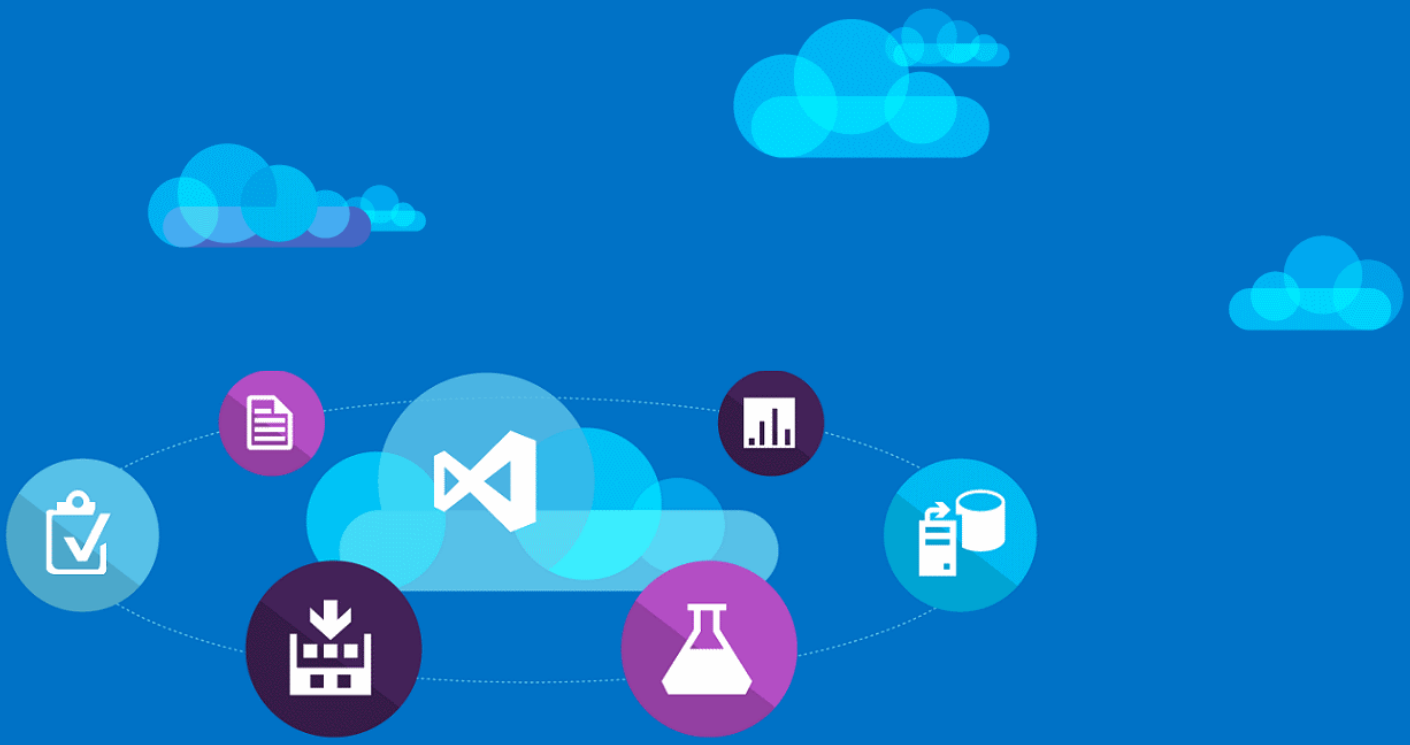

Our journey to Cloud Cadence, lessons learned at Microsoft Developer Division

By Sam Guckenheimer

This book tells a few of the lessons learned as a "box" software company, delivering on-premises software releases on a multi-year cadence, became a SaaS provider as well, with continuous delivery from the public cloud. It covers the DevOps engineering practices and the tools and culture that the organization needed to evolve as it transformed to the second decade of Agile.

From Agile to DevOps at Microsoft Developer Division

On November 13, 2013, we launched Visual Studio 2013 and announced the commercial terms of our hosted service, *Visual Studio Online* (VSO). We promptly experienced a seven-hour outage. At that time, we were running our service in a single scale unit, serving about a million users. It was our peak traffic time, when Europe and both coasts of the US are online. We had many new capabilities hidden behind "feature flags" that we lifted right before the marketing announcement. We discovered that we did not have telemetry on a key piece of our network layer, the IP ports that communicate between services. And we were generating new demand. From a technical standpoint, this was a large embarrassment, as shown in Figure 1. You can follow the gory details here.[1]

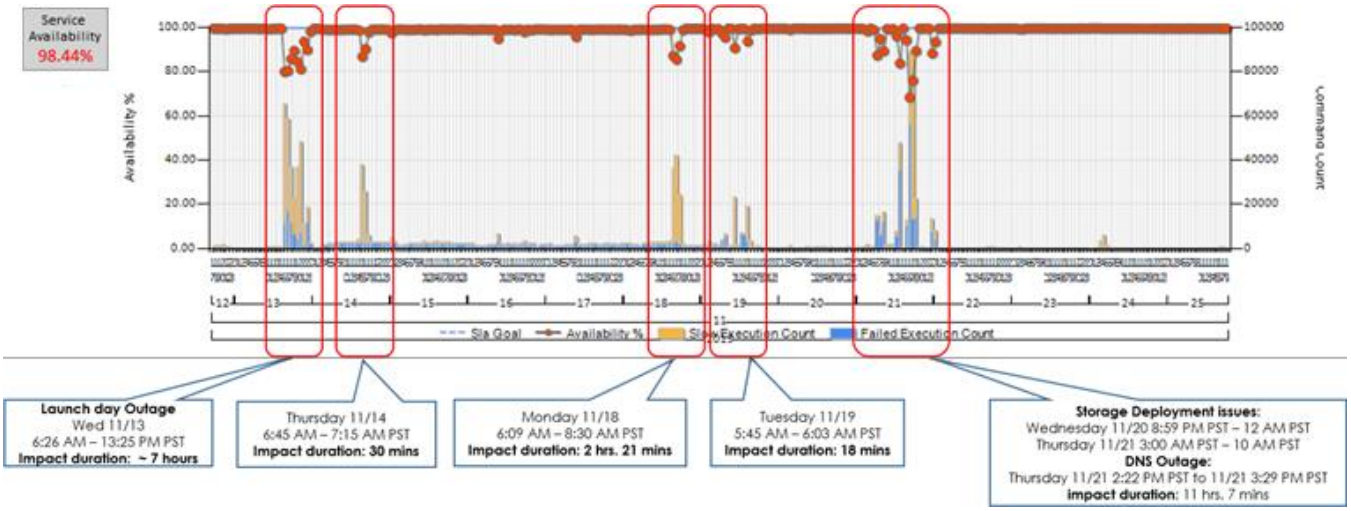

Figure 1. The VS 2013 launch was marred by an embarrassing series of outages with the VS Online service.

From a marketing standpoint, the launch was actually a big success. This was the inflection point at which VSO quickly started climbing to double-digit, month-over-month growth. (The growth is continuing and we more than doubled the million users in the following year).

Balancing feature and live site work

During that time, VSO was hosted on only one scale unit, in the Chicago data center. We knew that we would need to scale out across multiple, individually deployed stamps, but we had always prioritized feature work over the live site work to move to multiple scale units. The launch experience changed those priorities. The team decided to postpone planned feature work and to accelerate the live site work. We needed a way to canary the deployment of the VSO updates. The existing Chicago scale unit remained unchanged, but we added another in front of it in the deployment sequence, which we named SU0.

In the new process, we would first deploy every sprint to San Antonio (Scale Unit 0), where we worked, and use the new release for a few hours. When satisfied, we would allow the deployment to continue to roll to Chicago (Scale Unit 1), as shown in Figure 2. Of course, if there is a problem in SU0, we can remediate and restart. Over time, we added several other Azure data centers in the sequence. In the Fall of 2014, we added Amsterdam as the fifth. More recently, we have added Australia, as we continue to expand internationally. Visual Studio Release Management handles the workflow and automation of rolling out each sprint worldwide.

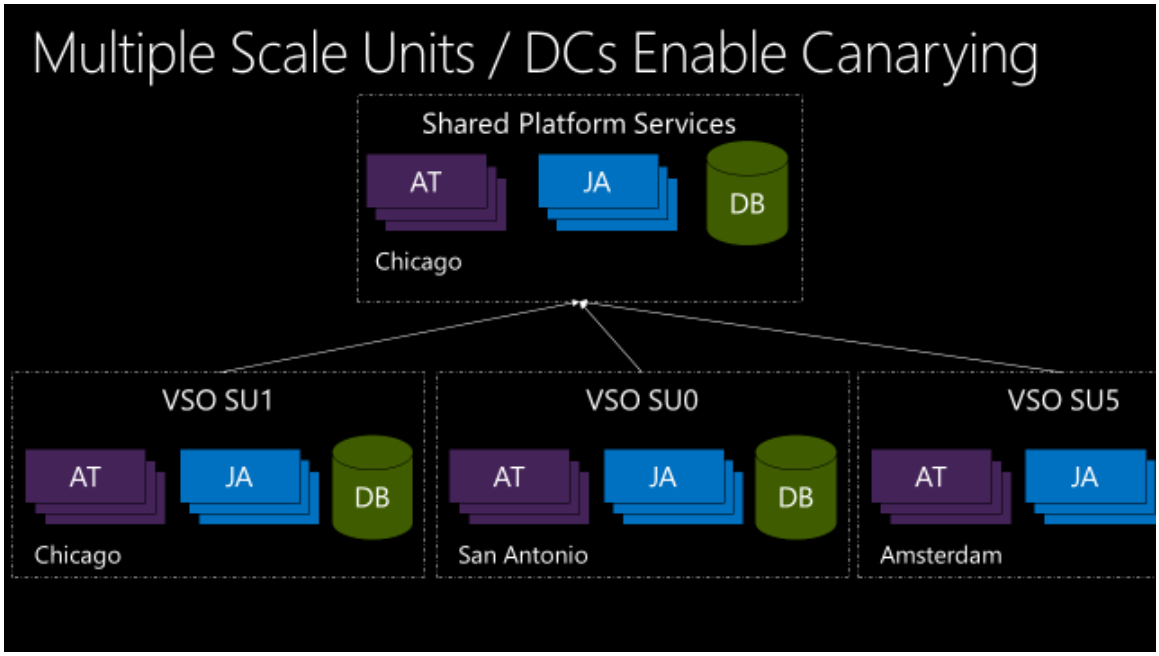

Figure 2. After November 2013, VSO expanded from a single data center to many. This allows both for "canary" deployment and for global reach.

The effect of emphasizing the live site quality of service could be clearly seen in the results. If you look at the Monthly Service Review from April 2014 (in Figure 3), you can see 43 Live Site Incidents (LSIs) in the November 2013 data, descending to a total of seven LSIs six months later, only two of which came from generally available (as opposed to preview) services. We have continued to improve our DevOps practices and, for the most part, our operational health since then, although it has been a far from linear journey.

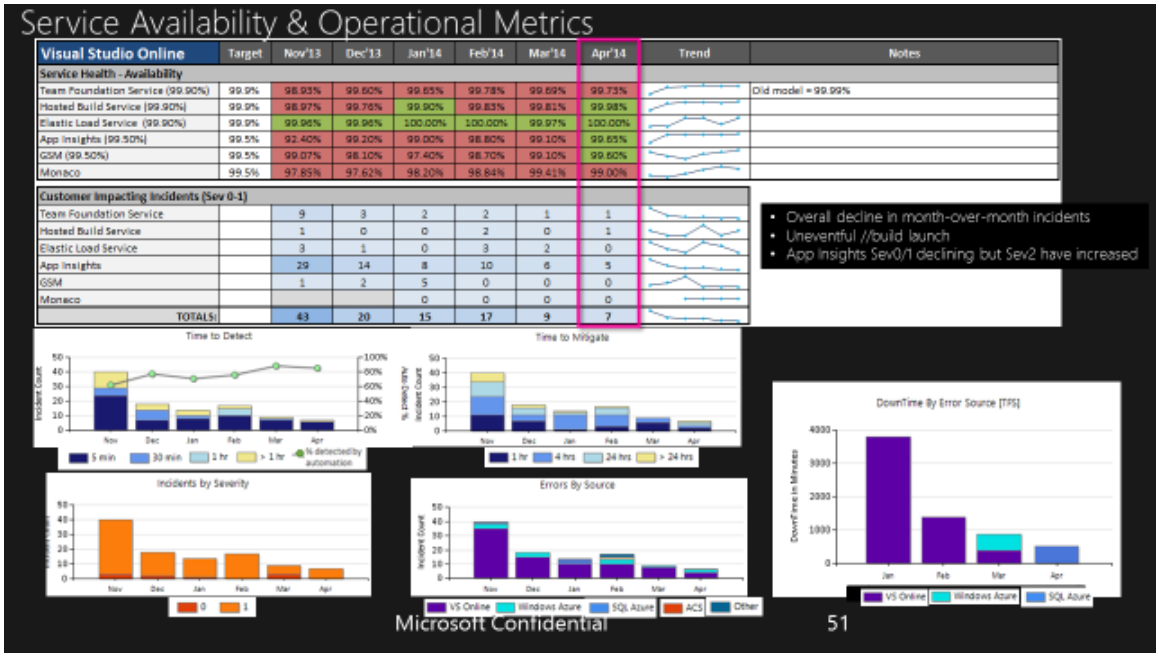

Figure 3. By April 2014, VSO triggered only seven LSIs, as opposed to 43 six months earlier. Of the seven, only two came from services that were in general availability.

How we moved from Agile to DevOps

Over seven years, Microsoft Developer Division (DevDiv) embraced Agile. We had achieved a 15x reduction in technical debt through solid engineering practices, drawn heavily from XP. We trained everyone on Scrum, multidisciplinary teams and product ownership across the division. We significantly focused on the flow of value to our customers. By the time we shipped VS2010, the product line achieved a level of customer recognition that was unparalleled.[2]

After we shipped VS2010, we knew that we needed to begin to work on converting Team Foundation Server into a Software as a Service (SaaS) offering. The SaaS version, now called Visual Studio Online (VSO), would be hosted on Microsoft Azure, and to succeed with that we needed to begin adopting DevOps practices. That meant we needed to expand our practices from Agile to DevOps. What's the difference?

Part of a DevOps culture is learning from usage. A tacit assumption of Agile was that the Product Owner was omniscient and could groom the backlog correctly. In contrast, when you run a high-reliability service, you can observe how customers are actually using its capabilities in near real-time. You can release frequently, experiment with improvements, measure, and ask customers how they perceive the changes. The data you collect becomes the basis for the next set of improvements you do. In this way, a DevOps product backlog is really a set of hypotheses that become experiments in the running software and allow a cycle of continuous feedback.

As shown in Figure 4, DevOps grew from Agile based on four trends:

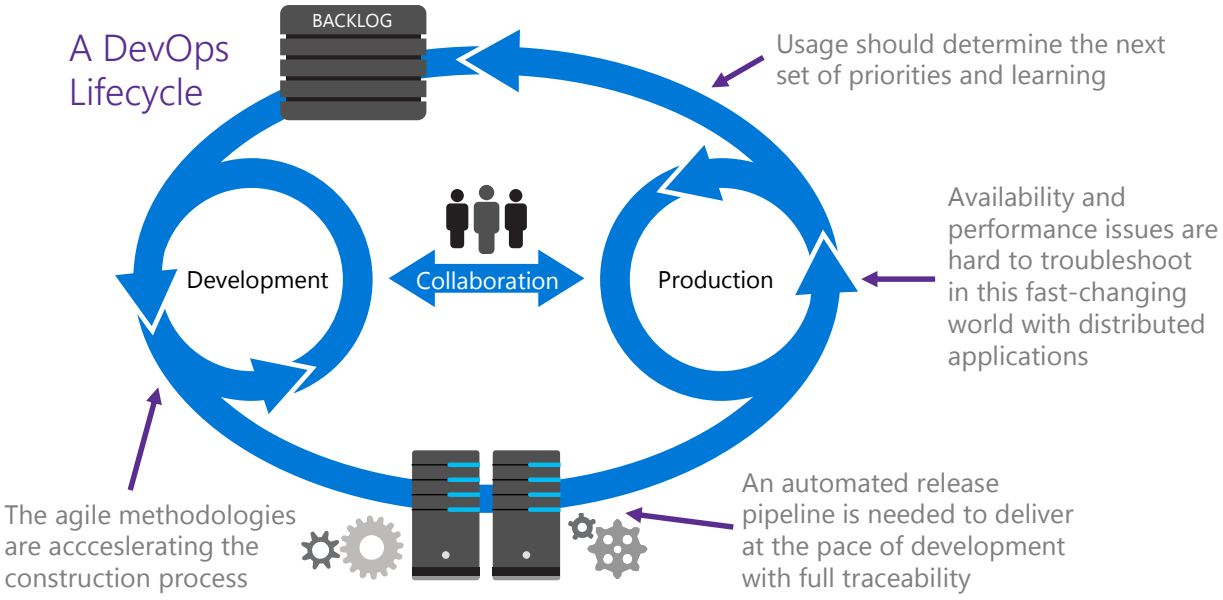

Figure 4. Four trends can be thought of as pushing the progression to DevOps.

Unlike many "born-in–the-cloud" companies, we did not start with a SaaS offering. Most of our customers are using the on-premises version of our software (Team Foundation Server, originally released in 2005 and now available in Version 2015). When we started VSO, we determined that we would maintain a single code base for both the SaaS and "box" versions of our product, developing cloud-first. When an engineer pushes code, it triggers a continuous integration pipeline. At the end of every three-week sprint, we release to the Cloud, and after four to five sprints, we release a quarterly update for the on-premises product, as shown in Figure 5.

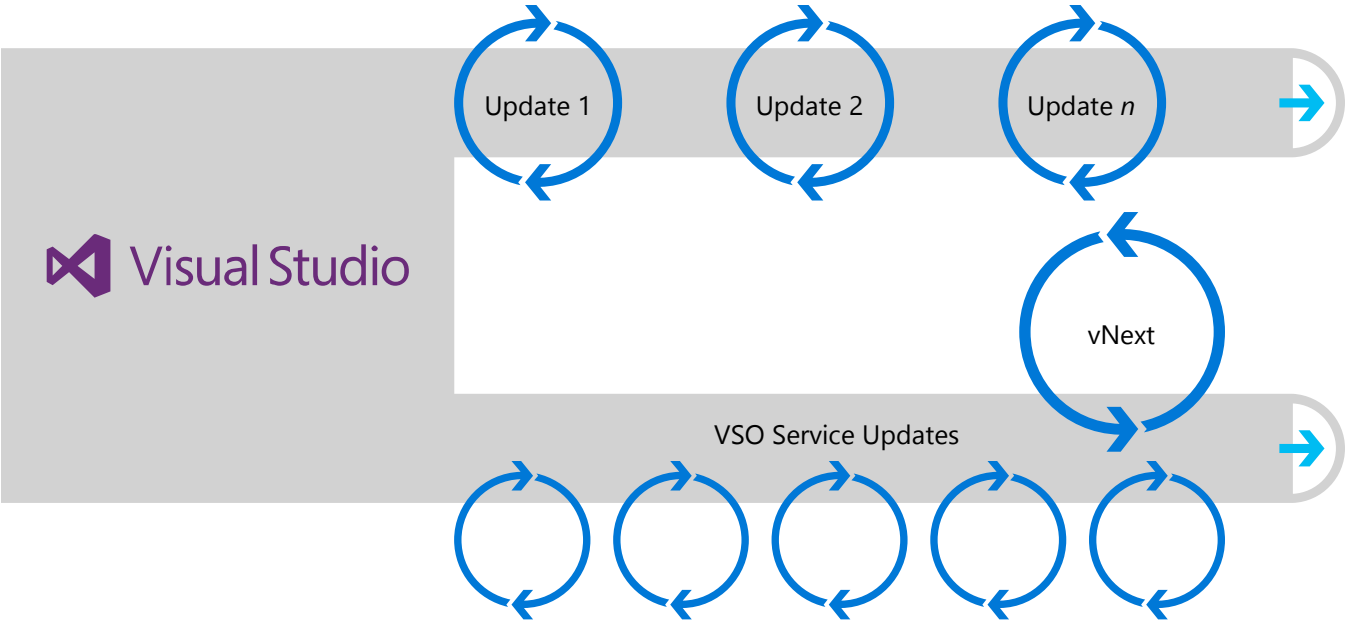

Figure 5. We maintain a single code base for VSO (the SaaS product) and Team Foundation Server (TFS; the on-premises release).
Every three weeks new features are deployed to VSO live. Every quarter these are rolled up into a TFS update. When we ship a major TFS update, e.g., 2015, it is drawn from the current VSO.

Exposure control

When you are working on a service, you have the blessing of frequent releases, in our case at the end of every three-week sprint. This creates a great opportunity to expose work, and a need to control when it is exposed. Some of the issues that arise are:

- How do you work on features that span sprints?
- How do you experiment with features in order to get usage and feedback, when you know they are likely to change?
- How do you do "dark launches" that introduce services or capabilities before you are ready to expose or market them?

In all of these cases, we have started to use the feature flag pattern. A feature flag is a mechanism to control *production exposure* of any feature to any user or group of users. As a team working on the new feature, you can register a flag with the feature flag service, and it will default down. When you are ready to have someone try your work, you can raise the flag for that identity in production as long as you need. If you want to modify the feature, you can lower the flag with no redeployment and the feature is no longer exposed.

By allowing progressive exposure control, feature flags also provide one form of testing in production. We will typically expose new capabilities initially to ourselves, then to our early adopters, and then to increasingly larger circles of customers. Monitoring the performance and usage allows us to ensure that there is no issue at scale in the new service components.

Code velocity and branching

When we first moved to Agile in 2008, we believed that we would enforce code quality with the right quality gates and branching structure. In the early days, developers worked in a fairly elaborate branch structure and could only promote code that satisfied a stringent definition of done, including a gated check-in that effectively did a "get latest" from the trunk, built the system with the new changesets, and ran the build policies.

The unforeseen consequence of that branch structure was many days—sometimes months—of impedance in the flow of code from the leaf nodes to the trunk, and long periods of code sitting in branches unmerged. This created significant merge debt. When work was ready to merge, the trunk had moved considerably, and merge conflicts abounded, leading to a long reconciliation process and lots of waste.

The first step we made, by 2010, was to significantly flatten the branch structure so that there are now very few branches, and they are usually quite temporary. We created an explicit goal to optimize *code flow*, in other words, to minimize the time between a check-in and that changeset becoming available to every other developer working.

The next step was to move to distributed version control, using Git, which is now supported under VSO and TFS. Most of our customers and colleagues continue to use centralized version control, and VSO and TFS support both models. Git has the advantage of allowing very lightweight, temporary branches. A topic branch might be created for a work item, and cleaned up when the changes are merged into the mainline.

All the code lives in Master (the trunk) when committed, and the pull-request workflow combines both code review and the policy gates. This makes merging continuous, easy, and in tiny batches, while the code is fresh in everyone's mind.

This process isolates the developers' work for the short period it is separate and then integrates it continuously. The branches have no bookkeeping overhead, and shrivel when they are no longer needed.

Agile on steroids

We continue to follow Scrum, but stripped to its essentials for easy communication and scaling across geographies. For example, the primary work unit is a feature crew, equivalent to a scrum team, with the product owner sitting inside the team and participating day in, day out. The product owner and engineering lead jointly speak for the team.

We apply the principle of team autonomy and organizational alignment. There is a certain chemistry that emerges in a feature crew. Teams are multidisciplinary, and we have collapsed the career ladders for developers and testers into a common engineering role. (This has been a big morale booster.)

Teams are cohesive. There are 8-12 engineers in a crew. We try to keep them together for 12–18 months minimum, and many stay together much longer. If there is a question of balancing work, we will ask a second crew to pull more backlog items, rather than try to move engineers across crews.

The feature crew sits together in a team room, as shown in Figure 6. This is not part of a large open space, but a dedicated room where people who are working on a common area share space and can talk freely. Around the team room are small focus rooms for breakouts, but freeform conversations happen in the team room. When it's time for a daily standup, everyone stands up.

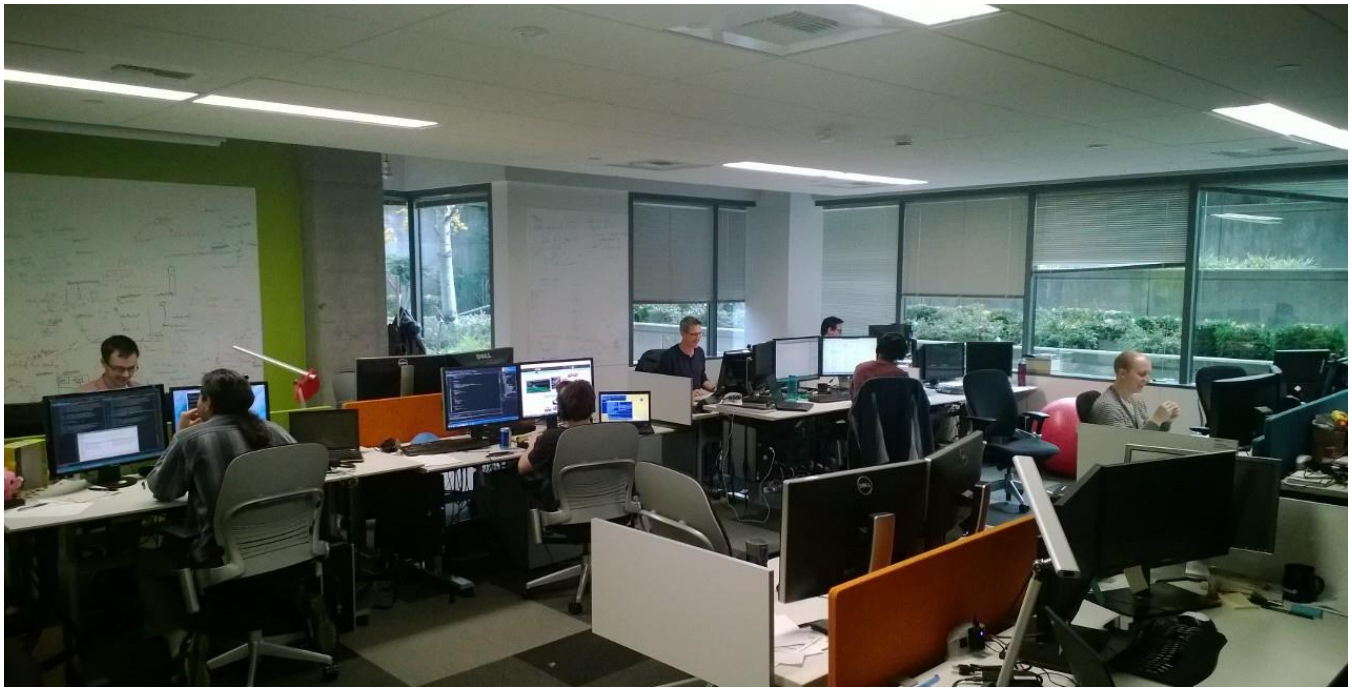

Figure 6. Feature crews sit together in a team room.

We have settled on three-week sprints, empirically. It has proven difficult for us to deliver enough value in shorter periods and coordinate across worldwide sites. Longer periods have left too much work dark. Some groups in Microsoft use two-week sprints, while others effectively flight experiments of new work many times per day without the time-boxing of sprints at all.

The definition of done is conceptually very simple. You build it, you run it. Your code will be deployed live to millions of users at the end of the sprint, and if there are live site incidents, you (and everyone else) will know immediately. You will remediate to root cause.

We rotate the role of scrum master, so that everyone experiences the responsibility of running the scrum. We keep the sprint ceremonies as lightweight as possible for outward communication. At the beginning of the sprint, the crew pulls its work from the product backlog and communicates its sprint plan in a one page email, hyperlinked to the product backlog items in VSO. The sprint review is distributed as an update to this mail, with a roughly three-minute video added. The video shows a demo, in customer terms, of what you can now do as a result of the team's accomplishments in the sprint.

We also keep planning lightweight. We will generally work toward an 18-month vision, which acts as a true north. This may be captured in some technology spikes, storyboards, conceptual videos, and brief documents. We think of every six months as a season, "spring" or "fall," and during this period we'll be firmer about commitments and dependencies across teams. After every three sprints, the feature crew leads will get together for a "feature chat," in which they share their intended stack ranks, check for course correction, ask for news, and synchronize against other teams. For example, a feature chat might be used to re-rank live site work and user-visible features.

Build, measure learn

In classic Agile practices, the Product Owner ranks the Product Backlog Items (PBIs) and these are treated more or less as requirements. They may be written as user stories, and they may be lightweight in form, but they have been decided. We used to work this way, but we have since developed a more flexible and effective approach.

In keeping with DevOps practices, we think of our PBIs as hypotheses. These hypotheses need to be turned into experiments that produce evidence to support or diminish the experiment, and that evidence in turn produces validated learning.[3]

Running a good experiment requires a statistically meaningful sample and a control group for comparison. You need to understand the factors influencing results. Consider the example in Figures 7–11. We had a problem that new users of VSO were not creating a team project right away, and without a team project, they really couldn't do much else. Figure 7 shows the starting experience as it was on the web.

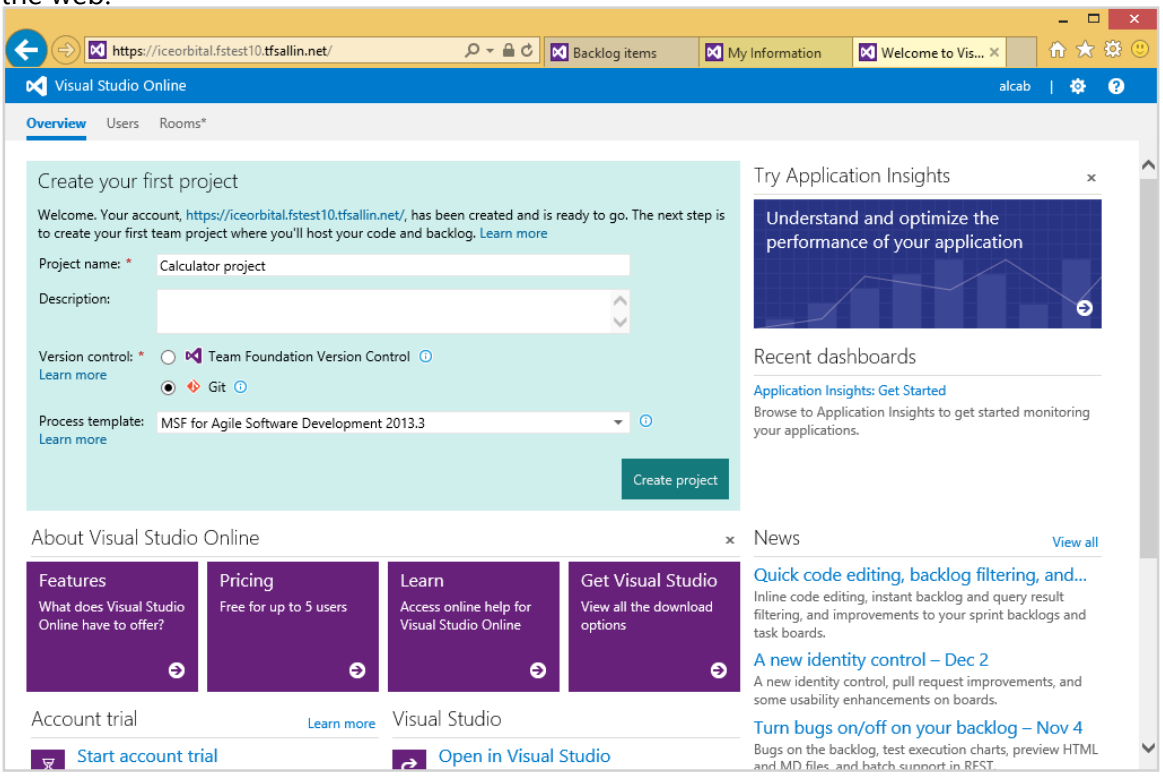

Figure 7. The old experience had too many actions and distractions preventing customers from taking the next natural step of creating a project.

We modified the experience both from the web and in the IDE. Figure 8 shows the new experience in the IDE, which encouraged users to sign up for VSO and create a project in two screens. We ran this new experience in two of six scale units as an experiment.

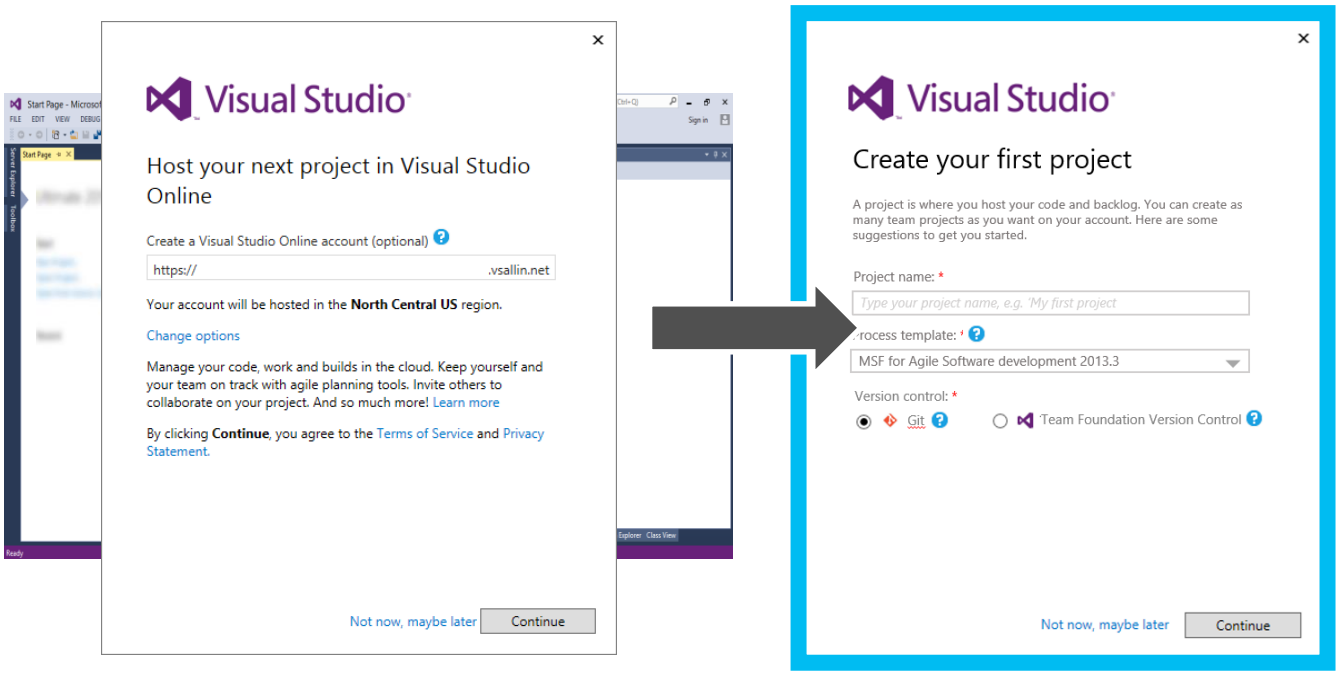

Figure 8. The focused project creation experience in the IDE flow were implemented in two scale units.

The impact of the IDE experiment is shown in Figure 9. It increased Team Project signup on new account creation from 3% to 20% (a 7x improvement)!

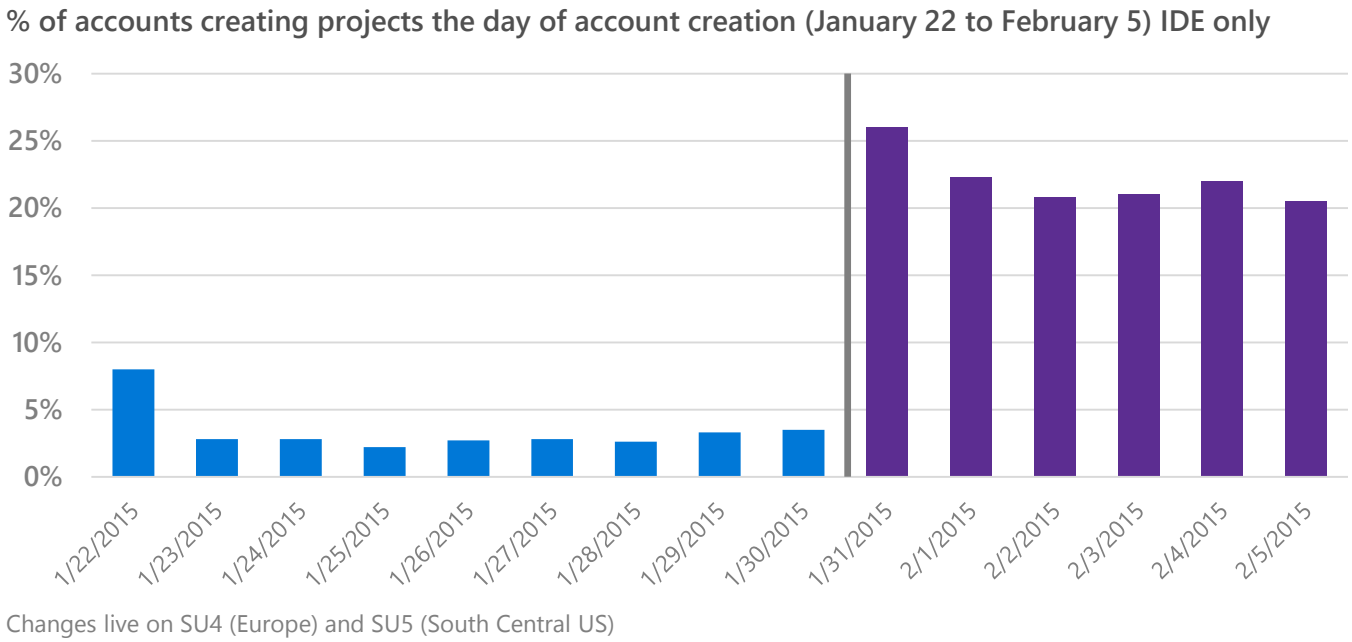

Figure 9. The IDE flow changes lead to First account creation from IDE jumped from 3% to 20%.

At the same time, we made changes to the web signup and saw a very different pattern of results. Figure 10 shows the changes that we made to the web signup.

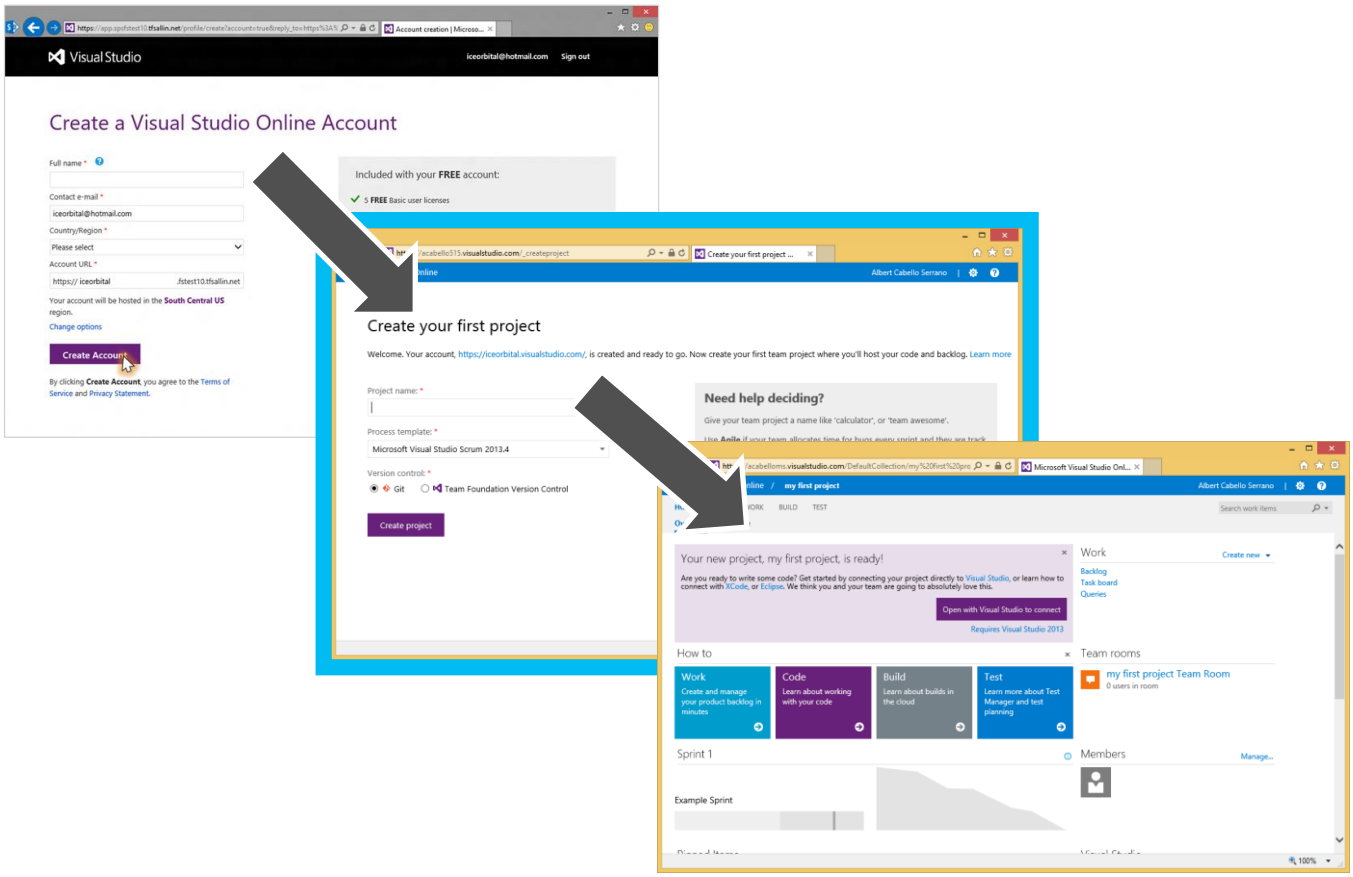

Figure 10. The web signup was similarly simplified to have project creation by itself on the second screen.

We measured the number of users who created Team Projects on the day of creating their accounts. The number improved from roughly 18% to 30% (1.7x), as shown in the transition from dark blue to purple bars in Figure 11. While good, this was surprisingly low, relative to both expectations and the IDE results.

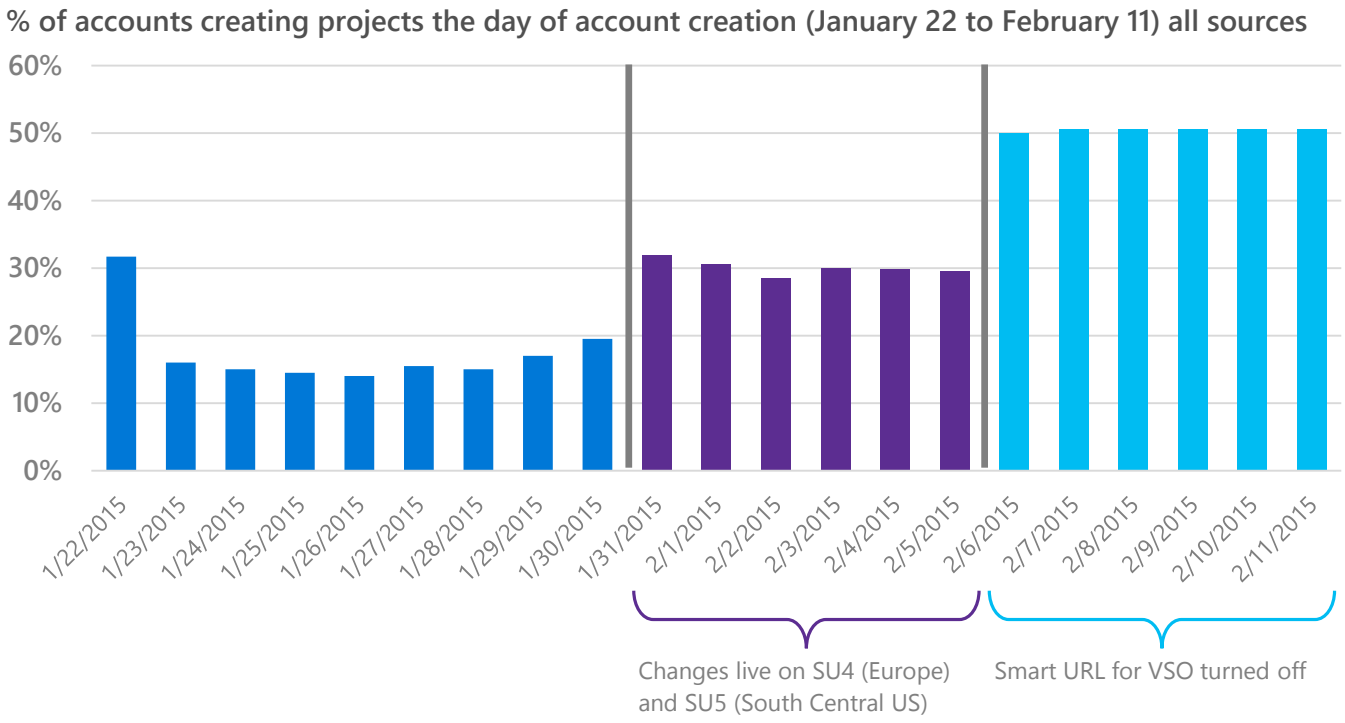

Figure 11. The results of web account creation were misleading due to the unforeseen interaction of two experiments. When one was shut off, the main results were seen clearly.

Upon investigation, we discovered that there was a second experiment running. Under this experiment, many unqualified users were being encouraged to sign up for accounts at the top of the funnel, and they dropped out. This was an example of a poor control group, as the two experiments were interfering with each other. When we stopped the accelerated account creation, project creation jumped, as shown in the cyan bars to 50% (a 2.8x increase, and 2.5x better than the IDE).

Essentially, we learned a lesson in executing our experiments. It is not always immediately obvious whether or not results are valid, repeatable, and correctly controlled.

An engineering system for DevOps

VSO is a global service, available 24x7x365, with 99.9% reliability, backed by a financial guarantee. Bear in mind that this service-level agreement (SLA) is not a goal, but is the worst we can achieve before refunding money. The goal is 100% availability and customer satisfaction. There is no downtime for maintenance. There are no clean installs, only updates, and they must be fully automated. This requires that all services need to be decoupled and have clear contracts, with clear versioning. VSO runs on the Azure public cloud, just as any customer would, so that the infrastructure is handled by Azure.

Deployment automation

Deployment and site reliability engineering are managed by a small team that we call Service Delivery (SD), which reports into Engineering. On the Monday after each sprint, SD starts deployment. It is automated with Visual Studio Release Management, as shown in Figure 12. We deploy during work hours, in case we need to investigate or remediate an incident. Because VSO stores customer data, and the database schema is frequently updated, there is also no real rollback, only roll-forward. The automation rolls from one scale unit to the next, starting with our canary, SU0, an approach which allows us to progressively control exposure and check health and user experience before proceeding to the next.

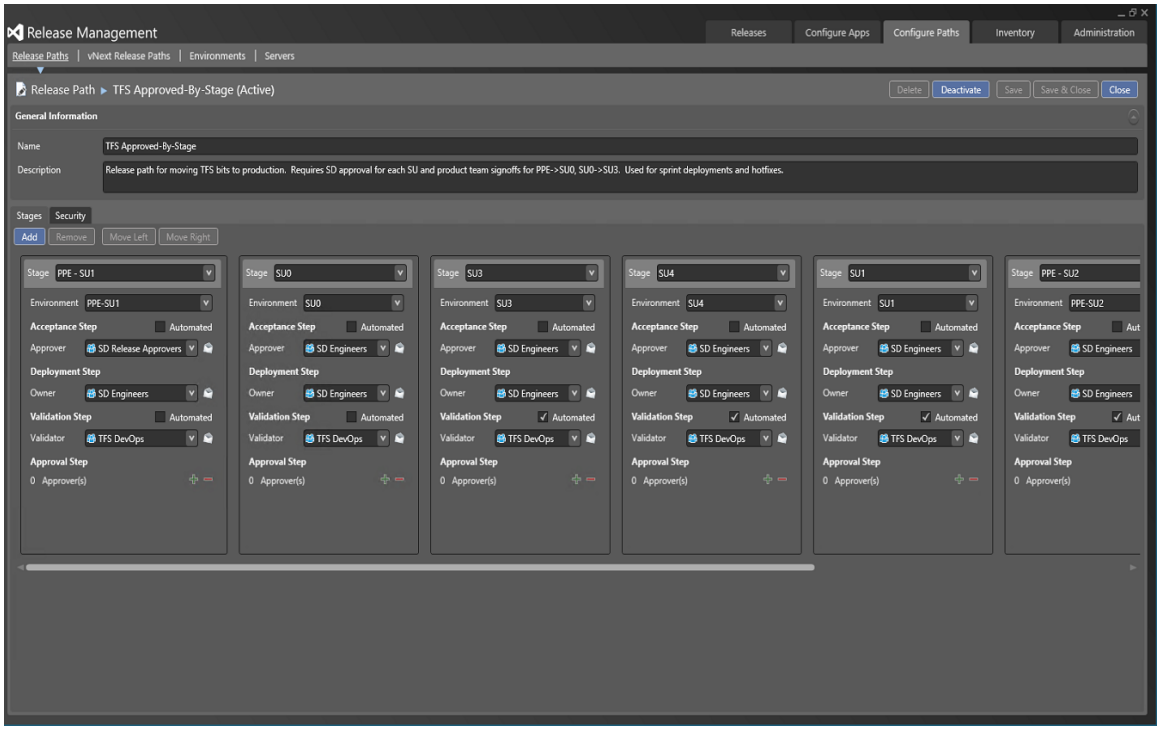

Figure 12. VS Release Management controls the deployment from one scale unit to the next in sequence.

Every deployment starts in SU0 and we run there for a few hours before letting the deployment continue. If we find a problem, we find it in SU0, and we remediate at root cause before allowing the deployment to continue. If data is corrupted and we have to exercise disaster recovery (which has fortunately happened only once), we do so here, on our own data, not on our customers' data.

Telemetry

Arguably, the most important code in VSO is its telemetry. The monitoring cannot become unavailable when the service has an issue. If there is a problem with the service, the monitoring alerts and dashboards need to report the issue immediately. Figure 13 shows an example of the service health overview.

Figure 13. This is one of many top level charts leading into the telemetry of the VSO service.

Every aspect of VSO is instrumented to provide a 360^0 view of availability, performance, usage, and troubleshooting. Three principles apply to our telemetry.

We err on the side of gathering everything.

We try hard to focus on actionable metrics over vanity metrics.

Rather than just *inputs,* we measure *results* and *ratios* that allow us to see the effectiveness of measures that we take.

On any given day, we will gather 60-150GB of telemetry data. Although we can drill to the individual account level, per our privacy policies, data is anonymized unless a customer chooses to share details with us. Some examples of the data that we gather are:

Activity logging. We gather all data about web requests made against the VSO services. This allows us to track execution time and count every command, so we can determine if particular calls or dependent services are being slow or retrying too often.

Traces. Any error triggers a stack trace, so that we can debug a call sequence after the fact.

Job history. Jobs are workflows that orchestrate activities across the service.

Perf counters. These counters are familiar to anyone who has done performance debugging, and they track that the system resources are in a healthy state. VSO generates about 50M events per day.

Ping mesh. This is a very quick visualization of the network base layer to make sure that connectivity is available worldwide.

Synthetic transactions. These are also called "outside-in tests" and are run with our Global Service Monitoring. They report health from points of presence around the world.

Customer usage. For usage, we measure our "front-doors," conversion funnel, engagement, and top customers.

KPI metrics. These are aggregated metrics that the telemetry system calculates to determine the business health of the service.

Tracking

Of course, we take nothing for granted. Everyone is trained in "Live Site Culture." In other words, the status of the live service always comes first. If there is a live site issue, that takes precedence over any other work, and detecting and remediating the problem is top priority. We have live dashboards in all our hallways, so everyone is aware of our success or shortcoming.

All Live Site Incidents (LSIs) are logged in our VSO, driven to root cause, and reviewed weekly. Figure 14 shows an example of an LSI.

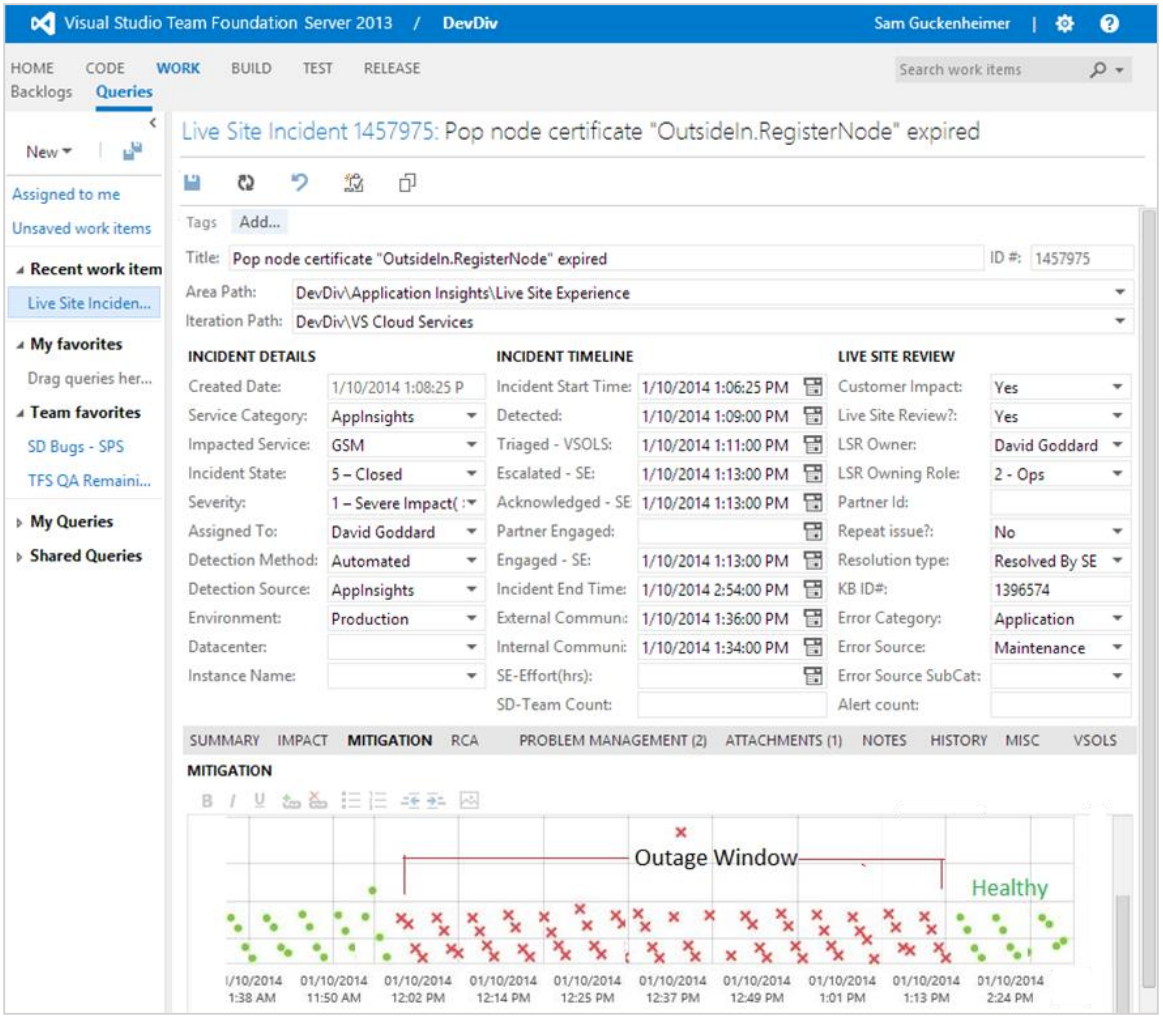

Figure 14. An LSI is tracked for every incident that occurs in the production service and includes the details of remediation to root cause.

Swarming

A frequent DevOps question is, "Who carries the pager?" In our case, the Service Delivery (SD) team is the first line of defense for 24x7 coverage, with colleagues placed globally. If they need to escalate to development, they refer to an hourly schedule of Designated Responsible Individuals (DRIs) from the feature crews. Our service-level expectation is that the DRI will be in the live site "bridge" (i.e., conference call) within five minutes during business hours or fifteen minutes in off hours.

It is important to understand how we think about "remediation." This is not bouncing the VM; rather, it is fixing at root cause. The philosophy is similar to the andon cord in a Toyota factory, which any worker can pull if a defective part is found on the line.[4] We want to fix the code or configuration at the source, harden the tests, and release pipeline to prevent recurrence. We measure our excellence in minutes for all of our live site metrics.

One of our biggest acheivements was tuning our alerts accurately enough to autodial the DRI without the need for human escalation. We achieve this by having a health model to eliminate noisy, redundant alerts, and smart boundaries to indicate when action is actually needed, as shown in Figure 15. It has given us a 40x improvement in alert precision, to the extent that by February 2015, all P0 and P1 alerts were routed correctly by the autodialer.

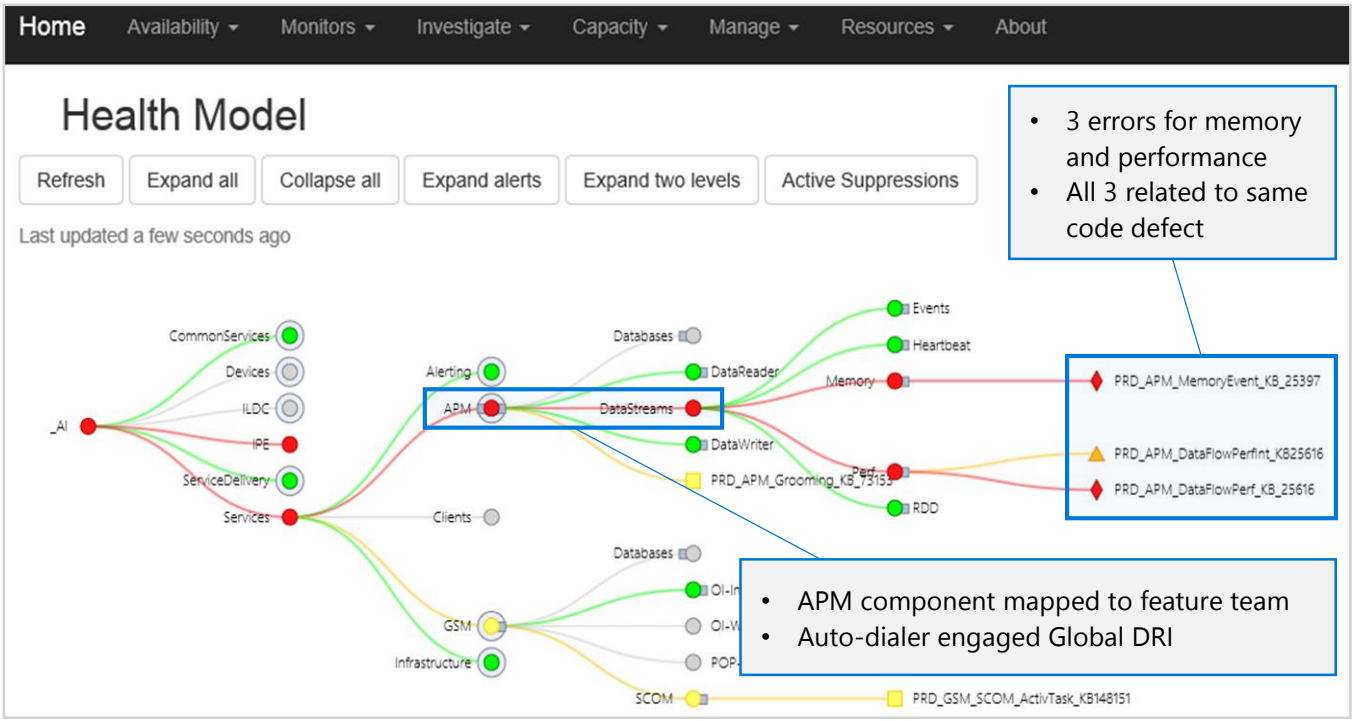

Figure 15. The health model automatically weeds out duplicate alerts and identifies which feature area is the suspect root cause.

Learning from the users' experiences

We have tried to get progressively closer to user experience in all aspects of engineering, even when that makes our jobs harder. A good example of that is how we calculate our 99.9% service-level agreement (SLA).

Our goal is not the SLA. Our goal is 100% satisfaction. This means that we need to obsess about the 0.001% of cases who are the outliers. Because outliers are often hidden by the averages, we have gone through three generations of algorithms to make our SLA calculation more stringent. These are shown next to each other on Figure 16. Initially, we used outside-in monitoring, shown in the dotted line, to track server availability. Our second algorithm looked at slow and failed commands relative to total command execution, in an attempt to identify the pain of cases that were greater outliers (see the yellow and blue lines). As the service grew, the denominator washed out too many cases that customers might have considered substandard. Our current calculation is to compute user minutes of service unavailability over total user minutes, the black line.

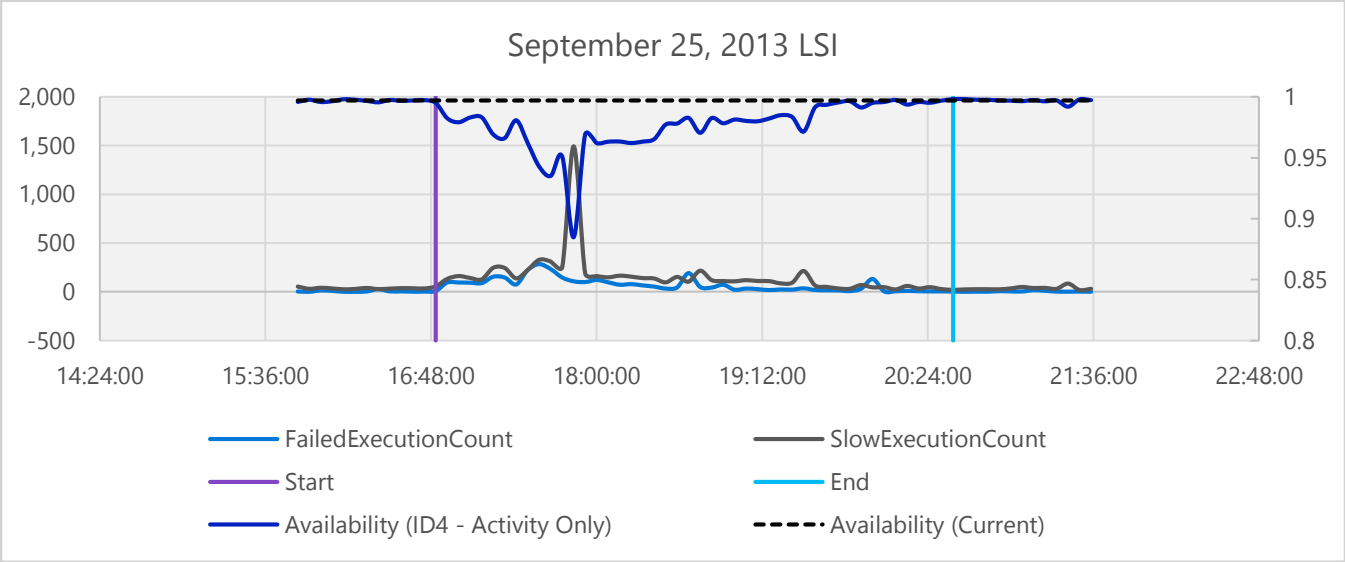

Figure 16. This LSI contrasts the three methods of calculating SLA. Outside-in monitoring finds no problem. Tracking command performance shows an hour of slow down, whereas real-user monitoring shows about three hours of delayed response.

As you can see, in a four-hour period, the first (dashed) SLA calculation shows no problem. The second (black) shows about an hour, while the third shows about three hours of poor performance. Contractually, we don't "need" to make our measurements harsher on ourselves, but we do. It's the muscle we should be building.

Security

As part of our regular security practices, we ensure data privacy, data protection, service availability, and service security. In the past, for on-premises software, we went to great pains to build defense in depth to prevent security intrusions. Obviously we still do that, but in addition, we start now with the assumption that the service may have already been penetrated and that we need to detect an intrusion in progress.[5]

In addition to using SU0 as a deployment canary, having this data center lets us run security drills, not unlike fire drills, to harden the VSO service. It is very valuable to have a scale unit in which we can run "game-day" attacks such that they will not disrupt paying customers.

New development patterns

When we exclusively produced on-premises software, we tuned our engineering system and processes to minimize *mean time between failures*. In other words, we shipped software to our customers, and when there were errors to correct, the cost of repair was very expensive. So, we did everything we could to anticipate the possible problems in advance so that we would not have to reship a new version.

DevOps turns that thinking on its head. If you can minimize the cost of repair by making a redeployment painless, then your goals are different. You need to minimize the *cycle time, time to detect, time to remediate,* and *time to learn*.

This leads to several patterns of designing for resiliency. Dependencies will fail, most likely during peak traffic. The service needs to degrade gracefully, have a thoughtful retry strategy—for example, with "circuit breakers"[6]—and get hardened with practices like Chaos Monkeys, as popularized by Netflix.[7]

Disaster Recovery plans are vital, as are "game days" in which the plans get executed. Both accidental mistakes and major disasters do happen and you need to be prepared.

Open Source

Part of our journey has been an increasing participation in the Open Source Software (OSS) community as users and contributors. OSS Workflows are all about sharing, reuse, and collaboration. OSS encourages loosely coupled, collaborating services that ship independently.

If there is an OSS solution to one of our problems, we want to use it, not reinvent it. If we create something of broad value, we want to share it. We reward engineers for creating popular components and we try to enable anyone in the company to search for components that have been approved for use. Any engineer can offer improvements to anything in the company. At the same time, internally, we do provide enterprise code governance and requirements for products supported over a long period.

Mean time to learning

No incident is done until we have captured and communicated the learning from it. We use the "5 Whys"[8] to capture the multiple perspectives on what can be done better next time. In the instance of an outage that affects our customers, we have a standard response. Brian Harry, as the responsible executive, publishes a blog explaining what went wrong and what we learned. The blog creates a public record and accountability that holds us to the improvement.[9]

It's quite remarkable to see the customer reaction to these blogs. We describe how badly we have failed, and customers thank us. This transparency is quite a pleasure, because it leads to a much more fruitful relationship than the usual vendor-customer interaction.

Connecting the business and the engineering

In the past, it was hard to integrate business and engineering concerns because it took too long to see the impact of decisions. DevOps changes that. We hope to improve the users' experiences from the moment that they first connect to us, perhaps through a web search or a trial. We watch the overall engagement statistics through telemetry, and we engage with top customers one-on-one to understand what they are thinking.

For example, we think of every customer's journey as part of a funnel, and there are multiple funnels. The journey may start on a web page, go through a trial, and lead to a deployment in Azure, on-premises, or elsewhere. If we are not happy with the success rates at any step of a funnel, we will hypothesize how to improve, experiment with the change, and measure results. We review these business experiments alongside other aspects of the service, such as live site and month-over-month growth rates in current and active users. We're careful to make sure that we focus on actionable, not vanity, metrics.

We have also found that metrics are best mixed with qualitative customer feedback. We collect feedback in many ways. Our highest-touch outreach is the "top customer program." There, we ask engineering team members to buddy up voluntarily with one of the top user accounts of our service and have a roughly monthly conversation. This provides an opportunity to ask questions about likes, dislikes, wishes, confusion, etc., which the numbers don't tell you. We have an inner circle of "champs" who are mostly MVPs and close customers with whom we preview ideas in web meetings, roughly weekly. They get to critique scenarios in the hypothesis stage, often as storyboards, and comment on priorities long before execution. In the middle, we have channels, like uservoice.visualstudio.com, where everyone can make suggestions and vote.

Learnings: the seven habits of effective DevOps

As we have moved to DevOps, we have come to assess our growth in seven practice areas, which we collectively think of as the Second Decade of Agile.

Figure 17. There are seven practice areas in which we try to improve continually as we work to master DevOps.

Agile scheduling and teams. This is consistent with Agile, but more lightweight. Feature crews are multidisciplinary, pull from a common product-backlog, minimize work in process, and deliver work ready to deploy live at the end of each sprint.

Management of technical debt. Any technical debt you carry is a risk, which will generate unplanned work—such as Live Site Incidents—that will interfere with your intended delivery. We are very careful to be conscious of any debt items and to schedule paying them off before they can interfere with the quality of service we deliver. (We have occasionally misjudged, as in the VS 2013 launch story above, but we are always transparent in our communication).

Flow of value. This means keeping our backlog ranked according to what matters to the customers and focusing on the delivery of value for them. We always spoke of this during the first decade of Agile, but now with DevOps telemetry, we can measure how much we are succeeding and whether we need to correct our course.

Hypothesis-based backlog. Before DevOps, the product owner groomed the backlog based on the best input from stakeholders. Nowadays, we treat the backlog as a set of hypotheses, which we need to turn into experiments, and for which we need to collect data that supports or diminishes the hypothesis. Based on that evidence, we can determine the next move in the backlog and persevere (do more) or pivot (do something different).

Evidence and data. We instrument everything, not just for health, availability, performance, and other qualities of service, but to understand usage and to collect evidence relative to the backlog hypotheses. For example, we will experiment with changes to user experience and measure the impact on conversion rates in the funnel. We will contrast usage data among cohorts, such as weekday and weekend users, to hypothesize ways of improving the experience for each.

Production first mindset. That data is reliable only if the quality of service is consistently excellent. We always track the live site status, remediate any live site incidents at root cause, and proactively identify any outliers in performance to see why they are experiencing slowdowns.

Cloud ready. We can only deliver a 24x7x365 service by continually improving our architecture to refactor into more independent, discrete services and by using the flexible infrastructure of the public cloud. When we need more capacity, the cloud (in our case, Azure) provides it. We develop every new capability cloud-first before moving into our on-premises product, with a few very intentional exceptions. This gives us confidence that it has been hardened at scale and that we have received continuous feedback from constant usage.

End notes

[1] For the detailed retrospective, see http://blogs.msdn.com/b/bharry/archive/2013/11/25/a-rough-patch.aspx

[2] For example, see the Gartner ALM Magic Quadrant

[3] Ries, Eric (2011), *The Lean Startup*.

[4] Liker, Jeffrey (2003), The Toyota Way: 14 Management Principles from the World's Greatest Manufacturer

[5] See http://aka.ms/redblueteam

[6] https://msdn.microsoft.com/en-us/library/dn589784.aspx

[7] http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html and http://blog.smarx.com/posts/wazmonkey-chaos-monkey-for-windows-azure

[8] http://en.wikipedia.org/wiki/5_Whys

[9] For example, https://social.msdn.microsoft.com/Search/en-US?query=outage&beta=0&rn=Brian+Harry%26%2339%3bs+blog&rq=site:blogs.msdn.com/b/bharry/&ac=4