Meeting the challenges of agile development at enterprise scale

Microsoft IT develops and maintains software services for different groups within Microsoft. Over the last several years as part of our move to modern engineering, all our teams have adopted agile methodology. We found that it solved many of the problems arising from the traditional waterfall model, where projects are planned well in advance and can take months, or even years, to complete. Under this model, a service could be out of date almost as soon as we released it. Using agile software development for small services shortened our release cycles from months to weeks. And during the course of a project, our teams nimbly responded to changing requirements and better met current customer needs.

Agile worked well for us on smaller projects, but when we began using it at scale on large, enterprise-sized projects, we ran into bottlenecks that slowed down the work and reduced its quality. The problem was team size. Agile works best for small teams of up to nine members. The more people we added to work on large projects, the more inefficient our teams became. Some of the teams were very large—150 members or more—and were getting bogged down. Also, we couldn't predict the time and resources that would be required to complete a larger project that extended beyond a two-week sprint. This created budgeting and resourcing problems. We had to find a better way to use agile at enterprise scale.

To address these issues, we studied third-party frameworks for applying agile to enterprise projects, such as Scaled Agile Framework for Lean Software and Systems Engineering (SAFe), and used them as a basis for developing our own approach. We're now in the process of applying our framework to enterprise projects.

Our teams have started the journey to using agile at enterprise scale and are learning lessons along the way. Applying the agile process to this journey, we identify issues and pain points, and continuously strive to improve through incremental change. So far, we've handled many of the big challenges and are now fine-tuning the process. Customers like the quality we're delivering, leadership can effectively plan and budget, and team morale is up.

How our agile teams deliver services

To better understand how we're scaling agile, it's helpful to first know how the small agile teams in Microsoft IT function. Using the established agile model, small, cross-functional teams build and deliver services. Internal business owners or customers create user stories that define what the service must provide—the business value. User stories are written in a simple and concise way with few details. In a clear and simple manner, they define the business value that developers must deliver during a two-week sprint. The agile team then breaks user stories into tasks that can each be completed in a day. Using this methodology, teams quickly iterate and solve problems based on real-world information and input from customers.

Our small agile teams have people in the following roles:

- **Product owner.** This person corresponds to the business owner in the agile model. The product owner acts on behalf of the customer and helps ensure that the work meets customer needs.
- **Program manager.** This person coordinates the work of the team as it converts user stories into tasks. During a sprint, the scrum master leads daily 15-minute standup meetings and tracks progress.
- **Software engineers.** These people design, code, build, test, and release the software.
- **Service engineers.** Service engineers coordinate:
  - Incident management.
  - Change requests.
  - Legacy release methodologies, including updates, patches, and release management.
  - Legacy issues pertaining to datacenter and cloud core acquisition.
  - Any required service monitoring.

This role is constantly changing as we migrate to modern service engineering best practices.
Figure 1 illustrates a standard two-week sprint during which the team plans, executes, and delivers business value.

![Figure 1. A small agile team delivers business value](image)

Challenges of building larger solutions

Our Microsoft IT team that supports sales and marketing was an early adopter of agile. Like other teams in our organization, we wanted to improve responsiveness to customers and gain efficiencies by adopting modern engineering tools and processes. Over time, along with other teams, we met the challenges of changing our processes and tools and evolving our culture to support our goals.

When we first moved to this model, we started off with small, low-level development projects that had few dependencies and had good success. However, when we branched out from developing small services to larger-scale solutions that bring together multiple services and multiple agile teams, we ran into problems.

The ideal size for an agile team is five to nine members. To build larger enterprise-scale solutions, we needed more people. We started adding more teams and adding more people to those teams; in some cases, going well beyond nine members. As each team grew, so did its complexity. The more people we added, the slower the teams’ ability to deliver became—approaching their sitting idle. Daily standup meetings got longer and longer. People on the same team ended up working on radically different tasks that were difficult to coordinate. We were continually facing these challenges:

- **Poorly understood and managed dependencies.** When we added too many members to a team, the work became inefficient and difficult to coordinate. But when we tried using small teams, with some specialized team members, we got bottlenecks. For example, say some teams are trying to build a quilt together, and one team is responsible for raw materials. Another team is working on corners, another on batting (quilt stuffing), and another on part of a square. Several teams are asking for raw material, so the raw-material team must stack-rank the requests. Some teams will have their progress blocked because they’re lower in the stack and have to wait for material.

- **Difficulty setting expectations.** We weren’t planning beyond the next sprint, so we didn’t know how big any project was going to be. That meant we couldn’t tell product owners when we’d finish the work. In turn, the product owners couldn’t set expectations with their customers.

- **Missed deadlines.** For large projects, teams couldn’t hit deadlines that were set because they didn’t have a clear understanding of how much work was involved.

- **Overworked team members.** As we tried to take on big projects, our productivity went down, and team management was pressured to add more developers and push people to work extra hours. This only made matters worse. Because no one knew how much work was required for a project, developers often found themselves buried under excessive workloads as they tried to complete projects with arbitrary deadlines.
• **Reduced quality.** As we scaled up, quality started going down. Team members were burning out from doing too much work. They stopped thinking and just did what they were told. New features got priority, so sustained engineering and current product issues didn’t get the attention they needed.

• **Difficulty budgeting.** It was hard to tell how much money would be needed to complete projects when we didn’t understand their true scope. Customers were dissatisfied with the results we delivered. They saw agile as a step backward from rational waterfall methods—into chaos.

**Our framework for agile at scale**

We wanted to retain the efficiency and effectiveness of small agile teams, but enable them to work together on large projects. Referring to third-party frameworks, such as SAFe, we developed a framework that scales up agile in levels. From the top down, the framework levels are:

• **Portfolio.** At the top of our framework, this is where product suites are conceived and their development coordinated. At this level, we create and manage epics and scenarios. An epic is a group of services that are integrated into a single offering. An analogy might be a car model. Scenarios are large components of the epic, for example the car body, frame, suspension, or electrical system. Epics and scenarios are described in plain language that anyone should be able to understand. Planning consists of an annual roadmap. Epics generally take between two and six quarters to complete. A scenario takes more than one quarter to complete. A principal program manager, general manager, or director oversees the work at this level, depending on the span of the project.

• **Program.** At this level, we manage feature development. Features are pieces of a scenario. We call them increments. Using the car example, if the electrical system were a scenario, increments might include things like the fuse box, cigarette lighter adaptor, wiring, and so forth. Each small agile development team has a representative on the program team. The work of the program team involves integrating small agile team deliverables, feature planning, and making sure work aligns with customer requirements. Feature planning is done quarterly—each increment is completed in one quarter. A program manager oversees the work of the small agile teams and aligns it with customer requirements. Instead of the daily standup meetings in the small agile teams, we have weekly or biweekly alignment meetings for small team representatives. If a feature is part of a larger suite, each feature team sends a representative to the portfolio team where the feature is integrated.

• **Team/execution.** At the base of the framework are the small agile development teams, ideally limited to nine members each. Coding takes place in the classic agile manner, based on user stories and tasks. Business value is delivered during two-week sprints.

![Diagram of Our scaled agile framework](image-url)

*Figure 2. Our scaled agile framework*
The next figure shows how the scaled agile process works at the program level.

Figure 3. At the program level, agile team deliverables are integrated into features, called increments.

Areas of focus when scaling up agile

Our framework is the foundation for using agile at scale for enterprise projects. For projects to be successful within the new framework, we must continually focus on reducing dependencies, balancing autonomy and alignment, changing our culture, and decoupling sprints from releases.

Reducing dependencies

We’ve learned that it’s important to constantly work at reducing dependencies, which can slow down the work and create bottlenecks. On a case-by-case basis, we identify dependencies and determine their impact, then remove or mitigate them. Dependencies can occur in the areas of teams, architecture, and process.

Teams

The way that we organize and structure our teams affects dependencies. For this reason, we’re moving away from a horizontally aligned team structure to a vertical one—from specialized teams to teams that can holistically solve their own problems.

For example, when several teams rely on a single database team, the database team must stack rank and prioritize its work. This creates a bottleneck. To avoid this, each team must be able to function autonomously with the complete set of skills necessary to complete their work.

We’re also working on organizing our geographically remote teams so they can be relatively independent and only touch base periodically at the sprint or increment level. For example, some of our teams are in India. We try to give those teams their own work, so they don’t have to constantly communicate with our Redmond teams, which would slow them down.

Architecture

We try to design our products to reduce dependencies and bottlenecks and avoid duplication of effort. We don’t want multiple teams to depend on a single product or function. When we can’t completely remove a dependency, we try to reduce its impact. For example, five of our teams in the sales and marketing area use an SAP system to make payments. Although they work on a two-week sprint cycle, SAP updates are released only once every six months. Staying in sync with these releases was burdensome, and each team managed the synchronization work separately. To fix the problem, we created a team to build an abstraction layer called Pay as a Service. It has simple input and
output architecture and handles the sync issues that individual teams used to manage themselves. Now our teams can move quicker because they’re not duplicating effort with everyone solving the same problem.

Processes
Processes include tracking and organizing data, change approval board (CAB) meetings, daily cross-team triage meetings, and Helpdesk. Shared processes increase alignment between teams and give us economy of scale. On the other hand, processes can create a lot of overhead, so we’re thoughtful about where it’s really necessary to align teams. We want to make sure that any process we adopt has a clear benefit, and that the impact on our teams is minimal.

Focusing on process also involves focusing on continuous improvement of the team and asking how we can keep reducing dependencies.

Balancing autonomy and alignment
Putting together a solution from a set of services is like having each agile team work on part of a quilt. If everyone goes off and works on their own, without coordination, what will happen when they stitch together all the independently built sections of the quilt? The result may not be as expected or desired. Also, the work may be done inefficiently. So we need a level of coordination between teams.

On the other hand, our agile teams want to move quickly and continuously improve. Being autonomous may be the easiest way for them to do this, but with large projects, each team is a member of a larger team and must coordinate with other teams. To balance team goals with the requirements of large projects, we determine where teams need to be aligned and where they can be autonomous.

We do this bearing in mind that it’s easy to take the term “alignment” and turn it into “standards” or “requirements.” If we try to give teams rigid details about how they can work, it hampers them. Instead, we align the small teams just enough to give the larger team the data and capabilities it needs to function. Our intention isn’t to make the small teams do things in a certain way, but to increase efficiency in the areas where they must work together with other teams. Within these constraints, each team is free to come up with solutions that work best for them. Everyone doesn’t have to work the same way, but all teams are aligned in key areas so the larger team can function.

We align teams in specific areas. Today, for example, all our teams have daily standup meetings, use the same processes and infrastructure, and share the same two-week delivery cadence. All teams have migrated to Visual Studio Online. Teams working on the same program also share the same backlog, and the order in which items are pulled off the backlog and assigned are based on the larger goals. We also use a common language for reporting on team progress, velocity, and demonstrating team value and impact. This gives us greater visibility into the future and some level of predictability.

We also remind the teams that, although using agile at scale creates dependencies and reduces autonomy, it also yields economies of scale. Systems and infrastructure are already set up, and teams don’t have to spend time and resources on them.

Changing our culture
When scaling up agile for enterprise projects, issues are exaggerated. It takes longer to do everything, including changing culture throughout the organization. We had to take a step back and figure out how to bridge the communication gap between leadership, stakeholders, and agile teams. We’re working to remove roadblocks at the human level—to increase buy-in and get the vision across at all levels. Clear communication, mitigating dependencies, and resolving blocks is an evolving and recurring process. We treat this process in the backlog in the same way as the project.

We’ve found that it’s important to properly set expectations. When we first started working with agile at scale, projects were all labeled priority one. We learned that we need to bring the right people into discussions up front so
that everyone understands what will be done and when. We communicate the priorities for what we deliver and talk about the order of delivery.

Another area we changed is how we define what will be delivered. In the traditional waterfall model, stakeholders and customers had a clear idea of what would be delivered far into the future at a deep level of detail—like color, size, and capabilities. In the agile model, we commit to a product line and date, but to allow for course correction as needed, we’re vague about exactly what the end product will be. For example, we might say that we’ll be producing the world’s leading virtual reality goggles by the end of 2017, but nothing more specific.

Leadership’s role in scaled agile is also different and requires a change in mindset and a learning process. Leaders used to get together annually to create roadmaps and projects, and hand them off to the engineering teams who responded with promised delivery dates. Leadership would then drive the teams to meet the stated goals. With scaled agile, leadership is learning that the teams have a certain work capacity, and that the leaders must feed the appropriate amount of work to those teams. Leaders also need to understand in advance how much the organization needs to grow and shrink according to the workload. They’re learning that results are their responsibility. As the leaders in our sales and marketing team become more involved in planning epics and scenarios, they’re taking greater ownership of the development process. By going through the development lifecycle, they’re actually sprinting at a larger scale. Through this experience, they’re learning to feed the backlog just like a scrum master.

Decoupling sprints from releases

In addition to software bits, business value may be a plan, decisions about the design, frameworks for test environments, epics, and scenarios. To deliver business value, it isn’t necessary to push it to production. Thus, in the scaled agile model, releases occur outside of the sprint.

This issue spans both process and engineering best practices. It requires a change in our mindset. Typically, releases under the waterfall model have a target date for delivery of a project. We find that large teams moving from waterfall to agile typically try to shoehorn the waterfall process into a sprint, including the usual waterfall release milestones—alpha, beta, pre-release, release to production, and cleanup. They think that delivering business value requires a release. If a team needs six months to release a service, they make their first sprint six months long. When such a team is told to get to two-week sprints, they often try to force the waterfall method into this shortened sprint cycle, and it doesn’t work.

Instead, when using modern engineering best practices for a release, we move to a flighted system. We start with planning and architecture, build a feature, and then flight the code. We put telemetry in place to keep track of what’s happening in the system, and automate testing. A developer should be able to easily publish code—and if it’s bad, it gets kicked back. The ultimate goal is for the team to push a button and release bits into production.

As teams mature in the scaled agile model, they’re learning to keep the release process separate from the sprint. What occurs in a sprint doesn’t necessarily culminate in a release, but rather represents defined business value. As teams adopt shorter and shorter sprints, they’re realizing that they need to continue releasing in longer cycles, even if it still takes six months. Releases can get shorter over time, but they don’t ever need to match up with sprints. By decoupling the sprint concept from the release concept, we can mature the agile system and mature the release cycle without burdening one with the other. Releases should occur in the background.

Lessons learned

When scaling agile, we learned that it’s important to prioritize the following areas:

1. **Minimize dependencies.** This prevents the bottlenecks that happen when several teams rely on one team for a deliverable. No one team owns any part of the overall project. If a deliverable gets on the critical path and starts slowing down the project, any other team can pick up work to get things moving again.

2. **Deliver value.** In the classic agile model involving small teams that deliver services, services are released into production as value. In the scaled up agile model, value is assessed differently because the work product isn’t
necessarily pushed into production immediately. It’s integrated with the work products of other teams. The focus is still on delivering value.

3. **Mature and align a team.** Initially, each of our agile teams operated with relative autonomy and adopted its own cadence, tools, and processes. This created issues when we tried to coordinate the work of multiple teams to deliver a large-scale product or suite of products. Today, teams are much more aligned, and as time goes on, we’ll continue to focus on maturing our teams and approaches to create a better and better system.

**How we’re doing**

As we continue this journey to modern engineering, we’re delivering higher business value than we delivered under the waterfall model. Customers are happier with the results. Areas where we’ve improved using our model for scaling agile include:

- Delivering better quality code with lower service volatility.
- Setting and meeting expectations.
- Team members becoming actively engaged in the process.
- Creating more accurate plans that enable accurate budgeting
- Delivering close to 100 percent of the planned increment features.

**What’s next**

Under scaled agile, we find that the teams where leadership best understands and participates in the process experience the greatest benefits. Our small teams are working in the agile environment every day, becoming proficient through daily experience. Team leaders at the program and portfolio levels, though, are removed from the day-to-day agile processes. Moreover, it takes longer to go through a complete cycle at scale. Rather than two-week sprints, a full cycle at scale can take months. Thus, it takes longer for leadership to gain a full understanding based on experience of how agile works at scale. And even after our leaders experience a complete cycle, we expect to continue scaling the agile model to handle ever larger projects, and they’ll need to learn how to manage agile at these higher levels.

**For more information**

**Microsoft IT**

microsoft.com/ITShowcase

Transforming IT for modern engineering

Moving to modern engineering

Adopting modern engineering processes and tools at Microsoft IT

Creating a culture of modern engineering within Microsoft IT

© 2017 Microsoft Corporation. This document is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS OR IMPLIED, IN THIS SUMMARY. The names of actual companies and products mentioned herein may be the trademarks of their respective owners.