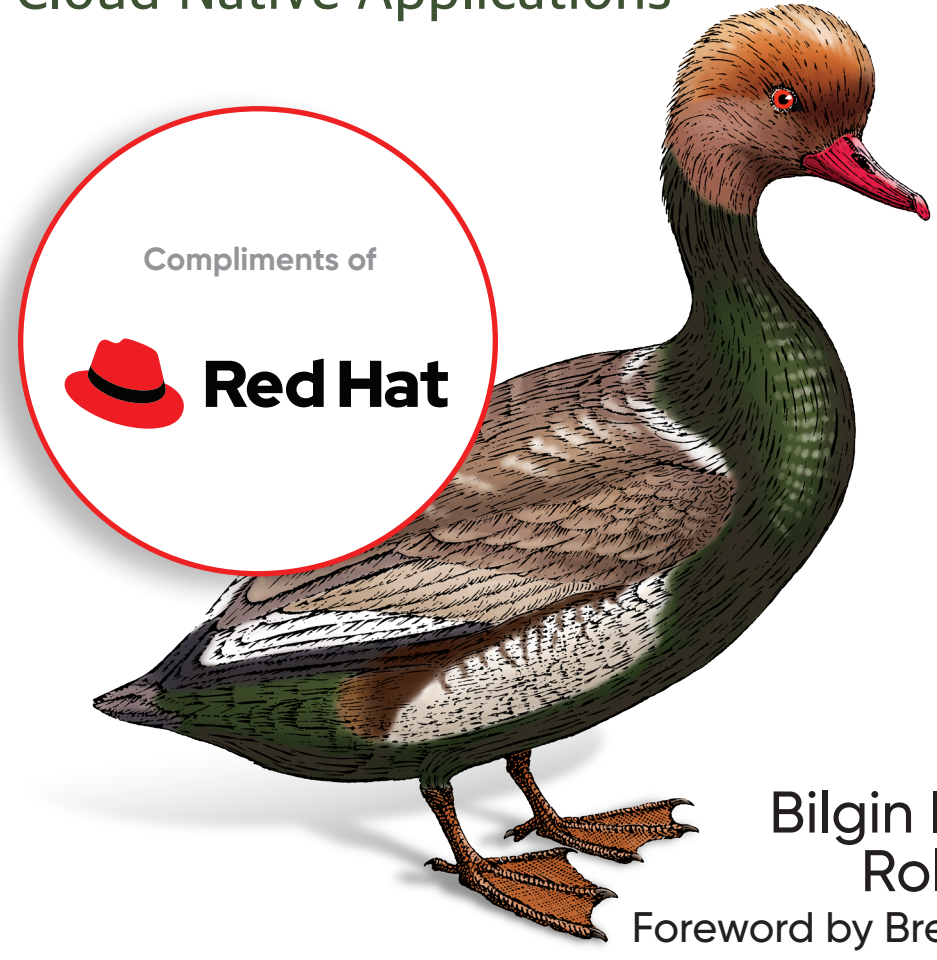
O'REILLY®

Second Edition

# Kubernetes Patterns

## Reusable Elements for Designing Cloud Native Applications




Bilgin Ibryam &
Roland Huß

Foreword by Brendan Burns

O'REILLY®

# Kubernetes Patterns

The way developers design, build, and run software has changed significantly with the evolution of microservices and containers. These modern architectures offer new distributed primitives that require a different set of practices than many developers, tech leads, and architects are accustomed to. With this focused guide, Bilgin Ibryam and Roland Huß provide common reusable patterns and principles for designing and implementing cloud native applications on Kubernetes.

Each pattern includes a description of the problem and a Kubernetes-specific solution. All patterns are backed by and demonstrated with concrete code examples. This updated edition is ideal for developers and architects who are familiar with basic Kubernetes concepts but want to learn how to solve common cloud native challenges with proven design patterns.

You'll explore:

- Foundational patterns covering core principles and practices for building and running container-based cloud native applications
- Behavioral patterns for managing various types of container and platform interactions
- Structural patterns for organizing containers to address specific use cases
- Configuration patterns that provide insight into how application configurations can be handled in Kubernetes
- Security patterns for hardening applications running on Kubernetes and making them more secure
- Advanced patterns covering more complex topics such as operators, autoscaling, and in-cluster image builds

"Bilgin and Roland have written a wonderful, incredibly informative, and intensely useful book."

—**Grady Booch**
Chief Scientist for Software Engineering, IBM; Coauthor, *Unified Modeling Language*

"An updated set of patterns to enable developers to take full advantage of the capabilities and features found in Kubernetes."

—**Andrew Block**
Distinguished Architect, Red Hat

**Bilgin Ibryam** is a principal product manager at Diagrid, where he leads the company's product strategy.

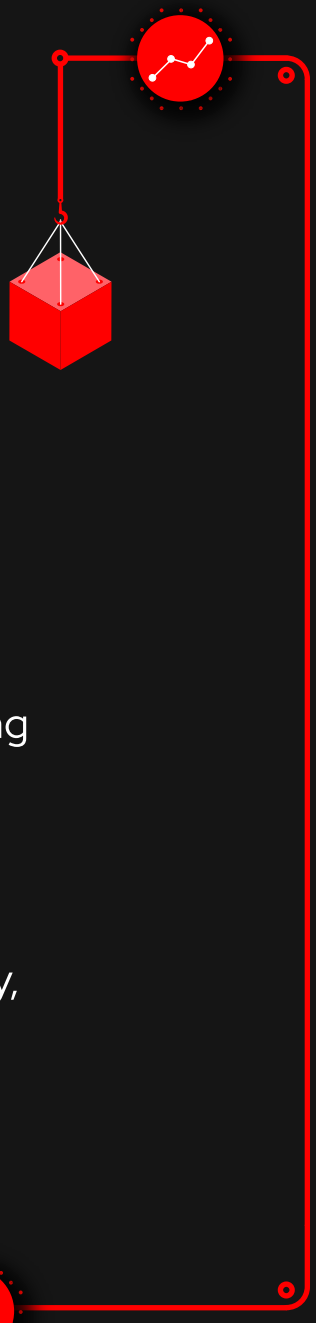
**Dr. Roland Huß** is a senior principal software engineer at Red Hat and the architect of OpenShift Serverless.

KUBERNETES

Twitter: @oreillymedia
linkedin.com/company/oreilly-media
youtube.com/oreillymedia

ISBN: 978-1-098-13988-9

Red Hat



# Build **Smarter.**
# Ship **Faster.**

To make the most of the cloud, IT needs to approach applications in new ways. Cloud-native development means packaging with containers, adopting modern architectures, and using agile techniques.

Red Hat can help you arrange your people, processes, and technologies to build, deploy, and run cloud-ready applications anywhere they are needed. Discover how with **cloud-native development solutions.**

SECOND EDITION

# Kubernetes Patterns

## *Reusable Elements for Designing Cloud Native Applications*

*Bilgin Ibryam and Roland Huß*

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

**Kubernetes Patterns**
by Bilgin Ibryam and Roland Huß



Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

**Acquisitions Editor:** John Devins
**Development Editor:** Rita Fernando
**Production Editor:** Beth Kelly
**Copyeditor:** Piper Editorial Consulting, LLC
**Proofreader:** Sharon Wilkey

**Indexer:** Judy McConville
**Interior Designer:** David Futato
**Cover Designer:** Karen Montgomery
**Illustrator:** Kate Dullea

April 2019: First Edition
March 2023: Second Edition

**Revision History for the Second Edition**
2023-03-25: First Release
2023-05-26: Second Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098131685* for release details.



978-1-098-13988-9

[LSI]

# Table of Contents

## Part VI. Advanced Patterns

# Foreword

When Craig, Joe, and I started Kubernetes nearly eight years ago, I think we all recognized its power to transform the way the world developed and delivered software. I don't think we knew, or even hoped to believe, how quickly this transformation would come. Kubernetes is now the foundation for the development of portable, reliable systems spanning the major public clouds, private clouds, and bare-metal environments. However, even as Kubernetes has become ubiquitous to the point where you can spin up a cluster in the cloud in less than five minutes, it is still far less obvious to determine where to go once you have created that cluster. It is fantastic that we have seen such significant strides forward in the operationalization of Kubernetes itself, but it is only a part of the solution. It is the foundation on which applications will be built, and it provides a large library of APIs and tools for building these applications, but it does little to provide the application architect or developer with any hints or guidance for how these various pieces can be combined into a complete, reliable system that satisfies their business needs and goals.

Although the necessary perspective and experience for what to do with your Kubernetes cluster can be achieved through past experience with similar systems, or via trial and error, this is expensive both in terms of time and the quality of systems delivered to our end users. When you are starting to deliver mission-critical services on top of a system like Kubernetes, learning your way via trial and error simply takes too much time and results in very real problems of downtime and disruption.

This then is why Bilgin and Roland's book is so valuable. *Kubernetes Patterns* enables you to learn from the previous experience that we have encoded into the APIs and tools that make up Kubernetes. Kubernetes is the by-product of the community's experience building and delivering many different, reliable distributed systems in a variety of different environments. Each object and capability added to Kubernetes represents a foundational tool that has been designed and purpose-built to solve a specific need for the software designer. This book explains how the concepts in Kubernetes solve real-world problems and how to adapt and use these concepts to build the system that you are working on today.

In developing Kubernetes, we always said that our North Star was making the development of distributed systems a CS 101 exercise. If we have managed to achieve that goal successfully, it is books like this one that are the textbooks for such a class. Bilgin and Roland have captured the essential tools of the Kubernetes developer and distilled them into segments that are easy to approach and consume. As you finish this book, you will become aware not just of the components available to you in Kubernetes but also the "why" and "how" of building systems with those components.

*— Brendan Burns*
*Cofounder, Kubernetes*

# Preface

With the mainstream adoption of microservices and containers in recent years, the way we design, develop, and run software has changed radically. Today's applications are optimized for availability, scalability, and speed-to-market. Driven by these new requirements, today's modern applications require a different set of patterns and practices. This book aims to help developers discover and learn about the most common patterns for creating cloud native applications with Kubernetes. First, let's take a brief look at the two primary ingredients of this book: Kubernetes and design patterns.

## Kubernetes

*Kubernetes* is a container orchestration platform. The origin of Kubernetes lies somewhere in the Google data centers where Google's internal container orchestration platform, Borg, was born. Google used Borg for many years to run its applications. In 2014, Google decided to transfer its experience with Borg into a new open source project called "Kubernetes" (Greek for "helmsman" or "pilot"). In 2015, it became the first project donated to the newly founded Cloud Native Computing Foundation (CNCF).

From the start, Kubernetes gained a whole community of users, and the number of contributors grew incredibly fast. Today, Kubernetes is considered one of the most popular projects on GitHub. It is fair to claim that Kubernetes is the most commonly used and feature-rich container orchestration platform. Kubernetes also forms the foundation of other platforms built on top of it. The most prominent of those Platform-as-a-Service systems is Red Hat OpenShift, which provides various additional capabilities to Kubernetes. These are only some reasons we chose Kubernetes as the reference platform for the cloud native patterns in this book.

This book assumes you have some basic knowledge of Kubernetes. In Chapter 1, we recapitulate the core Kubernetes concepts and lay the foundation for the following patterns.

## Design Patterns

The concept of *design patterns* dates back to the 1970s and is from the field of architecture. Christopher Alexander, an architect and system theorist, and his team published the groundbreaking *A Pattern Language* (Oxford University Press) in 1977, which describes architectural patterns for creating towns, buildings, and other construction projects. Sometime later, this idea was adopted by the newly formed software industry. The most famous book in this area is *Design Patterns—Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides—the Gang of Four (Addison-Wesley). When we talk about the famous Singleton, Factories, or Delegation patterns, it's because of this defining work. Many other great pattern books have been written since then for various fields with different levels of granularity, like *Enterprise Integration Patterns* by Gregor Hohpe and Bobby Woolf (Addison-Wesley) or *Patterns of Enterprise Application Architecture* by Martin Fowler (Addison-Wesley).

In short, a *pattern* describes a *repeatable solution to a problem*.[1] This definition works for the patterns we describe in this book, except that we probably don't have as much variability in our solutions. A pattern is different from a recipe because instead of giving step-by-step instructions to solve a problem, it provides a blueprint for solving a whole class of similar problems. For example, the Alexandrian pattern *Beer Hall* describes how public drinking halls should be constructed where "strangers and friends are drinking companions" and not "anchors of the lonely." All halls built after this pattern look different but share common characteristics, such as open alcoves for groups of four to eight and a place where a hundred people can meet to enjoy beverages, music, and other activities.

However, a pattern does more than provide a solution. It is also about forming a language. The patterns in this book form a dense, noun-centric language in which each pattern carries a unique *name*. When this language is established, these names automatically evoke similar mental representations when people speak about these patterns. For example, when we talk about a table, anyone speaking English assumes we are talking about a piece of wood with four legs and a top on which you can put things. The same thing happens in software engineering when discussing a "factory." In an object-oriented programming language context, we immediately associate with a "factory" an object that produces other objects. Because we immediately know the solution behind the pattern, we can move on to tackle yet-unsolved problems.

1 Alexander and his team defined the original meaning in the context of architecture as follows: "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice." (*A Pattern Language*, Christopher Alexander et al., 1977.)

There are also other characteristics of a pattern language. For example, patterns are interconnected and can overlap so that they cover most of the problem space. Also, as already laid out in the original *A Pattern Language*, patterns have a different level of granularity and scope. More general patterns cover an extensive problem space and provide rough guidance on how to solve the problem. Granular patterns have a very concrete solution proposal but are less widely applicable. This book contains all sorts of patterns, and many patterns reference other patterns or may even include other patterns as part of the solution.

Another feature of patterns is that they follow a rigid format. However, each author defines a different form; unfortunately, there is no common standard for how patterns should be laid out. Martin Fowler gives an excellent overview of the formats used for pattern languages at "Writing Software Patterns".

## How This Book Is Structured

We chose a simple pattern format for this book. We do not follow any particular pattern description language. For each pattern, we use the following structure:

*Name*
: Each pattern carries a name, which is also the chapter's title. The name is the center of the pattern's language.

*Problem*
: This section gives the broader context and describes the pattern space in detail.

*Solution*
: This section shows how the pattern solves the problem in a Kubernetes-specific way. This section also contains cross-references to other patterns that are either related or part of the given pattern.

*Discussion*
: This section includes a discussion about the advantages and disadvantages of the solution for the given context.

*More Information*
: This final section contains additional information sources related to the pattern.

We organized the patterns in this book as follows:

- Part I, "Foundational Patterns", covers the core concepts of Kubernetes. These are the underlying principles and practices for building container-based cloud native applications.

- Part II, "Behavioral Patterns", describes patterns that build on top of foundational patterns and add the runtime aspect concepts of managing various types of containers.
- Part III, "Structural Patterns", contains patterns related to organizing containers within a *Pod*, which is the atom of the Kubernetes platform.
- Part IV, "Configuration Patterns", gives insight into the various ways application configuration can be handled in Kubernetes. These are granular patterns, including concrete recipes for connecting applications to their configuration.
- Part V, "Security Patterns", addresses various security concerns that arise when an application is containerized and deployed on Kubernetes.
- Part VI, "Advanced Patterns", is a collection of advanced concepts, such as how the platform itself can be extended or how to build container images directly within the cluster.

Depending on the context, the same pattern might fit into several categories. Every pattern chapter is self-contained; you can read chapters in isolation and in any order.

## Who This Book Is For

This book is for *developers* who want to design and develop cloud native applications and use Kubernetes as the platform. It is most suitable for readers who have some basic familiarity with containers and Kubernetes concepts and want to take it to the next level. However, you don't need to know the low-level details of Kubernetes to understand the use cases and patterns. Architects, consultants, and other technical personnel will also benefit from the repeatable patterns described here.

The book is based on use cases and lessons learned from real-world projects. It is an accumulation of best practices and patterns after years of working in this space. We want to help you understand the Kubernetes-first mindset and create better cloud native applications—not reinvent the wheel. It is written in a relaxed style and is similar to a series of essays that can be read independently.

Let's briefly look at what this book is *not*:

- This book is not an introduction to Kubernetes, nor is it a reference manual. We touch on many Kubernetes features and explain them in some detail, but we are focusing on the concepts behind those features. Chapter 1, "Introduction", offers a brief refresher on Kubernetes basics. If you are looking for a comprehensive book on Kubernetes, we highly recommend *Kubernetes in Action* by Marko Lukša (Manning Publications).
- This book is not a step-by-step guide on how to set up a Kubernetes cluster itself. Every example assumes you have Kubernetes up and running. You have several

options for trying out the examples. If you are interested in learning how to set up a Kubernetes cluster, we recommend *Kubernetes: Up and Running* by Brendan Burns, Joe Beda, Kelsey Hightower, and Lachlan Evenson (O'Reilly).

- This book is not about operating and governing a Kubernetes cluster for other teams. We deliberately skipped administrative and operational aspects of Kubernetes and took a developer-first view into Kubernetes. This book can help operations teams understand how a developer uses Kubernetes, but it is not sufficient for administering and automating a Kubernetes cluster. If you are interested in learning how to operate a Kubernetes cluster, we recommend *Kubernetes Best Practices* by Brendan Burns, Eddie Villalba, Dave Strebel, and Lachlan Evenson (O'Reilly).

## What You Will Learn

There's a lot to discover in this book. Some patterns may read like excerpts from a Kubernetes manual at first glance, but upon closer look, you'll see the patterns are presented from a conceptual angle not found in other books on the topic. Other patterns are explained with detailed steps to solve a concrete problem, as in Part IV, "Configuration Patterns". In some chapters, we explain Kubernetes features that don't fit nicely into a pattern definition. Don't get hung up on whether it is a pattern or a feature. In all chapters, we look at the forces involved from the first principles and focus on the use cases, lessons learned, and best practices. That is the valuable part.

Regardless of the pattern granularity, you will learn everything Kubernetes offers for each particular pattern, with plenty of examples to illustrate the concepts. All these examples have been tested, and we tell you how to get the complete source code in "Using Code Examples" on page xix.

## What's New in the Second Edition

The Kubernetes ecosystem has continued to grow since the first edition came out four years ago. As a result, there have been many Kubernetes releases, and more tools and patterns for using Kubernetes have become de facto standards.

Fortunately, most of the patterns described in our book have stood the test of time and remain valid. Therefore, we have updated these patterns, added new features up to Kubernetes 1.26, and removed obsolete and deprecated parts. For the most part, only minor changes were necessary, except for Chapter 29, "Elastic Scale", and Chapter 30, "Image Builder", which underwent significant changes due to new developments in these areas.

Additionally, we have included five new patterns and introduced a new category, Part V, "Security Patterns", which addresses a gap in the first edition and provides important security-related patterns for developers.

Our GitHub examples have been updated and extended. And, lastly, we added 50% more content for our readers to enjoy.

## Conventions Used in This Book

The following typographical conventions are used in this book:

*Italic*
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

As mentioned, patterns form a simple, interconnected language. To emphasize this web of patterns, each pattern is capitalized and set in italics, (e.g., *Sidecar*). When a pattern name is also a Kubernetes core concept (such as *Init Container* or *Controller*), we use this specific formatting only when we directly reference the pattern itself. Where it makes sense, we also interlink pattern chapters for ease of navigation.

We also use the following conventions:

- Everything you can type in a shell or editor is rendered in `constant width font`.
- Kubernetes resource names are always rendered in uppercase (e.g., Pod). If the resource is a combined name like ConfigMap, we keep it like this in favor of the more natural "config map" for clarity and to make it clear that it refers to a Kubernetes concept.
- Sometimes, a Kubernetes resource name is identical to a common concept like "service" or "node." In these cases, we use the resource name format only when referring to the resource itself.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

## Using Code Examples

Every pattern is backed with fully executable examples, which you can find on the accompanying web page. You can find the link to each pattern's example in each chapter's "More Information" section.

The "More Information" section also contains links to further information related to the pattern. We keep these lists updated in the example repository.

The source code for all examples in this book is available on GitHub. The repository and the website also have pointers and instructions on how to get a Kubernetes cluster to try out the examples. Please look at the provided resource files when you go through the examples. They contain many valuable comments that will further your understanding of the example code.

Many examples use a REST service called *random-generator* that returns random numbers when called. It is uniquely crafted to play well with the examples in this book. Its source can be found on GitHub as well, and its container image `k8spatterns/random-generator` is hosted on Docker Hub.

We use a JSON path notation to describe resource fields (e.g., `.spec.replicas` points to the `replicas` field of the resource's `spec` section).

If you find an issue in the example code or documentation or have a question, don't hesitate to open a ticket at the GitHub issue tracker. We monitor these GitHub issues and are happy to answer any questions.



We love code contributions too! If you think we can improve our examples, we are happy to hear from you. Just open a GitHub issue or create a pull request, and let's start a conversation.

## O'Reilly Online Learning

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, please visit *http://oreilly.com*.

## How to Contact Us

Please address comments and questions concerning this book to the publisher:


O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-889-8969 (in the United States or Canada)
707-829-7019 (international or local)
707-829-0104 (fax)
*support@oreilly.com*
*https://www.oreilly.com/about/contact.html*


We have a web page for this book where we list errata, examples, and additional information. You can access this page at *https://oreil.ly/kubernetes_patterns-2e*.

For news and information about our books and courses, visit *https://oreilly.com*.

Find us on LinkedIn: *https://linkedin.com/company/oreilly-media*

Follow us on Twitter: *https://twitter.com/oreillymedia*

Watch us on YouTube: *https://youtube.com/oreillymedia*

Follow the authors on Twitter: *https://twitter.com/bibryam*, *https://twitter.com/ro14nd*

Follow the authors on Mastodon: *https://fosstodon.org/@bilgin*, *https://hachyderm.io/@ro14nd*

Find the authors on GitHub: *https://github.com/bibryam*, *https://github.com/rhuss*

Follow their blogs: *https://www.ofbizian.com*, *https://ro14nd.de*

# Acknowledgments

Bilgin is forever grateful to his wonderful wife, Ayshe, for her endless support and patience as he worked on yet another book. He is also thankful for his adorable daughters, Selin and Esin, who always know how to bring a smile to his face. You mean the world to him. Finally, Bilgin would like to thank his fantastic coauthor, Roland, for making this project a reality.

Roland is deeply grateful for his wife Tanja's unwavering support and forbearance throughout the writing process, and he also thanks his son Jakob for his encouragement. Furthermore, Roland wishes to extend special recognition to Bilgin for his exceptional insights and writing, without which the book would not have come to fruition.

Creating two editions of this book was a long multiyear journey, and we want to thank our reviewers who kept us on the right track.

For the first edition, special kudos to Paolo Antinori and Andrea Tarocchi for helping us through the journey. Big thanks to Marko Lukša, Brandon Philips, Michael Hüttermann, Brian Gracely, Andrew Block, Jiri Kremser, Tobias Schneck, and Rick Wagner, who supported us with their expertise and advice. Last but not least, big thanks to our editors Virginia Wilson, John Devins, Katherine Tozer, Christina Edwards, and all the awesome folks at O'Reilly for helping us push this book over the finish line.

Completing the second edition was no easy feat, and we are grateful to all who supported us in finishing it. We extend our thanks to our technical reviewers, Ali Ok, Dávid Šimanský, Zbyněk Roubalík, Erkan Yanar, Christoph Stäbler, Andrew Block, and Adam Kaplan, as well as to our development editor, Rita Fernando, for her patience and encouragement throughout the whole process. Many kudos go out to the O'Reilly production team, especially Beth Kelly, Kim Sandoval, and Judith McConville, for their meticulous attention in finalizing the book.

We want to express a special thank you to Abhishek Koserwal for his tireless and dedicated efforts in Chapter 26, "Access Control". His contributions came at a time when we needed them the most and made an impact.

CHAPTER 1

# Introduction

In this introductory chapter, we set the scene for the rest of the book by explaining a few of the core Kubernetes concepts used for designing and implementing cloud native applications. Understanding these new abstractions, and the related principles and patterns from this book, is key to building distributed applications that can be automatable by Kubernetes.

This chapter is not a prerequisite for understanding the patterns described later. Readers familiar with Kubernetes concepts can skip it and jump straight into the pattern category of interest.

## The Path to Cloud Native

Microservices is among the most popular architectural styles for creating cloud native applications. They tackle software complexity through modularization of business capabilities and trading development complexity for operational complexity. That is why a key prerequisite for becoming successful with microservices is to create applications that can be operated at scale through Kubernetes.

As part of the microservices movement, there is a tremendous amount of theory, techniques, and supplemental tools for creating microservices from scratch or for splitting monoliths into microservices. Most of these practices are based on *Domain-Driven Design* by Eric Evans (Addison-Wesley) and the concepts of bounded contexts and aggregates. *Bounded contexts* deal with large models by dividing them into different components, and *aggregates* help to further group bounded contexts into modules with defined transaction boundaries. However, in addition to these business domain considerations, for each distributed system—whether it is based on microservices or not—there are also technical concerns around its external structure, and runtime coupling. Containers and container orchestrators such as Kubernetes bring in new

primitives and abstractions to address the concerns of distributed applications, and here we discuss the various options to consider when putting a distributed system into Kubernetes.

Throughout this book, we look at container and platform interactions by treating the containers as black boxes. However, we created this section to emphasize the importance of what goes into containers. Containers and cloud native platforms bring tremendous benefits to your distributed applications, but if all you put into containers is rubbish, you will get distributed rubbish at scale. Figure 1-1 shows the mixture of the skills required for creating good cloud native applications and where Kubernetes patterns fit in.

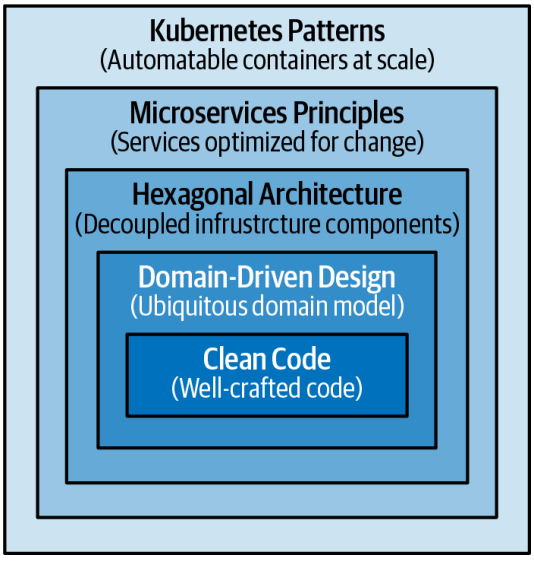



*Figure 1-1. The path to cloud native*

At a high level, creating good cloud native applications requires familiarity with multiple design techniques:

- At the lowest *code level*, every variable you define, every method you create, and every class you decide to instantiate plays a role in the long-term maintenance of the application. No matter what container technology and orchestration platform you use, the development team and the artifacts they create will have the most impact. It is important to grow developers who strive to write clean code, have the right number of automated tests, constantly refactor to improve code quality, and are guided by Software Craftsmanship principles at heart.
- *Domain-driven design* is about approaching software design from a business perspective with the intention of keeping the architecture as close to the real

world as possible. This approach works best for object-oriented programming languages, but there are also other good ways to model and design software for real-world problems. A model with the right business and transaction boundaries, easy-to-consume interfaces, and rich APIs is the foundation for successful containerization and automation later.

- The *hexagonal architecture* and its variations, such as Onion and Clean architectures, improve the flexibility and maintainability of applications by decoupling the application components and providing standardized interfaces for interacting with them. By decoupling the core business logic of a system from the surrounding infrastructure, hexagonal architecture makes it easier to port the system to different environments or platforms. These architectures complement domain-driven design and help arrange application code with distinct boundaries and externalized infrastructure dependencies.
- The *microservices architectural style* and the twelve-factor app methodology very quickly evolved to become the norm for creating distributed applications and they provide valuable principles and practices for designing changing distributed applications. Applying these principles lets you create implementations that are optimized for scale, resiliency, and pace of change, which are common requirements for any modern software today.
- *Containers* were very quickly adopted as the standard way of packaging and running distributed applications, whether these are microservices or functions. Creating modular, reusable containers that are good cloud native citizens is another fundamental prerequisite. *Cloud native* is a term used to describe principles, patterns, and tools to automate containerized applications at scale. We use *cloud native* interchangeably with *Kubernetes*, which is the most popular open source cloud native platform available today.

In this book, we are not covering clean code, domain-driven design, hexagonal architecture, or microservices. We are focusing only on the patterns and practices addressing the concerns of the container orchestration. But for these patterns to be effective, your application needs to be designed well from the inside by using clean code practices, domain-driven design, hexagonal architecture-like isolation of external dependencies, microservices principles, and other relevant design techniques.

## Distributed Primitives

To explain what we mean by new abstractions and primitives, here we compare them with the well-known object-oriented programming (OOP), and Java specifically. In the OOP universe, we have concepts such as class, object, package, inheritance, encapsulation, and polymorphism. Then the Java runtime provides specific features and guarantees on how it manages the lifecycle of our objects and the application as a whole.

The Java language and the Java Virtual Machine (JVM) provide local, in-process building blocks for creating applications. Kubernetes adds an entirely new dimension to this well-known mindset by offering a new set of distributed primitives and runtime for building distributed systems that spread across multiple nodes and processes. With Kubernetes at hand, we don't rely only on the local primitives to implement the whole application behavior.

We still need to use the object-oriented building blocks to create the components of the distributed application, but we can also use Kubernetes primitives for some of the application behaviors. Table 1-1 shows how various development concepts are realized differently with local and distributed primitives in the JVM and Kubernetes, respectively.

*Table 1-1. Local and distributed primitives*

| Concept | Local primitive | Distributed primitive |
|---|---|---|
| Behavior encapsulation | Class | Container image |
| Behavior instance | Object | Container |
| Unit of reuse | *.jar* | Container image |
| Composition | Class A contains Class B | Sidecar pattern |
| Inheritance | Class A extends Class B | A container's `FROM` parent image |
| Deployment unit | *.jar*/*.war*/*.ear* | Pod |
| Buildtime/Runtime isolation | Module, package, class | Namespace, Pod, container |
| Initialization preconditions | Constructor | Init container |
| Postinitialization trigger | Init-method | `postStart` |
| Predestroy trigger | Destroy-method | `preStop` |
| Cleanup procedure | `finalize()`, shutdown hook | - |
| Asynchronous and parallel execution | `ThreadPoolExecutor`, `ForkJoinPool` | Job |
| Periodic task | `Timer`, `ScheduledExecutorService` | CronJob |
| Background task | Daemon thread | DaemonSet |
| Configuration management | `System.getenv()`, `Properties` | ConfigMap, Secret |

The in-process primitives and the distributed primitives have commonalities, but they are not directly comparable and replaceable. They operate at different abstraction levels and have different preconditions and guarantees. Some primitives are supposed to be used together. For example, we still have to use classes to create objects and put them into container images. However, some other primitives such as CronJob in Kubernetes can completely replace the `ExecutorService` behavior in Java.

Next, let's see a few distributed abstractions and primitives from Kubernetes that are especially interesting for application developers.

## Containers

*Containers* are the building blocks for Kubernetes-based cloud native applications. If we make a comparison with OOP and Java, container images are like classes, and containers are like objects. The same way we can extend classes to reuse and alter behavior, we can have container images that extend other container images to reuse and alter behavior. The same way we can do object composition and use functionality, we can do container compositions by putting containers into a Pod and using collaborating containers.

If we continue the comparison, Kubernetes would be like the JVM but spread over multiple hosts, and it would be responsible for running and managing the containers. Init containers would be something like object constructors; DaemonSets would be similar to daemon threads that run in the background (like the Java Garbage Collector, for example). A Pod would be something similar to an Inversion of Control (IoC) context (Spring Framework, for example), where multiple running objects share a managed lifecycle and can access one another directly.

The parallel doesn't go much further, but the point is that containers play a fundamental role in Kubernetes, and creating modularized, reusable, single-purpose container images is fundamental to the long-term success of any project and even the containers' ecosystem as a whole. Apart from the technical characteristics of a container image that provide packaging and isolation, what does a container represent, and what is its purpose in the context of a distributed application? Here are a few suggestions on how to look at containers:

- A container image is the unit of functionality that addresses a single concern.
- A container image is owned by one team and has its own release cycle.
- A container image is self-contained and defines and carries its runtime dependencies.
- A container image is immutable, and once it is built, it does not change; it is configured.
- A container image defines its resource requirements and external dependencies.
- A container image has well-defined APIs to expose its functionality.
- A container typically runs as a single Unix process.
- A container is disposable and safe to scale up or down at any moment.

In addition to all these characteristics, a proper container image is modular. It is parameterized and created for reuse in the different environments in which it is going to run. Having small, modular, and reusable container images leads to the creation of more specialized and stable container images in the long term, similar to a great reusable library in the programming language world.