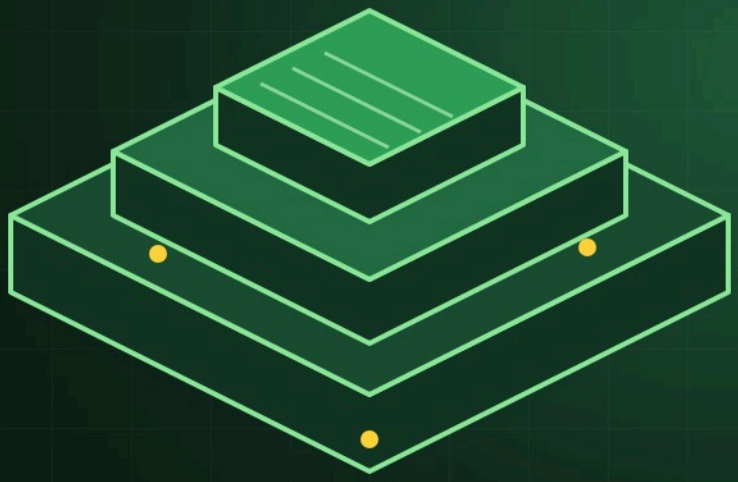


resource { }

A PRACTICAL GUIDE

Terraform and Infrastructure as Code

Build, Change, and Scale Infrastructure Safely

HCL · State · Modules · AWS · Azure · GCP · Kubernetes
Testing · Policy as Code · CI/CD · OpenTofu

PLAN · REVIEW · APPLY · REPEAT

THENEXBOOKS

Terraform and Infrastructure as Code

A Practical Guide

First Edition

THENEXBOOKS

Terraform and Infrastructure as Code
A Practical Guide

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages — including cloud charges or data loss — resulting from the use of the information contained herein. Tools, providers, licenses, and cloud services change frequently; always consult the official documentation and license terms for the versions you use, and practice in a sandbox account.

This book is an independent publication and is not affiliated with, sponsored by, or endorsed by HashiCorp, IBM, or any cloud provider. Terraform and HCP Terraform are trademarks of HashiCorp, an IBM company. OpenTofu is a trademark of The Linux Foundation. AWS, Microsoft Azure, Google Cloud, Kubernetes, GitHub, Pulumi, and other product names are trademarks of their respective owners; their use here is for identification only. Fieldmark is a fictional company.

Preface

Every team that runs software in the cloud eventually meets the same problem. The first environment is built by hand in a web console. It works. Then a second environment is needed, and a third; someone changes a security group in production and forgets to mention it; nobody is quite sure which settings matter; and rebuilding anything feels risky. Infrastructure has become a collection of undocumented decisions.

Infrastructure as Code solves this by describing infrastructure in version-controlled files that can be reviewed, tested, and applied repeatedly. Terraform is one of the most widely used tools for doing so: a single language and workflow for managing resources across cloud providers, SaaS platforms, and on-premises systems.

This book is a practical guide to Terraform and to Infrastructure as Code as a discipline. We start from zero — the language, the workflow, and the state file that makes it all work — and progress to reusable modules, multi-account environments, real networking, compute, and data infrastructure, testing, security scanning, policy as code, CI/CD pipelines, and the day-to-day operations that keep infrastructure healthy for years. Throughout, we emphasize the habits that prevent outages: reading plans carefully, protecting state, limiting blast radius, and changing production through reviewed pipelines.

Who This Book Is For

- **Developers** who deploy their own services and want reliable, repeatable infrastructure.
- **DevOps, platform, and cloud engineers** who manage environments for many teams.
- **System administrators** moving from manual or script-based operations to Infrastructure as Code.
- **Tech leads and architects** who need to design safe workflows, module strategies, and governance for Terraform.

What You Will Learn

- The principles of Infrastructure as Code and how Terraform compares with OpenTofu, Pulumi, CloudFormation, and the CDK.
- HCL in depth: resources, data sources, variables, outputs, expressions, functions, and validation.
- State: how it works, how to store and lock it remotely, and how to import, move, and refactor safely.
- Reusable modules, registries, versioning, project structure, multiple environments, and multi-account setups.
- Building real AWS infrastructure — networking, containers, databases, storage, and serverless — plus Azure, Google Cloud, and Kubernetes basics.

- Testing, static analysis, security scanning, policy as code, secrets handling, and CI/CD for Terraform.
- Day-2 operations: drift, cost awareness, upgrades, and scaling Infrastructure as Code across teams.

How This Book Is Organized

Part I — Foundations of Infrastructure as Code (Chapters 1–5) introduces the ideas, the tool, and the language. **Part II — State and the Terraform Workflow** (Chapters 6–10) explains state, remote backends, safe planning and applying, refactoring, and meta-arguments. **Part III — Modules and Code Organization** (Chapters 11–15) covers modules, versions, project layout, environments, and providers. **Part IV — Building Real Infrastructure** (Chapters 16–20) builds networking, compute, data, multi-cloud, and Kubernetes infrastructure. **Part V — Quality, Security, and Operations** (Chapters 21–25) covers testing, scanning, policy, secrets, CI/CD, and day-2 operations. **Part VI — Real-World Projects** (Chapters 26–28) brings everything together. The appendices provide a cheat sheet, module templates, a production checklist, and a glossary.

Every chapter ends with a **Try It Yourself** section and a **Key Takeaways** box.

The Running Example: Fieldmark

Throughout the book we manage the infrastructure of **Fieldmark**, a fictional agriculture-technology company. Farms install soil-moisture and weather sensors; readings flow into Fieldmark's cloud platform, which powers a farmer app, frost and drought alerts, and analytics dashboards. Fieldmark's platform team runs separate development, staging, and production environments, and manages every piece of them with Terraform.

Versions, Licensing, and Safety

Examples target **Terraform 1.x** in recent versions and recent major versions of the providers. Where a feature arrived in a specific release, we say so; where behavior differs between versions, we point you to the documentation. Terraform's license changed in 2023, and the community fork OpenTofu continues under an open-source license; Chapter 1 explains the history neutrally, and you should review the license terms for your own use case.

Provider version constraints in the examples (for instance `~> 5.0` or `~> 6.0` for the AWS provider) reflect the major version each example was written and reviewed against. They are not a recommendation to use an old release: in your own projects, pin the current major version you have tested, read the upgrade guide when you move between major versions, and commit the dependency lock file (Chapter 12).

Terraform changes real infrastructure. Practice in a dedicated sandbox account, set up budget alerts, read every plan before applying it, and run `terraform destroy` when you finish an exercise.

Conventions Used in This Book

> **TIP**
>
> A recommended practice or a technique that makes your work easier.

> **NOTE**
>
> Background information, version details, or a useful aside.

> **WARNING**
>
> A common mistake or an action that can destroy resources, leak secrets, or cause outages.

Code appears in `monospace`. When a code block shows a file, its path appears above it — for example `modules/network/main.tf`. Sample account IDs and domains are fictional.

Contents

Preface

PART I — FOUNDATIONS OF INFRASTRUCTURE AS CODE

1 Why Infrastructure as Code?

2 Getting Started: Installing Terraform and Your First Deployment

3 HCL in Depth: Blocks, Expressions, Types, and Functions

4 Resources, Data Sources, and Dependencies

5 Variables, Outputs, Locals, and Validation

PART II — STATE AND THE TERRAFORM WORKFLOW

6 Understanding Terraform State

7 Remote State Backends and Locking

8 The Core Workflow: Plan, Apply, and Reviewing Changes Safely

9 Importing, Moving, and Refactoring Infrastructure

10 Meta-Arguments: count, for_each, depends_on, lifecycle, and Dynamic Blocks

PART III — MODULES AND CODE ORGANIZATION

11 Writing Reusable Modules

12 Using Registry Modules and Managing Versions

13 Structuring Projects: Repositories, Environments, and Workspaces

14 Multiple Environments and Accounts

15 Providers in Depth: Configuration, Aliases, and Multi-Region

PART IV — BUILDING REAL INFRASTRUCTURE

16 Networking on AWS with Terraform

17 Compute and Containers: EC2, Auto Scaling, and ECS

18 Data, Storage, and Serverless

19 Multi-Cloud Basics: Azure and Google Cloud

20 Kubernetes and Helm with Terraform

PART V — QUALITY, SECURITY, AND OPERATIONS

21 Testing Terraform

22 Static Analysis, Security Scanning, and Policy as Code

23 Secrets and Sensitive Data

24 CI/CD for Terraform

25 Day-2 Operations: Drift, Cost, and Upgrades

PART VI — REAL-WORLD PROJECTS

26 Project: The Fieldmark Platform Foundation

27 Project: The Fieldmark Application Stack

28 Project: Scaling IaC Across Teams — OpenTofu, Terragrunt, Stacks, and Next Steps

APPENDICES

A Terraform CLI and HCL Cheat Sheet

B Module Templates and Reference

C Production Checklist

D Glossary

PART I

Foundations of Infrastructure as Code

Why we describe infrastructure as code, how Terraform works, and the language you will use to express every resource, variable, and dependency.

1 · Why Infrastructure as Code?

2 · Your First Deployment

3 · HCL in Depth

4 · Resources, Data Sources, and Dependencies

5 · Variables, Outputs, Locals, and Validation

CHAPTER 1 · PART I — FOUNDATIONS OF INFRASTRUCTURE AS CODE

Why Infrastructure as Code?

Before we write a single line of Terraform, we need a clear picture of the problem it solves. This chapter explains what goes wrong when infrastructure is built by hand, what "Infrastructure as Code" really means, how declarative tools like Terraform think, and where Terraform sits among the other tools you will hear about. We also meet Fieldmark, the company whose platform we will build throughout the book.

The Problem with Clicking

Almost every team's cloud journey starts in a web console. Someone needs a database for a prototype, clicks through a wizard, accepts most defaults, and copies the connection string into a config file. For a single experiment, that is often the right call.

The trouble starts when that experiment becomes a product. The database that was created by hand now holds customer data. A second engineer creates a staging copy, but chooses a slightly different instance class and forgets to enable automated backups. A third engineer opens a firewall port to debug a connection problem and never closes it. Six months later nobody can say with confidence how production is configured, why it is configured that way, or whether staging resembles it closely enough for testing to mean anything.

None of these engineers did anything unusual. The problems are structural, and each maps to a specific benefit of Infrastructure as Code.

Snowflake servers and environments

A *snowflake* is a server or environment that is unique in ways nobody fully understands. It was configured by hand, patched by hand, and adjusted over time in response to incidents. Because it is unique, it cannot be recreated with confidence. Teams become afraid to touch it, which means it drifts further from any documented baseline, which makes it more unique. The cycle feeds itself.

Configuration drift

Drift is the gap between what you believe your infrastructure looks like and what it actually looks like. Drift appears whenever a change is made outside the agreed process: a hotfix in the console at two in the morning, a tag edited by a cost-reporting script, a security group rule added "temporarily". Manual environments drift constantly, and because nothing records the intended state, drift is invisible until it causes an outage or an audit finding.

Knowledge trapped in people

When infrastructure is built by hand, the real documentation lives in the heads of the people who built it. Wiki pages describe what someone intended on the day they wrote them, not what was actually done. When those people leave, the knowledge leaves with them.

Slow, risky recovery

If an account is compromised or a critical resource is deleted by mistake, how long would it take to rebuild? With manual infrastructure, the honest answer is usually "we don't know": every decision must be reconstructed from memory, under pressure.

No review, no history, no audit trail

Application code goes through pull requests, reviews, and tests. Manual infrastructure changes usually skip all of that. The provider's audit log shows that an API call happened, but not *why* or who reviewed it — a serious gap under compliance requirements.

What Infrastructure as Code Means

Infrastructure as Code (IaC) is the practice of defining infrastructure — networks, servers, databases, queues, DNS records, permissions, and everything else — in machine-readable files that are stored in version control and applied by automated tooling. Instead of clicking through a console, you describe what you want in code, review that code like any other change, and let a tool make the real world match the description.

That short definition hides several properties that matter in practice:

- **Versioned.** Every change to infrastructure is a commit. You can see who changed what, when, and (if your commit messages are any good) why. You can compare today's configuration with last quarter's.
- **Reviewable.** Changes go through the same pull-request workflow as application code. A colleague can spot the missing backup setting before it reaches production rather than after.
- **Repeatable.** The same code can create the same infrastructure again — in another environment, another region, or another account — with differences expressed explicitly as inputs rather than hidden in someone's memory.
- **Testable.** Because infrastructure is code, you can lint it, scan it for security problems, run automated tests against it, and preview its effects before applying them. Part V of this book covers these practices in depth.
- **Self-documenting.** The code is an accurate description of the infrastructure it manages, at least for the parts it manages. It is not a perfect substitute for architecture documentation, but it never goes stale in the way a wiki page does.
- **Automatable.** Once infrastructure changes are code, they can flow through CI/CD pipelines with approvals, policy checks, and audit trails built in.

It is important to be honest about what IaC is *not*. It is not a guarantee of correctness: you can encode a bad design in code just as easily as you can click one together. It does not eliminate the need to understand the underlying cloud services; if anything, it demands a more precise understanding, because you must name every setting explicitly. And it is not free: it introduces new concepts (like Terraform's state, covered in Part II), new tooling, and new failure modes. We think the trade is overwhelmingly worth it for any infrastructure that matters, but you should go in with your eyes open.

Declarative vs Imperative

There are two broad ways to describe infrastructure in code. The difference between them is the single most important idea in this chapter.

The imperative approach

An **imperative** approach describes the *steps* to take. A shell script that calls a cloud CLI is the classic example. Suppose Fieldmark wants a message queue and a server to process sensor readings. An imperative script might look like this:

scripts/create-ingest.sh

```
#!/usr/bin/env bash
set -euo pipefail

# Step 1: create the queue
aws sqs create-queue --queue-name fm-dev-ingest-readings

# Step 2: launch a worker instance
aws ec2 run-instances \
  --image-id ami-0123456789abcdef0 \
  --instance-type t3.small \
  --tag-specifications \
    'ResourceType=instance,Tags=[{Key=Name,Value=fm-dev-ingest-worker}]'
```

This script works the first time. Now run it a second time. Depending on the API, some steps may succeed harmlessly and others will not: `run-instances`, for example, launches a *second* worker, because the command means "launch an instance", not "make sure one instance exists". To make the script safe to rerun, you must add checks:

```
# Only launch a worker if none is running (fragment)
existing=$(aws ec2 describe-instances \
  --filters "Name=tag:Name,Values=fm-dev-ingest-worker" \
            "Name=instance-state-name,Values=pending,running" \
  --query 'Reservations[].Instances[].InstanceId' --output text)

if [ -z "$existing" ]; then
  aws ec2 run-instances ...   # as before
fi
```

And that only handles *creation*. What if you want to change the instance type? You need another branch that detects the current type, compares it with the desired type, and stops, modifies, and restarts the instance. What if you want to delete the worker? Another script. What if two engineers run the script at the same time? Now you need locking. Every new requirement multiplies the logic, and the script slowly turns into a home-grown, under-tested infrastructure tool.

The declarative approach

A **declarative** approach describes the *end state* you want, and leaves the steps to the tool. Here is the same intent in Terraform's configuration language, HCL:

ingest.tf

```
resource "aws_sqs_queue" "readings" {
  name = "fm-dev-ingest-readings"
}

resource "aws_instance" "worker" {
  ami           = "ami-0123456789abcdef0"
  instance_type = "t3.small"

  tags = {
    Name = "fm-dev-ingest-worker"
  }
}
```

This file doesn't say "create". It says "there should be a queue with this name and an instance with these properties". When you run Terraform, it compares this description with what it knows already exists and works out the steps itself:

- If nothing exists yet, it creates both resources.
- If both already exist and match, it does nothing.
- If you change `instance_type` to `t3.medium`, it updates the instance.
- If you delete the `aws_instance` block, it destroys the instance.

All four behaviors come from the same eight lines. You don't write separate create, update, and delete scripts; you edit the description and let the tool compute the difference. This is why declarative tools scale so much better than scripts as infrastructure grows.

Aspect	Imperative (scripts)	Declarative (Terraform)
You describe	Steps to perform	Desired end state
Rerunning	Needs manual guards	Safe by design
Changes	Separate logic per change	Edit the description
Deletion	Separate script	Remove the block
Preview	Rarely available	Built in (`plan`)

Flexibility	Anything the CLI can do	What providers support

Declarative isn't better in every dimension. Scripts suit genuinely procedural operations, such as "drain this node, run a migration, then restart". Declarative tools shine when the thing you manage *has* a stable desired state — which describes most infrastructure.

> **NOTE**
> The line between the two styles blurs in practice. Pulumi and the AWS CDK let you write infrastructure in general-purpose languages with loops and functions, but the result is still a declarative desired state that an engine reconciles. Ansible playbooks read as ordered steps, yet most Ansible modules are written to be idempotent. When comparing tools, ask "what does the engine reconcile?" rather than "what does the syntax look like?"

Idempotence and Convergence

Two related ideas explain why declarative tools are safe to run repeatedly.

An operation is **idempotent** if performing it once has the same effect as performing it many times. "Set the thermostat to 20 degrees" is idempotent; "raise the temperature by two degrees" is not. Terraform's apply operation aims to be idempotent: if the real infrastructure already matches your configuration, applying again should report that there is nothing to do.

Convergence describes what happens when reality and the description differ. A converging tool moves the real system *toward* the desired state, whatever the starting point. If someone deletes a resource by hand, the next Terraform run notices the resource is missing and proposes to recreate it. If someone changes a setting in the console, the next run notices the difference and proposes to change it back. Terraform converges when you run it — it is not a background process that continuously corrects drift. That is a deliberate design choice with real consequences, and we will return to it in Chapter 25 when we discuss drift detection.

In practice, idempotence depends on the provider and the underlying API behaving well. Some APIs normalize values (for example, reformatting a JSON policy document), which can make Terraform see a difference on every run, and some settings can only be set at creation time, so changing them replaces the resource. Chapters 4 and 8 cover these edge cases.

How Terraform Works at a Glance

Terraform is a command-line tool, written in Go, that reads configuration files, talks to cloud and service APIs through plugins, and records what it manages in a state file. At a high level, a Terraform run has four ingredients:

1. **Configuration** — your `.tf` files, written in HCL (the HashiCorp Configuration Language), describing the desired state.

2. **Providers** — plugins that translate Terraform's generic operations (create, read, update, delete) into specific API calls. The `hashicorp/aws` provider knows how to create an S3 bucket; the `hashicorp/azurerm` provider knows how to create an Azure resource group. There are providers for major clouds, SaaS products, databases, DNS services, and much more.
3. **State** — a record of which real objects correspond to which blocks in your configuration, along with their last known attributes. State is how Terraform knows that `aws_sqs_queue.readings` is the queue whose URL ends in `fm-dev-ingest-readings`. Part II is devoted to state.
4. **The core engine** — the part of Terraform that reads configuration and state, builds a dependency graph, and computes and executes a plan.



The everyday workflow maps onto a handful of commands. `terraform init` downloads the providers your configuration needs. `terraform plan` shows what would change. `terraform apply` makes the changes, after showing you the plan and asking for confirmation. `terraform destroy` removes everything the configuration manages. Chapter 2 walks through all four with a real (and cheap) example.

The **plan** step deserves emphasis. Before changing anything, Terraform shows which resources it will create, update, replace, or destroy. Reading plans carefully is the most important safety habit in Terraform, and Chapter 8 is dedicated to it.

The Tool Landscape

Terraform is not the only IaC tool, and choosing one is a real decision with trade-offs. Here is a fair, brief tour of the tools you are most likely to encounter.

Terraform

Terraform, created by HashiCorp, uses the declarative HCL language and a large ecosystem of providers to manage infrastructure across many clouds and services with one workflow. Its strengths are breadth (one tool for AWS, Azure, Google Cloud, Kubernetes, DNS, monitoring, identity providers, and more), a mature module ecosystem, and a plan-first workflow that many teams find easy to review. Its main costs are the need to manage state carefully and the limits of HCL compared with a general-purpose programming language. HCP Terraform (formerly Terraform Cloud) is HashiCorp's hosted service for running Terraform with remote state, policy, and collaboration features.

OpenTofu

OpenTofu is a community fork of Terraform maintained under the Linux Foundation. It started from Terraform's code base, uses the same HCL language and provider protocol, and is largely compatible with Terraform configurations: most of what you learn in this book applies to OpenTofu, whose command is `tofu` rather than `terraform`. The two projects have diverged somewhat since the fork. OpenTofu has added features of its own, such as client-side state encryption, and newer Terraform features do not always appear in OpenTofu in the same form (or vice versa). We discuss the migration considerations in Chapter 28.

Pulumi

Pulumi lets you define infrastructure using general-purpose programming languages such as TypeScript, Python, Go, C#, and Java. Under the hood it is still declarative: your program builds a desired-state graph, and the Pulumi engine computes a preview and applies the difference, much like Terraform. Pulumi appeals to teams who want real functions, classes, and familiar testing frameworks; the trade-off is that infrastructure code can become as complex as application code.

AWS CloudFormation and the AWS CDK

CloudFormation is AWS's native IaC service. You write templates in JSON or YAML, and AWS manages the deployment and the state (as "stacks") for you, so there is no state file to store or lock. It supports change sets for previewing changes, but its scope is AWS (plus some extensions) rather than other clouds or SaaS products. The AWS Cloud Development Kit (CDK) lets you write infrastructure in TypeScript, Python, Java, C#, or Go and synthesizes CloudFormation templates from it — combining a programming-language experience with CloudFormation's managed deployment engine.

Ansible and configuration management

Ansible is primarily a *configuration management* and automation tool: it connects to machines (usually over SSH) and brings their software and settings into a desired state using mostly idempotent modules. It can also call cloud APIs to create resources, but it does not keep a state file mapping configuration to real objects, so deletion and drift handling work quite differently from Terraform. Many teams use both: Terraform to create the infrastructure, and Ansible (or a similar tool) to configure what runs inside long-lived servers.

Crossplane

Crossplane extends Kubernetes so that cloud resources are declared as custom resources and continuously reconciled by controllers in a cluster. It suits organizations that have standardized on Kubernetes as their control plane; the cost is that you need a cluster to manage your infrastructure.

Tool	Language	Scope	State
Terraform	HCL	Multi-cloud, SaaS	State file (you manage)
OpenTofu	HCL	Multi-cloud, SaaS	State file (you manage)
Pulumi	TS, Python, Go, C#, Java	Multi-cloud, SaaS	Service or self-managed
CloudFormation	JSON / YAML	AWS	Managed by AWS
AWS CDK	TS, Python, Java, C#, Go	AWS	Via CloudFormation
Ansible	YAML playbooks	Hosts, some cloud	No persistent state
Crossplane	Kubernetes YAML	Multi-cloud	Kubernetes API

How to choose

There is no universally correct answer, but a few questions narrow the choice quickly:

- **How many platforms do you manage?** If you are all-in on one cloud and happy with its native tooling, CloudFormation or the CDK are reasonable choices. If you manage several clouds, or cloud plus SaaS (DNS, CDN, monitoring, identity), a multi-provider tool like Terraform, OpenTofu, or Pulumi avoids learning several systems.
- **Who will read and review the code?** HCL is deliberately limited, which makes it readable by people who are not full-time programmers. General-purpose languages are more powerful but raise the bar for reviewers.
- **What does your ecosystem already use?** Existing modules, internal expertise, and CI/CD integrations matter as much as language features.
- **What licensing and governance model fits your organization?** This brings us to an important piece of history.

Licensing History and What It Means for You

For most of its life, Terraform was released under the Mozilla Public License 2.0 (MPL 2.0), an open-source license. In August 2023, HashiCorp changed the license of Terraform (and several of its other products) to the Business Source License 1.1 (BSL 1.1), a source-available license that places restrictions on certain competitive uses. In response, a group of companies and community members created a fork of the last MPL-licensed version. That fork became **OpenTofu**, which is hosted by the Linux Foundation and continues under MPL 2.0.

Since then, both projects have continued to develop. They remain largely compatible at the configuration level, but they have diverged in some features and release timelines, so "Terraform" and "OpenTofu" should not be treated as perfectly interchangeable names. Separately, IBM acquired HashiCorp, with the acquisition completed in 2025, and HashiCorp's hosted service formerly called Terraform Cloud is now named **HCP Terraform**.

What does this mean for you? For many teams — those using Terraform to manage their own infrastructure — day-to-day usage looks the same as before. But whether a particular license fits a particular use case is a legal question, not a technical one, and it depends on your organization and what you build. This book does not give legal advice. If licensing matters to your situation (for example, if you are building a product or service around infrastructure tooling), review the license text for your use case and involve your legal team.

> **TIP**
> This book uses Terraform and the `terraform` command throughout. If your organization has chosen OpenTofu, you can follow along with `tofu` in most places. Where a feature is specific to Terraform or HCP Terraform, we say so; when in doubt, check the OpenTofu documentation for the equivalent.

Where Infrastructure as Code Stops

A common source of frustration is trying to make one tool do everything. It helps to draw rough boundaries early.

- **Provisioning** — creating and configuring cloud resources such as networks, databases, queues, clusters, and IAM roles. This is Terraform's core job.
- **Configuration management** — installing packages and managing files inside long-lived servers. Terraform *can* pass startup scripts to instances, but it is not designed to manage the internals of a running machine over time.
- **Application deployment** — shipping new versions of your code many times a day. Terraform can create the ECS service or Lambda function, but high-frequency releases are usually better handled by a deployment pipeline or GitOps tool, with Terraform managing the surrounding infrastructure. Chapter 17 and Chapter 20 revisit this boundary.
- **Data** — the contents of your databases and buckets. Terraform creates the bucket; it should not manage the millions of sensor readings inside it.

The Costs and Trade-offs of IaC

We promised honesty about trade-offs, so here are the costs you will pay when you adopt Terraform:

- **Learning curve.** Your team must learn HCL, the Terraform workflow, the concept of state, and the specific behaviors of providers. This book is designed to shorten that curve, but it is real.

- **State management.** Terraform's state file is critical: lose it or corrupt it and Terraform loses track of what it manages. State can also contain sensitive values. You must store it remotely, lock it, encrypt it, and control access to it. Part II explains how.
- **Slower "first time".** Writing code for a resource takes longer than clicking through a wizard once. The payoff comes the second, third, and hundredth time.
- **Abstraction leaks.** Terraform sits on top of cloud APIs, and those APIs have quirks: eventual consistency, values normalized by the service, settings that force replacement. You will sometimes need to understand the API beneath the provider.
- **Process discipline.** IaC only works if changes actually go through the code. A team that keeps making console changes "just this once" ends up with drift and loses trust in its plans.

The last point matters most: Infrastructure as Code is as much a team agreement as a technology.

> **WARNING**
>
> Don't adopt Terraform by importing everything at once and then continuing to change resources in the console. Mixed ownership, where Terraform and humans both edit the same resource, is worse than either approach alone: every plan shows unexpected changes, reviewers learn to ignore diffs, and eventually someone applies a plan that silently reverts an important manual fix. Decide which resources Terraform owns and make that ownership exclusive.

Meet Fieldmark

Throughout this book we follow a fictional company called **Fieldmark**, an agriculture-technology business. Farms install Fieldmark's soil-moisture and weather sensors in their fields. The sensors send readings to Fieldmark's cloud platform, which powers a web and mobile app for farmers, sends alerts about frost, drought, and irrigation needs, and provides analytics dashboards that help agronomists spot trends across seasons.

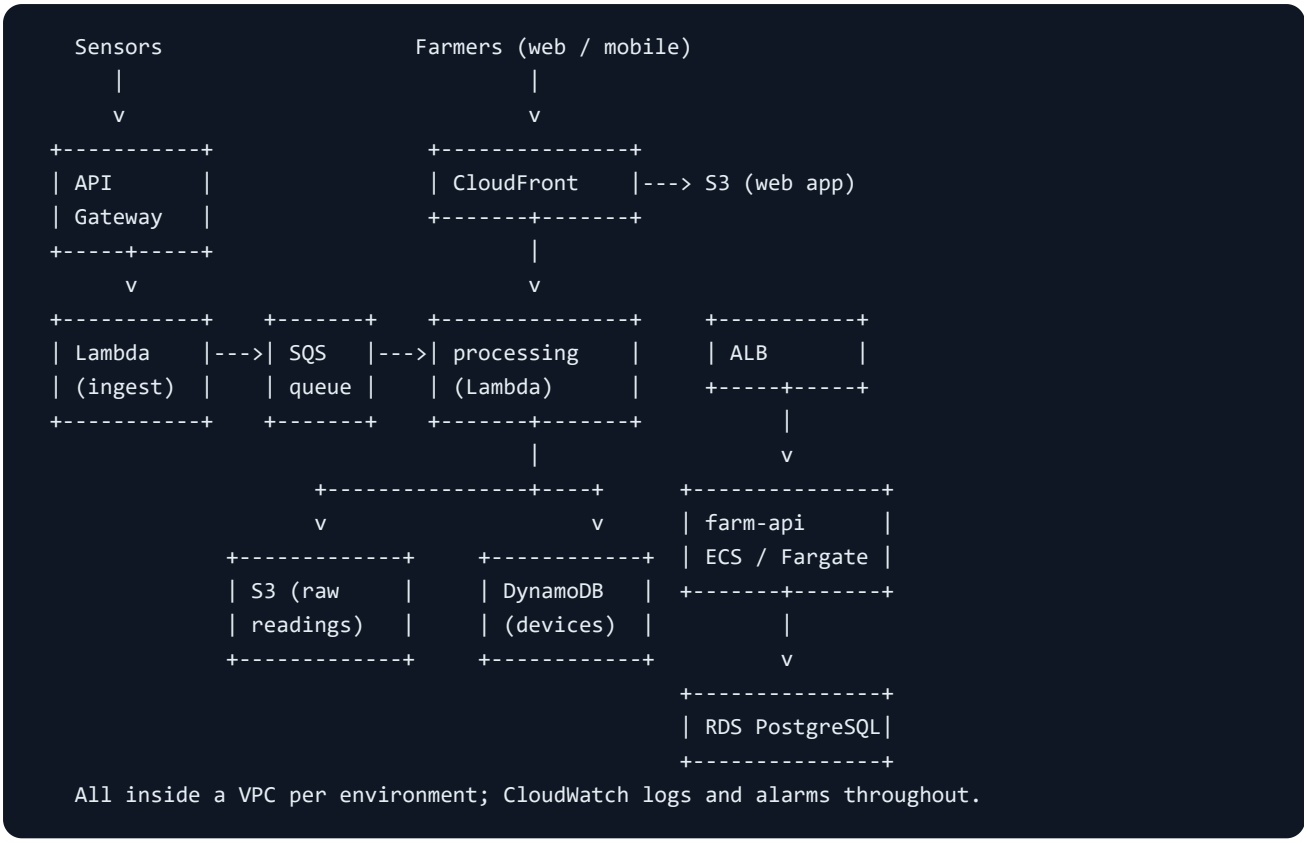
Fieldmark is a useful example because its needs are typical of a growing platform: an ingestion path that must not lose readings, a customer-facing API with a relational database, static web assets behind a CDN, and several environments that must stay consistent.

Fieldmark's starting point

When we meet Fieldmark's platform team, the first version of the platform was built by hand in a single AWS account. It works, but staging doesn't quite match production, nobody is sure which security group rules are still needed, and a plan to launch in a second region stalled because nobody could list everything that would need recreating. The team has decided to move to Terraform, and we will build their platform piece by piece.

The target architecture

By the end of Part VI, Fieldmark's platform will look roughly like this:



The major components are:

- **Accounts and environments.** Separate AWS accounts for `dev`, `staging`, and `prod`, plus a `shared` account for networking and tooling. Separate accounts give each environment a hard security and billing boundary.
- **Networking.** A VPC per environment with public and private subnets across availability zones, NAT for outbound traffic, and VPC endpoints for AWS services.
- **`ingest`.** Sensor readings arrive through an HTTPS API (API Gateway and Lambda) and are placed on an SQS queue for processing. Raw readings land in S3; a DynamoDB table acts as the device registry.
- **`farm-api`.** A containerized service running on ECS with Fargate behind an Application Load Balancer, backed by a PostgreSQL database on RDS.
- **`web`.** The farmer-facing web app, served as static files from S3 through CloudFront, with DNS in Route 53 under `fieldmark.example.com`.
- **Observability.** CloudWatch logs and alarms, so the team hears about failed ingestion before farmers do.

Later chapters add a Kubernetes cluster (Chapter 20) and a small Azure and Google Cloud footprint for a partner integration (Chapter 19), but the core platform stays on AWS.