IT Insights

A service of Microsoft IT Showcase

Streamlining the development process with feature flighting and Azure cloud services

June 2015

Cross-discipline teams at Microsoft IT engaged in a joint effort to overcome many challenges of traditional software development models. A large organization within Microsoft IT successfully streamlined its development processes, reduced overall risk, and improved the customer experience by using feature flighting and Microsoft Azure cloud services to deliver smaller changes more frequently.

Situation

One division of Microsoft IT identified common constraints inherent in traditional software development models, including branching code lines, multiple environments, overlapping resources, and insufficient automation. These constraints frequently led to code instability, bugs, high maintenance effort, schedule complications, and strained resources.

Solution

Cross-discipline teams within Microsoft IT decided to adopt a new approach that would allow faster and more adaptive processes that are better suited to a mobile-first, cloud-first business world. A clear, unified goal—reduce risk and increase quality by delivering smaller changes more often—became the motivating force behind an entirely remodeled development process.

Benefits

- Flexible, low-risk deployments
- Improved code stability
- More accurate and effective testing
- Fewer dedicated environments
- Improvements to the user experience
- Reduced development cost

Products and Technologies

- Microsoft Azure Virtual Machines
- Microsoft Azure Application Insights
- Microsoft Azure Table Storage
- Microsoft Azure Redis Cache
- Visual Studio, C#

Situation

In 2014, individual development teams within one large Microsoft IT organization were struggling to satisfy business needs within the constraints of the traditional waterfall software development model. In this model, development occurs in long, structured, sequential phases that cannot begin until preceding phases are complete. The model was proving to be inadequate for development in a rapidly changing, cloud-first business market.

Traditional development models

Microsoft IT development teams using the waterfall model faced the tremendous challenge of delivering current business software using a quarterly implementation process, which took longer to complete than the rate of change in the business. To stay responsive throughout the entire quarterly release process and maintain the efficiency of team members with very different skills, the organization overlapped work so that one cycle would still be in process when the next began. This required an increasingly complicated process of maintaining multiple code branches that were deployed to multiple sets of environments running simultaneously, that were tested by different test teams, fixed by different development teams, and managed by different sets of managers.

These long development cycles with multiple simultaneous code branches caused resource overlap and code churn. For instance, code branches were maintained independently for long periods of time but eventually had to be merged together. The longer each development cycle lasted, the greater the number of conflict occurrences between the code changes, which increased the potential for mistakes. And because features in the same branch had to be tested together and built on the same code changes, completed features could not be released early. Such interdependencies created bottlenecks that made large releases with huge numbers of changes to the code both necessary and inevitable. These large releases required significant investment in hardware and maintenance; the physical environments needed for development, integration, testing, and production required procurement and maintenance of many servers.

The consequence of traditional development models included unstable code and bugs, unmanageable schedules, a large maintenance investment, and an increasingly large strain on resources. Because of the historical expectation that software development is a lengthy process, this model was considered acceptable until recently. However, the technology and business landscape is changing rapidly, and software development models must change with it.

Today's business needs

The business climate is changing faster than ever, and when the business needs of an application change during development, the development process must have the flexibility to accommodate them. Application development in a mobile- and cloud-centric market can no longer be a one-time process; rather, it must comprise an ongoing series of continuous deliverables and regular releases. The software development model must evolve to meet the ever-increasing demand for immediate fixes, regular improvements, higher quality, and rapid adaptation to new device requirements.

The Service Engineering, Quality, Development, and Program Management teams in Microsoft IT all had a history of regular but largely independent efforts to increase agility in the development process. These individual, targeted solutions and methodologies proved valuable, but had limited scope for the larger organization. Only as part of a large-scale effort that involved an entire organization within Microsoft IT were they able to begin moving beyond the waterfall model and toward a truly agile and streamlined process that could effectively support today's business demands.

Solution

Although the specific needs of each discipline within this IT organization vary, the teams share the overarching goal of increased development speed and quality. This made it easy to invest across disciplines in a new, unified process that meets both individual needs and those of the organization as a whole.

The first step was to identify the major constraints that were preventing rapid adaptation. Teams across the organization recognized that they were hindered by long, complex cycles with multiple code branches and environments, insufficient automation, invalid results derived from synthetic testing, and strained resources (both people and hardware).

These teams investigated emerging industry solutions to these problems and identified a common theme to drive the effort toward a new process:

Reduce risk and increase quality by delivering smaller changes more often.

Concept

This theme is based on the concept that shorter release cycles will result in reduced code quantity and complexity, which will result in reduced code churn and instability, which will then result in fewer users affected by bugs and faster, more frequent releases. In a model that supports small, frequent changes, errors are easier to identify and troubleshoot because the amount of code changed is small; when problems are smaller in scale, they can be solved much more quickly.

The potential for reducing the impact of bugs on users is significant in this model. *Impact* has been defined by Ken Johnston, Principal Software Engineer, Data & Fundamentals Quality R&D, as:

Number of bugs **x** Time the bugs are in production **x** Number of users affected

Consider this simple example: A bug is found in production that would typically take three months to fix. If 1,000 users are exposed to the bug, the impact of the bug can be calculated as follows.

1 bug **x** 90 days in production **x** 1,000 users = 90,000

With a release schedule that permits small changes and allows for bi-weekly deployments, the impact of the bug is reduced from 90,000 to 14,000.

1 bug **x** 14 days in production **x** 1,000 users = 14,000

A strong user segmentation strategy used along with the implementation of feature flighting (discussed in the next section), reduces the number of affected users even more, in this example to 10, reducing the overall impact to 140.

1 bug **x** 14 days in production **x** 10 users = 140

Finally, if the problematic feature is turned off through use of feature flighting, users are affected for even less time (in this example to half a day), further reducing the impact of the bug to five. This example represents a 99.99 percent overall reduction in user impact.

1 bug **x** .5 days in production **x** 10 users = 5

With shorter release cycles, it is unnecessary to wait for the next full release to deliver a bug fix, which can drastically reduce the impact of bugs by controlling the amount of time that bugs exist in

production. Figure 1 illustrates how a shorter release cycle improves the user experience by reducing the impact of bugs.

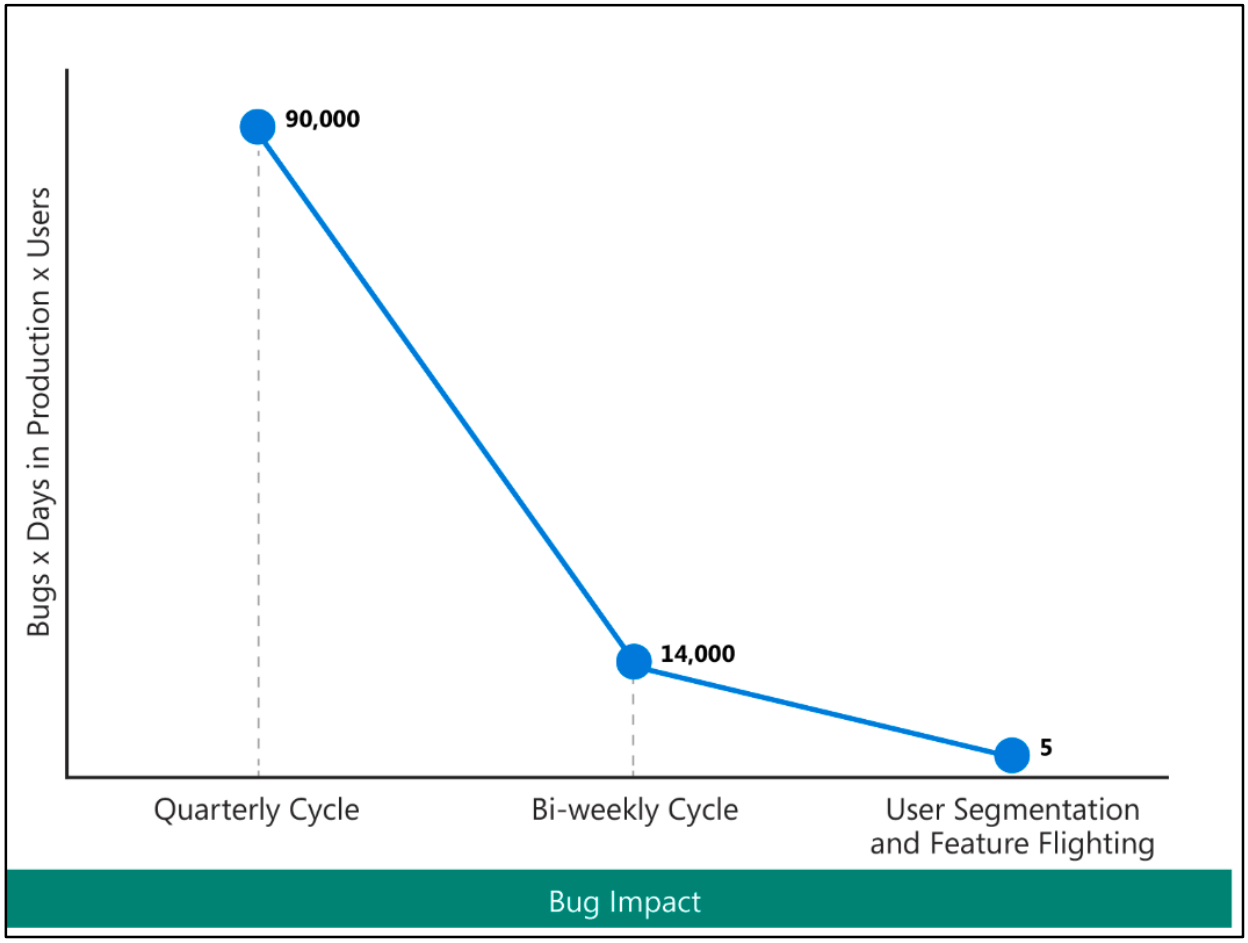

Figure 1. Bug impact reduction enabled by shorter release cycles and feature flighting

Isolating code by flagging features (discussed next) and delivering smaller changes more frequently greatly reduces code churn, bugs, strain on resources, and overall risk, resulting in an improved overall user experience.

Elements

Achieving the goals of reduced risk and increased quality involves implementing a comprehensive set of strategies and tools. The essential first step in creating this more agile model is to adopt the practice of feature flighting, which, in turn, makes code branch reduction possible.

Using feature flighting

Feature flighting, also known as feature toggling or feature switching, is a technique in which a feature or set of feature-related changes can be *flagged*, effectively turning them *on* or *off* in any environment, including production. Individual features can be tested and turned on as soon as they are ready, without dependency on other code. This flexibility allows integration of code changes early and often and eliminates the need for the multiple code branches that are inherently problematic with the traditional waterfall development cycle. Feature flighting is closely tied to the practice of user segmentation, which allows for impact reduction and collection of valuable telemetry data, two essential aspects of this development model.

"Feature flagging enables IT to put features into production earlier and gives the business the flexibility to deploy those features to users when they are ready—and often earlier—than they otherwise could have without it."

- Kevin Young, Principal IT Solution Manager for Microsoft IT

Code branch reduction

Feature flighting reduces the need to maintain multiple code branches and decreases much of the complexity encountered in overlapping waterfall development cycles. Instead of branching code, making changes, and then merging the branches back together—which becomes more problematic the further removed the code is and the longer the delay before the final merge—developers, in most cases, can work within the same branch.

Feature flighting forces developers to decide how a feature will work in both the *on* and *off* states during the initial development process. While writing the code, they make decisions about how it will work when the changes are fully enabled and exactly how the old code will be removed, rather than delegating complicated merging decisions to the future—and often to another developer, who may not understand the code as well. Although this increases complexity by enforcing a new quality standard that requires testing in various states, having fewer code branches reduces the disadvantages and inherent risk of merging code at the end of a long cycle.

Implementation

Taking full advantage of a model that delivers smaller changes more often involves implementing a continuous and iterative development cycle. As code is developed and revised, it follows a cyclical path in which features are flagged, exposed to appropriate users, analyzed via telemetry tools, and revised again as necessary (Figure 2).

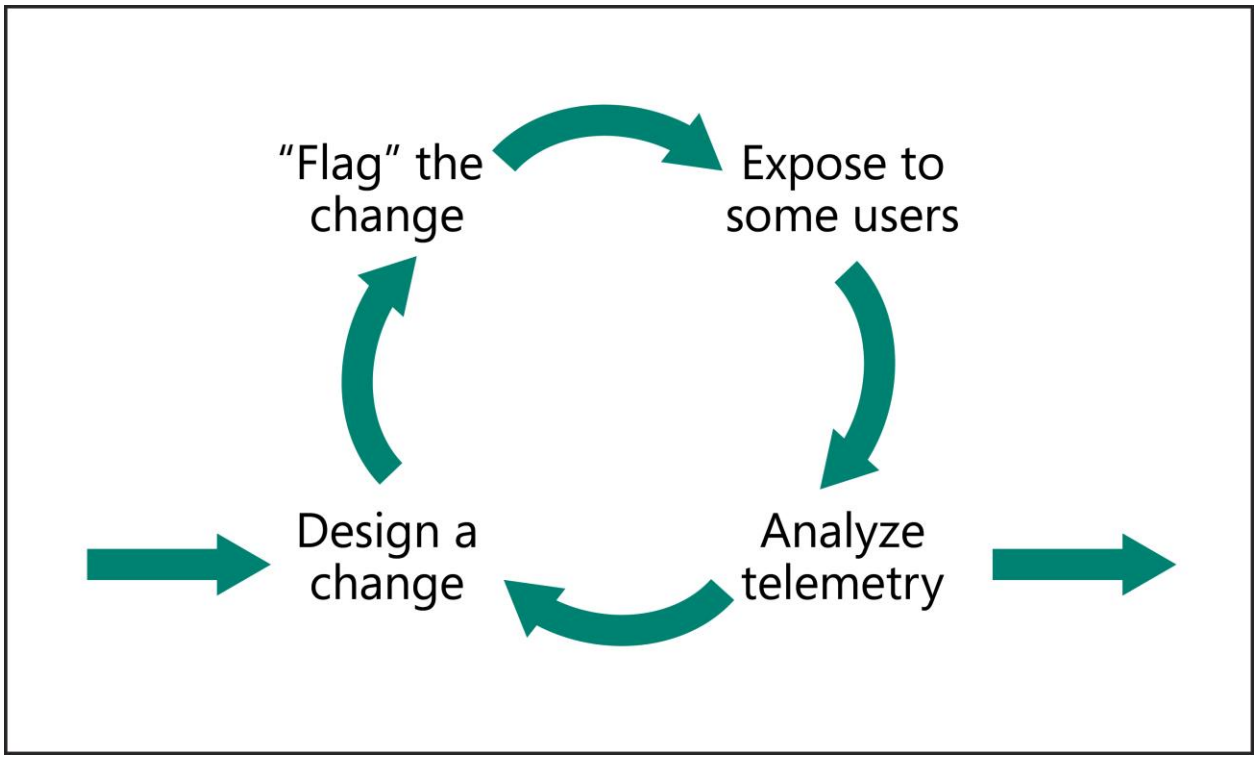

Figure 2. An iterative development cycle, incorporating flagging, user segmentation, and telemetry

Feature flighting

One organization within Microsoft IT defines feature flags by using simple table storage and a Redis cache in an Azure portal. A flag is created during the development of each new feature, and a helper library in the code checks the state of the flag. At run time, an *if* statement executes to determine whether each feature is turned on (enabled). By default, all code is deployed in the off (disabled) state, and it can be turned on at any time for all users or a segment of users. Essentially, feature flags have three basic states:

- OFF for all users
- ON for some users
- ON for all users

If a feature is off, no users will see the feature. If the feature is on, the logic determines whether the feature is enabled by default for *all* users, and then determines whether the feature should be enabled for that *particular* user. If a particular user is not in the user segment in which the feature is enabled, the feature is disabled for that user.

User segments can be defined by a variety of factors, including personal identifiers (such as user ID, email address, account ID, and IP range), general characteristics (such as org type, geography, locale, language, and browser type and version), percentage of the user population (calculated by using, for example, the last digit of user IDs or the hash code of any string), or any combination of these.

Figure 3 shows an example implementation of feature flighting from one team in Microsoft IT. In this case, a simple flag is used to expose a new page design to some users while leaving the experience unchanged for other users.

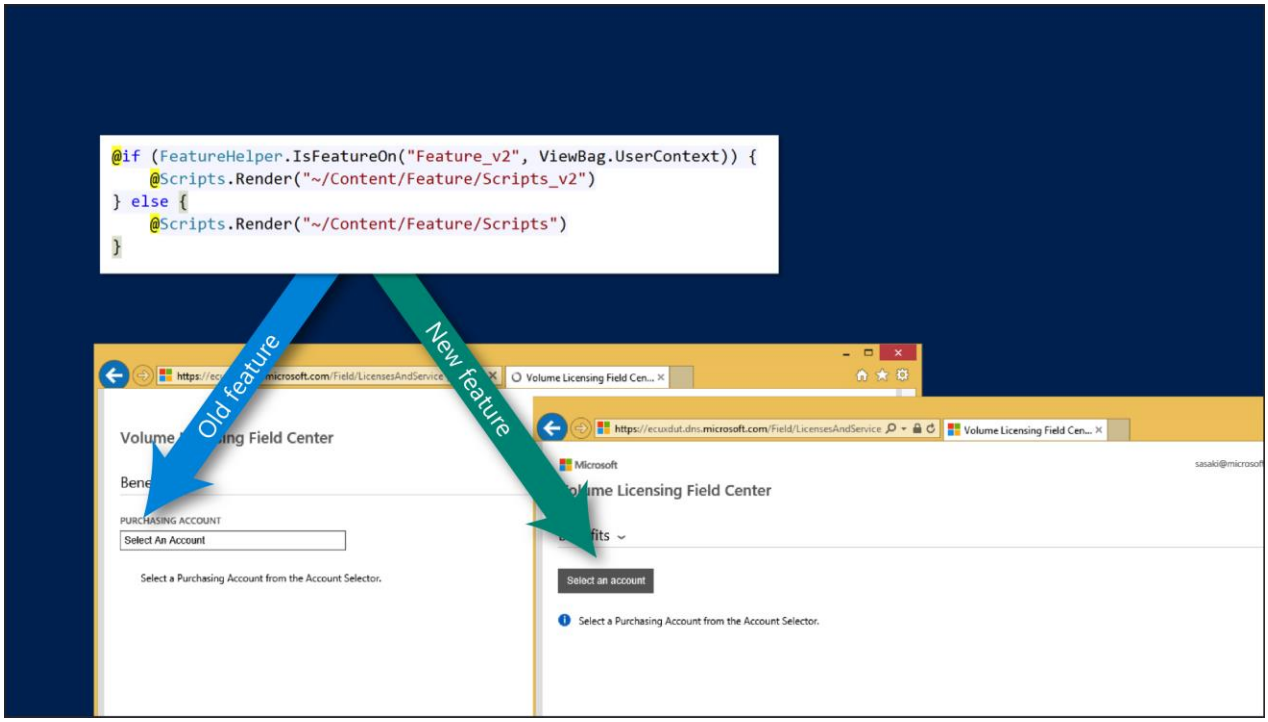

Figure 3. Feature flighting used to control exposure of change to users

Behind the scenes, a simple admin page provides full control of each feature's flag status and user exposure (Figure 4).

Figure 4. Feature administration page, used to set flagging and user segmentation

The feature administration portal is hosted as a Microsoft Azure website and uses Azure Table storage and an Azure Redis cache service to maintain the list of feature flags and user segments (Figure 5).

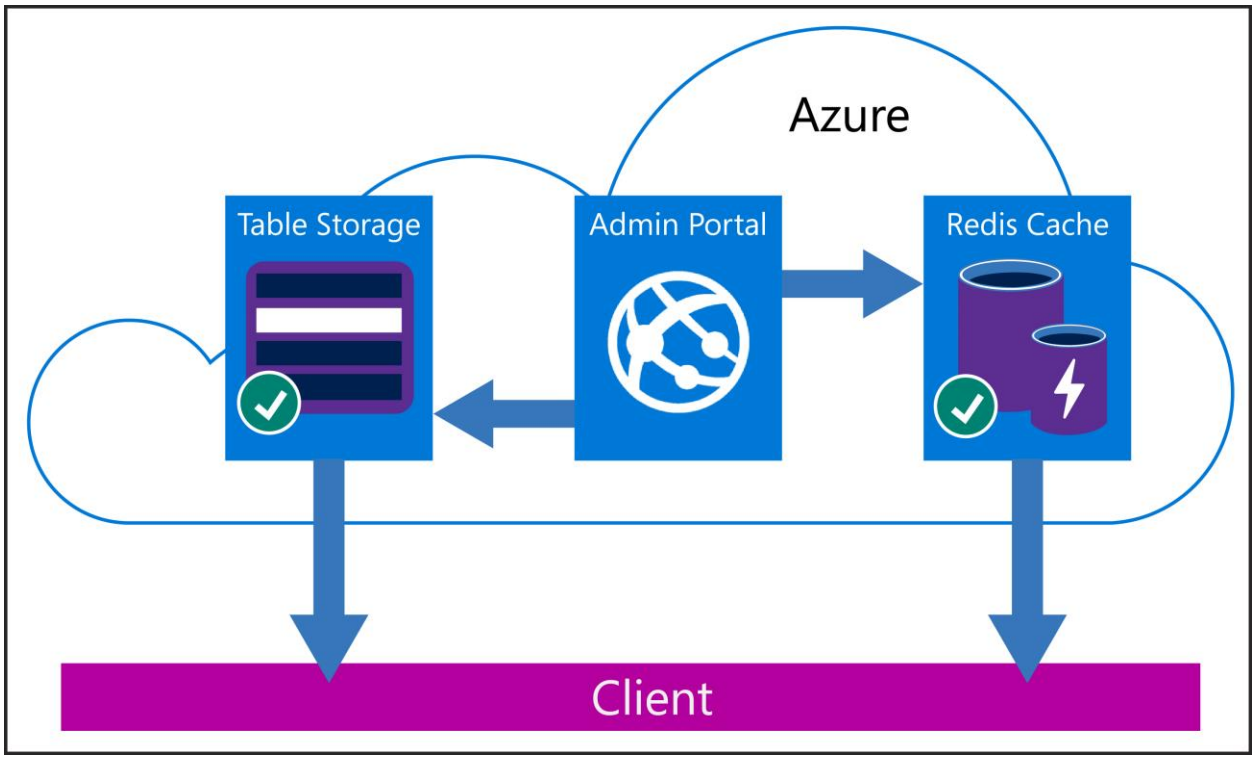

Figure 5. The feature flighting platform

Components that are dependent on the feature flags use a small client library that initializes a connection to the table storage and subscribes to updates from the cache. When the state of a flag changes, a signal is sent via the Azure Redis cache to each of the components, telling them to reload the flags and user segment values from the server.

User segmentation

Flagging features to allow carefully controlled exposure to specific users and user groups is a particularly valuable capability in an agile development environment. In fact, user segmentation has three distinct functions:

- **Code isolation.** By implementing user segmentation, developers can work on code in the *off* state, check it in, and test it without affecting any users. Two different versions of the code can run simultaneously in the same environment, allowing work to be performed independently on different releases in the same code branch.
- **Exposure control.** The impact of bugs can be reduced significantly by exposing changes incrementally. For example, a change could first be exposed to developers only, then to a wider group of internal users, then to preview users, and finally to all users. This process is extremely valuable for risk management: as a changed feature is tested, developers can slowly increase the risk level by increasing the exposure of a change. The highest risk occurs at the beginning with the smallest set of users. As risk is reduced, a feature can be exposed to progressively larger groups of users, as shown in Figure 6.

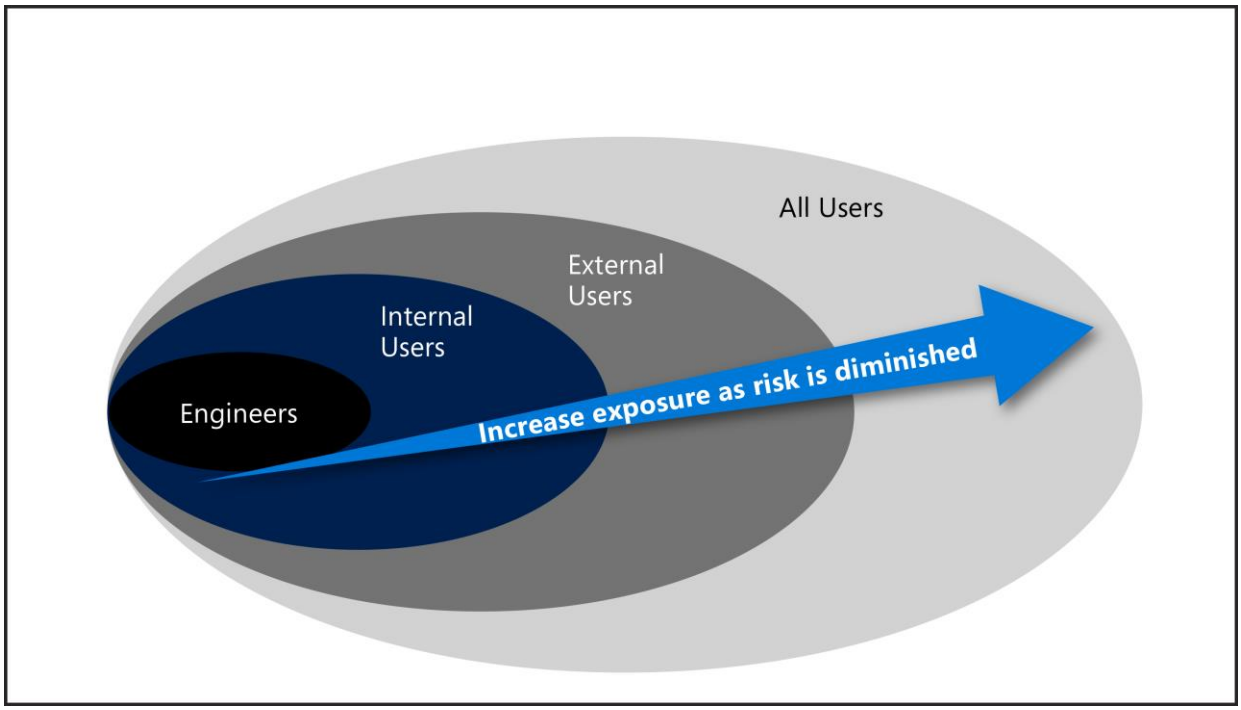

Figure 6. Exposing change gradually with user segmentation

- **A/B testing.** User segmentation is a valuable tool for collecting data on feature performance and user response. For example, two versions of a change can be released, user data gathered (via telemetry, discussed next), and results compared to determine which version is better received or understood by users.

Telemetry

Incorporating telemetry (user data collected for analysis) is necessary to realize the full value of feature flighting and user segmentation. Telemetry allows development teams to prove the value of a

change in the real-world environment by testing on a subset of users before increasing exposure to wider audiences. This reduces risk and allows for *real* (rather than synthetic) testing. Collection of real user data provides immediate feedback about the quality of the system and the potential impact of any change, which allows for rapid and targeted adaptation of the feature and eliminates code rollback scenarios. The Service Engineering team in Microsoft IT collects telemetry data by using Microsoft Azure Application Insights, shown in Figure 7.

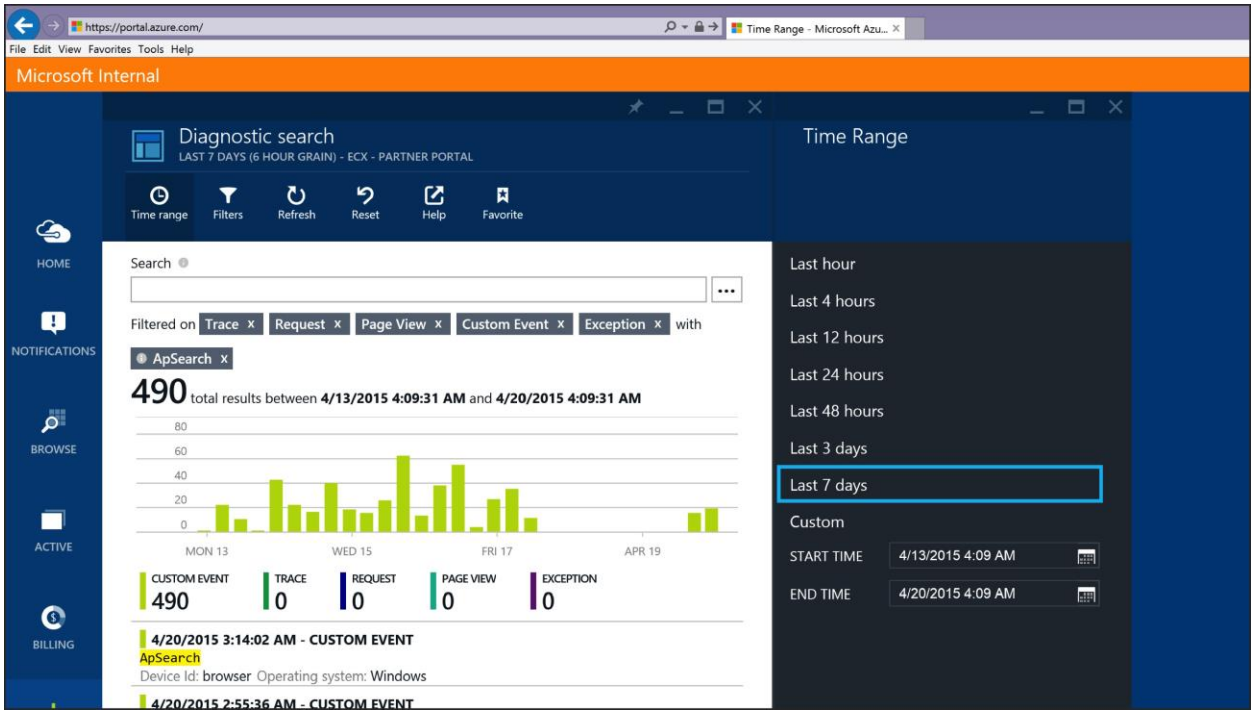

Figure 7. Analyzing telemetry with Microsoft Azure Application Insights.

This tool allows development teams to monitor a live application for availability, performance, and use, and includes immediate alerts for downtime and failures. For example, the performance of an application can be monitored before and after a feature flag is enabled to quantify the exact effect the change has in production and help determine whether to leave the feature on or increase its exposure. With straightforward configuration and minimal coding required to get started, Application Insights makes it easy to include collection and analysis of telemetry data in the development cycle.

Cloud development environment

With a more agile development process comes the need for faster deployments to production. When the Service Engineering team began using Microsoft Azure Virtual Machine environments, the procurement and provisioning of servers was reduced from days to just a few hours. A few hours after provisioning, automation could be run on these servers to meet the platform software needs of engineering, followed immediately by deployment of the software being developed. With virtual machines, a complete development environment can be established within a couple of days.

The benefits of virtual machines and feature flighting are reciprocal. While the use of virtual machines gives development teams rapid access to development environments, feature flighting reduces the need for multiple dedicated environments for overlapping releases. Because feature flighting enables branching through code, as opposed to physically branching, multiple versions of the code can reside in the same physical environment; coupled with a strong segmentation strategy, this greatly relieves the environment dependency. Only one or two sets of environments must be provisioned, releases need not be rolled back when problems are encountered, and the strain on hardware resources is greatly reduced.

Benefits

Increasing quality and delivering smaller changes more often by implementing feature flighting, user segmentation, and telemetry offers a variety of benefits to the process, user experience, and environmental footprint.

Improvements in engineering processes

- **Flexible, low-risk deployments.** Separation of release from deployment greatly simplifies the release process—a release can be a simple matter of turning on flags—and significantly reduces risk—the release of code does not have to alter the user experience at all.
- **Increased stability.** Reducing or eliminating major code merges results in more stable code, and simplifying branches minimizes the complexity of managing dependencies. Deploying and testing small changes more often can dramatically reduce problems at release time.
- **More accurate and effective testing.** Collection of real user data through user segmentation and automated telemetry allows for testing in a real—rather than a synthetic—production environment, producing more accurate and actionable results.
- **Faster time to market.** Reducing the size of deployments and changes allows code to flow into production faster. The time between gathering requirements and deploying the code to production is directly proportional to the size of the code change.

Improvements in user experience

Users also benefit substantially from feature flighting and the development model it enables. The overall impact of bugs is greatly reduced, because fewer bugs reach end users and bug fixes can be released much faster. Improvements to the application are also faster and more meaningful because of the telemetry data that can be acted upon as part of an iterative development process.

Reduction in environmental footprint

Feature flighting reduces the risk of rollback, which in turn reduces strain on personnel resources; development teams at Microsoft IT who have implemented this model no longer need to spend entire weekends on deployments. By reducing overlapping code branches, Service Engineering can dedicate resources to fewer environments and place heavier focus on ensuring higher availability and increasing build and deploy automation to improve overall build speeds.

Success stories

Since the initial implementation of feature flighting and Azure virtual machines into the development process, Microsoft IT teams have observed a variety of notable improvements in processes and direct benefits to users and the business. The following are a few of their success stories.

Reducing the impact of change with user segmentation

Microsoft IT was tasked with switching from an on-premises account search feature to Azure's new search-as-a-service features to take advantage of significant performance benefits. The feature was finished early and tested in a pre-production environment using synthetic testing. Because it was developed in the common code branch, the completed feature was delivered to production (in the *off* state) several weeks ahead of schedule, allowing the opportunity to turn it on and verify it in production for an internal user account. An unexpected issue surfaced that had not occurred in the pre-production environment, which used simulated data and fake user accounts. Because the feature was delivered early, the problem was fixed with time to spare; at release time, there was no doubt that the feature would work.

Without feature flighting and user segmentation, this would not have been possible; the issue would not have been discovered until the entire project went live at the end of the quarter. The problem

would have required a very expensive hotfix, or the customer would have had to wait for the next quarterly release for a fix.

Avoiding rollbacks and hotfixes by disabling features

At the same time that teams within Microsoft IT were preparing to transition to the new delivery process, they were nearing completion of a final full waterfall release. The release was massive, involving significant changes to the user onboarding process and more than a million lines of code. During the typical pre-production and user acceptance testing process, synthetic user accounts and data represented production conditions, which included some data provided by an external data source. When all the deployment and database updates were complete at the end of a long weekend, the team discovered that the external data source was not providing the correct data, and the data could not be corrected for more than a week. However, waiting a week to invite new users to the platform would have caused a significant impact to the business and placed the entire release at risk.

With traditional development models, there are two options in this situation: roll back over a million lines of code changes, fail over to an old version of the database, and postpone the entire release, or release a broken user onboarding process—an embarrassing and extremely expensive option. However, in preparation for conversion to the new delivery process, IT teams had begun implementing flighting for new features. It quickly became obvious: the feature could simply be turned off until the external data dependency was corrected. No rollback or hotfix would be necessary, and the impact to users was eliminated.

The benefits of feature flighting were fully realized after the external data was corrected and the feature was turned on for a single user account. This offered real—not synthetic—assurance that the feature was functioning correctly and could be enabled for all users.

Cost savings through environment reduction

When Microsoft IT successfully implemented its feature flagging capability in November 2014, the February 2015 quarterly release was already in progress. While preparing for the February release, teams also had to plan for a special April release to meet an important business requirement. With the previous development model, developers would have needed environments for Unit Test, Test Automation, Systems Integration, Performance, and User Acceptance for both of these releases. The Service Engineering team would have had to build and support two sets of environments simultaneously, one for the quarterly release and another for the business-critical release in April.

Introduction of feature flighting eliminated the need for two sets of environments. Because the code was segregated by release, the Service Engineering team simply needed to add a single additional environment; they tested the code in both the enabled and disabled states for both releases to ensure that code for both was working as expected. In this scenario, the combined use of feature flighting and Azure virtual machines reduced the environmental footprint by approximately 37.5 percent. Deploying to the same environment also reduced the time required for deployment and the resources associated with maintaining them, which translated directly to business cost savings.

Lessons learned

As Microsoft's "first and best customer," Microsoft IT values the learning process as an educational tool for both internal and external IT customers. The following are lessons learned and best practices developed as part of this development model transition.

- **Embrace cultural and ideological change.** Long-held standards about successful development practices can be challenging to change, but conceptual shifts are necessary to evolve with changing standards. For example, the practice of moving unused, untested code to production without fully testing it has historically been considered a bad practice. However, feature flighting addresses and challenges this concept, and a necessary cultural shift is taking place in software development best practices throughout the industry. Acknowledging that smaller changes

released more often can create stability, rather than reduce it, is another shift that must take place. Best practices regarding testing must also change; developers implementing flighting must consider how a feature works when turned on, as well as how the system will work when the feature is turned off, which introduces new but worthwhile complexities.

- **Identify and work toward common goals.** Setting unified goals across an entire organization is critical to successful change. Without identifying clear benefits for each party, change can be nearly impossible to enforce in a large organization. Investing in an evaluation of team currencies is critical to the establishment of common goals. The concept of *currency* means identifying each team's independent goals, objectives, and approaches to success. Examples of currency might be customer satisfaction for one team, quantity of delivered features for another, and number of escalations observed for yet another. Making an effort to identify and meet currency needs can help each team sell an idea to its leadership and make the change effective, ultimately providing an overall business value goal that is shared across teams.
- **Integrate automation**. Feature flighting addressed many of the challenges that Microsoft IT teams initially faced by reducing branching and simplifying environments. And teams are aggressively working toward improving testing automation. Frequent releases require frequent testing; if testing cannot keep up with release cycles, the ability to maintain iterative release cycles is limited. Microsoft IT understands that maximizing automation and fully integrating it into the regular process deliverables is critical to successful implementation of testing in a development model that uses feature flighting.
- **Take advantage of cloud solutions.** The ability to flag features makes sophisticated user segmentation possible, and telemetry collects and interprets user data; Azure Application Insights is an ideal telemetry tool for gathering meaningful user data. Because feature flighting also reduces overlapping code branches, Service Engineering can focus its resources on a single set of environments, ensuring higher availability and increasing overall build speeds. This streamlined focus and the use of virtual machines allows faster procurement and provisioning of development environments, reducing the time investment from days or weeks to a matter of hours.

Conclusion

One organization within Microsoft IT found that they could significantly reduce risk and increase quality in the development process by delivering smaller changes more often. This approach was implemented using feature flighting, user segmentation, and telemetry. The new development model resulted in improved code stability and flexibility, more accurate and effective testing, the ability to release more quickly, and an improved user experience. By taking advantage of cloud services such as Azure Virtual Machine environments, Microsoft IT teams further streamlined the development process and significantly reduced strain on resources. Together, the use of feature flighting and Azure cloud services resulted improved customer satisfaction, a more efficient IT organization, and an overall reduction in both time and development costs.

Resources

Video - Streamlining the development process with feature flighting and Azure cloud services
https://www.microsoft.com/itshowcase/Article/Video/560

MSDN blog post: "The future of quality is easy with EaaSy and MVQ"
http://blogs.msdn.com/b/kenj/archive/2014/05/20/the-future-of-quality-is-easy-with-eaasy-and-mvq.aspx

Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation
http://www.informit.com/store/continuous-delivery-reliable-software-releases-through-9780321601919

Products and technologies

Microsoft Azure Application Insights
http://azure.microsoft.com/en-us/services/application-insights/

Microsoft Azure Virtual Machines
http://azure.microsoft.com/en-us/documentation/services/virtual-machines/

Microsoft Azure Table Storage
http://azure.microsoft.com/en-us/services/storage/tables/

Microsoft Azure Redis cache
http://azure.microsoft.com/en-us/services/cache/

Azure Search
http://azure.microsoft.com/en-us/services/search/

For more information

For more information about Microsoft products or services, call the Microsoft Sales Information Center at (800) 426-9400. In Canada, call the Microsoft Canada Order Centre at (800) 933-4750. Outside the 50 United States and Canada, please contact your local Microsoft subsidiary. To access information via the World Wide Web, go to:

www.microsoft.com

www.microsoft.com/ITShowcase