The Release Pipeline Model

Applied to managing Windows Server and the Microsoft Cloud

Publication Date: April 2016

Authors: Michael Greene (Microsoft), Steven Murawski (Chef Software, Inc)

Contributors: Editor: Glenn Minch (Aquent LLC); Technical Reviews: Levi Geinert, Jason Rutherford (Model Technology Solutions); Matthew Wetmore (Microsoft); Jim Britt (Microsoft)

Summary: There are benefits to be gained when patterns and practices from developer techniques are applied to operations. Notably, a fully automated solution where infrastructure is managed as code and all changes are automatically validated before reaching production. This is a process shift that is recognized among industry innovators. For organizations already leveraging these processes, it should be clear how to leverage Microsoft platforms. For organizations that are new to the topic, it should be clear how to bring this process to your environment and what it means to your organizational culture. This document explains the components of a Release Pipeline for configuration as code, the value to operations, and solutions that are used when designing a new Release Pipeline architecture.

Table of Contents

Foreword

It is time to stop lying to ourselves. What we've been doing is not working. Our infrastructure is a mess. Our application deployments are a nightmare. We spend our nights and weekends fighting fires of our own making. We put more effort into shifting blame than improving our environments. Documentation lags because of last minute changes or ever-shifting deployment targets. Convoluted change management leaves critical, important tasks waiting for urgent, less important work. We need to find a better way, but we are not alone.

Over the past few years, research has been ongoing into the state of IT and the impact of these wacky "DevOps" ideas. Well, the data is in and it doesn't look good for traditional IT processes. Per the "State of DevOps Report" and the "DevOps and the Bottom Line" studies and data analysis performed by Dr. Nicole Fosgren, IT performance is a key factor in an organization's profitability, productivity, and market share. IT can be a competitive advantage, but it requires a mind shift and process shift from the traditional way of doing things.

And the businesses are noticing. They are finding cheaper and faster ways to access the services we once held the sole dominion over. Once the shackles of traditional IT management have been shed, business is being unleashed to try new things at a faster pace than before and at less cost.

All is not lost for the enterprise IT admin, though. To steal a saying from the United States Marine Corps and that wonderful movie Heartbreak Ridge, we need to "improvise, adapt, and overcome." We are in a fight for our place in the IT industry. Cloud services offer alternatives to in-house offerings everywhere from email and productivity tools to bare machine images, to raw services for developers. These services are attractive to businesses because there is little impediment to getting up and running. We, as IT professionals, need to compete with these services (or use them where they make sense), and offer business a value for the cost of our time and the infrastructure we've been entrusted with. The tools we equip ourselves with for this challenge are the four major components of DevOps: Culture, Automation, Measurement, and Sharing. In the following whitepaper, you will be challenged to look at the delivery of IT infrastructure in a new way. Millions of managed servers in environments that are enjoying unprecedented levels of stability can't be wrong (60 times greater success of changes and 168 times better mean time to recover—MTTR—per the 2015 State of DevOps Report). Organizations adopting workflows like those outlined here enjoy shorter lead time (200 times shorter) for deployments (code committed to deployed in production). Another benefit to these workflows is more frequent deployments (30 times greater) than lower performing traditional IT. It doesn't matter where the software comes from; these results are possible with packaged software, consultant written software, or software developed in-house. While these practices and tools are still evolving on the Microsoft stack, they have been proven in the real world and can deliver tangible value for you in your environments.

Steve Murawski, Cloud and Datacenter Management MVP
Principal Engineer, Chef Software

Motivation

IT operations have traditionally managed Windows servers with many of the same practices used to manage Windows desktops, and their success has been (and continues to be) measured by the strictest of expectations for service continuity. In the event of a service outage, functionality must be restored no matter what the cost.

In most traditional, on-premises environments, incremental changes to the baseline configuration are delivered using a more or less manual approach to integration and release management. Even when there is automation of the process, a person usually has to manually perform certain key actions to deploy changes, such as installing an update or configuring system settings. The administrators who build and configure servers and desktops require administrative user privileges to perform their tasks, often granted on a permanent basis. Testing, when there is formalized testing, is generally performed manually. This approach is error-prone, difficult to troubleshoot, and it exposes the organization to security risks.

None of this is news; these are problems that IT operations has struggled with for decades. While it is possible to manage the situation without automation, there is an insidious hidden cost to "manageable manual tasks" in small to mid-sized environments. While manual changes and testing, with a smattering of automation, can allow IT operations to "keep up" with change, it ties up precious and expensive person hours doing work that computers should do. This either slows the rate of innovation or stifles it completely.

The modern industry shift towards cloud-based and hybrid IT architectures is driving a parallel evolution in development and management frameworks. The forces of change in the industry have resulted in the development of new IT operations processes that are more precise, automated, and far more repeatable and scalable than traditional approaches, without rejecting the accomplishments from years of prior development in operational practices.

This duality of goals, addressing the problems we have always had and addressing the new challenges we face with the rising popularity of cloud services, satisfies the observed criteria for a successful paradigm shift:

> *First, the new candidate must seem to resolve some outstanding and generally recognized problem that can be met in no other way. Second, the new paradigm must promise to preserve a relatively large part of the concrete problem-solving ability that has accrued to science through its predecessors. (Kuhn n.d.)*

There are industry terms that have become popularized to describe the modernized approach. The cultural shift is often described as *DevOps*, referring to developers and operations working together to share responsibility, communicate openly, improve quality of life, and drive the delivery of value to the business. DevOps is a cultural shift for people that occurs with the support of both tools and practices. Another common term is *agile,* which refers to practices that enable stakeholders to organize and deliver on projects in a more effective cycle that is intended to reduce time to value.

These industry shifts have brought attention to the practice of using a concept borrowed from software development, the *release pipeline*, applied to the management of core infrastructure technologies.

This paper is all about implementing a release pipeline for IT operations, as seen from the perspective of an IT professional managing Windows servers and the Microsoft Cloud.

Implementing a release pipeline is a process that is sometimes euphemistically referred to as "a journey." Others would characterize it as "a learning experience." No matter how you think of it, two things are certain: a) it won't necessarily be easy, and b) the payoff is worth the trouble.

This document is for IT Operations Professionals

This document is intended to address IT Operations professionals with a skillset and career developed in managing environments that have been predominantly Microsoft technologies. The intention is to bootstrap someone with this background in service management in to future-facing IT operational practices for managing core infrastructure. Throughout the document, many references will be made to Microsoft products and solutions. This is simply to serve as references for the reader who is mostly familiar with Microsoft branding. There are many products and tools available both from software vendors and the community that can and will be components of your release pipeline.

If you are grappling with these concepts for the first time, you might find yourself thinking "these are mostly developer concepts". You would be correct. The tools and solutions are largely borrowed from practices associated with software development. The patterns and practices expect a willingness to learn new skills including advanced scripting and automation. By applying these practices to managing infrastructure, we align with an industry shift and become ready for the new career opportunities in IT.

References

For a list of books that provide original, industry-changing thought on the subject of DevOps, see the DevOps Reading List[1].

The following list of books in particular have influenced the authors of this whitepaper and deserve special mention:

- *The Goal*. Eliyahu M. Goldratt; North River Press, 2014.
- *The Phoenix Project*. Gene Kim, Kevin Behr, and George Spafford; IT Revolution Press, 2013.
- *Continuous Delivery: Anatomy of the Deployment Pipeline*. Jez Humble and David Farley; Addison-Wesley Professional, 2007
- *Leading The Transformation*. Gary Gruver, Tommy Mouser, and Gene Kim; IT Revolution Press, 2015.
- *Working Effectively with Legacy Code*. Michael Feathers; Prentice Hall, 2004.

Readers of this document might also want to reference the following whitepaper which addresses many of the same challenges from a software development point of view: Building a Release Pipeline with Team Foundation Server 2012[2].

The release pipeline model

The term “release pipeline” is often used in Microsoft documentation when referring to patterns and practices for developers. Pipeline refers to moving a project through stages of work and release refers to publishing software as a final product. Other documents use similar terms such as “deployment pipeline” or “build pipeline” to describe the same model.

In the release pipeline model, deployment and configuration details are stored alongside the developer's code, or alongside installation files for packaged software products.

Storing the configuration details together with applications code helps to ensure that there is alignment between the version of software that is currently released and the configuration of the server or service that is hosting the software.

Test scripts, test parameters, and the scripts used by automation to deploy the solution and apply configurations are also stored in the repository. This is done to facilitate automation of the pipeline.

When changes are needed, either a new version of the service is released or an updated file that describes incremental changes to already deployed servers is pushed to the hosting environment. Storing server configuration within deployment scripts or in a settings file (often some combination of both) is also known as *configuration as code*.

Configuring infrastructure "as code"

The introduction of hosted services has prompted the development of a management framework where resource configuration is simplified by representing it as a collection of declared properties enforced through idempotent[a] scripting. Collectively, these properties define a *configuration*, which you can think of as a type of contract between the developers and a hosting environment that specifies exactly how a server should be configured. Automation platforms read the configuration and then implement the required settings in the host environment. Automation orchestrates the provisioning and configuration process after it has been initiated in the host cloud. The term *configuration as code* refers to both the configuration properties and the automation that implements the configuration in the target environment.

In fact, the model is already expanding to include declaring a service, not just a server, through the Azure Resource Manager (ARM) REST API[3]. The ARM API is available in Microsoft Azure (public cloud) and Microsoft Azure Stack[4] (private cloud), and it provides a high level of consistency across public, private, and hybrid environments. Consistency permits deployments to occur in either private datacenters or publicly-hosted services with few to no changes to the configuration code.

Some examples of configuration as code to manage server platforms include PowerShell Desired State Configuration (DSC), Chef, Puppet, and CFEngine. PowerShell DSC provides a platform for managing Windows Server using declarative models that any solution can leverage. Both Chef and Puppet offer integration with PowerShell DSC to manage Windows Server, and Azure Resource Manager includes extensions for declarative platforms such as PowerShell DSC for server node configuration.

---

[a] This means that no matter how many times you apply a given configuration script to an environment, the environment will not be changed after the first application of the configuration. For more information about idempotence, see Wikipedia contributors, "Idempotence," *Wikipedia, The Free Encyclopedia,* https://en.wikipedia.org/w/index.php?title=Idempotence&oldid=710074029.

PowerShell Desired State Configuration to create and manage a simple file:

```
Configuration example
{
    File textFile
    {
        Ensure = 'Present'
        DestinationPath = 'c:\examplefile.txt'
        Contents = 'Hello DSC'
    }
}
```

And using the same PowerShell DSC resource inside a Chef Cookbook:

```
dsc_resource 'example' do
   resource :file
   property :ensure, 'Present'
   property :destinationpath, 'c:\examplefile.txt'
   property :contents, 'Hello DSC'
 end
```

For more practical examples of configuration as code, see [Configuration in a DevOps world – Windows PowerShell Desired State Configuration[5].]

Segment of an ARM template that describes a Windows Server virtual machine:

```
{ "type": "Microsoft.Compute/virtualMachines",
"name": "server01",
"location": "Central US",
"apiVersion": "2015-06-15",
"dependsOn": [
  "[concat('Microsoft.Storage/storageAccounts/', 'myservice1')]",
  "[concat('Microsoft.Network/networkInterfaces/', 'nic1')]"
],
"properties": {
  "hardwareProfile": {
    "vmSize": "Standard_D1"
  },
  "osProfile": {
    "computerName": "server01",
    "adminUsername": "admin1234",
    "adminPassword": "[parameters('Password')]" // this value is input at
deployment
  },
  "storageProfile": {
    "imageReference": {
      "publisher": "MicrosoftWindowsServer",
      "offer": "WindowsServer",
      "sku": "2012-R2-Datacenter",
      "version": "latest"
    },
    "osDisk": {
      "name": "osdisk",
      "vhd": {
          "uri": "http://myservice1.blob.core.windows.net/vhd/osdiskforwindows.vhd"
      },
      "caching": "ReadWrite",
      "createOption": "FromImage"
    }
  },
  "networkProfile": {
    "networkInterfaces": [
      {
          "id": "[resourceId('Microsoft.Network/networkInterfaces','nic1'))]"
        }
      ]
    }
  }
}
```

The concept of configuration as code follows many of the same practices presently used in traditional server deployment. Tools such as the Microsoft Deployment Toolkit[6] and PowerShell Deployment Toolkit[7] already provide a framework that is used to define the details of how a machine should be built, and the deployment scripts act as a single source of truth[8] to define how a machine should be configured. Administrators familiar with System Center Configuration Manager or System Center Virtual Machine Manager can relate to the GUI experience of defining how applications will be deployed from trusted installation files, or defining application service templates to configure servers as they are provisioned.

Configuration as code builds on these experiences by integrating best practices commonly associated with developer tools to the use of tools that IT professionals are already familiar with. Implementing configuration as code is essential to implementing a release pipeline.

Why would Operations apply the release pipeline model?

The objectives of implementing a release pipeline include the adoption of new technology concepts such as increased frequency in release cadence with high degrees of trust; promoting smaller incremental changes instead of large, monolithic changes; easing the transition to cloud service deployment models; and using tools that help operations and development perform work as one collaborative unit.

The release pipeline also addresses gaps in traditional service management operational processes that have historically been difficult to overcome. Some of the key improvements that a release pipeline brings to operational processes include improving security in the IT infrastructure; reducing recovery time to bring servers back online after an outage or migration; and establishing consistent, complete change management for server configurations.

Improve security

Only the automation platform should introduce changes to production servers in an environment managed by a release pipeline. It is imperative to avoid giving individuals long-term administrative access to the IT infrastructure. When it is necessary to grant plenipotentiary privileges, grant them only for the minimum time necessary to perform the task at hand.

Advice to restrict administrative privileges does not ignore the need to perform routine maintenance on servers that exist in long-term deployment scenarios. Tasks such as patching, password changes, and other operational requirements often require administrative privileges on the target machines. These tasks can be performed without giving an operator elevated privileges by using tools such as PowerShell Just Enough Administration (JEA).

JEA is a framework for creating "constrained" endpoints on the machine to be administered. Constrained endpoints are associated with a defined collection of allowable operations, which are executed under a local machine administrator account. Constrained endpoints enable automation platforms such as Azure Automation[9] to safely perform tasks that would ordinarily require administrative privileges. This, in combination with a release pipeline, provides ability to deploy changes to production servers and perform routine activities without granting administrative access to operators, greatly improving the security of the environment.

Reduce recovery time

In a service outage situation, servers with no state information, such as web servers, can be re-deployed quickly using the last known-good configuration files stored in version control. Servers with state information, such as database servers, benefit from an improved recovery plan because the history of changes for any node is contained by the present version of the configuration file. A recovery procedure can use this information to initiate a node rebuild activity that leads to restoring or re-synchronizing data to the last known good point. Similarly, in a migration scenario where major environmental changes are going to occur, rather than rebuilding solutions on new hardware or new virtualization platforms, a re-deployment can occur that targets a new instance of the application to the new environment rather than attempting to "lift and shift" the prior installation.

Improve change management

In this context, change management refers to evaluating, validating, and tracking the changes that are applied to production servers. In a mature IT operations practice, changes to the environment are implemented only after close consideration and agreement across stakeholders. This is true whether the organization uses a release pipeline or not, but for organizations that lack a release pipeline, testing and implementation are typically handled in a manual fashion. Manual testing is rarely able to keep up the need for testing, and even when it does, it is time consuming and ties up valuable resources that could be more effectively used elsewhere.

The drawbacks of manual testing are especially apparent when there is an outage and changes need to be made in a short period of time; it simply takes too long to manually perform a full test pass. This leads to ad hoc changes, that is, makeshift (not validated by testing) solutions that are implemented on short notice.

Any change made on an ad hoc basis breaks the rule that all changes must be validated before release. Changes that are not validated by the standard testing protocol greatly increase the risk of incompatibility with future changes. Ad hoc changes during outage situations complicate change management because the state of the environment is often never fully understood. If almost all your changes qualify as ad hoc changes, it will limit IT operations to only introduce changes during off-hours, so that there is time to recover if a problem is discovered after deployment.

In contrast to a manual approach, managing infrastructure using a release pipeline automates change management by organizing changes into small increments, and by only allowing production changes following in-depth validation by automated tests. Essential automated tests can often be run in minutes whereas it would take hours (at least) to perform the same testing manually. This greatly reduces the incidence of ad hoc changes, which reduces the problems that tend to be closely associated with ad hoc changes. Changes to the environment can be made more often, with greater confidence that nothing will break, and a complete record of all the changes is automatically maintained so that there is no doubt about the actual state of the environment.

The key components of a release pipeline

An architecture for a release pipeline environment consists of four core components: Source, Build, Test, and Release. The following sections discuss the core components and their roles in the release pipeline.

Source

Source control (also known as *revision control* or *version control*) platforms, such as Visual Studio Team Foundation Server[10] (TFS), Git[11], and GitHub[12], are recognized as tools that are used by developers. IT professionals would traditionally be more familiar with using a file share, or possibly a Microsoft SharePoint library to fulfill the need for document storage. However, a file share provides almost none of the services that a version control system provides. SharePoint, though it does have version control abilities, does not include tools to view and merge changes, resolve conflicts, or create change lists. Managing configuration as code calls for the use of a source control system. Source control is really just a system to manage changes to electronic files, deployment scripts, code, etc. The source control system provides special services to make management of the files more effective when working with multiple authors and frequent changes, and with extensibility to connect with other services.

Source control systems automate much of the process of keeping track of changes made to documents under version control. Most of the time, the person checking in the change only has to enter a summary description of what was done to the document; all the other metadata such as date, time, and author is collected automatically. Every change made to the file is tracked in detail by the system to record which lines include changes from the previous version. When operations are performed on the files (as opposed to a modification to the file), that information is also recorded by the system. The *change list* is a report of all of the specific modifications that were made to the file since the file was first added to source control.

Inevitably, there will be conflicts between changes submitted by different contributors. Mature source control platforms include conflict resolution features to help identify and resolve conflicts between changes.

In an environment where source control is used to manage configuration as code, a change to a service might begin by opening a text file in an editor such as PowerShell Integrated Scripting Environment[13] or Visual Studio Code[14], and then changing the value of a setting. When managing a server using a platform such as PowerShell Desired State Configuration[15] (DSC), if a value in the Windows registry was previously set to 0 and should now be 1, that would be an identifiable change in that file when it is saved. In the case of managing a service such as ARM deployment templates[16], the change might include modifying the number of deployed virtual machines, changing which configuration will be

applied to them, or how they are connected to a software defined load balancer. These modifications are made as changes to text in a JSON file.

Source control has a specialized vocabulary for referring to artifacts, metadata, actions, state changes, and so on. Three key terms related to source control are used frequently in this paper: commit, push, and pull.

Commit refers to accepting changed files into to a project. Many authoring tools are able to compare before and after views of the file side-by-side so that it is easy to see what has changed since the previous commit.

Push refers to uploading ("pushing") all of the changes and the metadata about changes that were committed on a local workstation to a central server or a service. This operation is also called a *check-in*. The next time you look at the files in source control you can view the commits you performed on your workstation and the related metadata including date/time, author, and comments.

Pull is a means of updating files from a remote copy. When an author would like to modify files and they are not the owner of the files, the proposed change is submitted as a *pull request*. If the pull request is accepted by the owner of the files, the changed files are merged with the primary source control repository.

What does source control mean to me as an IT Operations Professional?

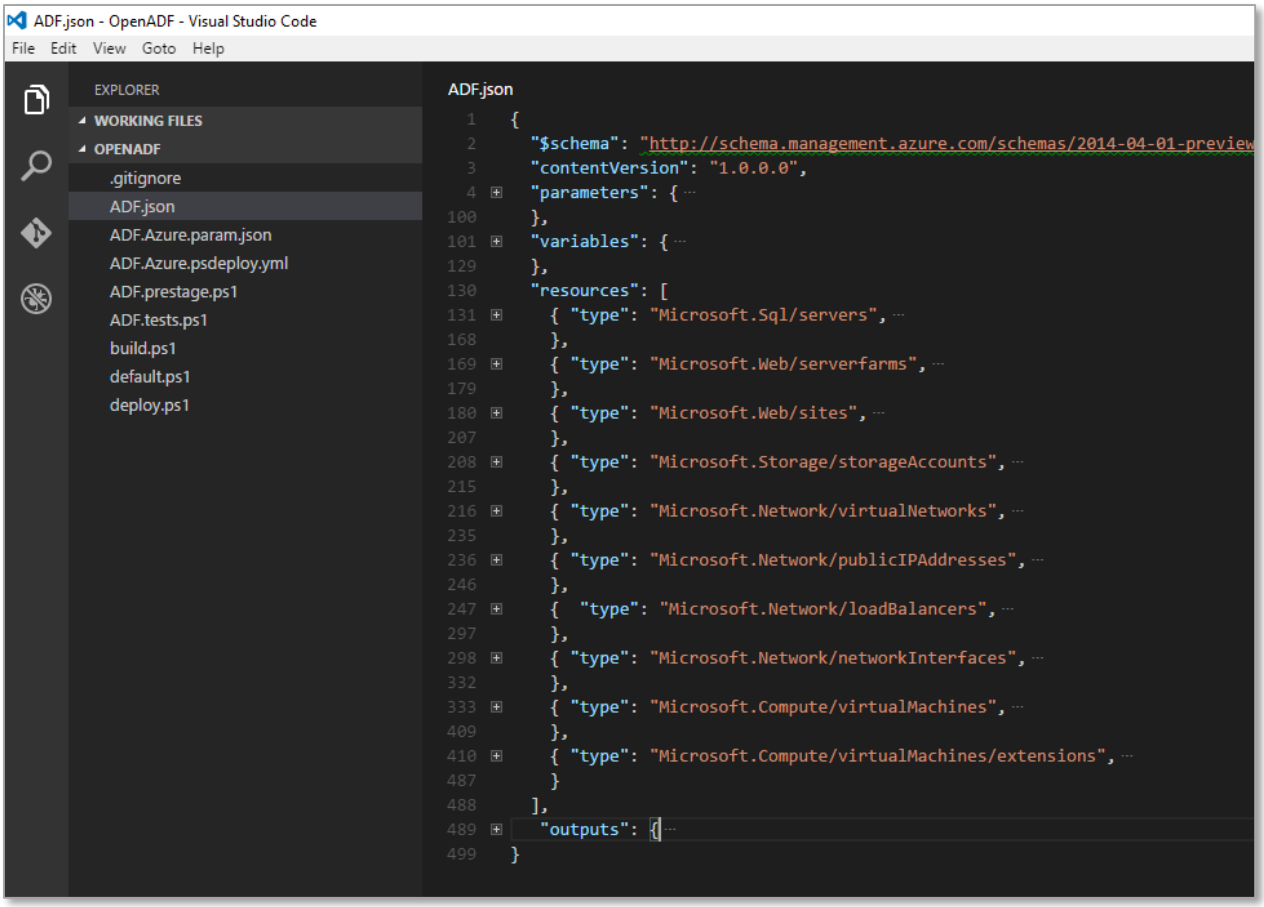
In organizations (large or small) with mature IT practices there are processes in place to identify when changes will be made, who will make them, and what work will be performed. As the work is performed, a record of changes and who made them has traditionally been kept in a change control log. The operator has to manually enter most of the information associated with the change, such as author/operator name, description of change, how the change will be implemented, degree of risk, expected down time, and so on.

When deployments are managed using configuration as code, the entire history of an artifact can be traced from inception to production deployment. All of the changes made along the way are visible. You can control who can make changes to the source files, and establish "gates" where changes are reviewed before being accepted into the baseline version. Multiple instances of the project can be maintained at the same time using branches that permit the active work to occur without disrupting the stable production-ready files. The release pipeline model integrates the source control system with the build and test automation systems. For example, pushing a file from the dev branch to the test branch is a logged source control operation that will result in that file automatically being included in the next test automation pass.

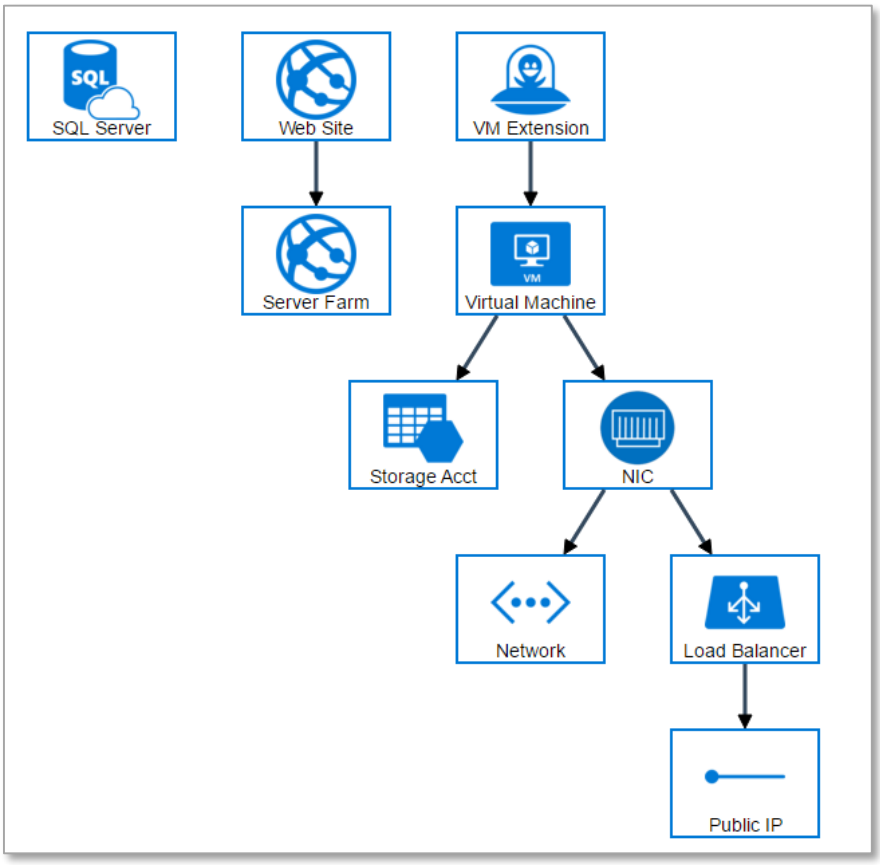
It might still be necessary for a workgroup such as a change advisory board to review changes before they go into production but we will see later how test automation can provide much more information than what was ever available before in traditional change control process.

Imagine a change control meeting that is reviewing proposed changes, and two teams have requested different values for the same group policy setting. It is up to the lead of the meeting or the attendees to notice that the values requested for the setting would be in conflict. If the conflict is overlooked, it could result in an inconsistent server configuration during the next maintenance window. When file changes are submitted as pull requests, the source control system can automatically identify when there are multiple outstanding pull requests that modify the same settings in the same files. This is normally followed by a notification to the requester that their change creates a conflict and that something must be done to resolve the issue.

Viewing an Azure Resource Manager project in Visual Studio Code



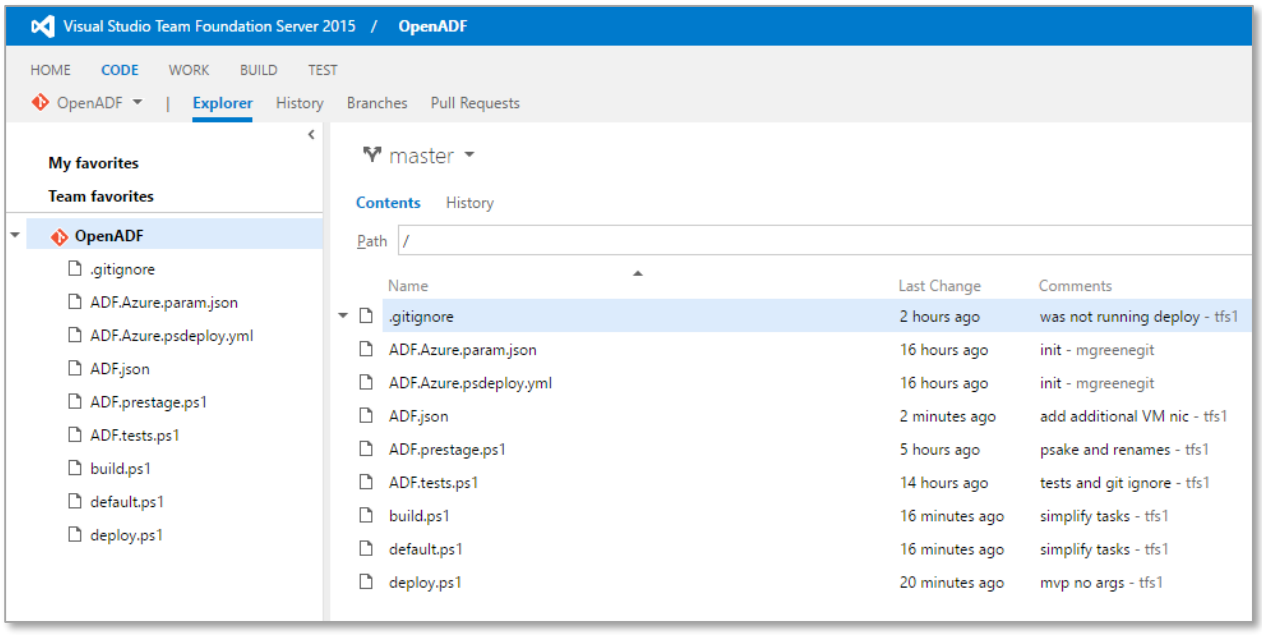
Logical visualization of a configuration as code project using ArmViz



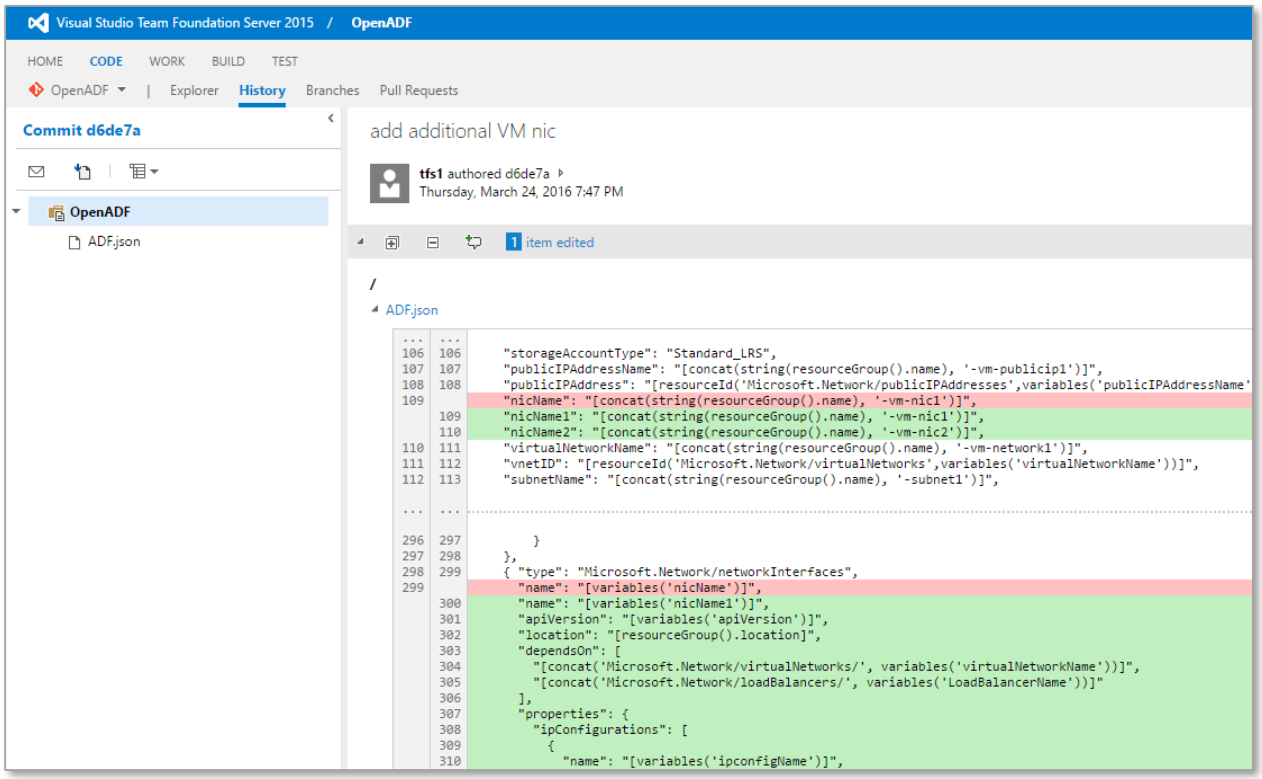
Committing a change using the Git command line to a TFS repo

```
PS C:\OpenADF> git add -A
PS C:\OpenADF> git commit -m 'add additional VM nic'
[master d6de7a3] add additional VM nic
 1 file changed, 71 insertions(+), 5 deletions(-)
PS C:\OpenADF> git push
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 599 bytes | 0 bytes/s, done.
Total 3 (delta 2), reused 0 (delta 0)
remote: Analyzing objects... (3/3) (14 ms)
remote: Storing packfile... done (98 ms)
remote: Storing index... done (112 ms)
To http://mgtfs.eastus2.cloudapp.azure.com:8080/tfs/Operations/_git/OpenADF
   844ba09..d6de7a3  master -> master
PS C:\OpenADF>
```

The project stored in source control



Observing the change history by viewing the Push to source control when adding a second network adapter to a server



Key concepts in this section

For IT operations, source control provides an elegant method of storing configuration as code and documenting changes to core infrastructure.

- Everything that goes through the release pipeline begins its journey when it is added to the source control system.
- Source control provides a mechanism for multiple teams to submit changes to the same files, and to identify and resolve conflicting changes.
- Source control establishes a centralized, durable record of the history of an artifact along with storing previous versions of the artifact itself. This can be invaluable for troubleshooting, fixing bugs, and for long-term maintenance and modifications.

When you are designing a source control environment, evaluate products such as Visual Studio Team Foundation Server, GitHub Enterprise, and Apache Subversion, or online services such as Visual Studio Online and GitHub.

Build

A *build system* is an orchestration service that is connected to the source control platform, so that actions can be triggered when files change in the repository. Source control and the build system work hand in hand. The build system will execute a file or files stored in the source control that run as scripts to perform work. There are multiple options for triggering a build activity:

Web hook refers to a web service call from the source control environment to the build service. Web hooks are used by tools that provide continuous integration with a web-based source control system such as GitHub. Examples include Travis CI[17], Visual Studio Team Foundation Server, and AppVeyor.

An *agent* polls the source control environment waiting for changes to occur. Polling is used by continuous integration systems such as Jenkins[18] and TeamCity[19].

Scheduled builds run on a regular basis. Most build services support scheduled builds.

A *manual* request initiates a build activity on demand. Most build services support manual requests.

A build system can be as simple as a platform that can run scripts or execute commands. Developers traditionally refer to the build as a process that begins when code is committed to the source control system. Software source code is compiled to binaries which are then output to a secondary location as artifacts. For example, a C# project might be compiled to produce an executable file that is intended to be run as a command-line tool.

When working with configuration as code, the build system is usually responsible for running a script that performs all the actions that must occur before a change can be released into production. This might include running tests locally on the build server or provisioning a test environment by making calls to external services. In the case of validating changes to a server, the configuration might be pushed to a temporary virtual machine. In the case of managing a service such as a deployment to Microsoft Azure or Microsoft Azure Stack, a template would be deployed to create resources such as network, compute, and storage.

The script executed by the build system is referred to as the build script. There is a community tool known as PSake[20] to organize tasks in build scripts written in PowerShell. This community-driven effort

standardizes build scripts using keywords that align with the syntax used in tools such as Make[21] and Rake[22].

```
Example of the PSake domain specific language (DSL) where key words
implement automatic tasks such as compiling artifacts, executing tests, and
resetting the build server to a clean state:

properties {
  $testMessage = 'Executed Test!'
  $compileMessage = 'Executed Compile!'
  $cleanMessage = 'Executed Clean!'
}

task default -depends Test

task Test -depends Compile, Clean {
  $testMessage
}

task Compile -depends Clean {
  $compileMessage
}

task Clean {
  $cleanMessage
}

task ? -Description "Helper to display task info" {
        Write-Documentation
}
```

What does a build system mean to me as an IT Operations Professional?

When a new server is built, it often requires software to be installed. There could be tasks that need to be completed on a storage device to make volumes ready for the application. Networking equipment might require new ports to be opened that allow incoming traffic to reach the application. Before any of this is performed in production, it is first attempted in a test environment, and the entire setup process is documented.

Unfortunately, given the velocity of change in most environments, it has become impossible to give every task the attention it deserves. In traditional IT operations, a human is expected to perform these tasks and is given limited time to complete them, in addition to a massive backlog of other tasks that might be even more visible to stakeholders. As the backlog grows, documentation is delivered after the fact if at all. This is amplified by the likelihood of configuration drift in environments with multiple administrators that might have different philosophies about how settings should be applied.

The role of the build system when managing infrastructure using configuration as code is to perform work in the environment by executing scripts. The scripts are stored in source control, as described in the previous section. The scripts can include any actions to make changes in the environment by remotely connecting to devices or services.

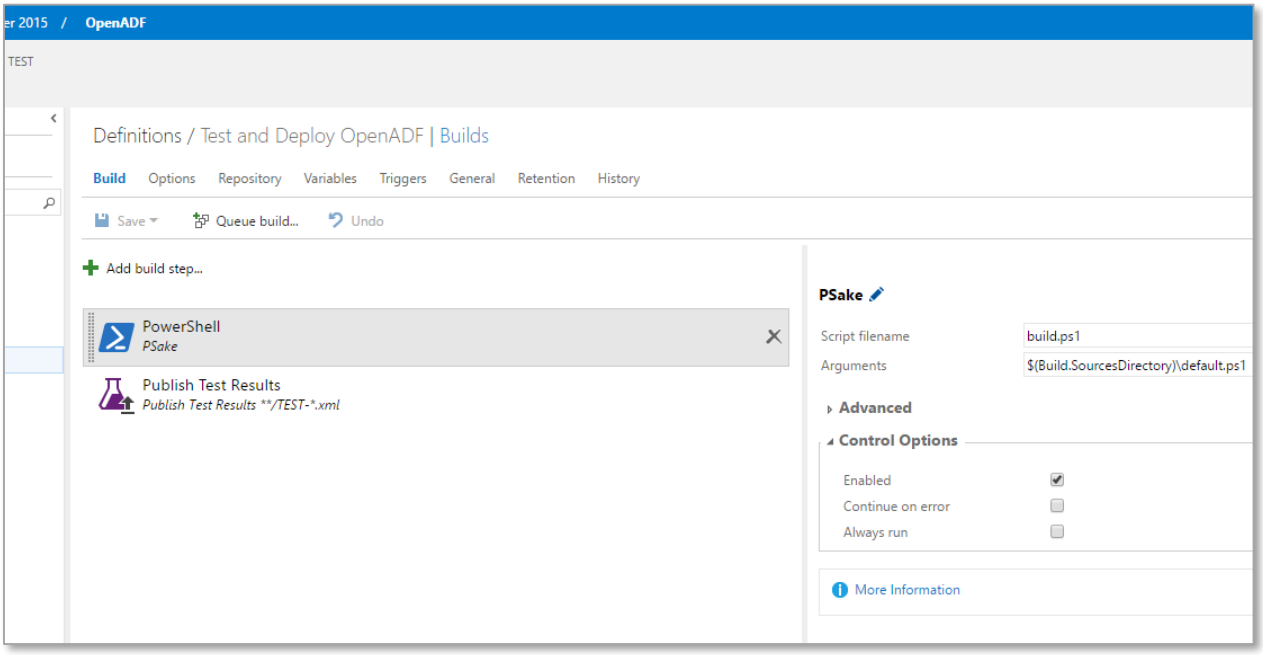
Build activities can include interaction with IT tools and solutions that are already familiar in the environment. For example, a script might interact with System Center Operations Manager[23] to onboard

a new server for monitoring, or it might use Maintenance Mode[24] to suppress alerts during the process of releasing changes to a server that requires a reboot. A script might call out to Azure Automation to activate a runbook that completes a routine set of work that is common across many releases. If there is a need to test changes to web servers and the test should be performed together with a hardware load balancing platform, the runbooks executed by Azure Automation might create and configure a new load balancing pool that points to the temporary test machines.

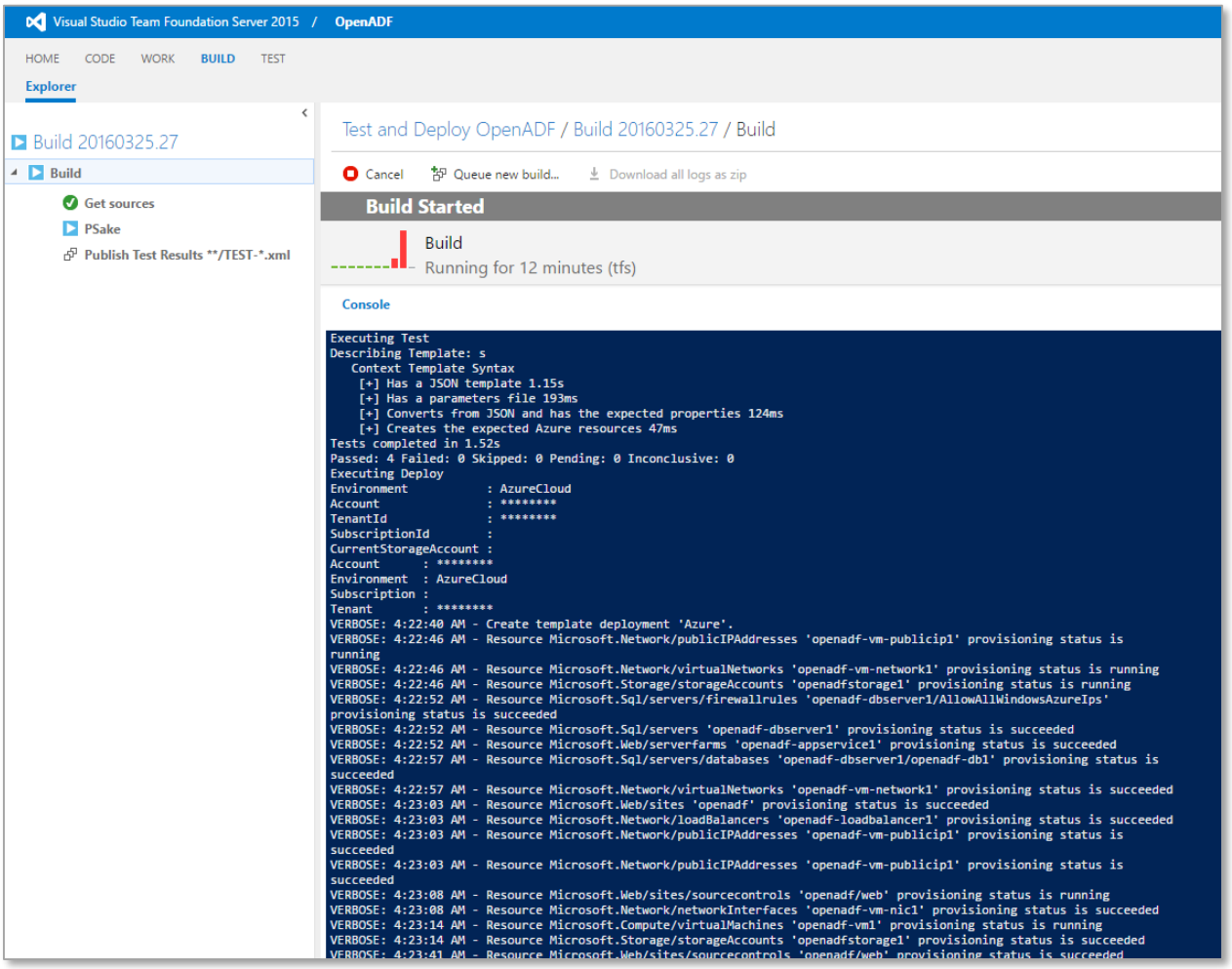
Triggering automation runbooks during build

Build systems can call user-defined automated processes to perform custom tasks during the build process. The success of the build is dependent on successful execution of these tasks. These repeatable tasks are stored as runbooks[25], which include secure credential management using the Azure Automation solution, and then the runbooks are called by the build system at the appropriate time using either PowerShell modules or a web service. Runbook activities are executed locally in your datacenter by the Hybrid Runbook Worker[26] component of Azure Automation.

PowerShell script as a Build task, using the PSake community project to simplify build automation



Real time output from PowerShell as a build is running

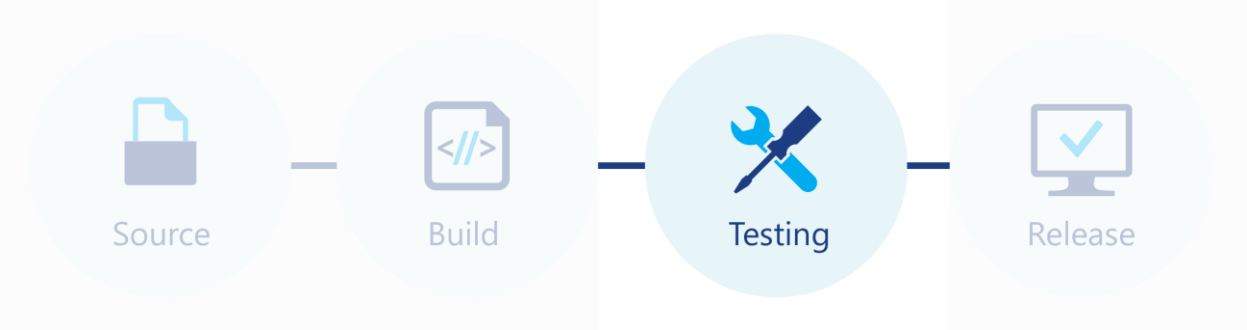


Key concepts in this section

The build system provides an environment to execute the scripts stored in source control to do work in the environment.

- The build service executes repeatable tasks when certain changes are made to files in source control.
- The output of the build system is not limited to binaries. The build system doesn't just compile code; it can also perform complex actions on an environment.
- Due to their flexibility and ability to interact with external systems, build systems often allow you to integrate your existing tools, solutions, and devices into your release pipeline.
- The real-time state of the build is a key performance indicator that needs to be visible to all stakeholders.

When you designing your build environment, consider products such as Visual Studio, Jenkins, TravisCI, AppVeyor, and Team City.



Test

When changes are accepted in source control, the build system will orchestrate the process to evaluate the quality of each change. This process usually consists of several steps. The first step is linting[b], which checks the code for style and formatting. This is followed by unit testing, where the functionality of the various elements of the code are tested in isolation. Finally, code is made available to test environments for integration or acceptance testing.

Applying these concepts to configuration as code:

- Linting and unit testing is typically performed on local build servers. Linting and syntax checking occurs when configuration files and/or service templates are loaded and evaluated to look common problems such as formatting that does not comply with the standards for your environment. Unit testing often includes tasks such as executing a test script to check parameter limits on new or changed functions, or checking to be sure that all sections of a deployment template have implemented the required properties.
- Integration and acceptance tests in configuration as code scenarios refer to applying a configuration to a machine, or deploying a template to a service, and then running scripts to validate that the resulting environment meets both technical and business requirements. For example, integration tests can validate that a website renders as expected, or that a report provides all the expected data, or that changes to the application do not prevent successful user authentication when integrated with external services such as Active Directory.

There are script language extensions available to simplify the process of authoring tests. Windows 10 shipped with an extension for PowerShell, Pester[27], which is designed for authoring tests. PowerShell Operational Validation Framework[28] offers a solution to organize and execute scripts specifically intended to validate operation. ServerSpec[29] is a popular extension for Rspec[30] that is popular for authoring tests. Chef InSpec[31] is an auditing and testing framework that is also based on Rspec. InSpec follows a similar approach as what you would expect with ServerSpec, but with a focus on compliance.

[b] The term *linting* is derived from the name of a Unix utility, lint, which flags suspicious or malformed code.

A high level example of testing a PowerShell function by passing in values and validating the output:

```
Describe 'function' {

    $team = 'blue'
    $country = 'USA'
    $result = Get-Team -team 'blue' -country 'USA'

    It 'returned the correct team' {
        $result.team | Should Be $team
        }
    It 'returned the correct country' {
        $result.country | Should Be $country
        }
}
```

Applying the same methodology to evaluating the state of a machine managed by DSC:

```
Describe 'Test-TargetResource'{
    Mock -ModuleName xTimeZone -CommandName Get-TimeZone -MockWith {
        Write-Output 'Pacific Standard Time'
    }

    It 'Should return true when Test is passed Time Zone thats already set'{
        Test-TimeZoneTargetResource -TimeZone 'Pacific Standard Time' -
IsSingleInstance 'Yes' | Should Be $true
    }

    It 'Should return false when Test is passed Time Zone that is not set'{
        Test-TimeZoneTargetResource -TimeZone 'Eastern Standard Time' -
IsSingleInstance 'Yes' | Should Be $false
    }
}
```

Example NUnit report output in XML format:

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<test-results xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="nunit_schema_2.5.xsd" name="Pester" total="2"
errors="0" failures="0" not-run="0" inconclusive="0" ignored="0" skipped="0"
invalid="0" date="2016-02-25" time="13:43:21">
  <environment user="greene" machine-name="DESKTOP" cwd="C:\github\pesterExample2"
user-domain="domain" platform="Microsoft Windows 10
Enterprise|C:\WINDOWS|\Device\Harddisk0\Partition4" nunit-version="2.5.8.0" os-
version="10.0.10586" clr-version="4.0.30319" />
  <culture-info current-culture="en-US" current-uiculture="en-US" />
  <test-suite type="Powershell" name="Pester" executed="True" result="Success"
success="True" time="0.1929" asserts="0">
    <results>
      <test-suite type="TestFixture" name="function" executed="True"
result="Success" success="True" time="0.1929" asserts="0" description="function">
        <results>
          <test-case description="returned the correct team"
name="function.returned the correct team" executed="True" time="0.1618" asserts="0"
success="True" result="Success" />
          <test-case description="returned the correct country"
name="function.returned the correct country" executed="True" time="0.031"
asserts="0" success="True" result="Success" />
        </results>
      </test-suite>
    </results>
  </test-suite>
</test-results>
```

The results of software tests for .NET are often returned in an XML format named NUnit, which is

supported by Pester and other language extensions. The results might be transformed to a report format and then hosted in the continuous integration platform, or simply emailed by a script when the build system receives a success response as an exit code from the tests.

There are also acceptance and integration testing tools such as Test Kitchen[32]. Test-Kitchen is an acceptance and integration testing harness that can be used to automate test scenarios including matrices of PowerShell Desired State Configuration and Azure Resource Manager deployment templates.

Test-Kitchen is very useful in scenarios where the same tests should be conducted against variances in application platforms (such as multiple versions of .NET) and across multiple environments (such as multiple versions of Windows Server or multiple platforms such as IIS, Microsoft Azure Stack websites, Microsoft Azure websites, and additional platforms such as Amazon AWS and OpenStack).

Test-Kitchen example describing a scenario where tests need to be run against multiple versions of an Operating System:

```
driver:
  name: azurerm
  subscription_id: <%= ENV['AZURE_SUBSCRIPTION_ID'] %>
  location: 'Central US'
  machine_size: 'Standard_A2'

provisioner:
  name: dsc

transport:
  name: winrm

verifier:
  name: pester

platforms:
  - name: windows-2012r2-wmf4
    provisioner:
      dsc_local_configuration_manager_version: wmf4_with_update
    driver:
      image_urn: MicrosoftWindowsServer:WindowsServer:2012-R2-DataCenter:latest
  - name: windows-tp4-wmf5
    provisioner:
      dsc_local_configuration_manager_version: wmf5
    driver:
      image_urn: MicrosoftWindowsServer:WindowsServer:Windows-Server-Technical-
Preview:latest

suites:
  - name: Sample_xWebsite_NewWebsite
    provisioner:
      configuration_script: Sample_xWebsite_NewWebsite.ps1
      configuration_data:
        AllNodes:
          - nodename: localhost
            websitename: test
            destinationpath: c:/sites/BakeryWebsite
            sourceurl: 'http://blogs.msdn.com/cfs-file.ashx/__key/communityserver-
blogs-components-weblogfiles/00-00-00-63-74-
metablogapi/3124.Demo_5F00_WindowServer2012R2_2D00_Preview_5F00_4677B514.zip'
```

Best practice is for contributors to run standardized unit tests locally before submitting the change for review by others. Local testing happens throughout the process, ideally. To simplify and expedite the process of creating a local virtual machine on a laptop computer, copying in a project, and executing tests, many authors use Vagrant to perform local test work. Vagrant provides a command line interface so that instantiating a new virtual machine, copying in a project, and executing a script, is accomplished by changing to the project directory and issuing the command "Vagrant Up". Vagrant is a common driver for Test-Kitchen.

In some environments the test is written first and then the change is made to the configuration scripts to create the outcome that is validated by the test (this is known as *Test-Driven Development or TDD*). Test Driven Development can lead to increased quality when authoring configuration as code. A simple to follow process is given in the article Guidelines for Test-Driven Development[33].

When all tests pass, the contributor submits their pull request to the owner of the project in source control. This is also an opportunity where impact of your changes can be evaluated together with other contributor's requests to make sure that even if your scripts do not overlap, the outcomes are not going to cause a conflict. The results are made available as reports or logs so both the contributor and the owner of the source can consider whether the change should be deployed to production. This makes it easy for the owner to feel confident in accepting the change.

What does testing mean to me as an IT Operations Professional?

Testing changes before they are put into a production environment is a complicated task. Even with the best, most agile, and well-developed system, testing is still hard. Test environments are never the same as production. Even in public cloud services where teams of service engineers work diligently to make sure test environments are as close as possible to production environments, there are still environmental differences beyond their control.

When a human connects to a test environment to validate that a change does not affect the environment in a negative way, we are counting on them to test the change perfectly. If the change impacts many systems, we expect the human component to perfectly test the required change across every type of system.

Benefits of test automation include:

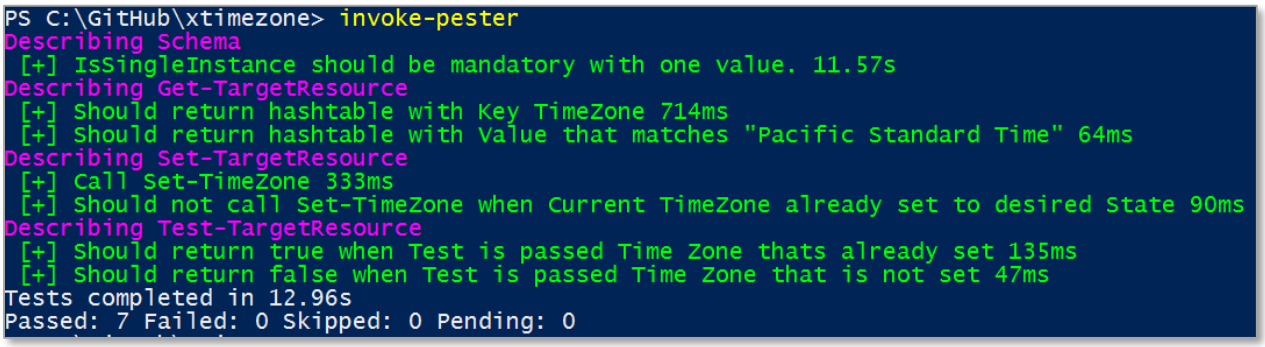
- Script quality is improved by catching errors as early as possible. During development, changes are evaluated for quality through lint and unit tests.
- Operational validation assesses the potential impact of changes, including changes to application code and infrastructure configuration, and even Windows Update and policy decisions. Changes must flow through a series of quality gates before they reach production to eliminate (as much as possible) the possibility of negative impact.
- Complex tasks are handled by the test automation platform including, if necessary, deployment of multiple operating system versions or combinations of applications to verify every combination that could exist in production.

Think about the manual steps you might take to test a change: You might create virtual machines, install the operating system with all the applications needed to mirror the configuration you would expect to find in production (you might use snapshots, differencing disks, or clones to simplify setup). Then you would manually make the change across those machines and validate that everything still works as it should. Therefore, that sequence of tasks is exactly the minimum set of capabilities you would expect out of a test automation solution.

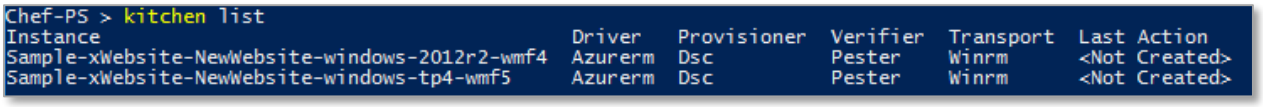
This is where the differences between a cloud platform and traditional virtualization concepts become apparent. Creating a new environment that includes network, storage, and virtual machines where this testing can be performed in a cloud platform is a series of calls to an API from within a script, and your build platform might include plugins to make this simple. For environments that cannot be automated, the system will need to provide a workflow where manual steps or requests for work can be handled, and this can significantly reduce the velocity of work.

After the test scripts have completed, the environments that were created for operational validation are usually decommissioned to reduce the cost of hosting in public cloud environments, or to free up resources in a private cloud. The most common approach is to remove a test environment if all the tests pass, but persist the environment if one or more tests fail. The person who investigates the failure will then be able to examine the logs and attempt to isolate the cause of the problem using the same environment in which the problem occurred.

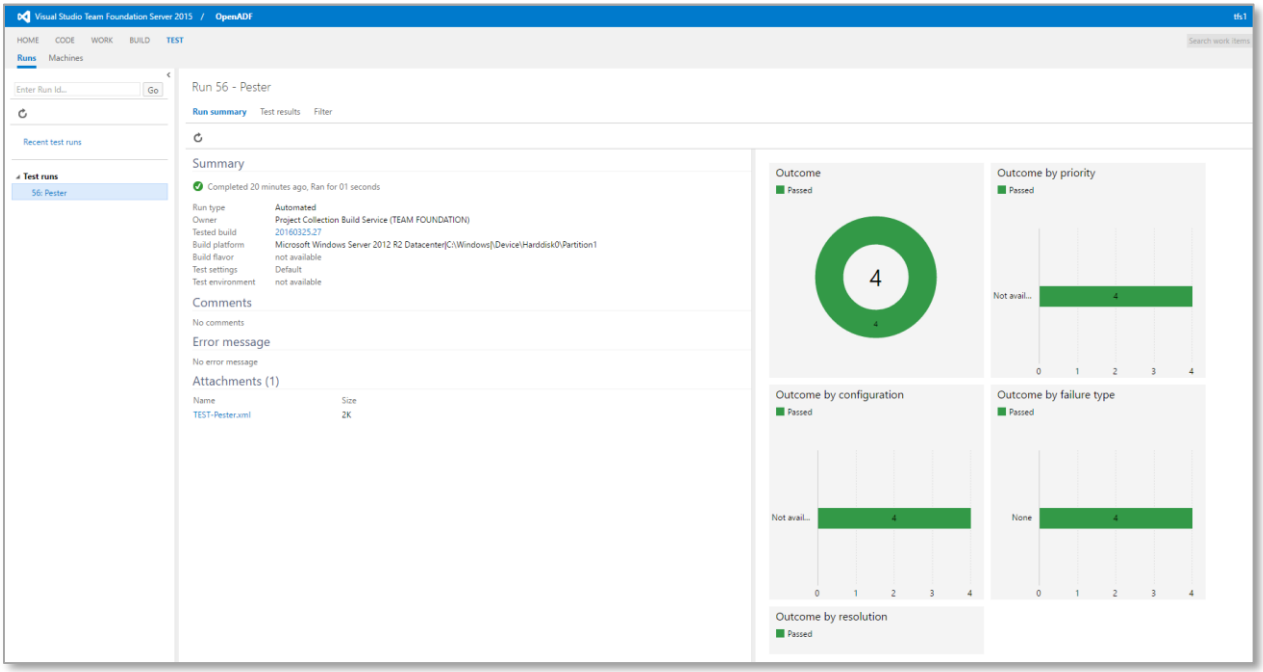
Pester console output provides an easy to understand review of a project



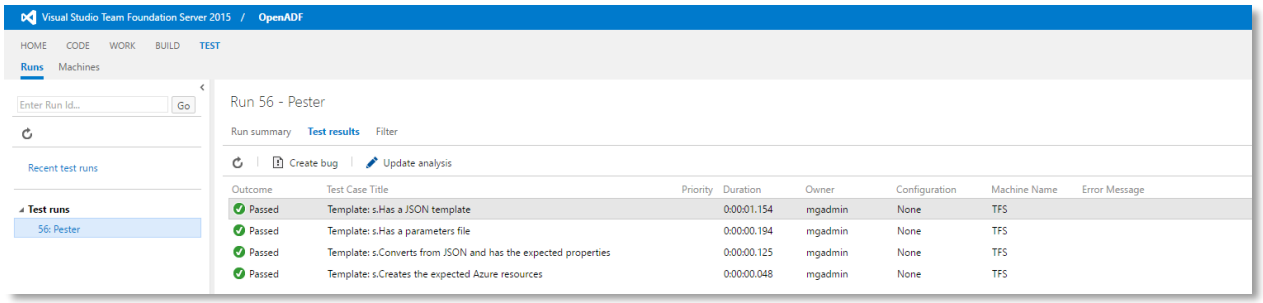
Test-Kitchen output simplifies complicated long-running scenarios where Pester is used to validate work across multiple environments



When using a platform such as Visual Studio TFS, teams can review test reports to understand where problems exist before introducing changes to production



Drill in to individual tests to understand impact



Certifying operating environments using test automation

An automated test platform can also help address complex certification requirements in organizations that are heavily regulated. While automated testing obviously will not address vendor support, having a mature test solution provides an elegant method of introducing new operating system versions or new releases of runtime environments and validating the new release across all applications.

Certifying an application on a new operating system using a release pipeline would include:

1. Use a release pipeline to create the virtual machine images for the new application.
2. Create templates based on the new images, and then add them to the cloud environment where test automation is hosted.
3. Add the new operating system version to the test matrix for the application.

4. Trigger the continuous integration environment to conduct tests against the latest version of the application.
5. Review the test reports with application owners to confirm acceptance or address compatibility issues.

Test is the most important aspect of the release pipeline concept

Looking at a release pipeline as four distinct elements (source, build, test, release), the most important element is testing. One of the main goals of moving to configuration as code is to enable efficient, automated testing. Testing is the most significant part of your effort to make sure that issues are never introduced to production. Without effective testing, chaos will still be present at the end of the pipeline despite the other improvements you may have made to your process.

It will be difficult for you to author a comprehensive set of tests when you are first getting started with a test automation platform. You should plan to start small, initially authoring only the essential unit and acceptance tests that you need to get started, to demonstrate the value of the concept. You will increase the number of standard tests that you run against all releases over time, implementing unique tests for each set of files as you gain experience and refine your test platform.

Eventually, every operations team is likely to release a change that impacts the production environment in some negative way, ideally in a very small way that is quickly identified and fixed, but sometimes the impact throws up a lot of debris and leaves a big crater. The worst impacts become the stuff of legends that are handed down to future generations of IT operations personnel. Generally speaking, most of us hope not to feature prominently in that sort of legend, which is yet another great reason to go to the trouble of implementing a release pipeline.

If—when—a defect does slip past the automated tests, make sure to work with your partner teams to modify existing tests or add new tests to your system so that the problem will be detected in the future. Over time, gaps in the testing protocol will be closed through a process of learning by experience, and your confidence in the integrity of the build will improve accordingly.

Monitoring is not a substitute for testing

One approach to validating released changes is to use the monitoring components to detect anomalies in the environment, and then raise an alert when a problem is detected. The expectation is that if something is wrong, monitoring should catch it. If monitoring did not catch the problem, the monitoring system is augmented to check for this new condition.

In this model, a change is typically introduced during a scheduled maintenance window. During the maintenance window, alerts from the production monitoring system are suppressed to prevent erroneous messages from being sent out during planned disruptions. After the change, the monitoring system is checked to determine if new conditions exist that would indicate a service outage. If not, alerts are re-enabled. If there are concerns, the problems are investigated and a rollback is initiated if needed. The maintenance window is sized to accommodate the expected time needed to complete a rollback.

Using monitoring as a solution to identify issues with changes to an environment has some theoretical faults that explain why test automation is so compelling. The list below provides examples of such faults:

- Operational validation can be a resource intensive procedure involving many in-depth checks. It is expensive in terms of compute resources to monitor at this level of scrutiny.
- Some monitoring systems focus on capturing application errors and service performance diagnostics. For example, if normal usage of an application includes an average of 30 requests per minute and requests have dropped to zero, that might indicate a failure. Neither a lack of application errors nor a lack of usage are substitutes for active operational validation, where the service is thoroughly checked to make sure code that might not be used until it is needed is still behaving as expected.
- Monitoring does not check for code-level issues. For example, a new function might be introduced as a change including unit tests that make sure the function returns only an integer, or only a specific range of string values that align with parameter validation. Without the tests, a condition could be released to production where the function has the potential to return unexpected values. This could be the root cause of future errors where changes are introduced that use the function in a new way. This type of issue can be very expensive to troubleshoot.

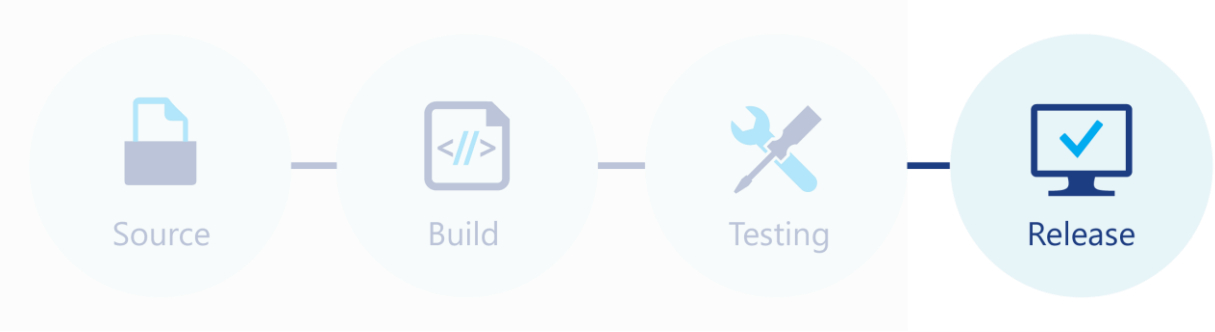
Finally, using monitoring systems for production validation can create cultural consternation when there is a dispute over who is responsible for identifying and correcting the problems that are discovered after deployment. The more time that elapses between making a change and identifying a consequent problem, the more likely it is that there will be a conflict over who "owns" the response. Monitoring, even in near real time, is inherently an after-the-fact activity. A robust testing protocol helps to avoid internecine conflicts by reducing a common source of friction between teams.

Key concepts in this section

The test platform provides confidence that every change to a production environment has been extensively validated.

- Testing is the single most important aspect of the release pipeline.
- Automated testing is integrated with the build system:
    - Committing or checking-in changed files to source control triggers the build system, which initiates linting, syntax checking, and unit testing prior to allowing the commit to succeed
    - The build system can initiate complex integration and acceptance tests
- Monitoring the environment after deployment is not a substitute for automated testing before deployment.

Consider the following tools when you are designing your test platform: script language extensions such as Pester, platforms such as Test-Kitchen to automate complex mixtures of requirements so that tests can be thoroughly evaluated before release to production, and tools such as Vagrant to simplify rapid creation of virtual machines for testing during development.



Release

When a build system is used to compile code, the resulting output is a "build artifact." Typically, this refers to a binary file that is released as the deployable software product. A build script that is focused on managing configuration as code will also include build artifacts but rather than binaries that artifacts are production ready script or data files. For example, the build service might execute scripts that create MOF files[34] for PowerShell DSC, and then publish them to Azure Automation.

There are two common deployment models, continuous deployment and incremental deployment.

Continuous deployment is popular for "fungible servers" or "immutable servers", such as web servers where every server is an exact configuration replica and it is trivial to retire one node from a farm and then replace it with a new node. In this case, the release scenario might call for the build system to proceed node by node, removing the node from a load balancer, deleting the VM (node), replacing the old VM with a new VM that has the latest code, and then adding the node to the load balancer to move it into production.

Certain kinds of changes require a *cutover* from one set of nodes to another. A cutover occurs when a system cannot be incrementally updated; the old system continues to operate while the new system is prepared. When the new system is ready, a rapid change (the "cutover") is made to take the old system offline and then replace it with the new system. Including a cutover in a continuous deployment scenario can be accomplished by changing the configuration of the load balancing solution after the nodes with the latest code have been provisioned. In the future, Windows Containers will simplify this process even more by isolating applications so that only the affected container has to be updated.

Incremental deployment is popular for servers that must be maintained for a long period of time and cannot be easily disposed and re-deployed. Database servers, failover cluster nodes, and domain controllers are among the kinds of servers that are typically managed under an incremental deployment model. When you check in a change the configuration is delivered to the server or servers as an updated version of the existing configuration.

The industry expects IT architects who are designing new services to create solutions that leverage continuous deployment whenever possible. This is because servers that require incremental deployments tend to be more expensive to maintain than continuous deployment servers.

What does the release phase mean to me as an IT Operations Professional?

Imagine that a change is accepted to modify 20 values in the Windows registry, and the change will affect 75 server nodes. That means we are counting on a human to be perfect 1500 times, or else there is a chance we might cause an issue with a setting that could lead to a service outage. In practice, most operators would leverage existing tools to make bulk changes to as many of those 1500 changes as possible, but it is still likely to involve many repetitious and error-prone manual steps to complete the update. Now, what if we need to release a change that is slightly different across Windows Server 2008 R2, Windows Server 2012, and Windows Server 2012 R2?

Consider how the same change would be implemented using configuration as code. To review steps discussed thus far in the process:

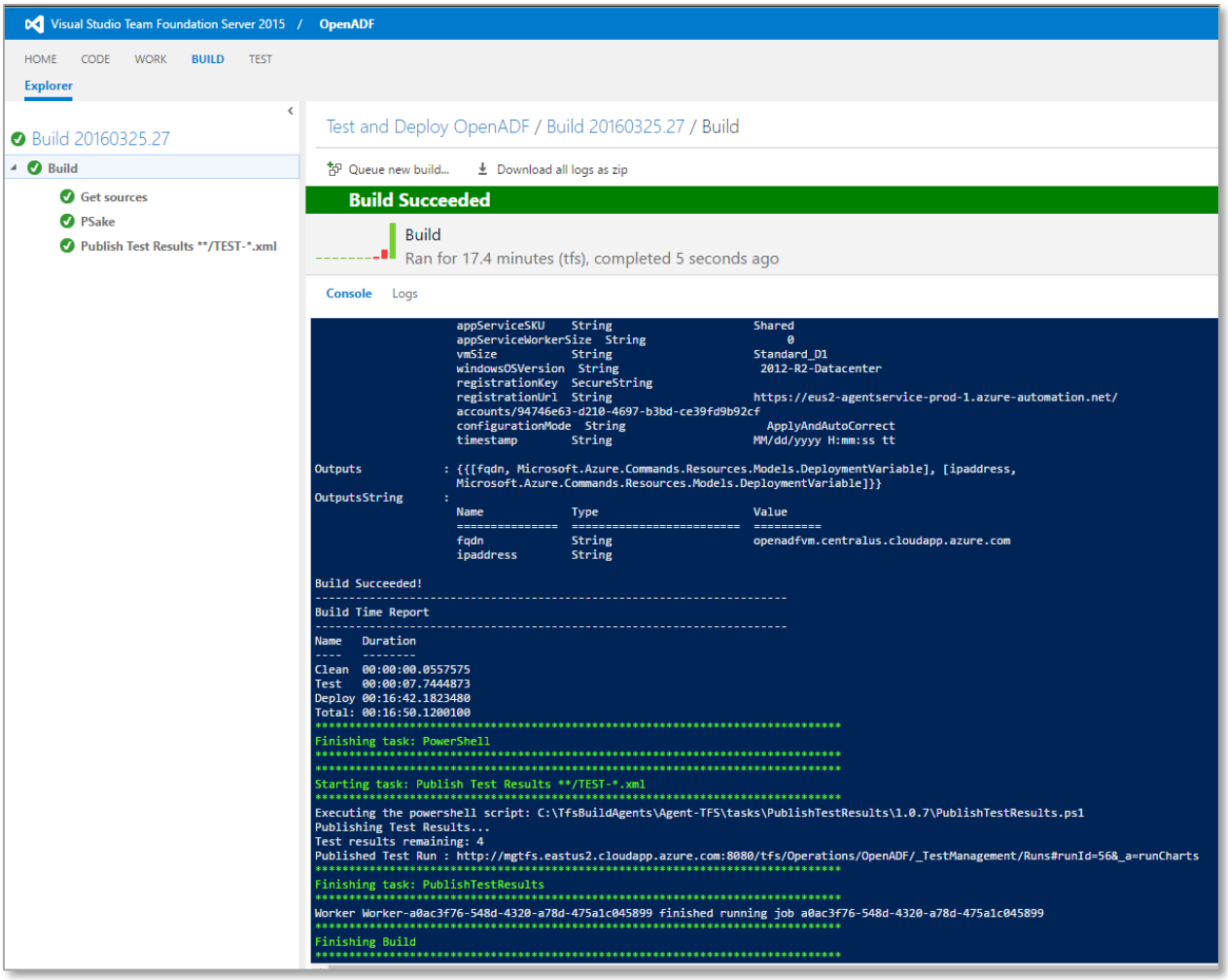
- Edit a text file to introduce the registry settings.
- Execute lint and unit tests during development.
- Test the changes in your private virtualization environment; make modifications if needed, until you are satisfied that the changes will have the desired effect.
- Check the file into source control together with test and deployment scripts.

The rest can happen automatically via the release pipeline.

- Tests are run to verify that your change meets the requirements agreed upon for your organization and does not have functional errors.
- Operational validation tests verify that the change does not cause a malfunction for any applications or services that represent a production impact across multiple operating system versions with various combinations of software.
- The release artifacts are published to the environment, ready for deployment.
- Build service executes the scripts you provided that take action to implement changes.

It is possible that your deployment will be released to more than one environment. In some cases, even with automated test validation, projects are released to quality assurance (QA) environments for further hands-on validation before being fully trusted for production. This might include non-production scenarios such as user acceptance experimentation. Products such as Microsoft Release Management[35] offer the ability to manage changes to the project based on the target environment, should environmental factors dictate the need for feature flags or changes in configuration data (for more information, see Understanding Microsoft Release Management[36] on MSDN. These solutions also allow for "promoting" a build from one environment to the next by triggering each deployment scenario through a workflow of approvals.

In a fully automated pipeline, changes to production are trusted after in-depth testing for quality



Reversing a change to the production environment

Consider the concept of a *rollback* in change management planning. Rollback is a popular term used to describe taking a change back out of an environment after it has been implemented. For a service that can be easily re-deployed, performing a rollback means triggering a rebuild and re-deployment of the previous version of the configuration, all of which will happen automatically once the process is started. As mentioned previously, there will be times when a defect will get through the pipeline. The usual priority once an issue has been recognized is to restore the environment to a known good state as quickly as possible. In those instances, recovering a server or service that is managed using configuration as code and a mature release pipeline is a relatively low cost operation.

In contrast, with servers that are managed under the incremental deployment model there are changes that are so complex that they are effectively impossible to roll back using a straightforward re-build and re-deployment process. Changes that alter the structure or location of data are often particularly difficult to roll back. In these cases, the inability to roll back certain changes would mean re-deploying

the servers from the last known good configuration and restoring the server state and data from backup.

Patterns for deploying configuration as code

The following section describes two Microsoft tools that directly support configuration as code scenarios. PowerShell DSC is ideally suited for use in the release phase of a pipeline. Azure Resource Manager and the Hybrid Runbook Worker are suited for configuration as code deployments in hybrid environments.

PowerShell Desired State Configuration

For PowerShell DSC, a release follows one of the following techniques depending on whether you are releasing a module or a configuration.

The work performed by PowerShell DSC is provided by PowerShell scripts packaged in modules. These must be distributed to machines that require them for configuration.

A private NuGet feed makes the module consumable from any machine in your environment. The PowerShellGet module[37] provides commands that assist you in publishing to and consuming from a NuGet feed. You might consider having multiple feeds in your environment for modules that are still in development vs modules that are deemed production-ready.

DSC pull server or Azure Automation DSC Service provide solutions where the released modules can be automatically distributed to target nodes. The build service would need to distribute the files to the folder where modules are contained on the pull server (this is a property of the configuration for the pull server itself). For Azure Automation DSC[38], there is a cmdlet to publish available modules to the service.

You might combine these methods so that a build script publishes to a feed and kicks off an automation job to update the pull server or Azure service from the feed. This would provide ubiquity throughout the environment to always use the latest modules from any process on a target node.

Configurations provide declarative language to describe machine settings. There are two models for applying a PowerShell DSC configuration: push and pull.

Push is initiated using the cmdlet `Start-DSCConfiguration` with parameter `-ComputerName` to specify a target node where the configuration should be applied. A connection is made using WinRM and the MOF file is encrypted locally on the node by Local Configuration Manager.

Pull mode is configured in Local Configuration Manger to register the machine as a client of a service. The build service in a release pipeline compiles MOF files and then copies them to a DSC pull server and then create file checksums or publishes the configuration to Azure Automation DSC and verifies it was compiled successfully by the service.

Azure Resource Manager templates

The ARM API is consistent across public and private cloud environments, which means that the same REST calls and PowerShell cmdlets are used to deploy to any available Service Endpoint[39]. The available resource providers and technical requirements have potential to change based on the environment.

Provide a build script that includes cmdlets from the Azure module available in the PowerShell Gallery. Pre-stage any required resources and use the PowerShell cmdlet `New-AzureRmResourceGroupDeployment` to reference the JSON file for your deployment template. The build platform might offer capabilities to store information that needs to be passed as parameters such as account passwords.

Also consider the case where you will deploy nodes across both Microsoft Azure and Microsoft Azure Stack. This would include hybrid cloud scenarios where some components of a deployment will be provisioned in a public cloud environment and others may only be hosted in your private cloud environment. An example might be runbooks in Azure Automation that will maintain virtual machines hosted in your datacenter.

Eventual consistency

One of the core concepts behind many of the configuration management products in the marketplace today is the idea of *eventual consistency*. Products that incorporate the idea of eventual consistency include Chef, Puppet, CFEngine[40], and PowerShell DSC. Eventual consistency is where one's infrastructure is given a policy and at some interval after that the node (the server that retrieved the policy) is responsible for getting to its desired state. This may take several runs of the configuration management agent (maybe a reboot is needed or something else in the environment has to happen first). When developing configuration management to be eventually consistent, there are some things to keep in mind:

- Code will need to be written. Either the application being supported or the configuration management policy (or both) need to understand that the environment is designed to be eventually consistent.
- A single node fulfilling a role is likely to cause outages
- The application and the resources it requires need to be loosely coupled as they can be independently upgraded. Loosely coupled means that they interact through a defined interface and part of that interface may include a negotiation as to what level of interaction both sides support.
- Applications should be able to either fail gracefully or operate in a degraded state when missing dependencies. For example, if the database is not available, rather than getting a "yellow screen of death" (a common ASP.NET error page), a user would see a page with data from a local cache, be redirected to a support page, or a nicer maintenance page.

Key concepts in this section

For Operations, release refers to the automated deployment that can be executed by the build platform after scripts stored in source control are validated.

- There are two models for releases, continuous and incremental.
- The concept of eventual consistency means as changes are released, especially in some high velocity environments, the end result might not be immediate.

When designing a release platform, search for products such as Microsoft Release Management, and Chef Delivery[41].

Continuous integration

The service that looks for check-ins to occur in source control and queues tasks on the build environment is often referred to as *continuous integration*. This platform provides a management interface and workflow for tracking and reporting on when changes have occurred and work that the build system has performed. Typically, there is also an approval system where work can be queued for the build servers to perform, but a person has to approve the work before the server will proceed with build. This might be required when manual steps need to be taken such as making a request to a team that manages key resources that are not yet automatable.

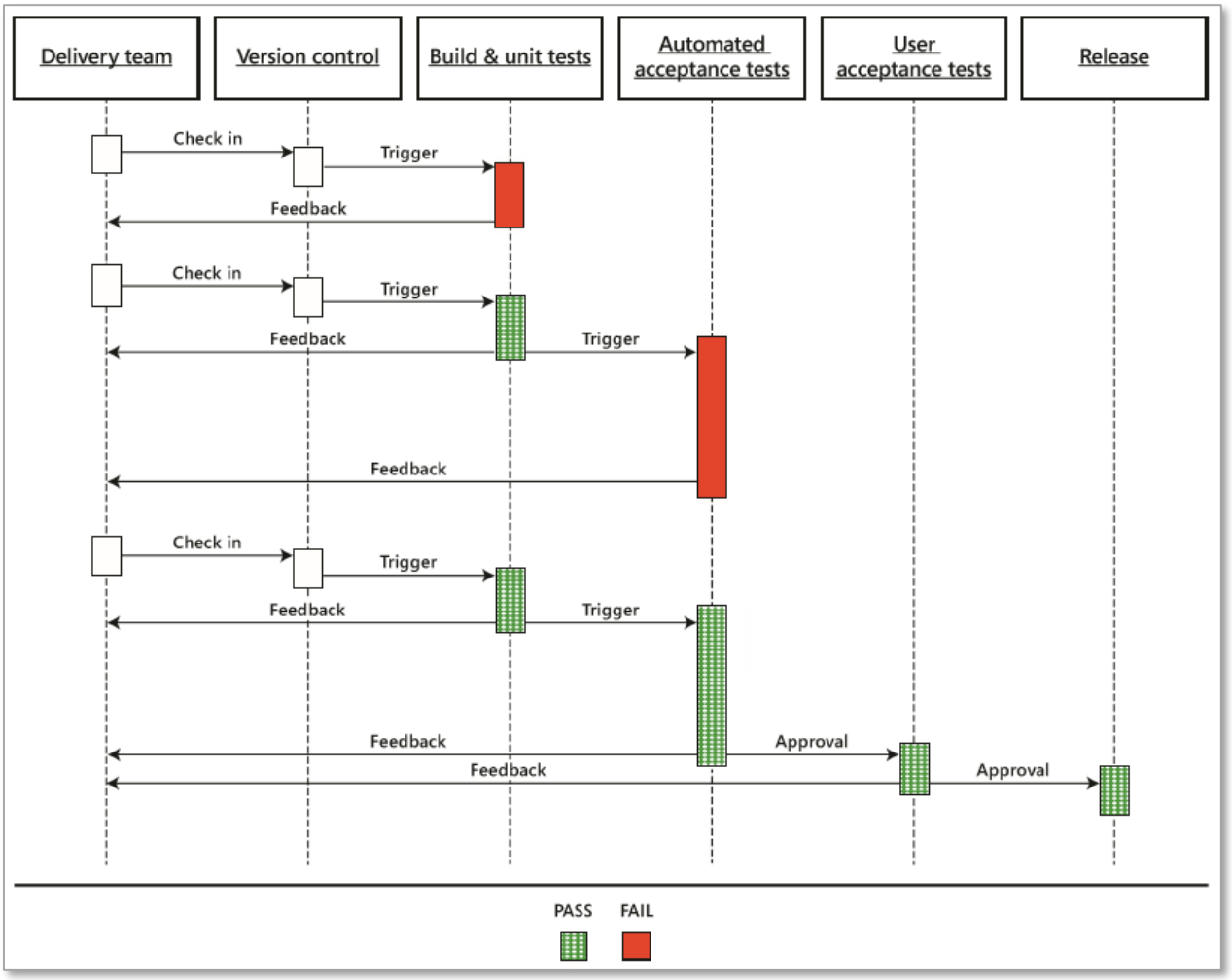


Figure 1: The deployment pipeline workflow

A benefit of continuous integration (CI) is the opportunity to create quality gates using an automated workflow so that tests are run against every check-in. In this scenario, new changes are not merged into the master branch until the tests succeed. One of the advantages of a CI pipeline (versus ad hoc or scheduled builds) is that the state of the master branch is continually being verified by the test suite. The tests in the project drive quality, and no new contributions are allowed until all tests pass. This is referred to as a gated check-in, meaning that certain tests need to pass before a commit is pushed to the master branch. If the tests pass, the gate is open and the push occurs. If the tests fail, the gate is closed and the push is blocked until the problem is corrected and the tests pass.

When a build fails due to an error or failed test (an event commonly called “breaking the build”), new check-ins are suspended except for changes needed to fix the build. Only after all critical tests are passed is the change integrated into the main branch, and then new check-ins can resume. In this way, the master branch is continuously maintained in a known good state.

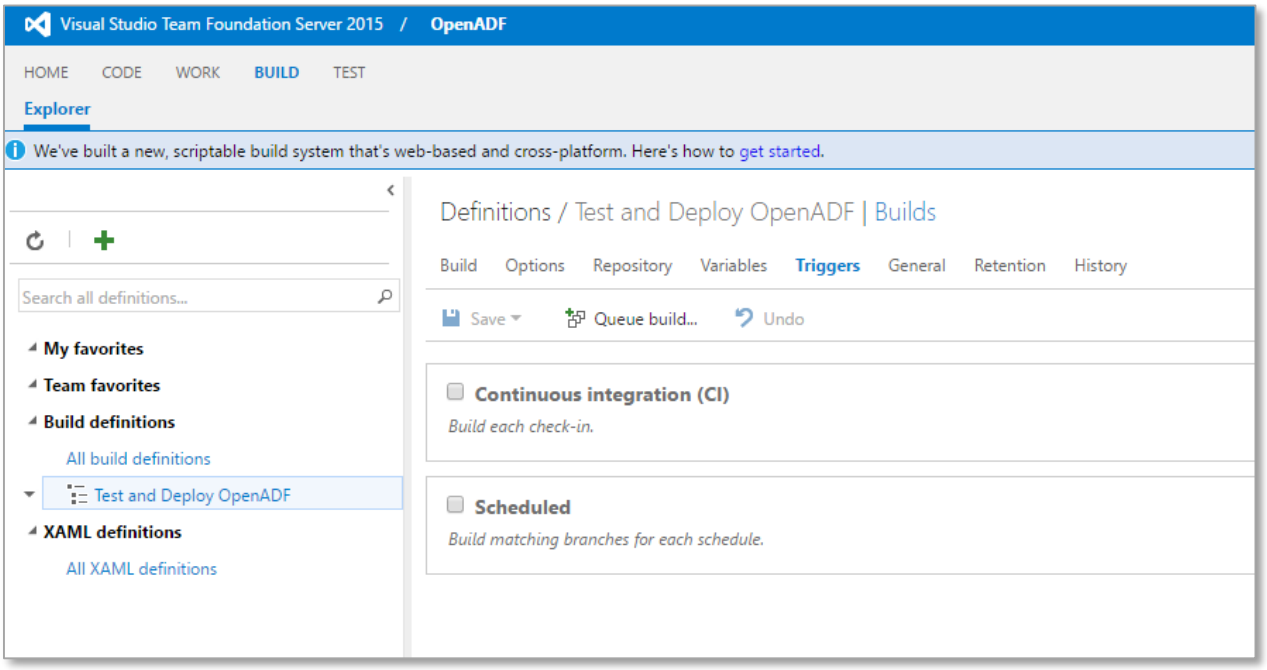
The state of the build is commonly referred to using the colors red and green. A "green" build is in a positive state, indicating that all tests were passed. A "red" build has failed one or more tests.

As an alternative to running tests against every check-in, many CI platforms allow you to schedule builds to occur at specific times. Often, users can manually trigger an on-demand build. Either option will create a bottleneck in the CI process, which is why it is preferable to run tests on every check-in. This bottleneck occurs because contributors will not know the state of the build until after the next scheduled build and test run, leaving them to investigate all the change sets from the previous time interval. There will be cases where a complete change requires input from multiple contributors. In this scenario, consider using a combined check-in from one author; the objective is to ensure that multiple check-ins result in only one test run. It is critical that tests be executed locally before check-in to ensure reliable results for integration tests as work is combined.

The emphasis on running builds at check in does not mean that all scheduled builds are bad. The CI platform should rarely be idle. Off-peak hours can be used to run scheduled builds to validate test scenarios that are too time consuming or unnecessary to run at check-in. For example, backwards compatibility tests might be run against a current version of project A and all previous versions of project B that are deployed in production. Such tests take time and would probably never be fully conducted by humans, even though it is understood that it would be valuable to do so. Additionally, many projects use scheduled builds (affectionately known as "nightlies") to make artifacts that early adopters can test drive.

There are online services that offer continuous integration for projects hosted on Internet-accessible source control systems. For example, open source PowerShell DSC modules published to the PowerShell team's GitHub repository[42] are connected with a service hosted by AppVeyor[43]. There are a variety of other services available including a service hosted by Visual Studio Online (both source control and build service). PowerShell.org[44] offers a service to the community using a platform named TeamCity[45] that is free for open source PowerShell-related projects. On-premises it is common to host services such as Visual Studio Team Foundation Server, which includes the ability to locally host build services in your datacenter.

Some continuous integration platforms include options for build triggers, including automatically building at every check-in or scheduling builds



Additional scenarios

While a detailed discussion of these scenarios is outside the scope of this document, they are worth consideration when implementing a release pipeline model for managing Windows servers.

Combining new tools with existing tools

While the process improvements referenced in this paper are relatively new, the tools used to support the process do not necessarily need to be. In this context, a *tool* is any application, utility, API, or technology that you use to implement your release pipeline.

New tools and existing tools are not required to be mutually exclusive; if a tool you already have continues to provide value in your release pipeline, use it. In particular, your existing investments in high level management tools are not obsolete. Most of the management suite tools have APIs and features that are well-suited to the release pipeline model, although you may not presently be using the capabilities that they provide.

IT operations professionals must recognize their ability to integrate tools to architect a solution that brings forward the strengths of their existing tools and skillset in to a process that helps them to be more valuable to the organization.

It makes sense to take small steps such as integrating key components into the existing server lifecycle management solution. For example, if there is a working deployment solution in place today, select roles that are deployed frequently and that do not require preservation of state information for your

first pipeline deployment effort. Consider implementing a declarative model in the sequence of tasks that run on the server during deployment.

The changes that a release pipeline will introduce to your usage of existing tools include managing configuration as code in source control, multiple teams collaborating on server configuration through source control pull requests, and the most significant improvement: testing the configuration code for style, syntax, unit tests, and operational validation, before changes are implemented by the deployment technologies.

As a working example, consider one of the ways that System Center Configuration Manager (SCCM) might fit into a release pipeline solution. SCCM includes a feature named Task Sequence that allows the administrator to define commands that should be executed in each phase of an operating system deployment. A finalizing pass[46] of Windows Setup occurs just before displaying the Welcome screen for the first time. During this phase, SCCM can run commands such as Start-DSCConfiguration[47] to apply configurations defined by a MOF file, or apply meta-configuration to Local Configuration Manager[48] to add registration information for Azure Automation DSC.

In this model, SCCM is used to bootstrap a node during deployment. Administrators would use SCCM reporting to track and analyze deployments, but they would probably use the reporting capabilities of Azure Automation or Microsoft Operations Management Suite[49] to track and analyze changes to servers after they have been deployed. This example of blending existing and new tools could be applied to a variety of deployment tools and technologies.

Another example is combining configuration as code with existing environments where Group Policy[50] is a trusted method for applying configurations to domain joined servers. It is technically possible to continue delivering security baselines using Group Policy and at the same time deliver application requirements using configuration as code, but there is no integration between the two systems during the authoring process. Lack of integration leads to the risk of conflicting changes and it does not allow for testing frameworks that are not joined to a domain. If you choose to use Group Policy together with a declarative configuration solution, consider generating and applying policies using scripts stored in source control maintained by application owners.

Consider transitioning settings from Group Policy to configuration as code in a graceful fashion so that security teams and application owners can work collaboratively to overcome the inevitable challenges. Make decisions about which settings should immediately move to configuration as code, and which should be governed by existing policies based on the impact of changes to operational process and alignment with project goals. Shifting to configuration as code resolves the requirement for application owners to have permissions in Active Directory to self-support managed changes to their servers, and lays a foundation for introducing server footprints where Group Policy is not available such as the Nano installation option for Windows Server 2016.

When you are reviewing your tools, focus on which role each tool is best suited for. Try to avoid the ambiguity that comes from having two or more tools with overlapping roles. You will probably find that you need to add at least a few new solutions to compliment what you already use. In some cases,

familiar tools may need to be retired immediately, while others will be retained or gradually phased out. If you are starting from scratch, you get to choose exactly the tools that are optimally suited to your environment.

In some cases, IT only consists of operations personnel. If IT operations is working with application owners, you might find that they are already using the tools referenced in this document. This is especially true if you have developers who author applications specific to your organization or business. There is much to gain by aligning with these teams and sharing resources, as well as integrating process to create a common release model. As an IT Operations professional, look for opportunities to work together with these teams so that they can derive more value from the environments you are supporting.

Automate builds of Operating system images

Tools such as Packer[51] and Boxstarter[52] create operating system images that are managed by flat files intended to be incorporated in a release pipeline methodology. In this model, the scripts needed to customize the image are stored in source control (unattend files, DISM scripts, custom startup scripts) and the build script references media to create and publish a new image to deployment environments. When combined with offline patching scripts that pull from Windows Server Update Services[53] (WSUS) and use the DISM cmdlets[54] to apply updates to an operating system image, this can be an elegant process to automatically ensure that new servers meet security baselines.

Packages and containers

The same methodology can be used to publish packages to NuGet feeds, Chocolatey feeds, and in the near future, run PowerShell commands that produce Windows Containers. In this model, your project might contain the files required to support a website and a build script would run to construct the package, either locally on a build server, or by capturing a container on a virtual machine, and then releasing the artifact to a private location.

Visibility

Where a release pipeline model has been implemented for other platforms, it has been found that *visibility* is a key requirement for acceptance and success.

Visibility means that everyone on the team has access to all the information required to know what's happening to the software as it goes through the pipeline. – Building a Release Pipeline with Team Foundation Server 2012, https://msdn.microsoft.com/en-us/library/dn449955.aspx

This means that across the environment, as visibility increases for IT operations stakeholders to make everyone aware of changes that are happening, especially in production, the easier it is to predict the outcome of future changes and create plans that avoid conflicts.

Notifications

Timely notification of changes to the status of the pipeline is a critical aspect of visibility. Your release pipeline needs to include a notification component. The method you choose to deliver notifications depends on your continuous integration platform. Here are a few options:

- Plug-ins for popular collaboration tools, such as Slack or HipChat
- Internet Relay Chat (IRC) networks
- Email notifications

In addition to push notifications, a web-based solution may be added so that system users and project stakeholders can refer to it for overall status and reporting. The continuous integration platform can become a dashboard for tracking change in the environment. The status of the build is a critical indicator of the health of the process, and it is important to ensure that it is always visible to stakeholders. Visibility for continuous integration is extremely important as it reveals failures. When the build is broken, the ability to deploy changes has been compromised and requires immediate attention to restore that process to health.

Runbooks together with Azure Automation Hybrid Runbook Worker can handle common notification tasks, including scripted calls to a local chat service or sending email. CI platforms also include plug-ins to handle Email notifications. An example of using Email for notification would be the build system notifying members of a distribution list any time a new release is deployed in to production on Microsoft Azure Stack. Or, a runbook that checks the status of a managed node in Azure Automation DSC, and based on the node name, notifies project stakeholders and operations that a deployed machine did not finish in the desired state.

Monitoring and status

After nodes are released to production, it is the role of a monitoring system to alert operations staff when problems occur, and to provide a detailed reporting warehouse of information. Reporting on machine state and monitoring for health issues should be a planned outcome and defined in the configuration as code for any server node that is released in to production.

Monitoring Windows servers

Server configurations in source control can include deployment of the System Center Operations Manager agent or the Microsoft Management Agent for Microsoft Azure Operations Management Suite. Especially in scenarios where bursting (rapidly deploying additional resources to meet a sudden increase in demand) or continuous deployment models might be present, the OMS platform provides an elegant solution where data is gathered about all nodes and reports and alerts are created by application owners.

Node status reporting

If the server configurations also register the node with Azure Automation DSC service, operations staff are able to view all target nodes to determine whether any server is no longer configured correctly.

When supplemented with runbooks, detection of a noncompliant node could be followed by remediation during maintenance hours, if scheduling is required.

Servicing deployed environments

In a perfect world, servers would be immutable infrastructure[55] and they would never require servicing. In a continuous deployment scenario, it actually is possible to come close to the ideal: If a server is broken, redeploy to new servers. If a server needs to be updated with operating system level security updates, generate a new release of the template used to deploy servers, and redeploy to new servers. However, as discussed earlier in this document, not all applications will fit in to a continuous deployment management process.

Accounting for both immutable servers and long term deployments, there should be only two methods to access production server nodes:

- Deploy a change through the release pipeline
- Routine tasks performed by an automation platform

The exception to this rule would be a remote administration model of choice (for example, Remote Desktop or PowerShell remoting with full language) that is only present for dealing with outages. This is frequently referred to as an "in case of emergency, break glass" scenario. While this might be attractive in the early stages of adoption for a release pipeline model, it should be avoided if possible because it requires that some accounts must have long term administrative access, unless a sophisticated solution is used to add and remove accounts from privileged groups on an on-demand basis. This approach also increases the risk of introducing untested changes to the production environment. These changes have to be backported to source control later and vetted by service owner test validation. This could easily result in a cascade of trouble if it turns out that the change does cause a new issue to occur.

Whenever possible, routine tasks on servers that will be maintained as long term deployments should be automated. An example would be patch management using a script run through an automation platform such as Azure Automation and Hybrid Runbook Worker. This provides a systemic approach to routine tasks where every step is scripted and never deviates from a known procedure, eliminating human error from execution. The scripts used to perform the work and the administration of the platform such as scheduling should be stored in source control and maintained along with project files such as the latest DSC configuration for the nodes.

Connecting to Windows Server remotely for servicing, whenever possible, should be performed through a PowerShell Just Enough Administration[56] endpoint. Just Enough Administration, or JEA, provides a technical solution for limiting which commands, parameters, and parameter values are available for administrative tasks. JEA runs in the context of a temporary administrative account, so the account used to run the commands does not require administrative access at all. Since JEA configurations are delivered using DSC it is easily integrated into the release pipeline model and is just another aspect of the machine configuration. JEA should be used for automated routine tasks, too. The automation platform should connect to remote nodes and perform work using a JEA endpoint. JEA allows

automation to perform specific administrator tasks without actually granting administrative privileges to the automation service accounts. For more information about JEA, see Just Enough Administration, Step by Step[57].

Implementations

This section provides a high level overview of real world environments that demonstrate how organizations have implemented a release pipeline (or key components) for managing core infrastructure using configuration as code. Shared learnings from these organizations were influential to the authors of this document.

Stack Overflow

The Stack Overflow question and answer platform is an ASP.NET MVC application backed by a SQL Server AlwaysOn Availability Group. There are several Redis servers providing shared cache and pub/sub services. An ElasticSearch cluster supports search on the site. There are also several machines providing tag lookup and correlation. All of the essential functions have redundant servers.

Web application code changes go out on an irregular schedule, whenever code changes are ready. Application changes and database changes are deployed independently. The web application has been written to work with older or newer versions of the database. While most of the web servers are upgraded in a close time frame, there can be periods of time where multiple versions of the deployed application are available across the web farm. Feature toggles prevent exposure of new features until they are ready to release from a business perspective. Many of the new features in Stack Overflow have been on the production web servers for months, waiting to be called into action.

If you are interested in more details about the Stack Overflow infrastructure, please take a look at the excellent blog posts of Nick Craver, a developer and sysadmin at Stack Overflow.

Turn 10 Studios

Owner and cloud service provider for the popular video game series Forza Motorsport, and Forza Horizon, Turn 10 Studios uses configuration as code patterns and practices to optimize change delivery throughout their hosting environment. PowerShell DSC is used to manage web servers that provide an online multiplayer game environment for millions of racing fans.

Operations managing the cloud service behind Forza Motorsports use a variety of community tools in their release system, notably from the PowerShell.org Github repo. The tooling includes an automated build of DSC artifacts, and configuration data at scale is managed through a series of cascading data files.

Testing all modules and configurations is absolutely critical to the success of the service and allows Turn10 to administer an environment under high-demand with only minimal operations staff.

Closing

The release pipeline model offers IT operations a framework to help resolve some of the most complicated issues in service management. It is important for engineers to understand that a process change, while supported by new and interesting tools, does not mean a wholesale departure from existing skillsets and the tools that are already deployed. Just the opposite; process change offers an opportunity to provide value to your organization by applying your existing skills and tools to with new process that helps the business to move faster, introduce change in small increments more quickly but with less risk, and reduce operational fatigue.

Long term, a release pipeline will create ubiquity in deployment and configuration models. By standardizing the deployment and configuration of servers, operations and developers can care less about individual servers and more about the service(s) the server is delivering. Applications and application data become the focus. This leads to an environment where servers are treated as a commodity.

The ultimate goal is to improve the value that IT operations can provide to their organization by moving faster, delivering more, and at the same time improving their own quality of life by increasing confidence and decreasing randomization.

The solutions in this document intentionally refer to how the release pipeline model can be applied to Windows Servers and the Microsoft Cloud environment, but the ideas in this document originated from many of the external references given in the introduction. Be on the lookout for new and interesting tools. The community is constantly publishing open source tools solutions that can serve as very powerful components of your operations architecture.

[1] Murawski, Steven. "DevOps Reading List," *Can you DevOp with Windows?* Blog. http://stevenmurawski.com/devops-reading-list/

[2] "Building a Release Pipeline with Team Foundation Server 2012," Microsoft Patterns & Practices eBook. https://msdn.microsoft.com/en-us/library/dn449957.aspx

[3] "Azure Resource Manager REST API Reference," *Microsoft Azure product documentation.* https://msdn.microsoft.com/en-us/library/azure/dn790568.aspx

[4] "Microsoft Azure Stack," *Microsoft product landing page.* https://azure.microsoft.com/en-us/overview/azure-stack/

[5] Gray, Mark. "Configuration in a DevOps world – Windows PowerShell Desired State Configuration," *Building Clouds Blog*. https://blogs.msdn.microsoft.com/powershell/2013/11/01/configuration-in-a-devops-world-windows-powershell-desired-state-configuration/

[6] "Microsoft Deployment Toolkit," *Microsoft Windows product documentation*. https://technet.microsoft.com/en-us/windows/dn475741.aspx

[7] Willis, Rob. "Deployment—Introducing PowerShell Deployment Toolkit," *Building Clouds Blog*. https://blogs.technet.microsoft.com/privatecloud/2013/02/08/deploymentintroducing-powershell-deployment-toolkit/

[8] Wikipedia contributors, "Single source of truth," *Wikipedia, The Free Encyclopedia,* https://en.wikipedia.org/w/index.php?title=Single_source_of_truth&oldid=707024994.

[9] "Azure Automation overview," *Microsoft Azure documentation*. https://azure.microsoft.com/en-us/documentation/articles/automation-intro/

[10] "Team Foundation Server," *Microsoft product landing page.* https://www.visualstudio.com/en-us/products/tfs-overview-vs.aspx

[11] Git web site. http://www.git-scm.com/

[12] GitHub web site. https://github.com/

[13] "Windows PowerShell Integrated Scripting Environment (ISE)," *Windows PowerShell product documentation.* https://technet.microsoft.com/en-us/library/dd819514.aspx

[14] "Visual Studio Code," *Microsoft product landing page*. https://code.visualstudio.com/

[15] "Windows PowerShell Desired State Configuration Overview," *Windows PowerShell product documentation.* https://msdn.microsoft.com/en-us/powershell/dsc/overview

[16] "Deploy Azure resources using .NET libraries and a template," *Microsoft Azure product documentation*. https://azure.microsoft.com/en-us/documentation/articles/arm-template-deployment/

[17] "Test and Deploy with Confidence," Travis CI open source testing platform, home page. https://travis-ci.org/

[18] "Jenkins," Jenkins open source automation server, home page. http://jenkins-ci.org/

[19] "TeamCity," TeamCity continuous integration tools, home page. http://www.jetbrains.com/teamcity/

[20] "Welcome to the PSake project," *PSake project page on GitHub*. PSake is a build automation tool written in PowerShell. https://github.com/psake/psake

[21] "GNU Make," *GNU Operating System documentation*. Build tool. https://www.gnu.org/software/make/

[22] "RAKE—Ruby Make," *Rake project page on GitHub*. Rake is similar to Make; implemented in Ruby. https://github.com/ruby/rake

[23] "Operations Manager," *Microsoft System Center documentation*. https://technet.microsoft.com/en-us/library/hh205987.aspx

[24] "Start Maintenance Mode," *Microsoft System Center product documentation*. https://technet.microsoft.com/en-us/library/hh531754.aspx

[25] "Automation Runbooks," *Service Management Automation documentation*. https://technet.microsoft.com/en-us/library/dn441448.aspx

[26] "Azure Automation Hybrid Runbook Workers," *Microsoft Azure documentation*. https://azure.microsoft.com/en-us/documentation/articles/automation-hybrid-runbook-worker/

[27] "Pester," *GitHub project home page*. Pester provides a framework for running unit tests to execute and validate PowerShell commands. https://github.com/pester/Pester/wiki/Pester
[28] "Operation-Validation-Framework," *PowerShell Team GitHub project home page*. A set of tools for executing validation of the operation of a system. https://github.com/PowerShell/Operation-Validation-Framework
[29] Serverspec project home page. http://serverspec.org/
[30] RSpec project home page. "Behavior Driven Development for Ruby." http://rspec.info/
[31] Inspec project home page. "An open-source testing framework for infrastructure …" https://www.chef.io/inspec/
[32] Test Kitchen home page. https://www.chef.io/inspec/
[33] Palermo, Jeffrey. "Guidelines for Test-Driven Development," *Visual Studio 2005 Team System, Technical Articles, MSDN Library*. https://msdn.microsoft.com/en-us/library/aa730844(v=vs.80).aspx
[34] "Using MOF Files," *Appendix B: Windows Management Instrumentation documentation*. https://technet.microsoft.com/en-us/library/cc180827.aspx
[35] Microsoft Visual Studio Release Management product page. https://www.visualstudio.com/en-us/features/release-management-vs.aspx
[36] "Understanding Microsoft Release Management," *Visual Studio ALM and DevOps, Release*. MSDN. https://msdn.microsoft.com/library/vs/alm/release/getting-started/understand-rm
[37] "PowerShellGet Module," *Windows PowerShell 5.0 documentation*. https://technet.microsoft.com/en-us/library/dn835097.aspx
[38] "Azure Automation DSC Overview," *Microsoft Azure Automation documentation*. https://azure.microsoft.com/en-us/documentation/articles/automation-dsc-overview/
[39] "Azure Service Endpoints," *Microsoft Azure, Java Developer Center documentation*. https://azure.microsoft.com/en-us/documentation/articles/azure-toolkit-for-eclipse-azure-service-endpoints/
[40] "What is CFEngine?" *CFEngine product documentation*. https://cfengine.com/learn/what-is-cfengine/
[41] "Chef Delivery," *product landing page*. https://www.chef.io/delivery/
[42] "PowerShell Team," *GitHub repository*. https://github.com/powershell
[43] AppVeyor product home page. Continuous delivery service for Windows. http://www.appveyor.com/
[44] PowerShell.org. Forums, blogs, and other resources devoted to PowerShell. http://powershell.org/wp/
[45] "TeamCity," *product landing page*. Continuous integration platform. https://www.jetbrains.com/teamcity/
[46] "How Configuration Passes Work," *How Windows Setup Works, Windows product documentation*. https://technet.microsoft.com/en-us/library/cc749307(v=ws.10).aspx
[47] "Start-DSCConfiguration," *Windows PowerShell 5.0 product documentation.* https://technet.microsoft.com/en-us/library/dn521623.aspx
[48] "Configuring the Local Configuration Manager," *Windows PowerShell 5.0 product documentation.* https://technet.microsoft.com/en-us/library/dn521623.aspx
[49] "Microsoft Operations Management Suite." https://www.microsoft.com/en-us/server-cloud/operations-management-suite/overview.aspx
[50] "Group Policy for Beginners," *TechNet technical article*. https://technet.microsoft.com/en-us/library/hh147307(v=ws.10).aspx
[51] Packer product home page. "Packer is a tool for creating machine and container images for multiple platforms from a single source configuration." https://www.packer.io/
[52] Boxstarter home page. Boxstarter is a tool to automate Windows installations. http://www.boxstarter.org/
[53] "Windows Server Update Services," *WSUS product home page*. https://technet.microsoft.com/en-us/windowsserver/bb332157.aspx
[54] "Deployment Imaging Servicing Management (DISM) Cmdlets in Windows Powershell," *DISM product documentation*. https://technet.microsoft.com/en-us/library/hh852126.aspx
[55] Fowler, Chad. "Trash Your Servers and Burn Your Code: Immutable Infrastructure and Disposable Components." Blog post. http://chadfowler.com/blog/2013/06/23/immutable-deployments/

[56] "Just Enough Administration (JEA)." PowerShell product documentation. https://msdn.microsoft.com/en-us/powershell/wmf/jea_overview
[57] Greene, Michael. "Just Enough Administration, Step by Step." Blog post, whitepaper. https://blogs.technet.microsoft.com/privatecloud/2014/05/14/just-enough-administration-step-by-step/