use of UI tests, because they tend to be unreliable.

Write functional tests that can run anywhere

Tests might use specialized integration points designed specifically to enable testing. One reason for this practice is a lack of testability in the product itself. Unfortunately, tests like these often depend on internal knowledge and use implementation details that don't matter from a functional test perspective. These tests are limited to

environments that have the secrets and configuration necessary to run the tests, which generally excludes production deployments. Functional tests should use only the public API of the product.

Design products for testability

Organizations in a maturing DevOps process take a complete view of what it means to deliver a quality product on a cloud cadence. Shifting the balance strongly in favor of unit testing over functional testing requires teams to make design and implementation choices that support testability. There are different ideas about what constitutes well-designed and well-implemented code for testability, just as there are different coding styles. The principle is that designing for testability must become a primary part of the discussion about design and code quality.

Treat test code as product code

Explicitly stating that test code is product code makes it clear that the quality of test code is as important to shipping as that of product code. Teams should treat test code the same way they treat product code, and apply the same level of care to the design and implementation of tests and test frameworks. This effort is similar to managing configuration and infrastructure as code. To be complete, a code review should consider the test code and hold it to the same quality bar as the product code.

Use shared test infrastructure

Lower the bar for using test infrastructure to generate trusted quality signals. View testing as a shared service for the entire team. Store unit test code alongside product code and build it with the product. Tests that run as part of the build process must also run under development tools such as Azure DevOps. If tests can run in every environment from local development through production, they have the same reliability as the product code.

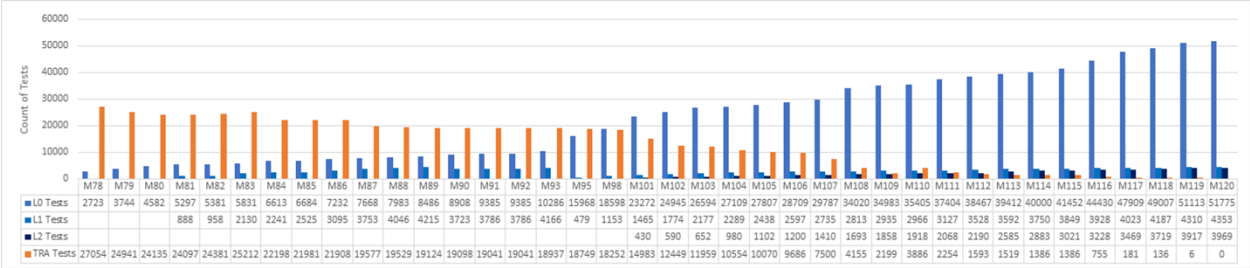
Make code owners responsible for testing

Test code should reside next to the product code in a repo. For code to be tested at a component boundary, push accountability for testing to the person writing the component code. Don't rely on others to test the component.

Case study: Shift left with unit tests

A Microsoft team decided to replace their legacy test suites with modern, DevOps unit tests and a shift-left process. The team tracked progress across triweekly sprints, as shown in the following graph. The graph covers sprints 78-120, which represents 42 sprints over 126 weeks, or about two and half years of effort.

The team started at 27K legacy tests in sprint 78, and reached zero legacy tests at S120. A set of L0 and L1 unit tests replaced most of the old functional tests. New L2 tests replaced some of the tests, and many of the old tests were deleted.



	M78	M79	M80	M81	M82	M83	M84	M85	M86	M87	M88	M89	M90	M91	M92	M93	M95	M98	M101	M102	M103	M104	M105	M106	M107	M108	M109	M110	M111	M112	M113	M114	M115	M116	M117	M118	M119	M120
L0 Tests	2723	3744	4582	5297	5381	5831	6613	6684	7232	7668	7983	8486	8908	9385	9385	10286	15968	18598	23272	24945	26594	27109	27807	28709	29787	34020	34983	35405	37404	38467	39412	40000	41452	44430	47909	49007	51113	51775
L1 Tests				888	958	2130	2241	2525	3095	3753	4046	4215	3723	3786	3786	4166	479	1153	1465	1774	2177	2289	2438	2597	2735	2813	2935	2966	3127	3528	3592	3750	3849	3928	4023	4187	4310	4353
L2 Tests																			430	590	652	980	1102	1200	1410	1693	1858	1918	2068	2190	2585	2883	3021	3228	3469	3719	3917	3969
TRA Tests	27054	24941	24135	24097	24381	25212	22198	21981	21908	19577	19529	19124	19098	19041	19041	18937	18749	18252	14983	12449	11959	10554	10070	9686	7500	4155	2199	3886	2254	1593	1519	1386	1386	755	181	136	6	0

In a software journey that takes over two years to complete, there's a lot to learn from the process itself. Overall, the effort to completely redo the test system over two years was a massive investment. Not every feature team did the work at the same time. Many teams across the organization invested time in every sprint, and in some sprints it was most of what the team did. Although it's difficult to measure the cost of the shift, it was a non-negotiable requirement for the team's quality and performance goals.

Getting started

At the beginning, the team left the old functional tests, called TRA tests, alone. The team wanted developers to buy into the idea of writing unit tests, particularly for new features. The focus was on making it as easy as possible to author L0 and L1 tests. The team needed to develop that capability first, and build momentum.

The preceding graph shows unit test count starting to increase early, as the team saw the benefit of authoring unit tests. Unit tests were easier to maintain, faster to run, and had fewer failures. It was easy to gain support for running all unit tests in the pull request flow.

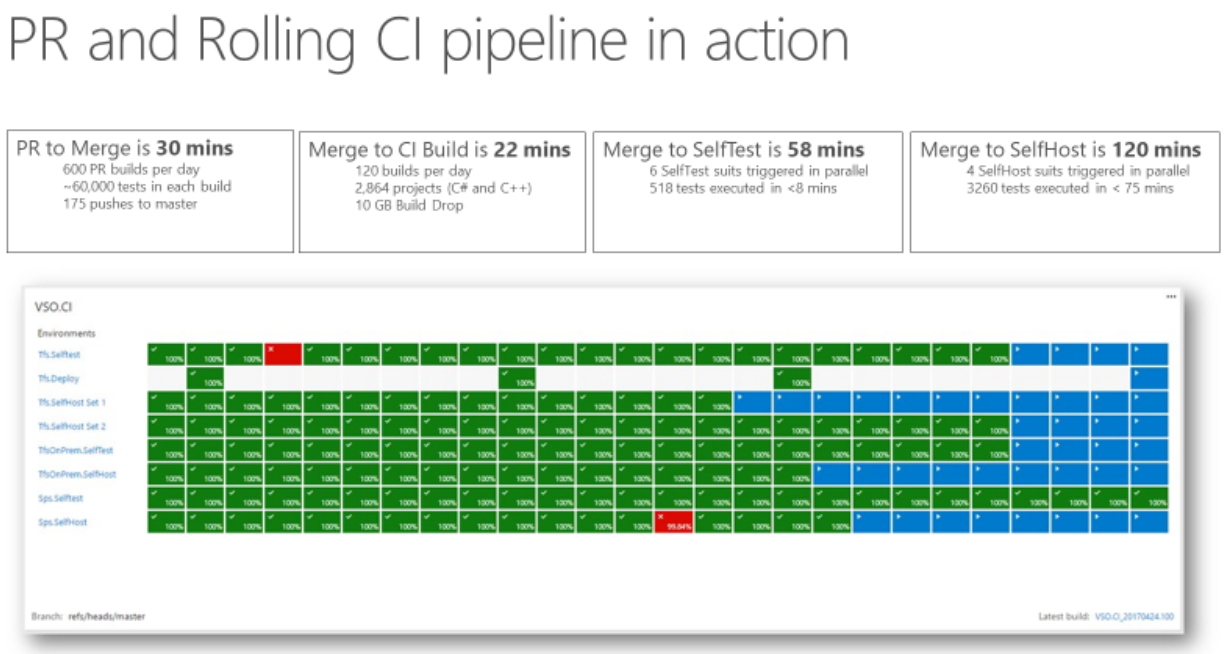
The team didn't focus on writing new L2 tests until sprint 101. In the meantime, the TRA test count went down from 27,000 to 14,000 from Sprint 78 to Sprint 101. New unit tests replaced some of the TRA tests, but many were simply deleted, based on team analysis of their usefulness.

The TRA tests jumped from 2100 to 3800 in sprint 110 because more tests were discovered in the source tree and added to the graph. It turned out that the tests had

always been running, but weren't being tracked properly. This wasn't a crisis, but it was important to be honest and reassess as needed.

Getting faster

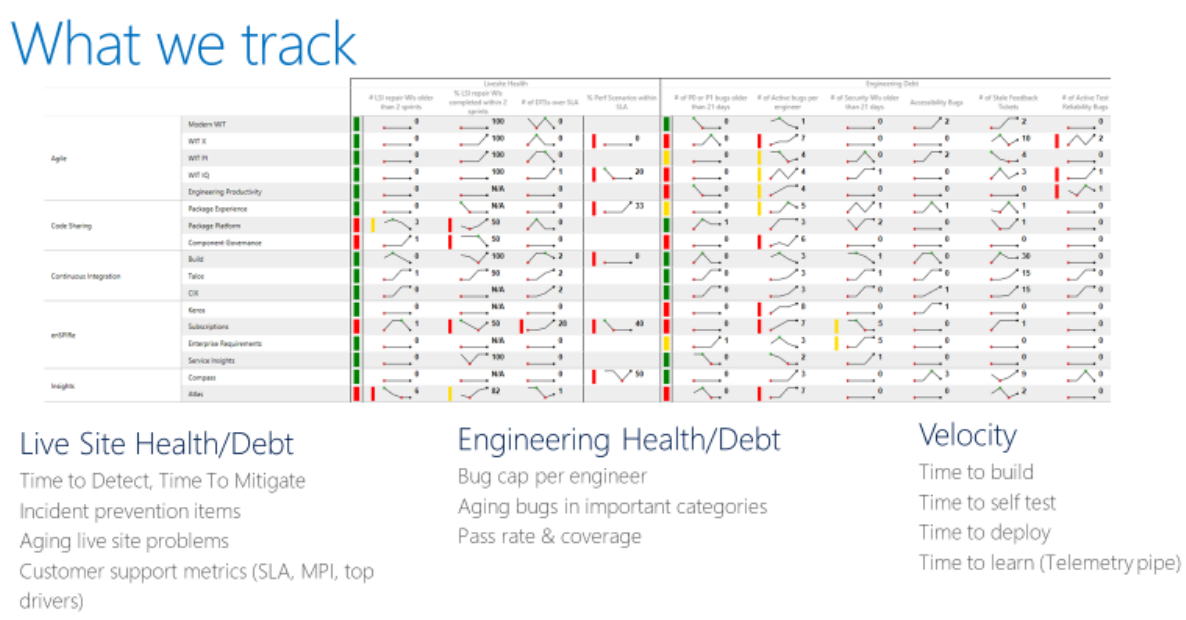
Once the team had a continuous integration (CI) signal that was extremely fast and reliable, it became a trusted indicator for product quality. The following screenshot shows the pull request and CI pipeline in action, and the time it takes to go through various phases.



It takes around 30 minutes to go from pull request to merge, which includes running 60,000 unit tests. From code merge to CI build is about 22 minutes. The first quality signal from CI, SelfTest, comes after about an hour. Then, most of the product is tested with the proposed change. Within two hours from Merge to SelfHost, the entire product is tested and the change is ready to go into production.

Using metrics

The team tracks a scorecard like the following example. At a high level, the scorecard tracks two types of metrics: Health or debt, and velocity.



For live site health metrics, the team tracks the time to detect, time to mitigate, and how many repair items a team is carrying. A repair item is work the team identifies in a live site retrospective to prevent similar incidents from recurring. The scorecard also tracks whether teams are closing the repair items within a reasonable timeframe.

For engineering health metrics, the team tracks active bugs per developer. If a team has more than five bugs per developer, the team must prioritize fixing those bugs before new feature development. The team also tracks aging bugs in special categories like security.

Engineering velocity metrics measure speed in different parts of the continuous integration and continuous delivery (CI/CD) pipeline. The overall goal is to increase the velocity of the DevOps pipeline: Starting from an idea, getting the code into production, and receiving data back from customers.

Next steps

- Learning path: Build applications with Azure DevOps
- Use continuous integration
- Shift right to test in production
- Mocks Aren't Stubs

How Microsoft develops with DevOps

Article • 11/28/2022

Microsoft strives to use *One Engineering System* to build and deploy all Microsoft products with a solid DevOps process centered on a Git branching and release flow. This article highlights practical implementation, how the system scales from small services to massive platform development needs, and lessons learned from using the system across various Microsoft teams.

Adopting a standardized development process is an ambitious undertaking. The requirements of different Microsoft organizations vary greatly, and requirements of different teams within organizations scale with size and complexity. To address these varied needs, Microsoft uses a trunk-based branching strategy to help develop products quickly, deploy them regularly, and deliver changes safely to production.

Microsoft also uses platform engineering principles as part of its *One Engineering System*.

Microsoft release flow

Every organization should settle on a standard code release process to ensure consistency across teams. The Microsoft release flow incorporates DevOps processes from development to release. The basic steps of the release flow consist of branch, push, pull request, and merge.

Branch

To fix a bug or implement a feature, a developer creates a new branch off the main integration branch. The Git lightweight branching model creates these short-lived *topic* branches for every code contribution. Developers commit early and avoid long-running feature branches by using feature flags.

Push

When the developer is ready to integrate and ship changes to the rest of the team, they push their local branch to a branch on the server, and open a pull request. Repositories with several hundred developers working in many branches use a naming convention for server branches to alleviate confusion and *branch proliferation*. Developers usually create branches named `users/<username>/feature`, where `<username>` is their account name.

Pull request

Pull requests control topic branch merges into the main branch and ensure that branch policies are satisfied. The pull request process builds the proposed changes and runs a quick test pass. The first- and second-level test suites run around 60,000 tests in less than five minutes. This isn't the complete Microsoft test matrix, but is enough to quickly give confidence in pull requests.

Next, other members of the team review the code and approve the changes. Code review picks up where the automated tests left off, and is particularly useful for spotting architectural problems. Manual code reviews ensure that other engineers on the team have visibility into the changes and that code quality remains high.

Merge

Once the pull request satisfies all build policies and reviewers have signed off, the topic branch merges into the main integration branch, and the pull request is complete.

After merge, other acceptance tests run that take more time to complete. These traditional post-checkin tests do a more thorough validation. This testing process provides a good balance between having fast tests during pull request review and having complete test coverage before release.

Differences from GitHub Flow

GitHub Flow is a popular trunk-based development release flow for organizations to implement a scalable approach to Git. However, some organizations find that as their needs grow, they must diverge from parts of the GitHub Flow.

For example, an often overlooked part of GitHub Flow is that pull requests must deploy to production for testing before they can merge to the main branch. This process means that all pull requests wait in the deployment queue for merge.

Some teams have several hundred developers working constantly in a single repository, who can complete over 200 pull requests into the main branch per day. If each pull requests requires a deployment to multiple Azure data centers across the globe for testing, developers spend time waiting for branches to merge, instead of writing software.

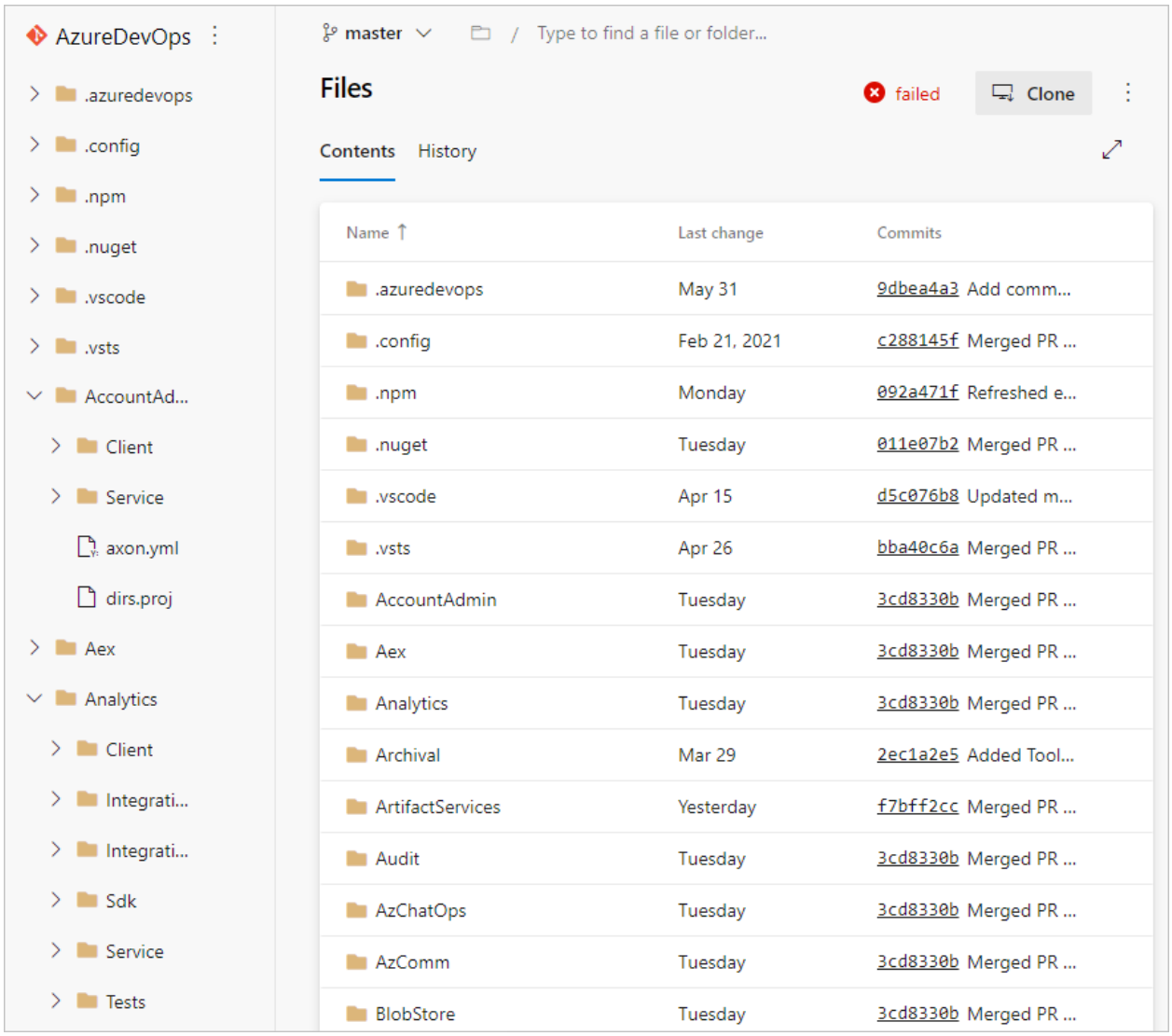
Instead, Microsoft teams continue developing in the main branch and batch up deployments into timed releases, usually aligned with a three-week sprint cadence.

Implementation details

Here are some key implementation details of the Microsoft release flow:

Git repository strategy

Different teams have different strategies for managing their Git repositories. Some teams keep the majority of their code in one Git repository. Code is broken up into components, each in its own root-level folder. Large components, especially older components, may have multiple subcomponents that have separate subfolders within the parent component.



Adjunct repositories

Some teams also manage adjunct repositories. For instance, build and release agents and tasks, the VS Code extension, and open-source projects are developed on GitHub. Configuration changes check in to a separate repository. Other packages that the team depends on come from other places and are consumed via NuGet.

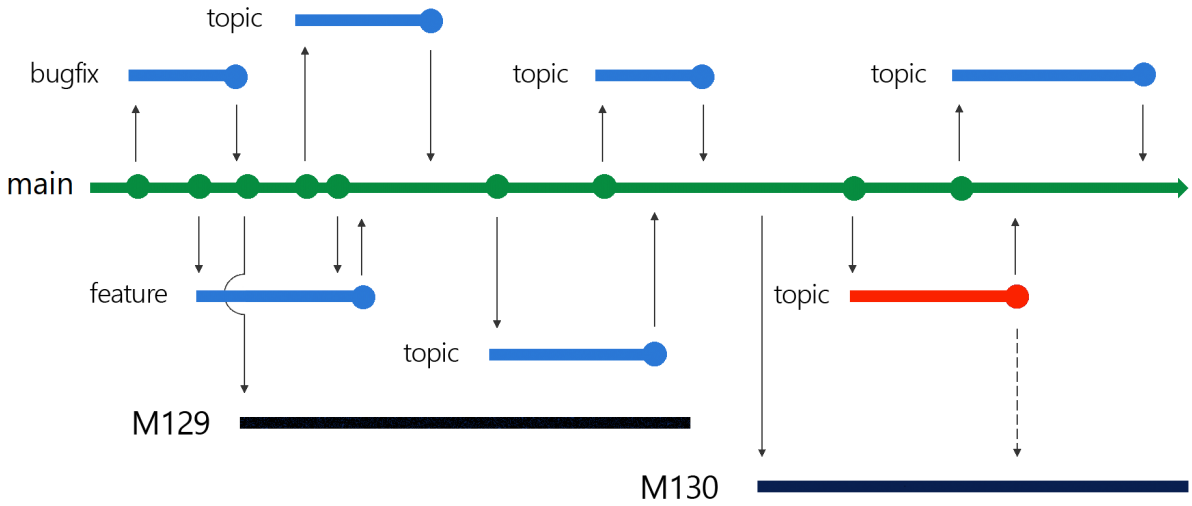
Mono repo or multi-repo

While some teams elect to have a single monolithic repository, the *mono-repo*, other Microsoft products use a *multi-repo* approach. Skype, for instance, has hundreds of small repositories that stitch together in various combinations to create many different clients, services, and tools. Especially for teams that embrace microservices, multi-repo can be the right approach. Usually, older products that began as monoliths find a mono-repo approach to be the easiest transition to Git, and their code organization reflects that.

Release branches

The Microsoft release flow keeps the main branch buildable at all times. Developers work in short-lived topic branches that merge to `main`. When a team is ready to ship, whether at the end of a sprint or for a major update, they start a new release branch off the main branch. Release branches never merge back to the main branch, so they might require *cherry-picking* important changes.

The following diagram shows short-lived branches in blue and release branches in black. One branch with a commit that needs cherry-picking appears in red.

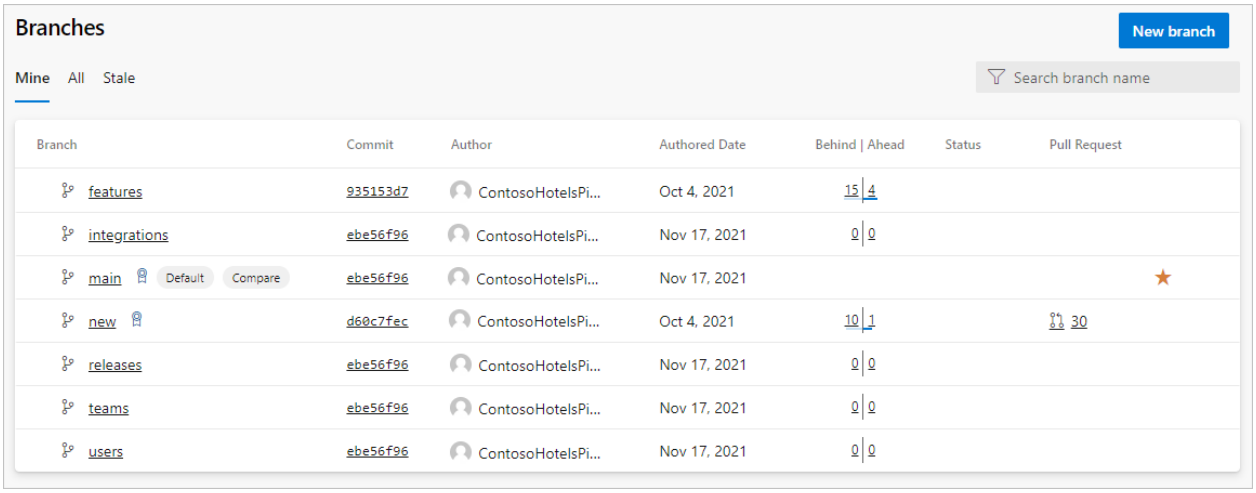


Branch policies and permissions

Git branch policies help enforce the release branch structure and keep the main branch clean. For example, branch policies can prevent direct pushes to the main branch.

To keep branch hierarchy tidy, teams use permissions to block branch creation at the root level of the hierarchy. In the following example, 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.



Git repository workflow

Within the repository and branch structure, developers do their daily work. Working environments vary heavily by team and by individual. Some developers prefer the command line, others like Visual Studio, and others work on different platforms. The structures and policies in place on Microsoft repositories ensure a solid and consistent foundation.

A typical workflow involves the following common tasks:

Build a new feature

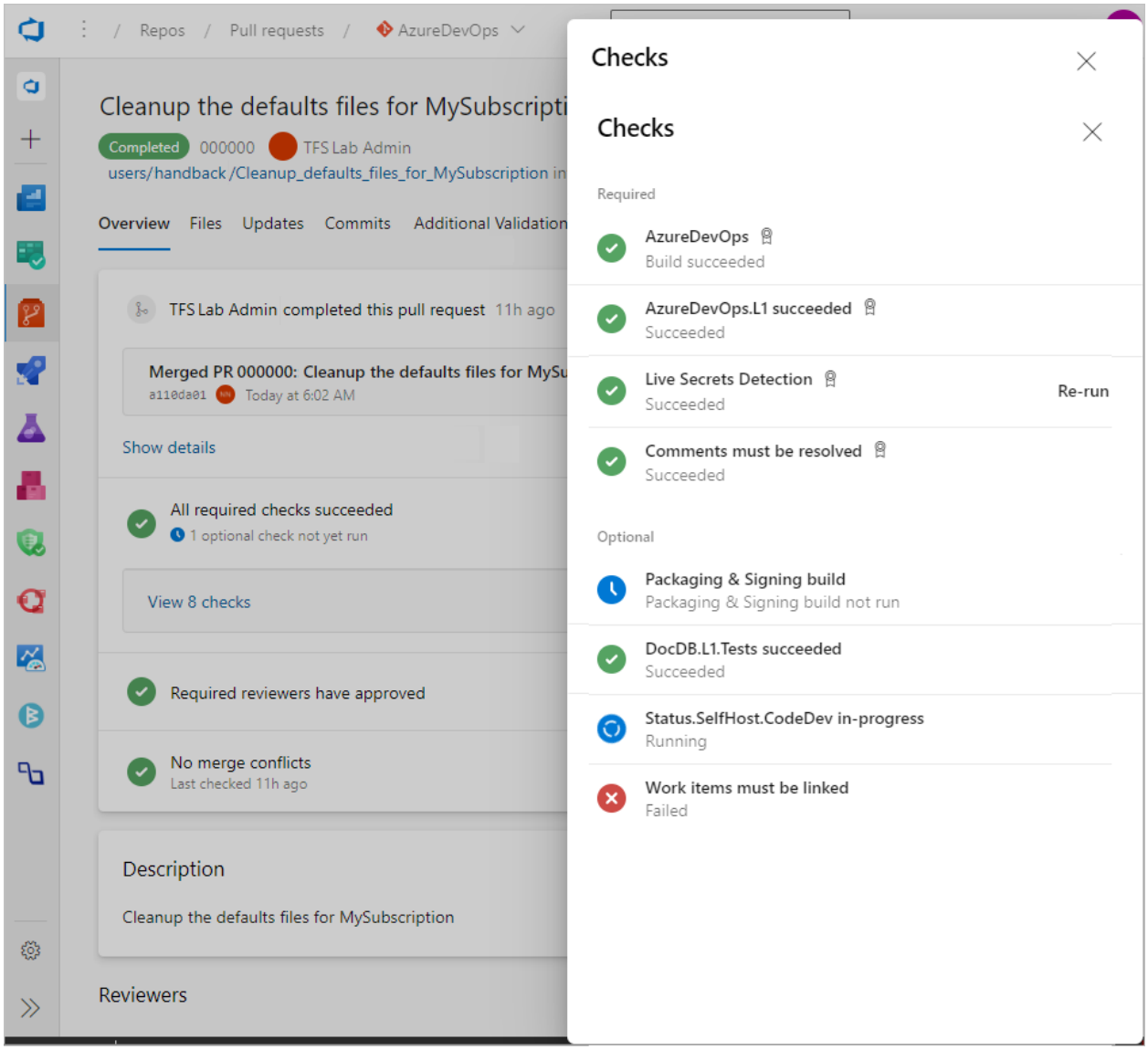
Building a new feature is the core of a software developer's job. Non-Git parts of the process include looking at telemetry data, coming up with a design and a spec, and writing the actual code. Then, the developer starts working with the repository by syncing to the latest commit on `main`. The main branch is always buildable, so it's guaranteed to be a good starting point. The developer checks out a new feature branch, makes code changes, commits, pushes to the server, and starts a new pull request.

Use branch policies and checks

Upon creation of a pull request, automated systems check that the new code builds, doesn't break anything, and doesn't violate any security or compliance policies. This process doesn't block other work from happening in parallel.

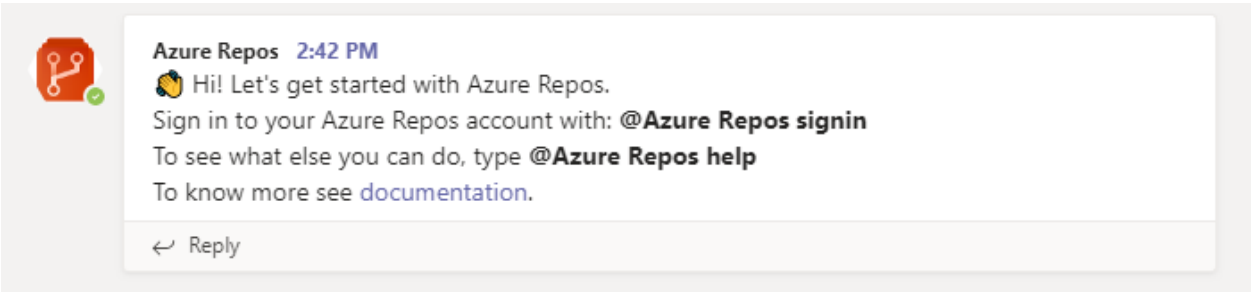
Branch policies and checks can require a successful build including passed tests, signoff by the owners of any code touched, and several external checks to verify corporate

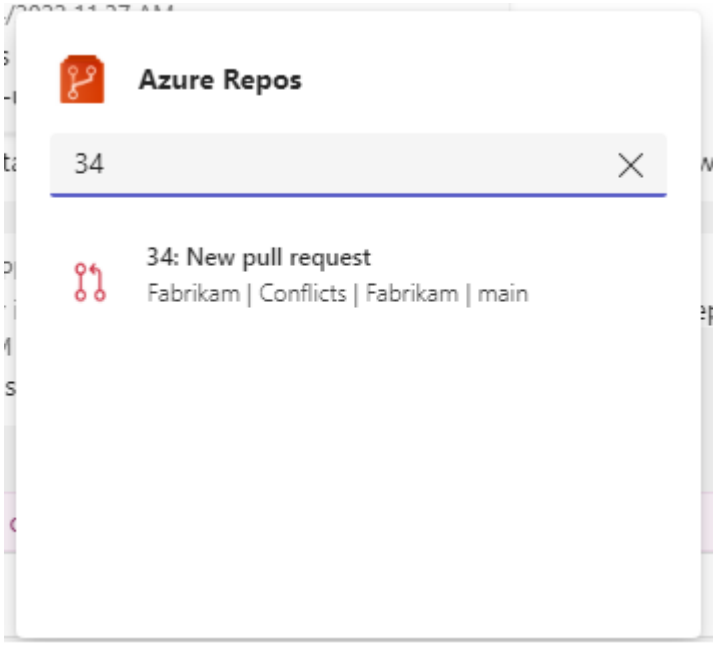
policies before a pull request can be completed.



Integrate with Microsoft Teams

Many teams configure integration with Microsoft Teams, which announces the new pull request to the developers's teammates. The owners of any code touched are automatically added as reviewers. Microsoft teams often use optional reviewers for code that many people touch, like REST client generation and shared controls, to get expert eyes on those changes.





Deploy with feature flags

Once the reviewers, code owners, and automation are satisfied, the developer can complete the pull request. If there's a merge conflict, the developer gets instructions on how to sync to the conflict, fix it, and re-push the changes. The automation runs again on the fixed code, but humans don't have to sign off again.

The branch merges into `main`, and the new code deploys in the next sprint or major release. That doesn't mean the new feature will show up right away. Microsoft decouples the deployment and exposure of new features by using feature flags.

Even if the feature needs a little more work before it's ready to show off, it's safe to go to `main` if the product builds and deploys. Once in `main`, the code becomes part of an official build, where it's again tested, confirmed to meet policy, and digitally signed.

Shift left to detect issues early

This Git workflow provides several benefits. First, working out of a single main branch virtually eliminates merge debt. Second, the pull request flow provides a common point to enforce testing, code review, and error detection early in the pipeline. This shift left strategy helps shorten the feedback cycle to developers because it can detect errors in minutes, not hours or days. This strategy also gives confidence for refactoring, because all changes are tested constantly.

Currently, a product with 200+ pull requests might produce 300+ continuous integration builds per day, amounting to 500+ test runs every 24 hours. This level of testing would be impossible without the trunk-based branching and release workflow.

Release at sprint milestones

At the end of each sprint, the team creates a release branch from the main branch. For example, at the end of sprint 129, the team creates a new release branch `releases/M129`. The team then puts the sprint 129 branch into production.

After the branch of the release branch, the main branch remains open for developers to merge changes. These changes will deploy three weeks later in the next sprint deployment.

Release hotfixes

Sometimes changes need to go to production quickly. Microsoft won't usually add new features in the middle of a sprint, but sometimes wants to bring in a bug fix quickly to unblock users. Issues might be minor, such as typos, or large enough to cause an availability issue or *live site incident*.

Rectifying these issues starts with the normal workflow. A developer creates a branch from `main`, gets it code reviewed, and completes the pull request to merge it. The process always starts by making the change in `main` first. This allows creating the fix quickly and validating it locally without having to switch to the release branch.

Following this process also guarantees that the change gets into `main`, which is critical. Fixing a bug in the release branch without bringing the change back to `main` would mean the bug would recur during the next deployment, when the sprint 130 release branches from `main`. It's easy to forget to update `main` during the confusion and stress that can arise during an outage. Bringing changes to `main` first means always having the changes in both the main branch and the release branch.

Git functionality enables this workflow. To bring changes immediately into production, once a developer merges a pull request into `main`, they can use the pull request page to cherry-pick changes into the release branch. This process creates a new pull request that targets the release branch, backporting the contents that just merged into `main`.

Using the cherry-pick functionality opens a pull request quickly, providing the traceability and reliability of branch policies. Cherry-picking can happen on the server, without having to download the release branch to a local computer. Making changes, fixing merge conflicts, or making minor changes due to differences between the two branches can all happen on the server. Teams can edit changes directly from the browser-based text editor or via the Pull Request Merge Conflict Extension for a more advanced experience.

Once a pull request targets the release branch, the team code review it again, evaluates branch policies, tests the pull request, and merges it. After merge, the fix deploys to the first *ring* of servers in minutes. From there, the team progressively deploys the fix to more accounts by using deployment rings. As the changes deploy to more users, the team monitors success and verifies that the change fixes the bug while not introducing any deficiencies or slowdowns. The fix eventually deploys to all Azure data centers.

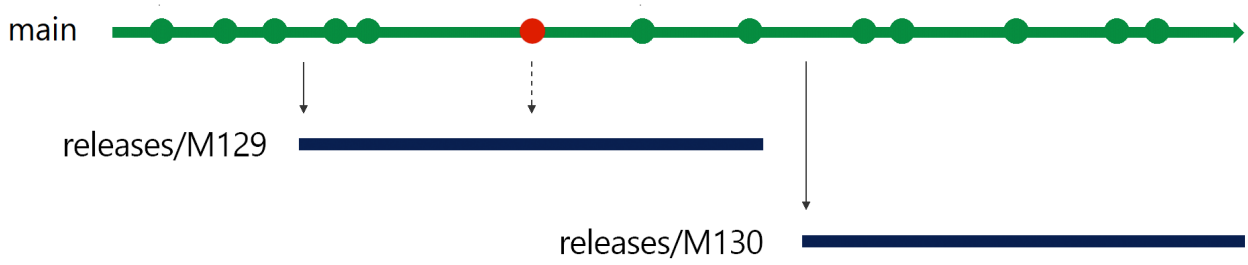
Move on to the next sprint

During the next three weeks, the team finishes adding features to sprint 130 and gets ready to deploy those changes. They create the new release branch, `releases/M130` from `main`, and deploy that branch.

At this point, there are actually two branches in production. With a ring-based deployment to bring changes to production safely, the fast ring gets the sprint 130 changes, and the slow ring servers stay on sprint 129 while the new changes are validated in production.

Hotfixing a change in the middle of a deployment might require hotfixing two different releases, the sprint 129 release and the sprint 130 release. The team ports and deploys the hotfix to both release branches. The 130 branch redeploys with the hotfix to the rings that have already been upgraded. The 129 branch redeploys with the hotfix to the outer rings that haven't upgraded to the next sprint's version yet.

Once all the rings are deployed, the old sprint 129 branch is abandoned, because any changes brought into the sprint 129 branch as a hotfix have also been made in `main`. So, those changes will also be in the `releases/M130` branch.



Summary

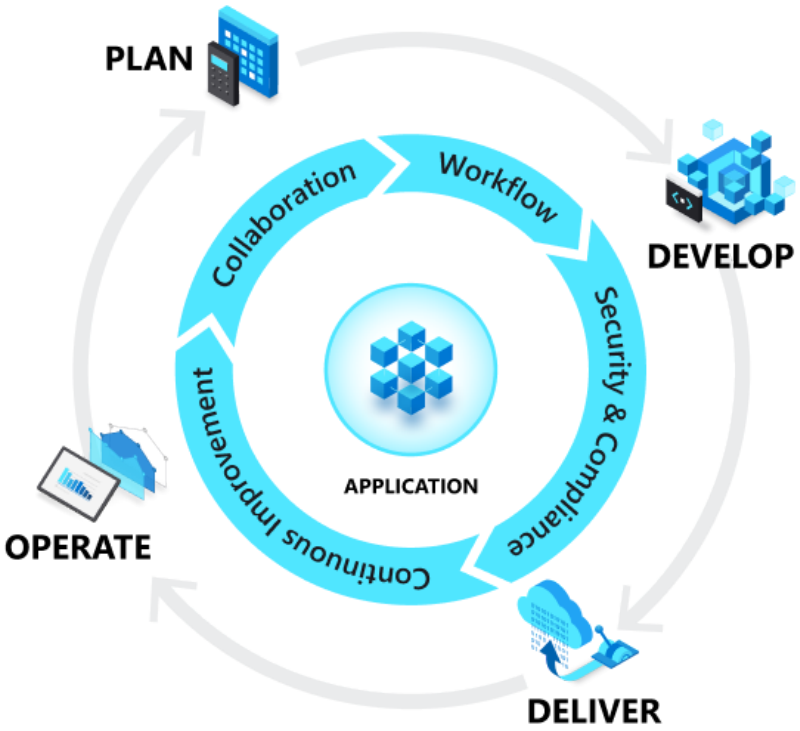
The release flow model is at the heart of how Microsoft develops with DevOps to deliver online services. This model uses a simple, trunk-based branching strategy. But instead of keeping developers stuck in a deployment queue, waiting to merge their changes, the Microsoft release flow lets developers keep working.

This release model also allows deploying new features across Azure data centers at a regular cadence, despite the size of the Microsoft codebases and the number of developers working in them. The model also allows bringing hotfixes into production quickly and efficiently.

Introduction to delivering quality services with DevOps

Article • 11/28/2022

In the delivery phase of DevOps, the code moves through the release pipeline to the production environment. Code delivery typically comes after the continuous integration build and is run through several test environments before reaching end users. Along the way, its quality is tested across many different measures that include functionality, scale, and security.



Employ continuous delivery

Continuous Delivery (CD) is the process to automatically build, test, configure, and deploy from a build environment to a production environment. CD provides the foundation for delivery in DevOps where tests are run, gates are checked, and bits are deployed. There are several different DevOps platforms that offer delivery automation, including GitHub Actions and Azure Pipelines.

Design for optimal deployment

As software projects grow, they can become difficult to manage across teams, versions, and environments. Fortunately, several paradigms are available to help address these challenges. One paradigm is the advent of the microservices architecture, which makes it

easier to build and deploy independent services that can be composed into larger, more maintainable applications. Another practice to aid in the deployment of services is to manage your application environments as Infrastructure as Code.

Shift right to test in production

The Develop phase showed you how project quality and velocity can be improved by shifting left so that some aspects of testing are performed earlier in the process. In a similar way, product quality can be improved with a sustained focus on shifting right to test in production. Testing in production offers quality assurance that simply can't be replicated anywhere else in the pipeline.

Next steps

Microsoft has been one of the world's largest software development companies for decades. Learn about how Microsoft delivers in DevOps.

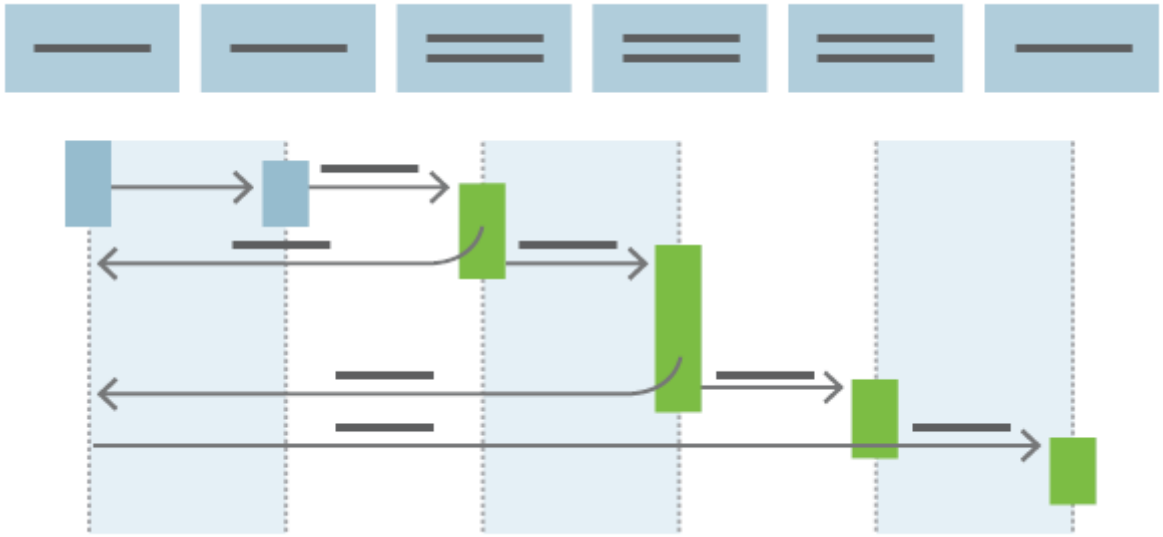
Looking for a hands-on DevOps experience with continuous delivery? Learn to set up release pipelines using GitHub Actions or Azure Pipelines.

What is continuous delivery?

Article • 11/28/2022

Continuously delivering value has become a mandatory requirement for organizations. To deliver value to your end users, you must release continually and without errors.

Continuous delivery (CD) is the process of automating build, test, configuration, and deployment from a build to a production environment. A *release pipeline* can create multiple testing or staging environments to automate infrastructure creation and deploy new builds. Successive environments support progressively longer-running integration, load, and user acceptance testing activities.



Before CD, software release cycles were a bottleneck for application and operations teams. These teams often relied on manual handoffs that resulted in issues during release cycles. Manual processes led to unreliable releases that produced delays and errors.

CD is a *lean* practice, with the goal to keep production fresh with the fastest path from new code or component availability to deployment. Automation minimizes the time to deploy and *time to mitigate (TTM)* or *time to remediate (TTR)* production incidents. In lean terms, CD optimizes process time and eliminates idle time.

Continuous integration (CI) starts the CD process. The release pipeline stages each successive environment to the next environment after tests complete successfully. The automated CD release pipeline allows a *fail fast* approach to validation, where the tests most likely to fail quickly run first, and longer-running tests happen only after the faster ones complete successfully.

The complementary practices of infrastructure as code (IaC) and monitoring facilitate CD.

Progressive exposure techniques

CD supports several patterns for progressive exposure, also called "controlling the blast radius." These practices limit exposure to deployments to avoid risking problems with the overall user base.

- CD can sequence multiple *deployment rings* for progressive exposure. A ring tries a deployment on a user group, and monitors their experience. The first deployment ring can be a *canary* to test new versions in production before a broader rollout. CD automates deployment from one ring to the next.

  Deployment to the next ring can optionally depend on a manual approval step, where a decision maker signs off on the changes electronically. CD can create an auditable record of the approval to satisfy regulatory procedures or other control objectives.

- *Blue/green deployment* relies on keeping an existing blue version live while a new green version deploys. This practice typically uses load balancing to direct increasing amounts of traffic to the green deployment. If monitoring discovers an incident, traffic can be rerouted to the blue deployment still running.

- Feature flags or *feature toggles* are another technique for experimentation and *dark launches*. Feature flags turn features on or off for different user groups based on identity and group membership.

Modern release pipelines allow development teams to deploy new features fast and safely. CD can quickly remediate issues found in production by rolling forward with a new deployment. In this way, CD creates a continuous stream of customer value.

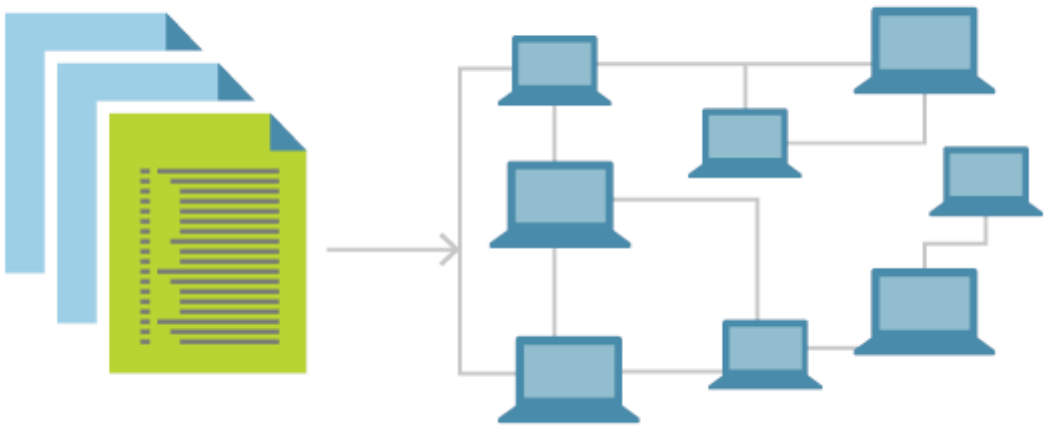
Next steps

- GitHub Actions
- Azure Pipelines
- Azure Pipelines documentation

What is infrastructure as code (IaC)?

Article • 11/28/2022

Infrastructure as code (IaC) uses DevOps methodology and versioning with a descriptive model to define and deploy infrastructure, such as networks, virtual machines, load balancers, and connection topologies. Just as the same source code always generates the same binary, an IaC model generates the same environment every time it deploys.



IaC is a key DevOps practice and a component of continuous delivery. With IaC, DevOps teams can work together with a unified set of practices and tools to deliver applications and their supporting infrastructure rapidly and reliably at scale.

Avoid manual configuration to enforce consistency

IaC evolved to solve the problem of *environment drift* in release pipelines. Without IaC, teams must maintain deployment environment settings individually. Over time, each environment becomes a "snowflake," a unique configuration that can't be reproduced automatically. Inconsistency among environments can cause deployment issues. Infrastructure administration and maintenance involve manual processes that are error prone and hard to track.

IaC avoids manual configuration and enforces consistency by representing desired environment states via well-documented code in formats such as JSON. Infrastructure deployments with IaC are repeatable and prevent runtime issues caused by configuration drift or missing dependencies. Release pipelines execute the environment

descriptions and version configuration models to configure target environments. To make changes, the team edits the source, not the target.

Idempotence, the ability of a given operation to always produce the same result, is an important IaC principle. A deployment command always sets the target environment into the same configuration, regardless of the environment's starting state. Idempotency is achieved by either automatically configuring the existing target, or by discarding the existing target and recreating a fresh environment.

Helpful tools

- Discover misconfiguration in IaC with Microsoft Defender for Cloud

Deliver stable test environments rapidly at scale

IaC helps DevOps teams test applications in production-like environments early in the development cycle. Teams can provision multiple test environments reliably on demand. The cloud dynamically provisions and tears down environments based on IaC definitions. The infrastructure code itself can be validated and tested to prevent common deployment issues.

Use declarative definition files

IaC should use declarative definition files if possible. A definition file describes the components and configuration that an environment requires, but not necessarily how to achieve that configuration. For example, the file might define a required server version and configuration, but not specify the server installation and configuration process. This abstraction allows for greater flexibility to use optimized techniques the infrastructure provider supplies. Declarative definitions also help reduce the technical debt of maintaining imperative code, such as deployment scripts, that can accrue over time.

There's no standard syntax for declarative IaC. The syntax for describing IaC usually depends on the requirements of the target platform. Different platforms support file formats such as YAML, JSON, and XML.

Deploy IaC on Azure

Azure provides native support for IaC via the Azure Resource Manager model. Teams can define declarative ARM templates that specify the infrastructure required to deploy

solutions.

Third-party platforms like Terraform, Ansible, Chef, and Pulumi also support IaC to manage automated infrastructure.

Deploy to Azure infrastructure with GitHub Actions

Article • 11/30/2022

In this guide, we'll cover how to utilize CI/CD and Infrastructure as Code (IaC) to deploy to Azure with GitHub Actions in an automated and repeatable fashion.

This article is an architecture overview and presents a structured solution for designing an application on Azure that's scalable, secure, resilient, and highly available. To see more real world examples of cloud architectures and solution ideas, browse Azure architectures.

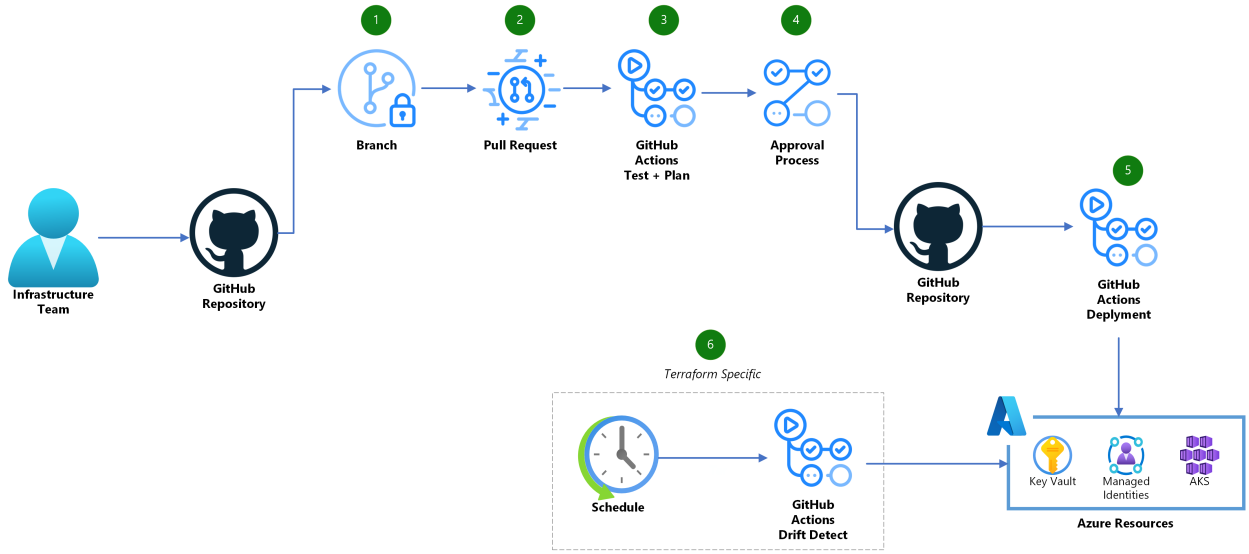
Benefits of using IaC and automation for deployments

There are many ways to deploy to Azure including the Azure portal, CLI, API, and more. For this guide, we'll use IaC and CI/CD automation. Benefits of this approach include:

- **Declarative**: When you define your infrastructure and deployment process in code, it can be versioned and reviewed using the standard software development lifecycle. IaC also helps prevent any drift in your configuration.
- **Consistency**: Following an IaC process ensures that the whole organization follows a standard, well-established method to deploy infrastructure that incorporates best practices and is hardened to meet your security needs. Any improvements made to the central templates can easily be applied across the organization.
- **Security**: Centrally managed templates can be hardened and approved by a cloud operations or security team to meet internal standards.
- **Self-Service**: Teams can be empowered to deploy their own infrastructure by utilizing centrally managed templates.
- **Improved Productivity**: By utilizing standard templates, teams can quickly provision new environments without needing to worry about all the implementation details.

Additional information can be found under repeatable infrastructure in the Azure Architecture Center or what is infrastructure as code in the DevOps Resource Center.

Architecture



Dataflow

1. Create a new branch and check in the needed IaC code modifications.
2. Create a Pull Request (PR) in GitHub once you're ready to merge your changes into your environment.
3. A GitHub Actions workflow will trigger to ensure your code is well formatted, internally consistent, and produces secure infrastructure. In addition, a Terraform Plan or Bicep what-if analysis will run to generate a preview of the changes that will happen in your Azure environment.
4. Once appropriately reviewed, the PR can be merged into your main branch.
5. Another GitHub Actions workflow will trigger from the main branch and execute the changes using your IaC provider.
6. *(exclusive to Terraform)* A regularly scheduled GitHub Action workflow should also run to look for any configuration drift in your environment and create a new issue if changes are detected.

Prerequisites

Use Bicep

1. **Create GitHub Environments**

   The workflows utilize GitHub environments and secrets to store the Azure identity information and set up an approval process for deployments. Create an environment named `production` by following these instructions. On the `production` environment, set up a protection rule and add any required approvers

you want that need to sign off on production deployments. You can also limit the environment to your main branch. Detailed instructions can be found here.

2. **Set up Azure Identity**:

   An Azure Active Directory application is required that has permissions to deploy within your Azure subscription. Create a single application and give it the appropriate read/write permissions in your Azure subscription. Next set up the federated credentials to allow the GitHub to utilize the identity using OpenID Connect (OIDC). See the Azure documentation for detailed instructions. Three federated credentials will need to be added:

   - Set Entity Type to `Environment` and use the `production` environment name.
   - Set Entity Type to `Pull Request`.
   - Set Entity Type to `Branch` and use the `main` branch name.

3. **Add GitHub secrets**

   > **Note**
   >
   > While none of the data about the Azure identities contain any secrets or credentials, we still utilize GitHub secrets as a convenient means to parameterize the identity information per environment.

   Create the following secrets on the repository using the Azure identity:

   - `AZURE_CLIENT_ID` : The application (client) ID of the app registration in Azure
   - `AZURE_TENANT_ID` : The tenant ID of Azure Active Directory where the app registration is defined.
   - `AZURE_SUBSCRIPTION_ID` : The subscription ID where the app registration is defined.

   Instructions to add the secrets to the repository can be found here.

Use Terraform

1. **Configure Terraform State Location**

   Terraform utilizes a state file to store information about the current state of your managed infrastructure and associated configuration. This file will need to be persisted between different runs of the workflow. The recommended approach is to store this file within an Azure Storage Account or other similar remote backend.

Normally, this storage would be provisioned manually or via a separate workflow. The Terraform backend block will need updated with your selected storage location (see here for documentation).

2. **Create GitHub environment**

   The workflows utilize GitHub environments and secrets to store the Azure identity information and set up an approval process for deployments. Create an environment named `production` by following these instructions. On the `production` environment set up a protection rule and add any required approvers you want that need to sign off on production deployments. You can also limit the environment to your main branch. Detailed instructions can be found here.

3. **Set up Azure Identity**:

   An Azure Active Directory application is required that has permissions to deploy within your Azure subscription. Create a separate application for a `read-only` and `read/write` accounts and give them the appropriate permissions in your Azure subscription. In addition, both roles will also need at least `Reader and Data Access` permissions to the storage account where the Terraform state from step 1 resides. Next, set up the federated credentials to allow GitHub to utilize the identity using OpenID Connect (OIDC). See the Azure documentation for detailed instructions.

   For the `read/write` identity create one federated credential as follows:

   - Set `Entity Type` to `Environment` and use the `production` environment name.

   For the `read-only` identity create two federated credentials as follows:

   - Set `Entity Type` to `Pull Request`.
   - Set `Entity Type` to `Branch` and use the `main` branch name.

4. **Add GitHub secrets**

   > **Note**
   >
   > While none of the data about the Azure identities contain any secrets or credentials, we still utilize GitHub secrets as a convenient means to parameterize the identity information per environment.

   Create the following secrets on the repository using the `read-only` identity:

   - `AZURE_CLIENT_ID` : The application (client) ID of the app registration in Azure

- `AZURE_TENANT_ID` : The tenant ID of Azure Active Directory where the app registration is defined.
- `AZURE_SUBSCRIPTION_ID` : The subscription ID where the app registration is defined.

Instructions to add the secrets to the repository can be found here.

Create another secret on the `production` environment using the `read-write` identity:

- `AZURE_CLIENT_ID` : The application (client) ID of the app registration in Azure

Instructions to add the secrets to the environment can be found here. The environment secret will override the repository secret when doing the deploy step to the `production` environment when elevated read/write permissions are required.

Deploy with GitHub Actions

Use Bicep

There are two main workflows included in the reference architecture:

1. **Bicep Unit Tests**

   This workflow runs on every commit and is composed of a set of unit tests on the infrastructure code. It runs bicep build to compile the bicep to an ARM template. This ensures there are no formatting errors. Next it performs a validate to ensure the template is deployable. Lastly, checkov, an open source static code analysis tool for IaC, will run to detect security and compliance issues. If the repository is utilizing GitHub Advanced Security (GHAS), the results will be uploaded to GitHub.

2. **Bicep What-If / Deploy**

   This workflow runs on every pull request and on each commit to the main branch. The what-if stage of the workflow is used to understand the impact of the IaC changes on the Azure environment by running what-if. This report is then attached to the PR for easy review. The deploy stage runs after the what-if analysis when the workflow is triggered by a push to the main branch. This stage will deploy the template to Azure after a manual review has signed off.

Use Terraform

There are three main workflows included in the reference architecture:

1. **Terraform Unit Tests**

   This workflow runs on every commit and is composed of a set of unit tests on the infrastructure code. It runs terraform fmt to ensure the code is properly linted and follows terraform best practices. Next it performs terraform validate to check that the code is syntactically correct and internally consistent. Lastly, checkov, an open source static code analysis tool for IaC, will run to detect security and compliance issues. If the repository is utilizing GitHub Advanced Security (GHAS), the results will be uploaded to GitHub.

2. **Terraform Plan / Apply**

   This workflow runs on every pull request and on each commit to the main branch. The plan stage of the workflow is used to understand the impact of the IaC changes on the Azure environment by running terraform plan. This report is then attached to the PR for easy review. The apply stage runs after the plan when the workflow is triggered by a push to the main branch. This stage will take the plan document and apply the changes after a manual review has signed off if there are any pending changes to the environment.

3. **Terraform Drift Detection**

   This workflow runs on a periodic basis to scan your environment for any configuration drift or changes made outside of Terraform. If any drift is detected, a GitHub Issue is raised to alert the maintainers of the project.

Related Resources

- What is Infrastructure as Code
- Repeatable infrastructure
- Comparing Terraform and Bicep
- Checkov and source code
- GitHub Advanced Security

What are Microservices?

Article • 11/28/2022

Microservices describes the architectural process of building a distributed application from separately deployable services that perform specific business functions and communicate over web interfaces. DevOps teams enclose individual pieces of functionality in microservices and build larger systems by combining the microservices like building blocks.

Microservices apply an example of the open/closed principle:

- They're open for extension (using the interfaces they expose)
- They're closed for modification (each is implemented and versioned independently)

Microservices provide many benefits over monolithic architectures:

- They can remove *single points of failure* (SPOFs) by ensuring issues in one service don't crash or affect other parts of an application.
- Individual microservices can be scaled out independently to provide extra availability and capacity.
- DevOps teams can extend functionality by adding new microservices without unnecessarily affecting other parts of the application.

Using microservices can increase team velocity. DevOps practices, such as Continuous Integration and Continuous Delivery, are used to drive microservice deployments. Microservices nicely complement cloud-based application architectures by allowing software development teams to take advantage of scenarios such as event-driven programming and autoscale. The microservice components expose APIs (application

programming interfaces), typically over REST protocols, for communicating with other services.

An increasingly common practice is to use container clusters to implement microservices. Containers allow for the isolation, packaging, and deployment of microservices, while orchestration scales out a group of containers into an application.

Next steps

Learn more about microservices on Azure.

Shift right to test in production

Article • 11/28/2022

Shift right is the practice of moving some testing later in the DevOps process to *test in production*. Testing in production uses real deployments to validate and measure an application's behavior and performance in the production environment.

One way DevOps teams can improve velocity is with a shift-left test strategy. Shift left pushes most testing earlier in the DevOps pipeline, to reduce the amount of time for new code to reach production and operate reliably.

But while many kinds of tests, such as unit tests, can easily shift left, some classes of tests can't run without deploying part or all of a solution. Deploying to a QA or staging service can simulate a comparable environment, but there's no full substitute for the production environment. Teams find that certain types of testing need to happen in production.

Testing in production provides:

- The full breadth and diversity of the production environment.
- The real workload of customer traffic.
- Profiles and behaviors as production demand evolves over time.

The production environment keeps changing. Even if an app doesn't change, the infrastructure it relies on changes constantly. Testing in production validates the health and quality of a given production deployment and of the constantly changing production environment.

Shifting right to test in production is especially important for the following scenarios:

Microservices deployments

Microservices-based solutions can have a large number of microservices that are developed, deployed, and managed independently. Shifting testing right is especially important for these projects, because different versions and configurations can reach production in many ways. Regardless of pre-production test coverage, it's necessary to test compatibility in production.

Ensuring quality post-deployment

Releasing to production is just half of delivering software. The other half is ensuring quality at scale with a real workload in production. Because the environment keeps

changing, a team is never done with testing in production.

Test data from production is literally the test results from the real customer workload. Testing in production includes monitoring, failover testing, and fault injection. This testing tracks failures, exceptions, performance metrics, and security events. The test telemetry also helps detect anomalies.

Deployment rings

To safeguard the production environment, teams can roll out changes in a progressive and controlled way by using ring-based deployments and feature flags. For example, it's better to catch a bug that prevents a shopper from completing their purchase when less than 1% of customers are on that deployment ring, than after switching all customers at once. The feature value with detected failures must exceed the net losses of those failures, measured in a meaningful way for the given business.

The first ring should be the smallest size necessary to run the standard integration suite. The tests might be similar to those already run earlier in the pipeline against other environments, but testing validates that the behavior is the same in the production environment. This ring identifies obvious errors, such as misconfigurations, before they impact any customers.

After the initial ring validates, the next ring can broaden to include a subset of real users for the test run. If everything looks good, the deployment can progress through further rings and tests until everyone is using it. Full deployment doesn't mean that testing is over. Tracking telemetry is critically important for testing in production.

Fault injection

Teams often employ *fault injection* and *chaos engineering* to see how a system behaves under failure conditions. These practices help to:

- Validate that implemented resiliency mechanisms actually work.
- Validate that a failure in one subsystem is contained within that subsystem and doesn't cascade to produce a major outage.
- Prove that repair work for a prior incident has the desired effect, without having to wait for another incident to occur.
- Create more realistic training drills for live site engineers so they can better prepare to deal with incidents.

It's a good practice to automate fault injection experiments, because they are expensive tests that must run on ever-changing systems.

Chaos engineering can be an effective tool, but should be limited to *canary environments* that have little or no customer impact.

Failover testing

One form of fault injection is *failover* testing to support business continuity and disaster recovery (BCDR). Teams should have failover plans for all services and subsystems. The plans should include:

- A clear explanation of the business impact of the service going down.
- A map of all the dependencies in terms of platform, technology, and people devising the BCDR plans.
- Formal documentation of disaster recovery procedures.
- A cadence to regularly execute disaster recovery drills.

Circuit breaker fault testing

A *circuit breaker* mechanism cuts off a given component from a larger system, usually to prevent failures in that component from spreading outside its boundaries. You can intentionally trigger circuit breakers to test the following scenarios:

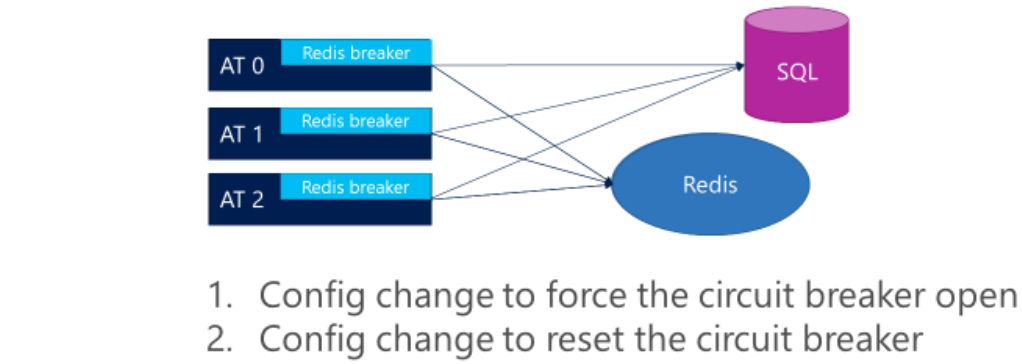
- Whether a fallback works when the circuit breaker opens. The fallback might work with unit tests, but the only way to know if it will behave as expected in production is to inject a fault to trigger it.

- Whether the circuit breaker has the right sensitivity threshold to open when it needs to. Fault injection can force latency or disconnect dependencies to observe breaker responsiveness. It's important to verify not only that the correct behavior occurs, but that it happens quickly enough.

Example: Testing a Redis cache circuit breaker

Redis cache improves product performance by speeding up access to commonly used data. Consider a scenario that takes a non-critical dependency on Redis. If Redis goes down, the system should continue to work, because it can fall back to using the original data source for requests. To confirm that a Redis failure triggers a circuit breaker and that the fallback works in production, periodically run tests against these behaviors.

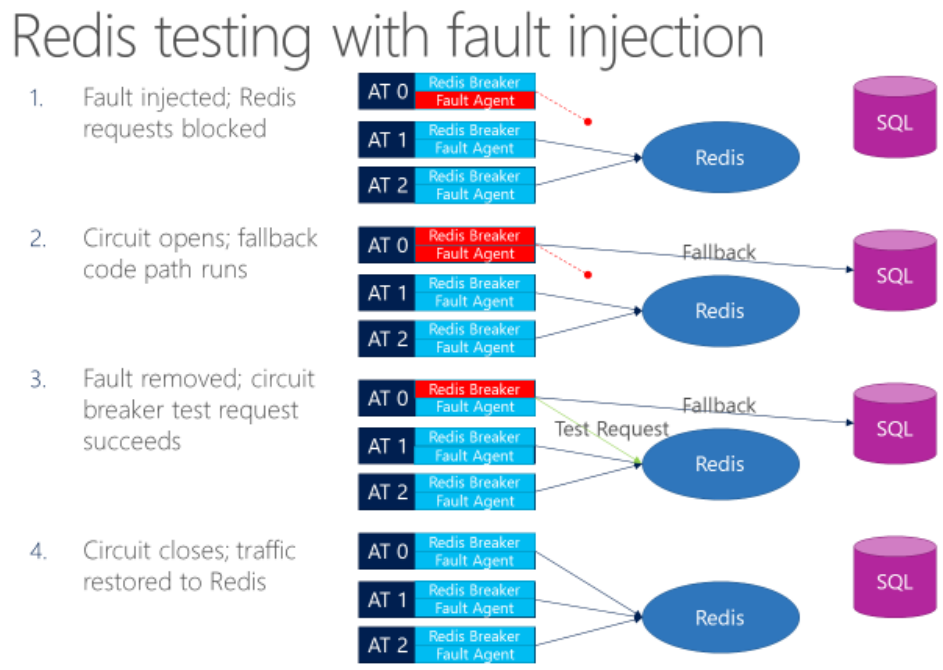
The following diagram shows tests for the Redis circuit breaker fallback behavior. The goal is to make sure that when the breaker opens, calls ultimately go to SQL.



The preceding diagram shows three ATs, with the breakers in front of the calls to Redis. One test forces the circuit breaker to open through a configuration change, and then observes whether the calls go to SQL. Another test then checks the opposite configuration change, by closing the circuit breaker to confirm that calls return back to Redis.

This test validates that the fallback behavior works when the breaker opens, but it doesn't validate that the circuit breaker configuration opens the breaker when it should. Testing that behavior requires simulating actual failures.

A fault agent can introduce faults in calls going to Redis. The following diagram shows testing with fault injection.



1. The fault injector blocks Redis requests.
2. The circuit breaker opens, and the test can observe whether fallback works.
3. The fault is removed, and the circuit breaker sends a test request to Redis.
4. If the request succeeds, calls revert back to Redis.

Further steps could test the sensitivity of the breaker, whether the threshold is too high or too low, and whether other system timeouts interfere with the circuit breaker behavior.

In this example, if the breaker doesn't open or close as expected, it could cause a *live site incident (LSI)*. Without the fault injection testing, the issue might go undetected, as it's hard to do this type of testing in a lab environment.

Next steps

- [Shift testing left with unit tests]shift-left
- What are microservices?
- Run a test failover (disaster recovery drill) to Azure
- Safe deployment practices
- What is monitoring?

How Microsoft delivers software with DevOps

Article • 11/28/2022

Microsoft has decades of experience delivering highly scalable services to production environments. As Microsoft services and environments have expanded, their delivery practices have also evolved over time. Many Microsoft customers have also adopted and benefit from these efficient delivery practices. The following core DevOps principles and processes can apply to any modern software delivery effort.

To implement DevOps delivery processes, Microsoft adopted the following initiatives:

- Focus organizational mindset and cadence on delivery.
- Form autonomous, accountable teams who own, test, and deliver features.
- Shift right to test and monitor systems in production.

Focus on delivery

Shipping faster is an obvious benefit that organizations and teams can easily measure and appreciate. The typical DevOps cadence involves short *sprint* cycles with regular deployments to production.

Fearing a lack of product stability with short sprints, some teams had compensated with stabilization periods at the end of their sprint cycles. Engineers wanted to ship as many features as possible during the sprint, so they incurred test debt that they had to pay down during stabilization. Teams that managed their debt during the sprint then had to support the teams that built up debt. The extra costs played out through the delivery pipelines and into production.

Removing the stabilization period quickly improved the way teams managed their debt. Instead of pushing off key maintenance work to the stabilization period, teams that built up debt had to spend the next sprint catching up to their debt targets. Teams quickly learned to manage their test debt during sprints. Features deliver when they're proven and worth the cost of deployment.

Fully automate pipelines

Much of the improvement teams can gain immediately is to fully automate the pipelines from code repository to production. Automation includes release pipelines with continuous integration (CI), automated testing, and continuous delivery (CD).

Teams might avoid deploying because it's hard, but the less frequently they deploy, the harder it is. The more time between deployments, the more issues pile up. If the code isn't fresh, there's deployment debt.

It's easier to work in smaller chunks by deploying frequently. This idea might seem obvious in hindsight, but at the time it can seem counterintuitive. Frequent deployments also motivate teams to prioritize creating more efficient and reliable deployment tools and pipelines.

Use in-house tools

Microsoft uses the release management system they build, and ships it to customers. A single investment improves both team productivity and Microsoft products. Using a secondary system would siphon off development and delivery velocity.

Team autonomy and accountability

No specific key progress indicators (KPIs) measure team productivity or performance, or whether a feature is on track. Teams need to be able to manage their own plans and backlogs, while finding a way to align with organizational goals.

It's important to communicate directly with teams to track progress. Tools should facilitate communication, but conversation is the most transparent way to communicate.

Prioritize features

An important goal is to focus on delivering features. Schedules can assess how much teams and individuals can reasonably complete over a given period of time, but some features will deliver earlier and some will come later. Teams can prioritize work so the most important features make it to production.

Use microservices

Microservices offer various technical benefits that improve and simplify delivery. Microservices also provide natural boundaries for team ownership. When a team has autonomy over investment in a microservice, they can prioritize how to implement features and manage debt. Teams can focus on plans for factors like versioning, independent of the overall services that depend on the microservice.

Work in main

Engineers used to work in separate branches. Merge debt on each branch grew until the engineer tried to integrate their branch into the main branch. The more teams and engineers there were, the bigger the integration became.

For integration to happen faster, more continuously, and in smaller chunks, engineers now work in the main branch. One big reason for moving to Git was the lightweight branching Git offers. The benefit to internal engineering was eliminating the deep branch hierarchy and its waste. All the time that used to be spent integrating is now poured into delivery.

Use feature flags

Some features aren't completely finished for a sprint deployment, but can still benefit from testing in production. Teams can merge and deploy this code with feature flags to turn on the feature for specific users, such as the development team or a small segment of early adopters. Feature flags control exposure without risking problems with the overall user base, and can help teams determine whether and how to complete the feature.

Testing in production

Shifting right to test in production helps ensure that pre-production tests are valid, and that ever-changing production environments are ready to handle deployments.

Instrument tests and metrics

Regardless of where an app deploys, it's important to instrument everything. Instrumentation not only helps identify and fix issues, but can provide invaluable research on usage and what to add next.

Test resiliency patterns

A risk for complex deployments is *cascading failures*, in which one component failure causes dependent components to fail, and so on until the entire system breaks down. It's important to understand where *single points of failure (SPOFs)* are and how they're mitigated, and to test the mitigation processes, especially in production.

Choose the right metrics

Designing metrics can be difficult. A common mistake is to include too many metrics, to avoid missing anything. But this can lead to ignoring or mistrusting the value of metrics that don't meet a specific need. Instead, Microsoft teams take time to determine the data they need to measure success. Teams might add or change metrics, but understanding the purpose from the beginning facilitates that process.

Besides the basis of a metric, teams consider what they need the metric to measure. For example, the velocity or acceleration of user gains might be a more useful metric than total number of users. Metrics vary from project to project, but the most helpful are those with the potential to drive business decisions.

Use metrics to guide work

Microsoft includes metrics with reviews at the highest leadership levels. Every six weeks, organizations present how they're doing on health, business, scenarios, and customer telemetry. Organizations discuss the metrics with executives and with their teams.

Teams throughout the organization examine engaged user metrics to determine the meaning for their features. Teams don't just ship features, but look to see whether and how people are using them. Teams use these metrics to adjust backlogs and determine whether features need more work to meet goals.

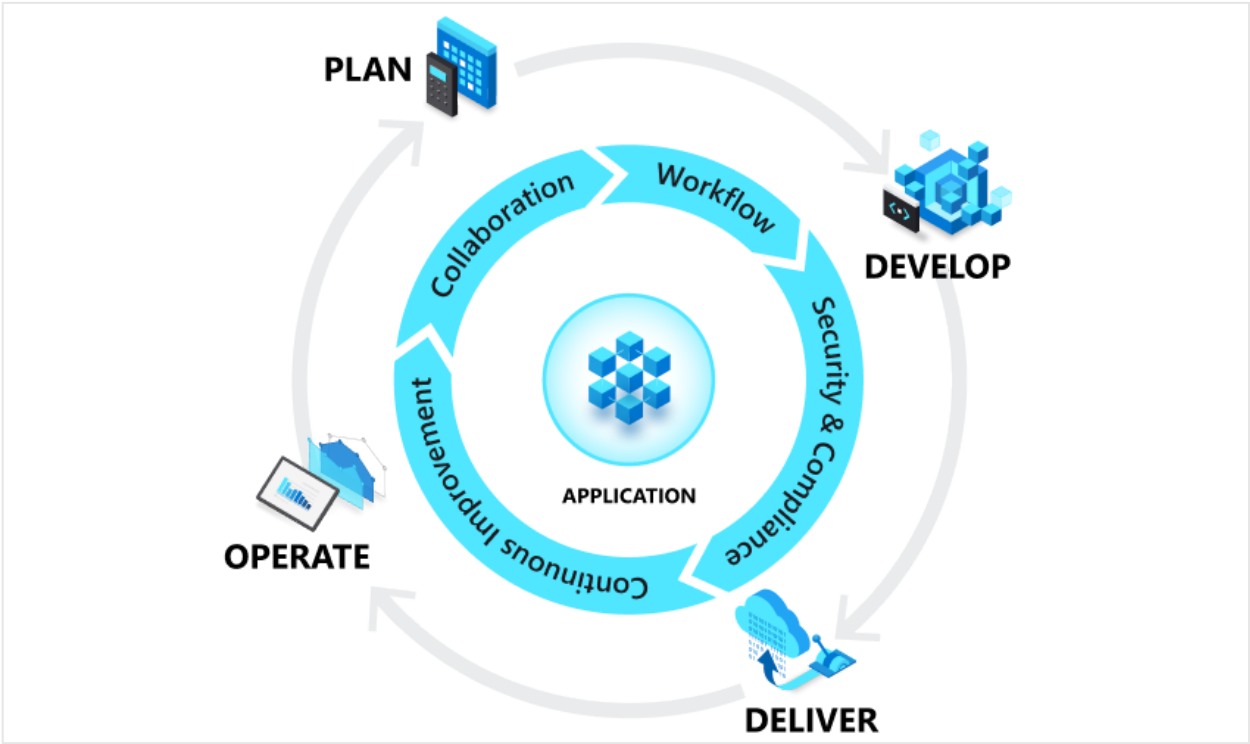
Delivery guidelines

- It's never a straight line to get from A to B, nor is B the end.
- There will always be setbacks and mistakes.
- View setbacks as learning opportunities to change tactics for completing a given part of the process.
- Over time, every team evolves its DevOps practices by building on experience and adjusting to meet changing needs.
- The key is to focus on delivering value, both to end users and to the delivery process itself.

Introduction to operating reliable systems with DevOps

Article • 11/28/2022

The operations phase of DevOps comes after a successful delivery and encompasses everything that teams must consider to maintain, monitor, and troubleshoot the application. The build gets exposed to real customers in the production environment, where reliability becomes a critical factor.



Manage release exposure

Getting the product deployed to its production environment might seem like the final step, but it's only the beginning of a whole new world. A lot can go wrong, so it's important that teams employ safe deployment practices that provide the right balance of customer exposure and risk. Teams can also experiment with changes using feature flags to explore how new updates and features impact a potential audience.

Operate at full potential

Teams need to ensure that the systems they operate are always available—regardless of updates, changes, or underlying issues. Staying on top of everything requires a firm grasp of all the tools and features available for monitoring production systems. The

right approach can ensure that systems receive updates and continue to operate with no downtime.

Secure production deployments

Security has become a central concern for applications. DevSecOps describes the set of practices a team follows to build and maintain systems that are as secure as possible. These practices reach beyond code and infrastructure to also include policies for humans to follow, as well as guidance for handling and recovering from potential breaches.

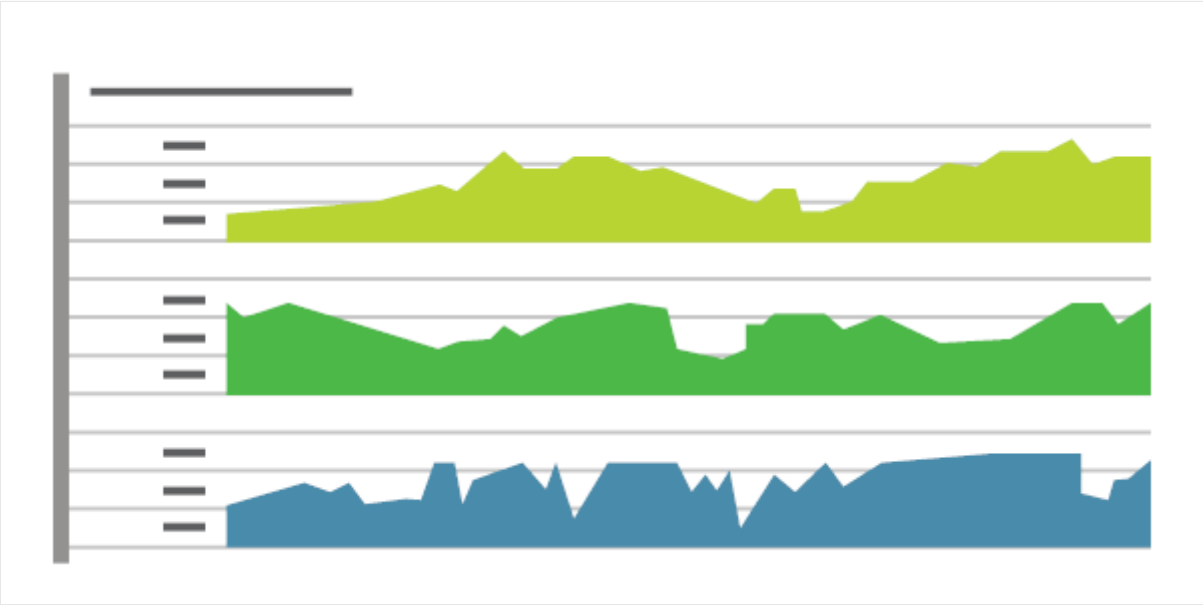
Next steps

Learn how effective monitoring helps to ensure high system availability and allows DevOps teams to deliver results quickly.

What is monitoring?

Article • 11/28/2022

Once an application is deployed to production, monitoring provides information about the application's performance and usage patterns so you can identify, mitigate, or resolve issues.



Goals of monitoring

One goal of monitoring is to achieve high availability by minimizing key metrics that are measured in terms of time:

- *Time to detect* (TTD): When performance or other issues arise, rich diagnostic data about the issues are fed back to development teams via automated monitoring.
- *Time to mitigate* (TTM): DevOps teams act on the information to mitigate issues as quickly as possible so that users are no longer affected.
- *Time to remediate* (TTR): Resolution times are measured, and teams work to improve over time. After mitigation, teams work on how to remediate problems at root cause so that they don't recur.

A second goal of monitoring is to enable *validated learning* by tracking usage. The core concept of validated learning is that every deployment is an opportunity to track experimental results that support or diminish the hypotheses that led to the deployment. Tracking usage and differences between versions allows teams to measure the impact of change and drive business decisions. If a hypothesis is diminished, the team can *fail fast* or *pivot*. If the hypothesis is supported, then the team can double

down or *persevere*. These data-informed decisions lead to new hypotheses and prioritization of the backlog.

Key concepts

Telemetry is the mechanism for collecting data from monitoring. Telemetry can use agents that are installed in deployment environments, an SDK that relies on markers inserted into source code, server logging, or a combination of these. Typically, telemetry will distinguish between the data pipeline optimized for real-time alerting and dashboards and higher-volume data needed for troubleshooting or usage analytics.

Synthetic monitoring uses a consistent set of transactions to assess performance and availability. Synthetic transactions are predictable tests that have the advantage of allowing comparison from release to release in a highly predictable manner. *Real user monitoring* (RUM), on the other hand, measures experience from the user's browser, mobile device, or desktop. It accounts for *last mile* conditions such as cellular networks, internet routing, and caching. Unlike synthetics, RUM typically doesn't provide repeatable measurement over time.

Monitoring is often used to test in production. A well-monitored deployment streams data about its health and performance so that you can spot production incidents immediately. Combined with a continuous deployment release pipeline, monitoring will detect new anomalies and allow for prompt mitigation. This allows discovery of the *unknown unknowns* in application behavior that can't be foreseen in pre-production environments.

Effective monitoring is essential to allow DevOps teams to deliver at speed, get feedback from production, and increase customer satisfaction, acquisition, and retention.

Next steps

Read more about the monitoring capabilities of Azure Monitor.

Learn how to set up and use Application Insights for monitoring.

Safe deployment practices

Article • 11/28/2022

Sometimes a release doesn't live up to expectations. Despite using best practices and passing all quality gates, there are occasionally issues that result in a production deployment causing unforeseen problems for users. To minimize and mitigate the impact of these issues, DevOps teams are encouraged to adopt a *progressive exposure* strategy that balances the exposure of a given release with its proven performance. As a release proves itself in production, it becomes available to tiers of broader audiences until everyone is using it. Teams can use safe deployment practices in order to maximize the quality and speed of releases in production.

Control exposure to customers

DevOps teams can employ various practices to control the exposure of updates to customers. Historically, A/B testing has been a popular choice for teams looking to see how different versions of a service or user interface perform against target goals. A/B testing is also relatively easy to use since the changes are typically minor and often only compare different releases at the customer-facing edge of a service.

Safe deployment through rings

As platforms grow, infrastructure scale and audience needs tend to grow as well. This creates a demand for a deployment model that balances the risks associated with a new deployment with the benefits of the updates it promises. The general idea is that a given release should be first exposed only to a small group of users with the highest tolerance for risk. Then, if the release works as expected, it can be exposed to a broader group of users. If there are no problems, then the process can continue out through broader groups of users, or *rings*, until everyone is using the new release. With modern continuous delivery platforms like GitHub Actions and Azure Pipelines, building a deployment process with rings is accessible to DevOps teams of any size.

Feature flags

Certain functionality sometimes needs to be deployed as part of a release, but not initially exposed to users. In those cases, feature flags provide a solution where functionality may be enabled via configuration changes based on environment, ring, or any other specific deployment.

User opt-in

Similar to feature flags, user opt-in provides a way to limit exposure. In this model, a given feature is enabled in the release, but not activated for a user unless they specifically want it. The risk tolerance decision is offloaded to users so they can decide how quickly they want to adopt certain updates.

Multiple practices are commonly employed simultaneously. For example, a team may have an experimental feature intended for a very specific use case. Since it's risky, they'll deploy it to the first ring for internal users to try out. However, even though the features are in the code, someone will need to set the feature flag for a specific deployment within the ring so that the feature is exposed via the user interface. Even then, the feature flag may only expose the option for a user to opt in to using the new feature. Anyone who isn't in the ring, on that deployment, or hasn't opted in won't be exposed to the feature. While this is a fairly contrived example, it serves to illustrate the flexibility and practicality of progressive exposure.

Common issues teams face early on

As teams move toward a more Agile DevOps practice, they may run into problems consistent with others who've migrated away from traditional monolithic deliveries. Teams used to deploying once every few months have a mindset that buffers for stabilization. They expect that each deployment will introduce a substantial shift in their service, and that there will be unforeseen issues.

Payloads are too big

Services that are deployed every few months are usually filled with many changes. This increases the likelihood that there will be immediate issues, and also makes it difficult to troubleshoot those issues since there's so much new stuff. By moving to more frequent deliveries, the differences in what's deployed become smaller, which allows for more focused testing and easier debugging.

No service isolation

Monolithic systems are traditionally scaled by leveling up the hardware on which they're deployed. However, when something goes wrong with the instance, it leads to problems for everyone. One simple solution is to add multiple instances so that you can load balance users. However, this can require significant architectural considerations as many

legacy systems aren't built to be multi-instance. Plus, significant duplicate resources may need to be allocated for functionality that may be better consolidated elsewhere.

As new features are added, explore whether a microservices architecture can help you operate and scale thanks to better service isolation.

Manual steps lead to mistakes

When a team only deploys a few times per year, automating deliveries may not seem worth the investment. As a result, many deployment processes are manually managed. This requires a significant amount of time and effort, and is prone to human error. Simply automating the most common build and deployment tasks can go a long way toward reducing lost time and unforced errors.

Teams can also make use of infrastructure as code to have better control over deployment environments. This removes the need for requests to the operations team to make manual changes as new features or dependencies are introduced to various deployment environments.

Only Ops can do deployments

Some organizations have policies that require all deployments to be initiated and managed by the operations staff. While there may have been good reasons for that in the past, an Agile DevOps process greatly benefits when the development team can initiate and control deployments. Modern continuous delivery platforms offer granular control over who can initiate which deployments, and who can access status logs and other diagnostic information, making sure the right people have the right information as quickly as possible.

Bad deployments proceed and can't be rolled back

Sometimes a deployment goes wrong and teams need to address it. However, when processes are manual and access to information is slow and limited, it can be difficult to roll back to a previous working deployment. Fortunately, there are various tools and practices for mitigating the risk of failed deployments.

Core principles

Teams looking to adopt safe deployment practices should set some core principles to underpin the effort.

Be consistent

The same tools used to deploy in production should be used in development and test environments. If there are issues, such as the ones that often arise from new versions of dependencies or tools, they should be caught well before the code is close to being released to production.

Care about quality signals

Too many teams fall into the common trap of not really caring about quality signals. Over time, they may find that they write tests or take on quality tasks simply to change a yellow warning to a green approval. Quality signals are really important as they represent the pulse of a project. The quality signals used to approve deployments should be tracked constantly every day.

Deployments should require zero downtime

While it's not critical for every service to always be available, teams should approach their DevOps delivery and operation stages with the mindset that they can and should deploy new versions without having to take them down for any time at all. Modern infrastructure and pipeline tools are advanced enough now where it's feasible for virtually any team to target 100% uptime.

Deployments should happen during working hours

If a team works with the mindset that deployments require zero downtime, then it doesn't really matter when a deployment is pushed. Further, it becomes advantageous to push deployments during working hours, especially early in the day and early in the week. If something goes wrong, it should be traced early enough to control the blast radius. Plus, everyone will already be working and focused on getting issues fixed.

Ring-based deployment

Teams with mature DevOps release practices are in a position to take on ring-based deployment. In this model, new features are first rolled out to customers willing to accept the most risk. As the deployment is proven, the audience expands to include more users until everyone is using it.

An example ring model

A typical ring deployment model is designed to find issues as early as possible through the careful segmentation of users and infrastructure. The following example shows how rings are used by a major team at Microsoft.

Ring	Purpose	Users	Data Center
0	Finds most of the user-impacting bugs introduced by the deployment	Internal only, high tolerance for risk and bugs	US West Central
1	Areas the team doesn't test extensively	Customers using a breadth of the product	A small data center
2	Scale-related issues	Public accounts, ideally free ones using a diverse set of features	A medium or large data center
3	Scale issues in internal accounts and international related issues	Large internal accounts and European customers	Internal data center and a European data center
4	Remaining scale units	Everyone else	All deployment targets

Allow bake time

The term *bake time* refers to the amount of time a deployment is allowed to run before expanding to the next ring. Some issues may take hours or longer to start showing symptoms, so the release should be in use for an appropriate amount of time before it's considered ready.

In general, a 24-hour day should be enough time for most scenarios to expose latent bugs. However, this period should include a period of peak usage, requiring a full business day, for services that peak during business hours.

Expedite hotfixes

A *live site incident* (LSI) occurs when a bug has a serious impact in production. LSIs necessitate the creation of a *hotfix*, which is an out-of-band update designed to address a high-priority issue.

If a bug is *Sev 0*, the most severe type of bug, the hotfix may be deployed directly to the impacted scale unit as quickly as responsibly possible. While it's critical that the fix not make things worse, bugs of this severity are considered so disruptive that they must be addressed immediately.

Bugs rated *Sev 1* must be deployed through ring 0, but can then be deployed out to the affected scale units as soon as approved.

Hotfixes for bugs with lower severity must be deployed through all rings as planned.

Key takeaways

Every team wants to deliver updates quickly and at the highest possible quality. With the right practices, delivery can be a productive and painless part of the DevOps cycle.

- Deploy often.
- Stay green throughout the sprint.
- Use consistent deployment tooling in development, test, and production.
- Use a continuous delivery platform that allows automation and authorization.
- Follow safe deployment practices.

Next steps

Learn how feature flags help control the exposure of new features to users.

Progressive experimentation with feature flags

Article • 11/28/2022

As DevOps teams shift to an Agile methodology that focuses on continuous delivery of features, the need to control how they become available to users becomes increasingly important. Feature flags are a great solution for limiting user access to new features, either for marketing purposes or for testing in production.

Decoupling deployment and exposure

With *feature flags*, a team can choose whether a given set of features is visible in the user experience and/or invoked within the functionality. New features can be built and deployed as part of the ordinary development process without having those features available for broad access. The deployment of features is conveniently decoupled from their exposure.

Flags provide runtime control down to individual user

Flags also provide granular control all the way down to the individual user. When it's time to enable a feature, whether for one user, a small group, or everyone, the team can simply change the feature flag in order to light it up without having to redeploy.

The scope of a feature flag will vary based on the nature of the feature and the audience. In some cases, a feature flag will automatically enable the functionality for everyone. In other cases, a feature will be enabled on a user-by-user basis. Teams can also use feature flags to allow users to opt in to enable a feature, if they so desire. There's really no limit to the way the feature flags are implemented.

Support early feedback and experimentation

Feature flags are a great way to support early experimentation. Some features can have rough edges early on, which may be interesting only to the earliest of adopters. Trying to push those not-quite-ready features onto a broader audience could produce dissatisfaction. But the benefit of gathering feedback from users willing to deal with an in-progress feature is invaluable.

Quick off switch

Sometimes it's helpful to be able to turn something off. For example, suppose a new feature isn't working the way it was intended, and there are side effects that cause problems elsewhere. You can use feature flags to quickly turn off the new functionality in order to roll back to trusted behavior without having to redeploy. While feature flags are often thought of in terms of user interface features, they can also easily be used for changes in architecture or infrastructure.

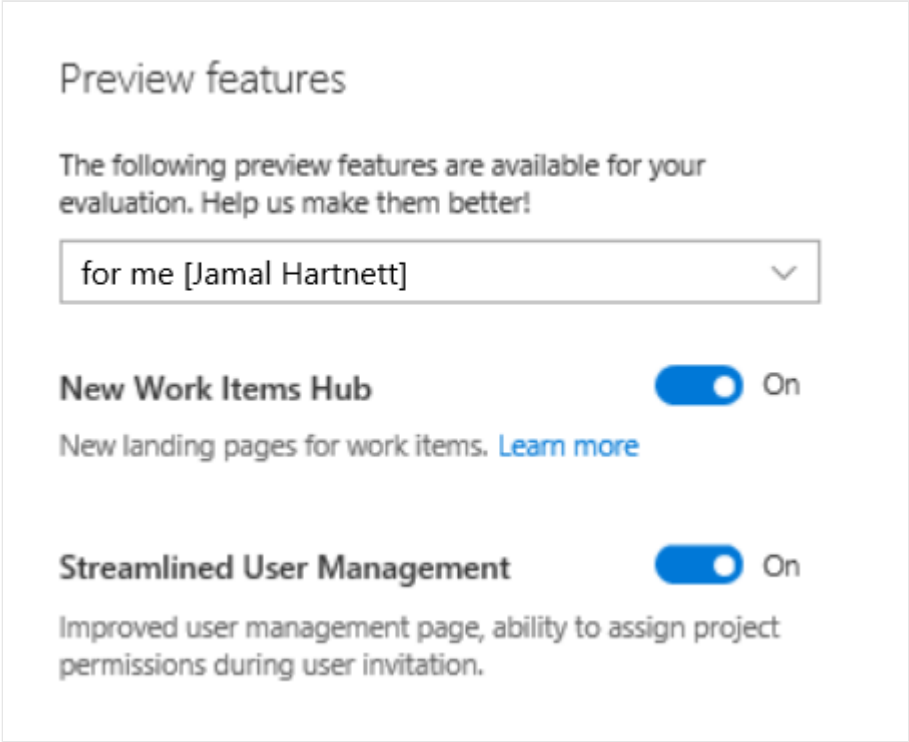
Standard stages

Microsoft uses a standard rollout process to turn on feature flags. There are two separate concepts: *rings* are for deployments, and *stages* are for feature flags. Learn more about rings and stages.

Stages are all about disclosure or exposure. For example, the first stage could be for a team's account and the personal accounts of members. Most users wouldn't see anything new because the only place flags are turned on is for this first stage. This allows a team to fully use and experiment with it. Once the team signs off, select customers would be able to opt into it via the second stage of feature flags.

Opt in

It's a good practice to allow users to opt in to feature flags when feasible. For example, the team may expose a preview panel associated with the user's preferences or settings.

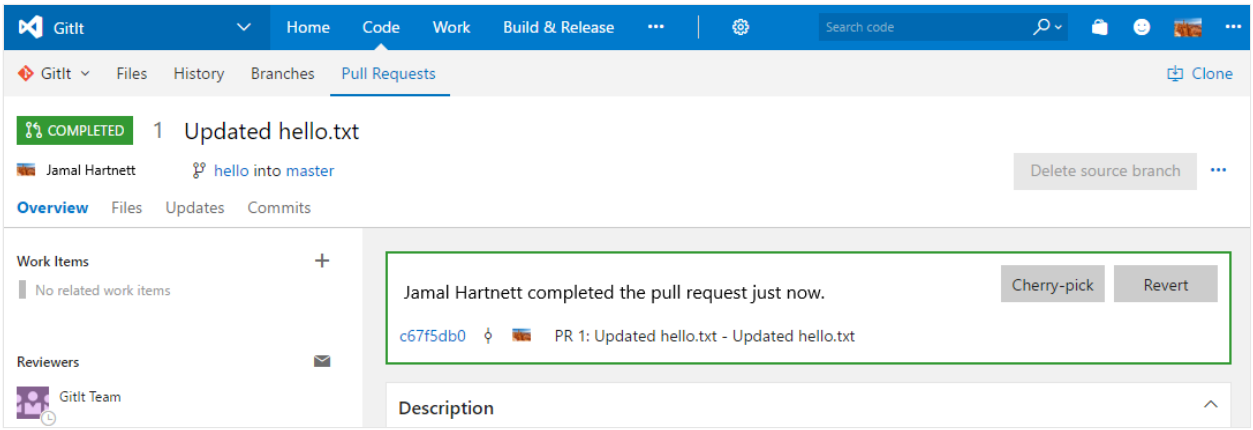


Use flags with telemetry

Feature flags provide a way to incrementally expose updates. However, teams must continuously monitor the right metrics to gauge readiness for broader exposure. These metrics should include usage behavior, as well as the impact of the updates on the health of the system. It's important to avoid the trap of assuming everything is okay just because nothing bad seems to be happening.

A feature flag example

Consider the example below. The team added a couple buttons here for **Cherry-pick** and **Revert** in the pull request UI. These were deployed using feature flags.



Define feature flags

The first feature exposed was the **Revert** button. The solution uses an XML file to define all of the feature flags. There's one file per service in this case, which creates an incentive to remove old flags to prevent the section from getting really long. The team will delete old flags because there's a natural motivation to control the size of that file.

XML

```
<?xml version="1.0" encoding="utf-8"?>
<!--
  In this group we should register Azure DevOps specific features and sets
their states.
-->
<ServicingStepGroup name="AzureDevOpsFeatureAvailability" … >
  <Steps>
    <!-- Feature Availability -->
    <ServicingStep name="Register features"
stepPerformer="FeatureAvailability" … >
      <StepData>
        <!--specifying owner to allow implicit removal of features -->
        <Features owner="AzureDevOps">
           <!-- Begin TFVC/Git -->
```

```
            <Feature name="SourceControl.Revert" description="Source control
revert features" />
```

A common server framework encourages reuse and economies of scale across the whole team. Ideally, the project will have infrastructure in place so that a developer can simply define a flag in a central store and have the rest of the infrastructure handled for them.

Check feature flags at runtime

The feature flag used here is named **SourceControl.Revert**. Here's the actual TypeScript from that page that illustrates the call for a feature availability check.

TypeScript

```
private addRevertButton(): void {
 if (FeatureAvailability.isFeatureEnabled(Flags.SourceControlRevert)) {
     this._calloutButtons.unshift(
         <button onClick={ () => Dialogs.revertPullRequest(
             this.props.repositoryContext,
             this.props.pullRequest.pullRequestContract(),

this.props.pullRequest.branchStatusContract().sourceBranchStatus,

this.props.pullRequest.branchStatusContract().targetBranchStatus)
         }
         >
             {VCResources.PullRequest_Revert_Button}
         </button>
        );
     }
}
```

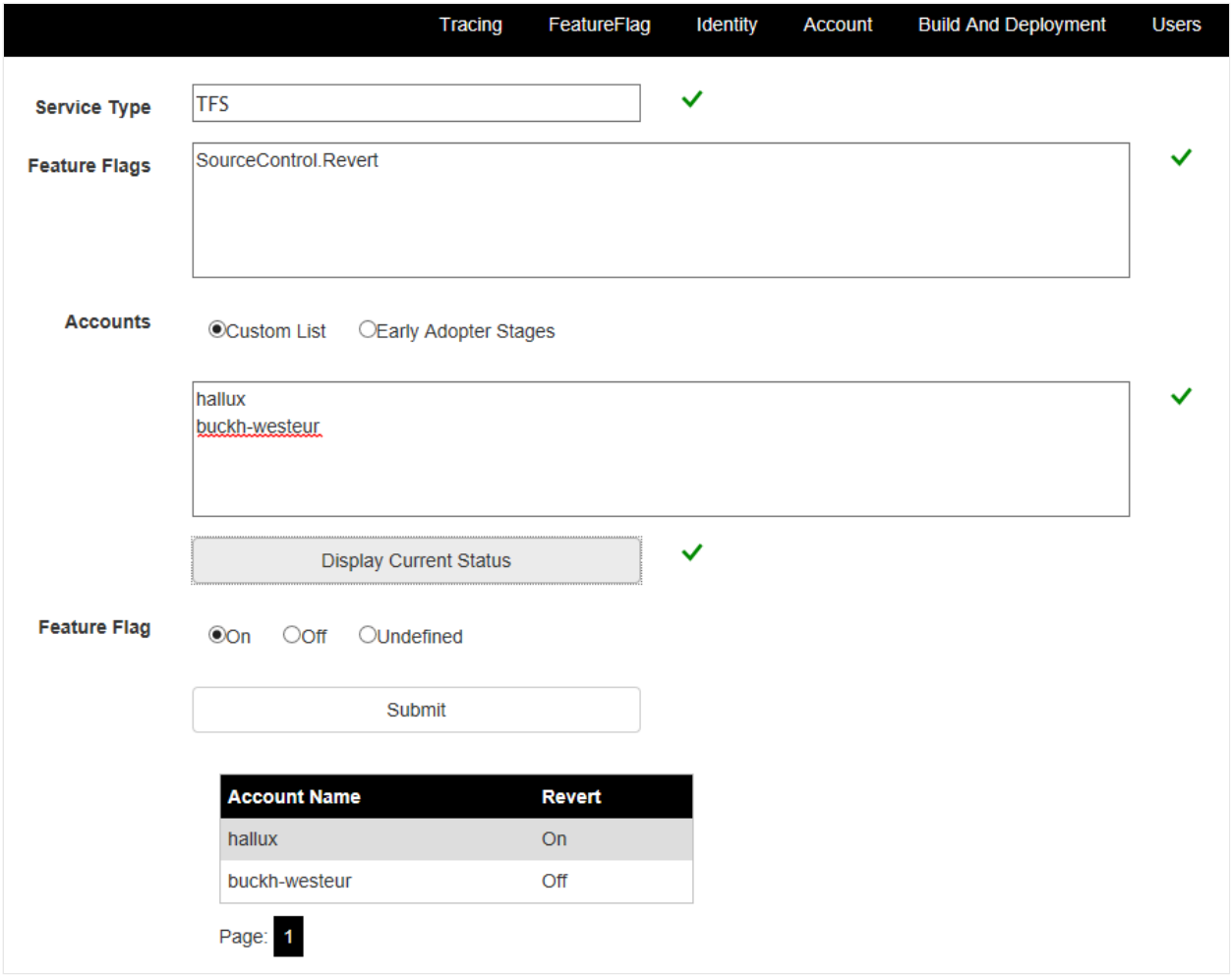
The example above illustrates usage in TypeScript, but it could just as easily be accessed using C#. The code checks to see if the feature is enabled and, if so, renders a button to provide the functionality. If the flag isn't enabled, then the button is skipped.

Control a feature flag

A good feature flag platform will provide multiple ways to manage whether a given flag is set. Typically, there are usage scenarios for the flag to be controlled via PowerShell and web interface. For PowerShell, all that really needs to be exposed are ways to get and set a feature flag's status, along with optional parameters for things like specific user account identifiers, if applicable.

Control feature flags through web UI

The following example uses the web UI exposed for this product by the team. Note the feature flag for **SourceControl.Revert**. There are two personal accounts listed here: **hallux** and **buckh-westeur**. The state is set for **hallux**, which happens to be in North Central, and cleared for the other account in West Europe.



The nature of the feature flag will drive the way in which the features are exposed. In some cases, the exposure will follow a ring and stage model. In others, users may opt in through configuration UI, or even by emailing the team for access.

Considerations for feature flags

Most feature flags can be retired once a feature has been rolled out to everyone. At that point, the team can delete all references to the flag in code and configuration. It's a good practice to include a feature flag review, such as at the beginning of each sprint.

At the same time, there may be a set of feature flags that persist for various reasons. For example, the team may want to keep a feature flag that branches something infrastructural for a period of time after the production service has fully switched over. However, keep in mind that this potential codepath could be reactivated in the future

during an explicit clearing of the feature flag, so it needs to be tested and maintained until the option is removed.

Feature flags and branching strategy

Feature flags enable development teams to include incomplete features in `main` without affecting anybody else. As long as the codepath is isolated behind a feature flag, it's generally safe to build and publish that code without side effects impacting normal usage. But if there are cases where a feature requires dependencies, such as when exposing a REST endpoint, teams must consider how those dependencies can create security or maintenance work even without the feature being exposed.

Feature flags to mitigate risk

Sometimes new features have the potential to introduce destructive or disruptive changes. For example, the product may be undergoing a transformation from a wide database schema to a long one. In that scenario, the developer should create a feature branch for a small amount of time. They then make the destabilizing changes on the branch and keep the feature behind a flag. A popular practice is for teams to then merge changes up to `main` as soon as they're not causing any harm. This wouldn't be feasible without the ability to keep the unfinished feature hidden behind a feature flag.

Feature flags help work in main

If you follow the common-sense practices discussed in the Develop phase, working in `main` is a good way to tighten a DevOps cycle. When combined with feature flags, developers can quickly merge features upstream and push them through the test gauntlet. Quality code can quickly get published for testing in production. After a few sprints, developers will recognize the benefits of feature flags and use them proactively.

How to decide whether to use a feature flag

The feature teams own the decision as to whether they need a feature flag or not for a given change. Not every change requires one, so it's a judgment call for a developer when they choose make a given change. In the case of the **Revert** feature discussed earlier, it was important to use a feature flag to control exposure. Allowing teams to own key decisions about their feature area is part of enabling autonomy in an effective DevOps organization.

Build vs. buy

While it's possible to build your own feature flag infrastructure, adopting a platform like LaunchDarkly is generally recommended. It's preferable to invest in building features instead of rebuilding feature flag functionality.

Next steps

Learn more about using feature flags in an ASP.NET Core app.

Eliminate downtime through versioned service updates

Article • 11/28/2022

Historically, administrators needed to take a server offline to update and upgrade on-premises software. However, downtime is a complete nonstarter for global 24×7 services. Many modern cloud services are a critical dependency for users to run their businesses. There's never a good time to take a system down, so how can a team provide continuous service while installing important security and feature updates?

Bu using versioned updates, these critical services can be transitioned seamlessly from one version to another *while customers are actively using them*. Not all updates are hard. Updating front-end layouts or styles is easy. Changes to features can be tricky, but there are well-known practices to mitigate migration risks. However, changes that emanate from the data tier introduce a new class of challenges that require special consideration.

Update layers separately

With a distributed online service in multiple datacenters and separate data storage, not everything can change simultaneously. If the typical service is split into application code and databases, which are presumably versioned independently of each other, one of those sides needs to absorb the complexity of handling versioning.

Often, versioning is easier to handle in the application code. Larger systems usually have quite a bit of legacy code, such as SQL that lives inside its databases. Rather than further complicating this SQL, the application code should handle the complexity. Specifically, you can create a set of factory classes that understand SQL versioning.

During every sprint, create a new interface with that version so there's always code that matches each database version. You can easily roll back any binaries during deployment. If something goes wrong after deploying the new binaries, revert to the previous code. If the binary deployment succeeds, then start the database servicing.

So how does this actually work? For example, assume that your team is currently deploying Sprint 123. The binaries understand Sprint 123 database schema and they understand Sprint 122 schema. The general pattern is to work with both versions/sprints N and N-1 of the SQL schema. The binaries query the database, determine which schema version they're talking to, and then load the appropriate binding. Then, the application code handles the case when the new data schema isn't yet available. Once

the new version is available, the application code can start making use of the new functionality that's enabled by the latest database version.

Roll forward only with the data tier

Once databases are upgraded, the service is in a *roll-forward* situation if a problem occurs. Online database migrations are complex and often multi-step, so rolling forward is usually the best way to address a problem. In other words, if the upgrade fails, then the rollback would likely fail as well. There's little value in investing in the effort to build and test rollback code that your team never expects to use.

Deployment sequence

Consider a scenario where you need to add a set of columns to a database and transform some data. This transition needs to be invisible to users, which means avoiding table locks as much as possible and then holding locks for the shortest time possible so that they aren't perceptible.

The first thing we do is manipulate the data, possibly in parallel tables using a SQL trigger to keep data in sync. Large data migrations and transformations sometimes have to be multi-step over several deployments across multiple sprints.

Once the extra data or new schema has been created in parallel, the team goes into *deployment mode* for the application code. In deployment mode, when the code makes a call to the database, it first grabs a lock on the schema and then releases it after running the stored procedure. The database can't change between the time the call to the database is issued and when the stored procedure runs.

The upgrade code acts as a schema writer and requests a writer lock on the schema. The application code takes priority in taking a reader lock, and the upgrade code sits in the background trying to acquire the writer lock. Under the writer lock, only a small number of very fast operations are allowed on the tables. Then the lock is released and the application records the new version of the database is in use and uses the interface that matches the new database version.

The database upgrades are all performed using a migration pattern. A set of code and scripts look at the version of the database and then make incremental changes to migrate the schema from the old to the new version. All migrations are automated and rolled out via release management service.

The web UI must also be updated without disrupting users. When upgrading JavaScript files, style sheets, or images, avoid mixing old and new versions being loaded by the

client. That can lead to errors that could lose work in progress, such as a field being edited by a user. Therefore, you should version all JavaScript, CSS, and image files by putting all files associated with a deployment into a separate, versioned folder. When the web UI makes calls back to the application tier, assets with a specified version are loaded. Only when a user action results in a full page refresh does the new web UI get loaded into the browser. The user's experience isn't disrupted by the upgrade.

Next steps

Microsoft has been one of the world's largest software development companies for decades. Learn how Microsoft operates reliable systems with DevOps.

How Microsoft operates reliable systems with DevOps

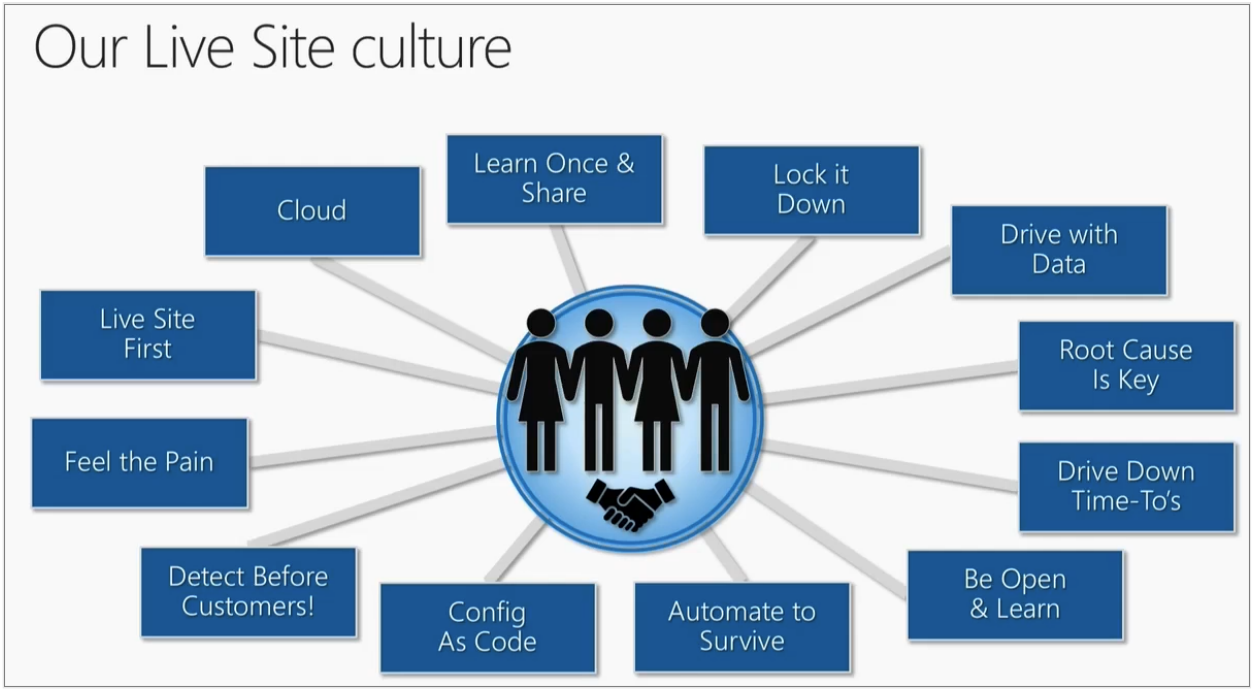
Article • 11/28/2022

Microsoft has been operating complex online platforms since the earliest days of the commercial internet. Along the way, we've evolved a substantial set of practices to keep systems available, healthy, and secure. These practices are part of a larger initiative to maintain and improve a *live site culture*.

Live site culture

Live site culture is the focus of an organization to prioritize the experience and reliability of the live site over everything else. After all, customers can move across service providers fairly easily nowadays with the cloud and internet-based services, greatly amplifying the importance of customer trust. The live site must always be available and perform as-promised to customers.

There are various factors that contribute to a successful live site culture.



Live site first

Putting the live site experience first is integral to a successful platform. Teams can't put all of their focus on new, shiny features and disregard the avenue in which those features are presented to users. We rely on safe deployment practices that help ensure

that our customers enjoy uninterrupted platform access. This can get especially complicated when it comes to releasing versioned service updates with no downtime.

Control exposure through feature flags

As we deploy out through our rings and stages, controlling exposure with feature flags, we occasionally discover an issue in production. Despite all our automation and reviews, things sometimes still happen. As they say, *there's no place like production!*

Usually, health monitoring and telemetry alert us when something isn't right. A developer can create a branch off `main`, make a fix, and pull request it into `main`. Keeping the same general workflow means that developers don't have to context-switch or learn a different process for a different code change.

To address a hotfix deployment, one more step is required, which is 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 `main` first, we know it won't regress the next sprint when a new release branch is created from `main`.

Releases of on-premises products 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.

Security should be taken personally

The focus is to make vulnerabilities real and personal. This ensures that people really care. We also make extensive use of war games to find and address security risks throughout the system, whether in code or not. When the red team can show that they got into code by turning a dialog box upside down, it really motivates the code owner to address the issue and make sure it doesn't happen again anywhere else. That sort of competition is much more real and personal than a static analysis warning about a potential XSS risk. We create this kind of culture and dynamic through war games and other security exercises. People take pride in hacking into each other's code or being able to block the attempts. This instills a secure code culture.

We can't plan for every attack vector, but what we can do is assume that there's going to be a breach, and plan how fast we can react to that breach. A lot of the security work has been around that for our teams.

Finally, humans make mistakes. They sometimes get lazy and do things like store passwords on file shares. We can tell them not to and we can send them to security training and we can do all sorts of other things. Most people learn, but it only takes one person to break the system. You can have all sorts of lists of best practices but unless you're making that real, you have to assume that people are going to make mistakes. This requires a certain level of oversight to ensure critical processes are being followed.

Engineering is more than an ops partner

We learned early on to make the live site an important part of the engineering team's responsibilities. That was huge for us because, in the past, one person could go deploy something, leave for the weekend, and return Monday to find 900 customer issues that the customer support and ops teams were dealing with all weekend. It's important that engineering pay the price for live site issues. Otherwise there's no incentive to build systems that avoid those problems. When you get called at 2 A.M. to fix something you broke, you remember.

As we evolved this responsibility, *Live site is the most important thing that we do* became the whole team's mantra. It's the customer experience they have right now and it's not just a tax. It's actually something people count on from us and we take pride in it. It needs to be a differentiating feature of our product.

Production telemetry is the heartbeat of your service

In order to survive in the fast-paced world where virtually anything can go wrong, we need great alerting systems. Unactionable alerts, redundant alerts, or overwhelming alert volumes make you ignore all the alerts. It's easy to create way too many alerts, so the process really distills down to a simple question: *Is this alert actionable?* This ensures we're engaging on the right customer issues and handling them as quickly as possible.

As the engineering team zeroed in on actionable alerts, they noticed that a lot of problems that come up, especially in the middle of the night, tend to have similar fixes, at least temporarily. This resulted in a focus on systems that were better at failing over and self-healing. Now the issues happen, raise alerts, and then fix themselves well enough for the engineering team to wait until morning to fix. This wouldn't have happened if the engineering team just pushed out bits that kept other people up at night. Now they work to balance these improvements as a part of not just feature velocity, but engineering improvement velocity.

Summary

Adopting a live site culture has impacted the way Microsoft builds and delivers software. By making engineering teams a key part of security and operations, the quality of our code and end-user experience have improved drastically. Being a full participant in operations has made engineering a key stakeholder, resulting in systems that are designed for better operations.

Security in DevOps (DevSecOps)

Article • 11/28/2022

Security is a key part of DevOps. But how does a team know if a system is secure? Is it really possible to deliver a completely secure service?

Unfortunately, the answer is *no*. DevSecOps is a continuous and ongoing effort that requires the attention of everyone in both development and IT operations. While the job is never truly done, the practices that teams employ to prevent and handle breaches can help produce systems that are as secure and resilient as possible.

> "Fundamentally, if somebody wants to get in, they're getting in...accept that. What we tell clients is: number one, you're in the fight, whether you thought you were or not. Number two, you almost certainly are penetrated." -- Michael Hayden, Former Director of NSA and CIA

The security conversation

Teams that don't have a formal DevSecOps strategy are encouraged to begin planning as soon as possible. At first there may be resistance from team members who don't fully appreciate the threats that exist. Others may not feel that the team is equipped to face the problem and that any special investment would be a wasteful distraction from shipping features. However, it's necessary to begin the conversation to build consensus as to the nature of the risks, how the team can mitigate them, and whether the team needs resources they don't currently have.

Expect skeptics to bring some common arguments, such as:

- **How real is the threat?** Teams often don't appreciate the potential value of the services and data they're charged with protecting.
- **Our team is good, right?** A security discussion may be perceived as doubt in the team's ability to build a secure system.
- **I don't think that's possible.** This is a common argument from junior engineers. Those with experience usually know better.
- **We've never been breached.** But how do you know? How *would* you know?
- **Endless debates about value.** DevSecOps is a serious commitment that may be perceived as a distraction from core feature work. While the security investment should be balanced with other needs, it can't be ignored.

The mindset shift

DevSecOps culture requires an important shift in mindset. Not only do you need to *prevent* breaches, but *assume* them as well.

Security strategy components

There are many techniques that can be applied in the quest for more secure systems.

Preventing breaches	**Assuming breaches**
Threat models	War game exercises
Code reviews	Central security monitors
Security testing	Live site penetration tests
Security development lifecycle (SDL)	

Every team should already have at least some practices in place for preventing breaches. Writing secure code has become more of a default, and there are many free and commercial tools to aid in static analysis and other security testing features.

However, many teams lack a strategy that assumes system breaches are inevitable. Assuming that you've been breached can be hard to admit, especially when having difficult conversations with management, but that assumption can help you answer questions about security on your own time. You don't want to figure it all out during a real security emergency.

Common questions to think through include:

- How will you detect an attack?
- How will you respond if there is an attack or penetration?
- How will you recover from an attack, such as when data has been leaked or tampered with?

Key DevSecOps practices

There are several common DevSecOps practices that apply to virtually any team.

First, focus on improving *mean time to detection* and *mean time to recovery*. These metrics indicate how long it takes to detect a breach and how long it takes to recover, respectively. They can be tracked through ongoing live site testing of security response plans. When evaluating potential policies, improving these metrics should be an important consideration.

Practice *defense in depth*. When a breach happens, attackers can get access to internal networks and everything inside them. While it would be ideal to stop attackers before it gets that far, a policy of assuming breaches drives teams to minimize exposure from an attacker who has already gotten in.

Finally, perform periodic post-breach assessments of your practices and environments. After a breach has been resolved, your team should evaluate the performance of the policies, as well as their own adherence to them. Policies are most effective when teams actually follow them. Every breach, whether real or practiced, should be seen as an opportunity to improve.

Strategies for mitigating threats

There are too many threats to enumerate them all. Some security holes are due to issues in dependencies like operating systems and libraries, so keeping them up-to-date is critical. Others are due to bugs in system code that require careful analysis to find and fix. Poor secret management is the cause of many breaches, as is social engineering. It's a good practice to think about the different kind of security holes and what they mean to the system.

Attack vectors

Consider a scenario where an attacker has gained access to a developer's credentials. What can they do?

Privilege	**Attack**
Can they send emails?	Phish colleagues
Can they access other machines?	Log on, mimikatz, repeat
Can they modify source	Inject code
Can they modify the build/release process?	Inject code, run scripts
Can they access a test environment?	If a production environment takes a dependency on the test environment, exploit it
Can they access the production environment?	So many options...

How can your team defend against these vectors?

- Store secrets in protected vaults

- Remove local admin accounts
- Restrict SAMR
- Credential Guard
- Remove dual-homed servers
- Separate subscriptions
- Multi-factor authentication
- Privileged access workstations
- Detect with ATP & Microsoft Defender for Cloud

Secret management

All secrets must be stored in a protected vault. Secrets include:

- Passwords, keys, and tokens
- Storage account keys
- Certificates
- Credentials used in shared non-production environments, too

You should use a hierarchy of vaults to eliminate duplication of secrets. Also consider how and when secrets are accessed. Some are used at deploy-time when building environment configurations, whereas others are accessed at run-time. Deploy-time secrets typically require a new deployment in order to pick up new settings, whereas run-time secrets are accessed when needed and can be updated at any time.

Platforms have secure storage features for managing secrets in CI/CD pipelines and cloud environments, such as Azure Key Vault and GitHub Actions.

Helpful tools

- Microsoft Defender for Cloud is great for generic infrastructure alerts, such as for malware, suspicious processes, etc.
- Source code analysis tools for static application security testing (SAST).
- GitHub advanced security for analysis and monitoring of repos.
- mimikatz extracts passwords, keys, pin codes, tickets, and more from the memory of `lsass.exe`, the Local Security Authority Subsystem Service on Windows. It only requires administrative access to the machine, or an account with the debug privilege enabled.
- BloodHound builds a graph of the relationships within an Active Directory environment. It can be used the red team to easily identify attack vectors that are difficult to quickly identify.

War game exercises

A common practice at Microsoft is to engage in *war game exercises*. These are security testing events where two teams are tasked with testing the security and policies of a system.

The *red* team takes on the role of an attacker. They attempt to model real-world attacks in order to find gaps in security. If they can exploit any, they also demonstrate the potential impact of their breaches.

The *blue* team takes on the role of the DevOps team. They test their ability to detect and respond to the red team's attacks. This helps to enhance situational awareness and measure the readiness and effectiveness of the DevSecOps strategy.

Evolve a war games strategy

War games are effective at hardening security because they motivate the red team to find and exploit issues. It'll probably be a lot easier than expected early on. Teams that haven't actively tried to attack their own systems are generally unaware of the size and quantity of security holes available to attackers. The blue team may be demoralized at first since they'll get run over repeatedly. Fortunately, the system and practices should evolve over time such that the blue team consistently wins.

Prepare for war games

Before starting war games, the team should take care of any issues they can find through a security pass. This is a great exercise to perform before attempting an attack because it will provide a baseline experience for everyone to compare with after the first exploit is found later on. Start off by identifying vulnerabilities through a manual code review and static analysis tools.

Organize teams

Red and blue teams should be organized by specialty. The goal is to build the most capable teams for each side in order to execute as effectively as possible.

The red team should include some security-minded engineers and developers deeply familiar with the code. It's also helpful to augment the team with a penetration testing specialist, if possible. If there are no specialists in-house, many companies provide this service along with mentoring.

The blue team should be made up of ops-minded engineers who have a deep understanding of the systems and logging available. They have the best chance of detecting and addressing suspicious behavior.

Run early war games

Expect the red team to be effective in the early war games. They should be able to succeed through fairly simple attacks, such as by finding poorly protected secrets, SQL injection, and successful phishing campaigns. Take plenty of time between rounds to apply fixes and feedback on policies. This will vary by organization, but you don't want to start the next round until everyone is confident that the previous round has been mined for all it's worth.

Ongoing war games

After a few rounds, the red team will need to rely on more sophisticated techniques, such as cross-site scripting (XSS), deserialization exploits, and engineering system vulnerabilities. Bringing in outside security experts in areas like Active Directory may be helpful in order to attack more obscure exploits. By this time, the blue team should not only have a hardened platform to defend, but will also make use of comprehensive, centralized logging for post-breach forensics.

> "Defenders think in lists. Attackers think in graphs. As long as this is true, attackers win." -- John Lambert (MSTIC)

Over time, the red team will take much longer to reach objectives. When they do, it will often require discovery and chaining of multiple vulnerabilities to have a limited impact. Through the use of real-time monitoring tools, the blue team should start to catch attempts in real-time.

Guidelines

War games shouldn't be a free-for-all. It's important to recognize that the goal is to produce a more effective system run by a more effective team.

Code of conduct

Here is a sample code of conduct used by Microsoft:

1. Both the red and blue teams will do no harm. If the potential to cause damage is significant, it should be documented and addressed.

2. The red team should not compromise more than needed to capture target assets.
3. Common sense rules apply to physical attacks. While the red team is encouraged to be creative with non-technical attacks, such as social engineering, they shouldn't print fake badges, harass people, etc.
4. If a social engineering attack is successful, don't disclose the name of the person who was compromised. The lesson can be shared without alienating or embarrassing a team member everyone needs to continue to work with.

Rules of engagement

Here are sample rules of engagement used by Microsoft:

1. Do not impact the availability of any system.
2. Do not access external customer data.
3. Do not significantly weaken in-place security protections on any service.
4. Do not intentionally perform destructive actions against any resources.
5. Safeguard credentials, vulnerabilities, and other critical information obtained.

Deliverables

Any security risks or lessons learned should be documented in a backlog of repair items. Teams should define a service level agreement (SLA) for how quickly security risks will be addressed. Severe risks should be addressed as soon as possible, whereas minor issues may have a two-sprint deadline.

A report should be presented to the entire organization with lessons learned and vulnerabilities found. It's a learning opportunity for everyone, so make the most of it.

Lessons learned at Microsoft

Microsoft regularly practices war games and has learned a lot of lessons along the way.

- War games are an effective way to change DevSecOps culture and keep security top-of-mind.
- Phishing attacks are very effective for attackers and should not be underestimated. The impact can be contained by limiting production access and requiring two-factor authentication.
- Control of the engineering system leads to control of everything. Be sure to strictly control access to the build/release agent, queue, pool, and definition.
- Practice defense in depth to make it harder for attackers. Every boundary they have to breach slows them down and offers another opportunity to catch them.

- Don't ever cross trust realms. Production should never trust anything in test.

Next steps

Learn more about the security development lifecycle and DevSecOps on Azure.

Enable DevSecOps with Azure and GitHub

Article • 11/28/2022

DevSecOps, sometimes called Secure DevOps, builds on the principles of DevOps but puts security at the center of the entire application lifecycle. This concept is called "shift-left security": it moves security upstream from a production-only concern to encompass the early stages of planning and development. Every team and person that works on an application is required to consider security.

Microsoft and GitHub offer solutions to build confidence in the code that you run in production. These solutions inspect your code and allow its traceability down to the work items and insights on the third-party components that are in use.

Secure your code with GitHub

Developers can use code scanning tools that quickly and automatically analyze the code in a GitHub repository to find security vulnerabilities and coding errors.

You can scan code to find, triage, and prioritize fixes for existing problems. Code scanning also prevents developers from introducing new problems. You can schedule scans for specific days and times, or trigger scans when a specific event occurs in the repository, such as a push. You can also track your repository's dependencies and receive security alerts when GitHub detects vulnerable dependencies.

- Scan your code with CodeQL and token scanning
- Manage security advisories for your projects
- Secure your code's dependencies with Dependabot

Track your work with Azure Boards

Teams can use Azure Boards web service to manage software projects. Azure Boards provides a rich set of capabilities, including native support for Scrum and Kanban, customizable dashboards, and integrated reporting.

- Plan and track work with Azure Boards
- Connect Azure Boards with GitHub

Build and deploy containers with Azure Pipelines

Integrate Azure Pipelines and Kubernetes clusters with ease. You can use the same YAML documents to build multi-stage pipelines-as-code for both continuous integration and continuous delivery.

Azure Pipelines integrates metadata tracing into your container images, including commit hashes and issue numbers from Azure Boards, so that you can inspect your applications with confidence.

The ability to create deployment pipelines with YAML files and store them in source control helps drive a tighter feedback loop between development and operation teams who rely on clear, readable documents.

- Store Docker images in Azure Container Registry
- Build a Docker image with Azure Pipelines
- Deploy to Kubernetes with full traceability
- Secure your Azure Pipelines

Run and debug containers with Bridge to Kubernetes

Developing a Kubernetes application can be challenging. You need Docker and Kubernetes configuration files. You need to figure out how to test your application locally and interact with other dependent services. You might need to develop and test multiple services at once and with a team of developers.

Bridge to Kubernetes allows you to run and debug code on your development computer, while still connected to your Kubernetes cluster with the rest of your application or services. You can test your code end-to-end, hit breakpoints on code running in the cluster, and share a development cluster between team members without interference.

- Learn more about Bridge to Kubernetes

Enforce container security with Microsoft Defender for Containers and Azure Policy

Microsoft Defender for Containers is the cloud-native solution for securing your containers.

- Overview of Microsoft Defender for Containers
- Understand Azure Policy for Kubernetes clusters
- Azure Kubernetes Service (AKS)

Manage identities and access with the Microsoft identity platform

The Microsoft identity platform is an evolution of the Azure Active Directory (Azure AD) developer platform. It allows developers to build applications that sign in all Microsoft identities and get tokens to call Microsoft APIs, such as Microsoft Graph, or APIs that developers have built.

- Microsoft identity platform documentation

Azure AD B2C provides business-to-customer identity as a service. Your customers use their preferred social, enterprise, or local account identities to get single sign-on access to your applications and APIs.

- Azure AD B2C documentation

Access management for cloud resources is a critical function for any organization that uses the cloud. Azure role-based access control (Azure RBAC) helps you manage who has access to Azure resources, what they can do with those resources, and what areas they can access.

- Learn about access management with Azure RBAC

You can use the Microsoft identity platform to authenticate with the rest of your DevOps tools, including native support within Azure DevOps and integrations with GitHub Enterprise.

- Authenticate to GitHub Enterprise

Currently, an Azure Kubernetes Service (AKS) cluster (specifically, the Kubernetes cloud provider) requires an identity to create additional resources like load balancers and managed disks in Azure. This identity can be either a managed identity or a service principal. If you use a service principal, you must either provide one or AKS creates one on your behalf. If you use managed identity, one will be created for you by AKS automatically. For clusters that use service principals, the service principal must be renewed eventually to keep the cluster working. Managing service principals adds

complexity, which is why it's easier to use managed identities instead. The same permission requirements apply for both service principals and managed identities.

Managed identities are essentially a wrapper around service principals, and make their management simpler.

- Use managed identities in AKS

Manage keys and secrets with Azure Key Vault

Azure Key Vault can be used to securely store and control access to tokens, passwords, certificates, API keys, and other secrets. Centralizing storage of application secrets in Key Vault allows you to control their distribution. Key Vault greatly reduces the chances that secrets may be accidentally leaked. When you use Key Vault, application developers no longer need to store security information in their application, which eliminates the need to make this information part of the code. For example, an application may need to connect to a database. Instead of storing the connection string in the app's code, you can store it securely in Key Vault.

- Store certificates, keys, and secrets with Azure Key Vault

Monitor your applications

With Azure Monitor, you can monitor both your application and infrastructure in real-time, identifying issues with your code and potential suspicious activities and anomalies. Azure Monitor integrates with release pipelines in Azure Pipelines to enable automatic approval of quality gates or release rollback based on monitoring data.

Learn how to monitor your applications and infrastructure using Azure Application Insights and Azure Monitor.

- Application performance management with Application Insights
- Monitor containerized applications with Azure Monitor

Build the right architecture

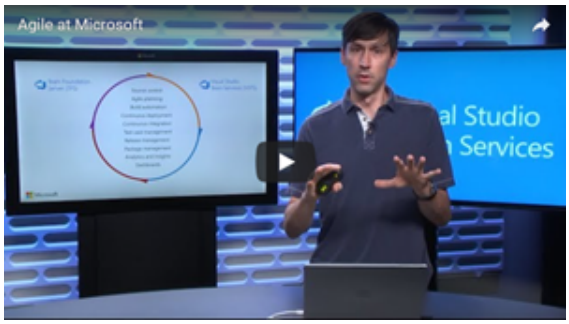
Security is one of the most important aspects of any architecture. Security provides confidentiality, integrity, and availability assurances against deliberate attacks and abuse of your valuable data and systems. Losing these assurances can negatively impact your business operations and revenue, as well as your organization’s reputation in the marketplace.

- Applications and services architecture
- DevSecOps architecture

DevOps events and talks

Article • 11/28/2022

Highlights

Introduction to Azure DevOps	Agile at Microsoft
	

Azure DevOps services

Videos and presentation decks for Azure DevOps services.

TITLE	DESCRIPTION	VIDEO	DOWNLOAD
Azure DevOps overview	Plan smarter, collaborate better, and ship faster with a set of modern developer services.	Video	PPT
Plan your work with Azure Boards	Anyone who works on software projects knows that there are issues to track, manage, and prioritize. Azure Boards has all the features your team needs to successfully manage your work. Visualize projects with Kanban boards, execute in sprints, manage your backlog, and use queries to find work and visualize results. Learn how to get started with Azure Boards.	Video	PPT
Manage and store your code in Azure Repos	If you write code, then you need a place to store and manage that code with a reliable version control system like Git. Azure Repos provides a best-in-class Git solution. You get free private and public repos, social code reviews, and more. Learn how to get started with Git in Azure Repos and how your team can use pull requests to work together on code.	Video	

TITLE	DESCRIPTION	VIDEO	DOWNLOAD
Use Azure Pipelines to add continuous builds to GitHub projects	Learn how to take a GitHub repo and add continuous builds using Azure Pipelines. You'll see each step in taking a Node.js GitHub project and adding continuous builds to validate the code quality of each pull request. Azure Pipelines is free for open-source projects.	Video	
Build and deploy your code with Azure Pipelines	With Azure Pipelines, you can build and deploy code written in any language, using any platform. In this video, you'll learn why Azure Pipelines is the best tool on the planet for continuous integration and continuous deployment (CI/CD) of your code.	Video	
Get started with Azure Artifacts	Azure Artifacts helps you manage software components by providing an intuitive UI, as well as helpful tools to ensure immutability and performance for the components you create or consume. Learn how to get started by creating a feed for an npm package to use in your Azure Pipeline.	Video	PPT
Automated and manual testing with Azure Test Plans	Azure DevOps Test Plan provides all the tools you need to successfully test your applications. Create and run manual test plans, generate automated tests, and collect feedback from users. In this video, you'll see the basic aspects on how to get started with Azure Test Plan, so you can start testing your application today.	Video	

Azure DevOps Server

Presentation deck for Azure DevOps Server.

TITLE	DESCRIPTION	DOWNLOAD
Azure DevOps Server	Share code, track work, and ship software using integrated developer tools, hosted on-premises.	PPT

Azure DevOps lessons learned and journey stories

Videos and presentation decks for Azure Devops lessons learned and journey stories.

TITLE	DESCRIPTION	VIDEO	DOWNLOAD
60,000 tests in six minutes: Create a reliable testing pipeline & deploy safely with Azure Pipelines	Good test coverage is essential for catching issues before a pull request has been merged, but they have to be the right kind of tests and must be reliable. Sam Guckenheimer digs into the testing transformation his team at Microsoft underwent as they started on their DevOps journey. He walks you through the changes they went through and why, and explains the data they found to prove their case for change and what they did to move. Sam also details which things are best covered by unit tests, which you should leave to manual code review in the pull request, and which are best suited to testing in production.	Video	PPT
Progressive deployment, experimentation, multitenancy, no downtime, cloud security	This experience report is about changing architecture from a monolith to cloud-native practices. It covers moving stepwise from single tenancy to multitenancy, scaling up to scaling out, fixed resources to optimized variable costs, periodic upgrades to zero-downtime updates, single backlog to continual experimentation, linear to progressive deployment with a controlled blast radius, long release cycles to continual testing, opacity to observability, and pre-release security reports to continuous security practices.	Video	PPT
DevOps for AI	Because the AI field is young compared to traditional software development, best practices and solutions around life cycle management for these AI systems have yet to solidify. This talk will discuss what Microsoft did in different departments, including Bing.	Video	PPT
Evolving Windows: This journey to DevOps	An overview of the journey Windows has been on to transform its process, tools, and culture to a DevOps model.	Video	PPT

Find more videos

- Azure DevOps on YouTube
- DevOps videos on Microsoft shows