

THE COMPLETE PRACTICAL GUIDE

# Docker and Kubernetes for Developers

From your first container
to a production platform

DOCKERFILES · COMPOSE · K8S · HELM · GITOPS · SECURITY

THENEXBOOKS

# Docker and Kubernetes for Developers

*The Complete Practical Guide from Containers to Production*

First Edition

THENEXBOOKS

**Docker and Kubernetes for Developers**
The Complete Practical Guide from Containers to Production

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, Dockerfiles, and manifests may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. Container tooling and Kubernetes evolve quickly; always consult the official documentation for the versions you run. The publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

Docker is a trademark of Docker, Inc. Kubernetes is a registered trademark of The Linux Foundation. Helm, Argo CD, Flux, Prometheus, Grafana, OpenTelemetry, and other names are trademarks of their respective owners. This book is independent and is not affiliated with or endorsed by any of these organizations.

# Preface

"It works on my machine" used to be the end of a conversation. Containers turned it into the beginning of a solution: package the application with everything it needs, and run the same artifact on a laptop, in CI, and in production. Kubernetes then took the next step — running thousands of those containers reliably, healing them when they fail, scaling them with demand, and rolling out new versions without downtime.

Today these two technologies are the default way modern software is built and shipped. Yet many developers learn them in fragments: a Dockerfile copied from a blog, a YAML file from a colleague, a Helm chart nobody fully understands. The result is slow builds, bloated and insecure images, fragile deployments, and late-night incidents.

This book is a complete, practical path from your first container to a production platform. We explain how containers actually work, how to build small and secure images, how to develop productively with Compose, and how Kubernetes schedules, connects, configures, scales, and protects your workloads. Then we go further — packaging with Helm and Kustomize, GitOps, observability, security, cloud cost, and troubleshooting — and finish with two end-to-end case studies.

## Who This Book Is For

- **Application developers** who want to containerize and deploy their own services with confidence.
- **Backend and full-stack engineers** moving from a single server or PaaS to Kubernetes.
- **DevOps and platform engineers** who need a clear, structured reference for teams they support.
- **Students and career changers** preparing for cloud-native roles.

## What You Will Learn

- How containers work under the hood — namespaces, cgroups, layers, and the OCI standards.
- How to write fast, small, secure Dockerfiles with multi-stage builds and BuildKit.
- How to run multi-service apps locally with Docker Compose and dev containers.
- How to build, scan, sign, and publish images in CI/CD pipelines.
- Kubernetes from the ground up: Pods, Deployments, Services, Gateway API, configuration, and storage.
- Autoscaling, health checks, zero-downtime rollouts, Helm, Kustomize, GitOps, and observability.
- Kubernetes security, managed clusters in the cloud, cost optimization, and systematic troubleshooting.

## How This Book Is Organized

**Part I — Container Foundations** (Chapters 1–6) explains containers and Docker in depth. **Part II — Docker in Development and Delivery** (Chapters 7–11) covers Compose, local workflows, registries, security, and CI/CD. **Part III — Kubernetes Fundamentals** (Chapters 12–18) builds your understanding of the cluster and its core objects. **Part IV — Operating Kubernetes** (Chapters 19–23) covers scaling, reliability, packaging, GitOps, and observability. **Part V — Security and Production** (Chapters 24–26) addresses security, cloud, cost, and troubleshooting. **Part VI — Case Studies** (Chapters 27–28) brings everything together. The appendices provide cheat sheets, a manifest reference, a production checklist, and a glossary.

## The Running Example: Orbit Store

Throughout the book we containerize and deploy **Orbit Store**, a fictional e-commerce marketplace: a web frontend, a Node.js API, a Python catalog service, a background worker, PostgreSQL, and Redis. It is small enough to understand completely and realistic enough to exercise every technique in this book.

## 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 a behavior that can cause outages, data loss, or security problems.

Commands, Dockerfiles, and YAML appear in `monospace`. A trailing backslash (`\`) continues a shell command on the next line.

# Contents

Preface

PART I — CONTAINER FOUNDATIONS

1 Why Containers: From "Works on My Machine" to Cloud Native

2 How Containers Work: Namespaces, cgroups, Images, and the OCI

3 Getting Started with Docker: Install, Run, and Inspect

4 Writing Dockerfiles: Layers, Caching, and Best Practices

5 Multi-Stage Builds and Small, Secure Images

6 Docker Networking and Storage

PART II — DOCKER IN DEVELOPMENT AND DELIVERY

7 Docker Compose for Multi-Service Applications

8 Local Development Workflows: Hot Reload, Dev Containers, and Debugging

9 Registries, Tagging, and Versioning Images

10 Container Security Fundamentals

11 CI/CD for Containers: Building, Testing, and Publishing Images

PART III — KUBERNETES FUNDAMENTALS

12 Kubernetes Architecture: Control Plane, Nodes, and the API

13 Getting Started: Local Clusters and kubectl

14 Pods, ReplicaSets, and Deployments

15 Services, Ingress, and the Gateway API

16 Configuration: ConfigMaps, Secrets, and Environment

17 Storage and Stateful Workloads: PersistentVolumes and StatefulSets

18 Jobs, CronJobs, DaemonSets, and Workload Patterns

PART IV — OPERATING KUBERNETES

19 Resource Management and Autoscaling

20 Health Checks, Rollouts, and Reliability

21 Packaging with Helm and Kustomize

22 GitOps with Argo CD and Flux

23 Observability: Logs, Metrics, and Traces

**PART V — SECURITY AND PRODUCTION**

24 Kubernetes Security: RBAC, Pod Security, Network Policies, and Supply Chain

25 Managed Kubernetes in the Cloud and Cost Optimization

26 Troubleshooting Containers and Kubernetes

**PART VI — CASE STUDIES**

27 Case Study: Containerizing and Deploying Orbit Store

28 Case Study: A Production Platform — CI/CD, GitOps, Observability, and Scale

**APPENDICES**

A Docker and kubectl Cheat Sheet

B Manifest Reference

C Production Checklist

D Glossary

PART I

# Container Foundations

What containers are, how they work under the hood, and how to build small, fast, secure images with Docker.

1 · Why Containers
2 · How Containers Work
3 · Getting Started with Docker
4 · Writing Dockerfiles
5 · Multi-Stage Builds and Small Images
6 · Docker Networking and Storage

CHAPTER 1 · PART I — CONTAINER FOUNDATIONS

# Why Containers: From "Works on My Machine" to Cloud Native

*Before we type a single `docker` command, we need a clear picture of the problems containers solve, where they came from, and what Kubernetes adds on top. This chapter builds that picture and introduces Orbit Store, the application we will containerize, run, and operate throughout the book.*

## The Problem: Software Is More Than Code

Every developer has heard, or said, the phrase "but it works on my machine." It is usually spoken with a mix of frustration and defensiveness, and it is usually true. The code really does work on the laptop where it was written. The problem is that the code is never the whole story. A running application is the combination of your source code, a language runtime, a set of third-party libraries, system libraries such as OpenSSL or libc, configuration files, environment variables, and assumptions about the operating system it runs on. Change any one of those and behavior can change with it.

Traditional deployment treated the code as the artifact and everything else as "the environment", something that operations teams prepared by hand or with configuration-management scripts. That split worked well enough when an organization ran a handful of applications on long-lived servers. It works poorly when a team ships dozens of services, several times a day, to machines that are created and destroyed automatically. Let us look at the specific failure modes that containers were designed to eliminate.

### Environment drift

Environment drift is the slow divergence between machines that are supposed to be identical. The staging server received a security patch last Tuesday; production did not. A developer installed a newer Python on their laptop to try a feature and forgot to mention it. Someone logged into a production host during an incident and changed a kernel parameter by hand. None of these changes are recorded anywhere a build system can see, so the next deploy runs in a slightly different world from the last one.

Drift is dangerous precisely because it is invisible. Tests pass in CI, the release goes out, and a subtle difference in a system library causes a crash only in production. The team then spends hours comparing package lists between hosts. Containers address drift by packaging the application together with its user-space dependencies into an immutable image. The same image, identified by the same content hash, runs in CI, staging, and production. If you want a different environment, you build a different image, and that change is visible in version control.

### Dependency conflicts

Two applications on the same host often need different versions of the same thing. The legacy billing service requires Node.js 16; the new API wants Node.js 22. One Python service pins an old version of a native library, another needs the latest. On a shared server, you end up juggling version managers, virtual environments, and carefully ordered `PATH` entries, and every new service risks breaking an existing one.

Containers solve this by giving each application its own filesystem view. The billing service's container sees Node.js 16 at `/usr/local/bin/node`; the API's container sees Node.js 22 at the same path. Neither knows the other exists. This isolation is what lets a single host run dozens of services built with different stacks without them stepping on each other.

### Slow onboarding

A new engineer joins the team. Their first week is spent following a wiki page titled "Local Setup" that was last accurate two years ago. They install a specific database version, discover it conflicts with something already on their machine, find the right Redis configuration in a Slack thread, and eventually get the application to start, minus one feature that nobody can explain. Multiply that week by every hire and every laptop replacement and the cost is significant.

With containers, the setup instructions become executable. A Compose file (covered in Chapter 7) declares every service, database, and cache the application needs, and one command starts them all with the correct versions. The wiki page shrinks to "install a container runtime, clone the repository, run `docker compose up`." Onboarding goes from days to minutes, and, just as importantly, every developer runs the same versions of the backing services that production uses.

### Inconsistent deployments

Without a standard packaging format, each application tends to invent its own deployment process. The Java service ships a JAR and a systemd unit. The Python service is deployed with a shell script that runs `git pull` and `pip install` on the server. The Node.js service uses a process manager configured differently on every host. Operations teams must understand each variant, and every variant fails in its own way.

A container image is a uniform unit of deployment. Whatever language or framework is inside, the outside looks the same: an image with a name and a tag, a declared entry point, exposed ports, and environment variables for configuration. The tools that build, scan, store, and run images do not care what is inside. That uniformity is what makes the rest of the ecosystem, from CI pipelines to Kubernetes, possible.

> **NOTE**
>
> Containers do not remove the need to think about the environment. They move that thinking into a file, the Dockerfile, that lives next to your code, is reviewed like code, and produces a reproducible artifact. That shift from tribal knowledge to versioned definition is the real win.

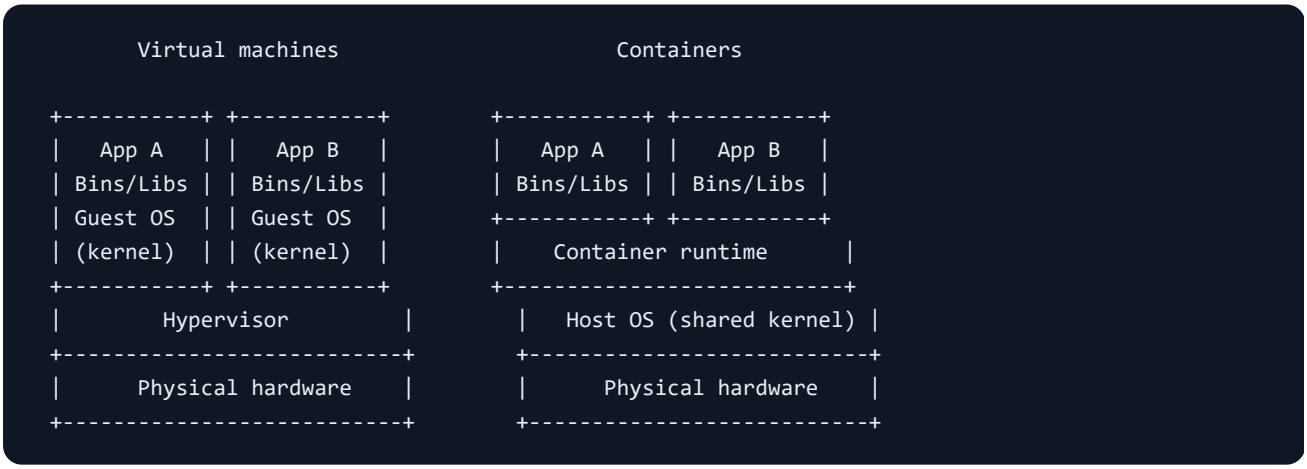
## What a Container Actually Is

Chapter 2 explores the mechanics in depth, but a working definition helps now. A **container** is an ordinary Linux process (or group of processes) that the kernel runs with restricted visibility and restricted resources. Using kernel features called *namespaces*, the process gets its own view of the process table, network interfaces, hostname, and filesystem mounts. Using *control groups* (cgroups), the kernel limits how much CPU, memory, and I/O it may consume. The filesystem the process sees comes from a **container image**: a stack of read-only layers containing the application and everything it needs, plus a thin writable layer on top.

The key word is *process*. A container is not a small computer. It does not boot an operating system; it does not have its own kernel. When you start a container, the host kernel starts a process, applies the isolation settings, and that process begins running your application almost immediately. This is why containers start in well under a second in typical cases and why you can run many more of them than virtual machines on the same hardware.

## Containers vs Virtual Machines

Virtual machines (VMs) also isolate workloads, and they remain an important technology, including as the foundation that most cloud container platforms run on. The difference is where the isolation boundary sits. A VM uses a hypervisor to emulate hardware; each VM runs a complete guest operating system with its own kernel. A container shares the host kernel and isolates only at the process level.




The table below summarizes the practical differences. The figures describe typical behavior, not guarantees; a tuned micro-VM can start quickly, and a container with a heavy application can take a long time to become ready.

| Aspect | Virtual machine | Container |
|---|---|---|
| Isolation unit | Full guest OS | Process group |
| Kernel | Own kernel per VM | Shared host kernel |
| Typical startup | Tens of seconds | Sub-second to seconds |
| Image size | Gigabytes | Megabytes to hundreds of MB |
| Density per host | Lower | Higher |
| Security boundary | Strong (hardware virt.) | Weaker (shared kernel) |
| Guest OS choice | Any OS | Same kernel family |
| Best for | Multi-tenant isolation | Packaging apps |

Two rows deserve emphasis. First, the security boundary. Because all containers on a host share one kernel, a kernel vulnerability can potentially let a process escape its container. VMs have a much smaller attack surface between guests. This is why public clouds run customer containers inside VMs, and why projects such as gVisor and Kata Containers exist to add a stronger boundary around containers when you run untrusted code. Second, the kernel family. A Linux container needs a Linux kernel. When you run Docker Desktop on macOS or Windows, a lightweight Linux VM is quietly running underneath, a detail we unpack in Chapter 2.

In practice, containers and VMs are complementary rather than competing. A typical production setup runs containers on VMs: the cloud provider's hypervisor isolates tenants from each other, and containers pack many services efficiently onto each VM.

## A Brief History of Containers

Containers feel like a recent invention, but the ideas behind them evolved over four decades. Knowing the lineage helps you understand why the technology looks the way it does today.

### chroot and early isolation (1979–2000s)

The `chroot` system call appeared in Version 7 Unix in 1979. It changes the apparent root directory of a process, so that the process sees a subdirectory as `/`. That was the first step toward filesystem isolation, although chroot alone isolates nothing else and is not a security boundary. FreeBSD Jails (2000) extended the idea to include separate process and network visibility, and Solaris Zones (2004) offered a similar, more complete model. These systems proved that OS-level virtualization was practical, but they were tied to their respective operating systems.

### Namespaces, cgroups, and LXC (2002–2008)

Linux acquired the necessary building blocks piece by piece. The first namespace, for mount points, arrived in 2002; others for process IDs, networking, hostnames, and inter-process communication followed over the next decade, with user namespaces completed in 2013. Control groups, originally developed by engineers at Google under the name "process containers", were merged into the kernel in 2008. The same year, the LXC project (Linux Containers) combined namespaces and cgroups into a usable toolset that could run a full Linux userland in isolation. LXC worked, but it was aimed at system administrators and felt like managing lightweight VMs rather than shipping applications.

### Docker (2013)

In March 2013, a platform-as-a-service company called dotCloud open-sourced an internal tool named Docker. Docker did not invent container isolation; early versions actually used LXC under the hood before switching to its own library, libcontainer. What Docker invented was the *developer experience*: a simple text format (the Dockerfile) for describing an image, a layered image format that made builds fast and images shareable, a public registry (Docker Hub) for distributing images, and a CLI that made `docker run` feel as easy as running a local binary. The company later renamed itself Docker, Inc., and the tool spread through the industry at remarkable speed.

### The Open Container Initiative (2015)

As Docker's popularity grew, so did concern about a single company controlling the de facto container format. In June 2015, Docker and other industry players founded the **Open Container Initiative** (OCI) under the Linux Foundation. Docker donated its image format and runtime code; the runtime became `runc`, the reference implementation of the OCI Runtime Specification. The OCI now maintains three specifications: the Image Specification, the Runtime Specification, and the Distribution Specification (how registries serve images). Because of the OCI, an image built with Docker can be run by containerd, CRI-O, Podman, or any other compliant tool, and a registry does not care which tool pushed an image to it.

### Kubernetes and the CNCF (2014–2015)

Running one container is easy. Running thousands across hundreds of machines, keeping them healthy, and connecting them to each other is not. Google had been solving that problem internally for years with a system called Borg. In June 2014, Google announced Kubernetes, an open-source container orchestrator inspired by lessons from Borg. Kubernetes 1.0 shipped in July 2015, and at the same time Google and the Linux Foundation formed the **Cloud Native Computing Foundation** (CNCF), with Kubernetes as its seed project. The CNCF has since become home to much of the ecosystem you will meet in this book, including containerd, Prometheus, Helm, Argo, Flux, and OpenTelemetry.

One more milestone matters for readers today. Early Kubernetes talked to Docker Engine through a built-in adapter called dockershim. That adapter was removed in Kubernetes 1.24, and clusters now use runtimes that implement the Container Runtime Interface directly, most commonly containerd or CRI-O. This caused some alarm at the time, but it changed nothing about the images you build: images produced with Docker are OCI images, and they run on any modern cluster.

| Year | Milestone |
|---|---|
| 1979 | chroot in Version 7 Unix |
| 2000 | FreeBSD Jails |
| 2004 | Solaris Zones |
| 2008 | cgroups merged; LXC released |
| 2013 | Docker open-sourced |
| 2014 | Kubernetes announced |
| 2015 | OCI founded; Kubernetes 1.0; CNCF |
| 2022 | dockershim removed (K8s 1.24) |

## What Kubernetes Adds

Docker gives you a way to package and run a container on one machine. Production systems need more. They need many copies of each service for capacity and redundancy, spread across machines so that a single failure does not take everything down. They need a way for services to find each other when containers move around. They need rolling upgrades without downtime, automatic restarts when something crashes, and a way to add capacity when traffic grows. Kubernetes provides all of this through a handful of core ideas.

### Orchestration and scheduling

A Kubernetes cluster is a set of machines, called *nodes*, managed by a *control plane*. You do not tell Kubernetes which machine to run a container on. You describe the workload, including how much CPU and memory it needs, and the scheduler picks a node with enough free capacity, respecting any constraints you set such as "spread these replicas across availability zones." Chapter 12 covers the architecture in detail.

### Self-healing

If a container crashes, Kubernetes restarts it. If a node dies, the workloads that were on it are rescheduled elsewhere. If a health check fails, traffic stops flowing to the unhealthy instance until it recovers. This happens continuously and automatically, without a human being paged for every transient failure. Chapter 20 shows how to configure the probes that drive this behavior.

### Scaling

You can change the number of replicas of a service with a single command or a single line of YAML. With the Horizontal Pod Autoscaler, Kubernetes adjusts the replica count for you based on CPU usage, memory, or custom metrics. Cluster autoscalers can add and remove nodes to match. Chapter 19 covers resource management and autoscaling.

### Declarative configuration

This is the most important idea, and the one that takes longest to internalize. With Kubernetes, you do not issue a sequence of commands like "start three containers, then attach them to a load balancer." You submit a description of the *desired state*, such as "there should be three replicas of the api image, version 1.4.2, reachable on port 8080," and Kubernetes controllers work continuously to make the actual state match. If something drifts, a controller notices and corrects it. Here is a small taste of what that looks like; we will build up to manifests like this gradually from Chapter 14 onward.

api-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api
  labels:
    app: api
spec:
  replicas: 3
  selector:
    matchLabels:
      app: api
  template:
    metadata:
      labels:
        app: api
    spec:
      containers:
        - name: api
          image: registry.example.com/orbit/api:1.4.2
          ports:
            - containerPort: 8080
```

Because the desired state lives in files, it can live in Git. Changes go through code review, the history of every change is preserved, and rolling back means reverting a commit. Taken to its conclusion, this becomes GitOps, where a controller inside the cluster continuously syncs the cluster to what is in a repository. Chapter 22 is devoted to that practice.

### A common API and ecosystem

Finally, Kubernetes provides a consistent, extensible API. Load balancers, storage volumes, secrets, scheduled jobs, and certificates are all represented as API objects with the same structure. Tools such as Helm, Argo CD, Prometheus, and service meshes plug into that API. Once you know how to read

and write Kubernetes objects, you can work with a large ecosystem of tools that all speak the same language, and the same manifests can run on a laptop cluster, an on-premises data center, or any major cloud's managed Kubernetes service.

## When NOT to Use Kubernetes

A book about Kubernetes should be honest about its costs. Kubernetes is powerful, but it is also a distributed system with many moving parts, and running it well requires real expertise. Adopting it for the wrong reasons produces complexity without benefit. Consider simpler options in the following situations.

- **You run one or two services with modest traffic.** A single VM running Docker Compose, or a platform-as-a-service, will be easier to operate and cheaper to run. You can still package everything as containers, which keeps the door open to Kubernetes later.
- **Your team has no one who can own the platform.** Even with a managed control plane, someone must understand networking, upgrades, RBAC, resource tuning, and troubleshooting. Without that ownership, clusters decay into fragile, outdated systems.
- **A serverless container platform fits your needs.** Managed services that run a container per request or scale containers to zero can deliver most of the benefits for stateless HTTP services with far less operational work.
- **Your workload is mostly a single stateful database.** Running databases on Kubernetes is possible and increasingly common, as Chapter 17 discusses, but a managed database service is often the simpler and safer choice.

The good news is that containers themselves are almost always worth adopting. The Docker skills in Parts I and II of this book pay off whether or not you ever deploy to a cluster. Kubernetes becomes compelling when you have many services, multiple teams, a need for high availability, or variable load that benefits from automatic scaling.

> **TIP**
>
> A useful test: list the operational problems you have today (manual deploys, downtime during releases, services that do not restart after crashes, uneven load). If Kubernetes solves several of them and you have people to operate it, it is likely a good fit. If the list is short, start with Compose or a managed container service.

## Meet Orbit Store

Abstract explanations only go so far. Throughout this book, we work with a single, realistic application, **Orbit Store**, a fictional online marketplace where independent sellers list products and shoppers browse, search, and place orders. Orbit Store is small enough to understand completely but

has the shape of a real production system: multiple services in different languages, a relational database, a cache, background processing, and both synchronous and asynchronous communication.

## The services

| Service | Technology | Responsibility |
|---|---|---|
| `web` | Next.js on Node.js | Storefront UI |
| `api` | Node.js, TypeScript | Orders, carts, users |
| `catalog` | Python, FastAPI | Products and search |
| `worker` | Node.js | Background jobs |
| PostgreSQL | Relational database | Durable data |
| Redis | In-memory store | Cache, sessions, queue |

The `web` service renders the storefront. It serves HTML and static assets to browsers and calls the backend services for data. Because Next.js can render pages on the server, `web` is itself a Node.js process, not just a bundle of static files.

The `api` service is the heart of the business logic. Written in TypeScript on Node.js, it exposes a REST API for user accounts, shopping carts, checkout, and order history. It stores data in PostgreSQL and uses Redis for session data and short-lived caching. When a customer places an order, `api` writes the order to the database and pushes a job onto a Redis-backed queue rather than doing slow work, such as sending email, during the HTTP request.

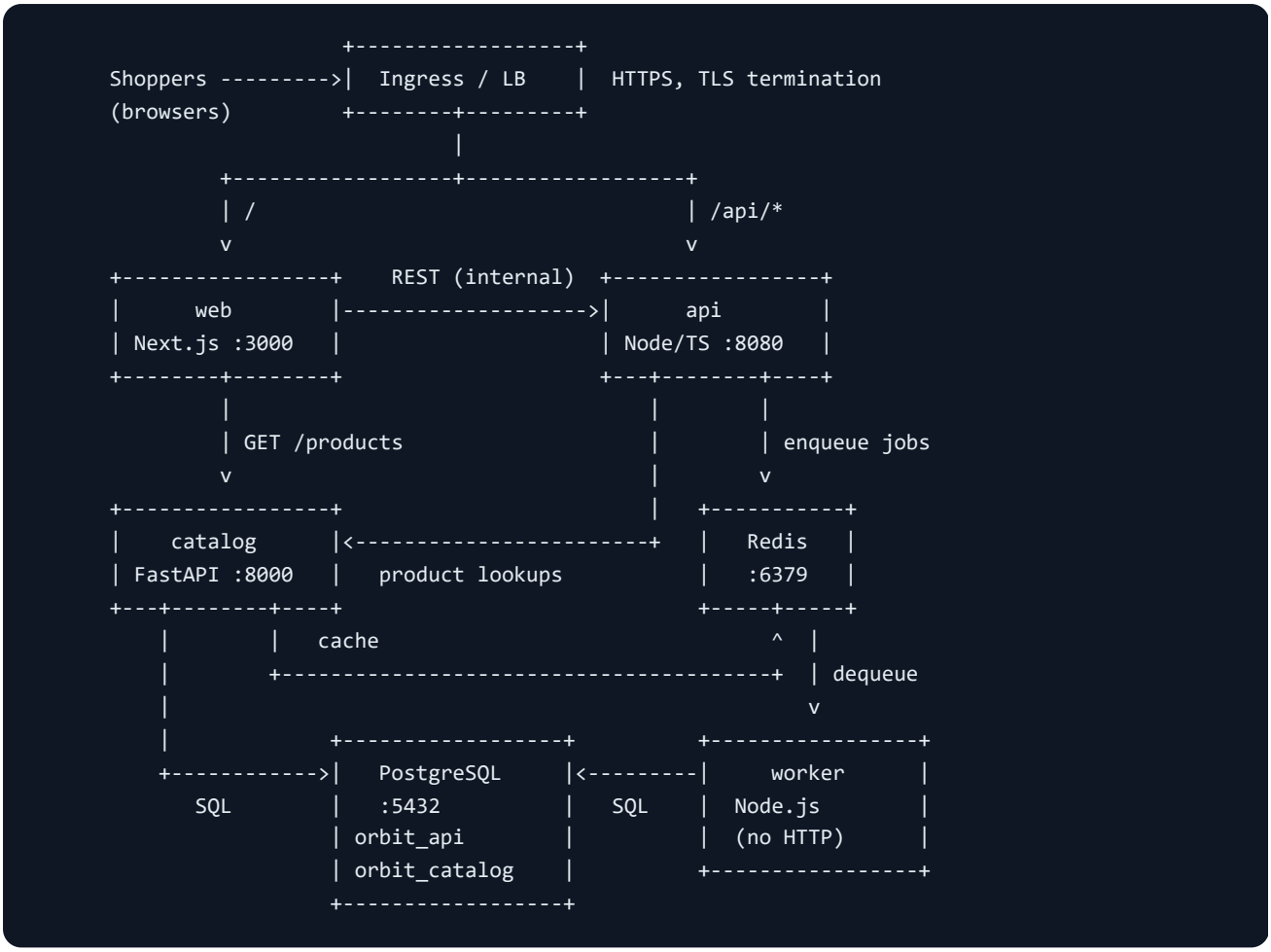
The `catalog` service owns product data: titles, descriptions, prices, images, and inventory counts. It is written in Python with FastAPI, partly because the team that owns it prefers Python and partly to show that containers make polyglot systems easy. Product pages are read far more often than they are written, so `catalog` caches heavily in Redis.

The `worker` service consumes jobs from the queue: sending order confirmation emails, generating invoice PDFs, and updating search indexes. It has no HTTP interface for customers; it scales based on how many jobs are waiting. That makes it a good example for autoscaling discussions later in the book.

PostgreSQL holds durable data for both `api` and `catalog`, in separate databases. Redis serves as cache, session store, and job queue. In local development we run both in containers; in production, Chapter 25 discusses whether to run them in the cluster or use managed cloud services.

## Architecture

The diagram below shows how requests flow through the system. A browser talks only to the entry point. Internal services talk to each other over the private network and never expose their ports to the internet directly.




A few design choices in this architecture are worth calling out, because they will shape how we containerize and deploy it.

- **Each service is stateless.** The `web`, `api`, `catalog`, and `worker` processes keep no durable state on local disk. Anything that must survive a restart goes to PostgreSQL or Redis. This is what allows us to run many replicas and replace them freely.
- **Configuration comes from the environment.** Database URLs, Redis addresses, and feature flags are read from environment variables, following the widely used twelve-factor app guidelines. The same image runs in every environment with different configuration injected.
- **Services communicate by name.** The `api` reaches the catalog at `http://catalog:8000`, not a hard-coded IP address. Both Docker Compose and Kubernetes provide DNS-based service discovery that makes those names resolve.
- **Each service exposes health endpoints.** The HTTP services expose `/healthz` (the process is alive) and `/readyz` (the process can serve traffic, including reaching its dependencies). These feed Docker health checks in Chapter 4 and Kubernetes probes in Chapter 20.
- **Slow work is asynchronous.** Moving email and PDF generation to the `worker` keeps checkout fast and lets us scale the two kinds of work independently.

### Repository layout

Orbit Store lives in a single repository with one directory per service. This "monorepo" layout keeps examples easy to follow; everything in the book works equally well with one repository per service.

```
orbit-store/
|-- services/
|   |-- web/          # Next.js storefront
|   |-- api/          # Node.js + TypeScript REST API
|   |-- catalog/      # Python FastAPI product service
|   `-- worker/       # Node.js background job processor
|-- deploy/
|   |-- compose/      # Docker Compose files (Chapter 7)
|   |-- k8s/          # Raw Kubernetes manifests (Part III)
|   `-- helm/         # Helm chart (Chapter 21)
|-- .github/
|   `-- workflows/    # CI pipelines (Chapter 11)
`-- README.md
```

We will not reproduce the full application source code in the book; it is not the point. We show the parts that matter for containerization, such as the `package.json` scripts, the Python requirements, the health endpoints, and the configuration loading, and we keep every Dockerfile, Compose file, and manifest complete and runnable.

### The pain we are solving

To make the motivation concrete, imagine Orbit Store before containers. A new developer must install Node.js 22 for `api`, `web`, and `worker`, Python 3.12 for `catalog`, PostgreSQL 16, and Redis 7, each at matching versions. The `catalog` service depends on a Python image-processing library with native extensions that fail to compile on some laptops. Production runs on VMs configured by a shell script that nobody has run from scratch in a year. Deploys happen by SSH, one service at a time, and a bad release requires someone to remember which commit was running before.

By the end of Part II, the same developer will run the entire stack with one command, and CI will build and publish versioned images for every service on each merge. By the end of Part IV, Orbit Store will run on Kubernetes with rolling deploys, autoscaling, health checks, GitOps, and dashboards that show what is happening inside it. That transformation, step by step, is the story of this book.

## How to Read This Book

The book is organized into six parts that build on each other. You can read it front to back, and that is the recommended path if you are new to containers. Experienced readers can jump to the part they need; every chapter references earlier material by chapter number so you can backtrack when necessary.

### Part I — Container Foundations (Chapters 1–6)

We start with fundamentals. Chapter 2 opens the hood on namespaces, cgroups, image layers, and the OCI standards so that containers stop feeling like magic. Chapter 3 gets Docker installed and running, and Chapter 4 teaches you to write Dockerfiles that build quickly and behave well. Chapter 5 shows multi-stage builds that shrink images and reduce attack surface, and Chapter 6 covers networking and storage.

### Part II — Docker in Development and Delivery (Chapters 7–11)

Next we use containers in everyday work. Chapter 7 runs all of Orbit Store with Docker Compose. Chapter 8 covers hot reload, dev containers, and debugging inside containers. Chapter 9 explains registries, tags, and versioning strategies. Chapter 10 introduces container security, including image scanning and running as non-root. Chapter 11 builds a CI/CD pipeline that tests and publishes images.

### Part III — Kubernetes Fundamentals (Chapters 12–18)

Here we move to the cluster. Chapter 12 explains the architecture, and Chapter 13 sets up a local cluster and teaches `kubectl`. Chapters 14 through 18 cover the core objects: Pods and Deployments, Services and Ingress and the Gateway API, ConfigMaps and Secrets, persistent storage and StatefulSets, and batch and node-level workloads.

### Part IV — Operating Kubernetes (Chapters 19–23)

Running workloads is one thing; running them well is another. This part covers resource requests and autoscaling, health checks and rollout strategies, packaging with Helm and Kustomize, GitOps with Argo CD and Flux, and observability with logs, metrics, and traces.

### Part V — Security and Production (Chapters 24–26)

Chapter 24 covers RBAC, Pod Security Standards, network policies, and supply-chain security. Chapter 25 compares managed Kubernetes offerings and discusses cost optimization principles. Chapter 26 is a systematic troubleshooting guide for when things go wrong, which they will.

### Part VI — Case Studies (Chapters 27–28)

The final part pulls everything together. Chapter 27 walks through containerizing and deploying Orbit Store end to end. Chapter 28 extends it into a production platform with CI/CD, GitOps, observability, and scaling, and discusses the decisions a real team would face along the way.

### What you need

You need a computer running Linux, macOS, or Windows with enough memory to run a few containers and, later, a local Kubernetes cluster; 8 GB of RAM is a workable minimum and 16 GB is comfortable. You should be comfortable in a terminal, familiar with Git, and able to read code in at