Module 9 — Infrastructure as Code

Define infrastructure in version-controlled files: Terraform (provisioning), Ansible (configuration), cloud-init (first-boot), and Pulumi — and when to use each.

Lesson: What Infrastructure as Code Means

What you'll learn

- The difference between configuring servers by hand and defining them in code.
- What "declarative" and "imperative" mean, and why declarative tools dominate IaC.
- Why **idempotency** and **state** matter, with plain-language definitions.
- The three jobs IaC tools do: provisioning, configuration management, and bootstrap.
- How keeping infrastructure in Git ties back to version control (Module 4).

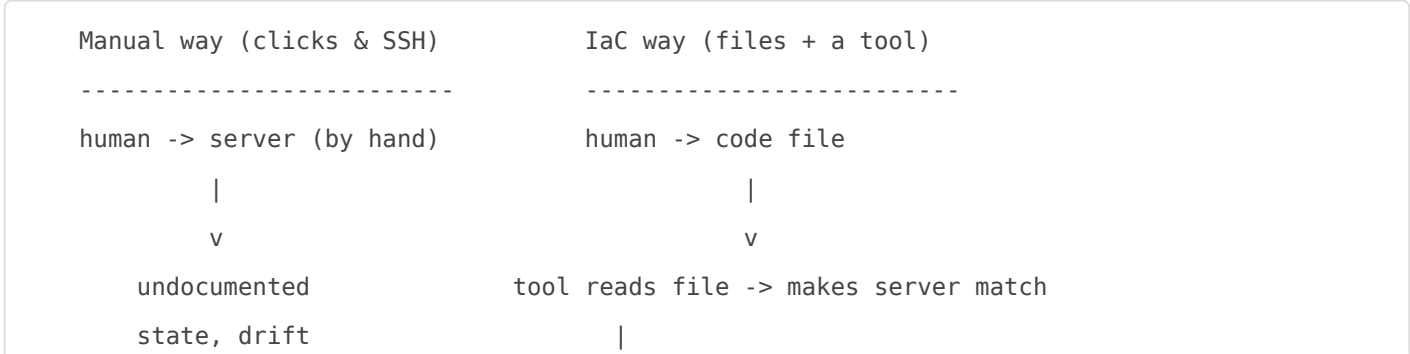
Skill gained: you can explain what Infrastructure as Code is, why teams use it, and which kind of tool does which job — the vocabulary you need for the rest of this module.

The lesson

1. The problem: "works on my server"

Imagine you set up a web server by hand. You SSH in, install packages, edit config files, open firewall ports, and reboot. It works. Six months later it crashes, and nobody remembers the exact steps. You rebuild it from memory and miss one setting. Now it behaves differently from before. This is **configuration drift** — the slow, invisible divergence of a system from what you *think* it is.

Infrastructure as Code (IaC) solves this by writing down *what the infrastructure should be* in plain text files that you store, review, and version like any other source code. Instead of remembering steps, you run a tool that reads the file and makes reality match it. Rebuilding becomes "run the tool again," not "try to remember."



```
                                v
                        same result every time
```

2. Codified infra vs manual infra

"Codified" just means *written as code/text*. The benefits flow from that one idea:

- **Repeatable:** the same file produces the same result on machine 1, 2, and 100.
- **Reviewable:** a teammate can read your change before it touches production (a pull request — see Module 4).
- **Auditable:** Git history shows who changed what, when, and why.
- **Recoverable:** lost a server? Re-run the code.

The lab you work in already does this. Every lab VM is created from one **golden template** and then configured on first boot by a small **cloud-init** snippet — a text file unique to that VM. That snippet *is* Infrastructure as Code you can read and copy. We'll dig into cloud-init in Chapter 4.

3. Declarative vs imperative

Two styles of telling a computer what to do:

- **Imperative** = you list the *steps*. "Install nginx. Then create this folder. Then start the service." Like a recipe. A shell script is imperative.
- **Declarative** = you describe the *desired end state*. "There should be an nginx server, running, with this config." The tool figures out the steps needed to get there.

```
Imperative:  step 1 -> step 2 -> step 3   (you own the "how")
Declarative: "I want THIS end state"       (tool owns the "how")
```

Most modern IaC tools are declarative (Terraform, much of Ansible, Pulumi, cloud-init). Declarative is powerful because the tool can compare *what is* to *what you declared* and change only the difference. You don't have to write "if it already exists, skip it" everywhere — the tool handles that.

4. Idempotency

Idempotent means: running the operation once or many times gives the same result. Press an elevator button five times — the elevator still comes once. Good IaC is idempotent: run it again on an already-correct server and *nothing* changes. Run it on a half-built server and it fixes only what's missing.

This is what makes IaC safe to re-run. A non-idempotent script that does `useradd bob` fails the second time ("user exists"). A declarative tool that says "user bob should exist" simply sees bob is already there and moves on.

5. State: how the tool knows reality

To make reality match your file, a tool must know what reality *currently is*. Tools handle this differently:

- **Terraform** keeps a **state file** — a record of every resource it created and its real-world ID. On the next run it reads state, checks the real world, and plans the difference. (More in Chapter 2.)
- **Ansible** is usually **stateless**: each run it logs into the machine and checks the actual current condition of each item live, then fixes what's off.
- **cloud-init** runs **once**, at first boot, so it tracks only "have I run on this instance already?"

Understanding "where does the tool keep its memory?" is one of the biggest differences between tools.

6. The three jobs (and one extra)

IaC is not one tool — it's a category. Map each tool to the *job* it's best at:

```
PROVISIONING        CONFIG MANAGEMENT      BOOTSTRAP
(create the box)    (set up what's inside) (first-boot setup)
e.g. Terraform      e.g. Ansible           e.g. cloud-init
      |                    |                     |
      v                    v                     v
"make a VM/network"  "install & configure"  "run once on launch"


GENERAL-PURPOSE: Pulumi (provisioning, but in a real language)
```

- **Provisioning** = creating the raw infrastructure: virtual machines, networks, disks, load balancers, DNS records. *Terraform* is the classic choice.
- **Configuration management** = setting up the software *inside* a machine that already exists: packages, files, services, users. *Ansible* is the classic choice.
- **Bootstrap** = the very first setup a fresh machine needs at boot. *cloud-init* is the standard on Linux cloud images. This is what your lab uses.
- **General-purpose / code-native** = *Pulumi* does provisioning like Terraform, but you write it in Python, TypeScript, or Go instead of a special config language.

These combine. A common real-world pipeline: **Terraform** creates the VM, hands it a **cloud-init** snippet for first-boot basics, then **Ansible** logs in to do detailed configuration. Chapter 5 covers how to choose and combine them.

7. Why this connects to version control

Because IaC is just text files, you put them in Git (Module 4). That means infrastructure changes get the same workflow as code changes: branches, pull requests, review, and history. "Infrastructure changed unexpectedly" turns into "show me the commit." This single habit — *infra lives in Git* — is the cultural core of IaC.

Dig deeper

- What is Infrastructure as Code? (Red Hat)
- Terraform: Intro & use cases (HashiCorp)
- Ansible: Getting started concepts
- cloud-init documentation
- Pulumi: What is Infrastructure as Code?

Search terms

- `infrastructure as code explained for beginners`
- `declarative vs imperative infrastructure`
- `what is idempotency in devops`
- `terraform vs ansible vs cloud-init`
- `configuration drift explained`

Check yourself

1. In your own words, what does "Infrastructure as Code" mean and what problem does it solve?
2. Give one example each of an imperative and a declarative instruction.
3. What does "idempotent" mean, and why does it make IaC safe to re-run?
4. Name the three core IaC jobs and one tool commonly used for each.
5. How does the lab let you see real IaC in your own environment today?

Lesson: Provisioning with Terraform

What you'll learn

- What Terraform is for: **provisioning** infrastructure (creating VMs, networks, DNS, etc.).
- The core building blocks: providers, resources, and HCL syntax.
- What the **state file** is and why it's central to how Terraform works.
- The everyday workflow: `init`, `plan`, `apply`, `destroy`.
- Where Terraform shines and where it doesn't.

Skill gained: you can read a basic Terraform file, explain what it would create, and describe the plan/apply loop — enough to follow real Terraform code and start small experiments.

The lesson

> Note: Terraform is **not deployed in the lab yet**. This lesson teaches it conceptually with small standalone examples you can read and try on your own machine.

1. What Terraform does

Terraform, by HashiCorp, is the most popular **provisioning** tool. Provisioning means creating the raw infrastructure: a virtual machine, a network, a firewall rule, a DNS record, a cloud storage bucket. You declare *what should exist*, and Terraform creates, changes, or deletes real resources to match.

It is **declarative** (you describe the end state) and tracks **state** (it remembers what it built). Those two facts explain almost everything about how it behaves.

2. Providers: Terraform's plugins

Terraform itself knows nothing about any specific platform. It learns through **providers** — plugins that translate your declarations into API calls. There's a provider for AWS, one for Azure, one for

Google Cloud, one for Proxmox, one for DNS, and hundreds more.

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

provider "aws" {
  region = "eu-central-1"
}
```

You declare which providers you need; `terraform init` downloads them. The `~> 5.0` is a version constraint meaning "5.x, but not 6.0" — pinning versions keeps runs reproducible.

3. Resources: the things you create

A **resource** is one piece of infrastructure. You write a `resource` block with a type, a name, and arguments:

```
resource "aws_instance" "web" {
  ami           = "ami-0abcd1234"   # which OS image
  instance_type = "t3.micro"        # size of the VM

  tags = {
    Name = "web-server"
    Env  = "lab"
  }
}
```

- `aws_instance` is the **type** (defined by the AWS provider).
- `web` is *your* name for it (used to reference it elsewhere).
- The block body declares the desired settings.

Resources can reference each other. Below, a firewall-rule resource points at the instance's ID — Terraform reads this and figures out it must create the instance *first*:

```
resource "aws_eip" "web_ip" {
  instance = aws_instance.web.id
}
```

That `aws_instance.web.id` reference builds a **dependency graph** so Terraform creates things in the right order automatically.

4. HCL: the language

Terraform files use **HCL** (HashiCorp Configuration Language), saved as `.tf` files. It's designed to be human-readable: blocks, key = value pairs, and a few helpers like variables and outputs.

```
variable "instance_size" {
  type    = string
  default = "t3.micro"
}

resource "aws_instance" "web" {
  ami           = "ami-0abcd1234"
  instance_type = var.instance_size
}

output "public_ip" {
  value = aws_instance.web.public_ip
}
```

- **variables** let you parameterize (no hard-coded values everywhere).
- **outputs** surface useful values after a run (e.g. the new server's IP — here it would print something like `203.0.113.10`).

5. State: Terraform's memory

After creating resources, Terraform records them in a **state file** (`terraform.tfstate`). State maps your code (`aws_instance.web`) to the real resource ID in the cloud. On the next run, Terraform:

```
   YOUR CODE  ──┐
                ├──>  Terraform compares  ──> PLAN (the diff)
   STATE FILE ──┤
                │
   REAL WORLD ──┘
```

1. Reads your `.tf` files (desired state).
2. Reads the state file (what it built last time).
3. Checks the real world (what's actually there).
4. Computes the **difference** and proposes changes.

This is why state matters: lose or corrupt it and Terraform forgets what it owns. In teams, state lives in shared **remote backends** (e.g. an S3 bucket or Terraform Cloud) with locking, so two people don't run at once. Never commit a state file to Git — it can contain secrets.

6. The core workflow

Four commands cover daily use:

```
terraform init     # download providers, set up the working dir (run once / when providers change)
terraform plan     # preview: show exactly what will be created/changed/destroyed
terraform apply    # do it (after showing the plan and asking yes)
terraform destroy  # tear it all down
```

A typical `plan` output reads like a diff:

```
  + create        (a new resource)
  ~ update         (change in place)
  - destroy        (remove)

Plan: 1 to add, 0 to change, 0 to destroy.
```

Always read the plan before you apply. The plan is your safety net — it tells you precisely what's about to change before anything real happens. If a plan says "destroy" something you care about, stop.

7. What Terraform is good at (and not)

Great at:

- Creating and managing the lifecycle of infrastructure across clouds.
- Multi-resource setups with dependencies (network + VMs + DNS together).
- Keeping a clear, reviewable record of *what infrastructure exists*.
- Tearing environments up and down repeatably (great for short-lived test envs).

Not its job:

- Installing and configuring software *inside* a machine over time — that's **configuration management** (Ansible, Chapter 3). Terraform can kick off a first-boot script, but it isn't designed to manage a server's ongoing internal state.
- One-off imperative tasks. Terraform wants to own the lifecycle of what it manages.

A common pattern: **Terraform builds the VM**, passes it a **cloud-init** snippet for first boot, and **Ansible** configures the software. Each tool does the job it's best at. Chapter 5 ties this together.

Dig deeper

- Terraform: Build infrastructure tutorial
- Terraform language documentation (HCL, resources, variables)
- Terraform: State explained
- Terraform CLI commands (plan/apply/destroy)
- Terraform Registry (browse providers)

Search terms

- `terraform getting started tutorial`
- `terraform plan apply destroy explained`
- `what is terraform state file`
- `terraform providers and resources HCL`
- `terraform remote backend why`

Check yourself

1. What job does Terraform do, and what does "provisioning" mean?
2. What is a provider, and why does Terraform need one?
3. Explain what the state file is and one reason losing it is bad.
4. What does `terraform plan` show, and why should you read it before `apply`?
5. Name one task Terraform is *not* the right tool for, and say which tool fits instead.

Lesson: Configuration Management with Ansible

What you'll learn

- What **configuration management** is and how Ansible does it.
- Why Ansible is **agentless** and works over plain SSH.
- The core pieces: **inventory**, **playbooks**, **tasks**, and **modules**.
- Why Ansible runs are **idempotent**, and what that looks like in practice.
- Where Ansible fits next to Terraform and cloud-init.

Skill gained: you can read a simple Ansible playbook, explain what it would do to a server, and describe how Ansible connects to and configures machines.

The lesson

> Note: Ansible is **not deployed in the lab yet**. This lesson teaches it conceptually with small standalone examples. The servers and IPs shown are illustrative.

1. What configuration management means

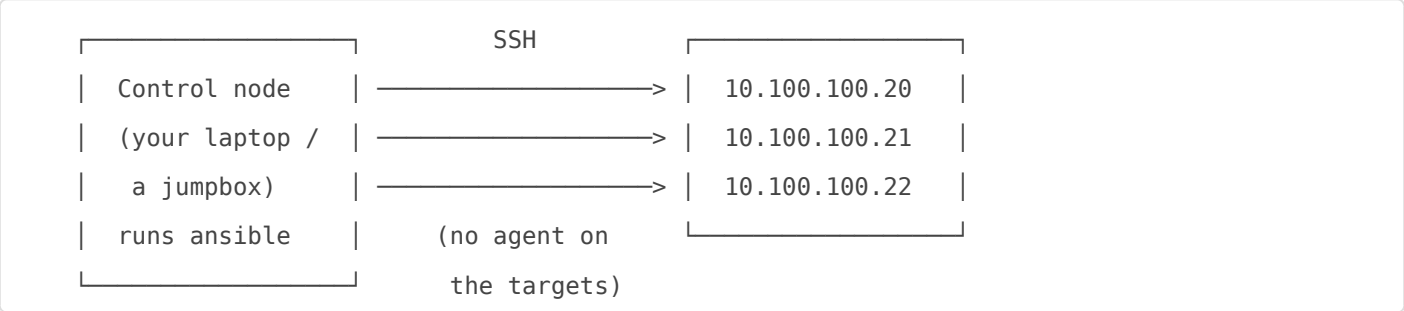
Provisioning (Terraform, Chapter 2) creates the empty machine. **Configuration management** is everything that happens *inside* it afterward: install packages, write config files, create users, start services, keep them that way over time. Ansible is the most popular configuration-management tool.

Ansible is largely **declarative**: you describe the state you want ("nginx installed and running, config file present"), and Ansible checks the live machine and makes only the changes needed.

2. Agentless over SSH — the big idea

Many older config tools need an **agent** — a piece of software permanently running on every managed machine. Ansible doesn't. It is **agentless**: it connects to each machine over **SSH** (the

same protocol you already use to log in), pushes over small instructions, runs them with Python, and disconnects.



Why this matters: nothing to install or maintain on the targets, and if a machine can be reached by SSH, Ansible can manage it. You run everything from one **control node** (your machine, or a bastion like the lab's **Jumpbox**).

3. Inventory: the list of machines

The **inventory** tells Ansible *which* machines to manage and how to group them. The simplest form is an INI-style file:

```
[webservers]
web1 ansible_host=10.100.100.20
web2 ansible_host=10.100.100.21

[dbservers]
db1 ansible_host=10.100.100.13

[all:vars]
ansible_user=omniops
```

Groups (`webservers`, `dbservers`) let you target many machines at once. You could also write inventory as YAML. In real setups, inventory can be **dynamic** — generated automatically from a cloud's API — but a static file is perfect to start.

4. Playbooks, plays, tasks, modules

The vocabulary, from biggest to smallest:

- **Playbook** — a YAML file describing what to do. The top-level thing you run.
- **Play** — a section of a playbook that maps a group of hosts to a list of tasks.
- **Task** — one unit of work ("install nginx").
- **Module** — the reusable code that actually performs a task (e.g. `apt`, `copy`, `service`). Ansible ships with thousands.

Here's a complete, readable playbook:

```
- name: Configure web servers
  hosts: webservers          # which group from inventory
  become: true               # use sudo
  tasks:

    - name: Install nginx
      ansible.builtin.apt:
        name: nginx
        state: present       # "present" = make sure it's installed

    - name: Deploy the site config
      ansible.builtin.copy:
        src: ./nginx.conf
        dest: /etc/nginx/nginx.conf
      notify: Restart nginx   # trigger a handler if this changed

    - name: Ensure nginx is running and enabled
      ansible.builtin.service:
        name: nginx
        state: started
        enabled: true

  handlers:
    - name: Restart nginx
      ansible.builtin.service:
        name: nginx
        state: restarted
```

Read it top to bottom: it targets the `webservers` group, becomes root, installs nginx, copies a config, and ensures the service runs. A **handler** is a special task that runs only when notified (here: restart nginx *only if* the config actually changed).

5. Idempotency in action

Each module checks the machine's real state before acting. `state: present` means "make sure nginx is installed" — if it's already there, the module does nothing. This is **idempotency** (Chapter 1): run the playbook ten times, the server ends up identical every time.

When you run a playbook, Ansible reports a colored summary per host:

```
PLAY RECAP *********************************************************
web1 : ok=4    changed=1    unreachable=0    failed=0
web2 : ok=4    changed=0    unreachable=0    failed=0
```

- **ok** = task checked, state already correct or just made correct.
- **changed** = Ansible actually altered something.
- A second run on an already-correct server should show **changed=0** everywhere. That's the proof your playbook is idempotent.

6. Running it

Two commands cover the basics:

```
# Run a quick one-off command (the "ping" module just checks connectivity)
ansible all -i inventory.ini -m ping

# Run a full playbook
ansible-playbook -i inventory.ini site.yml
```

There's also `--check` (a dry run that reports what *would* change without changing it) — Ansible's equivalent of `terraform plan`. Use it when you're nervous about a change.

7. Where Ansible fits

Great at:

- Installing and configuring software inside existing machines.
- Pushing changes to many servers at once, repeatably.
- Ongoing management — re-run anytime to correct drift.
- Orchestrating multi-step rollouts (update these, then those).

Not its strong suit:

- Creating the underlying infrastructure (VMs, networks). It *can*, via cloud modules, but **Terraform** is usually cleaner for provisioning.
- Things that should only ever happen once at boot, where **cloud-init** is simpler.

The common combination: **Terraform** makes the VMs, **cloud-init** does minimal first-boot setup, and **Ansible** does the detailed, ongoing configuration. In the lab, you reach internal VMs through the **Jumpbox** bastion — exactly the kind of place an Ansible control node would live. Chapter 5 covers choosing and combining tools.

Dig deeper

- Ansible getting started
- Ansible playbooks: intro
- How to build your inventory
- Ansible builtin modules index
- Ansible: check mode (dry run)

Search terms

- `ansible playbook tutorial for beginners`
- `ansible agentless ssh how it works`
- `ansible inventory file example`
- `ansible idempotent changed ok recap`
- `ansible modules apt copy service`

Check yourself

1. What does "configuration management" cover that provisioning does not?
2. What does "agentless" mean, and what protocol does Ansible use to reach machines?
3. Define inventory, playbook, task, and module in one sentence each.
4. On a second run against an already-configured server, what should the `changed` count be, and why?
5. Why might a team use Terraform *and* Ansible together rather than just one?

Lesson: First-Boot Bootstrap and Code-Native IaC

What you'll learn

- What **cloud-init** is and how it configures a fresh VM on its **first boot**.
- The shape of a cloud-init **user-data** file (packages, users, files, commands).
- How the **lab itself** uses cloud-init to turn the golden template into a working VM.
- What **Pulumi** is: IaC in real programming languages instead of a config language.
- When code-native IaC (Pulumi) is appealing versus HCL (Terraform).

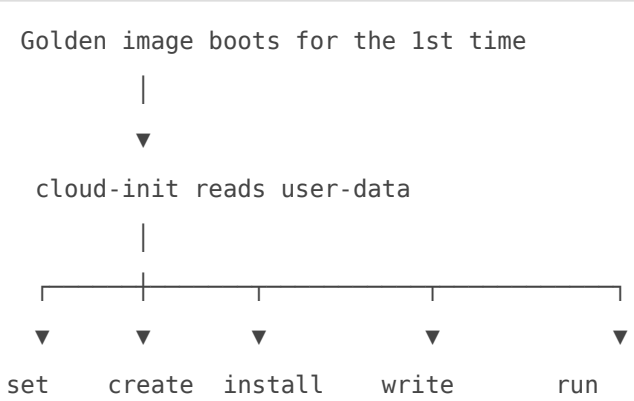
Skill gained: you can read and write a basic cloud-init file like the ones the lab uses, and explain what Pulumi offers compared to Terraform.

The lesson

1. cloud-init: the first-boot bootstrap tool

When a fresh Linux cloud image boots for the very first time, it's generic — no users beyond the default, no hostname you chose, nothing installed. **cloud-init** is the standard tool that runs *once, at first boot*, reads a config you supply (called **user-data**), and turns that generic image into the machine you wanted: sets the hostname, creates users, installs SSH keys, installs packages, writes files, runs commands.

This is **bootstrap** — the third IaC job from Chapter 1. It happens automatically, before you ever log in.



```
 hostname users  packages   files     commands
          |
          ▼
   VM is ready — cloud-init marks "done", won't repeat on reboot
```

2. This is exactly how the lab works

You don't have to imagine this — **the lab uses it right now**. Every lab VM starts from one **golden template**. On first boot it's configured by a small **cloud-init snippet** attached per-VM (the lab calls these per-VM snippets). That snippet is the difference between "generic template" and "this is the PostgreSQL server at 10.100.100.13." It's real Infrastructure as Code you can open and read in your own environment, and it's the best starting point for this whole module.

The gateway in front of those VMs is **pfSense**; the VMs sit on the `10.100.100.0/24` network and are reached through the Jumpbox bastion.

3. Anatomy of a cloud-init user-data file

A user-data file is YAML and **must start with the line** `#cloud-config` (cloud-init uses it to recognize the format). Here is a realistic one:

```
#cloud-config
hostname: web01
fqdn: web01.example.com

# Create a login user with sudo and an SSH key
users:
  - name: omniops
    groups: [sudo]
    shell: /bin/bash
    sudo: "ALL=(ALL) NOPASSWD:ALL"
    ssh_authorized_keys:
      - ssh-ed25519 AAAA...REPLACE_WITH_YOUR_PUBLIC_KEY

# Update the package list and install software
package_update: true
packages:
  - nginx
  - htop
```

```
# Write a file onto the new machine
write_files:
  - path: /etc/motd
    content: |
      Welcome to web01 — managed by cloud-init.

# Run shell commands at the end of first boot
runcmd:
  - systemctl enable --now nginx
```

Walking through the keys:

- `hostname` / `fqdn` — name the machine.
- `users` — create accounts; `ssh_authorized_keys` installs your **public** key so you can log in without a password. (Never put private keys or real passwords here — use `<REDACTED>` in any shared example.)
- `package_update` + `packages` — refresh the package index and install software.
- `write_files` — drop config files in place.
- `runcmd` — a list of shell commands run near the end of boot, for anything the structured keys don't cover.

4. Why "runs once" matters

cloud-init tracks, per instance, whether it has already run. On normal reboots it does **not** repeat the user-data — it's a *bootstrap* tool, not an ongoing-configuration tool. That's the key distinction from Ansible:

- **cloud-init** = get the machine to a usable baseline at birth, once.
- **Ansible** = keep configuring and re-configuring the machine over its whole life.

If you need a change *after* first boot, you don't edit user-data on a running VM — you use a config-management tool, or you rebuild the VM from an updated snippet. This "rebuild from the snippet" habit is exactly the IaC mindset.

5. Pulumi: IaC in a real programming language

Terraform uses its own language, HCL. **Pulumi** does the same *job* as Terraform — provisioning infrastructure declaratively, with providers and state — but you write it in a **general-purpose programming language** you may already know: Python, TypeScript, Go, or C#. It's the "code, not a config language" alternative.

The same VM-and-tag idea, in Pulumi with TypeScript:

```
import * as aws from "@pulumi/aws";

const web = new aws.ec2.Instance("web", {
  ami:          "ami-0abcd1234",
  instanceType: "t3.micro",
  tags: { Name: "web-server", Env: "lab" },
});

export const publicIp = web.publicIp;   // an output, like Terraform's output
```

Notice the shape is familiar: a resource, arguments, and an exported output. Under the hood Pulumi still talks to providers and keeps **state**, just like Terraform.

6. Why pick Pulumi (or not)?

Appealing because you get real language features for free:

- Loops and conditionals in normal syntax (`for`, `if`) instead of HCL's special constructs.
- Functions, classes, and your editor's autocomplete and type-checking.
- One language for both app code and infrastructure, if your team prefers that.

The trade-off:

- A general-purpose language gives you more rope — you *can* write tangled, hard-to-review infra logic. HCL's limitations keep things flat and predictable.
- Terraform has a larger community, more examples, and is what you'll most often meet first in the wild.

```
        Same job (provisioning, declarative, state, providers)
        ┌──────────────────────────────────────────────────────┐
        │  Terraform  ->  HCL (a purpose-built config language)│
        │  Pulumi     ->  Python / TypeScript / Go / C#        │
        └──────────────────────────────────────────────────────┘
```

For a beginner: learn the *concepts* (providers, resources, state, plan/apply) once, and they carry across both tools. The language is just the surface.

7. Putting Chapter 4 together

- **cloud-init** handles first-boot bootstrap — and it's the IaC you can touch in the lab today.
- **Pulumi** is a sibling of Terraform for provisioning, differing mainly in *how you write it*.

- Neither replaces the others; each owns a slice of the lifecycle. Chapter 5 shows how to choose and combine all of them.

Dig deeper

- cloud-init documentation (modules, examples)
- cloud-init: example user-data configs
- Pulumi: Get started
- Pulumi concepts: resources, state, providers
- Pulumi vs Terraform comparison (official)

Search terms

- `cloud-init user-data example #cloud-config`
- `cloud-init runcmd write_files users`
- `cloud-init first boot how it works`
- `pulumi vs terraform for beginners`
- `pulumi typescript ec2 instance example`

Check yourself

1. When does cloud-init run, and how many times per instance?
2. What must the first line of a cloud-config user-data file be, and why?
3. Name three things a cloud-init user-data file can set up on a fresh VM.
4. How does the lab use cloud-init, and why is that useful for learning IaC?
5. What does Pulumi do differently from Terraform, and what is one trade-off of that difference?

Lesson: When to Use Which

What you'll learn

- A simple decision rule for picking Terraform, Ansible, cloud-init, or Pulumi.
- The boundary between **provisioning**, **configuration management**, and **bootstrap**.
- How these tools **combine** in one workflow rather than competing.
- A worked example: Terraform + cloud-init + Ansible building one server together.
- How the lab's current setup maps onto this picture.

Skill gained: given a real task, you can name the right IaC tool (or combination) and justify the choice — the practical payoff of the whole module.

The lesson

1. Start with the job, not the tool

Beginners ask "which tool is best?" The better question is "**what job am I doing?**" Each tool owns a job (Chapter 1). Match the job, and the tool picks itself.

```
Question to ask                          Tool
_____________________________________    ________________________

"Create a VM / network / DNS record?"  -> Terraform (or Pulumi)
"Set up software inside a machine?"     -> Ansible
"First-boot baseline on a new VM?"      -> cloud-init
"Provisioning, but in Python/TS/Go?"    -> Pulumi
```

2. The three jobs, sharpened

- **Provisioning** = bring infrastructure into existence. The unit is a *resource*: a VM, a disk, a network, a load balancer, a DNS entry. Lives in the cloud/hypervisor API. → **Terraform** or **Pulumi**.
- **Bootstrap** = the one-time, first-boot setup that makes a freshly created machine usable. → **cloud-init**.
- **Configuration management** = the ongoing setup *inside* machines: packages, files, services, users, kept correct over the machine's whole life. → **Ansible**.

Pulumi isn't a fourth job — it's an *alternative way to do provisioning*. Choose Terraform vs Pulumi by **how you want to write it**: HCL (purpose-built config) vs a real programming language. Everything else about them is similar.

3. A quick decision checklist

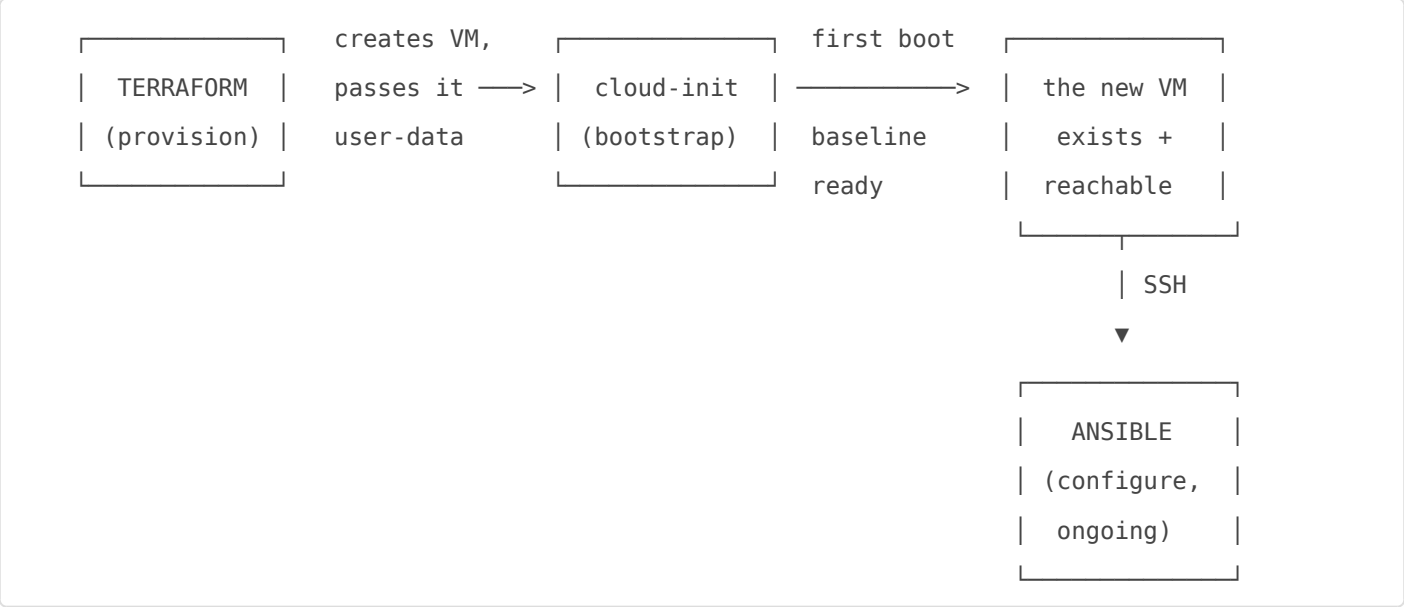
Ask, in order:

1. **Does the thing exist yet?** No → you need *provisioning* (Terraform/Pulumi).
2. **Is it a one-time first-boot setup?** Yes → *cloud-init*.
3. **Is it ongoing software/config inside an existing box?** Yes → *Ansible*.
4. **Provisioning, and the team prefers a real language?** → *Pulumi* instead of Terraform.

Most real situations need **more than one** of these. That's normal and correct — they're teammates, not rivals.

4. They combine: the classic pipeline

The most common professional pattern uses three tools, each for its job:



1. **Terraform** creates the VM, network, and firewall rules — and hands the VM a **cloud-init** user-data blob.
2. **cloud-init** runs at first boot: sets hostname, creates the login user, installs your SSH key, does minimal baseline setup so the machine is reachable.
3. **Ansible** then SSHes in to do the detailed, repeatable configuration — and you re-run it anytime over the machine's life to correct drift.

Each tool does what it's best at; none is stretched into a job it's poor at.

5. A worked example

Goal: stand up a web server.

- **Provision (Terraform):** declare one VM (`t3.micro`), a security group allowing ports 22 and 443, and a DNS record `web.example.com` pointing at the new public IP `203.0.113.10`.
- **Bootstrap (cloud-init, passed in by Terraform):**

```
#cloud-config
hostname: web01
users:
  - name: omniops
    groups: [sudo]
    ssh_authorized_keys:
      - ssh-ed25519 AAAA...your-key
package_update: true
packages: [python3]   # Ansible needs Python on the target
```

- **Configure (Ansible):** a playbook that installs nginx, deploys the site config, and ensures the service is running — re-runnable forever.

Notice cloud-init did just enough to make Ansible's job possible (a user to SSH as, Python present), then got out of the way. Clean division of labor.

6. Anti-patterns to avoid

- **Using Ansible to create cloud VMs from scratch** when Terraform would track their lifecycle far more cleanly. (Ansible *can*, but it's not its strength.)
- **Trying to re-run cloud-init for ongoing changes.** It's a first-boot tool; for day-2 changes use Ansible or rebuild from an updated snippet.
- **Putting secrets in any of these files.** Use a secrets manager or injected variables; in shared examples write `<REDACTED>`.
- **Skipping `plan` / `--check`.** Both Terraform (`plan`) and Ansible (`--check`) can preview changes. Preview before you change production.
- **Not version-controlling the files.** All of these are text — they belong in Git (Module 4), reviewed via pull requests.

7. How the lab maps onto this

Right now the lab uses exactly **one** of these tools in production, and that's deliberate for learning:

- **cloud-init is live:** every lab VM is built from a **golden template** and bootstrapped by a per-VM cloud-init snippet on first boot. That's your hands-on IaC today — open a snippet

and read it.
- **Terraform, Ansible, and Pulumi are not deployed yet** in the lab. You learned them here conceptually with small standalone examples. When the lab adds them, the natural fit is: Terraform (or Pulumi) to provision VMs on the hypervisor behind **pfSense**, cloud-init for first boot (already in place), and Ansible — run from the **Jumpbox** as its control node — for configuration.

If you can look at any task and say "that's provisioning / that's bootstrap / that's config management, so I'd reach for *this* tool," you've got the core skill of this module.

Dig deeper

- Terraform: provision (run cloud-init/scripts on new resources)
- Ansible: how it works / use cases
- cloud-init: what it is
- Red Hat: IaC tools compared
- Pulumi vs Terraform

Search terms

- `terraform vs ansible vs cloud-init when to use`
- `provisioning vs configuration management difference`
- `terraform cloud-init ansible together workflow`
- `day 1 vs day 2 operations devops`
- `infrastructure as code best practices beginners`

Check yourself

1. What single question should you ask first when picking an IaC tool?
2. Which tool owns provisioning, which owns bootstrap, and which owns configuration management?
3. In the classic combined pipeline, what does each of Terraform, cloud-init, and Ansible contribute?
4. Why is "re-run cloud-init to make a day-2 change" an anti-pattern, and what should you do instead?
5. Which of these tools is actually live in the lab today, and how is it used?

Assignment 1: Write a cloud-init config for a new VM

Goal: Write a valid `#cloud-config` user-data file that would bootstrap a fresh Linux VM into a usable, reachable machine — the same kind of first-boot IaC the lab uses on every VM.

Where: Do this on your own machine (or the Jumpbox). You'll write a file and validate it; you do **not** need to launch a real VM. The lab's per-VM cloud-init snippets (built on the golden template) are your reference for what a real one looks like.

Tasks

1. Create a file named `user-data.yaml`. Make the very first line `#cloud-config`.
2. Set the machine's `hostname` to `intern-web01` and `fqdn` to `intern-web01.example.com`.
3. Add a `users` entry that creates a user named `omniops`:
   - in the `sudo` group,
   - with shell `/bin/bash`,
   - with one `ssh_authorized_keys` entry — paste **your own public** key (an `ssh-ed25519 ...` or `ssh-rsa ...` line). Never paste a private key; never paste a password (use `<REDACTED>` if you must show the shape of one).
4. Set `package_update: true` and install at least these `packages`: `htop`, `curl`, and `python3` (Python is what a later Ansible run would need on the target).
5. Use `write_files` to create `/etc/motd` with a one-line welcome message naming the host.
6. Add a `runcmd` section with at least one command (for example, write the current date into `/var/log/firstboot.log`).
7. Validate the syntax. cloud-init ships a validator:

   ```
   cloud-init schema --config-file user-data.yaml --annotate
   ```

   If cloud-init isn't installed, validate that it's at least well-formed YAML, e.g. with `python3 -c "import yaml,sys; yaml.safe_load(open('user-data.yaml'))"`.
8. In a comment block at the bottom of the file, write 2-3 sentences: which keys run *once at first boot only*, and what you would use instead for a change you need *after* the VM is already running.

Deliverable

The single file `user-data.yaml`, syntactically valid, containing all sections above, committed to your Git repo (Module 4) with a clear commit message.

Acceptance criteria — you're done when:

- ☐ The file's first line is exactly `#cloud-config`.
- ☐ `hostname` and `fqdn` are set as specified.
- ☐ A `users` entry creates `omniops` with sudo, bash, and a real **public** SSH key (no private keys, no plaintext passwords).
- ☐ `package_update: true` is set and `htop`, `curl`, `python3` are all in `packages`.
- ☐ `write_files` creates `/etc/motd` with a welcome line naming the host.
- ☐ `runcmd` has at least one command.
- ☐ The file passes `cloud-init schema` (or at least parses as valid YAML).
- ☐ A bottom comment explains what runs only at first boot and what to use for day-2 changes.
- ☐ The file is committed to Git with a meaningful message.

Hints

- Indentation is everything in YAML — use spaces, never tabs. `users:` items are a list, so each starts with `- name: ...`.
- `ssh_authorized_keys` is a **list**, even if you have only one key.
- To find your public key: `cat ~/.ssh/id_ed25519.pub` (the `.pub` file — the one safe to share).
- Compare your file to a real lab per-VM snippet if you can open one; the structure should feel familiar.
- Re-read Chapter 4 sections 3 and 4 for the anatomy of user-data and the "runs once" rule.

blocked for >~30 min after re-reading the lessons? Bring what you've tried to your mentor.

Assignment 2: A small Ansible playbook (or Terraform plan) walkthrough

Goal: Read, explain, and (for the Ansible path) dry-run a small piece of IaC so you can describe exactly what it does *before* it touches anything. This builds the habit of "preview, then apply."

Where: On your own machine or the Jumpbox. Neither Ansible nor Terraform is deployed in the lab, so this is a standalone learning exercise — you can install the chosen tool locally, or do the read-and-explain parts without installing anything. Pick **one** track: A (Ansible) or B (Terraform).

Tasks

Track A — Ansible (recommended)

1. Create `inventory.ini` with one group `[local]` containing `localhost ansible_connection=local` (so you can run safely against your own machine, no SSH needed).
2. Create a playbook `site.yml` that targets `local` and has at least three tasks using built-in modules, for example:
   - `ansible.builtin.file` — ensure a directory `/tmp/ansible-demo` exists (`state: directory`).
   - `ansible.builtin.copy` — write a small file into that directory with some `content:`.
   - `ansible.builtin.debug` — print a message.
3. **Dry-run first:** `ansible-playbook -i inventory.ini site.yml --check`. Read the output. Note which tasks report `changed`.
4. **Run for real:** `ansible-playbook -i inventory.ini site.yml`. Read the `PLAY RECAP`.
5. **Run it again** unchanged. Record the `changed=` count from the second run.
6. Write 4-6 sentences explaining: what each task does, why the second run shows a different `changed` count from the first, and what "idempotent" means here.

Track B — Terraform (read & plan, no cloud account needed)

1. Create `main.tf` using the built-in `local_file` resource (the `hashicorp/local` provider — no cloud account, no credentials):

   ```
   resource "local_file" "hello" {
     filename = "${path.module}/hello.txt"
     content  = "Provisioned by Terraform\n"
   }
   ```

2. Run `terraform init`, then `terraform plan`. Read the plan; identify the `+ create` line.
3. Run `terraform apply` and confirm `hello.txt` appears. Open `terraform.tfstate` and find where the resource is recorded.
4. Run `terraform plan` again with no changes — note that it reports "No changes."
5. Run `terraform destroy` and confirm the file is removed.
6. Write 4-6 sentences explaining: what the plan showed before apply, what the state file is for, and why the second plan reported no changes.

Deliverable

The files for your chosen track (`inventory.ini` + `site.yml`, **or** `main.tf`) plus a short `WALKTHROUGH.md` containing your 4-6 sentence explanation and a pasted copy of the key command output (the recap or the plan). Commit everything to Git.

Acceptance criteria — you're done when:

☐ You completed one full track (A or B), files included.

☐ (Track A) The playbook has at least three tasks using built-in modules and runs successfully.

☐ (Track A) You ran `--check` first, then a real run, then a second real run, and recorded both `changed` counts.

☐ (Track B) `terraform plan` was run and read *before* `apply`; you located the resource in `terraform.tfstate`; you ran `destroy`.

☐ `WALKTHROUGH.md` explains, in your own words, what the preview showed and why the second run was a no-op (idempotency / no changes).

☐ `WALKTHROUGH.md` includes pasted command output (PLAY RECAP or the plan).

☐ Everything is committed to Git with meaningful messages.

☐ No secrets, real passwords, or private keys appear in any file (`<REDACTED>` if you must show shape).

Hints

- Track A is friendlier if you've never touched these tools — `ansible_connection=local` means it runs on your own box with no SSH setup.
- Install hints: Ansible via `pipx install ansible` or your package manager; Terraform from the official HashiCorp downloads page.
- The whole point is the *preview*: `--check` (Ansible) and `plan` (Terraform) both tell you what *would* happen. Always look before you leap.
- A second, unchanged run doing "nothing" is success, not failure — that's idempotency proving itself (Chapters 1 and 3).
- Re-read Chapter 3 (Ansible recap output) or Chapter 2 (Terraform plan/state) for the exact terms.

blocked for >~30 min after re-reading the lessons? Bring what you've tried to your mentor.