How to Install Terraform and Build Your First Real Infrastructure Environment

AIBX Editorial Team

Version 1.1 – June 2026

Abstract

Terraform is one of the most important tools in modern cloud engineering. Used across DevOps, platform engineering, enterprise infrastructure, and AI operations, Terraform allows teams to provision complex architecture using declarative code instead of manual, error-prone cloud console configuration. This comprehensive guide details the precise installation processes and configuration frameworks required to build an industry-standard, multi-version local Terraform environment on macOS and Windows (WSL2), culminating in your first AWS infrastructure deployment.

---

1 Introduction & Objectives

In our previous guide, we explored what Terraform is and why Infrastructure as Code (IaC) fundamentally changed modern operations engineering. Now, we move from theory into practical implementation.

This comprehensive guide walks you through:

- Installing and managing Terraform versions cleanly using a version manager.
- Configuring a professional development workstation environment on macOS or Windows (WSL2).
- Setting up local security scanners, linters, and helper utilities.
- Establishing secure, credential-less-adjacent AWS local authentication.
- Initializing and deploying your very first real-world infrastructure project.

By the end of this guide, you will have a fully operational, industry-standard Terraform workstation capable of deploying real cloud infrastructure safely and efficiently.

—

2 What You Need Before Starting

Before diving into the installation steps, ensure you have the following prerequisites ready:

- **Required System:** A computer running macOS (Apple Silicon or Intel) or Windows 10/11 with administrator privileges.
- **Internet & Clouds:** A stable internet connection and an active **AWS Account** with administrative access.

- **Pre-requisite Knowledge:** Basic familiarity with your operating system's command-line interface (Terminal or PowerShell).

—

3 Recommended Workstation Architecture

Professional DevOps teams rarely install a single static binary of Terraform directly to their local machines. Because different enterprise projects often rely on different versions of Terraform, production-grade workstations utilize **version managers**, **security linters**, and **git hooks** to catch bugs before they ever reach a cloud environment.

The table below outlines the complete software stack this guide covers:

Table 1: Professional Workstation Software Stack

Tool	Purpose	Status
Homebrew / WSL APT	Package managers to handle installations cleanly.	Required
tfenv	Version manager to switch versions dynamically per project.	Required
AWS CLI v2	Authenticates your workstation with AWS services.	Required
Git	Version control engine for infrastructure code repositories.	Required
VS Code	IDE with language parsing extensions for code authoring.	Required
Docker Desktop	Handles local containers, Lambda builds, and cloud emulation.	Required
Python 3 / Node.js	Drives serverless executions, AWS SDKs, and linting frameworks.	Recommended
tflint / Checkov / tfsec	Static code analyses, safety diagnostics, and security linters.	Recommended
terraform-docs	Automatically generates markdown documentation from HCL files.	Recommended
pre-commit	Automates linting checks natively on local Git commits.	Recommended
jq	CLI JSON processor to parse complex AWS outputs natively.	Recommended
AWS SSM Plugin	Secure system connections to cloud instances without exposing port 22.	Recommended

—

4 Installing Terraform on macOS

These steps are optimized for Apple Silicon (M1/M2/M3) and Intel-based Mac systems using the native Terminal app.

4.1 Step 1: Install Homebrew

Homebrew manages your package installations cleanly. Execute the following in your terminal:

```
/bin/bash -c "$(curl -fsSL
   https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

WARNING: Apple Silicon (M1/M2/M3) Systems Only

You must append Homebrew explicitly to your system path. Execute the following commands in

sequence:

```
echo 'eval "$(/opt/homebrew/bin/brew shellenv)"' >> ~/.zprofile
eval "$(/opt/homebrew/bin/brew shellenv)"
```

Confirm the package manager is operational:

```
brew --version
```

4.2 Step 2: Install and Manage Terraform with tfenv

Instead of locking your OS to a single static Terraform runtime, use `tfenv` to change versions dynamically.

```
# Install the version manager
brew install tfenv

# Retrieve and use the latest stable version of Terraform
tfenv install latest
tfenv use latest
```

PRO-TIP: Pinning Local Project Versions

To force a specific directory to run an explicit version of Terraform, drop a file named `.terraform-version` into your root project directory containing the raw version string (e.g., `1.9.5`). `tfenv` will automatically swap binaries behind the scenes when you `cd` into the folder.

4.3 Step 3: Dependencies & Runtime Tooling

Deploy the programmatic support libraries and auxiliary software packages:

```
# Install AWS command line utilities and Git version control
brew install awscli git

# Install editor software and Docker container virtualization runtimes
brew install --cask visual-studio-code docker

# Ensure a modern release of Python 3 is installed
brew install python

# Set up Node Version Manager (nvm) for serverless integrations
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.7/install.sh |
    bash
```

Note: Restart your terminal window once NVM completes, then run `nvm install --lts && nvm use --lts` to install the active LTS runtime.

4.4 Step 4: Linting & Security Suite

Install professional DevOps auditing and static analysis utilities:

```
# Standard enterprise validation utilities and JSON parsers
brew install tflint terraform-docs tfsec jq pre-commit
```

```
# Setup safe EC2 tunnel software
brew install --cask session-manager-plugin

# Checkov configuration scanning engine
pip3 install checkov
```

—

5 Installing Terraform on Windows (WSL2 + Ubuntu)

Windows workflows are best managed using the native **Windows Subsystem for Linux (WSL2)** running an Ubuntu distribution. This ensures your code is parsed in a Linux environment that mirrors production cloud systems.

5.1 Step 1: Enable WSL2

Open your native Windows **PowerShell** app **as an Administrator** and execute:

```
wsl --install
```

Restart your computer when prompted. On reboot, a terminal console will automatically appear, prompting you to set up a unique Unix username and password.

5.2 Step 2: Configure Terminal & Ubuntu Environment

Install the modern Microsoft Windows Terminal:

```
winget install Microsoft.WindowsTerminal
```

> **WARNING: Crucial Command Line Workspace Rules**
>
> From this point onward, all remaining setup commands must be typed into your new **Ubuntu Linux prompt**, not PowerShell! Open Windows Terminal, navigate to Settings → Startup, and change your **Default Profile** directly to **Ubuntu**.

Inside your Ubuntu Linux terminal, perform update steps and install build dependencies:

```
sudo apt-get update && sudo apt-get upgrade -y

sudo apt-get install -y \
  curl wget unzip git build-essential \
  software-properties-common gnupg \
  ca-certificates lsb-release \
  python3 python3-pip jq
```

5.3 Step 3: Setup tfenv & Terraform

Install your local binary version manager inside Ubuntu:

```
# Clone repo and map execution boundaries
git clone https://github.com/tfutils/tfenv.git ~/.tfenv
echo 'export PATH="$HOME/.tfenv/bin:$PATH"' >> ~/.bashrc
```

```
source ~/.bashrc

# Deploy the latest Terraform binary
tfenv install latest
tfenv use latest
```

5.4 Step 4: Install AWS CLI and Node.js in WSL2

Deploy native AWS execution binaries and Javascript platforms:

```
# Fetch and run the official 64-bit AWS bundle
curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o
    awscliv2.zip
unzip awscliv2.zip
sudo ./aws/install
rm -rf aws awscliv2.zip

# Retrieve and configure Node
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.7/install.sh |
    bash
source ~/.bashrc
nvm install --lts && nvm use --lts
```

5.5 Step 5: Setup Static Linting Utilities

Run the local setup hooks to deploy professional auditing code components:

```
# Checkov security scanner
pip3 install checkov

# Linting \& static security engine script installers
curl -s
    https://raw.githubusercontent.com/terraform-linters/tflint/master/install_linux.sh
    | bash
curl -s
    https://raw.githubusercontent.com/aquasecurity/tfsec/master/scripts/install_linux.sh
    | bash

# Dev-utility hooks and documentation generators
pip3 install pre-commit

curl -sSLo ./terraform-docs.tar.gz
    https://terraform-docs.io/dl/v0.17.0/terraform-docs-v0.17.0-$(uname)-amd64.tar.gz
tar -xzf terraform-docs.tar.gz
chmod +x terraform-docs && sudo mv terraform-docs /usr/local/bin/
rm -f terraform-docs.tar.gz
```

—

6 Configuring VS Code and Docker Desktop

Configure your visual environments to operate seamlessly across system layers.

6.1 Docker Desktop Integration (Windows)

On Windows, install the runtime engine natively through PowerShell:

```
winget install Docker.DockerDesktop
```

Navigate to Docker Desktop's user interface: **Settings** → **Resources** → **WSL Integration**, and enable full system access for your targeted **Ubuntu** distribution. macOS users can run the native application directly from their Applications folder.

6.2 VS Code Extensions Integration

Execute the following commands in your host terminal to install essential infrastructure development extensions:

```
# CRITICAL FOR WINDOWS: Mounts VS Code directly inside the WSL filesystem
    boundary
code --install-extension ms-vscode-remote.remote-wsl

# Installs the official infrastructure language extensions
code --install-extension hashicorp.terraform
code --install-extension amazonwebservices.aws-toolkit-vscode
code --install-extension eamodio.gitlens
code --install-extension mikestead.dotenv
code --install-extension redhat.vscode-yaml
code --install-extension esbenp.prettier-vscode
```

> **WARNING: File Path Boundaries on Windows**
>
> Always place your projects within the native WSL2 Linux filesystem structure (e.g., `~/projects/`). **Never** run configurations inside shared Windows mounting locations (such as `/mnt/c/...`). Mixing paths causes severe lock conditions, performance degradation, and file permission errors.

Add the following to your VS Code `settings.json` to enable automatic file formatting on save:

```
"editor.formatOnSave": true,
"[terraform]": {
    "editor.defaultFormatter": "hashicorp.terraform"
}
```

—

7 Shell Customization and Aliases

Professional infrastructure developers rely on terminal shorthand aliases to accelerate their workflows. Open your profile configuration file (macOS: `~/.zshrc`, Windows WSL2: `~/.bashrc`) and append the following block:

```
# ------------------------------------------------------------------
# AIBX Enterprise Terraform Shorthand Aliases
# ------------------------------------------------------------------
alias tf='terraform'
alias tfi='terraform init'
alias tfp='terraform plan'
alias tfa='terraform apply'
alias tfd='terraform destroy'
```

```
alias tff='terraform fmt -recursive'
alias tfv='terraform validate'

# AWS Identity Quick Diagnostics
alias awsid='aws sts get-caller-identity'

# Enable Native Shell Command Tab Completion directly for Terraform binaries
complete -C $(which terraform) terraform
```

Apply the changes by running: `source ~/.zshrc` (macOS) or `source ~/.bashrc` (WSL2).

—

8 Securing AWS Authentication

Before Terraform can build infrastructure, you must authorize your workstation programmatically with AWS.

1. Navigate to your AWS Console, open the **IAM Dashboard**, and retrieve your programmatic account tokens (**Access Key ID** and **Secret Access Key**).
2. Initialize configuration steps locally:

   ```
   aws configure
   ```

3. Input your credentials systematically as prompted:

   ```
   AWS Access Key ID [None]: AKIAIOSFODNN7EXAMPLE
   AWS Secret Access Key [None]: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
   Default region name [None]: us-west-2
   Default output format [None]: json
   ```

Verify your workspace connection using your new custom alias:

```
awsid
```

Expected output: A JSON block displaying your active AWS Account ID and IAM User ARN.

> **WARNING: Identity Governance Guardrail**
>
> Never hardcode your access keys directly within `.tf` files or commit them to source control. The CLI utility writes credentials securely to `~/.aws/credentials`, which Terraform will parse automatically during execution.

—

9 Your First Real Terraform Project

Let us construct a real piece of versioned, code-managed cloud infrastructure.

9.1 Step 1: Setup Workspace Files

Create a clean project folder:

```
mkdir -p ~/projects/terraform-demo
cd ~/projects/terraform-demo
```

Configure your local Git profile settings:

```
git config --global user.name "Your Name"
git config --global user.email "you@example.com"
git config --global init.defaultBranch main
```

9.2 Step 2: Define Your Infrastructure

Open your workspace in VS Code:

```
code .
```

Create a file named exactly `main.tf` and write the following configuration:

```
terraform {
  required_version = ">= 1.0.0"
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.0"
    }
  }
}

provider "aws" {
  region = "us-west-2"
}

resource "aws_s3_bucket" "demo_bucket" {
  # WARNING: AWS S3 Bucket names must be globally unique across all cloud
     accounts.
  # Replace the trailing "xxxxxx" with your unique initials or a random
     number string.
  bucket = "aibx-terraform-demo-bucket-xxxxxx"

  tags = {
    Environment = "Development"
    ManagedBy   = "Terraform"
  }
}
```

—

10 Mastering the Core Terraform Workflow

Terraform follows a strict operational lifecycle: **Write** → **Plan** → **Apply**. This is the core loop of modern cloud operations.

10.1 Step 1: Initialize (tfi)

Before executing any actions, initialize your directory. This downloads the necessary cloud provider plugins into your local workspace.

```
tfi
```

Expected output: ‘Terraform has been successfully initialized¡

10.2 Step 2: Validate Code (tfv)

Perform structural validation checks to identify formatting and syntax errors:

```
tfv
```

Expected output: ‘Success! The configuration is valid.‘

10.3 Step 3: Run the Dry-Run Preview (tfp)

Generate a preview plan. This step acts as a critical safety check, showing you exactly what will be created, modified, or destroyed on AWS before any changes are executed:

```
tfp
```

Review the planning summary. It will state that it intends to execute: `Plan: 1 to add, 0 to change, 0 to destroy.`

10.4 Step 4: Deploy Your Infrastructure (tfa)

Provision your resources live on AWS:

```
tfa
```

Confirm the execution by typing `yes` when prompted. Within seconds, Terraform will build your new S3 bucket in the cloud.

—

11 Understanding Local State Mechanics

Look closely at your project workspace inside VS Code. You will see a newly generated file: `terraform.tfstate`.

This JSON file is the master record of your deployment. It tracks real-world cloud resource IDs, operational dependencies, and configuration mappings.

> **WARNING: State Preservation Security Mandate**
>
> Never commit your raw `.tfstate` files to public source control repositories. State files can contain sensitive configuration values and private environment details. For production-grade environments, enterprise operations use **Remote State Backends** (e.g., secure AWS S3 buckets paired with DynamoDB state locking).

—

12 Teardown & Cleanup (tfd)

To clean up your environment and avoid cloud costs, destroy your deployed resources:

```
tfd
```

Review the destruction plan, type `yes` to confirm, and Terraform will automatically remove the S3 bucket from AWS.

—

13 Common Beginner Pitfalls to Avoid

- **Hardcoding Credentials:** Never place raw AWS access keys inside your code. Use environment variables or secure credential files via `aws configure`.
- **Skipping Planning Steps:** Always inspect `terraform plan` output to confirm changes before running `terraform apply`.
- **Missing Gitignore Rules:** Always configure a comprehensive `.gitignore` file to prevent committing directories like `.terraform/`, lock files, or state backups.

—

14 Summary & Next Learning Paths

Now that your workstation is operational, you are ready to transition from local deployments to production-grade architectures. In our next modules, we will cover:

1. **Terraform State Deep Dive** – Managing secure remote lock state setups with AWS S3 and DynamoDB.
2. **Terraform Modules** – Designing clean, reusable templates to scale cloud architecture.
3. **Terraform CI/CD Automation** – Integrating checks and deployments directly into GitHub Actions and Terraform Cloud.

Your environment is ready, your tools are tuned, and your terminal is customized. Open up VS Code, and start building modern, scalable infrastructure!