Infrastructure as Code Testing

Terraform to Kubernetes

modernQAcourse.com

THE MODERN QA ENGINEER SKILLSET

Infrastructure as Code Testing: Terraform to Kubernetes

Validating Terraform, Kubernetes, and Cloud Declarations Before They Become Production Reality

Yuri Syuganov

Modern QA Course

https://modernQAcourse.com

Infrastructure as Code Testing: Terraform to Kubernetes

Validating Terraform, Kubernetes, and Cloud Declarations Before They Become Production Reality

Book 8 of 26 in the series *The Modern QA Engineer Skillset*

First Edition · Version 1.0.0 · 2026-07-11

Set in Charter, IBM Plex Sans, and IBM Plex Mono. Typeset with Pandoc and XeLaTeX.

Modern QA Course · https://modernQAcourse.com · support@modernQAcourse.com

Contents

Introduction 1

PART I FOUNDATIONS OF INFRASTRUCTURE AS CODE TESTING 5

1. The Infrastructure Testing Pyramid 7

About This Sample 17

About the Author 19

Also in This Series 21

Stuck on a Real Problem? 23

Introduction

About This Book

This book is a hands-on, comprehensive guide to testing infrastructure as code. It is written for QA engineers who recognize that the boundary between application code and infrastructure code has dissolved – and that their testing skills must expand to cover both.

In 2026, a single pull request may contain a React component, its API handler, a Terraform module for the DynamoDB table it reads from, and a Kubernetes manifest for its deployment. If your testing strategy only covers the application layer, you are ignoring the foundation the application stands on.

Infrastructure misconfigurations are the leading cause of cloud security incidents. QA engineers who can validate infrastructure code bring disproportionate value to their organizations. This book will teach you how to become that engineer.

The cloud does not care about your intentions. It only enforces what you declared. Testing infrastructure as code means verifying your declarations before they become expensive, insecure, or irreversible production realities.

Who This Book Is For

This book is designed for:

- **QA engineers** who want to expand their skills into infrastructure testing
- **DevOps engineers** who want a systematic approach to validating IaC
- **Platform engineers** building internal developer platforms with quality guardrails
- **Software engineers** who write Terraform, Kubernetes manifests, or Dockerfiles and want to test them properly
- **Engineering managers** who want to understand what a mature IaC testing pipeline looks like

You do not need to be an infrastructure expert. This book starts from fundamentals and builds to advanced topics. What you do need is curiosity and a willingness to work through the labs.

Prerequisites

Before starting this book, you should have:

- **Basic cloud familiarity**: Experience with at least one cloud provider (AWS, GCP, or Azure). You should know what an S3 bucket, EC2 instance, or virtual network is, even if you have never created one via code.
- **Basic Terraform knowledge**: Understanding of HCL syntax, resources, variables, and outputs. If you have never written Terraform, complete the official "Get Started" tutorial first.
- **Docker fundamentals**: Ability to build and run containers. Understanding of Dockerfiles, images, and containers.
- **Command-line comfort**: Familiarity with bash, git, and running CLI tools.
- **Programming basics**: Ability to read Python, Go, or TypeScript code. You do not need expertise in all three – the concepts transfer.

No Kubernetes expertise is assumed. The Kubernetes chapters start from manifest basics.

How to Use This Book

This book is designed for sequential learning, but each part can also serve as a standalone reference.

If you are new to IaC testing: Read Parts I through VII in order. Each part builds on concepts from the previous one. Complete the exercises before moving to the next chapter.

If you have some experience: Skip to the parts that address your knowledge gaps. Use the self-assessment quizzes at the end of each chapter to verify your understanding.

If you are preparing for interviews: Read the Key Takeaways sections and the capstone project in Part VIII. The interview talking points throughout the book are specifically designed for technical interviews.

Difficulty ratings for exercises:

- [Beginner] – Can be completed with the information in this chapter alone
- [Intermediate] – Requires combining concepts from multiple chapters
- [Advanced] – Requires independent research beyond the book content
- [Expert] – Open-ended challenges with no single correct answer

Callout boxes throughout the book:

- **Common Mistake** boxes highlight errors that QA engineers frequently make when starting with IaC testing
- **Pro Tip** boxes share practical wisdom from production experience

PART I

FOUNDATIONS OF INFRASTRUCTURE AS CODE TESTING

CHAPTER 1

The Infrastructure Testing Pyramid

1.1 Why QA Engineers Must Own Infrastructure Testing

The era of "throw it over the wall to ops" is over. Modern software delivery is a continuous pipeline where application code and infrastructure code ship together. A feature is not just a React component and an API handler – it is also the database table, the message queue, the load balancer rule, and the IAM policy that make it work.

When a developer opens a pull request that modifies a Terraform module, someone needs to verify that the change is safe, secure, and compliant. That someone should be a QA engineer – because QA engineers bring a testing mindset that developers and ops engineers often lack.

Developers think about making things work. Ops engineers think about keeping things running. QA engineers think about what can go wrong. That adversarial mindset is exactly what infrastructure testing needs.

The Cost of Getting It Wrong

Infrastructure misconfigurations are not theoretical risks. They are the leading cause of cloud security incidents:

Incident Type	Root Cause	Detection Method
Data breach via public S3 bucket	Missing bucket policy	Static analysis (Checkov)
Database deletion during deployment	Unreviewed resource replacement	Plan-time assertion
Production outage from IAM change	Overly permissive policy removed	Integration test (Terratest)
Crypto-mining on open security group	0.0.0.0/0 ingress on port 22	Policy-as-code (OPA)
Compliance violation from unencrypted storage	Missing encryption configuration	Automated compliance scan

Each of these incidents could have been prevented by testing infrastructure code before deployment. The tools exist. The practices are well-defined. What is often missing is a QA engineer who owns the process.

1.2 The Validation Pyramid

Infrastructure validation follows a pyramid similar to the traditional test pyramid, but with its own layers. Each layer catches different classes of issues at different costs:

```
        /\
       /  \        Integration tests (Terratest, real cloud resources)
      /    \
     /------\      Plan-time analysis (terraform plan, drift detection)
    /        \
   /----------\    Static analysis (validate, tfsec, checkov)
  /            \
 /--------------\  Syntax and format (terraform fmt, terraform validate)
/________________\
```

The bottom of the pyramid is free and fast. The top is expensive and slow but catches issues nothing else can. A mature IaC testing strategy uses all four layers, running the cheap checks on every commit and the expensive checks on critical module changes.

Layer 1: Syntax and Format (Milliseconds, Zero Cost)

These checks run in milliseconds, require no cloud credentials, and catch typos, missing fields, and style inconsistencies. Every commit should pass these checks before anything else runs.

Tools: `terraform fmt`, `terraform validate`, `helm lint`, `kubeval`

Layer 2: Static Analysis (Seconds, Zero Cost)

Static analysis tools scan your code for known security misconfigurations, compliance violations, and best practice deviations. They check against databases of thousands of rules without ever touching a cloud API.

Tools: tfsec, Checkov, Trivy (config mode), OPA/Conftest, kube-score, Polaris

Layer 3: Plan-Time Analysis (Seconds to Minutes, Low Cost)

Running `terraform plan` shows you exactly what will change in your infrastructure. Automated assertions against the plan output catch dangerous operations like resource deletions, replacements, and security group changes.

Tools: `terraform plan` + custom scripts, Pulumi preview, AI review agents

Layer 4: Integration Tests (Minutes to Hours, High Cost)

Integration tests deploy real infrastructure, run assertions against it, and tear it down. This is the only way to verify that your code actually works with real cloud APIs, but it costs real money and takes real time.

Tools: Terratest, Pulumi Automation API, ephemeral environments

> **Common Mistake**: Many teams skip straight to integration tests because they feel more "real." This is a mistake. A single Terratest run can take 15-30 minutes and cost dollars. A static analysis scan takes 10 seconds and costs nothing. Always build from the bottom of the pyramid up.

> **Pro Tip**: The fastest way to improve your IaC testing maturity is to add a pre-commit hook that runs `terraform fmt -check` and `terraform validate`. This takes 5 minutes to set up and catches a surprising number of issues before they ever reach CI.

1.3 Mapping the Testing Landscape

The IaC testing ecosystem is vast. Here is how the major tools map to the pyramid layers:

Tool	Layer	What It Tests	Language	Cost
terraform fmt	Syntax	Code formatting	N/A	Free
terraform validate	Syntax	HCL structure	N/A	Free
TFLint	Static	Provider-specific rules	Go	Free
tfsec	Static	Security misconfigs	Go	Free
Checkov	Static	Security + compliance	Python	Free
Trivy (config)	Static	Multi-format scanning	Go	Free
OPA/Conftest	Static	Custom policies	Rego	Free
kube-score	Static	K8s best practices	Go	Free
Polaris	Static	K8s policy enforcement	Go	Free
kubeconform	Static	K8s schema validation	Go	Free
helm-unittest	Static	Helm template logic	YAML	Free
terraform plan	Plan	Change analysis	N/A	Free*
Terratest	Integration	Real infrastructure	Go	$$$
Pulumi Automation API	Integration	Real infrastructure	TS/Py/Go	$$$
LocalStack	Integration	AWS emulation	Python	Free
Testcontainers	Integration	Docker-based deps	Multi	Free

*terraform plan requires cloud credentials but does not create resources.

1.4 The Complete IaC Testing Pipeline

Before we dive into individual tools, let us see the complete picture. This is what a mature IaC testing pipeline looks like:

Figure 1 summarizes this design.

```
+------------------------------------------------------------------+
| Developer pushes Terraform + K8s changes                         |
|                                                                  |
|  Stage 1: Static (seconds)                                       |
|  +-- terraform fmt --check                                       |
|  +-- terraform validate                                          |
|  +-- checkov / tfsec / trivy config                              |
|  +-- kubeval / kubeconform                                       |
|  +-- OPA/Conftest custom policies                                |
|                                                                  |
|  Stage 2: Plan Analysis (30 seconds)                             |
|  +-- terraform plan -> JSON                                      |
|  +-- Automated plan assertions (no destroys, no public SGs)      |
|  +-- AI agent review of plan diff                                |
|                                                                  |
|  Stage 3: Ephemeral Deploy (5-15 minutes)                        |
|  +-- terraform apply to PR workspace                             |
|  +-- Wait for health checks                                      |
|  +-- Post preview URL                                            |
|                                                                  |
|  Stage 4: Integration Tests (5-20 minutes)                       |
|  +-- API tests against ephemeral environment                     |
|  +-- E2E browser tests                                           |
|  +-- Security scan (DAST)                                        |
|  +-- Performance baseline                                        |
|                                                                  |
|  Stage 5: Cleanup (on merge/close)                               |
|  +-- terraform destroy PR workspace                              |
+------------------------------------------------------------------+
```

Figure 1. A complete infrastructure-as-code testing pipeline

Each subsequent chapter in this book corresponds to one or more of these stages. By the end, you will be able to implement this entire pipeline.

Self-Assessment Quiz: Chapter 1

1. What are the four layers of the infrastructure validation pyramid, from bottom to top?
2. Which layer of the pyramid catches the most issues per dollar spent?
3. Why is `terraform validate` insufficient on its own for ensuring infrastructure security?
4. Name two tools that operate at the static analysis layer.
5. When should you use integration tests (Terratest) vs. static analysis?
6. What is the primary risk of skipping plan-time analysis?

Key Takeaways

- Infrastructure misconfigurations are the leading cause of cloud security incidents
- The IaC testing pyramid has four layers: syntax, static analysis, plan analysis, and integration
- Always build testing from the bottom of the pyramid up – cheap checks first
- Static analysis runs in seconds and catches the majority of security issues
- Integration tests are expensive but irreplaceable for critical modules
- QA engineers bring a unique adversarial mindset that infrastructure testing needs

Career Translation

Resume phrasing

- Designed and implemented multi-layer infrastructure validation strategy covering syntax, static analysis, plan-time assertions, and integration testing across Terraform and Kubernetes pipelines
- Reduced cloud security incidents by establishing automated IaC testing pyramid, catching 90%+ of misconfigurations before deployment

- Led adoption of infrastructure-as-code quality practices across engineering teams, integrating tools such as Checkov, OPA, Trivy, and Terratest into CI/CD workflows

Cover letter framing

I bring a QA-first perspective to infrastructure code, treating Terraform modules and Kubernetes manifests with the same rigor as application code. By structuring infrastructure validation into a cost-effective pyramid – cheap static checks at the base, expensive integration tests at the top – I help organizations catch dangerous misconfigurations early while keeping feedback loops fast. This approach directly reduces cloud security incidents and deployment failures.

Interview framing

"I approach infrastructure testing by applying the testing pyramid principle: start with zero-cost static checks like terraform fmt and validate, layer on security scanners like Checkov and tfsec, add plan-time assertions against the terraform plan JSON, and reserve real-infrastructure integration tests for critical reusable modules. I optimize for fast feedback at the bottom of the pyramid and high confidence at the top. Trade-offs include balancing test coverage against CI pipeline speed and cloud costs for integration tests."

What not to say

- "We just run terraform apply and see if it works" – shows no proactive testing mindset and treats production as the test environment
- "Static analysis catches everything" – ignores that static tools cannot verify runtime behavior or real cloud API interactions
- "Integration tests are too expensive so we skip them" – demonstrates unwillingness to invest in confidence for critical infrastructure modules
- "I let the DevOps team handle infrastructure quality" – signals you view infrastructure testing as someone else's responsibility

Interview Depth Check

Question 1

Prompt: Your team is introducing infrastructure testing for the first time on a Terraform-managed AWS environment. The CTO wants full coverage but the team has limited bandwidth. How would you prioritize the rollout?

What a strong answer should cover:

- Start from the bottom of the pyramid: pre-commit hooks with terraform fmt and validate first
- Add static analysis (Checkov or tfsec) as a CI gate next – zero cost, high value
- Introduce plan-time assertions for critical safety checks (no destroys, no public access)
- Defer integration tests (Terratest) to critical shared modules only
- Justify each layer by cost-to-value ratio

Example answer:

- "I would start with pre-commit hooks for terraform fmt and validate – five minutes to set up, catches formatting and syntax issues on every commit"
- "Next, I would add Checkov to CI, which ships with 1000+ built-in rules and requires zero configuration. This catches security misconfigurations like public S3 buckets and unencrypted databases"
- "For plan-time analysis, I would write Python assertions against the terraform plan JSON to block resource deletions and public access patterns"
- "Integration tests with Terratest would come last, targeted only at shared modules like the VPC or RDS module that multiple teams depend on"
- "This sequence delivers increasing value at each step while keeping initial investment low"

Question 2

Prompt: A developer argues that terraform validate is sufficient for ensuring infrastructure quality. How do you respond?

What a strong answer should cover:

- terraform validate checks syntax and structure, not security or compliance
- Specific examples of what validate accepts but is dangerous (public S3, open security groups, wildcard IAM)
- The role of static analysis tools in catching semantic issues
- Framing it as complementary layers, not either/or

Example answer:

- "terraform validate confirms that HCL parses correctly and required arguments are present, but it happily accepts an S3 bucket with public-read ACL, an RDS instance without encryption, or a security group open to 0.0.0.0/0 on all ports"
- "These are all syntactically valid Terraform but represent serious security risks"
- "Static analysis tools like Checkov and tfsec check against databases of security rules that validate is not designed to enforce"
- "I frame it as layers: validate ensures your code is correct HCL, static analysis ensures it is safe HCL"

Question 3

Prompt: You are evaluating IaC testing tools for a greenfield project. Walk me through your selection criteria and which tools you would choose.

What a strong answer should cover:

- Map tools to pyramid layers (syntax, static, plan, integration)
- Consider team language preferences, cloud provider, and CI platform
- Defense in depth – use multiple tools with different rule databases
- Cost considerations for integration testing

Example answer:

- "I map tool selection to the validation pyramid. For syntax: terraform fmt and validate plus TFLint for provider-specific rules. For static analysis: Checkov for broad built-in coverage plus OPA/Conftest for custom organizational policies"
- "For plan-time: Python scripts asserting against the plan JSON. For integration: Terratest for critical shared modules"
- "I use at least two static analysis tools because different scanners have different advisory databases – Checkov may catch issues tfsec misses and vice versa"
- "For container scanning, Trivy covers images, filesystems, and IaC config in a single binary, which simplifies toolchain management"

About This Sample

You have just read **Chapter 1 of Infrastructure as Code Testing: Terraform to Kubernetes** — the free sample. The complete book continues with every remaining chapter, the interview Q&A, and all of the code.

Get the full book

Sample readers get a discount at
https://leanpub.com/modernqa-08-infrastructure-as-code-testing/c/SAMPLE10

Read online first

Chapter one of **every** book in the series is free to read at
https://modernqacourse.com/library

About the Author

Yuri Syuganov is a software quality engineer with more than twenty years of hands-on testing experience, spanning the arc from manual test design through Selenium-era automation to today's AI-augmented, agent-driven frameworks. He is the author of The Modern QA Engineer Skillset, a 26-book series built on a blunt analysis of what companies actually expect from QA engineers, and the creator of modernQAcourse.com, where the course behind the series is free and readers can bring real testing problems to experienced practitioners. He revises the series as the industry changes — because it does.

Also in This Series

The Modern QA Engineer Skillset — a 26-book course in modern software quality assurance.

AI-Augmented QA (Books 1–10)

- **Book 1:** Agent Skills for Browser Automation: The Complete Guide
- **Book 2:** AI-Augmented Test Design: From Prompts to Test Suites
- **Book 3:** Agentic Testing Architectures: Building Autonomous QA Systems
- **Book 4:** API and Contract Testing with AI: Schema to Shield
- **Book 5:** Performance and Chaos Engineering: Breaking Things on Purpose
- **Book 6:** Observability-Driven Testing: Testing in Production
- **Book 7:** Security Testing for AI Applications: The New Attack Surface
- **Book 8:** Infrastructure as Code Testing: Terraform to Kubernetes
- **Book 9:** Mobile and Cross-Platform Testing: 24,000 Devices, One Strategy
- **Book 10:** Visual and Accessibility Testing: Pixels and People

Technical Foundations (Books 11–19)

- **Book 11:** Manual Testing Fundamentals: The Bedrock
- **Book 12:** Programming for QA: Python, JavaScript, and TypeScript from Zero to Fluent
- **Book 13:** Browser Test Automation: Playwright, Patterns, and the Death of Fragile Tests

- **Book 14:** API Testing Fundamentals: Postman to Pytest
- **Book 15:** SQL and Database Testing: Query Your Way to Quality
- **Book 16:** CI/CD Pipelines: From Commit to Confidence
- **Book 17:** Git and Version Control: Beyond Push and Pull
- **Book 18:** Test Management Tools: Making Quality Visible
- **Book 19:** Linux and Command Line: Your Production Survival Kit

Professional Skills (Books 20–26)

- **Book 20:** Agile and Scrum for QA: Quality in Every Sprint
- **Book 21:** Communication and Stakeholder Management for QA: Influence Without Authority
- **Book 22:** Test Strategy and Quality Metrics: Measuring What Matters
- **Book 23:** QA Leadership and Mentoring: Multiplying Your Impact
- **Book 24:** Technical Writing for QA: Documents That Drive Action
- **Book 25:** Interview Preparation and Career Strategy: Land the Job
- **Book 26:** Testing Like a Senior: Patterns & Anti-Patterns Across Software QA Career Levels

Visit https://modernQAcourse.com for the full course catalog, updates, and companion resources.

Stuck on a Real Problem?

The full course behind this series is free to read at https://modernQAcourse.com — and it stays free.

When a chapter meets the real world and something breaks, post it on the **Help Board** at https://modernQAcourse.com. Asking is free, and experienced practitioners answer right in the thread. Need hands-on help? Book a 1:1 screen-share session with a QA expert.