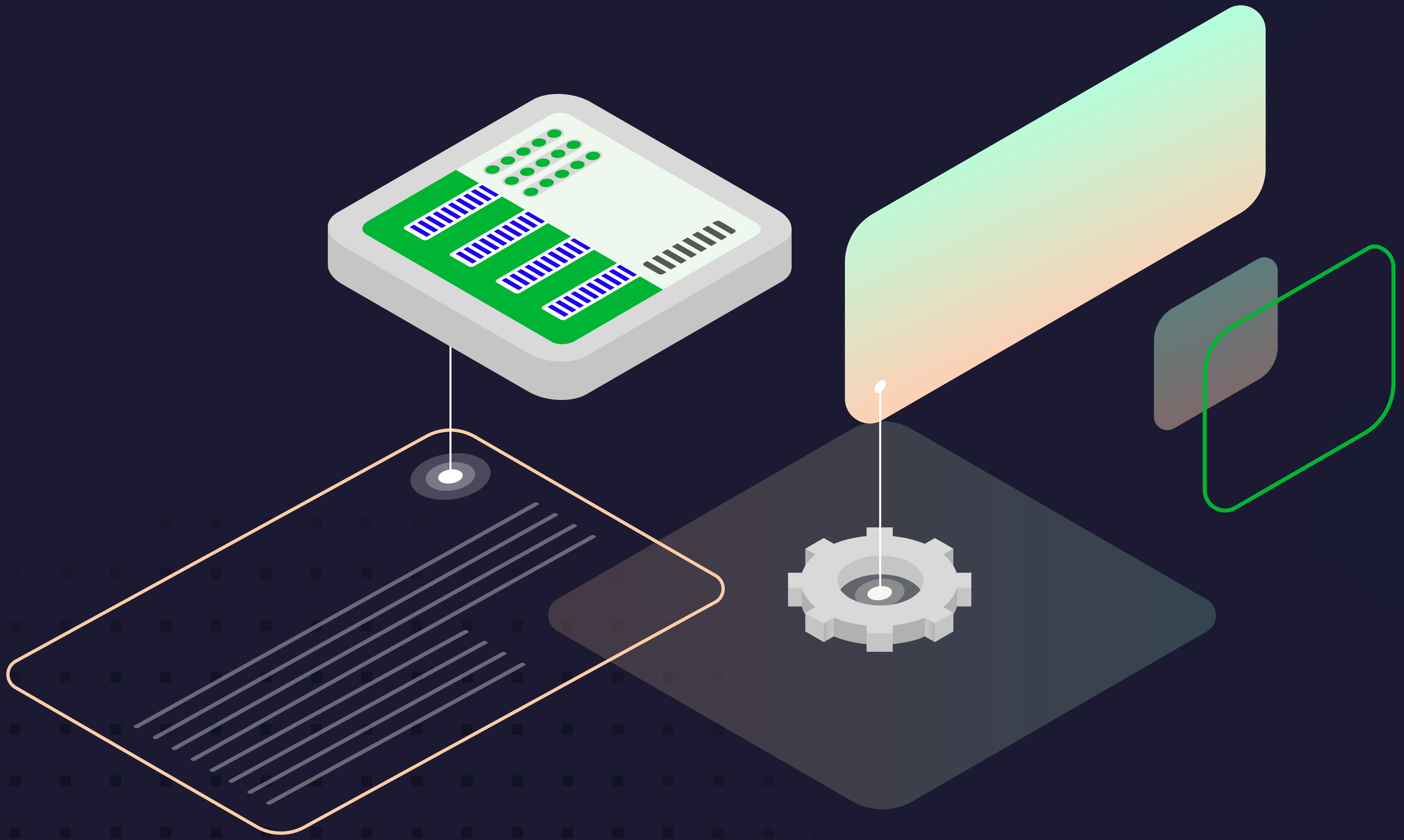
INTERCEPT

WHITEPAPER



Infrastructure as Code with Azure:

a practical guide to automation and CI/CD

Introduction

Welcome!

As cloud computing continues to evolve, so do the methods for deploying and managing Azure resources, offering options for every need and preference. Whether you are just starting your journey with Azure or an experienced user seeking to optimise your workflows, understanding the different methods available for resource deployment is helpful.

The Azure Portal is one of the most accessible and widely used options. Serving as the initial touchpoint for many users, it offers an intuitive graphical interface, making it easy to explore and manage resources, especially for beginners. The portal offers a visual experience that is particularly beneficial for initial learning and troubleshooting, guiding users through the process of setting up and managing resources.

However, as your Azure environment grows, relying on the portal, often called 'ClickOps' can lead to challenges such as configuration inconsistencies and a lack of a centralised documentation system. This is where infrastructure as code (IaC) solutions come into play, offering a more scalable and reliable approach to resource deployment.

In this e-book, we will dive into the various technologies, exploring their unique features, benefits, and use cases. By understanding the strengths and applications of each deployment method, you will be better equipped to choose the right tools to optimise your Azure experience.

Let's get started!

Table of Content

ClickOps vs. Infrastructure as Code (IaC):

When is ClickOps as a manual approach effective?

In this chapter, we shed light on the discussion clickOps vs. Infrastructure as Code (IaC): When does ClickOps work, and when does it not? After reading this chapter, you will understand the challenges and limitations of ClickOps, the key benefits of Infrastructure as Code (IaC), and you will know when to use each graphical interface.

When does ClickOps work, and when does it not?

To answer that question, let's start by asking why we use the Azure Portal in the first place. If you are familiar with the expression *'Friends don't let friends click in the Azure Portal,'* then it might ring a bell. But why? Is it a dislike for graphical interfaces or simply a preference for command-line tools?

Not quite. The real reason is simple: when you first start with Azure, clicking around helps you explore its features and get familiar with the platform. Even as you gain experience, the portal remains useful for quick troubleshooting. More importantly, a good user interface, like the Azure portal, offers a visual experience with instant feedback, which many users appreciate.

The problem with ClickOps

Relying on the Azure Portal to deploy and manage infrastructure manually comes with risks. Let's look at concrete examples of why this can be problematic.

→ Example 1:

Imagine you need to deploy 20 virtual machines (VMs) through the Azure Portal, each requiring identical configurations to maintain consistency across your environment. However, manually setting up each VM significantly increases the risk of human error. A missed setting or accidental misclick can create discrepancies, meaning that while all 20 VMs may be deployed, they might not be identical. These inconsistencies can lead to unforeseen issues, making troubleshooting and maintenance more complicated.

→ Example 2:

Another major issue with ClickOps is the absence of a single source of truth. Manually deploying infrastructure through the Azure Portal leaves no centralised record or script documenting the exact configuration. This makes it challenging to replicate environments or track the current state of your infrastructure. This problem escalates when multiple administrators are involved, each applying their own approach, which can lead to even further inconsistencies.

→ Example 3:

Documentation is another challenge with ClickOps. Manually recording each deployment and configuration step is tedious and error-prone. Without accurate documentation, it becomes difficult to audit changes, track configurations or onboard new team members effectively.

Relying on the Azure Portal to deploy and manage infrastructure manually comes with risks.

The alternative: The introduction of Infrastructure as Code (IaC)

The above examples show that ClickOps does not always provide the right capabilities. But what could be a suitable alternative? One such alternative is Infrastructure as Code (IaC), which offers a more reliable and efficient way to manage your infrastructure.

Tools like Azure Bicep address these challenges by ensuring consistency, providing a single source of truth, and simplifying documentation. Beyond improving reliability, IaC also makes managing and scaling your infrastructure more efficient.

What is Infrastructure as Code?

Infrastructure as Code (IaC) is a method of managing and provisioning computing infrastructure through machine-readable definition files, rather than manually configuring physical hardware or using interactive configuration tools. This approach allows you to automate resource management, monitoring, and provisioning, eliminating the need for manual intervention. Now that we understand what Infrastructure as Code entails, let's look into the potential benefits it brings.

Advantage 1: Scalability in deployments

One of the key advantages of IaC is its ability to scale deployments efficiently. By defining infrastructure through code, you can quickly deploy and manage multiple environments with consistency and minimal errors. This is especially valuable for large-scale apps and environments that require robust scalability. IaC also simplifies environment replication, maintaining consistency across development, staging, and production, and reduces the common *'it works on my machine'* issue.

Advantage 2: Consistency

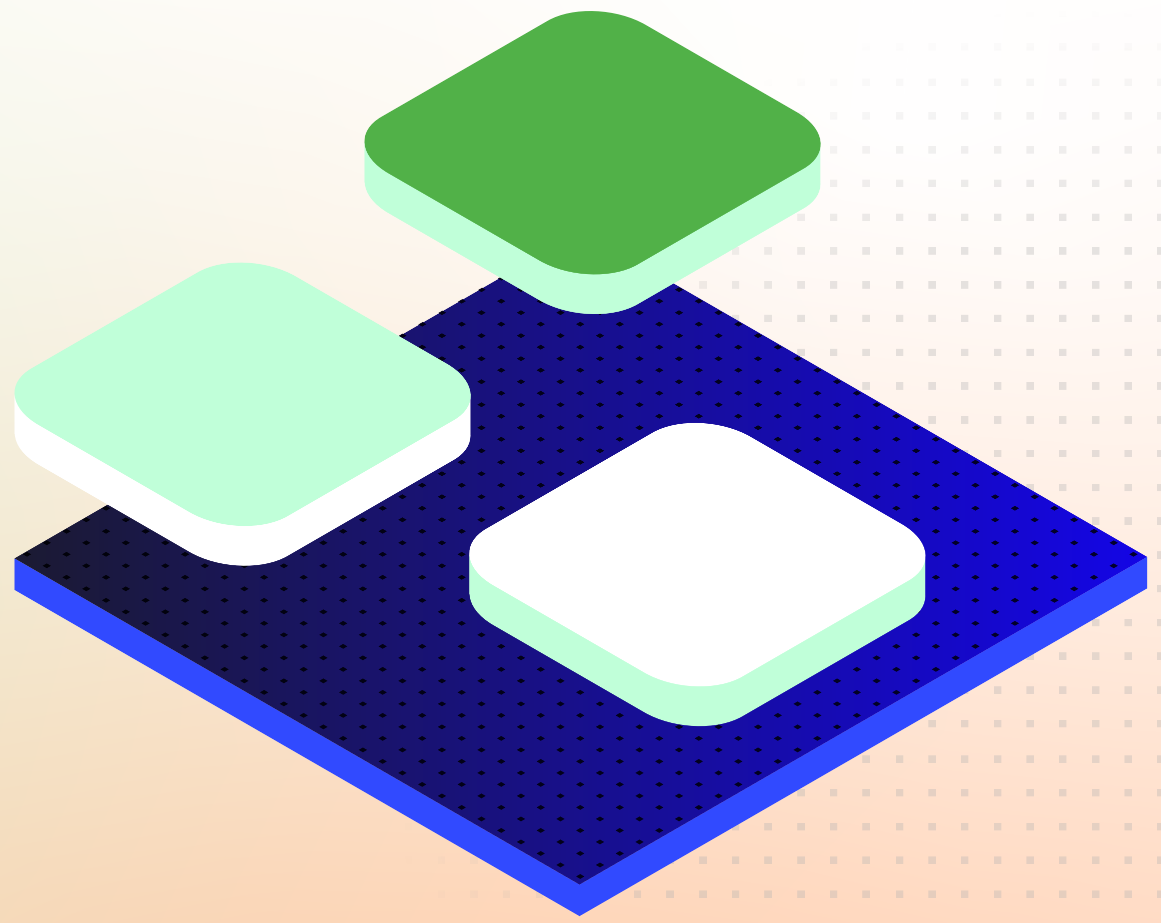
Another key advantage of IaC is consistency. By defining infrastructure as code, you ensure that every deployment is identical, reducing configuration drift and improving reliability. This consistency extends to updates and changes as well: any modifications made to the code are automatically applied across the infrastructure, ensuring all environments remain synchronised.

Coming back to the question: When does ClickOps work, and when does it not?

In this chapter, we compared ClickOps and IaC, outlining their strenghts and limitations. We saw that IaC automates infrastructure management using machine-readable code, offering advantages like scalability, consistency, and reliability. It ensures uniform deployments across multiple environments while reducing errors.

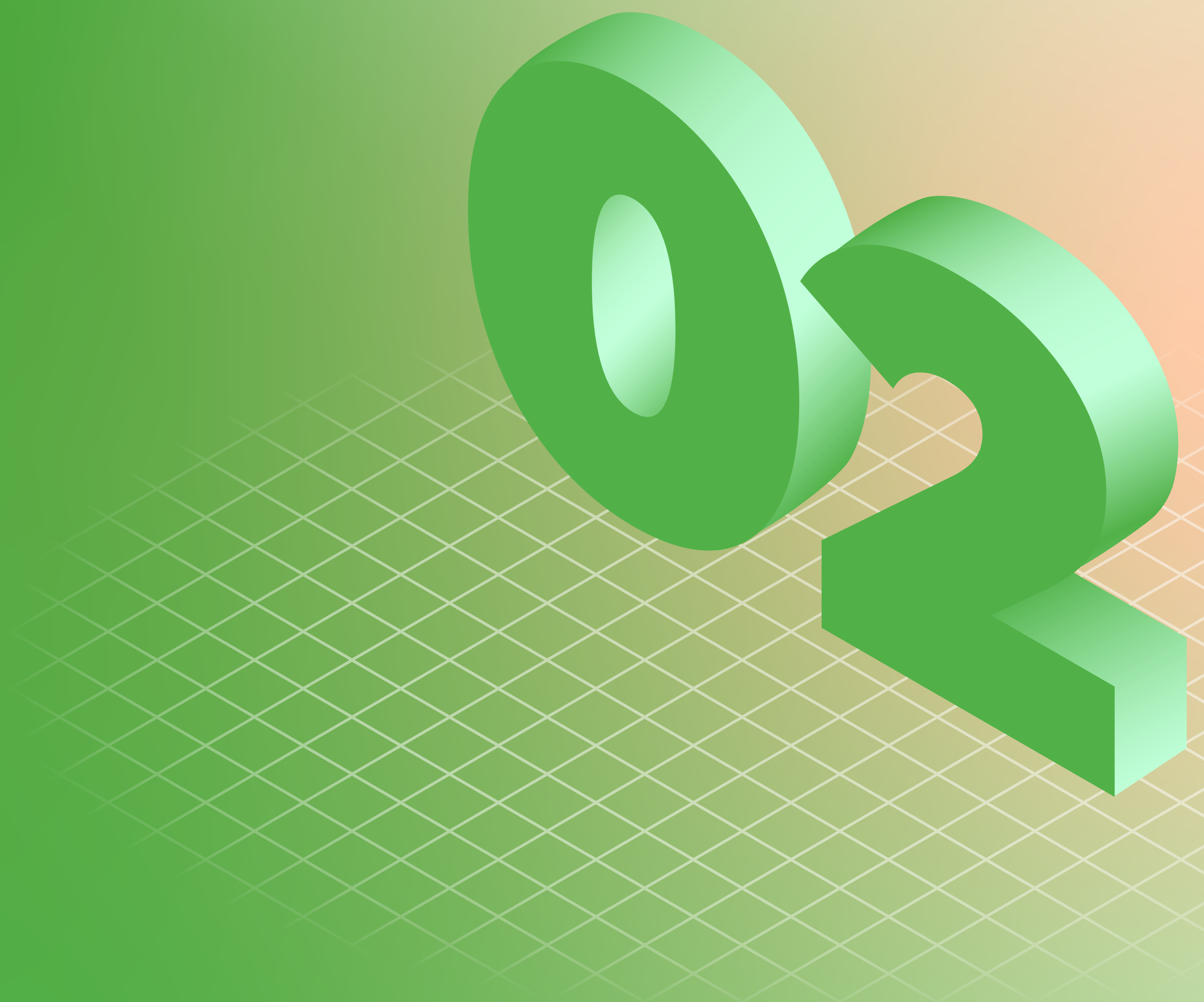
However, while IaC is ideal for large-scale, production environments, ClickOps still has its place. ClickOps, relying on graphical interfaces, is useful for tasks like troubleshooting, investigation, and learning. It lacks however the scalability and repeatability of IaC.

ClickOps therefore complements IaC, offering an intuitive way to manage and explore infrastructure. If you are interested in finding the best IaC language for your needs, keep reading as we will explore that in the next chapter.



CHAPTER 2

Choosing the right Infrastructure as Code language and getting started



In this chapter, we will help you choose the right IaC language, show you how to get started, and explain best practices like version control and automation. By the end, you will know how to pick the best IaC language for your environment, preferences, and project needs. Let's dive in!

How to choose an IaC language?

To answer this question, you will need to follow several steps:

Step 1: Check supported languages in your environment

Before selecting an IaC language, it is important to verify what your environment supports. Here is how to go about it:

→ **Check official documentation:**
Cloud providers regularly update their documentation to list supported IaC tools. For example, Azure officially supports Bicep, Terraform, Pulumi, and Azure Resource Manager (ARM) templates.

→ **Review policy and compliance requirements:**
Some organisations mandate the use of specific IaC languages for security and governance purposes.

→ **Examine existing infrastructure:**
If your team already uses Terraform or ARM templates, it is recommended to align with that choice for consistency.

→ **Experiment in a test environment:**
Set up a simple deployment with different IaC languages to see which one aligns best with your workflow.

Step 2: Choose the right language

The choice of an IaC language (e.g. Bicep, Terraform, or Pulumi) depends on the cloud provider, work environment, and personal preferences. Choosing the right language is the first challenge, as it shapes how you work and determines the scalability of your solutions. Before selecting a language, it is important to:

- **Verify which options** your environment supports. Once you know what's available, you can proceed confidently.
- **Decide on the best fit** based on your cloud provider, organisational needs, and personal preference.
- **Research how to get started** and ensure a smooth transition to writing IaC.

Once you have identified the supported languages, you can explore each option in detail to determine which aligns best with your requirements and workflow.

Different languages:

The languages we will discuss here are Bicep, Terraform, Pulumi, and the declarative language of ARM templates

Language: Bicep

If you are working with Azure, Bicep is an excellent choice. It is a domain-specific language (DSL) designed to simplify ARM templates. Bicep is declarative, meaning you define the desired state, and Azure takes care of the rest. It's easier to read and write than raw JSON-based ARM templates (mentioned below as well). Additionally, Bicep integrates easily with Azure, keeping you closely aligned with Microsoft's ecosystem. The example below demonstrates how to deploy a storage account using Bicep.

```
param location string = ‘East US’
param storageAccountName string = ‘mystorageaccount’

resource storageAccount ‘Microsoft.Storage/storageAccounts@2022-09-01’ =
{
name: storageAccountName
location: location kind: ‘StorageV2’
sku: {      name: ‘Standard_LRS’      }    }
```

Language: Terraform

Terraform is cloud-agnostic, allowing you to define infrastructure across multiple providers, not just Azure. Like Bicep, it is declarative but offers a broader scope. Terraform uses HashiCorp Configuration Language (HCL), which is easy to learn.

If you are working in or planning for multi-cloud environments, Terraform is a strong choice, though some features may require a license. Alternatively, OpenTofu, an open-source option governed by the Linux Foundation, offers similar functionality as fully open resource. The example below deploys a storage account through Terraform.

```
provider “azurerm” {
    features {}
  }

resource “azurerm_storage_account” “storage” {
name                     = “mystorageaccount”
resource_group_name      = “myResourceGroup”
  location                 = “East US”
  account_tier             = “Standard”
  account_replication_type = “LRS”
}
```

Language: Pulumi

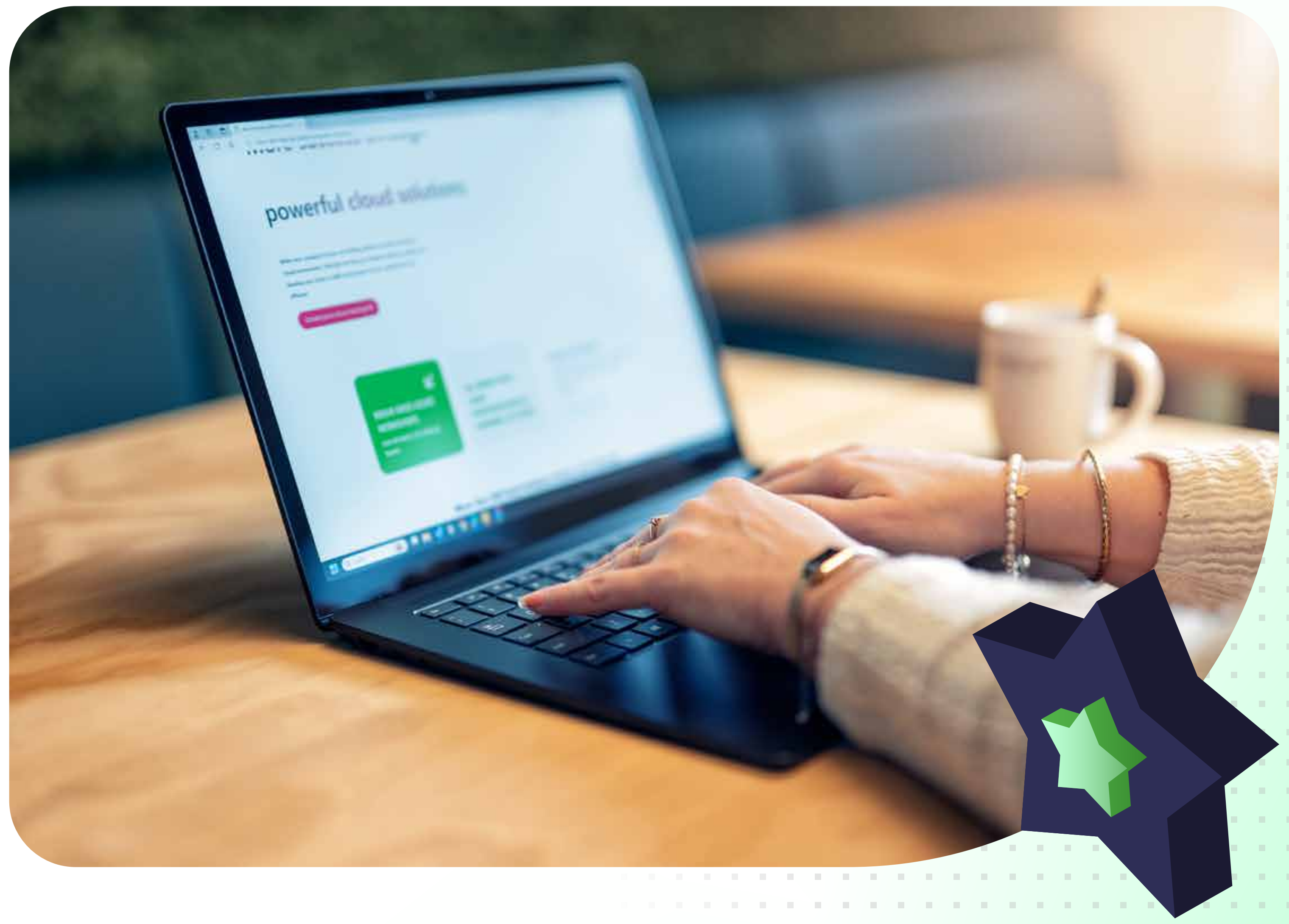
Unlike Bicep and Terraform, Pulumi allows you to write infrastructure using general-purpose programming languages like Python, TypeScript, and C#. This makes it particularly appealing to developers who prefer imperative coding. While Pulumi offers great flexibility, it requires a stronger programming knowledge. The example below deploys a storage account through Pulumi (TypeScript).

```
import * as azure from “@pulumi/azure”;

  const storageAccount = new azure.storage.Account(“storage”, {
   name: “mystorageaccount”,
   resourceGroupName: “myResourceGroup”,
   location: “East US”,
    accountTier: “Standard”,
    accountReplicationType: “LRS”,
 });
```

Declarative language: ARM Templates

ARM templates are JSON-based and used to define Azure resources declaratively.
They offer deep integration with Azure but can be complex to write and maintain manually.
Bicep was introduced as an abstraction layer over ARM templates to simplify the experience.
The example below deploys a storage account through an ARM template.

```
{
 "$schema": "https://schema.management.azure.com/schemas/2019-04-
 01/deploymentTemplate.json#",
 "contentVersion": "1.0.0.0",
 "resources": [
   {
     "type": "Microsoft.Storage/storageAccounts",
     "apiVersion": "2022-09-01",
     "name": "mystorageaccount",
     "location": "East US",
     "sku": {
       "name": "Standard_LRS"
     },
     "kind": "StorageV2"
   }
 ]
}
```



Each of these tools has its strengths. If you work exclusively in Azure, Bicep is often the best choice. For multi-cloud flexibility, Terraform is a great fit. If you prefer using traditional programming languages, Pulumi could be the solution you are looking for.

Step 3: Start writing Infrastructure as Code

Getting started with Infrastructure as Code (IaC) may seem overwhelming but breaking it down into manageable steps will make it more easy. Here are a few tips to help you get started successfully:

Tip 1: Set up your development environment:

- Install the necessary tools and CLI for your chosen IaC language.
- Ensure you have access to an Azure subscription.
- Use a development environment like **Visual Studio Code**, which provides extensive support for IaC languages. Install extensions such as the Bicep Extension for VS Code, Terraform Extension by HashiCorp, or Pulumi Extension to improve syntax highlighting, autocompletion, and deployment capabilities.
- Use the PowerShell Extension terminal in VS Code to run your scripts and deploy infrastructure directly

2. Tip 2: Learn from official resources:

- Microsoft provides excellent learning resources for Bicep. You can complete step-by-step learning modules on Microsoft Learn.
- Terraform and Pulumi also offer extensive documentation and hands-on labs on their respective websites.

Tip 3: Start with simple deployment

- Start with small resources, such as deploying a storage account (as shown in the examples).
- Test your scripts in a sandbox environment before deploying them to production.
- Follow best practices, such as using parameters and modular structures. Also, explore Azure Verified Modules, which support both Bicep and Terraform here.

Tip 4: Use version control and automation

- Store your IaC code in a Git repository.
- Implement Continuous Integration (CI) and Continuous Deployment/ Delivery (CD) (CI/CD) pipelines (e.g., Azure DevOps, GitHub Actions) to automate deployments.
- Validate and test your infrastructure code before applying any changes.

Let's return to the main question: How do you choose the right IaC language and get started?

In this chapter, we have seen that transitioning from manual processes to code can be challenging, but Infrastructure as Code (IaC) is the future. Choosing the right language depends on your environment and objectives:

- If you are working with Azure, Bicep is an excellent choice.
- If you need more flexibility, Terraform or Pulumi may suit you better.
- If you have existing ARM templates, Bicep could simplify your workflow.

As mentioned earlier, it is very important to first assess which languages your environment supports. This ensures your choice aligns with company policies, team workflows, and cloud provider capabilities.

We also discussed the importance of starting small, experimenting, and iterating. As you gain confidence, managing IaC will become as second nature as assembling a server once was. While the days of using a screwdriver may be behind us, the need for precision and structure remains. Now that you have learned how to choose the right IaC language and get started, you are ready for Chapter 3, where we will explore modularity. This is the next logical step, as it enables you to manage your infrastructure in a structured, reusable way. Modularity ensures your infrastructure stays manageable over time, especially as it grows and becomes more complex. Let's go!

Choosing the right language depends on your environment and objectives.

CHAPTER 3

Mastering modularity and reusability in Azure Bicep

In this chapter, we will show you how to create reusable Azure Bicep modules that keep your infrastructure standardised, scalable, and manageable. Embracing modularity will help minimise errors, improve collaboration, and ensure your infrastructure remains well-structured and adaptable.

Let’s dive in!

How to create and manage reusable Azure Bicep modules for scalable and standardised infrastructure?

Let's start with how modules standardise building blocks and simplify complex configurations. Among the many tools available for IaC, Azure Bicep stands out for its simplicity and modularity. Azure Bicep is a domain-specific language (DSL) for deploying Azure resources declaratively, offering a more concise and user-friendly syntax compared to traditional JSON ARM templates. A key feature of Azure Bicep is its support for modules, which allows you to build standardised, reusable building blocks for your infrastructure.

Understanding Azure Bicep and its modularity

Azure Bicep provides a more intuitive way to define and deploy Azure resources, making what was once a complex process with JSON ARM templates much simpler. It eases infrastructure management for both beginners and experienced users. However, the true strength of Azure Bicep lies in its modularity.

Modularity

Modules in Azure Bicep are reusable components that group together specific resources and configurations. They allow you to create standardised building blocks that can be shared across different projects and teams. By defining these modules, you encapsulate the logic needed to deploy specific resources and expose only the necessary parameters. This modular approach helps ensure consistency, reduces duplication, and streamlines the management of complex configurations.

Building standardised building blocks

Modularity in Azure Bicep allows you to create standardised building blocks that can be reused across your organisation. These building blocks can represent anything from a simple storage account to a complex multi-tier application. By defining these components as modules, you ensure that they are consistent and adhere to best practices.

For example, let's consider a module for deploying a virtual network (VNet). This module can include all the necessary configurations for creating a VNet, including subnets, network security groups, and route tables. By encapsulating these configurations in a module, you can easily reuse it in different projects without having to redefine the VNet configuration each time.

In this example, the `vnet.bicep` module contains the logic for deploying a VNet with two subnets. By using this module, you can deploy the VNet consistently across different environments with minimal effort.

```
module vnet 'vnet.bicep' = {
name: 'myVnetModule'
params: {
vnetName: 'myVnet'
addressPrefix: '10.0.0.0/16'
subnets: [
 {
   name: 'subnet1'
   addressPrefix: '10.0.1.0/24'
 },
 {
   name: 'subnet2'
   addressPrefix: '10.0.2.0/24'
 }
     ]
  }
}
```

Simplifying complex configurations

One of the key benefits of using modules in Azure Bicep is the ability to simplify complex configurations. Infrastructure as Code (IaC) can become quite intricate, especially when working with large-scale environments containing numerous interdependent resources. Modules enable you to hide this complexity behind a simplified interface, making it easier for users to deploy and manage infrastructure.

For example, imagine you need to deploy a highly available web application with multiple components, such as virtual machines, load balancers, and databases. Without modules, defining and managing these resources can be overwhelming. However, by encapsulating each component within a module, you can significantly simplify the deployment process.

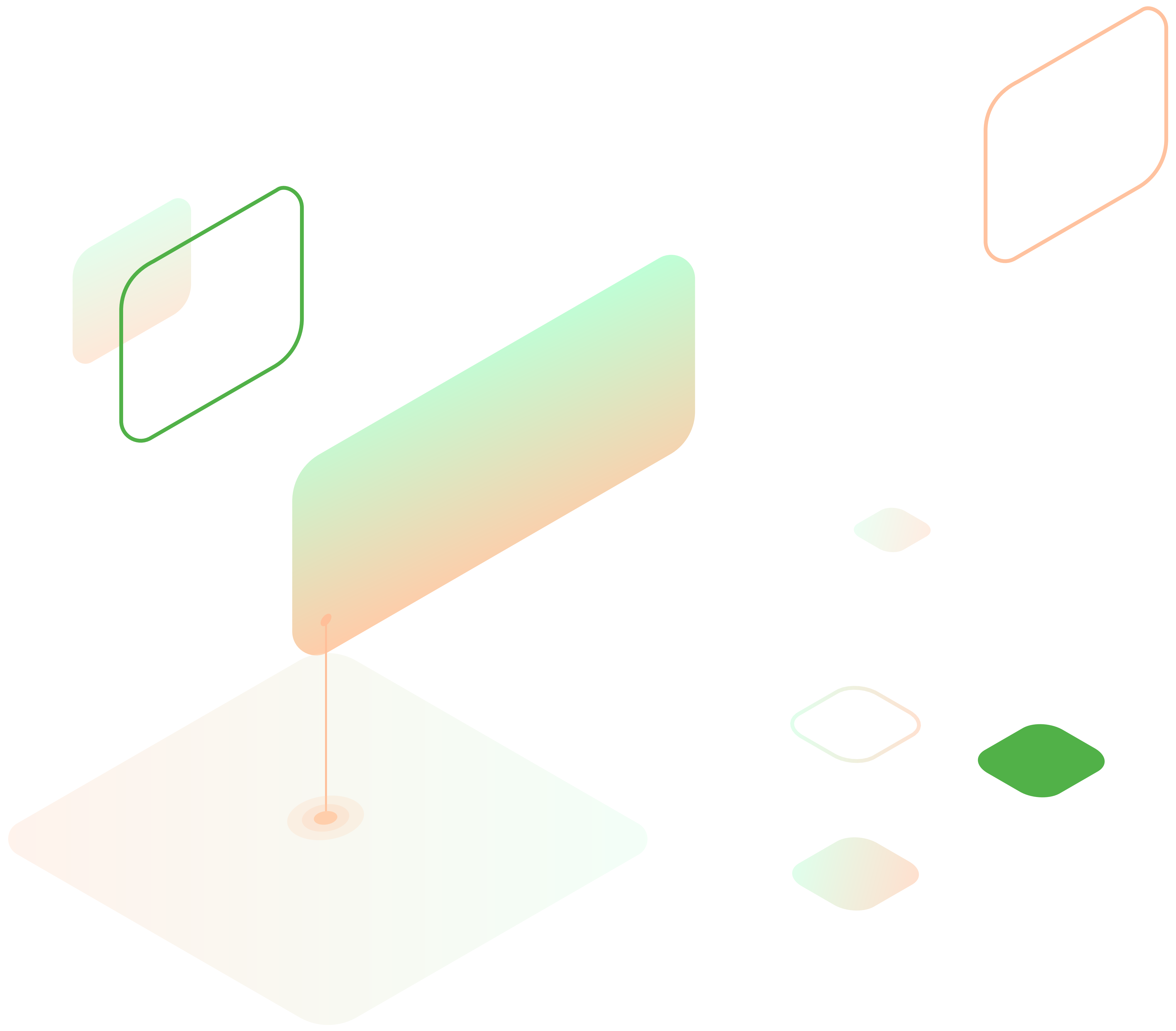
For example, imagine you need to deploy a highly available web application with multiple components, such as virtual machines, load balancers, and databases. Without modules, defining and managing these resources can be overwhelming. However, by encapsulating each component within a module, you can significantly simplify the deployment process.

```
module vm ‘vm.bicep’ = {
 name: ‘myVmModule’
      params: {
            vmName: ‘myVm’
            vmSize: ‘Standard_DS1_v2’
            adminUsername: ‘adminUser’
            adminPassword: ‘P@ssw0rd!’
            }
}
module lb ‘lb.bicep’ = {
 name: ‘myLbModule’
      params: {
            lbName: ‘myLb’
            frontendIpConfiguration: {
            name: ‘myFrontendConfig’
            publicIpAddressId: ‘myPublicIp’
            }
 }
}
module db ‘db.bicep’ = {
 name: ‘myDbModule’
      params: {
            dbName: ‘myDb’
            skuName: ‘Standard_DTU2’
            maxSizeBytes: ‘1073741824’
 }
}
```

Empowering teams with reusable modules

The modularity of Azure Bicep not only simplifies infrastructure management but also empowers teams within your organisation. By having a team of subject matter experts build and maintain the modules, you can ensure best practices are followed and that the modules stay up to date with the latest developments.

Other teams within the organisation can then utilise these modules by simply writing a main deployment file or parameter file, without needing to understand all the intricate details. This approach enables non-experts to deploy complex infrastructure with ease, reducing the learning curve and boosting productivity.



Creating and using modules

To create a module in Azure Bicep, you define it in a separate .bicep file and parameterise it as needed. The module file contains the logic for deploying a specific resource or set of resources. You can then reference this module in your main deployment file, passing in the necessary parameters. For example, let's create a module for deploying a storage account:

// storage.bicep

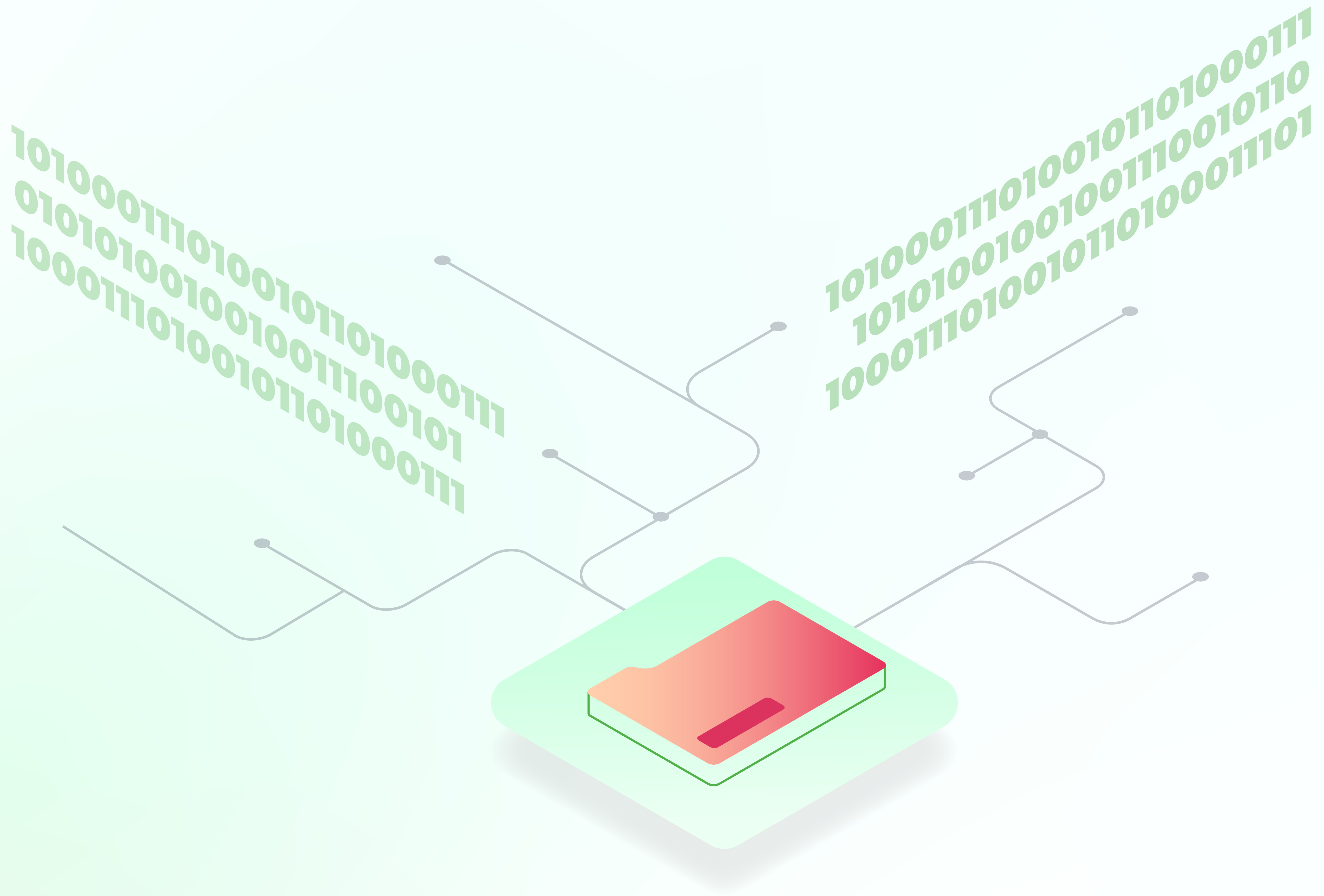
```
resource storageAccount ‘Microsoft.Storage/storageAccounts@2021-01-01’ = {
name: params.storageAccountName
location: resourceGroup().location
sku: {
name: ‘Standard_LRS’
}
kind: ‘StorageV2’
}
// main.bicep
module storage ‘storage.bicep’ = {
name: ‘myStorageModule’
params: {
storageAccountName: ‘mystorageaccount’
}
}
```

By following this approach, you can build a library of reusable modules that can be shared across projects and teams. This fosters consistency and reduces the chances of errors or misconfigurations.

Maintaining and updating modules

Having a dedicated team of subject matter experts to build and maintain the modules ensures they are of high quality and follow best practices. Your team can continuously update the modules as new features and improvements are released, keeping your infrastructure up to date.

By centralising module maintenance, you can quickly address any issues or bugs that arise. This lightens the load on individual teams, allowing them to focus on delivering value to your organisation.



Getting back to the question: How to create and manage reusable Azure Bicep modules for scalable and standardised infrastructure?

In this chapter, we focused on how to build, reuse, and manage Azure Bicep modules to keep your infrastructure standardised, scalable, and easy to manage. We saw how Azure Bicep's modular approach makes complex configurations easier, promoting consistency and scalability across your organisation.

By keeping your modules in one place, teams with different skill levels can deploy and manage infrastructure more easily, which helps improve productivity and teamwork. This approach makes your organisation more flexible and better able to meet business needs.

By centralising module maintenance, teams of all skill levels can deploy and manage infrastructure more efficiently, which helps improve productivity and teamwork. This modular approach makes your organisation more flexbile and better able to meet business needs.

This chapter has equipped you with the tools to build and manage reusable Azure Bicep modules for scalable infrastructure. In the next chapter, we will take it a step further by focusing on optimising these modules, improving efficiency, and making them easier to maintain.

CHAPTER 4

Deploying Infrastructure with Verified Modules and CI/CD

This chapter builds on the previous ones, focusing on how to automatically and reliably deploy infrastructure using Azure Verified Modules (AVM) and CI/CD. AVM and CI/CD (via Azure DevOps or GitHub Actions) make infrastructure management scalable, secure, and repeatable. In this chapter, we will show how AVM and CI/CD can simplify deployment, reduce risks, and save time.

How can AVM and CI/CD be used to automate and scale infrastructure deployment reliably?

Once you have chosen Azure Bicep (because we are passionate about Azure and Infrastructure as Code), you goal is to make it scalable, repeatable, predictable, and, most importantly, manageable. As the saying goes, "Automate, automate, automate" and that's exactly what we will do! By leveraging CI/CD, pipelines, and Azure Verified Modules, we will transform this into a professional, efficient solution. Let's see which steps to follow to make it scalable, repeatable, predictable and manageable.

Challenge: Making it scalable, repeatable, predictable, and manageable

Firstly, let's explore why an organisation might choose to adopt a CI/CD automated solution. A good starting point is to consider the pillars of the WAF and CAF frameworks, which we, as an organisation, should strive to follow.

Using a **main.bicep** file alongside parameters helps ensure these principles are adhered to while allowing certain values to be adjusted via the parameter file. For example, you could modify the Virtual Machine SKU, adjust backup policies, or enable/disable specific services during the development stage.

With CI/CD, you can also validate pull requests using **PSRule.Rules.Azure**, ensuring the **bicepparam** file is checked and validated before deployment. This keeps best practices at the heart of your infrastructure deployments.

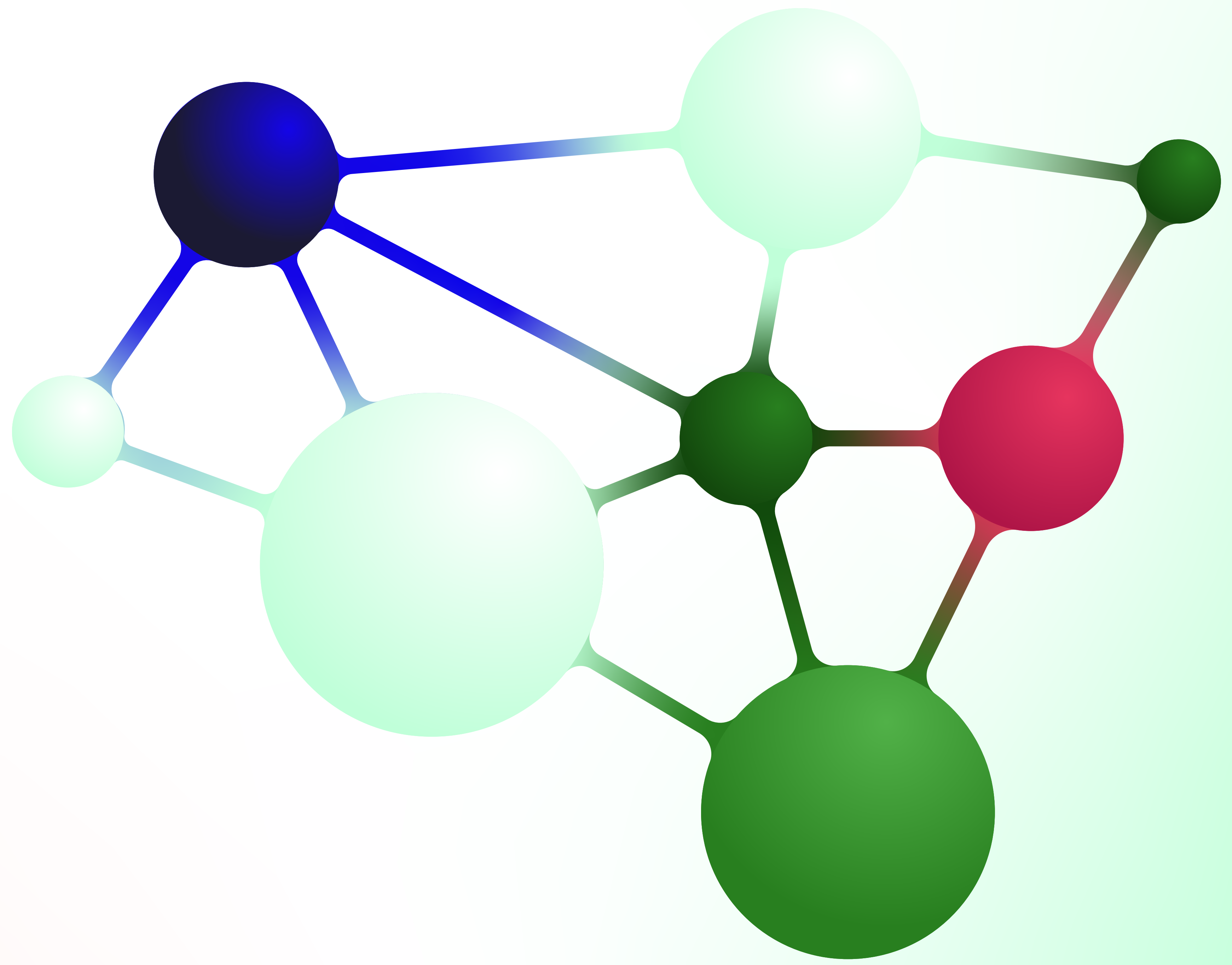
Azure Verified Modules (AVM) provide a standardised, reusable method for deploying Azure resources, ensuring compliance and adherence to best practices. Instead of manually defining resources, AVM Bicep modules simplify deployments by leveraging pre-verified, Microsoft-supported templates. In this capter, we will explore how to deploy Azure infrastructure using **Azure Verified Modules** with **Azure DevOps Pipelines** and **GitHub Actions**.

Why Use Azure Verified Modules (AVM)?

Azure Verified Modules provide:

- **Standardization**: Microsoft-validated templates ensure compliance with best practices.
- **Modularity**: Encapsulate infrastructure components into reusable modules.
- **Security and Compliance:** AVM includes built-in policies and guardrails.
- **Scalability:** Reuse modules across multiple environments and teams.

When working with multiple teams, source control is a must for any organisation as it ensures that code is clean and well-managed. Using a deployment pipeline removes the issue of git sync mismatches or local test code accidentally being pushed to production. Engineers must use pipeline validation before they can push code, ensuring quality control.



Structuring an AVM-Based Bicep deployment

Below is an example folder structure to help keep your CI/CD clean and organised.

Folder structure

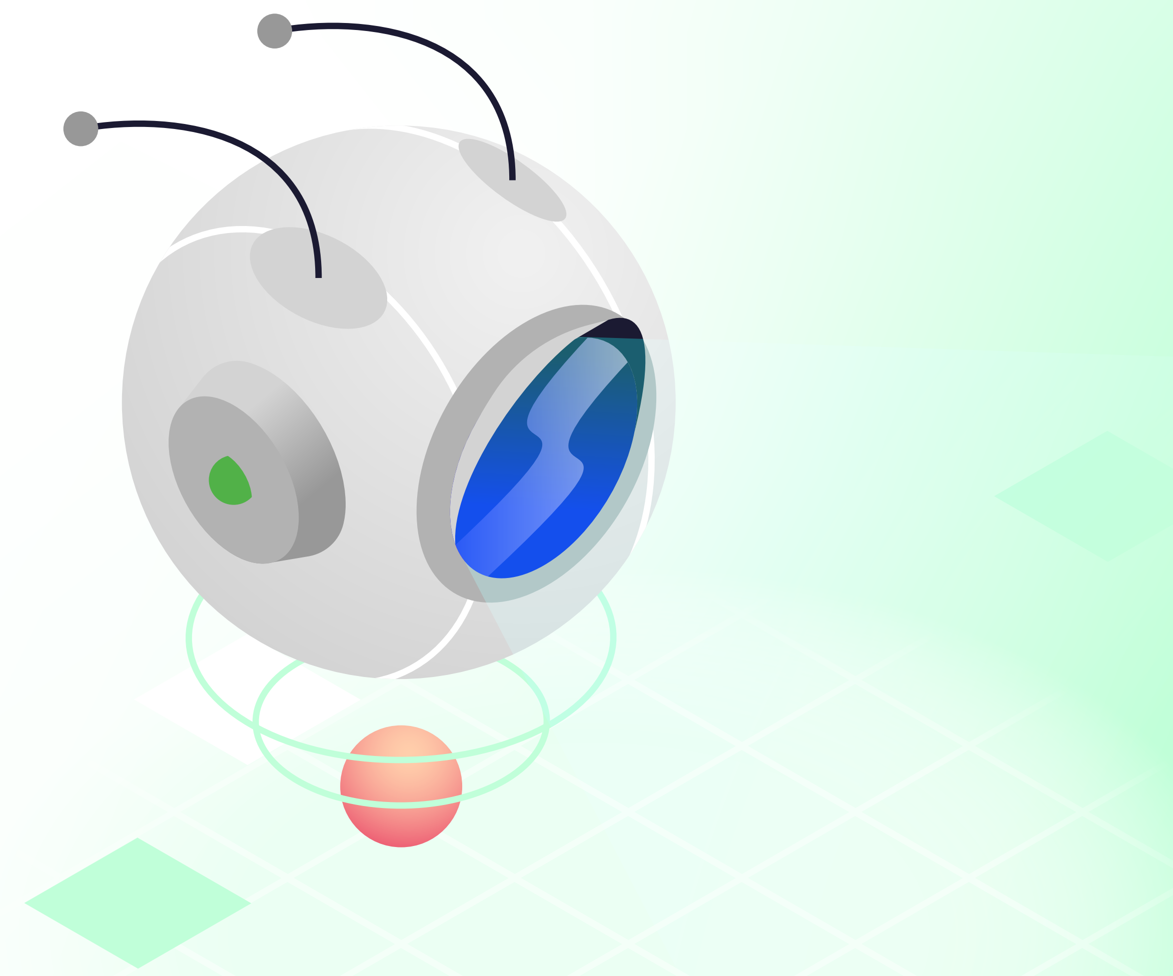
A well-organised AVM-based Bicep project should follow this structure:

```
/infra
  ├── bicepparam
  │   ├── dev.bicepparam
  │   ├── acc.bicepparam
  │   ├── prod.bicepparam
  ├── main.bicep
  ├── azure-pipelines.yml
  ├── .github/workflows/deploy.yml
```

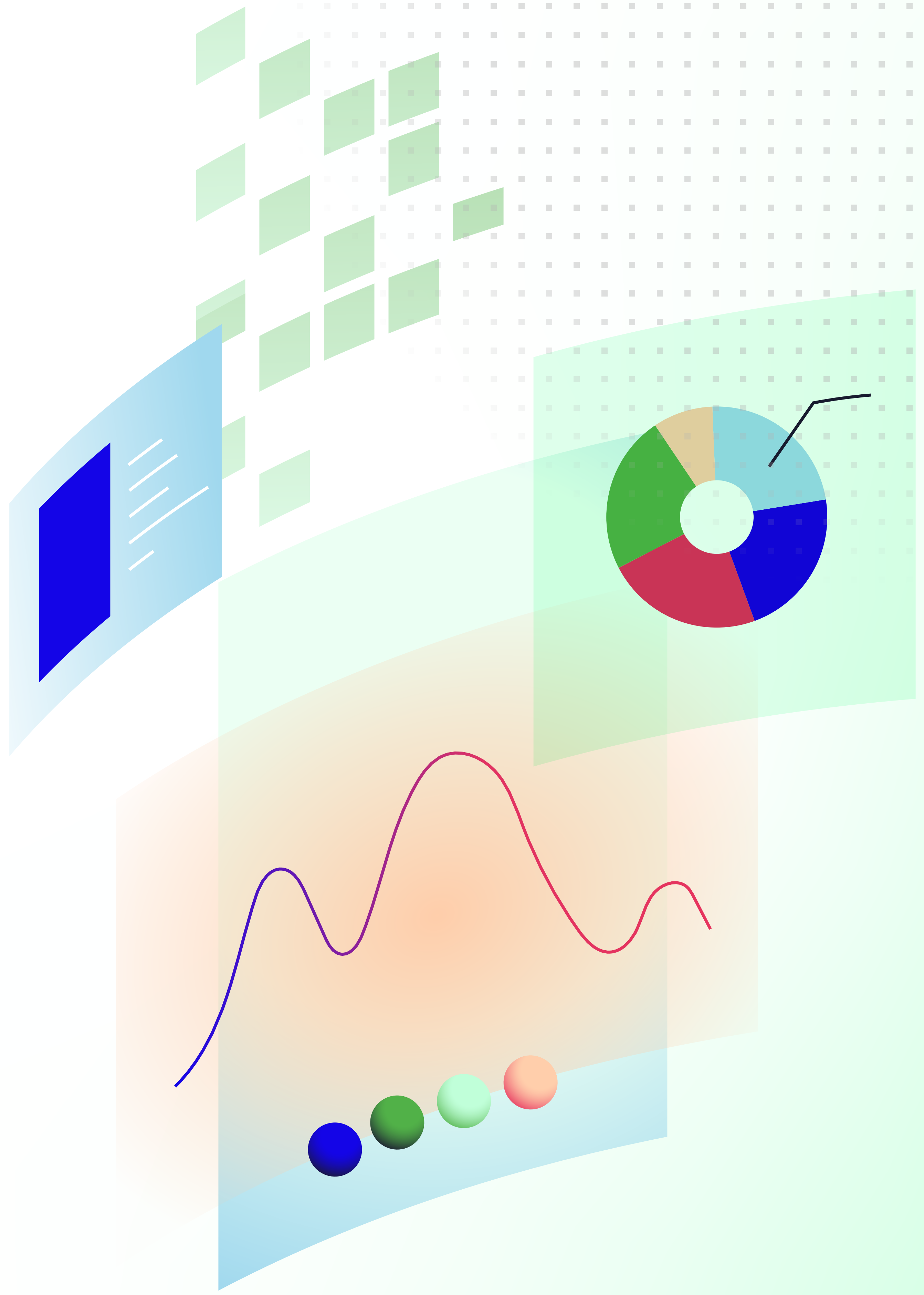
Using an Azure Verified Module in main.bicep

```
targetScope = ‘subscription’
@description(‘Customer Name (max 15 characters to prevent long resource names)’)
@minLength(3)
@maxLength(15)
param customerName string

@description(‘Deployment Location’)
param location string

@description(‘Location Short Code (2-4 characters, e.g., “weu” for West Europe)’)
@minLength(2)
@maxLength(4)
param locationShortCode string

@description(‘Environment Type’)
@allowed([
  ‘dev’
  ‘acc’
  ‘prod’
])
param environmentType string

// Resource Group Naming Convention
var resourceGroupName = ‘rg-${customerName}-${environmentType}-${locationShortCode}’

// Storage Account Name
@description(‘Storage Account Name (must be unique, 3-24 characters, lowercase)’)
@minLength(3)
@maxLength(24)
param storageAccountName string

// Azure Verified Modules - Start Here

module createResourceGroup ‘br/public:avm/res/resources/resource-group:0.4.1’ = {
  name: ‘create-resource-group’
  params: {
    name: resourceGroupName
    location: location
```

CODE CONTINUES
ON THE NEXT PAGE



```
  }
}

module storageAccount ‘br/public:avm/res/storage/storage-account:0.17.0’ = {
  name: ‘create-storage-account’
  scope: resourceGroup(resourceGroupName)
  params: {
    name: storageAccountName
    location: location
    kind: ‘StorageV2’
    skuName: ‘Standard_LRS’
  }
}

Environment-Specific Parameters (bicepparam/dev.bicepparam)
using ‘./main.bicep’

param customerName = ‘contoso’
param location = ‘westeurope’
param locationShortCode = ‘weu’
param environmentType = ‘dev’

// Derived storage account name (matches Azure’s naming rules)
param storageAccountName = ‘st${customerName}${environmentType}${locationShortCode}’
```



Deploying AVM Modules using Azure DevOps Pipelines
Pipeline YAML (azure-pipelines.yml)

```yaml
name: 'Infrastucure Deployment'

trigger:
- main

pool:
  vmImage: ubuntu-latest

parameters:
  - name: subscriptionId
    displayName: 'Subscription Id'
    type: string
    default: ''

  - name: environmentType
    displayName: 'Deployment Environment'
    type: string
    default: 'dev'
    values:
      - dev
      - acc
      - prd

steps:
- task: AzureResourceManagerTemplateDeployment@3
  inputs:
    azureResourceManagerConnection: <azureResourceConnectionId>)'
    deploymentScope: 'Subscription'
    subscriptionId: ${{ parameters.subscriptionId }}
    location: 'westeurope'
    templateLocation: 'Linked artifact'
    csmFile: './infra/main.bicep'
    csmParametersFile: infra/bicepparam/${{ parameters.environmentType }}.bicepparam
    deploymentMode: 'Incremental'
```

GitHub Actions

If your organisation is more comfortable using GitHub Actions, We have you covered. Here is an example workflow.

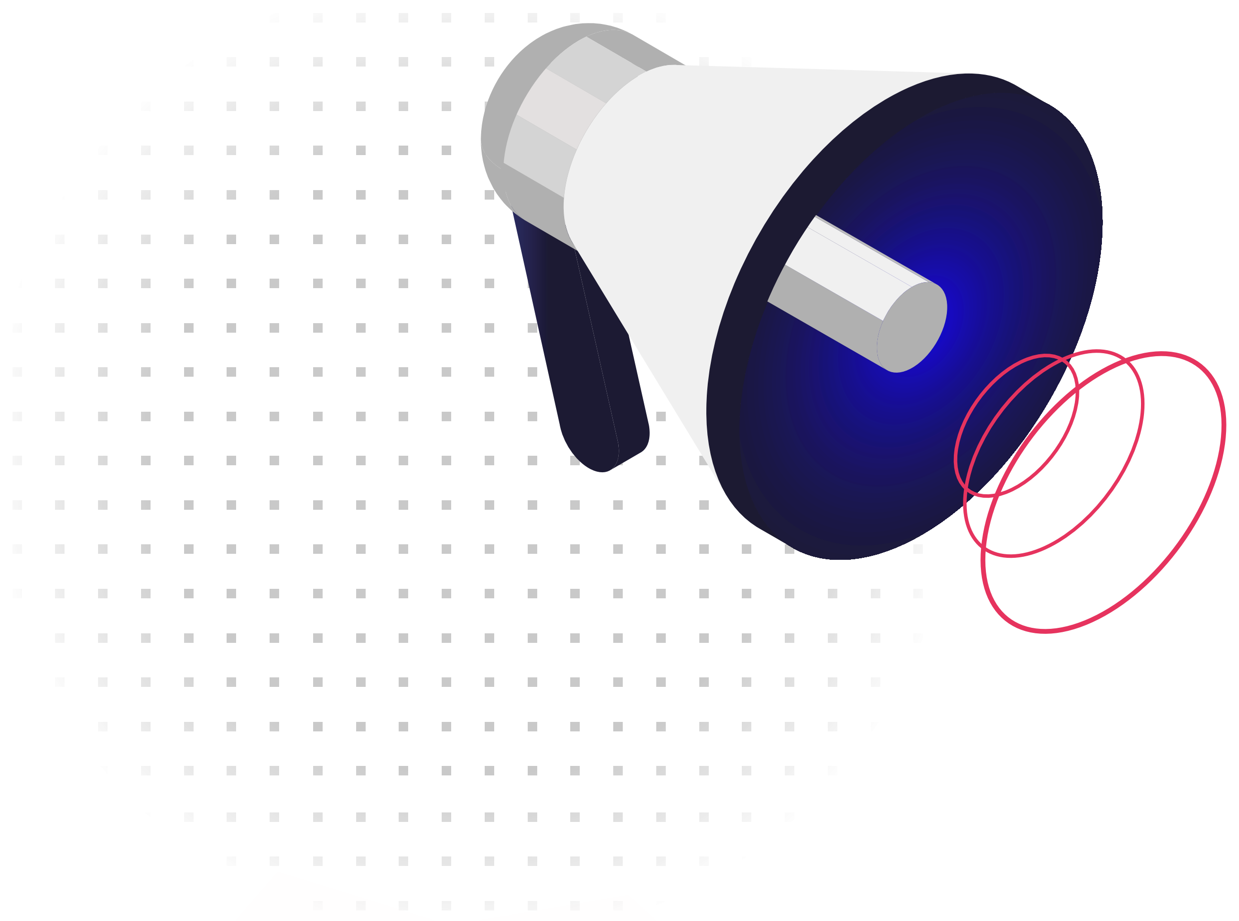
Deploying AVM Modules Using GitHub Actions

Workflow YAML (.github/workflows/deploy.yml)

```yaml
name: ‘Infrastructure Deployment’

on:
  workflow_dispatch:
    inputs:
      subscriptionId:
        description: ‘Azure Subscription ID’
        required: true

      environment:
        description: ‘Deployment Environment’
        required: true
        type: choice
        options:
          - dev
          - acc
          - prd

permissions:
  id-token: write

jobs:
  deploy:
    runs-on: ubuntu-latest
    env:
      AZURE_SUBSCRIPTION_ID: ${{ inputs.subscriptionId }}
      AZURE_ENVIRONMENT: ${{ inputs.environment }}

    steps:
      - name: Checkout repository
        uses: actions/checkout@v4

      - name: Azure Login
        uses: azure/login@v1
        with:
          client-id: ${{ secrets.AZURE_CLIENT_ID }}
          tenant-id: ${{ secrets.AZURE_TENANT_ID }}
          subscription-id: ${{ env.AZURE_SUBSCRIPTION_ID }}

      - name: Deploy Bicep Template
        uses: azure/arm-deploy@v1
        with:
          subscriptionId: ${{ env.AZURE_SUBSCRIPTION_ID }}
          scope: ‘subscription’
          region: ‘westeurope’
          template: ./infra/main.bicep
          parameters: “./infra/bicepparam/${{ env.AZURE_ENVIRONMENT }}.bicepparam”
          deploymentMode: Incremental
```

Best Practices for AVM Deployment (3)

While using the Azure Verified Modules is the way to go, let's look at some best practices regarding authentication and validation:



Best practice 1: Authentication best practices

- Use **OIDC** (OpenID Connect) for GitHub Actions instead of Service Principal passwords. [GitHub Docs - OIDC]
- Use **Managed Identity** for Azure DevOps where possible. [Microsoft Learn – Workload Identity Service]

Best practice 2: Validation before deployment

- Run az bicep build to **validate module syntax** before deployment.
- Use az deployment sub what-if to **preview changes** before applying.

Best practice 3: Reusability and scalability

- Store modules in an **Azure Bicep Registry** for centralized management.
- Use AVM to **leverage pre-tested, best-practice modules** instead of custom ones.

> **Note:**
> Using the –what-is flag will not work within a pipeline, as it requires you to complete a key entry to validate the what if with [Y] [Microsoft Learn].

Getting back to the main question: How can AVM and CI/CD be used to automate and scale infrastructure deployment reliably?

Using Azure Verified Modules helps make infrastructure deployments in Azure more modular, reusable, and compliant. By integrating Azure DevOps Pipelines and GitHub Actions, teams can automate deployments while following best practices. Both tools support CI/CD for deploying AVM modules. Which one you choose depends on your environment:

- Use Pipelines if you are using Azure DevOps.
- Use GitHub Actions if your code is on GitHub.

Lastly, now that we covered how AVM and CI/CD can automate and scale deployments, let's explore some next steps to further improve and expand your setup:

- Explore additional AVM modules for networking, compute, and security.
- Implement CI validation using the Bicep linter and what-if analysis.
- Store AVM modules in an Azure Bicep Registry to encourage broader adoption.

By using AVM Bicep modules in CI/CD workflows, you can create a scalable, compliant, and maintainable cloud infrastructure, while also improving efficiency and security.

Bringing it all together:

Your path to a scalable and efficient Infrastructure

In this e-book, you have discovered that while ClickOps is useful for learning and troubleshooting, **Infrastructure as Code** (IaC) is the key to scalable deployments. By using pipelines and CI/CD, you can standardise your deployments, ensuring consistency and reliability.

We have explored various technologies such as Terraform, Pulumi, ARM Templates, and Azure Bicep, all of which help build scalable, compliant, and maintainable cloud infrastructure.

Azure Verified Modules (AVM) improve the modularity, reusability, and compliance of Azure infrastructure deployments, making them easier to manage and maintain.

Integrating Azure DevOps Pipelines and GitHub Actions enables teams to automate deployments while adhering to best practices. If your codebase resides on GitHub, GitHub Actions is the perfect fit; otherwise, Azure DevOps Pipelines provides a solid alternative.

Incorporating AVM Bicep modules into your CI/CD workflows allows you to build robust, agile cloud infrastructure that fosters innovation and responsiveness to change.

With the insights from this e-book, you are now equipped to efficiently manage your infrastructure with IaC. By adopting CI/CD and AVM, you can quickly set up secure, reusable cloud environments that improves flexibility, consistency, and adaptability to change, while following best practices.

We hope you have enjoyed this e-book and that it has helped you optimise and future-proof your infrastructure management!

Team Intercept

INTERCEPT

Free Infrastructure Scan

Did you know you are eligible for a free Infrastructure Scan?
A **free Infrastructure scan** gives you immediate insight into the performance, vulnerabilities, and cost-efficiency of your infrastructure. Discover vulnerabilities for the future before they lead to unnecessary expenses!

Optimise your Azure infrastructure now to avoid expensive required fixes later. You will receive clear recommendations to strengthen your environment in line with Microsoft's best practices.

Why this matters
Infrastructure issues often go unnoticed... until it's too late. Act now to prevent costly disruptions, scaling problems, or system failures. Contact us today to get started with your free Infrastructure Scan!

→ Request your free Infrastructure Scan now!

Workshop DevOps on Azure

Master this **Azure DevOps workshop** to accelerate and streamline your software delivery. The skills you will gain during this workshop will help you release faster, reduce errors, and strengthen collaboration across your organisation.
Ready to work more agile?

→ Sign up now for free!