



# **Terraform for Azure: Infrastructure as Code Made Simple**

A practical guide to automating your Azure infrastructure with confidence and clarity



# The Problem with Manual Infrastructure

Imagine managing 50 virtual machines across different Azure regions, each requiring identical security configurations. With manual deployment through the Azure portal, you'd spend hours clicking through forms, inevitably introducing inconsistencies. One typo in a firewall rule could leave your infrastructure vulnerable. This is the reality many teams face daily.

# What is Infrastructure as Code?

Infrastructure as Code (IaC) is the practice of managing and provisioning infrastructure through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools.

Think of it like this: instead of manually assembling furniture by eye, you're following a precise instruction manual. The manual can be shared, reviewed, and used repeatedly to create identical results.



# The Traditional Way vs. Infrastructure as Code

## Manual Deployment (Traditional)

- Log into Azure Portal
- Click through multiple forms
- Configure settings manually
- Repeat for each environment
- Document changes in a spreadsheet

## Infrastructure as Code (Modern)

- Write configuration once in code
- Version control all changes
- Deploy consistently across environments
- Automate with a single command
- Self-documenting infrastructure

# Key Benefits of Infrastructure as Code



## Speed and Efficiency

Deploy complex infrastructure in minutes instead of hours. A development environment that took two days to provision manually now deploys in 10 minutes.



## Consistency and Reliability

Eliminate configuration drift between environments. Your staging environment becomes an exact replica of production, reducing "works on my machine" issues.



## Version Control and Audit Trail

Every infrastructure change is tracked in Git. You can see who changed what, when, and why—complete rollback capability if something goes wrong.



## Cost Optimization

Automate the teardown of non-production environments after hours. One team saved \$15,000 monthly by automatically shutting down dev/test resources overnight.





## Real-World Example: The Friday Night Deploy

A retail company needed to deploy infrastructure for their Black Friday sale. Using IaC, they created a tested configuration for 200 web servers, load balancers, and databases. On deployment day, they executed their Terraform code and had the entire infrastructure running in 15 minutes—fully tested and identical to their staging environment.

The previous year, a manual deployment took 8 hours and included three critical configuration errors that caused downtime during peak traffic.

# Enter Terraform



# What is Terraform?

Terraform is an open-source Infrastructure as Code tool created by HashiCorp. It allows you to define both cloud and on-premises resources in human-readable configuration files that you can version, reuse, and share.

What makes Terraform special is its ability to work with multiple cloud providers—Azure, AWS, Google Cloud—using the same workflow and language. You write code once and can adapt it across platforms.

"Terraform enables you to safely and predictably create, change, and improve infrastructure."





# Why Choose Terraform for Azure?

## Cloud-Agnostic Approach

Use the same tool and syntax whether you're deploying to Azure, AWS, or managing GitHub repositories. One skill set, multiple platforms.

## Declarative Syntax

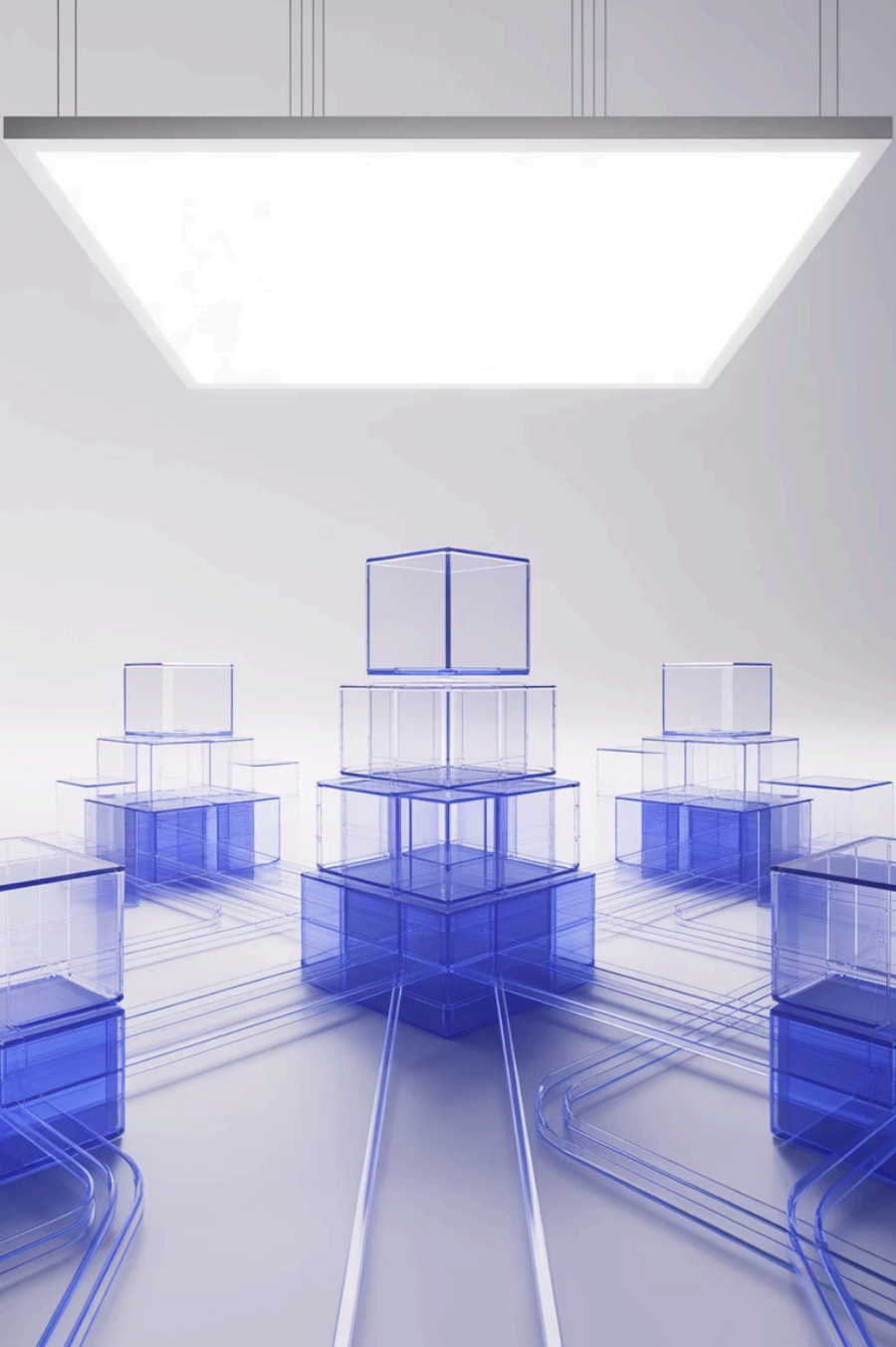
You describe *what* you want, not *how* to create it. Terraform figures out the execution plan. It's like telling a contractor "I want a three-bedroom house" instead of instructing them how to pour each foundation.

## State Management

Terraform tracks your infrastructure's current state, enabling it to determine what changes are needed. This prevents duplicate resources and manages dependencies automatically.

## Massive Provider Ecosystem

Over 3,000 providers available. Beyond Azure, manage Datadog monitoring, PagerDuty alerts, and DNS records—all from one tool.



# Terraform Architecture Overview

1

## **Terraform Core**

The engine that reads your configuration and creates an execution plan

2

## **Providers**

Plugins that interface with APIs (Azure, AWS, etc.)

3

## **State File**

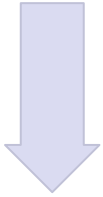
Records what infrastructure exists and its current configuration

# Understanding the Terraform Workflow



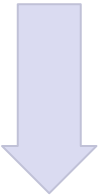
## Write Configuration

Define your infrastructure in .tf files using HashiCorp Configuration Language (HCL)



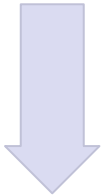
## Initialize

Run `terraform init` to download required providers and set up your working directory



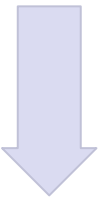
## Plan

Run `terraform plan` to preview what changes Terraform will make



## Apply

Execute `terraform apply` to create or modify your infrastructure



## Manage

Update your code and repeat the process as your infrastructure evolves

## Getting Started: Installation and Setup


## Prerequisites

- Azure subscription (free tier works fine)
- Azure CLI installed and configured
- Text editor (VS Code recommended)
- Basic command-line familiarity

## Installation Steps

1. Download Terraform from [terraform.io](https://terraform.io)
2. Add Terraform to your system PATH
3. Verify installation: `terraform version`
4. Authenticate with Azure CLI: `az login`



 **Pro Tip:** Use a version manager like tfenv to easily switch between Terraform versions for different projects.

# Terraform Core Concepts



## Providers

Plugins that enable Terraform to interact with cloud platforms, SaaS providers, and APIs



## Resources

The most important element—each resource block describes one or more infrastructure objects



## Variables

Input values that make your configurations flexible and reusable across environments



## Outputs

Values that are exposed after infrastructure is created, useful for other configurations



# Anatomy of a Terraform Configuration

```
# Configure the Azure Provider
terraform {
  required_providers {
    azurerm = {
      source = "hashicorp/azurerm"
      version = "~> 3.0"
    }
  }
}

provider "azurerm" {
  features {}
}

# Create a resource group
resource "azurerm_resource_group" "example" {
  name     = "my-resources"
  location = "East US"
}
```

This simple configuration tells Terraform to use the Azure provider and create a single resource group. Each block has a specific purpose, and the syntax is consistent across all providers.



# The Terraform Workflow in Detail

## **terraform init**

Downloads providers, prepares working directory.  
Run once per project or when adding new providers.

1

## **terraform apply**

Executes the plan. Creates, updates, or deletes  
resources to match your configuration.

2

3

## **terraform plan**

Shows what will change. Always review this before  
applying. Catches errors early.

4

## **terraform destroy**

Tears down infrastructure. Useful for cleaning up test  
environments and avoiding costs.



# Hands-On: Building Azure Resources

# Configuring the Azure Provider

The provider block tells Terraform how to authenticate and interact with Azure. You can authenticate using Azure CLI, a service principal, or managed identity.

For development, Azure CLI authentication is simplest. For production and CI/CD pipelines, use a service principal with limited permissions.

```
provider "azurerm" {  
  features {}  
  
  subscription_id = var.subscription_id  
  tenant_id       = var.tenant_id  
}
```

- ❏ The `features` block is required but can be empty. It controls provider-specific behaviors.

# Example 1: Creating a Resource Group

Resource groups are containers that hold related Azure resources. They're the foundation of Azure resource organization and a perfect first Terraform resource to create.

```
resource "azurerm_resource_group" "web_app" {  
  name     = "web-app-resources"  
  location = "East US"  
  
  tags = {  
    environment = "production"  
    project     = "customer-portal"  
  }  
}
```

## Breaking Down the Code

- `azurerm_resource_group`: The resource type
- `"web_app"`: Local name for reference
- `name`: Actual Azure resource name
- `location`: Azure region
- `tags`: Metadata for organization





## Example 2: Creating a Storage Account

Storage accounts provide cloud storage for blobs, files, queues, and tables. This example demonstrates how resources can reference other resources using Terraform's interpolation syntax.

```
resource "azurerm_storage_account" "data" {  
  name                = "mycompanydatalake"  
  resource_group_name = azurerm_resource_group.web_app.name  
  location            = azurerm_resource_group.web_app.location  
  account_tier        = "Standard"  
  account_replication_type = "LRS"  
  
  tags = {  
    environment = "production"  
  }  
}
```

Notice how we reference the resource group created earlier using `azurerm_resource_group.web_app.name`. This creates an implicit dependency—Terraform will create the resource group first.



# Understanding Resource Dependencies

1

## Implicit Dependencies

Created automatically when you reference one resource in another. Terraform analyzes these to determine creation order.

2

## Explicit Dependencies

Specified using `depends_on` when Terraform can't automatically detect the relationship.

3

## Dependency Graph

Terraform builds a graph of all resources and their dependencies, enabling parallel creation when possible.

# Using Variables for Flexibility

Variables make your Terraform configurations reusable. Instead of hardcoding values, you define variables that can change between environments.

## Defining Variables

```
variable "location" {  
  description = "Azure region"  
  type       = string  
  default    = "East US"  
}  
  
variable "environment" {  
  description = "Environment name"  
  type       = string  
}
```

## Using Variables

```
resource "azurerm_resource_group" "main" {  
  name     = "${var.environment}-resources"  
  location = var.location  
}
```

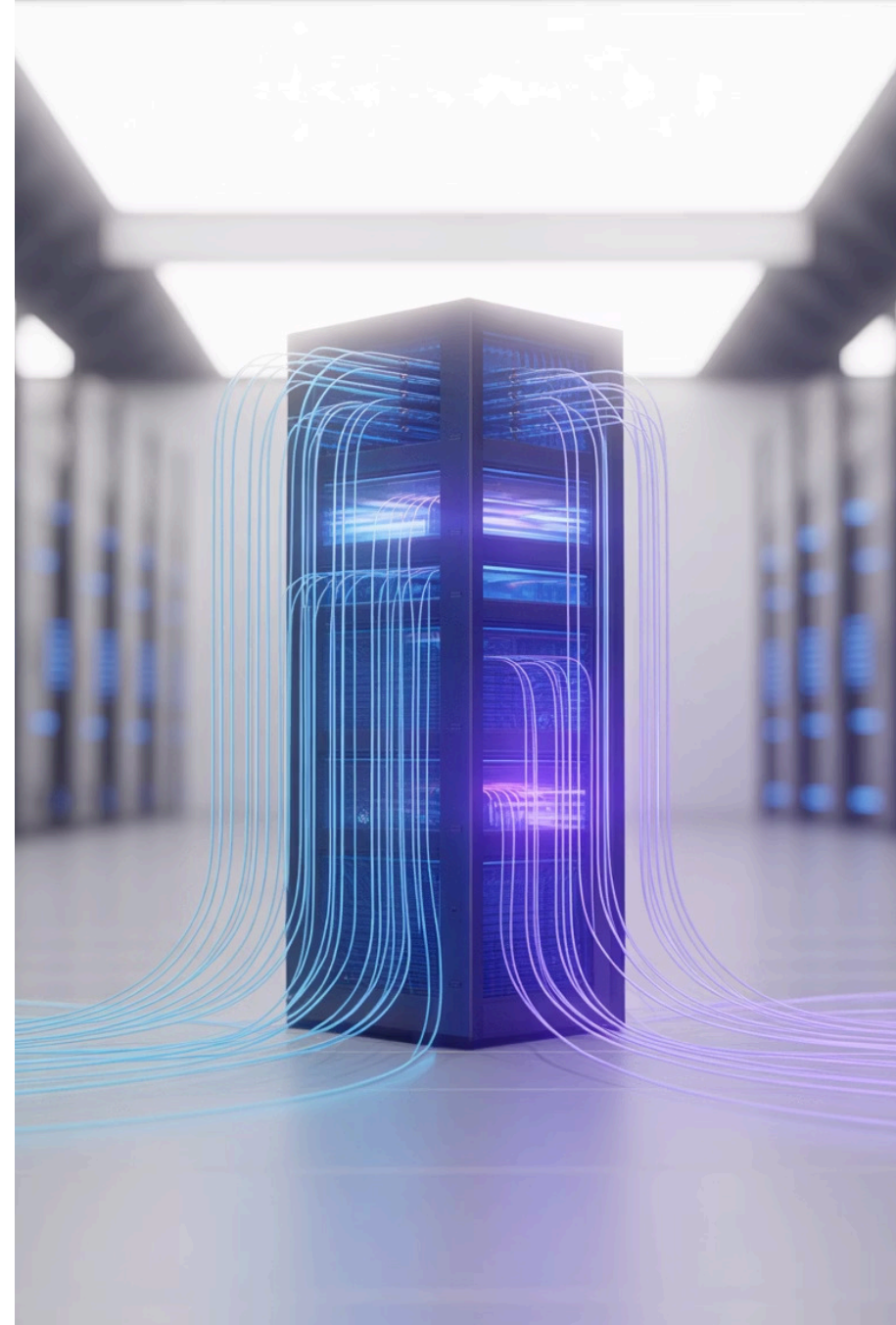
## Providing Values

- Command line: `-var="environment=dev"`
- File: `terraform.tfvars`
- Environment variables: `TF_VAR_environment`

## Example 3: Deploying a Virtual Machine

Creating a VM in Azure requires several interconnected resources: network interface, public IP, virtual network, subnet, and the VM itself. This demonstrates Terraform's power in managing complex dependencies.

```
resource "azurerm_virtual_network" "main" {  
  name = "production-network"  
  address_space = ["10.0.0.0/16"]  
  location = azurerm_resource_group.web_app.location  
  resource_group_name = azurerm_resource_group.web_app.name  
}  
  
resource "azurerm_subnet" "internal" {  
  name = "internal"  
  resource_group_name = azurerm_resource_group.web_app.name  
  virtual_network_name = azurerm_virtual_network.main.name  
  address_prefixes = ["10.0.2.0/24"]  
}
```



# Complete VM Configuration (Continued)

```
resource "azurerm_network_interface" "main" {  
  name = "production-nic"  
  location = azurerm_resource_group.web_app.location  
  resource_group_name = azurerm_resource_group.web_app.name
```

```
  ip_configuration {  
    name = "internal"  
    subnet_id = azurerm_subnet.internal.id  
    private_ip_address_allocation = "Dynamic"  
  }  
}
```

```
resource "azurerm_linux_virtual_machine" "main" {  
  name = "production-vm"  
  resource_group_name = azurerm_resource_group.web_app.name  
  location = azurerm_resource_group.web_app.location  
  size = "Standard_B2s"  
  admin_username = "adminuser"
```

```
  network_interface_ids = [  
    azurerm_network_interface.main.id,  
  ]
```

```
  os_disk {  
    caching = "ReadWrite"  
    storage_account_type = "Standard_LRS"  
  }
```

```
  source_image_reference {  
    publisher = "Canonical"  
    offer = "0001-com-ubuntu-server-focal"  
    sku = "20_04-lts"  
    version = "latest"  
  }  
}
```



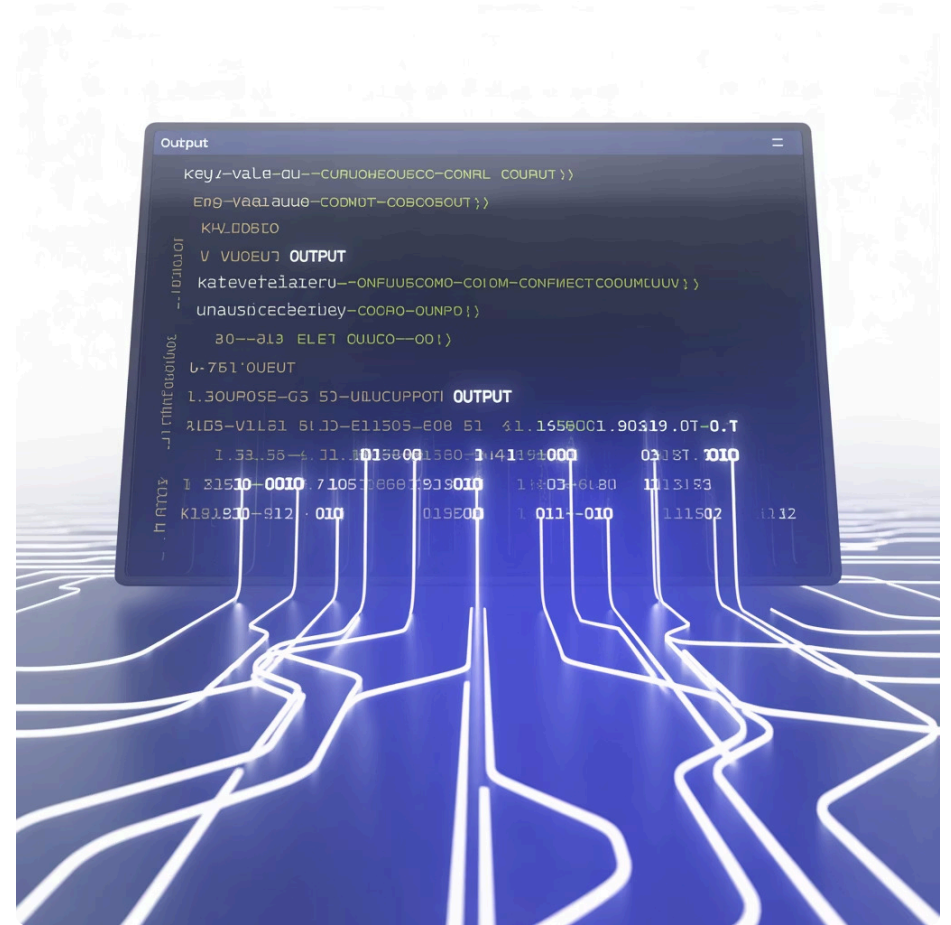
# Using Outputs to Share Information

Outputs expose values from your Terraform configuration that might be needed by other systems or configurations. They're also helpful for displaying important information after deployment.

```
output "resource_group_name" {
  value    = azurerm_resource_group.web_app.name
  description = "Name of the resource group"
}

output "vm_private_ip" {
  value = azurerm_network_interface.main.private_ip_address
  description = "Private IP of the VM"
}

output "storage_account_endpoint" {
  value = azurerm_storage_account.data.primary_blob_endpoint
  sensitive = true
}
```



After running `terraform apply`, outputs are displayed in the terminal. You can retrieve them later using `terraform output`.

Mark sensitive outputs with `sensitive = true` to prevent them from appearing in logs.

# Real-World Scenario: Multi-Environment Setup

A common challenge is maintaining separate development, staging, and production environments. With Terraform, you can use the same code with different variable files.

1

## dev.tfvars

```
environment = "dev"  
vm_size = "Standard_B1s"  
location = "East US"
```

2

## staging.tfvars

```
environment = "staging"  
vm_size = "Standard_B2s"  
location = "East US"
```

3

## prod.tfvars

```
environment = "prod"  
vm_size = "Standard_D4s_v3"  
location = "East US 2"
```

Deploy to each environment: `terraform apply -var-file="dev.tfvars"`

# Best Practices for Terraform Success

## 1 Always review terraform plan

Never run apply without checking the plan first. This preview has saved countless teams from costly mistakes like accidentally deleting production databases.

## 3 Implement naming conventions

Consistent naming makes your infrastructure easier to understand and manage. Use prefixes like `prod-` or `dev-` to identify environment at a glance.

## 2 Use remote state storage

Store your state file in Azure Storage or Terraform Cloud. Local state files get lost, corrupted, or cause conflicts in team environments.

## 4 Version control everything

Your `.tf` files belong in Git. Never lose track of infrastructure changes again. Use meaningful commit messages that explain why changes were made.

# Common Pitfalls to Avoid

## Hardcoding Values

Don't hardcode subscription IDs, resource names, or credentials in your code. Use variables and Azure Key Vault for sensitive data.

## Ignoring State File Security

State files contain sensitive information. Never commit them to version control. Use backend encryption and access controls.

## Making Manual Changes

Once you start using Terraform, avoid making manual changes in the Azure portal. They create drift between your code and reality.

## Skipping Documentation

Use comments and README files. Your future self (and teammates) will thank you when troubleshooting at 2 AM.



## Your Next Steps

1

### Practice with simple resources

Start with resource groups and storage accounts. Build confidence before tackling complex networking.

2

### Set up a test environment

Create a separate Azure subscription for experimentation. Test destructive operations without fear.

3

### Join the community

Participate in Terraform forums, follow HashiCorp's blog, and learn from others' configurations on GitHub.

4

### Automate something real

Pick a manual task you do regularly and automate it with Terraform. The best way to learn is by solving actual problems.



# Key Takeaways

## **IaC is the Future**

Infrastructure as Code eliminates manual processes, reduces errors, and enables automation at scale. It's not optional—it's essential for modern cloud operations.

## **Terraform is Powerful yet Simple**

With just four commands (init, plan, apply, destroy) and declarative syntax, you can manage infrastructure across any cloud provider.

## **Start Small, Think Big**

Begin with simple resources, but architect your code for reusability. Today's resource group is tomorrow's complete application environment.

---

You now have the foundation to begin your Terraform journey. The infrastructure you'll build with code will be more reliable, maintainable, and scalable than anything created manually. Welcome to the world of Infrastructure as Code!