Moving TFS to a Cloud Cadence

Buck Hodges, Valentina Keremidarska

August 20, 2012 (v. 1.3)

Introduction

Until the TFS 2012 product cycle, the TFS team had followed the divisional approach of milestones and shipping only a boxed product. Two things changed with TFS 2012. First, we made a decision at the beginning of the cycle to adopt Scrum. Second, we deployed tfspreview.com in April, 2011 and began the transformation to an online service team that also ships an on-premises server product that must be easy to setup and administer. In addition to the server, we also deliver rich client experiences in Visual Studio with Team Explorer and cross platform in Eclipse with Team Explorer Everywhere that are key to the success of both TFS and the online service. Here we describe our experience making these changes and what we have learned in the process. We now successfully ship the tfspreview.com service every three weeks with the aspiration to eventually get to every week. Additionally, we plan to ship the on-premises products, including both client and server, quarterly to deliver a constant stream of customer value.

Executive Summary

The adoption of Scrum for TFS 2012 was driven by our desire to deliver experiences incrementally, incorporate customer feedback on completed work before starting new experiences, and to work like our customers in order to build a great experience for teams using Scrum. We used team training, wikis, and a couple of pilot teams to start the adoption process.

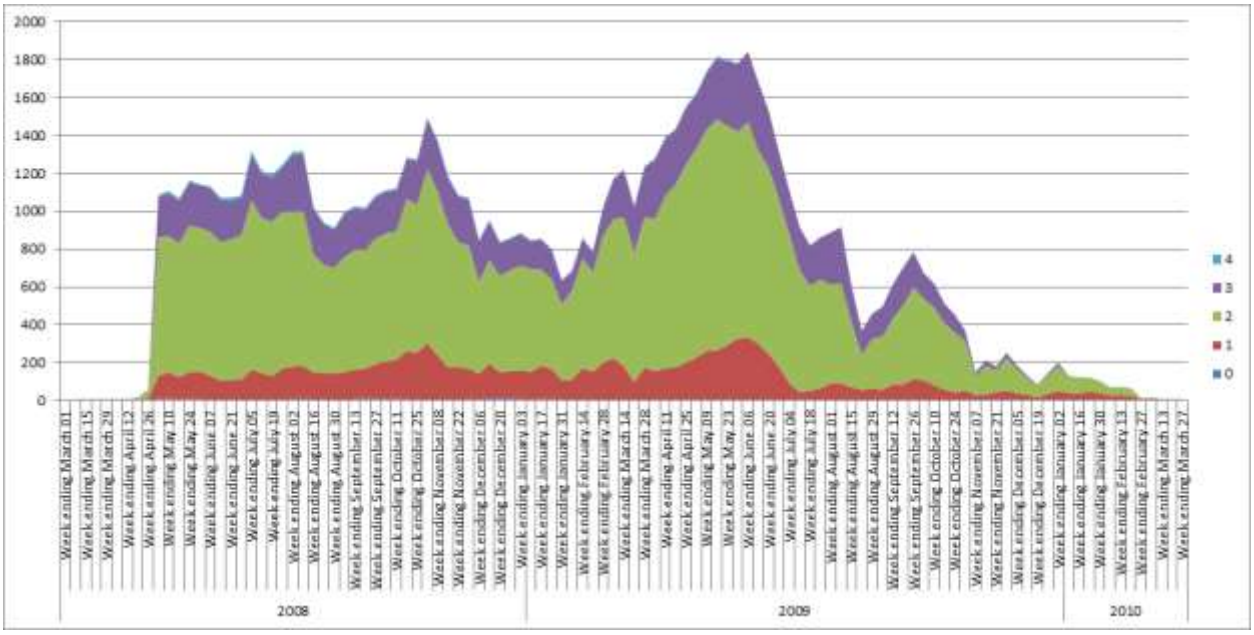
We organized our work in four pillars of cloud, raving fans, agile, and feedback with each having a prioritized backlog of experiences. Teams progress through the backlog in priority order, working on a small number of experiences at any point in time. When starting an experience, teams break down the experience into user stories and meet with leadership for an experience review. Each three-week sprint starts with a kick off email from each team, describing what will be built. At the end of the sprint, each team sends a completion email describing what was completed and produces a demo video of what they built. We hold feature team chats after every other sprint to understand each team's challenges and plans, identify gaps, and ensure a time for an interactive discussion. On a larger scale, we do ALM pillar reviews to ensure end to end cohesive scenarios.

With the first deployment of tfspreview.com in April 2011, we began our journey to cloud cadence. After starting with major and minor releases, we quickly realized that shipping frequently would reduce risk and increase agility. Our high-level planning for the service follows an 18 month road map and a six-month fall/spring release plan in alignment with Azure. To control disclosure, we use feature flags to control which customers can access new features.

Our engineering process emphasizes frequent checkins with as much of the team as possible working in one branch and using feature branches for disruptive work. We optimize for high quality checkins with a gated checkin build-only system and a rolling self-test system that includes upgrade tests. During verification week, we deploy the sprint update to a "golden instance" that is similar to production. Finally, we ensure continuous investment in engineering initiatives through engineering backlog.

Previous cycles

In every cycle prior to 2012 we had some ship stopper issues at the end of each release, from the merge rewrite in the 2005 cycle to the reporting/warehouse rework in order to ship TFS 2010. We had performance issues and scenarios with too many holes at the end of each cycle. Looking at the TFS 2010 product cycle, we built up a large amount of debt, as shown below. Additional, we built up debt in our test automation with many broken tests and low pass rates. It was clear that our approach was broken.



Adopting Scrum

We knew that there were other teams that were being successful with Scrum, such as Team Test during TFS 2010. There were also teams successfully dogfooding our early tools provided in Excel. Additionally, Scrum was rapidly taking over as the dominant process across the industry. We knew we wanted to be more like our customers. Finally, we wanted to be able to deliver features incrementally and get feedback on them, including a new set of features to produce a great Scrum experience for customers.

It wasn't clear from the outset that we could do this successfully with a team of 135 people. Scrum is designed for small teams. Fortunately, we were already in a mode of working in cross-discipline feature teams that are approximately the size appropriate for Scrum (about 12 people, including developers, testers, and PMs).

Educating the Team and Getting Started

We began the effort by interviewing other teams in the company and learning what had worked and not worked for them. We learned several things from that exercise.

- Training is really important. The team as a whole has to understand the concepts and the terminology and internalize the process.
- Adopting Scrum is hard in subtle ways. While it looks easy, it was clear from other teams' experiences that it would take multiple sprints to actually be as productive with Scrum as we had been before becoming more productive.
- Expect teams to want longer sprints and struggle to break work into incremental chunks.
- Scrum of scrums worked for some but not others. How would we keep ten feature teams together to produce a coherent product?

We scheduled day-long Scrum training events in both Redmond and North Carolina in June, 2010. The training comprised a lecture followed by a simulation of Scrum in the afternoon. The simulation opened our eyes to the challenges that lay ahead of us, particularly in breaking down working into increments that would fit within a sprint.

Next we created a sprint playbook on a SharePoint wiki. Creating that playbook forced us to capture how we plan, develop, and test in this new process.

In order to learn before having the whole team execute using Scrum, we had a couple of feature teams start TFS 2012 early using Scrum. This proved valuable in helping the overall TFS team understand challenges around standups (e.g., keeping them on topic and short), committing to work, internalizing story points, and tooling (many used sticky notes). This experience also validated what we had learned from other teams: that we should expect for it to take time for teams to be really productive with Scrum.

Using Backlogs and User Stories

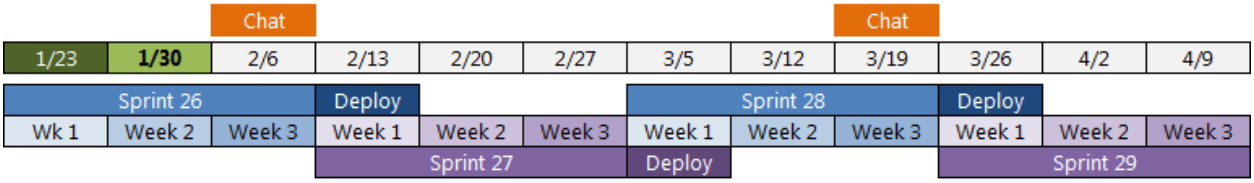
In parallel with our effort to adopt Scrum prior to the start of TFS 2012, the team created a set of pillars for the release: cloud, raving fans, agile, and feedback. For each pillar the teams involved developed a backlog and presented that backlog to leadership. That backlog captured the experiences for TFS 2012. The experiences were stacked in priority order. The feature teams then broke down the experiences into user stories when they began work on each experience. User stories are created when working on an experience and not all up front, allowing the team to iterate.

Feature teams committed to a set of user stories at the beginning of each sprint. When beginning to adopt Scrum, teams have no idea what they can get done, as they are still developing a common notion of story points and have no historical velocity data. It took teams four or five sprints, to get to a point of being able to do this with reasonable accuracy.

Using Scrum at Scale

We use a number of tools to manage the overall process with each of the feature area teams following Scrum including experience reviews, sprint kickoff and completion emails with demo videos, and feature

team chats. The following diagram shows our three-week cadence along with the deployment of tfspreview.com (more on that later).



Experience Reviews

In order to understand what we are building, we hold experience reviews. Experience reviews are designed to allow the leadership team to provide feedback and guidance when a feature team is ready to begin working on an experience. An experience is generally large enough in scope that it will take multiple sprints to complete and involve multiple feature area teams. Leads from the teams involved attend, and the discussion is primarily driven by use of storyboards to describe the user experience and the end-to-end context for it. Additionally, we discuss the main challenges and get an understanding of the scope of investment which we can use further evaluate our priorities. This has worked well as it allows for a frank discussion on not only what the priorities were and the completeness of the experience but also to spot gaps between feature teams.

Sprint Emails: Kickoff, Completion, and Demo Videos

To help understand what we are building, each feature team sends sprint kickoff and completion emails, and the sprint completion emails include demo videos. The sprint kickoff emails describe what will be built, list the user stories for the sprint, and call out anything special (e.g., absences or other work such as QFEs). The sprint completion emails list the user stories completed relative to the sprint kickoff email. This provides accountability on what was started and what was completed. The demo videos demonstrate the features in actual time. The videos aren't polished (in fact, polish is discouraged because it is a waste of time), and they provide the ability to see the user stories in action quickly and easily. They videos are available to everyone and make it easy to provide feedback.

Team Chats: Understanding the Picture Across Teams

Another practice that we started during the 2012 release cycle is holding feature area team chats. Feature team chats are where each feature team presents to the TFS leadership team for 15 – 20 min. on what their challenges are, what is coming up, and anything else that is on their minds. We hold these every other sprint. These help improve the communication, identify gaps, and ensure a time for an interactive discussion on issues that are top of mind for the feature teams.

Feature team chats also help us understand how far we were likely to get down the backlog. We would have teams use very rough costing (e.g., t-shirt size) to estimate where the "cut line" would fall in their backlog for the TFS 2012 release. This gave us the opportunity to make adjustments based on priorities across the team in order to ensure the highest priority work was done for the release.

Feature team chats are the closest thing we have to a scrum of scrums.

ALM Pillar Reviews

To look beyond TFS and across ALM, we held ALM pillar reviews every other sprint to assess the state of the product. During the 2012 cycle these were meetings of all three teams in ALM showing and discussing how we were doing in building the scenarios the teams had created at the beginning of TFS 2012. As part of that assessment, we produced OGFs (assessments of how well a scenario worked) for each scenario after each sprint. Members of each team working on the scenario tested it. The OGF walkthroughs were based on customer intent and not prescriptive. We started this early in the cycle to measure the quality and identify gaps in the scenarios.

As we moved into the latter part of the release cycle, we required that the demos be done using a build out of the ALM branch. We wanted to make sure we were able to see integrated scenarios come together and work in the ALM branch rather than being pieced together using bits across feature branches. Having all ALM teams adopt the same sprint cadence helped the process, as it compelled teams to produce working software for broad scenarios.

Managing Dependencies

One of the big challenges was representing cross-team dependencies. In pure Scrum, team members pick up the next work item from the list of user stories. In TFS, we do not do that. We have multiple technologies that take time to master, including SQL, VS, and web. As a result, not every engineer can do every task. To handle that, we had to coordinate across teams so that producing teams delivered to consuming teams at the appropriate time. We had to rely on traditional coordination here since we weren't strictly following Scrum, and managing dependencies is always hard.

Minimizing Experiences in Progress

At the beginning we established guidelines that teams should minimize the number of experiences in development at one time. The reason for this is that one of the Scrum principles was to ensure that we finished the experiences that we started rather than having a large set of incomplete experiences and running out of time to finish them. While this largely worked well, there were cases where we did not do this. For example, we completely overhauled the TFS web experience in TFS 2012. Since it was a rewrite and there were other teams depending on the framework, there were quite a few experiences in progress at any one time. Given our experience with Scrum now, we believe we could structure the work better (it was also underfunded for the scope of the work).

Teams initially struggled to adapt to breaking work into thee-week sprints. Almost every team wanted to move to longer sprints during the first half of the release cycle. The feedback was so strong, that we began to look into stretching our sprints to four weeks and had conversations with the other ALM teams about it, as that would be a different cadence than they were using (everyone was on three-week sprints). More time passed while we tried to work that out (it wasn't going to be easy), and just before starting the second half we talked to each team in retrospectives about whether to change the sprint length, and something interesting happened. Teams reported that they wanted to stay with three week sprints. From August to February, the teams' views of the length of a sprint had completely changed.

Importance of Code Flow

From the beginning we knew that we wanted teams to integrate often to make sure that experiences were coming together and to minimize integration debt that had been so problematic in the past. To achieve that, we required teams to merge up at the end of each sprint and to merge down at the beginning of each sprint. We wanted to make sure that we could move code around with just enough validation to ensure the teams were productive.

Finding Issues Quickly: Rolling Tests

Our gate to RI from a feature branch was being at self-test. To facilitate that, we created a rolling self-test build system between the TFS 2010 and 2012 product cycles and used it in both the ALM branch and in the feature branches. The result was that teams would find self-test breaks more quickly and be able to fix them as they happened rather than only after the nightly build and test run. Because of the effort required for perf, scale, and stress runs, we executed those in the ALM branch after the end of each sprint. This balance worked well and allowed us to find the issues generally no more than a sprint later than when they were introduced.

Walking in the Customer's Shoes

We wanted to make sure we took time to do customer-focused testing. The focus for the team is often on user stories, which by themselves fit into an end to end scenario. The QA teams planned work to do end to end walkthroughs to make sure the customer experience was coming together. The testing was based on customer intent, not "testing to the spec."

Some teams also had regular team walk-throughs, in which the QA lead, dev lead, and PM walked through specific scenarios, and invited the team to participate. If multiple teams were involved, all the teams participated. Like OGF scenario testing, this was effective at uncovering seams in the product and ensuring we delivered the right experience for our customers.

Benefits

There were a lot of benefits to adopting Scrum for TFS 2012.

- More focused on customer value by focusing on user stories: As a customer, I can….
- Splitting work into three week deliverables greatly aided our engineering system, as we required teams to merge up at the end of each sprint and merge down at the beginning of each sprint. This resulted in faster integration across feature teams than before. No more big surprises after three months of isolated development in a feature branch.
- Sprints created better alignment between dev and test. With the goal of producing working software every three weeks, the gap between dev and test narrowed significantly. At the beginning many user stories weren't being finished in a sprint with test work hanging over, and by the second half of the cycle that improved .
- We no longer created block schedules. In the past we would have the teams create a block schedule for the feature work for a milestone (or longer), and it was always wrong. Without deeply thinking through the experience, design, and tests, teams could not accurately estimate

the amount work, and they underestimated. The result was demoralizing for the team and encouraged bad behavior (e.g., cut the tests to move on to the next thing in the schedule).

- Each sprint should result in a demo video showing the user stories that were completed in the sprint. Unfinished user stories (e.g., testing not complete) cannot be included. The principle is that we value and celebrate completed user stories.
- Experience reviews, demo videos, feature team chats, and ALM-wide reviews helped us make sure we were building the right software to enable the scenarios we set out for TFS 2012.

Challenges

We still had a number of challenges with our process.

- We carried too much bug debt. Even with bug caps in place and acknowledgement from the teams that we should remain under the cap, the bug totals floated up to the cap. Then during the last sprints of the development phase, the total went through the cap. The problem was that there was a stabilization period coming up, and it was easy for teams to defer bugs until stabilization in favor of getting features in. We altered the bug cap for the second half of the cycle, and the result was less high-priority bug debt, but it did not get us to a consistent state of low bug debt.
- Dependency management remained challenging. Without being able to practice pure Scrum where any engineer can pick up the next task, we needed to manage dependencies. To manage that we work to ensure good communication between producing teams and consuming teams.
- Getting dev designs reviewed ahead of implementation is challenging. We have gotten much better at this over the last two years, but it is very easy for the change to becoming agile to become an excuse not to get designs reviewed. Our goal with the reviews is to provide a forum for feedback on the designs, to ensure that we are thinking about the design requirements for the experience, and education across the teams about what other teams are building. We began to proactively push folks to have the discussions earlier, and we have created a forum for devs to have design discussions in a cross-site meeting.
- Testing still got squeezed. Many teams would finish testing of user stories in the next sprint after the dev work. While a big improvement over the dev/test gap in TFS 2010, this was an issue throughout the cycle.

Swimming Upstream: Scrum and the Box Schedule

Given that we had adopted Scrum, why were we seeing high bug totals and other debt? The reason is that we were swimming upstream. We were asking the teams to work and deliver software at the end of each sprint, yet there was nothing shipping. The divisional schedule had clear phases for developing features during development milestones and stabilization periods after each milestone. Further, the stabilization period after the last development milestone was quite long. The result was that we struggled to convince teams to complete the work as they went when they were certain they could add features during development milestones but uncertain they would be able to add features after the end of the last milestone. We emphasized that having completed work as we went and having no debt would result in teams being allowed to build more features, but it was a hard sell. In the past it had

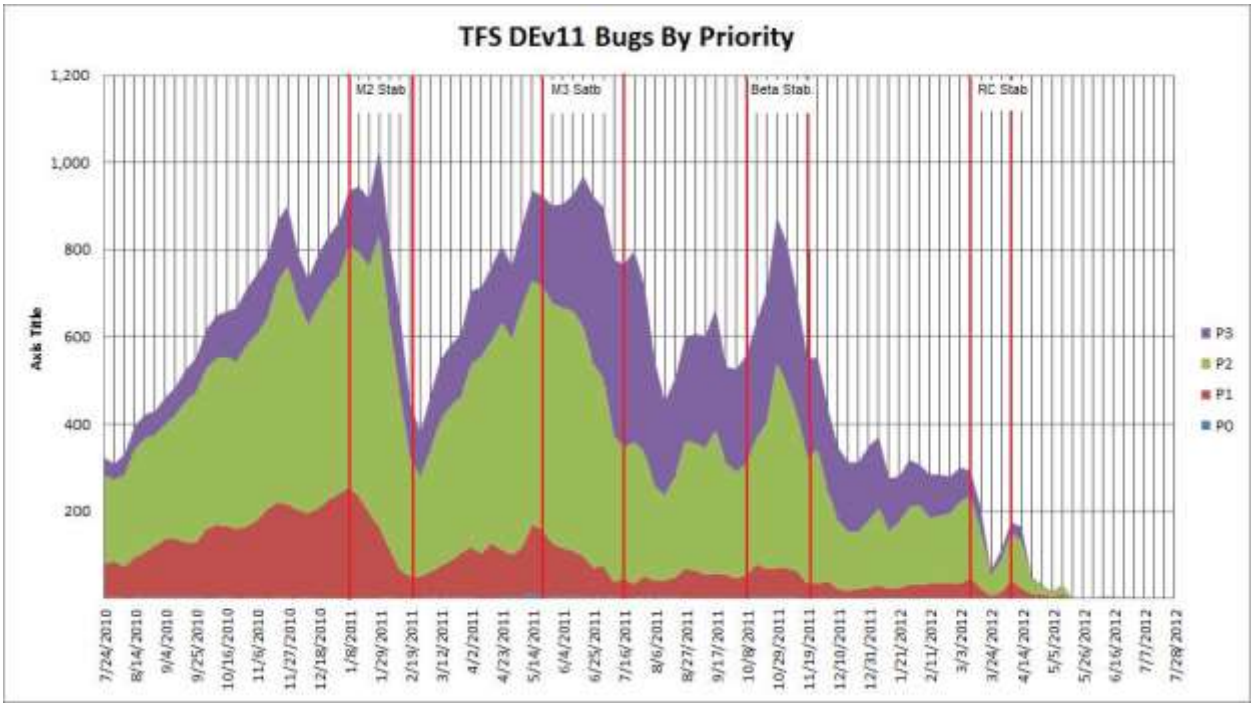
always required approval, and the teams that did have bandwidth were asked to go help the teams that built up debt, which is no reward for doing the right thing.

The box product schedule with feature development milestones and stabilization on a two to three year ship cycle encourages and rewards dysfunctional behaviors.

- Jam features into the product during development milestones because teams do not have to ask permission to finish what has been started but do have to ask for permission to add features during stabilization.
- Jam features into the product during development milestones or the feature may not get to customers for another 2-3 years if it has to wait for the next release.
- Long stabilization periods encourage accumulating debt. The message to the teams is that there is time to fix bugs, and since that time isn't for developing new features, push the bugs out to stabilization.

To help counteract this, we used bug caps for the teams and required certain tenets to be met in the ALM branch. However, the effectiveness of this was limited by the fact that teams were not bought into it.

Even with this, TFS 2012 was a much better release as measured by the bug debt graph, as shown below. Note that the peaks, while higher than they should have been, are significantly lower and that the area under the curve is significantly less. The lengths of the two cycles, TFS 2010 and TFS 2012, are largely the same, and we began delivering tfspreview.com half way through TFS 2012 (April 2011).



Notice that after the beta we had better success with teams meeting a lower bug cap, and that resulted in several teams being able to do new feature work in TFS 2012 while the division was in stabilization.

This showed that the team was starting to internalize that we were serious about continuing to do feature development if the bug debt remained low. However, it is not possible to separate it from the fact that we were also already operating on a three-week cloud deployment cadence at the same time, so there was more than one factor involved here.

Moving to a three-week ship cadence

When we deployed tfspreview.com in April, 2011, we began thinking hard about how to update it. How often should we change it? How should we handle "releases"? What would our customers accept? What could our engineering system support?

Major and Minor Updates: Box Product Thinking in a Cloud World

The initial thinking was that we would have major and minor releases. Every six months we would have a major release with significant features and every three months we would update the service with bug fixes and minor features. We spent significant time thinking about how to manage the service following this pattern.

Our early updates of the service roughly followed this pattern, except they were essentially always major updates. The first update we did was in the summer, containing four months of changes across the entire product. We had a number of issues after deployment that we had to fix and had we not done it nearly two months before the BUILD conference, we may not have been ready to go public as we had to make another significant update to address serious scale issues.

Then in December, 2011 we did another major update. This had about five months of changes in it. This update was our worst. It took a week to get through the upgrade, which was fraught with performance problems and excessive customer down time.

Even in the run up to the December deployment, we knew the likelihood of issues was high. We had to scramble to get everything tested and signed off. Because of the immense amount of churn, everything had to be tested. We knew we needed to find a better way.

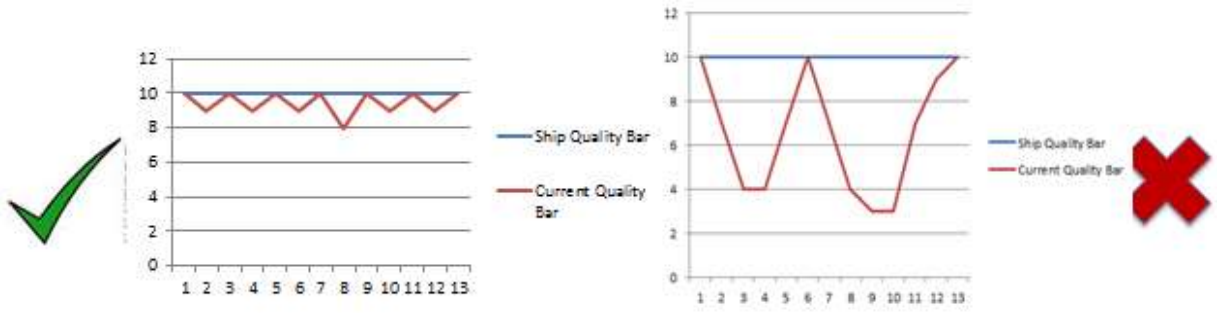
Going Incremental: Shipping Frequently

It became more apparent this was not the right way to run a service. For one, our competitors ship far more frequently (multiple times a day some cases), and major online services like Facebook do weekly deployments. Our service would fall behind while our competitors moved ahead. At the same time, we are different in that we have an on-premises product that is very important to our success. We found inspiration in how Google Chrome moved to shipping frequently as well.

Another problem with major updates is simply the churn. If we deploy major changes to every area in TFS at once, the likelihood of issues is significantly higher than if we deploy a small number changes for most of TFS and a few major changes to a subset. To put it another way, if one million lines of code are changed from one deployment to the next and something breaks, what is the likelihood of finding a problem quickly? If instead we deploy an update where only one line of code changes and something breaks, what's the likelihood? While the problem may not be in the changes that were deployed, we

can quickly determine whether the problem is due to the change in the one line case far faster than ruling out one million lines of change. As a result, there is far more risk and unknown in a deploying a large number of changes compared to a small number of changes.

The fundamental principle is risk is proportional to the ship cycle, assuming constant churn over time. Shipping more frequently is less risky because each update carries fewer changes. Since we want to build a great service that rapidly evolves to meet our customers' needs, this means that we want to ship as quickly as we are able. We embraced one of the main lessons from Bing – start with Done and stay Done, adding capabilities in short cycles while always being at ship quality.



Making it Possible: Engineering System Changes

To ship more frequently and be successful, we need to have the right process in place and an engineering system to support it. Our adoption of Scrum at the beginning of TFS 2012 made this a much more seamless transition because the underlying principle of Scrum is to produce shippable software at the end of every sprint. Further, by shipping we would provide incentive and reward to the teams to follow this core principle.

Rolling Tests, Better Dev Environment

The engineering system needed to support a faster shipping cycle, and we had made key changes prior to the 2012 cycle that provided the foundation for this as well. Our rolling self-test system ensured that we could quickly find self-test issues by having the surfaced every couple of hours rather than once a day. We made key investments in improving our rolling tests to ensure they covered not only the functional aspects of key scenarios in the product but also deployment and upgrade. Further, we invested in making this system run in both on-premises and Azure configurations. Additionally, we had already made investments in improving the dev and test environments.

More Effective Automation

In addition to rolling tests to keep quality high throughout the day, we needed to make a number of changes to our overall test automation in order to be more effective.

- Prior to the 2012 cycle, developers and testers could not run each other's tests. We fixed that by putting in place a common bootstrap framework that the entire engineering team uses. Automated tests "just work" on dev and QA boxes, and can be configured to target more complicated test environments.

- We reduced our existing test automation substantially, especially in UI automation. We eliminated duplication between development and QA, UI and API tests, and cut tests that covered areas that were unlikely to regress.

- We reuse tests for different purposes where possible, leveraging functional tests for performance and security.

- We eliminated "initial pass rate" and "final pass rate" as metrics for tests passes. That had resulted in a culture of accepting unreliable tests.

Upgrade Always Works

In preparation for the 2012 cycle, we committed to having upgrade working from day 1 of the product cycle. In prior releases, upgrade was not working until close to beta. That resulted in major scrambles to fix it. It is critical that upgrade not corrupt data, so we had to get this right.

Having upgrade working all of the time later became crucial to our success in shipping every three weeks. We made upgrade a part of the rolling self-test system to quickly catch regressions, and we populated it with meaningful data designed to catch errors in the upgrade process. We also run a larger set of upgrade tests at the end of each phase of the upgrade process.

Making the Switch to Deploying Every Three Weeks

In the summer we set a goal of shipping every week. That fall we decided that we would update every month going forward, as we were not able to turn the crank quickly enough to ship weekly. In January we realized that a monthly shipping cadence did not align well with the sprints and this misalignment would lead to a separate stabilization. Thus, we decided that shipping every three weeks, aligned with the sprints, was the best model for us.

Avoiding Stabilization: Verification Week

We knew that we didn't want any stabilization period to encourage dysfunctional behavior. As a result we decided that going forward we would have one week of verification between the end of a sprint and deployment. Teams must complete their testing and fix bugs before the end of the sprint. To emphasize this, verification week overlaps the first week of the following sprint – it is not a week between sprints. There should few bug fixes during verification week, and the teams should all be focused on the next sprint.

During verification week we deploy the sprint update to a "golden instance" that began with the original build deployed in April, 2011. The goal is to be as similar as possible to production in a test environment, including the upgrade history. At the end of verification week, the service delivery team deploys the sprint update on a pre-production environment to validate the deployment process.

With verification week completed, we deploy the update into production on the following Monday morning. We chose Monday mornings in order to maximize the amount time during the work week to deal with any issues that may arise. Following the deployment into production, we verify features in production. This practice of test after deployment is recent, and we still need to improve further.

Learning from Each Deployment

After each deployment, we held retrospectives and worked to learn from what went well and even more so from our mistakes. We began a healthy pattern of learning from every deployment that has resulted in constant improvement and mistakes not being repeated.

Since then, we have only missed one deployment, which we acknowledged as a failure and learned from. We aspire to get to deploying every week, but it's clear that's going to require another significant investment in improvements to the engineering system, both in the development environment and our test infrastructure.

Planning

While the engineering process and deployment work on a three-week sprint cadence, planning has different requirements in order to deliver customer value. We want to deliver a coherent set of great experiences that align with our business strategy.

Setting Direction: Storyboard of the Vision

Our planning has several different levels. First, we went through a high-level planning exercise that resulted in a set of storyboards showing the concepts and scenarios that we want to deliver over the next 18 months. This provides the high level roadmap of where we are going and is part of the VSOnline vision – more than just tfspreview.com.

Six-Month Directional Plans

Our more detailed planning begins with a six month cadence and aligns with Azure with spring and fall releases. We use this six-month directional planning to keep the team aligned on the strategy and goals, while the teams execute in sprints working down the backlog in priority order. We begin with settling on a set of themes for the coming six months, framing the customer value. The feature teams then begin determining experiences for those themes. Finally, we are able to prioritize the experiences so that we make sure we focus on the experiences that deliver the most value.

The teams then begin designing the experiences and producing storyboards as they work through the backlog. As the experience designs come together, we hold experience reviews, as we did during TFS 2012, to assess the experiences and provide feedback. Again, this is key to delivering great customer value with a coherent set of scenarios that integrate well.

Sprint planning remains the same as it was during TFS 2012 with sprint kickoff emails, sprint completion emails, and end-of-sprint demo videos. Every other sprint we hold feature team chats, also another practice that we started in TFS 2012.

Managing Disclosure: Feature Flags

As part of planning, we have to determine disclosure. The experiences that we plan and build will be delivered to the service incrementally. Very early deployments of features are incomplete. Marketing may want to delay disclosure until a particular event. As a result, we need a way to control who has access to an experience. To do that, we use feature flags. Feature flags control whether the experience is accessible, and they can be controlled both globally and on a per-account basis.

Feature flags also allow us to deploy features incrementally in preparation for a disclosure event. With the goal to ship incrementally as features are implemented, we do not want to hold on to the work and deploy it all at once in a major deployment. This provides us the flexibility to ship incrementally while still be able to have disclosure events.

Collectively, these form the core of our planning process.

Engineering

Once we decided to ship every three weeks, we needed to create a framework for how to do it. Here is our approach at the end of the 2012 cycle.

- The same code base is used for both the online service and the on-premises product. The differences between the Azure and on-premises product are handled by environmental checks in the code.
- Everyone in TFS works in one branch unless there is a need to make destabilizing change. This minimizes integration debt, which leads to a loss of predictability.
- We optimize for quality checkins. We maintain a rolling test system that quickly alerts us to issues. The gated checkin system only builds in order to maximize throughput.
- If a feature branch is required for a destabilizing change, such as was the case with the multi-tenancy work to dramatically shrink our COGS, the feature branch will merge up at the beginning of a sprint to allow for stabilization during the sprint before deployment.
- Each sprint ends on a Friday and on the following Monday the 2012 update branch is merged into the service deployment branch. The 2012 update branch must be shippable at the end of every sprint.

Scale Testing

Our online service scales in ways our on-premises product does not. Whereas the on-premises product scales to millions of files in version control and work items in work item tracking, the online service scales to many accounts with much smaller data sets each. Each order of magnitude increase in scale of the service has uncovered new set of scale issues. To get ahead of these issues in production, we have set up large test environments with targeted testing based on expected upcoming growth and usage patterns.

Performance Testing

Going into TFS 2012 we knew we wanted a better performance testing suite, and shipping the online service made this crucial. We invested in creating a server performance testing suite that is reliable, easier to maintain due to leveraging existing automation, and runs more quickly. We run this in addition to our client-based performance suite.

Verification Week

There is one week of verification. We chose this term very deliberately. We are verifying that the changes from the prior sprint are ready for deployment. We are not stabilizing! As always, the goal is to have a process that serves our needs as efficiently as possible. This process relies on the fact that tests have run continuously during the sprint.

- We established a cloud ship room email list that is used for approval for any fixes for issues discovered during verification. This keeps everyone in sync on the changes.
- The gated checkin for the service deployment branch for the last verification week fix is the source for producing the Azure deployment package, which is automated via a deployment script. There is a rolling self-test that also consumes the results of the gated build and runs verification in Azure.
- QA signs off on an upgrade of the "golden instance." This is a deployment of the service in Azure that is not production but has the same lineage as production. It started with the same build that was initially pushed to production in April, 2011. The goal is to have an environment that is as close as possible to production with the same settings and quirks. We do not attempt to simulate scale or load with this environment.
- The service delivery team (SD) deploys to a pre-production environment. SD uses this to validate the deployment process, which they will use in production. SD deploys to PPE the day before or the morning of production deployment.

Don't Look Back: Roll Data Forward

We had to decide what to do if an update to the service either fails or introduces problems post deployment. Online services must either roll back or roll forward. Rolling back means supporting a process for restoring the old files and reversing data transformations to return the service to the state that existed prior the deployment while retaining any customer data that changed while the upgrade was in process. For binaries, static text files such as HTML and Javascript, and simple settings, the process is straightforward. However, for data transformations performed during upgrade there must be a corresponding data transformation in the reverse direction. Now there are two engineering problems to solve and two solutions to test. Additionally the work to implement and test rollback is usually waste, as rollbacks should be rare.

We decided that rolling forward is the right approach for our service. What this means is that if something goes wrong, we will deploy another upgrade to address the issues. For executable files, that usually results in deploying a new version with a fix in it or, if the problem cannot be fixed quickly enough, an update that actually puts the old files in place. For data transformations it means that we build and deploy new data transformations that correct the issue. As a result, we only have to support upgrading, and we can focus all of our engineering effort on ensuring the upgrade process works well and is tested thoroughly. We also optimize our work for success. In practice we have found this to work well for us.

Improve Engineering as We Go

Because we now operate a service, there is no longer a time between releases in which to have an effort to improve our engineering system. As a result, we have created an Engineering Backlog that captures the investments we need to make in order for dev and test to become ever more productive and to get to our eventual goal of shipping every week. Each sprint, we invest a portion of time to pick off user stories from the engineering backlog and implement them. Some of the results of that effort include scripts to support much smaller workspaces for devs and testers which allow us to use local workspaces,

integrating JSLint into our workflow, switching to QUnit for much faster Javascript unit testing, and improvements in being able to test TFS in Azure at scale.

Shipping is the Reward

Shipping provides accountability and reward. Teams are accountable for making sure their areas are ready to ship every sprint. Not being ready results in additional work that interrupts the current sprint. The reward for teams is being able to get their work to customers quickly. It is very satisfying to build software that is in customers' hands in days after it is completed.

Here is a summary of the key factors in engineering that have enabled us to ship every three weeks.

- Frequent checkins with as many folks as possible working one branch
- Dev and QA working together to deliver user stories every sprint, which drives conversations about test contracts and interfaces earlier
- Gated checkin that only builds and rolling test systems that balance checkin speed with quality in a tight feedback loop
- Feature flags to control access to experiences
- Deployment retrospectives
- Engineering backlog to constantly make improvements to our engineering system
- Dogfood our own product with teams in the cloud
- Engage with the service delivery team early in the feature planning/development process
- Validate features in production after deployment

Constant Feedback

One of the great benefits of delivering the service during TFS 2012 was that we were able to get a constant stream of feedback. The service was always there, being updated regularly. It was always newer than the box product. Customers give us feedback, and we can react to it immediately. Since the customer didn't have to install anything and the service handles all upgrades, we got more feedback during the TFS 2012 cycle than any other. That feedback has come in through many channels – email, blogs, forums, StackOverflow, Twitter, etc.

With feature flags, we are able to provide access to a subset of our users and get feedback from them before making a feature broadly available. This not only allows for feedback from users but also feedback from the service in the form of operating it. We can see how a new feature, like build in the cloud, works in production and ramp up the scale in a controlled fashion.

The result is a new level of feedback not previously available. This allows us to move beyond traditional feedback and into operational and business intelligence.

Operational Intelligence

We needed the ability to see the health of the system and to understand the performance of the experience of our customers. Today we have multiple ways of doing that.

First, we have extensive tracing in the system. As we started the effort to build tfspreview.com, we invested in creating infrastructure to trace and log information about exceptions, retries, SQL calls, etc. We knew we needed to be able to find and diagnose issues without being able to have direct access to a running process under a debugger. Given the large amount of information generated by doing this, we needed a way to control it, and we use trace points to allow us to turn individual trace statements on and off. We have more work to do here, such as tagging exceptions as expected (e.g., user tried to access an artifact without sufficient permissions) or needing investigation (e.g., primary key violation).

Second, we have a monitoring system that collects data from the tracing, event log, and activity log and produces alerts for certain issues, such as builds that take too long or database utilization being too high. For this we leverage SCOM.

Third, we have built a dashboard that shows key metrics. These include the number of accounts, the number of users accessing the system in the last 24 hours, the number of builds, the top 15 commands based on execution time and count, etc. The dashboard allows us to spot global issues and see trends.

Fourth, we use an outside-in testing service to test the performance of the system from different locations in the world. This has also provided us early warning for global issues.

Finally, we hold monthly service reviews. In these reviews we go over the issues that customers have had (e.g., calls to CSS and threads in the forums), look at service metrics, and look for trends that indicate where we may need to invest (e.g., scale issues).

What's next?

We have some challenges going forward. One is getting to a sustainable pace. With a service, there is no down time. There is no breather that different disciplines get at times during the box product cycle. With no breaks, this becomes a never ending push to deliver the next thing.

We will also be investing in tooling to allow us to measure and track the health of each database and each tenant both visually and with alerting. Our goal is to spot problems before our customers do, rather than have our customers tell us about problems. We need the feature teams to consider what alerts to add and what metrics to report on as part of the design of any new capabilities in the service.

One of our most important challenges is to go deeper into the build/measure/learn cycle. We have done some of it to date. We constantly learn from our deployments and improve our ability to ship frequently. We get a constant stream of feedback from customers that we learn from and incorporate into our plans.

We are approaching the quarterly updates for the on-premises clients and server SKUs the same way. We will need to continue to refine our processes and improve our tooling to support it more efficiently. We also believe that we will learn a lot in the first quarterly release, as we did with our early deployment to the cloud. With our current update technology in the client and the server, customers will not be able to absorb updates as fast as in the cloud, and that will make learning from each quarterly update even more important.

What Works

Here is a summary of what has enabled us to ship a quality product every three weeks.

- Adopting Scrum
- Planning the direction for the team on a six month cadence
- Using experience reviews, ALM pillar reviews, sprint emails with demo videos, and team chats to build a cohesive set of experiences
- Keeping debt low with accountability through shipping frequently
- Frequent checkins with as many folks as possible working one branch
- Optimizing for code flow for feature branch integrations
- Dev and QA working together to deliver user stories every sprint, driving conversations about test contracts and interfaces earlier
- Gated checkin that only builds and rolling test systems that balance checkin speed with quality in a tight feedback loop
- Feature flags to control access to experiences
- Upgrade always works and is included in the rolling test system
- Deployment retrospectives
- Engineering backlog to constantly make improvements to our engineering system
- Dogfooding our own product with teams in the cloud
- Engaging with the service delivery team early in the feature planning/development process
- Validating features in production after deployment
- Frequent customer feedback

Conclusion

Over the last two years, we have completely changed how it approaches building and shipping software. We have adopted Scrum, made the transition to a cloud services team, and now ship on a three-week cadence. We have seen shipping frequently drive the right behavior because it rewards doing the right things. The team has embraced it, and we'll never go back to the old way.