**Introduction to Configuration as Code**

This module is about automatically configuring your infrastructure to meet the requirements for your application.

In the Provision infrastructure in Azure Pipelines  module, you learned how *infrastructure as code* enables you to describe, through code, the infrastructure you need for your application.

*Configuration as code* is a similar concept. However, instead of describing the infrastructure itself, with configuration as code you describe how your infrastructure needs to be configured to run your application or service.

Ever received an emergency support call in the middle of the night because of a crashed server? Then you know the pain of searching through multiple spreadsheets, or even your memory, to access the manual steps of setting up a new machine. There's also the difficulty of keeping the development and production environments consistent. An easier way to remove the possibility of human error when initializing machines is to configure them automatically through code. By treating configuration as code, you can configure many systems identically from a single consistent definition.

In this module, you continue your journey with the Tailspin team, as they add automatic configuration tasks to their CI/CD pipeline.

**Learning objectives**

In this module, you'll:

* Learn what we mean by *configuration as code* and discover the kinds of tools you can use to configure your systems.
* Apply an Ansible playbook that configures service accounts on a Linux virtual machine running on Azure.
* Create a pipeline in Azure Pipelines that configures your infrastructure when the pipeline runs.

**Prerequisites**

The modules in this learning path form a progression.

To follow the progression from the beginning, be sure to first complete these learning paths:

* Evolve your DevOps practices
* Build applications with Azure DevOps
* Deploy applications with Azure DevOps

We also recommend you start at the beginning of this learning path: Automate your deployments with Azure DevOps .

If you want to go through just this module, you need to set up a development environment on your Windows, macOS, or Linux system. You need:

* An Azure DevOps organization
* An Azure subscription
* A GitHub  account
* Visual Studio Code
* Git

You can get started with Azure and Azure DevOps for free. You don't need an Azure subscription to work with Azure DevOps, but here you'll use Azure DevOps to configure Azure resources that exist in your Azure subscription.

This environment lets you complete the exercises in this and future modules. You can also use it to apply your new skills to your own projects.

# What is configuration as code?

* 8 minutes

Configuration as code enables you to describe the configuration you need to run your application or service. In this section, you learn some of the important concepts that relate to configuration management, and how configuration management tools work.

In Provision infrastructure in Azure Pipelines , you used Terraform to provision the Space Game website on Azure. Tim's happy with how Terraform and other infrastructure tools make it easy to scale out. He can see how having a single configuration file that's kept in version control simplifies his life. He can control the deployment environments just by editing a file and running it through the pipeline. Everything feels much better organized, it's easy to keep track of when changes happen, and, if there's a problem, he can always roll back to the last version of the configuration file until he figures out what went wrong.

Although the Space Game web application runs on Azure App Service, Tim also maintains a growing number of virtual machines (VMs) that support billing and other functions. Tim is wondering how automation can help him get control of all the VMs that he needs to manage. His network is getting too large to maintain them on a server-by-server basis.

Andy has offered to help. Tim sets up a meeting with him at Andy's favorite coffee shop.

## The meeting

**Tim:** Terraform is working great for the Space Game website.

**Andy:** Yes. Management is really impressed at how we've been able to scale out our deployments.

**Tim:** You thought automation could help me configure VMs, too. They need a lot of attention. There's always something that needs to be updated, and I've got a big problem with configuration drift.

**Andy:** What's that exactly?

**Tim:** Configuration drift is where servers differ more and more as time goes on. The state of the machine deviates, or drifts, from the baseline because of manual changes and updates.

I used to think configuration drift was inevitable and just a part of running a datacenter. Now, I suspect it's because a lot of changes are rough, cobbled-together solutions. They might be put in place to fix an immediate problem, and they're not even written down. I think automation can fix the problem.

I've been impressed with what Terraform does for automatic provisioning on Azure. I want to do the same to define how our infrastructure is configured.

Andy nods.

**Andy:** I get it. I think what you're talking about here is configuration as code. Configuration as code enables you to describe the configuration you need to run your application or service. With it, you describe the packages, user accounts, firewall rules, and so on, that your application needs.

Here's a short video where Abel Wang, Cloud Advocate at Microsoft, explains the concepts of configuration as code.

**Ask Abel**

Just like the Terraform plan you built in the Provision infrastructure in Azure Pipelines  module, you typically check in your configuration code to source control, along with your application code. Doing so enables you to version, build, test, configure, and deploy your applications as a unit.

## What is configuration management?

Configuration management is a term that's associated with configuration as code. Configuration management refers to the automated management of configuration, typically in the form of automated scripts or programs that you run regularly on your deployments.

For example, say you have dozens or even hundreds of systems that serve the same or similar function, and you need each of them to be configured in the exact same way. How might you open a new firewall port on each system? You might connect to each system remotely and open the port. However, that task is time-consuming and prone to errors. A better way might be to modify an existing configuration management script. Then, through automation, you apply that script to your fleet. Doing so takes much less time and is less prone to error.

Comparing manual configuration processes to those that are more automated, automated processes typically enable you to:

* Reproduce bugs or inconsistencies more easily.
* More easily trace and audit changes over time.
* Configure many related systems more consistently and with fewer errors.
* Deploy more often and at greater scale.

Configuring your systems through automation also frees you from the need to document the differences among systems. Through automation, each system is configured in the exact same way. Your configuration code documents the requirements and the expected results.

Configuration as code helps you implement a configuration management strategy. A good configuration management strategy includes running automated tools to keep all of your systems configured the exact same way. When you run configuration tools in your CI/CD pipeline, such as Azure Pipelines, you ensure that your infrastructure is always configured the way you need it as you build out new features.

## What is configuration drift?

Configuration drift happens when resources change over time from their original deployment state. This is caused by changes made by people, processes, or programs, and can happen manually or through automated processes.

Eventually, an environment can become a snowflake. A snowflake is a unique configuration that can't be reproduced automatically, and is typically a result of configuration drift. Inconsistency among environments can lead to unexpected issues during deployment. With snowflakes, infrastructure administration and maintenance typically become manual processes, which can be hard to track and are prone to human error. The more an environment drifts from its original state, the more likely it is for an application to encounter issues. The greater the degree of configuration drift, the longer it takes to troubleshoot and fix issues.

### Security considerations

Configuration drift can also introduce security vulnerabilities into your environments. For example:

* Firewall ports can be opened that should be kept closed.
* Updates and security patches might not be applied across environments consistently.
* Software might be installed that doesn't meet compliance requirements.

While eliminating configuration drift entirely can be difficult, running a configuration management tool can help you manage it.

## How does configuration as code relate to infrastructure as code?

The concept of configuration as code is similar to the concept of infrastructure as code. In fact, you can combine the two models to automatically provision and then configure your systems all in one step.

For example, your CI/CD pipeline might use an Azure Resource Manager template, Terraform, or other automated provisioning process to bring up your infrastructure. Then, you might run Chef, Puppet, Ansible, or other configuration tool to set up your infrastructure with everything your application or service needs to run. You'll learn about these tools shortly.

## How do configuration management tools work?

Soon, you'll learn about some of the configuration management tools that you can use. But first, there are a few concepts you should understand. They are:

* Idempotency
* Imperative code versus declarative code
* Agent model versus agentless model
* Push model versus pull model

### Idempotency

You learned about idempotency in Provision infrastructure in Azure Pipelines . As a refresher, an idempotent operation is one that provides the same result each time you apply it. Idempotency is a term that's used in both mathematics and in computer science.

As with infrastructure as code tools such as Azure Resource Manager templates and Terraform, most configuration as code tools are also idempotent. Although we didn't yet discuss the various tools you can use, consider the following configuration written with Chef:

RubyCopy

powershell\_script 'Install IIS' do

action :run

code 'Add-WindowsFeature Web-Server'

end

service 'w3svc' do

action [ :enable, :start ]

end

template 'c:\inetpub\wwwroot\Default.htm' do

source 'Default.htm.erb'

rights :read, 'Everyone'

end

On Windows, this configuration:

* Installs Internet Information Services (IIS) web server.
* Starts the World Wide Web Publishing Service (W3SVC), the service that powers IIS.
* Sets the contents of the default homepage, c:\inetpub\wwwroot\Default.htm.

When you run this configuration for the first time, Chef ensures that IIS is installed, W3SVC is running, and that the home page is set. When you run this configuration a second time, Chef applies the configuration only if the desired state doesn't match. In other words, Chef applies the configuration only when there's work to do. In most cases, Chef makes no changes because the system is already in the desired state, so this configuration is safe to apply as many times as you'd like.

### Imperative versus declarative code

In Provision infrastructure in Azure Pipelines , you learned the difference between imperative and declarative code.

Generally speaking, imperative code defines both what the program should accomplish as well as how to achieve the result.

In contrast, declarative code defines what the program should accomplish, but does not define how to achieve the result.

Think about how you write code in languages like Python, C#, or Java. Each of these languages is imperative. You use variables, conditional statements, and loops to control the state of your program and express the steps you need.

Compare these languages to HTML, which is declarative. HTML describes what elements appear on the page, but it doesn't describe how to display them. The "how" is the web browser's responsibility.

Just like infrastructure as code tools such as Azure Resource Manager templates and Terraform, most configuration management tools use the declarative code model. Consider this very basic Chef configuration:

RubyCopy

package 'vim'

On Linux, this configuration ensures that the Vim text editor is installed. It declares that the **vim** package should be installed, but it doesn't specify how to install it. Chef understands the Linux distribution that's running, and can use the appropriate package manager to install the package. For example, Chef might use **apt** on Debian or Ubuntu systems, and **yum** on Red Hat Enterprise Linux or CentOS systems.

### Nodes and hosts

Most configuration management systems use the term node to refer to any device or machine under management. A managed node can also be referred to as a host.

Although the term node often refers to virtual machines and physical machines, it can refer to any kind of system you can manage, including network routers and switches.

### Agent model versus agentless model

In configuration management, an agent is a background process, such as a service or daemon, that listens for configuration requests or applies configuration changes on a schedule. The agent runs on each node under management. Some agents also report data to a central configuration server or database, so that administrators can track configuration changes over time.

Chef and Puppet, for example, use agents to apply configuration changes and report back on what changes were made. Ansible doesn't use agents. Rather, Ansible connects to Linux machines over the SSH protocol (and Windows machines over WinRM), and uses Python to apply configuration changes.

### Push model versus pull model

In a push model, a central configuration server pushes, or sends, the latest configuration requirements to each node under management.

In a pull model, each node under management pulls, or requests, the latest configuration changes from a central configuration server.

Generally speaking:

* The push model offers you greater control than the pull model, but requires you to schedule maintenance runs. For example, if a node is being provisioned or rebooted, you might need to reschedule the run to happen later, when the node is available.
* The pull model scales better than the push model, but typically requires agent software to be installed on each node under management.

As you evaluate the various configuration management tools, you should understand which models that tool supports. That way, you can best plan how and when you schedule maintenance runs on your nodes.

# What configuration tools can I use?

* 10 minutes

In this part, you learn about these popular configuration management tools:

* Ansible
* Azure Automation
* Azure Custom Script Extension
* Chef
* Cloud-init
* PowerShell DSC
* Puppet

For each tool, you'll get a general sense for how it works, which programming languages are involved, and how it integrates with Azure. At the end, Andy and Tim will choose one to try.

## Ansible

Ansible is an open-source product, sponsored by Red Hat, that automates cloud provisioning, configuration management, and application deployments. You can use Ansible to provision Azure resources such as virtual machines, containers, networks, and even complete cloud infrastructures. You can also use Ansible to configure your Azure resources after they're provisioned, which is the focus in this module.

In addition to Azure, Ansible supports other public clouds as well as private cloud frameworks.

With Ansible, you write playbooks that express your desired configuration. A playbook is a YAML file, making it a form of declarative automation.

A playbook is made up of modules. Ansible provides built-in module types that configure various parts of the system. For example, the file module enables you to manage a file. The user module enables you to manage a user account.

Here's a basic example that you'll work with later. It defines service accounts for users named testuser1 and testuser2.

ymlCopy

---

- hosts: all

become: yes

tasks:

- name: Add service accounts

user:

name: "{{ item }}"

comment: service account

create\_home: no

shell: /usr/sbin/nologin

state: present

loop:

- testuser1

- testuser2

You use the ansible-playbook command to apply this configuration, like this:

BashCopy

ansible-playbook \

--inventory ./azure\_rm.yml \

--user azureuser \

--private-key ~/.ssh/ansible\_rsa \

--limit=tag\_Ansible\_test1 \

./users.yml \

You'll see this process in greater detail later in this module.

If you run this command multiple times, Ansible configures the user accounts only if they don't exist or have changed. This command is therefore an idempotent operation.

Ansible is also agentless, so you don't have to install Ansible software on the managed machines. However, you do need to install Python on your managed machines. By default, Ansible connects to Linux machines over the SSH protocol, and Windows machines over WinRM.

You typically use a control machine to manage your systems. A control machine includes the Ansible software and the playbooks you need to run. The control machine pushes configuration changes to your nodes. Later in this module, you'll set up a control machine and run Ansible playbooks from that control machine in Azure Pipelines.

Although Ansible is agentless, both the control machine and managed nodes require Python to enable Ansible to connect to remote systems and issue commands on those systems.

### Ansible inventories

In Ansible, the inventory is a file that defines the hosts upon which the tasks in a playbook operate. Ansible represents what systems it manages by using an .ini or YAML file that puts all of your managed machines in groups of your own choosing.

For a VM deployment on Azure, you could define each VM and its IP address or hostname, similar to this:

ymlCopy

hosts:

vm1:

ansible\_host: 13.79.22.89

vm2:

ansible\_host: 40.87.135.194

This is an example of a static inventory. If these IP addresses change, or if you add or remove systems, you'd need to update this inventory file over time.

A more flexible approach is to use a dynamic inventory. A dynamic inventory enables Ansible to discover which systems to configure at runtime.

Here's the dynamic inventory file you're going to use in this module:

ymlCopy

plugin: azure\_rm

include\_vm\_resource\_groups:

- learn-ansible-rg

auth\_source: auto

keyed\_groups:

- prefix: tag

key: tags

This inventory specifies that each VM in the learn-ansible-rg resource group belongs to the inventory. The keyed\_groups part groups VMs by their tag names. You'll work with a complete example later in this module.

### Ansible on Azure

There are a number of ways you can use Ansible on Azure.

On [Azure Marketplace](https://azuremarketplace.microsoft.com/marketplace), you'll find a number of images that you can use. They include:

* [Red Hat Ansible instance on Linux](https://azuremarketplace.microsoft.com/marketplace/apps/redhat.ansible-tower?tab=Overview), published by Microsoft.

You can use this image to bring up a control machine, which includes Ansible, the Azure CLI, and other tools, to manage your fleet.

* [Ansible Tower](https://azuremarketplace.microsoft.com/marketplace/apps/redhat.ansible-tower), published by Red Hat.

Ansible Tower helps organizations scale IT automation and manage complex deployments across physical, virtual, and cloud infrastructures. With Ansible Tower, you can:

* + Provision Azure environments using pre-built Ansible playbooks.
  + Use role-based access control (RBAC) to define who or what can see, change, or delete objects, or utilize specific capabilities.
  + Maintain centralized logging for complete auditability and compliance.
  + Use the many content resources available on Ansible Galaxy.

You can also set up Ansible on a Linux VM running on Azure, or in your datacenter, and use that as your control machine. Although Ansible doesn't support Windows as the control machine, you can run Ansible from Windows through Windows Subsystem for Linux, Azure Cloud Shell, or Visual Studio Code.

## Azure Automation

Azure Automation is a service in Azure that helps you automate manual tasks. Automation has the concept of a runbook, which is a set of tasks that perform some automated procedure in Automation. Tasks in a runbook are written in PowerShell, [PowerShell Workflow](https://docs.microsoft.com/en-us/system-center/sma/overview-powershell-workflows), or Python. You can run a runbook either manually or on a schedule.

Here's a basic example that uses PowerShell Workflow to stop a running service:

PowerShellCopy

Workflow Stop-MyService

{

$Output = InlineScript {

$Service = Get-Service -Name MyService

$Service.Stop()

$Service

}

$Output.Name

}

Although the name implies that you can use Azure Automation only on Azure, it's more flexible than that. Automation has a feature called hybrid runbook worker. This feature gives Automation access to resources in other clouds or in your on-premises environment that would otherwise be blocked by a firewall.

Automation also provides a Desired State Configuration (DSC) pull server that enables you to create definitions for how a specified set of VMs should be configured. DSC then ensures that the required configuration is applied and that the VM stays consistent. Automation DSC runs on both Windows and Linux.

## Azure Custom Script Extension

The Custom Script Extension is a way to download and run scripts on your Azure VMs. You can run the extension when you create a VM, or any time after the VM is in use.

You can store your scripts in Azure Storage or in a public location, such as GitHub. You can run scripts manually or as part of a more automated deployment.

You can use the Custom Script Extension with Windows or Linux VMs. Here's an example that uses the az vm extension set command to run a Bash script that installs Nginx web server on a Linux VM.

Azure CLICopy

az vm extension set \

--resource-group my-rg \

--vm-name my-vm \

--name customScript \

--publisher Microsoft.Azure.Extensions \

--version 2.1 \

--settings '{"fileUris":["https://raw.githubusercontent.com/MicrosoftDocs/mslearn-welcome-to-azure/master/configure-nginx.sh"]}' \

--protected-settings '{"commandToExecute": "./configure-nginx.sh"}'

## Chef

Chef is an infrastructure automation tool that enables you to configure and manage your systems.

Chef helps you to manage your infrastructure in the cloud, on-premises, or in a hybrid environment. You express your configurations by writing recipes that describe everything your systems need to run your application. Chef recipes use a declarative syntax that's based on the Ruby programming language. A recipe uses the .rb file extension.

A Chef recipe is made up of resources. Chef provides built-in resource types that enable you to configure various parts of the system. For example, the package resource enables you to install or remove a package. The service resource enables you to manage a service.

Here's the Chef recipe that installs Internet Information Services (IIS) web server on Windows, which you saw earlier in this module:

RubyCopy

powershell\_script 'Install IIS' do

action :run

code 'Add-WindowsFeature Web-Server'

end

service 'w3svc' do

action [ :enable, :start ]

end

template 'c:\inetpub\wwwroot\Default.htm' do

source 'Default.htm.erb'

rights :read, 'Everyone'

end

Most Chef resources are idempotent, meaning you can apply the same configuration repeatedly.

You can package multiple recipes into a cookbook. A cookbook might contain recipes that configure the various parts of MySQL, Nginx, OpenSSL, or any other kind of software.

Building on the previous code example, an IIS cookbook might contain recipes that configure application pools, virtual directories, and virtual sites. You can define roles to specify which recipes are applied to a system based on that system's function. For example, you might define the "webserver" role to run recipes that install and configure IIS, Apache, or Nginx web servers. The "database" role might run recipes that install and configure MySQL or Microsoft SQL Server.

Chef and the Chef community maintain cookbooks on [Chef Supermarket](https://supermarket.chef.io/).

### Chef on Azure

There are a number of ways you can use Chef on Azure.

On [Azure Marketplace](https://azuremarketplace.microsoft.com/marketplace), you'll find a number of images that you can use. They include:

* [Chef Extension](https://docs.chef.io/azure_portal/) for Windows and Linux, published by Chef Software.

These images come with the Chef Client. Chef Client is an agent that runs on each node that's managed through Chef. Chef Client applies the cookbooks and recipes you specify. Chef Client can also send reporting data back to a Chef Server or a Chef Automate server, so that you can track and audit your configuration runs over time.

* [Chef Automate](https://azuremarketplace.microsoft.com/marketplace/apps/chef-software.chef-automate), published by Chef Software.

Chef Automate enables you to package and test your applications, and provision and update your infrastructure. Using Chef, you can manage all of it with compliance and security checks, and dashboards that give you visibility into your entire stack.

You can also set up Chef on a Linux or Windows VM running on Azure, or in your datacenter.

### Cloud-init

Cloud-init, by Canonical, is a way to customize a Linux VM as it boots for the first time. You can use cloud-init to install packages, write files, and configure users.

You write cloud-init files by using YAML. Consider this basic cloud-init configuration that installs PIP, the package manager for Python, and NumPy, a package for scientific computing with Python.

ymlCopy

#cloud-config

packages:

- python-pip

runcmd:

- pip install numpy

In this example, packages specifies the list of packages to install. Here, we install **python-pip**. runcmd specifies the list of commands to run on first boot. Here, we use PIP to install the NumPy package.

This configuration is declarative, meaning you don't need to specify how to install **python-pip**. Cloud-init recognizes the Linux distribution that's running, and can use the appropriate package manager to install the **python-pip** package. For example, it can install **apt** on Debian-based systems or **yum** on Red Hat Enterprise Linux.

Here's an example that uses the Azure CLI to bring up an Ubuntu VM on Azure and apply this configuration.

Azure CLICopy

az vm create \

--resource-group my-rg \

--name my-vm \

--admin-username azureuser \

--image UbuntuLTS \

--custom-data cloud-init.txt \

--generate-ssh-keys

The --custom-data argument specifies the cloud-init configuration to run when the VM boots for the first time. The cloud-init configuration is located in cloud-init.txt.

## PowerShell DSC

PowerShell Desired State Configuration (DSC) is a management platform that defines the configuration of target machines. You can use PowerShell DSC to manage Windows or Linux systems.

DSC configurations define what to install and configure on a machine. A local configuration manager (LCM) engine runs on each target node that processes requested actions based on pushed configurations. A pull server is a web service that runs on a central host to store the DSC configurations and associated resources. The pull server communicates with the LCM engine on each target node to provide the required configurations and report on compliance.

Here's a basic example that uses PowerShell DSC to configure IIS on Windows.

PowerShellCopy

Configuration MyWebsite {

Import-DscResource -ModuleName PsDesiredStateConfiguration

Node "localhost" {

WindowsFeature WebServer {

Ensure = "Present"

Name = "Web-Server"

}

}

}

You would then compile this configuration into a Management Object Format (MOF) file, which is the format that DSC can consume. To do this, you run the configuration like a function. Here's an example:

PowerShellCopy

. .\MyWebsite.ps1

MyWebsite

The first line makes the configuration function available in the console. The second line runs the configuration. The result is a folder, named MyWebsite, which contains a file named localhost.mof.

To apply the configuration, you run the Start-DscConfiguration cmdlet, like this:

PowerShellCopy

Start-DscConfiguration .\MyWebsite

## Puppet

Puppet is an automation platform that handles the application delivery and deployment process. Agents are installed on target machines to allow Puppet Master to run manifests that define the desired configuration of your infrastructure.

You express your configurations by writing manifest files known as Puppet Program files. Manifests describe everything your systems need to run your application. Puppet manifests use a declarative syntax that's based on the Ruby programming language. A manifest uses the .pp file extension.

A Puppet manifest is made up of resources. Puppet provides built-in resource types that enable you to configure various parts of the system. For example, the file resource enables you to manage a file. The service resource enables you to manage a service.

Here's a basic Puppet manifest that installs IIS web server on Windows:

RubyCopy

$iis\_features = ['Web-WebServer']

iis\_feature { $iis\_features:

ensure => 'present'

}

You can package multiple manifests into a module. Puppet and their partners maintain modules at [Puppet Forge](https://forge.puppet.com/).

In this example, the iis\_feature resource is provided by the [puppetlabs-iis](https://forge.puppet.com/puppetlabs/iis?azure-portal-true) module, which helps you manage IIS sites and application pools.

### Puppet on Azure

There are a number of ways you can use Puppet on Azure.

On [Azure Marketplace](https://azuremarketplace.microsoft.com/marketplace), you'll find a number of images that you can use. They include:

* [Puppet Agent](https://azuremarketplace.microsoft.com/marketplace/apps/Puppet.puppet-agent-windows-arm), published by Puppet, is a virtual machine extension that installs the Puppet agent on your Windows VM.
* [Puppet Enterprise](https://azuremarketplace.microsoft.com/marketplace/apps/puppet.puppet-enterprise-201818), published by Puppet, enables you to automate the entire lifecycle of your infrastructure.

You can also set up Puppet on a Linux or Windows VM running on Azure, or in your datacenter.

## The decision

In [Provision infrastructure in Azure Pipelines](https://docs.microsoft.com/en-us/learn/modules/provision-infrastructure-azure-pipelines), the Tailspin team chose Terraform to provision Azure resources. Now they need to make a decision on which configuration management tool to use to configure their virtual machines. Let's join the conversation.

Andy looks at Tim.

**Andy:** What do you think? Do you have any preferences?

**Tim:** I like many of these options. Can we continue to use Terraform to configure our systems?

**Andy:** Terraform is great for building infrastructure. It's designed, out of the box, to interact directly with cloud providers, to help you plan that infrastructure, and then to build it. However, Terraform is not a configuration management tool. That said, you can use Terraform to build your infrastructure, and then use what's called a provisioner to configure your infrastructure by using your favorite tools.

Tim thinks for a minute.

**Tim:** Let's go with Ansible. Maybe we can tie it in with Terraform at some point, but for now, I'd like to see it working and then add it to Azure Pipelines. Perhaps we can build a basic prototype, and then later we can add in our details.

**Andy:** Sounds great. Let's get to work.