LEVERAGING THE MICROSOFT PLATFORM FOR DEVOPS

PERFICIENT®

vision. execution. value.

This white paper offers a deep dive into utilizing the Microsoft platform for your organization's DevOps strategy. In addition to high-level definitions of DevOps and its importance to successful businesses, you'll learn where your organization falls on the DevOps maturity scale and best practices for executing on your DevOps strategies on the Microsoft stack, including leveraging Microsoft Azure and Visual Studio Team Services (VSTS).

WHAT IS DEVOPS?

DevOps is the collaboration of one or more individuals on a project to optimize:

- **ENVIRONMENT PROVISIONING**
- **ENVIRONMENT CONFIGURATION**
- **APPLICATION BUILD(S)**
- **APPLICATION DEPLOYMENT(S)**

DevOps is about improving how software is built, delivered, and operated by enabling IT departments to reduce cycle times, optimize the use of IT resources, and improve quality and availability. It is about increasing the scope of agility, and it should be viewed as a team undertaking. It requires teams to look at their full life cycle investments.

At its core, DevOps enables better software development and accelerates the last mile of delivery by focusing on:

1. Shortening of cycle times

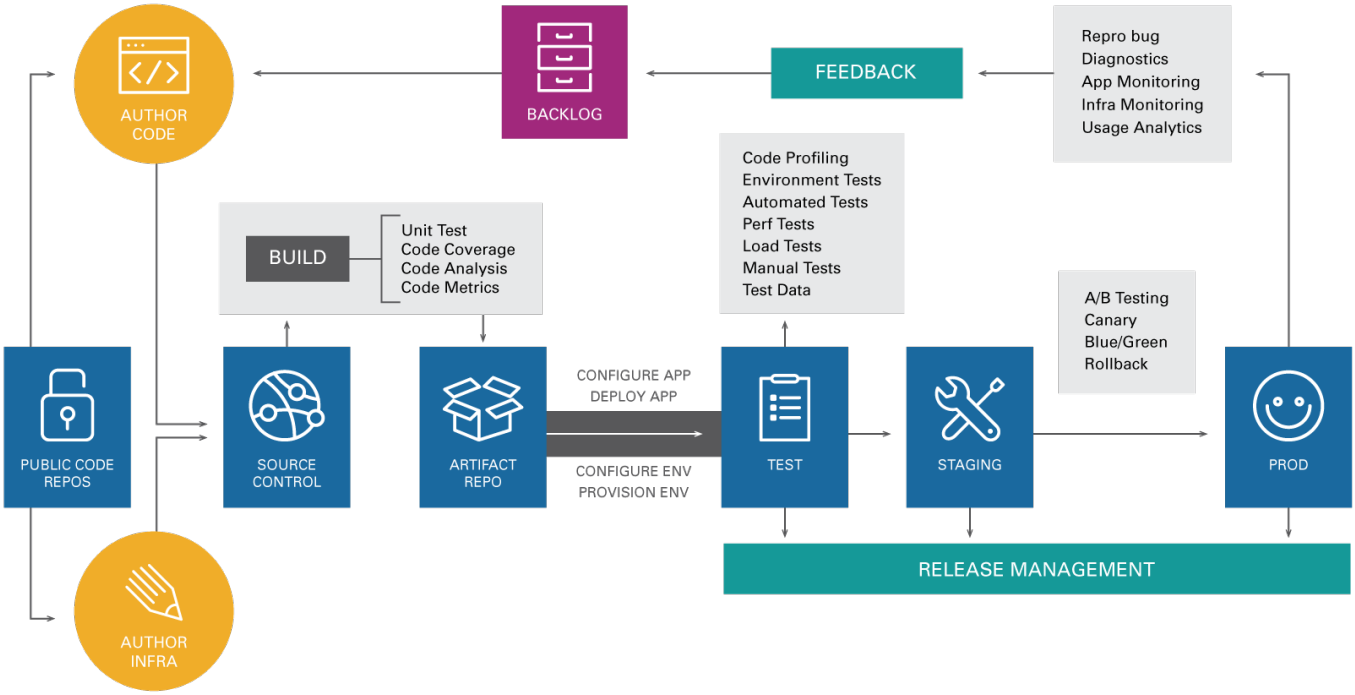
DevOps practices enable organizations to shorten cycle times and increase the traceability and auditability of each release by improving transparency and collaboration between development and operations teams, and by eliminating waste in current manual processes through automation.

2. Optimization of resources

DevOps practices enable organizations to efficiently manage environments in a way that supports self-service of environment provisioning/de-provisioning, controls costs, and uses the provisioned resources effectively while minimizing security risks.

3. Improving quality and availability

DevOps practices help to identify defects early in the development cycle, identify the root cause of issues, and quickly test and deploy fixes. DevOps also helps organizations capture rich telemetry on app performance and usage so teams can learn from the behavior of users to drive future priorities and investments.



WHY DEVOPS?

Successful organizations understand that competitive advantage is fleeting, and to continuously innovate in an era of digitized operations, they are required to write their own software.

It is important to view the development of custom applications not as a craft that happens independent of the business, but as a strategic process that generates value. This leads organizations to see development from the perspective of a value chain where value does not materialize when outputs are delivered through requirements implementation, but when outcomes are realized through the right requirements being implemented, tested, and deployed into production. More value delivered faster is, of course, better – hence the focus on throughput/flow.

Focusing on organizational aspects such as people, processes, and tools will help improve the flow of value and eliminate waste. Ultimately, successful organizations achieve the following processes and goals with DevOps:

1. ADAPT AND LEARN

Requirements are refined through a quick succession of feedback loops. More conservative organizations do this through process patterns, such as specification by example, where the business, developers, and testers collaborate closely on refining the specification and continuously verifying (testing) that the implementation is loyal to it. These specifications by example include behavior-driven development, acceptance testing development, or simply agile testing. This is all relevant in the context of improving quality. More mature organizations will perform these checks directly in production, by observing user reaction to features. Adapt and learn is also known as hypothesis-driven development or build-measure-learn.

2. SHORTEN TOTAL CYCLE TIME

Successfully adapting and learning leads to a shortened total cycle time. Knowledge – no matter how formal the requirements and specification definition process – is mostly tacit. For learning to take place in a meaningful way, feedback loops are better when shorter. Shorter cycle time is also directly related to quality. Shorter cycles impose smaller batches of work, which in turn (as researched and documented by queuing theory) also has a positive, non-linear effect on quality. Quality in this context is defined as structural and functional – valuable software is more likely to manifest when executed in short cycles rather than in long phases.

3. ELIMINATE WASTE

Short cycle times demand the removal of wasteful activities. Spending days promoting code to the acceptance testing environment is not conducive to rapid feedback loops and shortened lead time to value. Optimized use of resources (e.g., environment provisioning) is a common attribute in successful and high-performing organizations.

4. COLLABORATE

Successful organizations work better together by building camaraderie and stronger working relationships. Collaboration is created through adapt and learn, and it contributes to shortening the time cycle and eliminating waste.

5. REDUCE RISK

For shortened cycle times to be sustainable, safety is necessary. Availability and resilience (or better, anti-fragility) need to be key attributes of the product and the process. Monitoring and diagnostic tools are typically involved with this. More mature organizations will purposely inject faults into their production environment to practice detection, diagnosis, and remediation processes. Practice makes perfect.

LEVELS OF DEVOPS MATURITY

PATH TO MATURITY

As mentioned, DevOps is the collaboration of one or more individuals on a project to optimize:

1. Environment provisioning
2. Environment configuration
3. Application build(s)
4. Application deployment(s)

Most teams that evolve DevOps over time do so in a very predictable manner. Each of the four steps listed above is likely to be repeated many times over, but at different rates. For example, building an application happens much more frequently than the other actions, so therefore it's logical to automate it first. The typical order in which things are automated is as follows:

1. Application build(s)
2. Application deployment(s)
3. Environment configuration
4. Environment provisioning

LEVEL 0 - ARCHAIC

The most basic level, and therefore the least mature, is for these steps to be performed manually. The following is an example of the most basic level of DevOps.

A team has built an application and it needs to be deployed to a new environment. A server administrator manually provisions the servers needed for the new environment. Once that has been completed, he will configure the servers. A developer will build the application (and possibly package it up). The developer will then deploy the application, including databases, services, and whatever else is needed. Once all of this is done, one or more people will verify the application has been deployed correctly.

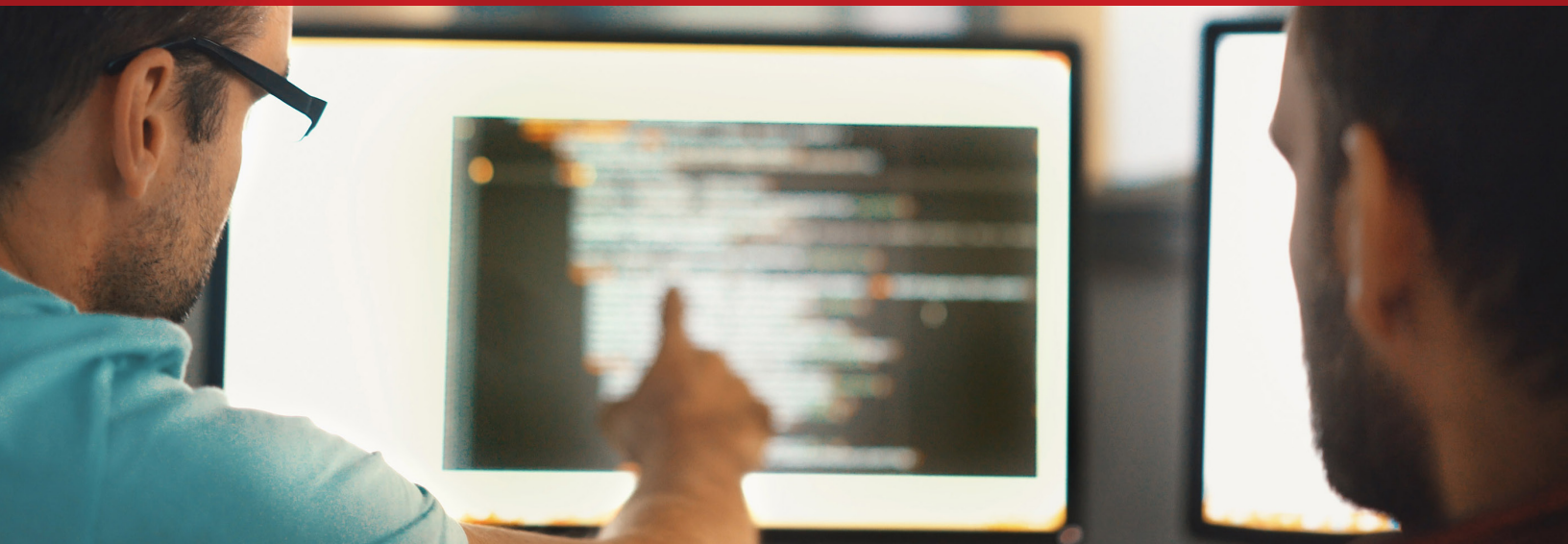
If this was going to be a one-time process, there would be nothing wrong with this approach. But when was the last time a single environment was set up for an application that was deployed only once and never deployed again?

LEVEL 1 - TRADITIONAL

To fully realize a Level 1 maturity, the automated build system should have the following characteristics:

- Run on a non-developer machine (preferably a server)
- Build on a trigger (i.e., check in), schedule, or both
- Execute unit tests
- Fail builds for multiple reasons (not just a compiler error)
- Notify team of failed builds
- Ability to block check-ins under certain conditions
- Package the application to simplify the deployment process

Leveraging the above ensures if source code is checked in that does not compile, the team is notified sooner rather than later so that the issue can be resolved (preferably by the person who checked in the breaking change). It also ensures that even if the code does compile, that the team is notified if the updated source code causes a unit test to fail. When automated builds are configured to run on a trigger or a frequent schedule, it is often referred to as continuous integration (CI) or continuous build (CB).



LEVEL 2 - MAINSTREAM

Manual deployments are rife with human error. Files could be copied to the wrong directory, or even worse, to the directory of a different environment. Database scripts may not be executed in the correct order. Services may be shut down for deployments and not restarted when the deployment is done. The list goes on and on.

Automating deployments is usually the next area where a DevOps team matures. The team has seen the benefits of automated builds and realizes that there will be many benefits to automated deployments, including:

- All steps are performed, and in the correct order, every time
- Long-term time savings (which translates into cost savings)
- Once configured, no special knowledge of the system is needed in order to perform the deployment

Not only are manual deployments fraught with human error, but they are also very time consuming. Depending on the complexity of the system, automating a deployment can be an investment, but it is an investment that will be paid back many times over the course of the project.

Configuring automated deployments to run on a trigger or a frequent schedule is often referred to as continuous deployment (CD). There are at least two major benefits of CD:

1. Developers can test functionality on a remote environment (there are many reasons why code may work locally, but not in other environments).
2. Non-developers can get a look at recent changes without having to look over the developer's shoulder.

In addition, this phase is usually where teams introduce Infrastructure-as-Code.

LEVEL 3 - MODERN

This level of maturity introduces the concept of immutable continuous delivery. This means you can blow away any environment, at any time. Your automated deployments will use containers, Infrastructure-as-Code, or some combination of both to ensure the proper provisioning and configuration of the environment.

Depending on the needs of the project, many teams will not mature past Level 2. There may not be a need for it. If there are a small, finite, number of environments, there may not be an advantage to going any further. But, if there will be *N* number of environments, then automating the build-up and tear-down of environments becomes appealing. Common reasons for this are:

- The application is installed on premises for each production installation
- The development team may have a need for a new environment for each feature branch
- The testing team may have a need to bring up and tear down their own environments

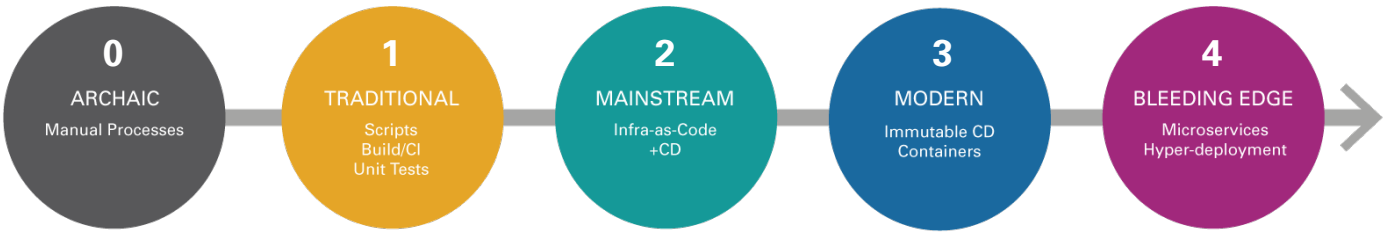
Whatever the reason, if there will be several environments then automating the configuration of the environment will be a major time saver. Activities that may need to be automated include:

- Installing/configuring software/services needed by the application
- These are typically items that only need to be done once every per environment (e.g. installing IIS)
- Initial creation of application assets, like databases
- Network configurations, like DNS entries

LEVEL 4 - BLEEDING EDGE

The final level in automation is using microservices to further increase the deployment capabilities of your immutable environments. The line between Level 3 and Level 4 are easily blurred. However, Level 4 generally refers to teams doing hundreds or even thousands of deployments to production per day.

Very few companies will achieve this level of automation. In fact, this level of automation is not necessary for most. These features require a considerable amount of effort to set up, process, and implement. Only applications that require hyper-deployment to production should be considered as candidates. This is often only the case with very large commercial, customer-facing applications.



BEST PRACTICES

The set of DevOps best practices continues to evolve and grow. Here are some of the best practices we have established over many client engagements.

1. Automate your build process
2. Automate your deployment process
3. Execute builds via trigger or a frequent schedule (at least once per hour)
4. Execute unit tests during builds and fail if any of the tests fail
5. Support multiple environments
   - Number of environments depends on the needs of the application
   - Development
   - Test/QA
   - Stage
   - Production
6. Only deploy a build if it has already been deployed to a lower environment (for example, a build should not be deployed to test without being deployed to development first)
7. Builds beyond the initial development environment should be packaged, and that same package should be used to deploy to the higher environments
   - Do not rebuild the package for each environment
   - Per-environment configurations are expected to be made as the packages are deployed
8. Ensure your team is both autonomous and aligns with your enterprise
9. Implement usage monitoring, telemetry collection, and gain feedback from stakeholders to integrate and refine back into your product backlog
10. Manage your technical debt
11. Conduct code reviews
12. Document your code
13. Use enterprise package management
14. Have a production-first mindset
15. Develop infrastructure-as-code practices
16. Use the cloud to your benefit (testing in production, cloud test, automatic scaling, enterprise dev/test labs, containers, microservices)

ENVIRONMENT GUIDELINES

A common question asked is, "How do I set up my environments, and how many do I need?" The old paradigm was Dev > Int > Test/QA > Stage > Prod. There were often five separate environments. Over the years, Dev has gradually fallen to local machines and away from full server farms.

Four server farms are still common: Dev/Int > Test/QA > Stage > Prod. Some organizations have a hard time letting go of old principles. When frameworks were different, there were often complex integrations that needed to be tested in a dev capacity working together before they could move to the Test/QA environment. With today's frameworks, that's not the case.

Read the diagram below starting at the bottom left. When developing web applications for the cloud, it's easy to have a local dev machine or virtual machine (VM) on your computer. Azure Enterprise Dev/Test Labs offer attractive features for those multiple-server dev instance requirements. Create templates and use quotas for automatic shutdowns.

When the developer commits changes to the source system, the automation begins across the gray arrow. The Build Agent in Visual Studio Team Services executes the build script tasks and runs the automated unit tests.

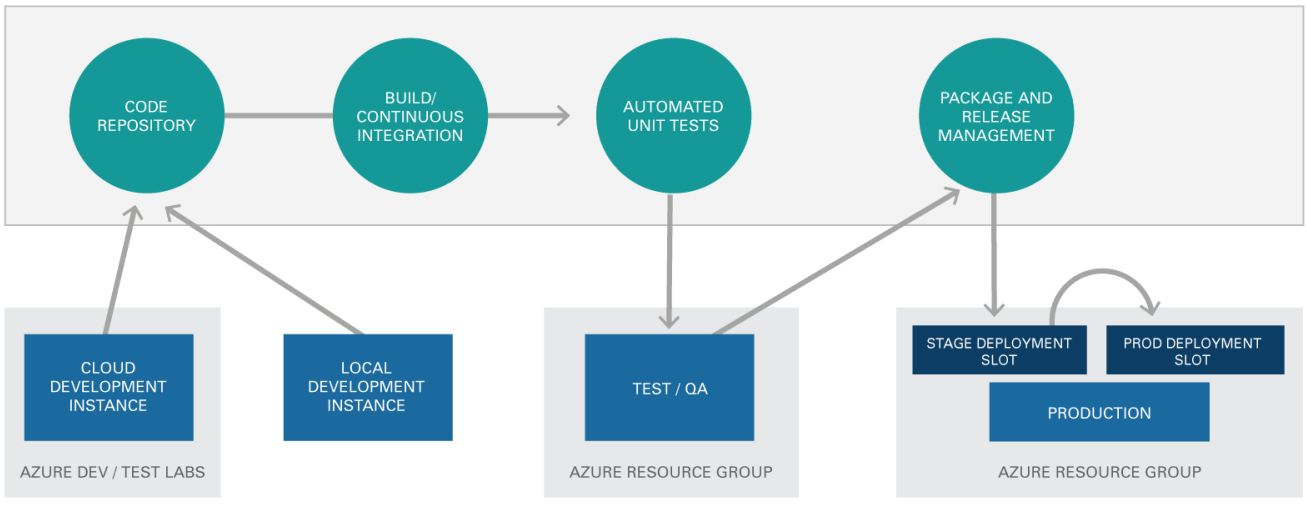
Upon successful completion of the tests, the package is deployed to the Test/QA system.

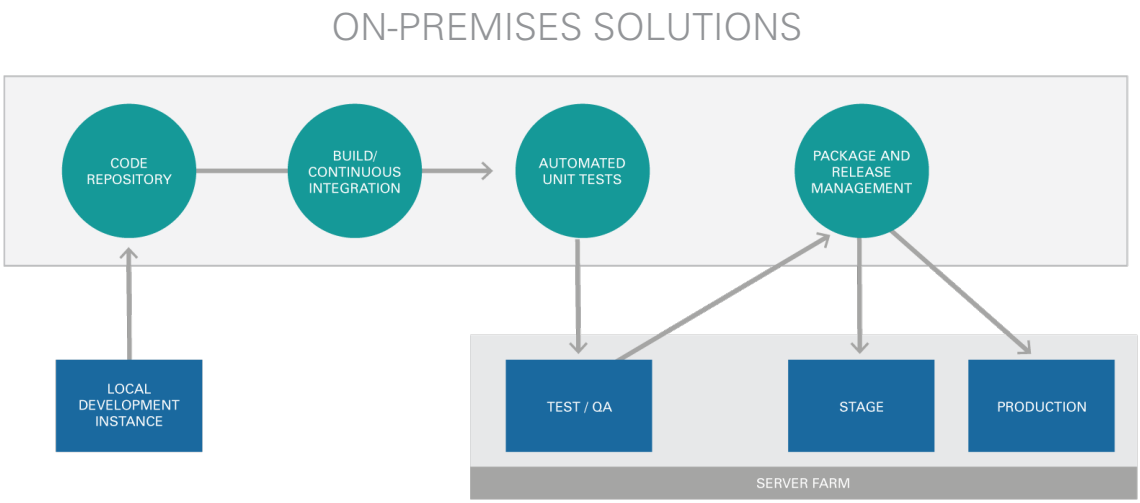
At this point, the Test/QA environment is the first real environment needed. You will have a single Azure App Service instance and any storage, DB, or other services. You can then do human testing, quality assurance, and UAT in this environment. We recommend using Azure Resource Groups to segment Test/QA from Prod. This separation provides security and role-based access permissions.

When testing is complete, you want to deploy to Stage. In the Azure cloud, this is called a Deployment Slot. Azure App Service provides up to five deployment slots per deployed web app. You can deploy any number of different versions of code to these slots. The slots are swappable at any time. They stay "warm" and can literally be changed over in seconds.

But the real power is with a feature called Testing In Production. This feature enables you to set a percentage of traffic to direct to Stage. You could direct, for instance, 5% of your traffic to the Stage slot, run your final verifications, then once you're completely satisfied, swap the entire Stage slot to Production (and reduce the TIP percentage back to zero). Of course, if there is a problem at any point in this process, you have your live production slot that you can you can revert back to.

CLOUD SOLUTIONS





These features are extremely powerful and enable you to:

- Reduce the number of environments – only two are needed in this model
- Automate build, integration, and testing tasks
- Implement more robust testing processes
- Better control versioning releases and package management

The flow of the on-premises solutions is very much the same as the cloud. The main differences are:

- On-premises Dev environments cannot use Azure Dev/Test Labs. This may require an additional server environment for developers.
- Deployment slots are not available. This means a separate Stage environment is required.

As you can see, that's three environments minimum, and in some cases four, for an on-premises solution, which will also result in a cost difference. We highly recommend a cloud solution.

Please note that these are only recommendations. They do not represent every situation for every organization. There will always be exceptions.

INFRASTRUCTURE-AS-CODE

Azure Resource Manager (ARM) Templates provide capabilities called Infrastructure-as-Code. This means using code files to deploy resources to Azure. This can be almost any type of Azure Resource: Virtual Machine, Azure SQL instance, Storage Account, Search Service, Redis Cache, Traffic Manager, etc.

ARM Templates, which are JSON, are an extremely valuable component of the DevOps journey. They allow you to build and configure your cloud infrastructure in a repeatable fashion. You can build the Dev environment, then copy/paste to start the QA environment, then to the production environment, and so on. This allows us to have different ARM Template versions to test unique environment configurations. They also enable rapid environment deployment for disaster recovery or system restore scenarios.

ARM Templates reduce the burden of provisioning and configuration of Azure resources on IT staff. Developers can now manage their own environment configurations completely within their own world – in code. Governance can (and should) be applied using ARM Templates. Be sure to include them as part of your managed code projects in TFS/VSTS.

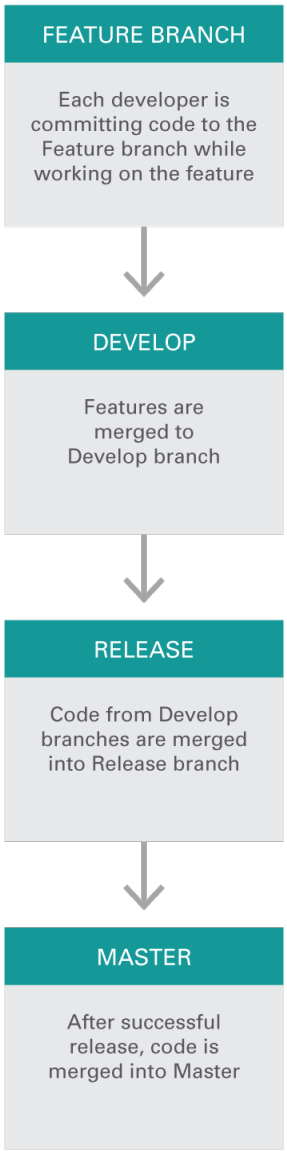
Creating ARM Templates is simple. Microsoft provides a gallery of sample templates. Often, it is easiest to start from a sample and customize from there. Another option is to build out an environment in Azure manually and have Azure generate the ARM template to use as a starting point. Starting from a blank template is also an option. Visual Studio provides a robust JSON editor to work with the template. Once finished, deployment is easy with Azure PowerShell or the CLI.

SOURCE CONTROL

The first step in automating a release pipeline is effective source control management. We use both TFSVC and Git. Your source code is always the first input into the pipeline (see diagrams above). The build agents then compile that code into artifacts that are used by the rest of the process.

Most developers have built some application at some point in their career without source control. You store the files locally on your hard drive. When you build your code, it creates the appropriate files in the file system on your computer. You then copy and paste those files onto a web server. Remember how much time it took to manage that process? Source control is the first fundamental step in automating your DevOps pipeline.

Using VSTS for source control is very easy. Microsoft is now the largest contributor to Git and have more published Git repositories than anyone else. We have been using Git for more than two years. Here's a sample of the flow we recommend:



CONTINUOUS BUILD/INTEGRATION

Continuous build (CB), or synonymously, continuous integration (CI), is a core DevOps practice that we recommend. It can be defined as the process by which code is compiled every time it is checked-in or committed. Rather than a developer initiating manual builds, the system does it automatically. The goal is to provide rapid feedback so that if a defect is introduced into the code base, it can be identified and corrected as soon as possible.

CI solves issues that many teams wrestle with on a daily basis, such as: delivery delays, non-working/low quality code, incomplete solutions, and rework. Each of these issues introduces cost, both in terms of resources to fix the issue and in time. Time-to-market is often a key business driver that enables highly effective teams. CI also helps increase quality. You know the latest build will work, which helps reduce stress and uncertainty in your teams.

Visual Studio Team Services (VSTS/TFS) has a rich feature set for implementing CI. It provides hosted build agents to start building your projects immediately, as well as a flexible build system that allows you to install your own agents. There are agents for Windows, OS X, and Linux. The out-of-the-box options cover your most commonly used tasks: Android, iOS, Maven, Jenkins, Gulp, Grunt, Gradle, and more.

The first step in a CI implementation is creating the build definition(s). Each application will likely only need one or two build definitions. It is common to have a non-optimized build. This can include compiled code, with debug information to help aid in troubleshooting, as well as un-minified resources (e.g. JavaScript and CSS files). This build typically results in a direct deployment to the development environment.

The second build definition is commonly used to produce a deployment package optimized for production. This can include compiled code (with optimizations), as well as JavaScript and CSS resources that have been bundled and minified to provide the end-user with a better browsing experience. This build does not result in a deployment being performed, but rather the creation of a package which includes all of the assets of the build needed for a deployment and makes those assets available to be deployed by another process.

The next step is to add your various build steps and configure each one. This could be a NuGet restore, Azure PowerShell script, MSBuild, or a Visual Studio Test (see below). Once all the tasks are configured and ordered, you will configure the rest of the build options for repositories, variables, triggers, and retention.

Once your definitions are built, you can manage access to your builds by defining groups and granting permissions for viewing, editing, creating, and queuing builds. You can also manage and share build resources across projects. In addition, you can control access to build resources (pools and queues) and audit changes to your builds.

Finally, the VSTS CI tools provide diagnostics, automated alerts, historical changes, and output logs to troubleshoot issues and audit the changes of your build definitions over time. This enterprise-ready toolset is essential to any DevOps solution.

CONTINUOUS TEST

Continuous test is exactly as it sounds: each time code is compiled, the system will run automated unit tests. This is an important step in the DevOps process. It enables you to find problems in code earlier and review the results immediately to begin resolving any errors. Microsoft supports a number of testing platforms and frameworks: NUnit, XUnit, Selenium, Java, Maven, Node.JS, Gradle, and MSTest. These tools work for applications hosted both in the cloud and on-premises.

Continuous test is quite easy to get started and set up. First, check-in your solution – including your test projects – to Visual Studio Team Services. Next, create a build definition that includes a task for Visual Studio Test. Once configured, tests will execute every time a build is initiated. You can go to the Tests tab to see the results summary. Test results between the current build and the last one can also be compared. Here you'll find changes in new, failed, and passed tests, how long these tests took to run, how long these tests have been failing, and more. You can organize the test results and open bugs directly for failed tests.

Microsoft provides a number of testing tools and services that include performance testing, exploratory testing, load testing, acceptance testing, and automation testing. You might not enable all of these continuously in a DevOps scenario. In fact, you are usually only going to automate unit tests. However, it is important to know that each of these exists as part of the testing capabilities in the Visual Studio platform.

CONTINUOUS DEPLOYMENT

Continuous deployment (CD) happens after a continuous integration (CI) has successfully executed (i.e., no build errors and no failed tests). CI and CD typically happen within the same process. CD is not right for everyone or every project.

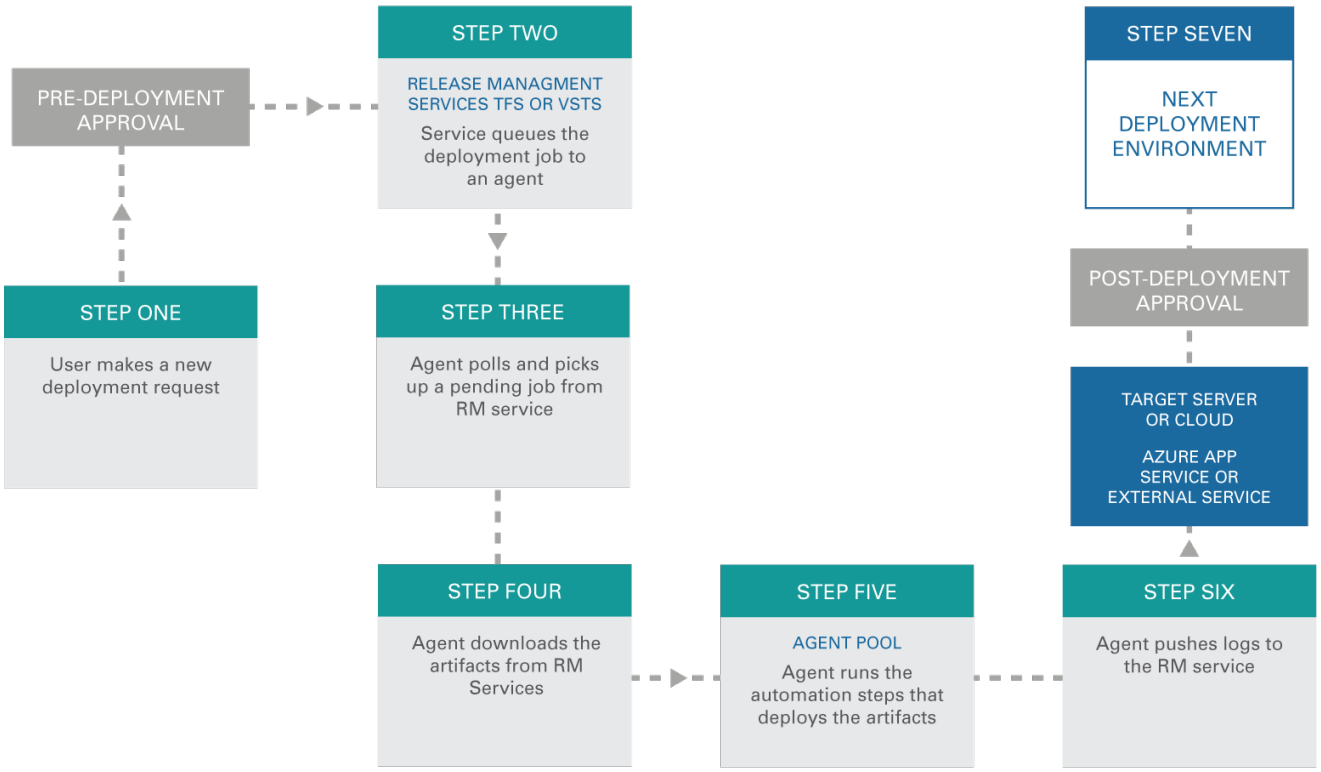
CD relies heavily on automation of all tasks and steps in a delivery pipeline. The initial tasks of setting up the automation can be daunting, but it will save a lot of headaches and effort going forward. It is highly recommended that developers not consider their task(s) to be complete until they have verified the functionality in another environment (i.e., not their local environment). There are any number of reasons why the source code will execute as expected locally, but fail to execute as expected in another environment. Requiring the team to validate in another environment ensures that all resources needed are persisted in source control and that nothing has been hard-coded to a local machine.

The environment used for CD should be considered unstable. It is not recommended that anyone outside of the development team use this environment. There are two primary reasons for this. First, a build can be triggered at any time, and the behavior of the environment can be unpredictable at that time. Second, even if a build successfully compiles and passes all tests, there still exists a possibility that it has a major issue and the site could be unusable until a fix is committed to source control.

PACKAGING AND RELEASE MANAGEMENT

Many software projects today rely on package managers (i.e. NuGet, npm) to successfully build. Others rely on packages as an output of the build or release process to share components across teams or make available as an open source component. Visual Studio Team Services provides a great set of features for effectively managing these packages.

- Privately hosted packages – NuGet is the first supported package type, but the service is built to support any programming language or platform, and can contain artifacts from your own build server, NuGet.org, npmjs.com, GitHub, and more
- Consistent access to components needed by your build
- Enterprise authentication to manage permissions on who is allowed access to those components
- Seamless integration with build and release management tools
- A friction-free way to bring in an open source component to your enterprise
- Discovery and search of packages across the enterprise



Release management helps you automate the deployment and testing of your software in multiple environments. You can either fully automate the delivery of your software all the way to production, or set up semi-automated processes with approvals and on-demand deployments.

Release management runs the following steps as part of every deployment:

1. Pre-deployment approval: When a new deployment request is triggered, release management checks whether a pre-deployment approval is required before deploying a release to an environment. If it is required, it sends out email notifications to the appropriate approvers.
2. Queue deployment job: Release management schedules the deployment job on an available automation agent. An agent is a piece of software that is capable of running tasks in the deployment.
3. Agent selection: An automation agent picks up the job. The agents for release management are exactly the same as those that run your builds in Team Services and Team Foundation Server. A release definition can contain settings to select an appropriate agent at runtime.
4. Download artifacts: The agent downloads all the artifacts specified in that release, provided you have not opted to skip the download. The agent currently understands two types of artifacts: Team Build artifacts and Jenkins artifacts.
5. Run the deployment tasks: The agent then runs all the tasks in the deployment job to deploy the app to the target servers for an environment.
6. Generate progress logs: The agent creates detailed logs for each step while running the deployment, and pushes these logs back to Team Services or Team Foundation Server.
7. Post-deployment approval: When deployment to an environment is complete, release management checks to see if there is a post-deployment approval required for that environment. If no approval is required, or upon completion of a required approval, release management proceeds to trigger deployment to the next environment.

CLIENT SUCCESS STORY

LEADING CORPORATE INSURANCE BROKER

Corporate insurance is a sophisticated and challenging business. Not only does this leading corporate insurance broker help its clients find the best insurance plans, but it also advises them on selecting the optimal business development strategies and set risk-management objectives uniquely suited to their long-term objectives.

To streamline this process, the organization asked us to build a custom data-gathering and reporting tool to be used by account managers when developing client strategies.

We used Microsoft Visual Studio Team Services to keep the software development process agile and fast-paced. From requirements management to source control, continuous integration, and deployment, Visual Studio Team Services supported the distributed development team and allowed the project to be delivered on time and with consistent quality.

WHY PERFICIENT

Recognized by Microsoft as a gold-certified partner and one of its premier national solution providers, we have built a successful business on Microsoft's cloud platforms, migrating more than 3.5 million users to Microsoft's Office 365 service. We continue our investments in the cloud by focusing on Microsoft Azure, Skype for Business Online, Yammer, SharePoint Online, Xamarin, and Sitecore.

We were honored to be named Microsoft's East Region NSP Partner of the Year and Cloud Partner of the Year, Central Region NSP Partner of the Year, and West Region NSP Partner of the Year in 2016. We will show you how to use Microsoft platforms, products, and best practices to connect employees to key communications and data, and especially to one another.

- Migrated 3.5 million users to the Microsoft Cloud
- Gold certified in all MPN Cloud Competencies
- 3,300+ engagements
- 700+ clients
- 350+ Microsoft-focused consultants
- 40+ Azure certified consultants

Perficient's Cloud Platform Practice encompasses our work across all four Microsoft Azure platform pillars – Cloud Development, Cloud Infrastructure, Cloud Security, and Cloud Data. Contact us to find out how our certified Azure experts can help you transform your digital cloud.

AUTHORS

JOE CRABTREE
Cloud Platform Practice Director

WIL BLOODWORTH
Cloud Dev Platform Practice Architect

BRIAN BALL
Cloud Dev Platform Practice Architect

ABOUT PERFICIENT

Perficient is the leading digital transformation consulting firm serving Global 2000® and enterprise customers throughout North America. With unparalleled information technology, management consulting and creative capabilities, Perficient and its Perficient Digital agency deliver vision, execution and value with outstanding digital experience, business optimization and industry solutions.

PERFICIENT.COM/BLOGS
TWITTER.COM/PERFICIENT
FACEBOOK.COM/PERFICIENT
PERFICIENT.COM/GUIDES