PRACTICAL #NIX FOR DEVOPS & PLATFORM ENGINEERING

"REPRODUCIBLE
DECLARATIVE
MAINTAINABLE
AT SCALE"

FROM ZERO TO PRODUCTION

REPRODUCIBLE SYSTEMS AT SCALE

- DEVELOPER ENVIRONMENTS
- REPRODUCIBLE BUILDS
- CLOUD-NATIVE INFRASTRUCTURE
- DEPLOYMENTS THAT SCALE
- RELIABLE BY DESIGN

JAMES HUBBARD

Practical Nix for DevOps & Platform Engineering

From Zero to Production — Reproducible Systems at Scale

Steve Publications

This book is available at
https://leanpub.com/practicalnixfordevopsplatformengineering

This version was published on 2026-07-31

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2026 Steve Publications

Contents

From Zero to Production — Reproducible Systems at Scale

Nix is the most powerful tool for building reproducible, declarative, and maintainable systems at scale, but its steep learning curve has kept it out of mainstream DevOps. This book bridges that gap by teaching practical patterns you can adopt incrementally, starting from individual developer environments all the way to cloud-native infrastructure. Every concept is explained with the why and when, every code example is complete and runnable, and every chapter builds on the last to take you from zero Nix knowledge to advanced production implementations. Whether you are a software engineer, DevOps engineer, SRE, platform engineer, or technical architect, this book gives you the hands-on guidance you need to build reliable, reproducible systems using Nix in real-world production environments.

Introduction: The Reproducibility Crisis

The Heisenbugs That Cost Teams

It starts with a single word from a junior developer on Slack: "it works on my machine."

You have heard this phrase enough times to know what comes next. The build passes in CI, the staging environment behaves perfectly, and then production erupts. A dependency was updated without notice, a library version drifted between environments, or some system package silently changed its behavior. The root cause is not a bug in your code, it is a gap between environments that no one could see until something broke.

These are Heisenbugs: failures that disappear when you try to observe them, because the act of debugging changes the environment just enough to hide the problem. They account for a disproportionate share of engineering time, and they are fundamentally caused by environmental drift. Traditional tooling does not solve this problem, it papered over it with conventions, documentation, and fragile automation scripts that themselves depend on the mutable state they are trying to manage.

The cost is real. A 2019 study from PagerDuty found that the average incident lasted 35 minutes, but the mean time to identify the root cause was often much longer when environmental differences were involved. Teams spend thousands of hours per year debugging issues that would not exist if their environments were truly reproducible.

Why Traditional Tooling Fails

Let us be clear about what traditional tooling does and does not do for reproducibility:

- **Docker** provides containerization, which isolates applications from the host system. But Docker images are built imperatively through Dockerfiles, where each layer depends on the exact state of the previous layer.

Two Dockerfiles that look identical can produce different images if their base layers have been updated. Reproducibility with Docker requires pinning every dependency, recreating it manually, and trusting that no one will accidentally use a newer base image.

- **Ansible, Chef, Puppet** manage system state through imperative recipes or declarative configurations, but they operate on mutable systems. They can bring a system to a desired state, but they cannot guarantee what happened between runs or roll back cleanly if something goes wrong. The underlying operating system remains mutable, and any package installed outside the configuration management tool creates drift.
- **Vagrant** provides reproducible development environments through virtual machines, but the VMs themselves are built on top of base images that can change, and provisioning scripts are still imperative. They solve a subset of the problem for a subset of use cases, at significant computational cost.
- **Terraform** manages infrastructure declaratively, but it does not manage the software running on that infrastructure. The gap between "I have a server" and "the server runs my application correctly" is where most reproducibility problems live.

The fundamental issue with all of these tools is that they treat the filesystem as a mutable resource. They install packages in place, overwrite files, and depend on global state. When something goes wrong, you are left trying to reverse-engineer what changed, when, and why.

A Different Approach: Declarative, Immutable Infrastructure

Nix takes a fundamentally different approach. Instead of treating the filesystem as mutable, Nix treats every piece of software as an immutable value stored in a content-addressable store at `/nix/store`. Every package gets its own unique path based on the cryptographic hash of everything that went into building it. Dependencies are never shared globally, they are linked through symlinks to their exact versions in the store.

This means:

1. **No dependency hell**: Two packages can coexist with different versions of the same library, because each package has its own private copy of every dependency.
2. **Atomic upgrades and rollbacks**: When you upgrade a system, Nix builds the new state independently of the old one. If something goes wrong, you can roll back to the previous generation instantly, because the old state is still intact in the store.
3. **Reproducible builds**: Given the same inputs, Nix always produces the same outputs. The hash in the path proves it. If someone else runs the same build on a different machine, they get the exact same result, bit for bit.
4. **Declarative configuration**: NixOS, the operating system built on top of Nix, lets you describe your entire system as a single declarative configuration file. You specify what you want, and Nix figures out how to build it. No imperative scripts, no state to track, no drift.
5. **Composable environments**: The same Nix language that builds packages also creates development environments, container images, and full operating systems. One toolchain for everything, from "run a single binary" to "deploy a multi-node cluster."

The trade-off is real: Nix has a steep learning curve. The mental model shift from mutable to immutable, from imperative to declarative, from global state to content-addressable isolation, takes time and practice. This book exists to make that journey as smooth as possible by focusing on practical patterns you can adopt immediately, with complete examples you can run and adapt.

What This Book Will Teach You

This book is organized to take you from zero Nix knowledge to advanced production implementations, building skills incrementally:

- **Chapters 1 and 2** establish the foundation. You will learn how Nix works under the hood, including its store model, hash-based addressing, and garbage collection. Then you will master the Nix language, the expression language that powers everything from package definitions to full system configurations.

- **Chapters 3 through 5** focus on development workflows. You will create reproducible development environments, manage packages with overlays, build packages from source for multiple languages, and cross-compile for different platforms. These are the skills that deliver immediate value to your team.
- **Chapters 6 and 7** cover system configuration. You will learn the NixOS module system, configure services, networking, security, and users declaratively, and extend that to user-level configuration with Home Manager.
- **Chapters 8 through 10** bring everything into production. You will set up CI/CD pipelines with binary caches, create container images, deploy to Kubernetes with GitOps, and integrate Nix with Terraform, OpenTofu, and Ansible for infrastructure automation.
- **Chapter 11** addresses the critical topic of secrets management, showing how to use SOPS, age encryption, and Vault integration with Nix.
- **Chapter 12** ties everything together into enterprise-scale patterns: monorepo strategies, build optimization, debugging, testing, performance tuning, maintenance, upgrades, and operational checklists.

By the end of this book, you will have the practical knowledge to evaluate whether Nix is right for your organization, adopt it incrementally without disrupting existing workflows, and build production-grade systems that are reproducible, maintainable, and scalable.

Chapter 1: Nix Fundamentals — How It Actually Works

Before you can use Nix effectively, you need to understand how it works under the hood. This is not optional background information, it is the foundation for everything that follows. If you try to use Nix without understanding its store model, hash-based addressing, and garbage collection, you will fight the tool constantly. Every error message, every unexpected behavior, every optimization opportunity will make sense once you internalize these concepts.

The Store Model and Content-Addressable Paths

The core innovation of Nix is the Nix store, a directory (by default `/nix/store`) that contains all packages and their dependencies in an immutable, content-addressable manner. Every file or directory in the store has a path that looks like this:

```
/nix/store/0q8k3z1f2h5m6v7w8x9y0a1b2c3d4e5f-hello-2.12
```

The path has two parts separated by the first hyphen:

1. **The hash prefix** (`0q8k3z1f2h5m6v7w8x9y0a1b2c3d4e5f`): A Base32-encoded SHA256 hash that is computed from the entire derivation of this store path, including all its inputs. If anything changes in the build process, the hash changes, and the path changes.
2. **The name** (`hello-2.12`): A human-readable identifier that describes what the package is and its version.

This design has profound implications:

- **Immutability**: Once a file exists at a path in the store, it cannot be modified. The only way to change it is to create a new path with a new hash. This eliminates accidental mutations and provides atomic upgrades.

- **Content-addressable**: You can verify the integrity of any store path by recomputing its hash and comparing it to the prefix. If they match, the contents are exactly what was intended.
- **No global state conflicts**: Since every package has its own path with all its dependencies linked via symlinks, there is no concept of "the system Python" or "the system libssl." Each package has its own private copy.

Let us look at a concrete example. Suppose you have two packages that both depend on OpenSSL, but one needs version 1.1.1 and the other needs version 3.0. In a traditional system, you would face dependency hell: installing one version breaks the other. In Nix, each package gets its own copy:

```
/nix/store/abc123...-openssl-1.1.1w
  -> /nix/store/def456...-package-needing-1.1.1/bin/myapp

/nix/store/ghi789...-openssl-3.0.2
  -> /nix/store/jkl012...-package-needing-3.0/bin/otherapp
```

The symlinks inside each package point to the exact version it needs. No conflicts, no global state, no surprises.

The store is world-readable but only writable by root. This is a security feature: since the hash proves integrity, anyone can read from the store, but only root (via the Nix daemon) can write to it. This prevents tampering and ensures that all builds go through the same controlled process.

How Hashes Create Deterministic Dependencies

The hash in a store path is not just a random identifier. It encodes the entire dependency tree of the package, which is what makes Nix reproducible. When Nix evaluates a derivation (a description of how to build something), it computes a hash that represents:

- The source code
- All direct dependencies
- All transitive dependencies (dependencies of dependencies)
- The build script and its parameters
- The environment variables during the build

If any of these inputs change, the hash changes. This means:

- **Reproducible builds**: Running the same derivation on two different machines produces the same hash, the same path, and the same output. If you get a different hash, something has changed in the inputs.
- **No hidden dependencies**: Every dependency is explicit in the derivation. You cannot accidentally depend on a package that happens to be installed on your system, because Nix builds each derivation in an isolated environment with only its declared dependencies available.
- **Cacheability**: If someone else has already built a derivation with the same hash, you can download their result instead of building it yourself. This is the foundation of binary caches.

Here is how the hash computation works in practice. Consider this simple derivation that builds a "hello" executable:

```
{ stdenv, fetchurl }:

stdenv.mkDerivation {
  pname = "hello";
  version = "2.12";
  src = fetchurl {
    url = "mirror://gnu/hello/hello-${version}.tar.gz";
    sha256 = "0ssi1wpaf7plaswqqjwigppsg5fyh99vdlb9kzl7c9nglx912184";
  };
}
```

The hash of the resulting store path depends on:

1. The `fetchurl` call, which downloads the source tarball and verifies its SHA256 hash. If the download produces a different hash, the build fails immediately.
2. The `stdenv.mkDerivation` function, which provides the standard build environment including GCC, make, and other tools. The specific versions of these tools are encoded in their own store paths.
3. The build phases: unpacking the source, configuring, building, and installing. Each phase's script is part of the derivation.

If you change the version from "2.12" to "2.13", every hash downstream changes: the source download hash, the derivation hash, and any package that depends on this hello binary. This cascading effect is what makes Nix both powerful and sometimes slow: a single change can invalidate a large portion of the dependency tree.

Dependency Graphs and the Nix Database

Nix maintains a database (typically at `/nix/var/nix/db`) that tracks the dependency relationships between all store paths. This database is essential for several operations:

- **Garbage collection**: To know what to delete when you clean up.
- **Querying**: To find out what depends on a particular package, or what a package depends on.
- **Closure computation**: To determine the complete set of files needed to run a program.

The dependency graph is a directed acyclic graph (DAG) where nodes are store paths and edges represent dependencies. When you install a package, Nix computes its closure: the minimal set of store paths that must exist for the package to work. For a simple binary like `hello`, the closure might include:

```
/nix/store/...-hello-2.12
  -> /nix/store/...-glibc-2.38
    -> /nix/store/...-linux-pam-1.5.3
      -> /nix/store/...-openssl-3.0.2
        -> /nix/store/...-zlib-1.2.13
```

You can explore this graph with Nix commands:

```
# Show the closure of a store path
nix-store -qR /nix/store/...-hello-2.12

# Show the reverse dependencies (what depends on this path)
nix-store -qr /nix/store/...-openssl-3.0.2

# Show the size of a closure
nix-store --query --requisites --size /nix/store/...-hello-2.12 | sort -rn
```

The Nix database also tracks **profiles**, which are named pointers to specific store paths. When you install something with `nix profile install`, it creates or updates a profile that points to the new package. Each profile can have multiple generations, allowing you to roll back to previous states.

```
# List profiles and their generations
nix profile list

# Roll back to a previous generation
nix profile rollback

# Show what changed between generations
nix profile diff-closures --profile /nix/var/nix/profiles/default 10
```

Garbage Collection and Space Management

Because Nix never modifies or deletes store paths on its own, the store can grow indefinitely. A system that has been running for months without garbage collection might have gigabytes of old packages taking up space. Understanding how garbage collection works is essential for managing disk usage.

Nix uses a **root-based** garbage collection model. A store path is considered "alive" if it is reachable from a **GC root**. GC roots include:

- **Profiles**: The active generations of all profiles (default user profile, system profile, etc.)
- **Symlink farms**: Directories like `/nix/var/nix/gcroots`
- **Active build processes**: Paths being used by currently running builds
- **User-declared roots**: Paths explicitly marked as roots with `nix-store --add-root`

Any store path not reachable from a GC root is considered garbage and can be deleted.

To run garbage collection:

```
# Collect garbage (delete unreachable paths)
nix-collect-garbage

# Delete old generations of profiles before collecting
nix-collect-garbage -d

# Delete everything older than 7 days
nix-collect-garbage --delete-older-than 7d

# Show what would be deleted without actually deleting it
nix-store --optimise --dry-run
```

The `--delete-older-than` flag is particularly useful for production systems. It keeps recent generations available for quick rollbacks while freeing up space from older ones.

Storage optimization: Over time, the store can accumulate duplicate content. Different packages might include identical files (like the same version of a shared library). The `nix-store --optimise` command detects these duplicates and replaces them with hard links, saving disk space:

```
# Find and replace duplicate files with hard links
sudo nix-store --optimise
```

On NixOS, you can enable automatic optimization:

```
{ config, pkgs, ... }: {
  nix.optimise.automatic = true;
  nix.optimise.dates = [ "03:00" ];
}
```

Disk space management in production: For systems with limited disk space, you need a strategy. Here is a recommended approach:

```
{ config, pkgs, ... }: {
  # Keep only 3 generations of the system profile
  boot.loader.systemd-boot.configurationLimit = 3;

  # Auto-collect garbage daily, keeping 7 days of history
  nix.gc.automatic = true;
  nix.gc.dates = "daily";
  nix.gc.options = "--delete-older-than 7d";

  # Optimize store weekly
  nix.optimise.automatic = true;
  nix.optimise.dates = [ "weekly" ];
}
```

Common pitfalls:

- **Running out of space during builds**: Nix needs temporary space to build packages before moving them into the store. If your disk is nearly full, builds can fail with "no space left on device." Always keep at least 10% free space.
- **Accidentally deleting needed packages**: Running `nix-collect-garbage` without the `-d` flag does not delete old generations, so it is relatively safe. But running it with `-d` and an aggressive time limit can remove generations you might want to roll back to.
- **The store growing unexpectedly**: If the store grows faster than expected, check for large builds that are not being garbage collected. Use `du -sh /nix/store/* | sort -rh | head -20` to find the largest packages.

Your First Nix Expressions

Now that you understand the store model, let us write your first Nix expression. We will build a simple "hello world" application step by step.

Step 1: Create a basic derivation

Create a file called `hello.nix`:

```
{ stdenv, fetchurl }:

stdenv.mkDerivation {
  pname = "hello";
  version = "2.12";

  src = fetchurl {
    url = "mirror://gnu/hello/hello-${version}.tar.gz";
    sha256 = "0ssi1wpaf7plaswqqjwigppsg5fyh99vdlb9kzl7c9nglx912184";
  };
}
```

This expression does the following:

- Takes two arguments: `stdenv` (the standard build environment) and `fetchurl` (a function to download files).
- Calls `stdenv.mkDerivation`, which creates a derivation using the standard C/C++ build process.
- Sets the package name (`pname`) and version.
- Downloads the source code from the GNU mirror, verifying its integrity with the SHA256 hash.

Step 2: Build it

```
nix-build hello.nix
```

This command evaluates the expression, computes the output path, checks if it already exists in the store (or a binary cache), and builds it if necessary. The result is a symlink called `result` pointing to the built package:

```
./result/bin/hello
# Hello, world!
```

Step 3: Understand what happened

Let us trace through the build process:

1. Nix evaluates the expression and determines the output path will be something like `/nix/store/0q8k3z1f2h5m6v7w8x9y0a1b2c3d4e5f-hello-2.12`.

2. It checks if this path already exists in the store or a binary cache. If not, it proceeds to build.
3. It creates an isolated build environment containing only the declared dependencies (in this case, just the standard C toolchain from `stdenv`).
4. It downloads the source tarball, verifying the SHA256 hash matches.
5. It runs the standard build phases: unpack, configure, build, install.
6. The result is placed in the output path, and a symlink `result` is created pointing to it.

Step 4: Install it permanently

```
nix profile install ./hello.nix
```

This adds the package to your default user profile. You can now run it from anywhere:

```
hello
# Hello, world!
```

Step 5: Explore the store path

```
# Find the actual store path
ls -la /nix/store/*-hello-2.12

# Show its dependencies
nix-store -qR /nix/store/...-hello-2.12

# Show the build log (if it was built locally)
nix log /nix/store/...-hello-2.12
```

What if the hash is wrong?

If you change the SHA256 hash to something incorrect, Nix will fail during the download phase:

```
# This will fail with a hash mismatch error
# because the hash does not match the downloaded file
```

This is by design. The hash ensures that the source code has not been tampered with, and that you are building exactly what you intended to build.

Key takeaway: Every Nix build follows this pattern: evaluate the expression, compute the output path from the hash of all inputs, check if it already exists, and build if necessary. The hash-based addressing is what makes everything reproducible and cacheable.

Chapter 2: The Nix Language — From Basics to Advanced Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Expressions, Values, and Types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Let-In Blocks and Local Bindings

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Functions and Argument Defaults

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Attribute Sets and Recursive Definitions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Conditionals, Lists, and Comprehensions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Importing Files and Building Abstractions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Chapter 3: Reproducible Development Environments

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Nix Shells and the devShell Concept

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Multi-Language Development Environments

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Flakes for Reproducible Dev Environments

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Environment Management with Home Manager

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Migrating from Docker Desktop to Nix DevShells

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Chapter 4: Package Management and Overlays

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

The Nix Package Manager in Practice

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Understanding the Flakes Input System

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Creating and Composing Overlays

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Custom Package Definitions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Managing Dependencies Across Projects

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Chapter 5: Building Packages from Source

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

The Derivation Build Process

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Building C and C++ Projects with CMake

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Rust Packages with Cargo

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Python Packages with Pip and Setuptools

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Go Modules and Node.js Packages

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Cross-Compilation for Multiple Platforms

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Chapter 6: NixOS — Declarative System Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

The NixOS Module System Architecture

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Configuring Your First NixOS Machine

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Service Management with systemd

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Networking, Firewall, and Security

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Users, Groups, and SSH Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Hardware Configuration and Drivers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Chapter 7: Home Manager — User-Level Declarative Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Home Manager and the Module System

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Managing Dotfiles Declaratively

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Shell and Editor Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Application Settings and Preferences

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Combining NixOS and Home Manager

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Chapter 8: CI/CD with Nix — Build Pipelines That Actually Work

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Setting Up Binary Caches

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

GitHub Actions with Nix

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

GitLab CI Integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Build Caching Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Testing in CI Pipelines

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Secrets Management in CI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Chapter 9: Containers and Kubernetes with Nix

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Creating Container Images with Nix

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Kubernetes Manifests as Nix Modules

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Helm Charts Generated from Nix

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

GitOps with Argo CD and Flux

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Service Mesh and Observability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Chapter 10: Infrastructure as Code — Terraform, OpenTofu, and Beyond

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Managing Terraform Providers with Nix

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

OpenTofu Integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Ansible Provisioning with Nix

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Cloud Provider Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Multi-Cloud Deployment Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Chapter 11: Secrets Management and Security

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Why Secrets Are Hard with Declarative Config

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Using SOPS for Encrypted Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Hashicorp Vault Integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Age Encryption for Simple Secrets

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Security Hardening Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Chapter 12: Platform Engineering at Scale

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

The Internal Developer Platform Pattern

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Monorepo Strategies with Nix

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Multi-Platform Development and Distribution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Build Optimization and Performance Tuning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Debugging and Troubleshooting Production Issues

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Testing Strategies for Nix Configurations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Maintenance, Upgrades, and Migration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Enterprise Operational Checklists

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

Conclusion: The Path Forward

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/practicalnixfordevopsplatformengineering.