

# Scripting cloud tasks

12 minutes

The concept of scripting has been around almost as long as computers. For decades, IT workers have interacted with computers by typing commands into a console. To this day, some users prefer a command-line interface (CLI) to a graphical user interface (GUI). Scripting allows otherwise independent commands to be merged into one or more text files ("scripts") and executed as a group. Scripting languages such as Bash allow scripts to do more than blindly execute sequences of commands; scripts can include statements such as `if` and `for` that execute commands conditionally or repeatedly, they can solicit input from users, and they can include symbolic variables.

The simplest way to automate cloud deployments is to assemble the commands required to create and configure a set of resources into a script. Popular cloud platforms support command-line interfaces (CLIs) and cloud shells (CLIs accessed through a browser) featuring commands that can be used in these scripts. Azure, for example, offers the `az` command, while AWS supports the `aws` command and Google supports `gcloud`.

## Scripting languages

A scripting language is a programming language used to automate the execution of tasks. There is no shortage of scripting languages to choose from; Wikipedia lists almost 200 of them<sup>1</sup>. Not all of them are applicable to cloud platforms. From the standpoint of an administrator whose goal is to automate activities involving cloud resources, scripting languages generally fall into three categories:

- General-purpose scripting languages such as Bash
- Special-purpose scripting languages such as PowerShell
- Fully featured programming languages such as Python that can also be used for scripting

General-purpose scripting languages offer abilities for branching, looping, and decision making, but they rely on operating-system commands or commands from add-ons such as cloud CLIs to do the bulk of the work. Bash, for example, doesn't include commands for creating VMs in AWS or Azure, but it can invoke CLI commands that do. And a single Bash script can include any number of CLI commands. A command in a Bash script is no different than a command an administrator would type at the command line. The following Bash script, for example, uses the Azure CLI's `az` command to create a resource group in Azure and an Ubuntu VM inside the resource group:

Bash

 Copy

```
az group create --name cmu-rg --location eastus

az vm create --resource-group cmu-rg \
  --name cmu-vm --image UbuntuLTS \
  --admin-username cmu-admin \
  --admin-password Micr0s0ft** \
  --location eastus
```

Another scripting tool that is popular among cloud administrators is PowerShell. Microsoft developed PowerShell to be the latest generation of the Windows command line, and also a more secure alternative to the Windows Script Host (WSH) originally developed for Windows Server. It has since become an open-source project and works on Windows and Linux.

Rather than rely on operating-system commands, PowerShell provides its own commands and its own terminology. Most every PowerShell command, or *cmdlet* (pronounced "command-let", defined as a little program unto itself) is an English-language verb attached to an object with a hyphen. For example, `Get-Location` to list the contents of the current directory, `Copy-Item` to copy a file, and `Start-Process` to launch a program. The PowerShell language avoids the use of opaque abbreviations such as Linux's `sudo` in favor of human-legible, if somewhat long, phrases.

What makes PowerShell unique is that a language that was originally intended for use with individual computers and on-premises networks has been extended to work with cloud platforms. Although it's possible to simply include Azure CLI instructions in a PowerShell script, Microsoft offers a module called *Azure PowerShell* that extends PowerShell with Azure-specific cmdlets. Amazon offers a similar module called *AWS Tools for PowerShell*. Here is a PowerShell script that is equivalent to the Bash script above:

PowerShell

 Copy

```
$VMAdmin = "cmu-admin"
$VMPassword = ConvertTo-SecureString "Micr0s0ft**" -AsPlainText -Force

$Cred = New-Object System.Management.Automation.PSCredential `
  ($VMAdmin, $VMPassword)

New-AzResourceGroup -Name cmu-rg -Location EastUS

New-AzVm -ResourceGroupName "cmu-rg" `
  -Name "cmu-vm" `
  -Location "East US" `
  -Image UbuntuLTS `
  -Credential $Cred
```

Administrators can also use full-featured programming languages such as Python to create and manage cloud resources. These languages don't include cloud-specific features of their own, but they can incorporate libraries and software development kits (SDKs) that do. Most major cloud service providers offer free SDKs for interfacing with their platforms in various languages. Microsoft, for example, offers Azure SDKs for C#, Go, Java, Python, Node.js, and other popular programming languages, as does AWS. Even without SDKs, programmers can use the REST APIs offered by major cloud platforms to invoke cloud services. SDKs and libraries simply make writing such programs easier.

As an example, AWS offers a Python SDK named Boto containing Python functions that closely mirror AWS CLI instructions. The syntax of these methods is adapted to suit the native style of Python. The following Python script uses Boto to create an EC2 instance in Amazon's East 1 region:

Python	 Copy
<pre>import boto3  try:     ec2 = boto3.client('ec2', region_name='us-east-1')     key_pair = ec2.create_key_pair(KeyName='ec2-key-pair')      ec2.run_instances(         ImageId='ami-00b6a8a2bd28daf19',         InstanceType='t2.micro',         KeyName='ec2-key-pair',         Placement='AvailabilityZone=us-east-1d',         MinCount=1,         MaxCount=1)  except ClientError as e:     print(e)</pre>	


One of the benefits to using a programming language for scripting is robust support for error handling and debugging. The downside, of course, is that you need programmers well versed in the language of choice and knowledgeable about cloud platforms and cloud SDKs.

## Bash scripting


Bash, short for "Bourne-again shell," is among the world's most popular scripting languages. It is an irreplaceable tool for system administrators and cloud administrators alike. It enjoys native support in Linux and mac OS and is available for Windows as well. Moreover, it is supported in cloud shells such as the Azure Cloud Shell.

The following Bash commands, typed one by one into the Azure CLI or Cloud Shell, create a resource group in Azure's East US region and an Ubuntu VM to go in it. These are identical to


the `az` commands above except for the `--generate-ssh-keys` switch, which replaces login passwords with public/private key pairs for added security:

Bash	 Copy
<pre>az group create --name cmu-rg --location eastus  az vm create --resource-group cmu-rg \ --name cmu-lamp-vm --image UbuntuLTS \ --admin-username lamp-admin \ --generate-ssh-keys \ --location eastus</pre>	

Once the VM is created, an administrator can connect to it securely via Secure Shell (SSH) using this command, where `IP_ADDRESS` is the public IP address assigned to the VM and output from the `az vm create` command:


Bash	 Copy
<pre>ssh lamp-admin@IP_ADDRESS</pre>	

Inside the SSH session, the administrator can use the following commands to install a LAMP (Linux/Apache/MySQL/PHP) stack in the VM and then terminate the SSH session:

Bash	 Copy
<pre>sudo apt update sudo apt install lamp-server^ exit</pre>	

That's a lot of steps, especially if you need to execute them not just once, but several times. As an alternative to entering these commands manually, an administrator might elect to script them by creating a text file named `ccvm.sh` (short for "create and configure VM") containing the statements in Figure 2.

The script doesn't merely recite the commands an operator would type at the console. Note the call to the `az vm show` command to get the VM's public IP address and assign it to the variable named `$IP_ADDRESS`, the use of redirection to `(<<)` to inject commands into the SSH session, and the addition of command-line switches (for example, the `-y` in `sudo apt -y install`) to suppress confirmation prompts so the script can run from start to finish without human intervention:

Bash	 Copy
------	--

```

az group create --name cmu-rg --location eastus

az vm create --resource-group cmu-rg \
--name cmu-lamp-vm --image UbuntuLTS \
--admin-username lamp-admin \
--generate-ssh-keys


IP_ADDRESS=$(az vm show --show-details \
--resource-group cmu-rg --name cmu-lamp-vm \
--query publicIps -o tsv)

ssh -o 'StrictHostKeyChecking no' lamp-admin@$IP_ADDRESS << EOF
    sudo apt update
    sudo apt -y install lamp-server^
    exit
EOF

```

*Figure 2: Bash script for creating an Azure VM and installing a LAMP stack.*

With this script in hand, an administrator could create a VM and install a LAMP stack with a single command:

cmd	 Copy
bash ccvm.sh	

The benefit of this approach is that many commands are reduced to one command, all but eliminating any chance for human error. But it is far from perfect. Each time it is executed, the script attempts to create the same VM and the same resource group in the same Azure region (East US). If the goal is to be able to quickly and easily create identical VMs in  $n$  different regions, then the script has to be smarter.

We could address this modifying the script to include a variable specifying the Azure region and modifying the variable each time we run the script to target a different region. Alternatively, we could place the region name in an environment variable and reference the environment variable in the script, or, better still, pass the region name in a command-line parameter. But we would immediately encounter other problems, such as the fact that you can't create two VMs or two resource groups with the same name in two different regions. This is solvable, too. We could, for example, append the region name to the VM name and the resource-group name to generate unique names for each region.

The point is that scripting cloud CLI commands delivers a degree of automation, but in practice, scripts can quickly grow complex and become more an exercise in programming than in administration. Here are four reasons that scripting alone is not an ideal solution for automating the management of cloud resources:

- **Complexity** -- Scripts are invariably more complex than the equivalent commands typed into a console, primarily because they must do everything a human operator would do and anticipate every condition that a human might encounter. In addition, scripts need to be parameterized so that one script, for example, can deploy the same solution to different regions, or so that an administrator running the script can specify resource names and ensure that they are unique.
- **Idempotency** -- Scripts require additional logic if they're to be *idempotent*, which means running the same script twice produces the same result. What should happen, for example, if a script tries to create a resource group but that resource group already exists? Scripts can (and should) be robust enough to account for circumstances such as these, but robustness comes at the expense of increased length and complexity. Idempotency is an important requirement if scripts are to update existing cloud resources as well as provision new ones.
- **Cloud specificity** -- The script shown in Figure 2 requires heavy modification -- in fact, rewriting -- to work with other cloud platforms such as AWS and GCP. The script could be written to run one way for Azure, another way for AWS, and yet another way for GCP (perhaps driven by an environment variable or command-line parameter), but the script would necessarily become more complex.
- **Version control** -- Scripts are programs and should be submitted to version control just like application code. Otherwise, how do you know when a script was modified, and why, and roll back to a previous version if the current version breaks? This adds a layer of administrative complexity as well as cost.

An added complication is that the usefulness of such scripts depends on how well they mesh with one another, and how they behave in an environment in which they might be running at the same time. When scripts are written and deployed on an ad-hoc basis (as is often the case) without proper planning and forethought, there's often no time or opportunity to test them in a safe environment where they don't affect services already in production. This is when it begins to become obvious that coordination is needed at a much higher level, involving not just IT platforms and infrastructure but the people who maintain and manage them.

So, what's supposed to provide simplification frequently ends up generating greater complexity. The good news is that this isn't the end of the story. In the next lesson, we introduce the notion of a data-center environment where coordination takes place as a matter of principle, and infrastructure is provisioned and updated based on explicitly defined requirements.

## References

1. Wikipedia. Scripting Languages.

[https://en.wikipedia.org/wiki/Category:Scripting\\_languages](https://en.wikipedia.org/wiki/Category:Scripting_languages).

## Check your knowledge

1. Which of the following statements regarding scripting languages such as Bash and PowerShell is NOT true?

- ☐ Scripts that provision resources in multiple cloud environments (for example, Azure and AWS) are inherently more complex than scripts that target a single cloud environment.

- ☒ One of the chief limitations of scripting languages is that their operation cannot be parameterized using command-line parameters. ✓

**Correct! Scripts CAN be parameterized using command-line parameters and frequently are.**

- ☐ Provisioning cloud resources with scripts is more reliable than provisioning them manually by typing commands into a console or clicking buttons in a cloud portal.
- ☐ Scripts can be written to allow cloud resources to be updated as well as provisioned, but they must be specially written to do so, and this adds complexity.

2. In the context of scripting, which of the following scenarios demonstrates idempotency?

- ☐ Each time it's executed, a script writes a record of the resources it created to a machine log.
- ☐ Each time it's executed, a script verifies that each of the resources it attempted to create was indeed created and logs any failures that occurred.
- ☐ A single script, run twice, creates one virtual machine the first time it's executed and a copy of that virtual machine when it's executed again.

- ☒ A single script, run twice, creates a virtual machine the first time it's executed and uses the existing virtual machine the second time. ✓

**Correct. In the context of scripting, idempotency means that the running the same script twice produces the same result.**

---

## Next unit: Infrastructure as code

Continue >

---