OʻREILLY

NGINX

Compliments of

Cloud Native DevOps with **Kubernetes**

Building, Deploying, and Scaling Modern Applications in the Cloud

John Arundel & Justin Domingus

| LIS

Try NGINX Plus N+ and NGINX WAF

**free for 30 days**

Get high-performance application deliver*y f*or microservices. NGINX Plus is a software load **balancer, web serve**r, and content cache.

The NGINX Web Application Firewall (W*A*F) **protects applications a**gainst sophisticated **Layer 7 attacks.**

Cost Savings Over 80% cost savings compared to hardware application delivery con trollers and WAFS, with all the performance and features you expect.

Reduced Complexity

The only all-in-one load balancer, content

cache, web server, and web application firewall helps reduce infrastructure sprawl.

Exclusive Features JWT authentication, high availability, the NGINX Plus API, and

other advanced functionality are only available in NGINX Plus.

NGINX WAF

A trial of the NGINX *W*AF, based

on ModSecurity, is included when you download a trial of

NGINX Plus.

NGINX

Download at nginx.com/freetrial

**Praise for *Cloud Native DevOps with Kubernetes***

*Cloud Native DevOps* is an essential guide to operating today’s distributed systems. A super clear and informative read, covering all the details without compromising readability. I learned a lot, and definitely have some action points to take away! *—Will Thames, Platform Engineer, Skedulo*

The most encompassing, definitive, and practical text about the care and feeding of Kubernetes infrastructure. An absolute must-have. *—Jeremy Yates, SRE Team, The Home Depot QuoteCenter*

I wish I’d had this book when I started! This is a must-read for everyone developing and running applications in Kubernetes. *—Paul van der Linden, Lead Developer, vdL Software Consultancy*

This book got me really excited. It’s a goldmine of information for anyone looking to use Kubernetes, and I feel like I’ve levelled up! *—Adam McPartlan (@mcparty), Senior Systems Engineer, NYnet*

I really enjoyed reading this book. It’s very informal in style, but authoritative at the same time. It contains lots of great practical advice. Exactly the sort of information that everybody wants to know, but doesn’t know how to get, other than through first-hand experience. *—Nigel Brown, cloud native trainer and course author*

**Cloud Native DevOps Kubernetes with *Building, Deploying, and Scaling Modern Applications in the Cloud***

Beijing Beijing

Boston Boston Farnham Farnham Sebastopol Sebastopol Tokyo Tokyo ***John Arundel and Justin Domingus***

**Cloud Native DevOps with Kubernetes** by John Arundel and Justin Domingus

Copyright © 2019 John Arundel and Justin Domingus. All rights reserved.

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/safari*). For more information, contact our corporate/insti‐ tutional sales department: 800-998-9938 or *corporate@oreilly.com*.

**Acquisitions Editor:** Rachel Roumeliotis **Developmental Editors:** Virginia Wilson and Nikki McDonald **Production Editor:** Nan Barber **Copyeditor:** Kim Cofer **Proofreader:** Amanda Kersey **Indexer:** Judith McConville **Interior Designer:** David Futato **Cover Designer:** Karen Montgomery **Illustrator:** Rebecca Demarest

February 2019: First Edition

**Revision History for the First Edition** 2019-01-24: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492040767* for release details.

The O’Reilly logo is a registered trademark of O’Reilly Media, Inc. *Cloud Native DevOps with Kubernetes*, the cover image, and related trade dress are trademarks of O’Reilly Media, Inc.

The views expressed in this work are those of the authors, and do not represent the publisher’s views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

This work is part of a collaboration between O’Reilly and NGINX. See our statement of editorial inde‐ pendence (*http://www.oreilly.com/about/editorial\_independence.html*).

978-1-492-04076-7

[LSI]

**Table of Contents**

**Foreword by NGINX. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . xix**

**Foreword by Ihor Dvoretskyi. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . xxi**

**Preface. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . xxiii**

**1. Revolution in the Cloud. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1** The Creation of the Cloud 2 Buying Time 3 Infrastructure as a Service 3 The Dawn of DevOps 3 Nobody Understands DevOps 5 The Business Advantage 5 Infrastructure as Code 6 Learning Together 6 The Coming of Containers 7 The State of the Art 7 Thinking Inside the Box 8 Putting Software in Containers 8 Plug and Play Applications 9 Conducting the Container Orchestra 10 Kubernetes 11 From Borg to Kubernetes 11 What Makes Kubernetes So Valuable? 11 Will Kubernetes Disappear? 13 Kubernetes Doesn’t Do It All 13 Cloud Native 14 The Future of Operations 16**vii**

Distributed DevOps 17 Some Things Will Remain Centralized 17 Developer Productivity Engineering 17 You Are the Future 18 Summary 19

**2. First Steps with Kubernetes. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21** Running Your First Container 21 Installing Docker Desktop 22 What Is Docker? 22 Running a Container Image 22 The Demo Application 23 Looking at the Source Code 23 Introducing Go 24 How the Demo App Works 24 Building a Container 25 Understanding Dockerfiles 25 Minimal Container Images 26 Running docker image build 26 Naming Your Images 27 Port Forwarding 27 Container Registries 28 Authenticating to the Registry 28 Naming and Pushing Your Image 28 Running Your Image 29 Hello, Kubernetes 29 Running the Demo App 30 If the Container Doesn’t Start 30 Minikube 31 Summary 31

**3. Getting Kubernetes. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33** Cluster Architecture 33 The Control Plane 34 Node Components 35 High Availability 35 The Costs of Self-Hosting Kubernetes 37 It’s More Work Than You Think 37 It’s Not Just About the Initial Setup 38 Tools Don’t Do All the Work for You 39 Kubernetes Is Hard 39 Administration Overhead 39

**viii | Table of Contents**

Start with Managed Services 40 Managed Kubernetes Services 41 Google Kubernetes Engine (GKE) 41 Cluster Autoscaling 42 Amazon Elastic Container Service for Kubernetes (EKS) 42 Azure Kubernetes Service (AKS) 43 OpenShift 43 IBM Cloud Kubernetes Service 43 Heptio Kubernetes Subscription (HKS) 43 Turnkey Kubernetes Solutions 44 Stackpoint 44 Containership Kubernetes Engine (CKE) 44 Kubernetes Installers 44 kops 45 Kubespray 45 TK8 45 Kubernetes The Hard Way 45 kubeadm 46 Tarmak 46 Rancher Kubernetes Engine (RKE) 46 Puppet Kubernetes Module 46 Kubeformation 46 Buy or Build: Our Recommendations 47 Run Less Software 47 Use Managed Kubernetes if You Can 48 But What About Vendor Lock-in? 48 Use Standard Kubernetes Self-Hosting Tools if You Must 49 When Your Choices Are Limited 49 Bare-Metal and On-Prem 49 Clusterless Container Services 50 Amazon Fargate 50 Azure Container Instances (ACI) 51 Summary 51

**4. Working with Kubernetes Objects. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53** Deployments 53 Supervising and Scheduling 54 Restarting Containers 54 Querying Deployments 55 Pods 55 ReplicaSets 56 Maintaining Desired State 57

**Table of Contents | ix**

The Kubernetes Scheduler 58 Resource Manifests in YAML Format 59 Resources Are Data 59 Deployment Manifests 59 Using kubectl apply 60 Service Resources 60 Querying the Cluster with kubectl 63 Taking Resources to the Next Level 64 Helm: A Kubernetes Package Manager 64 Installing Helm 65 Installing a Helm Chart 65 Charts, Repositories, and Releases 66 Listing Helm Releases 67 Summary 67

**5. Managing Resources. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69** Understanding Resources 69 Resource Units 70 Resource Requests 70 Resource Limits 71 Keep Your Containers Small 72 Managing the Container Life Cycle 72 Liveness Probes 72 Probe Delay and Frequency 73 Other Types of Probes 73 gRPC Probes 74 Readiness Probes 74 File-Based Readiness Probes 75 minReadySeconds 75 Pod Disruption Budgets 76 Using Namespaces 77 Working with Namespaces 78 What Namespaces Should I Use? 78 Service Addresses 79 Resource Quotas 79 Default Resource Requests and Limits 80 Optimizing Cluster Costs 81 Optimizing Deployments 82 Optimizing Pods 83 Vertical Pod Autoscaler 84 Optimizing Nodes 84 Optimizing Storage 85

**x | Table of Contents**

Cleaning Up Unused Resources 86 Checking Spare Capacity 88 Using Reserved Instances 88 Using Preemptible (Spot) Instances 89 Keeping Your Workloads Balanced 91 Summary 92

**6. Operating Clusters. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95** Cluster Sizing and Scaling 95 Capacity Planning 96 Nodes and Instances 98 Scaling the Cluster 101 Conformance Checking 102 CNCF Certification 103 Conformance Testing with Sonobuoy 104 Validation and Auditing 105 K8Guard 106 Copper 106 kube-bench 107 Kubernetes Audit Logging 107 Chaos Testing 107 Only Production Is Production 108 chaoskube 108 kube-monkey 109 PowerfulSeal 109 Summary 110

**7. Kubernetes Power Tools. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 111** Mastering kubectl 111 Shell Aliases 111 Using Short Flags 112 Abbreviating Resource Types 112 Auto-Completing kubectl Commands 113 Getting Help 113 Getting Help on Kubernetes Resources 114 Showing More Detailed Output 114 Working with JSON Data and jq 114 Watching Objects 115 Describing Objects 116 Working with Resources 116 Imperative kubectl Commands 116 When Not to Use Imperative Commands 117

**Table of Contents | xi**

Generating Resource Manifests 118 Exporting Resources 118 Diffing Resources 119 Working with Containers 119 Viewing a Container’s Logs 119 Attaching to a Container 121 Watching Kubernetes Resources with kubespy 121 Forwarding a Container Port 121 Executing Commands on Containers 122 Running Containers for Troubleshooting 122 Using BusyBox Commands 124 Adding BusyBox to Your Containers 124 Installing Programs on a Container 125 Live Debugging with kubesquash 125 Contexts and Namespaces 126 kubectx and kubens 128 kube-ps1 128 Kubernetes Shells and Tools 129 kube-shell 129 Click 129 kubed-sh 130 Stern 130 Building Your Own Kubernetes Tools 130 Summary 131

**8. Running Containers. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 133** Containers and Pods 133 What Is a Container? 134 What Belongs in a Container? 135 What Belongs in a Pod? 136 Container Manifests 136 Image Identifiers 137 The latest Tag 138 Container Digests 138 Base Image Tags 139 Ports 139 Resource Requests and Limits 139 Image Pull Policy 140 Environment Variables 140 Container Security 141 Running Containers as a Non-Root User 142 Blocking Root Containers 142

**xii | Table of Contents**

Setting a Read-Only Filesystem 143 Disabling Privilege Escalation 143 Capabilities 144 Pod Security Contexts 145 Pod Security Policies 145 Pod Service Accounts 146 Volumes 147 emptyDir Volumes 147 Persistent Volumes 148 Restart Policies 149 Image Pull Secrets 149 Summary 150

**9. Managing Pods. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 153** Labels 153 What Are Labels? 153 Selectors 154 More Advanced Selectors 155 Other Uses for Labels 156 Labels and Annotations 156 Node Affinities 157 Hard Affinities 158 Soft Affinities 158 Pod Affinities and Anti-Affinities 159 Keeping Pods Together 159 Keeping Pods Apart 160 Soft Anti-Affinities 161 When to Use Pod Affinities 161 Taints and Tolerations 162 Pod Controllers 163 DaemonSets 164 StatefulSets 165 Jobs 166 Cronjobs 167 Horizontal Pod Autoscalers 167 PodPresets 169 Operators and Custom Resource Definitions (CRDs) 170 Ingress Resources 171 Ingress Rules 172 Terminating TLS with Ingress 172 Ingress Controllers 173 Istio 174

**Table of Contents | xiii**

Envoy 175 Summary 175

**10. Configuration and Secrets. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 179** ConfigMaps 179 Creating ConfigMaps 180 Setting Environment Variables from ConfigMaps 181 Setting the Whole Environment from a ConfigMap 183 Using Environment Variables in Command Arguments 184 Creating Config Files from ConfigMaps 184 Updating Pods on a Config Change 186 Kubernetes Secrets 187 Using Secrets as Environment Variables 187 Writing Secrets to Files 188 Reading Secrets 189 Access to Secrets 190 Encryption at Rest 190 Keeping Secrets 191 Secrets Management Strategies 191 Encrypt Secrets in Version Control 191 Store Secrets Remotely 192 Use a Dedicated Secrets Management Tool 193 Recommendations 193 Encrypting Secrets with Sops 194 Introducing Sops 194 Encrypting a File with Sops 195 Using a KMS Backend 196 Summary 197

**11. Security and Backups. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 199** Access Control and Permissions 199 Managing Access by Cluster 199 Introducing Role-Based Access Control (RBAC) 200 Understanding Roles 201 Binding Roles to Users 201 What Roles Do I Need? 202 Guard Access to Cluster-Admin 202 Applications and Deployment 203 RBAC Troubleshooting 204 Security Scanning 204 Clair 204 Aqua 205

**xiv | Table of Contents**

Anchore Engine 205 Backups 206 Do I Need to Back Up Kubernetes? 206 Backing Up etcd 206 Backing Up Resource State 207 Backing Up Cluster State 207 Large and Small Disasters 208 Velero 208 Monitoring Cluster Status 211 kubectl 211 CPU and Memory Utilization 213 Cloud Provider Console 213 Kubernetes Dashboard 214 Weave Scope 216 kube-ops-view 216 node-problem-detector 216 Further Reading 217 Summary 217

**12. Deploying Kubernetes Applications. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 219** Building Manifests with Helm 219 What’s Inside a Helm Chart? 220 Helm Templates 221 Interpolating Variables 222 Quoting Values in Templates 223 Specifying Dependencies 223 Deploying Helm Charts 223 Setting Variables 223 Specifying Values in a Helm Release 224 Updating an App with Helm 225 Rolling Back to Previous Versions 225 Creating a Helm Chart Repo 226 Managing Helm Chart Secrets with Sops 226 Managing Multiple Charts with Helmfile 228 What’s in a Helmfile? 228 Chart Metadata 229 Applying the Helmfile 230 Advanced Manifest Management Tools 231 ksonnet 231 Kapitan 232 kustomize 233 kompose 233

**Table of Contents | xv**

Ansible 234 kubeval 234 Summary 235

**13. Development Workflow. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 237** Development Tools 237 Skaffold 237 Draft 238 Telepresence 238 Knative 238 Deployment Strategies 239 Rolling Updates 239 Recreate 240 maxSurge and maxUnavailable 240 Blue/Green Deployments 241 Rainbow Deployments 242 Canary Deployments 242 Handling Migrations with Helm 243 Helm Hooks 243 Handling Failed Hooks 244 Other Hooks 244 Chaining Hooks 244 Summary 245

**14. Continuous Deployment in Kubernetes. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 247** What Is Continuous Deployment? 247 Which CD Tool Should I Use? 248 Jenkins 248 Drone 249 Google Cloud Build 249 Concourse 249 Spinnaker 249 GitLab CI 250 Codefresh 250 Azure Pipelines 250 CD Components 250 Docker Hub 250 Gitkube 250 Flux 251 Keel 251 A CD Pipeline with Cloud Build 251 Setting Up Google Cloud and GKE 251

**xvi | Table of Contents**

Forking the Demo Repository 252 Introducing Cloud Build 252 Building the Test Container 252 Running the Tests 253 Building the Application Container 253 Validating the Kubernetes Manifests 254 Publishing the Image 254 Git SHA Tags 254 Creating the First Build Trigger 255 Testing the Trigger 255 Deploying from a CD Pipeline 257 Creating a Deploy Trigger 259 Optimizing Your Build Pipeline 259 Adapting the Example Pipeline 260 Summary 260

**15. Observability and Monitoring. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 263** What Is Observability? 263 What Is Monitoring? 263 Black-Box Monitoring 263 What Does “Up” Mean? 265 Logging 266 Introducing Metrics 268 Tracing 269 Observability 270 The Observability Pipeline 271 Monitoring in Kubernetes 272 External Black-Box Checks 272 Internal Health Checks 274 Summary 276

**16. Metrics in Kubernetes. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 277** What Are Metrics, Really? 277 Time Series Data 278 Counters and Gauges 278 What Can Metrics Tell Us? 279 Choosing Good Metrics 279 Services: The RED Pattern 280 Resources: The USE Pattern 281 Business Metrics 282 Kubernetes Metrics 283 Analyzing Metrics 286

**Table of Contents | xvii**

What’s Wrong with a Simple Average? 287 Means, Medians, and Outliers 287 Discovering Percentiles 288 Applying Percentiles to Metrics Data 288 We Usually Want to Know the Worst 290 Beyond Percentiles 290 Graphing Metrics with Dashboards 291 Use a Standard Layout for All Services 291 Build an Information Radiator with Master Dashboards 292 Dashboard Things That Break 293 Alerting on Metrics 295 What’s Wrong with Alerts? 295 On-call Should Not Be Hell 296 Urgent, Important, and Actionable Alerts 296 Track Your Alerts, Out-of-Hours Pages, and Wake-ups 297 Metrics Tools and Services 298 Prometheus 298 Google Stackdriver 300 AWS Cloudwatch 301 Azure Monitor 301 Datadog 301 New Relic 302 Summary 303

**Afterword. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 305**

**Index. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 307**

**xviii | Table of Contents**

**Foreword by NGINX**

Digital transformation initiatives in enterprises today are driving innovation and agil‐ ity as well as delivering superior customer experiences for global consumers. Embrac‐ ing technologies and software development methodologies such as cloud computing, containers, and DevOps have been pivotal to digital transformation.

Containers offer a host of advantages—they are portable and resource efficient, pro‐ vide effective isolation, and accelerate the development process. As you build new cloud-native applications using microservices or migrate existing ones to this new environment, building, deploying, and scaling these applications will present its own set of challenges. Service discovery and load balancing, storage, failure recovery, roll‐ backs, and configuration management all have to be performed at scale for objects that are usually ephemeral, sometimes across a multi-cloud environment. Creating a robust continuous integration and continuous delivery (CI/CD) process in order to realize agility and high feature velocity further adds to this complexity.

Kubernetes is the most pervasive container orchestration platform to address these challenges. It lets you manage complex application deployments quickly in a predict‐ able and reliable manner.

This is a must-have book for both beginning and experienced users of Kubernetes. Authors John Arundel and Justin Domingus provide in-depth guidance in an incre‐ mental and logical manner, covering the entire lifecycle of managing a Kubernetes environment, including initial deployment, runtime operation, security, and ongoing maintenance and monitoring. As you’ll see throughout this book, NGINX Open Source and our commercial-grade application delivery platform, NGINX Plus, are key components in successful production deployments with Kubernetes.

On the frontend, NGINX Open Source and NGINX Plus act as a stable entry point to your Kubernetes services—a persistent and reliable connection for external clients. NGINX Open Source provides foundational routing and load balancing capabilities to manage traffic across your containers. NGINX Plus, built on top of NGINX Open Source, provides advanced capabilities. It integrates well with service discovery plat‐**xix**

forms, achieves session persistence for stateful applications, authenticates APIs using JSON Web Token (JWT), and can be reconfigured dynamically when the set of ser‐ vice instances changes.

As you move to a Kubernetes infrastructure, NGINX Plus helps you achieve enterprise-grade traffic management for your containerized applications. We sin‐ cerely hope you enjoy this book as it helps you to succeed with your production deployments using Kubernetes.

*— Karthik Krishnaswamy Product Marketer, NGINX, Inc. December 2018*

**xx | Foreword by NGINX**

**Foreword by Ihor Dvoretskyi**

Welcome to *Cloud Native DevOps with Kubernetes*.

Kubernetes is a real industry revolution. Just a brief look at the Cloud Native Com‐ puting Foundation’s Landscape, which contains data about more than 600 projects that exist in the cloud native world today, highlights the importance of Kubernetes these days. Not all these tools were developed for Kubernetes, not all of them can even be used with Kubernetes, but all of them are part of the huge ecosystem where Kubernetes is one of the flagship technologies.

Kubernetes changed the way applications are developed and operated. It’s a core com‐ ponent in the DevOps world today. Kubernetes brings flexibility to developers and freedom to operations. Today you can use Kubernetes on any major cloud provider, on bare-metal on-premises environments, as well as on a local developer’s machine. Stability, flexibility, a powerful API, open code, and an open developer community are a few reasons why Kubernetes became an industry standard, just as Linux is a standard in the world of operating systems.

*Cloud Native DevOps with Kubernetes* is a great handbook for people who are per‐ forming their day-to-day activities with Kubernetes or are just starting their Kuber‐ netes journey. John and Justin cover all the major aspects of deploying, configuring, and operating Kubernetes and the best practices for developing and running applica‐ tions on it. They also give a great overview of the related technologies, including Prometheus, Helm, and continuous deployment. This is a must-read book for every‐ one in the DevOps world.

**xxi**

Kubernetes is not just yet another exciting tool; it is an industry standard and the foundation for next-generation technologies including serverless (OpenFaaS, Kna‐ tive) and machine learning (Kubeflow) tools. The entire IT industry is changing because of the cloud native revolution, and it’s hugely exciting to be living through it.

*— Ihor Dvoretskyi Developer Advocate, Cloud Native Computing Foundation December 2018*

**xxii | Foreword by Ihor Dvoretskyi**

**Preface**

In the world of IT operations, the key principles of DevOps have become well under‐ stood and widely adopted, but now the landscape is changing. A new application platform called Kubernetes has become rapidly adopted by companies all over the world and in all kinds of different industries. As more and more applications and businesses move from traditional servers to the Kubernetes environment, people are asking how to do DevOps in this new world.

This book explains what DevOps means in a cloud native world where Kubernetes is the standard platform. It will help you select the best tools and frameworks from the Kubernetes ecosystem. It will also present a coherent way to use those tools and frameworks, offering battle-tested solutions that are running right now, in produc‐ tion, for real.

**What Will I Learn?**

You’ll learn what Kubernetes is, where it comes from, and what it means for the future of software development and operations. You’ll learn how containers work, how to build and manage them, and how to design cloud native services and infra‐ structure.

You’ll understand the trade-offs between building and hosting Kubernetes clusters yourself, and using managed services. You’ll learn the capabilities, limitations, and pros and cons of popular Kubernetes installation tools such as kops, kubeadm, and Kubespray. You’ll get an informed overview of the major managed Kubernetes offer‐ ings from the likes of Amazon, Google, and Microsoft.

You’ll get hands-on practical experience of writing and deploying Kubernetes applica‐ tions, configuring and operating Kubernetes clusters, and automating cloud infra‐ structure and deployments with tools such as Helm. You’ll learn about Kubernetes support for security, authentication, and permissions, including Role-Based Access

**xxiii**

Control (RBAC), and best practices for securing containers and Kubernetes in pro‐ duction.

You’ll learn how to set up continuous integration and deployment with Kubernetes, how to back up and restore data, how to test your cluster for conformance and relia‐ bility, how to monitor, trace, log, and aggregate metrics, and how to make your Kubernetes infrastructure scalable, resilient, and cost-effective.

To illustrate all the things we talk about, we apply them to a very simple demo appli‐ cation. You can follow along with all our examples using the code from our Git repo.

**Who Is This Book For?**

This book is most directly relevant to IT operations staff responsible for servers, applications, and services, and developers responsible for either building new cloud native services, or migrating existing applications to Kubernetes and cloud. We assume no prior knowledge of Kubernetes or containers—don’t worry, we’ll walk you through all that.

Experienced Kubernetes users should still find much valuable material in the book: it covers advanced topics such as RBAC, continuous deployment, secrets management, and observability. Whatever your level of expertise, we hope you’ll find something useful in these pages.

**What Questions Does This Book Answer?**

In planning and writing this book, we spoke to hundreds of people about cloud native and Kubernetes, ranging from industry leaders and experts to complete begin‐ ners. Here are some of the questions they said they wanted a book like this to answer:

• “I’d like to learn why I should invest my time in this technology. What problems will it help to solve for me and my team?”

• “Kubernetes seems great, but it’s quite a steep learning curve. Setting up a quick demo is easy, but operating and troubleshooting it seems daunting. We’d like some solid guidance on how people are running Kubernetes clusters in the real world, and what problems we’re likely to encounter.”

• “Opinionated advice would be useful. The Kubernetes ecosystem has too many options for beginning teams to choose between. When there are multiple ways of doing the same thing, which one is best? How do we choose?”

And perhaps the most important question of all:

• “How do I use Kubernetes without breaking my company?”

**xxiv | Preface**

We kept these questions, and many others, firmly in mind while writing this book, and we’ve done our level best to answer them. How did we do? Turn the page to find out. **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 ele‐ ments such as variable or function names, databases, data types, environment variables, statements, and keywords.

**Constant width bold**

Shows commands or other text that should be typed literally by the user.

*Constant width italic*

Shows text that should be replaced with user-supplied values or by values deter‐ mined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

**Preface | xxv**

**Using Code Examples**

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/cloudnativedevops/demo*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you’re reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O’Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a signifi‐ cant amount of example code from this book into your product’s documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*Cloud Native DevOps with Kuber‐ netes* by John Arundel and Justin Domingus (O’Reilly). Copyright 2019 John Arundel and Justin Domingus, 978-1-492-04076-7.”

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*. **O’Reilly Safari**

*Safari* (formerly Safari Books Online) is a membership-based training and reference platform for enterprise, government, educators, and individuals.

Members have access to thousands of books, training videos, Learning Paths, interac‐ tive tutorials, and curated playlists from over 250 publishers, including O’Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesley Profes‐ sional, Microsoft Press, Sams, Que, Peachpit Press, Adobe, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, and Course Technology, among others.

For more information, please visit *http://oreilly.com/safari*.

**xxvi | Preface**

**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 800-998-9938 (in the United States or Canada) 707-829-0515 (international or local) 707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/2BaE1iq*.

To comment or ask technical questions about this book, send email to *bookques‐ tions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our web‐ site at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

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

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

**Acknowledgments**

Our grateful thanks are due to the many people who read early drafts of this book and gave us invaluable feedback and advice, or assisted us in other ways, including (but not limited to) Abby Bangser, Adam J. McPartlan, Adrienne Domingus, Alexis Richardson, Aron Trauring, Camilla Montonen, Gabriell Nascimento, Hannah Klemme, Hans Findel, Ian Crosby, Ian Shaw, Ihor Dvoretskyi, Ike Devolder, Jeremy Yates, Jérôme Petazzoni, Jessica Deen, John Harris, Jon Barber, Kitty Karate, Marco Lancini, Mark Ellens, Matt North, Michel Blanc, Mitchell Kent, Nicolas Steinmetz, Nigel Brown, Patrik Duditš, Paul van der Linden, Philippe Ensarguet, Pietro Mam‐ berti, Richard Harper, Rick Highness, Sathyajith Bhat, Suresh Vishnoi, Thomas Lia‐ kos, Tim McGinnis, Toby Sullivan, Tom Hall, Vincent De Smet, and Will Thames.

**Preface | xxvii**

**CHAPTER 1 Revolution in the Cloud**

There was never a time when the world began, because it goes round and round like a circle, and there is no place on a circle where it begins.

—Alan Watts

There’s a revolution going on. Actually, three revolutions.

The first revolution is the creation of the cloud, and we’ll explain what that is and why it’s important. The second is the dawn of DevOps, and you’ll find out what that involves and how it’s changing operations. The third revolution is the coming of con‐ tainers. Together, these three waves of change are creating a new software world: the *cloud native* world. The operating system for this world is called Kubernetes.

In this chapter, we’ll briefly recount the history and significance of these revolutions, and explore how the changes are affecting the way we all deploy and operate software. We’ll outline what *cloud native* means, and what changes you can expect to see in this new world if you work in software development, operations, deployment, engineer‐ ing, networking, or security.

Thanks to the effects of these interlinked revolutions, we think the future of comput‐ ing lies in cloud-based, containerized, distributed systems, dynamically managed by automation, on the Kubernetes platform (or something very like it). The art of devel‐ oping and running these applications—*cloud native DevOps*—is what we’ll explore in the rest of this book.

If you’re already familiar with all of this background material, and you just want to start having fun with Kubernetes, feel free to skip ahead to Chapter 2. If not, settle down comfortably, with a cup of your favorite beverage, and we’ll begin.

**1**

**The Creation of the Cloud**

In the beginning (well, the 1960s, anyway), computers filled rack after rack in vast, remote, air-conditioned data centers, and users would never see them or interact with them directly. Instead, developers submitted their jobs to the machine remotely and waited for the results. Many hundreds or thousands of users would all share the same computing infrastructure, and each would simply receive a bill for the amount of pro‐ cessor time or resources she used.

It wasn’t cost-effective for each company or organization to buy and maintain its own computing hardware, so a business model emerged where users would share the computing power of remote machines, owned and run by a third party.

If that sounds like right now, instead of last century, that’s no coincidence. The word *revolution* means “circular movement,” and computing has, in a way, come back to where it began. While computers have gotten a lot more powerful over the years— today’s Apple Watch is the equivalent of about three of the mainframe computers shown in Figure 1-1—shared, pay-per-use access to computing resources is a very old idea. Now we call it *the cloud*, and the revolution that began with timesharing main‐ frames has come full circle.

*Figure 1-1. Early cloud computer: the IBM System/360 Model 91, at NASA’s Goddard Space Flight Center*

**2 | Chapter 1: Revolution in the Cloud**

**Buying Time**

The central idea of the cloud is this: instead of buying a *computer*, you buy *compute*. That is, instead of sinking large amounts of capital into physical machinery, which is hard to scale, breaks down mechanically, and rapidly becomes obsolete, you simply buy time on someone else’s computer, and let them take care of the scaling, mainte‐ nance, and upgrading. In the days of bare-metal machines—the “Iron Age”, if you like —computing power was a capital expense. Now it’s an operating expense, and that has made all the difference.

The cloud is not just about remote, rented computing power. It is also about dis‐ tributed systems. You may buy raw compute resource (such as a Google Compute instance, or an AWS Lambda function) and use it to run your own software, but increasingly you also rent *cloud services*: essentially, the use of someone else’s soft‐ ware. For example, if you use PagerDuty to monitor your systems and alert you when something is down, you’re using a cloud service (sometimes called *software as a ser‐ vice*, or SaaS).

**Infrastructure as a Service**

When you use cloud infrastructure to run your own services, what you’re buying is *infrastructure as a service* (IaaS). You don’t have to expend capital to purchase it, you don’t have to build it, and you don’t have to upgrade it. It’s just a commodity, like elec‐ tricity or water. Cloud computing is a revolution in the relationship between busi‐ nesses and their IT infrastructure.

Outsourcing the hardware is only part of the story; the cloud also allows you to out‐ source the *software* that you don’t write: operating systems, databases, clustering, rep‐ lication, networking, monitoring, high availability, queue and stream processing, and all the myriad layers of software and configuration that span the gap between your code and the CPU. Managed services can take care of almost all of this *undifferenti‐ ated heavy lifting* for you (you’ll find out more about the benefits of managed services in Chapter 3).

The revolution in the cloud has also triggered another revolution in the people who use it: the DevOps movement. **The Dawn of DevOps**

Before DevOps, developing and operating software were essentially two separate jobs, performed by two different groups of people. *Developers* wrote software, and they passed it on to *operations* staff, who ran and maintained the software *in production* (that is to say, serving real users, instead of merely running under test conditions). Like computers that need their own floor of the building, this separation has its roots

**The Dawn of DevOps | 3**

in the middle of the last century. Software development was a very specialist job, and so was computer operation, and there was very little overlap between the two.

Indeed, the two departments had quite different goals and incentives, which often conflicted with each other (Figure 1-2). Developers tend to be focused on shipping new features quickly, while operations teams care about making services stable and reliable over the long term.

*Figure 1-2. Separate teams can lead to conflicting incentives (photo by Dave Roth)*

When the cloud came on the horizon, things changed. Distributed systems are com‐ plex, and the internet is very big. The technicalities of operating the system—recover‐ ing from failures, handling timeouts, smoothly upgrading versions—are not so easy to separate from the design, architecture, and implementation of the system.

Further, “the system” is no longer just your software: it comprises in-house software, cloud services, network resources, load balancers, monitoring, content distribution networks, firewalls, DNS, and so on. All these things are intimately interconnected and interdependent. The people who write the software have to understand how it relates to the rest of the system, and the people who operate the system have to understand how the software works—or fails.

The origins of the DevOps movement lie in attempts to bring these two groups together: to collaborate, to share understanding, to share responsibility for systems reliability and software correctness, and to improve the scalability of both the soft‐ ware systems and the teams of people who build them.

**4 | Chapter 1: Revolution in the Cloud**

**Nobody Understands DevOps**

DevOps has occasionally been a controversial idea, both with people who insist it’s nothing more than a modern label for existing good practice in software develop‐ ment, and with those who reject the need for greater collaboration between develop‐ ment and operations.

There is also widespread misunderstanding about what DevOps actually is: A job title? A team? A methodology? A skill set? The influential DevOps writer John Willis has identified four key pillars of DevOps, which he calls culture, automation, meas‐ urement, and sharing (CAMS). Another way to break it down is what Brian Dawson has called the DevOps trinity: people and culture, process and practice, and tools and technology.

Some people think that cloud and containers mean that we no longer need DevOps— a point of view sometimes called *NoOps*. The idea is that since all IT operations are outsourced to a cloud provider or another third-party service, businesses don’t need full-time operations staff.

The NoOps fallacy is based on a misapprehension of what DevOps work actually involves:

With DevOps, much of the traditional IT operations work happens before code rea‐ ches production. Every release includes monitoring, logging, and A/B testing. CI/CD pipelines automatically run unit tests, security scanners, and policy checks on every commit. Deployments are automatic. Controls, tasks, and non-functional require‐ ments are now implemented before release instead of during the frenzy and aftermath of a critical outage.

—Jordan Bach (AppDynamics)

The most important thing to understand about DevOps is that it is primarily an organizational, human issue, not a technical one. This accords with Jerry Weinberg’s *Second Law of Consulting*:

No matter how it looks at first, it’s always a people problem.

—Gerald M. Weinberg, *Secrets of Consulting*

**The Business Advantage**

From a business point of view, DevOps has been described as “improving the quality of your software by speeding up release cycles with cloud automation and practices, with the added benefit of software that actually stays up in production” (The Regis‐ ter).

Adopting DevOps requires a profound cultural transformation for businesses, which needs to start at the executive, strategic level, and propagate gradually to every part of

**The Dawn of DevOps | 5**

the organization. Speed, agility, collaboration, automation, and software quality are key goals of DevOps, and for many companies that means a major shift in mindset.

But DevOps works, and studies regularly suggest that companies that adopt DevOps principles release better software faster, react better and faster to failures and prob‐ lems, are more agile in the marketplace, and dramatically improve the quality of their products:

DevOps is not a fad; rather it is the way successful organizations are industrializing the delivery of quality software today and will be the new baseline tomorrow and for years to come.

—Brian Dawson (Cloudbees), Computer Business Review

**Infrastructure as Code**

Once upon a time, developers dealt with software, while operations teams dealt with hardware and the operating systems that run on that hardware.

Now that hardware is in the cloud, everything, in a sense, is software. The DevOps movement brings software development skills to operations: tools and workflows for rapid, agile, collaborative building of complex systems. Inextricably entwined with DevOps is the notion of *infrastructure as code*.

Instead of physically racking and cabling computers and switches, cloud infrastruc‐ ture can be automatically provisioned by software. Instead of manually deploying and upgrading hardware, operations engineers have become the people who write the software that automates the cloud.

The traffic isn’t just one-way. Developers are learning from operations teams how to anticipate the failures and problems inherent in distributed, cloud-based systems, how to mitigate their consequences, and how to design software that degrades grace‐ fully and fails safe.

**Learning Together**

Both development teams and operations teams are also learning how to work together. They’re learning how to design and build systems, how to monitor and get feedback on systems in production, and how to use that information to improve the systems. Even more importantly, they’re learning to improve the experience for their users, and to deliver better value for the business that funds them.

The massive scale of the cloud and the collaborative, code-centric nature of the DevOps movement have turned operations into a software problem. At the same time, they have also turned software into an operations problem, all of which raises these questions:

**6 | Chapter 1: Revolution in the Cloud**

• How do you deploy and upgrade software across large, diverse networks of dif‐ ferent server architectures and operating systems?

• How do you deploy to distributed environments, in a reliable and reproducible way, using largely standardized components?

Enter the third revolution: the container. **The Coming of Containers**

To deploy a piece of software, you need not only the software itself, but its *dependen‐ cies*. That means libraries, interpreters, subpackages, compilers, extensions, and so on.

You also need its *configuration*. Settings, site-specific details, license keys, database passwords: everything that turns raw software into a usable service.

**The State of the Art**

Earlier attempts to solve this problem include using *configuration management* sys‐ tems, such as Puppet or Ansible, which consist of code to install, run, configure, and update the shipping software.

Alternatively, some languages provide their own packaging mechanism, like Java’s JAR files, or Python’s eggs, or Ruby’s gems. However, these are language-specific, and don’t entirely solve the dependency problem: you still need a Java runtime installed before you can run a JAR file, for example.

Another solution is the *omnibus package*, which, as the name suggests, attempts to cram everything the application needs inside a single file. An omnibus package con‐ tains the software, its configuration, its dependent software components, *their* config‐ uration, *their* dependencies, and so on. (For example, a Java omnibus package would contain the Java runtime as well as all the JAR files for the application.)

Some vendors have even gone a step further and included the entire computer system required to run it, as a *virtual machine image*, but these are large and unwieldy, time- consuming to build and maintain, fragile to operate, slow to download and deploy, and vastly inefficient in performance and resource footprint.

From an operations point of view, not only do you need to manage these various kinds of packages, but you also need to manage a fleet of servers to run them on.

Servers need to be provisioned, networked, deployed, configured, kept up to date with security patches, monitored, managed, and so on.

This all takes a significant amount of time, skill, and effort, just to provide a platform to run software on. Isn’t there a better way?

**The Coming of Containers | 7**

**Thinking Inside the Box**

To solve these problems, the tech industry borrowed an idea from the shipping industry: the *container*. In the 1950s, a truck driver named Malcolm McLean pro‐ posed that, instead of laboriously unloading goods individually from the truck trailers that brought them to the ports and loading them onto ships, trucks themselves simply be loaded onto the ship—or rather, the truck bodies.

A truck trailer is essentially a big metal box on wheels. If you can separate the box— the container—from the wheels and chassis used to transport it, you have something that is very easy to lift, load, stack, and unload, and can go right onto a ship or another truck at the other end of the voyage (Figure 1-3).

McLean’s container shipping firm, Sea-Land, became very successful by using this system to ship goods far more cheaply, and containers quickly caught on. Today, hun‐ dreds of millions of containers are shipped every year, carrying trillions of dollars worth of goods.

*Figure 1-3. Standardized containers dramatically cut the cost of shipping bulk goods (photo by Pixabay, licensed under Creative Commons 2.0)*

**Putting Software in Containers**

The software container is exactly the same idea: a standard packaging and distribu‐ tion format that is generic and widespread, enabling greatly increased carrying

**8 | Chapter 1: Revolution in the Cloud**

capacity, lower costs, economies of scale, and ease of handling. The container format contains everything the application needs to run, baked into an *image file* that can be executed by a *container runtime*.

How is this different from a virtual machine image? That, too, contains everything the application needs to run—but a lot more besides. A typical virtual machine image is around 1 GiB.1 A well-designed container image, on the other hand, might be a hundred times smaller.

Because the virtual machine contains lots of unrelated programs, libraries, and things that the application will never use, most of its space is wasted. Transferring VM images across the network is far slower than optimized containers.

Even worse, virtual machines are *virtual*: the underlying physical CPU effectively implements an *emulated* CPU, which the virtual machine runs on. The virtualization layer has a dramatic, negative effect on performance: in tests, virtualized workloads run about 30% slower than the equivalent containers.

In comparison, containers run directly on the real CPU, with no virtualization over‐ head, just as ordinary binary executables do.

And because containers only hold the files they need, they’re much smaller than VM images. They also use a clever technique of addressable filesystem *layers*, which can be shared and reused between containers.

For example, if you have two containers, each derived from the same Debian Linux base image, the base image only needs to be downloaded once, and each container can simply reference it.

The container runtime will assemble all the necessary layers and only download a layer if it’s not already cached locally. This makes very efficient use of disk space and network bandwidth.

**Plug and Play Applications**

Not only is the container the unit of deployment and the unit of packaging; it is also the unit of *reuse* (the same container image can be used as a component of many different services), the unit of *scaling*, and the unit of *resource allocation* (a container can run anywhere sufficient resources are available for its own specific needs).

Developers no longer have to worry about maintaining different versions of the soft‐ ware to run on different Linux distributions, against different library and language

1 The *gibibyte* (GiB) is the International Electrotechnical Commission (IEC) unit of data, defined as 1,024 *mebi‐*

*bytes* (MiB). We’ll use IEC units (GiB, MiB, KiB) throughout this book to avoid any ambiguity.

**The Coming of Containers | 9**

versions, and so on. The only thing the container depends on is the operating system kernel (Linux, for example).

Simply supply your application in a container image, and it will run on any platform that supports the standard container format and has a compatible kernel.

Kubernetes developers Brendan Burns and David Oppenheimer put it this way in their paper “Design Patterns for Container-based Distributed Systems”:

By being hermetically sealed, carrying their dependencies with them, and providing an atomic deployment signal (“succeeded”/“failed”), [containers] dramatically improve on the previous state of the art in deploying software in the datacenter or cloud. But con‐ tainers have the potential to be much more than just a better deployment vehicle—we believe they are destined to become analogous to objects in object-oriented software systems, and as such will enable the development of distributed system design patterns. **Conducting the Container Orchestra**

Operations teams, too, find their workload greatly simplified by containers. Instead of having to maintain a sprawling estate of machines of various kinds, architectures, and operating systems, all they have to do is run a *container orchestrator*: a piece of soft‐ ware designed to join together many different machines into a *cluster*: a kind of uni‐ fied compute substrate, which appears to the user as a single very powerful computer on which containers can run.

The terms *orchestration* and *scheduling* are often used loosely as synonyms. Strictly speaking, though, *orchestration* in this context means coordinating and sequencing different activities in service of a common goal (like the musicians in an orchestra). *Scheduling* means managing the resources available and assigning workloads where they can most efficiently be run. (Not to be confused with scheduling in the sense of *scheduled jobs*, which execute at preset times.)

A third important activity is *cluster management*: joining multiple physical or virtual servers into a unified, reliable, fault-tolerant, apparently seamless group.

The term *container orchestrator* usually refers to a single service that takes care of scheduling, orchestration, and cluster management.

*Containerization* (using containers as your standard method of deploying and run‐ ning software) offered obvious advantages, and a de facto standard container format has made possible all kinds of economies of scale. But one problem still stood in the way of the widespread adoption of containers: the lack of a standard container orchestration system.

As long as several different tools for scheduling and orchestrating containers compe‐ ted in the marketplace, businesses were reluctant to place expensive bets on which technology to use. But all that was about to change.

**10 | Chapter 1: Revolution in the Cloud**

**Kubernetes**

Google was running containers at scale for production workloads long before anyone else. Nearly all of Google’s services run in containers: Gmail, Google Search, Google Maps, Google App Engine, and so on. Because no suitable container orchestration system existed at the time, Google was compelled to invent one.

**From Borg to Kubernetes**

To solve the problem of running a large number of services at global scale on millions of servers, Google developed a private, internal container orchestration system it called Borg.

Borg is essentially a centralized management system that allocates and schedules con‐ tainers to run on a pool of servers. While very powerful, Borg is tightly coupled to Google’s own internal and proprietary technologies, difficult to extend, and impossi‐ ble to release to the public.

In 2014, Google founded an open source project named Kubernetes (from the Greek word κυβερνήτης, meaning “helmsman, pilot”) that would develop a container orchestrator that everyone could use, based on the lessons learned from Borg and its successor, Omega.

Kubernetes’s rise was meteoric. While other container orchestration systems existed before Kubernetes, they were commercial products tied to a vendor, and that was always a barrier to their widespread adoption. With the advent of a truly free and open source container orchestrator, adoption of both containers and Kubernetes grew at a phenomenal rate.

By late 2017, the orchestration wars were over, and Kubernetes had won. While other systems are still in use, from now on companies looking to move their infrastructure to containers only need to target one platform: Kubernetes.

**What Makes Kubernetes So Valuable?**

Kelsey Hightower, a staff developer advocate at Google, coauthor of *Kubernetes Up & Running* (O’Reilly), and all-around legend in the Kubernetes community, puts it this way:Kubernetes does the things that the very best system administrator would do: automation, failover, centralized logging, monitoring. It takes what we’ve learned in the DevOps community and makes it the default, out of the box.

—Kelsey Hightower

Many of the traditional sysadmin tasks like upgrading servers, installing security patches, configuring networks, and running backups are less of a concern in the

**Kubernetes | 11**

cloud native world. Kubernetes can automate these things for you so that your team can concentrate on doing its core work.

Some of these features, like load balancing and autoscaling, are built into the Kuber‐ netes core; others are provided by add-ons, extensions, and third-party tools that use the Kubernetes API. The Kubernetes ecosystem is large, and growing all the time.

**Kubernetes makes deployment easy** Ops staff love Kubernetes for these reasons, but there are also some significant advan‐ tages for developers. Kubernetes greatly reduces the time and effort it takes to deploy. Zero-downtime deployments are common, because Kubernetes does rolling updates by default (starting containers with the new version, waiting until they become healthy, and then shutting down the old ones).

Kubernetes also provides facilities to help you implement continuous deployment practices such as *canary deployments*: gradually rolling out updates one server at a time to catch problems early (see “Canary Deployments” on page 242). Another com‐ mon practice is *blue-green* deployments: spinning up a new version of the system in parallel, and switching traffic over to it once it’s fully up and running (see “Blue/ Green Deployments” on page 241).

Demand spikes will no longer take down your service, because Kubernetes supports autoscaling. For example, if CPU utilization by a container reaches a certain level, Kubernetes can keep adding new replicas of the container until the utilization falls below the threshold. When demand falls, Kubernetes will scale down the replicas again, freeing up cluster capacity to run other workloads.

Because Kubernetes has redundancy and failover built in, your application will be more reliable and resilient. Some managed services can even scale the Kubernetes cluster itself up and down in response to demand, so that you’re never paying for a larger cluster than you need at any given moment (see “Autoscaling” on page 102).

The business will love Kubernetes too, because it cuts infrastructure costs and makes much better use of a given set of resources. Traditional servers, even cloud servers, are mostly idle most of the time. The excess capacity that you need to handle demand spikes is essentially wasted under normal conditions.

Kubernetes takes that wasted capacity and uses it to run workloads, so you can ach‐ ieve much higher utilization of your machines—and you get scaling, load balancing, and failover for free too.

While some of these features, such as autoscaling, were available before Kubernetes, they were always tied to a particular cloud provider or service. Kubernetes is *provider- agnostic*: once you’ve defined the resources you use, you can run them on any Kuber‐ netes cluster, regardless of the underlying cloud provider.

**12 | Chapter 1: Revolution in the Cloud**

That doesn’t mean that Kubernetes limits you to the lowest common denominator. Kubernetes maps your resources to the appropriate vendor-specific features: for example, a load-balanced Kubernetes service on Google Cloud will create a Google Cloud load balancer, on Amazon it will create an AWS load balancer. Kubernetes abstracts away the cloud-specific details, letting you focus on defining the behavior of your application.

Just as containers are a portable way of defining software, Kubernetes resources pro‐ vide a portable definition of how that software should run.

**Will Kubernetes Disappear?**

Oddly enough, despite the current excitement around Kubernetes, we may not be talking much about it in years to come. Many things that once were new and revolu‐ tionary are now so much part of the fabric of computing that we don’t really think about them: microprocessors, the mouse, the internet.

Kubernetes, too, is likely to disappear and become part of the plumbing. It’s boring, in a good way: once you learn what you need to know to deploy your application to Kubernetes, you’re more or less done.

The future of Kubernetes is likely to lie largely in the realm of managed services. Vir‐ tualization, which was once an exciting new technology, has now simply become a utility. Most people rent virtual machines from a cloud provider rather than run their own virtualization platform, such as vSphere or Hyper-V.

In the same way, we think Kubernetes will become so much a standard part of the plumbing that you just won’t know it’s there anymore.

**Kubernetes Doesn’t Do It All**

Will the infrastructure of the future be entirely Kubernetes-based? Probably not. Firstly, some things just aren’t a good fit for Kubernetes (databases, for example):

Orchestrating software in containers involves spinning up new interchangeable instan‐ ces without requiring coordination between them. But database replicas are not inter‐ changeable; they each have a unique state, and deploying a database replica requires coordination with other nodes to ensure things like schema changes happen every‐ where at the same time:

—Sean Loiselle (Cockroach Labs)

While it’s perfectly possible to run stateful workloads like databases in Kubernetes with enterprise-grade reliability, it requires a large investment of time and engineer‐ ing that it may not make sense for your company to make (see “Run Less Software” on page 47). It’s usually more cost-effective to use managed services instead.

**Kubernetes | 13**

Secondly, some things don’t actually need Kubernetes, and can run on what are some‐ times called *serverless* platforms, better named *functions as a service*, or *FaaS* plat‐ forms.

**Cloud functions and funtainers** AWS Lambda, for example, is a FaaS platform that allows you to run code written in Go, Python, Java, Node.js, C#, and other languages, without you having to compile or deploy your application at all. Amazon does all that for you.

Because you’re billed for the execution time in increments of 100 milliseconds, the FaaS model is perfect for computations that only run when you need them to, instead of paying for a cloud server, which runs all the time whether you’re using it or not.

These *cloud functions* are more convenient than containers in some ways (though some FaaS platforms can run containers as well). But they are best suited to short, standalone jobs (AWS Lambda limits functions to fifteen minutes of run time, for example, and around 50 MiB of deployed files), especially those that integrate with existing cloud computation services, such as Microsoft Cognitive Services or the Google Cloud Vision API.

Why don’t we like to refer to this model as *serverless*? Well, it isn’t: it’s just somebody else’s server. The point is that you don’t have to provision and maintain that server; the cloud provider takes care of it for you.

Not every workload is suitable for running on FaaS platforms, by any means, but it is still likely to be a key technology for cloud native applications in the future.

Nor are cloud functions restricted to public FaaS platforms such as Lambda or Azure Functions: if you already have a Kubernetes cluster and want to run FaaS applications on it, OpenFaaS and other open source projects make this possible. This hybrid of functions and containers is sometimes called *funtainers*, a name we find appealing.

A more sophisticated software delivery platform for Kubernetes that encompasses both containers and cloud functions, called Knative, is currently under active devel‐ opment (see “Knative” on page 238). This is a very promising project, which may mean that in the future the distinction between containers and functions may blur or disappear altogether. **Cloud Native**

The term *cloud native* has become an increasingly popular shorthand way of talking about modern applications and services that take advantage of the cloud, containers, and orchestration, often based on open source software.

**14 | Chapter 1: Revolution in the Cloud**

Indeed, the Cloud Native Computing Foundation (CNCF) was founded in 2015 to, in their words, “foster a community around a constellation of high-quality projects that orchestrate containers as part of a microservices architecture.”

Part of the Linux Foundation, the CNCF exists to bring together developers, end- users, and vendors, including the major public cloud providers. The best-known project under the CNCF umbrella is Kubernetes itself, but the foundation also incu‐ bates and promotes other key components of the cloud native ecosystem: Prome‐ theus, Envoy, Helm, Fluentd, gRPC, and many more.

So what exactly do we mean by *cloud native*? Like most such things, it means different things to different people, but perhaps there is some common ground.

Cloud native applications run in the cloud; that’s not controversial. But just taking an existing application and running it on a cloud compute instance doesn’t make it cloud native. Neither is it just about running in a container, or using cloud services such as Azure’s Cosmos DB or Google’s Pub/Sub, although those may well be important aspects of a cloud native application.

So let’s look at a few of the characteristics of cloud native systems that most people can agree on:

*Automatable*

If applications are to be deployed and managed by machines, instead of humans, they need to abide by common standards, formats, and interfaces. Kubernetes provides these standard interfaces in a way that means application developers don’t even need to worry about them.

*Ubiquitous and flexible*

Because they are decoupled from physical resources such as disks, or any specific knowledge about the compute node they happen to be running on, containerized microservices can easily be moved from one node to another, or even one cluster to another.

*Resilient and scalable*

Traditional applications tend to have single points of failure: the application stops working if its main process crashes, or if the underlying machine has a hardware failure, or if a network resource becomes congested. Cloud native applications, because they are inherently distributed, can be made highly available through redundancy and graceful degradation.

*Dynamic*

A container orchestrator such as Kubernetes can schedule containers to take maximum advantage of available resources. It can run many copies of them to achieve high availability, and perform rolling updates to smoothly upgrade serv‐ ices without ever dropping traffic.

**Cloud Native | 15**

*Observable*

Cloud native apps, by their nature, are harder to inspect and debug. So a key requirement of distributed systems is *observability*: monitoring, logging, tracing, and metrics all help engineers understand what their systems are doing (and what they’re doing wrong).

*Distributed*

Cloud native is an approach to building and running applications that takes advantage of the distributed and decentralized nature of the cloud. It’s about how your application works, not where it runs. Instead of deploying your code as a single entity (known as a *monolith*), cloud native applications tend to be com‐ posed of multiple, cooperating, distributed *microservices*. A microservice is sim‐ ply a self-contained service that does one thing. If you put enough microservices together, you get an application.

**It’s not just about microservices** However, microservices are not a panacea. Monoliths are easier to understand, because everything is in one place, and you can trace the interactions of different parts. But it’s hard to scale monoliths, both in terms of the code itself, and the teams of developers who maintain it. As the code grows, the interactions between its various parts grow exponentially, and the system as a whole grows beyond the capacity of a single brain to understand it all.

A well-designed cloud native application is composed of microservices, but deciding what those microservices should be, where the boundaries are, and how the different services should interact is no easy problem. Good cloud native service design consists of making wise choices about how to separate the different parts of your architecture. However, even a well-designed cloud native application is still a distributed system, which makes it inherently complex, difficult to observe and reason about, and prone to failure in surprising ways.

While cloud native systems tend to be distributed, it’s still possible to run monolithic applications in the cloud, using containers, and gain considerable business value from doing so. This may be a step on the road to gradually migrating parts of the monolith outward to modern microservices, or a stopgap measure pending the redesign of the system to be fully cloud native. **The Future of Operations**

Operations, infrastructure engineering, and system administration are highly skilled jobs. Are they at risk in a cloud native future? We think not.

Instead, these skills will only become more important. Designing and reasoning about distributed systems is hard. Networks and container orchestrators are compli‐

**16 | Chapter 1: Revolution in the Cloud**

cated. Every team developing cloud native applications will need operations skills and knowledge. Automation frees up staff from boring, repetitive manual work to deal with more complex, interesting, and fun problems that computers can’t yet solve for themselves.

That doesn’t mean all current operations jobs are guaranteed. Sysadmins used to be able to get by without coding skills, except maybe cooking up the odd simple shell script. In the cloud, that won’t fly.

In a software-defined world, the ability to write, understand, and maintain software becomes critical. If you can’t or won’t learn new skills, the world will leave you behind —and it’s always been that way.

**Distributed DevOps**

Rather than being concentrated in a single operations team that services other teams, ops expertise will become distributed among many teams.

Each development team will need at least one ops specialist, responsible for the health of the systems or services the team provides. She will be a developer, too, but she will also be the domain expert on networking, Kubernetes, performance, resilience, and the tools and systems that enable the other developers to deliver their code to the cloud.

Thanks to the DevOps revolution, there will no longer be room in most organizations for devs who can’t ops, or ops who don’t dev. The distinction between those two disci‐ plines is obsolete, and is rapidly being erased altogether. Developing and operating software are merely two aspects of the same thing.

**Some Things Will Remain Centralized**

Are there limits to DevOps? Or will the traditional central IT and operations team disappear altogether, dissolving into a group of roving internal consultants, coaching, teaching, and troubleshooting ops issues?

We think not, or at least not entirely. Some things still benefit from being centralized. It doesn’t make sense for each application or service team to have its own way of detecting and communicating about production incidents, for example, or its own ticketing system, or deployment tools. There’s no point in everybody reinventing their own wheel.

**Developer Productivity Engineering**

The point is that self-service has its limits, and the aim of DevOps is to speed up development teams, not slow them down with unnecessary and redundant work.

**The Future of Operations | 17**

Yes, a large part of traditional operations can and should be devolved to other teams, primarily those that involve code deployment and responding to code-related inci‐ dents. But to enable that to happen, there needs to be a strong central team building and supporting the DevOps ecosystem in which all the other teams operate.

Instead of calling this team *operations*, we like the name *developer productivity engi‐ neering* (DPE). DPE teams do whatever’s necessary to help developers do their work better and faster: operating infrastructure, building tools, busting problems.

And while developer productivity engineering remains a specialist skill set, the engi‐ neers themselves may move outward into the organization to bring that expertise where it’s needed.

Lyft engineer Matt Klein has suggested that, while a pure DevOps model makes sense for startups and small firms, as an organization grows, there is a natural tendency for infrastructure and reliability experts to gravitate toward a central team. But he says that team can’t be scaled indefinitely:

By the time an engineering organization reaches ~75 people, there is almost certainly a central infrastructure team in place starting to build common substrate features required by product teams building microservices. But there comes a point at which the central infrastructure team can no longer both continue to build and operate the infrastructure critical to business success, while also maintaining the support burden of helping product teams with operational tasks.

—Matt Klein

At this point, not every developer can be an infrastructure expert, just as a single team of infrastructure experts can’t service an ever-growing number of developers. For larger organizations, while a central infrastructure team is still needed, there’s also a case for embedding *site reliability engineers* (SREs) into each development or prod‐ uct team. They bring their expertise to each team as consultants, and also form a bridge between product development and infrastructure operations.

**You Are the Future**

If you’re reading this book, it means you’re going to be part of this new cloud native future. In the remaining chapters, we’ll cover all the knowledge and skills you’ll need as a developer or operations engineer working with cloud infrastructure, containers, and Kubernetes.

Some of these things will be familiar, and some will be new, but we hope that when you’ve finished the book you’ll feel more confident in your own ability to acquire and master cloud native skills. Yes, there’s a lot to learn, but it’s nothing you can’t handle. You’ve got this!

Now read on.

**18 | Chapter 1: Revolution in the Cloud**

**Summary**

We’ve necessarily given you a rather quick tour of the cloud native DevOps landscape, but we hope it’s enough to bring you up to speed with some of the problems that cloud, containers, and Kubernetes solve, and how they’re likely to change the IT busi‐ ness. If you’re already familiar with this, then we appreciate your patience.

A quick recap of the main points before we move on to meet Kubernetes in person in the next chapter:

• Cloud computing frees you from the expense and overhead of managing your own hardware, making it possible for you to build resilient, flexible, scalable dis‐ tributed systems.

• DevOps is a recognition that modern software development doesn’t stop at ship‐ ping code: it’s about closing the feedback loop between those who write the code and those who use it.

• DevOps also brings a code-centric approach and good software engineering practices to the world of infrastructure and operations.

• Containers allow you to deploy and run software in small, standardized, self- contained units. This makes it easier and cheaper to build large, diverse, dis‐ tributed systems, by connecting together containerized microservices.

• Orchestration systems take care of deploying your containers, scheduling, scal‐ ing, networking, and all the things that a good system administrator would do, but in an automated, programmable way.

• Kubernetes is the de facto standard container orchestration system, and it’s ready for you to use in production right now, today.

• *Cloud native* is a useful shorthand for talking about cloud-based, containerized, distributed systems, made up of cooperating microservices, dynamically man‐ aged by automated infrastructure as code.

• Operations and infrastructure skills, far from being made obsolete by the cloud native revolution, are and will become more important than ever.

• It still makes sense for a central team to build and maintain the platforms and tools that make DevOps possible for all the other teams.

• What will go away is the sharp distinction between software engineers and opera‐ tions engineers. It’s all just software now, and we’re all engineers.

**Summary | 19**

**CHAPTER 2 First Steps with Kubernetes**

To do anything truly worth doing, I must not stand back shivering and thinking of the cold and danger, but jump in with gusto and scramble through as well as I can.—Og Mandino

Enough with the theory; let’s start working with Kubernetes and containers. In this chapter, you’ll build a simple containerized application and deploy it to a local Kuber‐ netes cluster running on your machine. In the process, you’ll meet some very impor‐ tant cloud native technologies and concepts: Docker, Git, Go, container registries, and the kubectl tool.

This chapter is interactive! Often, throughout this book, we’ll ask you to follow along with the examples by installing things on your own computer, typing commands, and running containers. We find that’s a much more effective way to learn than just having things explained in words.

**Running Your First Container**

As we saw in Chapter 1, the container is one of the key concepts in cloud native development. The fundamental tool for building and running containers is Docker. In this section, we’ll use the Docker Desktop tool to build a simple demo application, run it locally, and push the image to a container registry.

If you’re already very familiar with containers, skip straight to “Hello, Kubernetes” on page 29, where the real fun starts. If you’re curious to know what containers are and how they work, and to get a little practical experience with them before you start learning about Kubernetes, read on.

**21**

**Installing Docker Desktop**

Docker Desktop is a complete Kubernetes development environment for Mac or Windows that runs on your laptop (or desktop). It includes a single-node Kubernetes cluster that you can use to test your applications.

Let’s install Docker Desktop now and use it to run a simple containerized application. If you already have Docker installed, skip this section and go straight on to “Running a Container Image” on page 22.

Download a version of the Docker Desktop Community Edition suitable for your computer, then follow the instructions for your platform to install Docker and start it up.

Docker Desktop isn’t currently available for Linux, so Linux users will need to install Docker Engine instead, and then Minikube (see “Minikube” on page 31).

Once you’ve done that, you should be able to open a terminal and run the following command:

**docker version** Client: Version: 18.03.1-ce

... The exact output will be different depending on your platform, but if Docker is cor‐ rectly installed and running, you’ll see something like the example output shown. On Linux systems, you may need to run sudo docker version instead.

**What Is Docker?**

Docker is actually several different, but related things: a container image format, a *container runtime* library, which manages the life cycle of containers, a command-line tool for packaging and running containers, and an API for container management. The details needn’t concern us here, since Kubernetes uses Docker as one of many components, though an important one.

**Running a Container Image**

What exactly is a container image? The technical details don’t really matter for our purposes, but you can think of an image as being like a ZIP file. It’s a single binary file that has a unique ID and holds everything needed to run the container.

**22 | Chapter 2: First Steps with Kubernetes**

Whether you’re running the container directly with Docker, or on a Kubernetes clus‐ ter, all you need to specify is a container image ID or URL, and the system will take care of finding, downloading, unpacking, and starting the container for you.

We’ve written a little demo application that we’ll use throughout the book to illustrate what we’re talking about. You can download and run the application using a con‐ tainer image we prepared earlier. Run the following command to try it out:

**docker container run -p 9999:8888 --name hello cloudnatived/demo:hello** Leave this command running, and point your browser to *http://localhost:9999/*.

You should see a friendly message:

Hello, 世界 Any time you make a request to this URL, our demo application will be ready and waiting to greet you.

Once you’ve had as much fun as you can stand, stop the container by pressing Ctrl-C in your terminal. **The Demo Application**

So how does it work? Let’s download the source code for the demo application that runs in this container and have a look.

You’ll need Git installed for this part.1 If you’re not sure whether you already have Git, try the following command:

**git version** git version 2.17.0 If you don’t already have Git, follow the installation instructions for your platform.

Once you’ve installed Git, run this command:

**git clone https://github.com/cloudnativedevops/demo.git** Cloning into *demo*... ... **Looking at the Source Code**

This Git repository contains the demo application we’ll be using throughout this book. To make it easier to see what’s going on at each stage, the repo contains each successive version of the app in a different subdirectory. The first one is named sim‐ ply *hello*. To look at the source code, run this command:

1 If you’re not familiar with Git, read Scott Chacon and Ben Straub’s excellent book *Pro Git* (Apress).

**The Demo Application | 23**

**cd demo/hello ls**Dockerfile README.md go.mod main.go Open the file *main.go* in your favorite editor (we recommend Visual Studio Code which has excellent support for Go, Docker, and Kubernetes development). You’ll see this source code:

**package** main

**import** (

"fmt" "log" "net/http" )**func** handler(w http.ResponseWriter, r \*http.Request) {

fmt.Fprintln(w, "Hello, 世界") }**func** main() {

http.HandleFunc("/", handler) log.Fatal(http.ListenAndServe(":8888", **nil**)) } **Introducing Go**

Our demo application is written in the Go programming language.

Go is a modern programming language (developed at Google since 2009) that priori‐ tizes simplicity, safety, and readability, and is designed for building large-scale con‐ current applications, especially network services. It’s also a lot of fun to program in.2

Kubernetes itself is written in Go, as are Docker, Terraform, and many other popular open source projects. This makes Go a good choice for developing cloud native appli‐ cations.

**How the Demo App Works**

As you can see, the demo app is pretty simple, even though it implements an HTTP server (Go comes with a powerful standard library). The core of it is this function, called handler:

2 If you’re a fairly experienced programmer, but new to Go, Alan Donovan and Brian Kernighan’s *The Go Pro‐*

*gramming Language* (Addison-Wesley) is an invaluable guide.

**24 | Chapter 2: First Steps with Kubernetes**

**func** handler(w http.ResponseWriter, r \*http.Request) {

fmt.Fprintln(w, "Hello, 世界") } As the name suggests, it handles HTTP requests. The request is passed in as an argu‐ ment to the function (though the function doesn’t do anything with it, yet).

An HTTP server also needs a way to send something back to the client. The http.ResponseWriter object enables our function to send a message back to the user to display in her browser: in this case, just the string Hello, 世界.

The first example program in any language traditionally prints Hello, world. But because Go natively supports Unicode (the international standard for text representa‐ tion), example Go programs often print Hello, 世界 instead, just to show off. If you don’t happen to speak Chinese, that’s okay: Go does!

The rest of the program takes care of registering the handler function as the handler for HTTP requests, and actually starting the HTTP server to listen and serve on port 8888.

That’s the whole app! It doesn’t do much yet, but we will add capabilities to it as we go on.**Building a Container**

You know that a container image is a single file that contains everything the container needs to run, but how do you build an image in the first place? Well, to do that, you use the docker image build command, which takes as input a special text file called a *Dockerfile*. The Dockerfile specifies exactly what needs to go into the container image.

One of the key benefits of containers is the ability to build on existing images to cre‐ ate new images. For example, you could take a container image containing the com‐ plete Ubuntu operating system, add a single file to it, and the result will be a new image.

In general, a Dockerfile has instructions for taking a starting image (a so-called *base image*), transforming it in some way, and saving the result as a new image.

**Understanding Dockerfiles**

Let’s see the Dockerfile for our demo application (it’s in the *hello* subdirectory of the app repo):

**FROM** golang:1.11-alpine AS build

**WORKDIR** /src/ COPY main.go go.\* /src/

**Building a Container | 25**

**RUN** CGO\_ENABLED=0 go build -o /bin/demo

**FROM** scratch COPY --from=build /bin/demo /bin/demo **ENTRYPOINT** ["/bin/demo"] The exact details of how this works don’t matter for now, but it uses a fairly standard build process for Go containers called *multi-stage builds*. The first stage starts from an official golang container image, which is just an operating system (in this case Alpine Linux) with the Go language environment installed. It runs the go build command to compile the *main.go* file we saw earlier.

The result of this is an executable binary file named *demo*. The second stage takes a completely empty container image (called a *scratch* image, as in *from scratch*) and copies the *demo* binary into it.

**Minimal Container Images**

Why the second build stage? Well, the Go language environment, and the rest of Alpine Linux, is really only needed in order to *build* the program. To run the pro‐ gram, all it takes is the *demo* binary, so the Dockerfile creates a new scratch container to put it in. The resulting image is very small (about 6 MiB)—and that’s the image that can be deployed in production.

Without the second stage, you would have ended up with a container image about 350 MiB in size, 98% of which is unnecessary and will never be executed. The smaller the container image, the faster it can be uploaded and downloaded, and the faster it will be to start up.

Minimal containers also have a reduced *attack surface* for security issues. The fewer programs there are in your container, the fewer potential vulnerabilities.

Because Go is a compiled language that can produce self-contained executables, it’s ideal for writing minimal (*scratch*) containers. By comparison, the official Ruby con‐ tainer image is 1.5 GiB; about 250 times bigger than our Go image, and that’s before you’ve added your Ruby program!

**Running docker image build**

We’ve seen that the Dockerfile contains instructions for the docker image build tool to turn our Go source code into an executable container. Let’s go ahead and try it. In the *hello* directory, run the following command:

**docker image build -t myhello .** Sending build context to Docker daemon 4.096kB Step 1/7 : FROM golang:1.11-alpine AS build ...

**26 | Chapter 2: First Steps with Kubernetes**

Successfully built eeb7d1c2e2b7 Successfully tagged myhello:latest Congratulations, you just built your first container! You can see from the output that Docker performs each of the actions in the Dockerfile in sequence on the newly formed container, resulting in an image that’s ready to use.

**Naming Your Images**

When you build an image, by default it just gets a hexadecimal ID, which you can use to refer to it later (for example, to run it). These IDs aren’t particularly memorable or easy to type, so Docker allows you to give the image a human-readable name, using the -t switch to docker image build. In the previous example you named the image myhello, so you should be able to use that name to run the image now.

Let’s see if it works:

**docker container run -p 9999:8888 myhello** You’re now running your own copy of the demo application, and you can check it by browsing to the same URL as before (*http://localhost:9999/*).

You should see Hello, 世界. When you’re done running this image, press Ctrl-C to stop the docker container run command.

**Exercise** If you’re feeling adventurous, modify the *main.go* file in the demo application and change the greeting so that it says “Hello, world” in your favorite language (or change it to say whatever you like). Rebuild the container and run it to check that it works.

Congratulations, you’re now a Go programmer! But don’t stop there: take the interac‐ tive Tour of Go to learn more.

**Port Forwarding**

Programs running in a container are isolated from other programs running on the same machine, which means they can’t have direct access to resources like network ports.

The demo application listens for connections on port 8888, but this is the *container’s* own private port 8888, not a port on your computer. In order to connect to the con‐ tainer’s port 8888, you need to *forward* a port on your local machine to that port on the container. It could be any port, including 8888, but we’ll use 9999 instead, to make it clear which is your port, and which is the container’s.

**Building a Container | 27**

To tell Docker to forward a port, you can use the -p switch, just as you did earlier in “Running a Container Image” on page 22:

**docker container run -p HOST\_PORT:CONTAINER\_PORT ...**

Once the container is running, any requests to HOST\_PORT on the local computer will be forwarded automatically to CONTAINER\_PORT on the container, which is how you’re able to connect to the app with your browser. **Container Registries**

In “Running a Container Image” on page 22, you were able to run an image just by giving its name, and Docker downloaded it for you automatically.

You might reasonably wonder where it’s downloaded from. While you can use Docker perfectly well by just building and running local images, it’s much more use‐ ful if you can push and pull images from a *container registry*. The registry allows you to store images and retrieve them using a unique name (like cloudnatived/ demo:hello).

The default registry for the docker container run command is Docker Hub, but you can specify a different one, or set up your own.

For now, let’s stick with Docker Hub. While you can download and use any public container image from Docker Hub, to push your own images you’ll need an account (called a *Docker ID*). Follow the instructions at *https://hub.docker.com/* to create your Docker ID.

**Authenticating to the Registry**

Once you’ve got your Docker ID, the next step is to connect your local Docker dae‐ mon with Docker Hub, using your ID and password:

**docker login** Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID, head over to https://hub.docker.com to create one. Username: ***YOUR\_DOCKER\_ID*** Password: ***YOUR\_DOCKER\_PASSWORD*** Login Succeeded **Naming and Pushing Your Image**

In order to be able to push a local image to the registry, you need to name it using this format: YOUR\_DOCKER\_ID/myhello.

To create this name, you don’t need to rebuild the image; instead, run this command:

**docker image tag myhello *YOUR\_DOCKER\_ID*/myhello**

**28 | Chapter 2: First Steps with Kubernetes**

This is so that when you push the image to the registry, Docker knows which account to store it in.

Go ahead and push the image to Docker Hub, using this command:

**docker image push *YOUR\_DOCKER\_ID*/myhello** The push refers to repository [docker.io/YOUR\_DOCKER\_ID/myhello] b2c591f16c33: Pushed latest: digest:

sha256:7ac57776e2df70d62d7285124fbff039c9152d1bdfb36c75b5933057cefe4fc7 size: 528 **Running Your Image**

Congratulations! Your container image is now available to run anywhere (at least, anywhere with access to the internet), using the command:

**docker container run -p 9999:8888 *YOUR\_DOCKER\_ID*/myhello Hello, Kubernetes**

Now that you’ve built and pushed your first container image, you can run it using the docker container run command, but that’s not very exciting. Let’s do something a little more adventurous and run it in Kubernetes.

There are lots of ways to get a Kubernetes cluster, and we’ll explore some of them in more detail in Chapter 3. If you already have access to a Kubernetes cluster, that’s great, and if you like you can use it for the rest of the examples in this chapter.

If not, don’t worry. Docker Desktop includes Kubernetes support (Linux users, see “Minikube” on page 31 instead). To enable it, open the Docker Desktop Preferences, select the Kubernetes tab, and check Enable (see Figure 2-1).

*Figure 2-1. Enabling Kubernetes support in Docker Desktop*

**Hello, Kubernetes | 29**

It will take a few minutes to install and start Kubernetes. Once that’s done, you’re ready to run the demo app!

**Running the Demo App**

Let’s start by running the demo image you built earlier. Open a terminal and run the kubectl command with the following arguments:

**kubectl run demo --image=*YOUR\_DOCKER\_ID*/myhello --port=9999 --labels app=demo** deployment.apps "demo" created Don’t worry about the details of this command for now: it’s basically the Kubernetes equivalent of the docker container run command you used earlier in this chapter to run the demo image. If you haven’t built your own image yet, you can use ours: --image=cloudnatived/demo:hello.

Recall that you needed to forward port 9999 on your local machine to the container’s port 8888 in order to connect to it with your web browser. You’ll need to do the same thing here, using kubectl port-forward:

**kubectl port-forward deploy/demo 9999:8888** Forwarding from 127.0.0.1:9999 -> 8888 Forwarding from [::1]:9999 -> 8888 Leave this command running and open a new terminal to carry on.

Connect to *http://localhost:9999/* with your browser to see the Hello, 世界 message.

It may take a few seconds for the container to start and for the app to be available. If it isn’t ready after half a minute or so, try this command:

**kubectl get pods --selector app=demo** NAME READY STATUS RESTARTS AGE demo-54df94b7b7-qgtc6 1/1 Running 0 9m When the container is running and you connect to it with your browser, you’ll see this message in the terminal:

Handling connection for 9999 **If the Container Doesn’t Start**

If the STATUS is not shown as Running, there may be a problem. For example, if the status is ErrImagePull or ImagePullBackoff, it means Kubernetes wasn’t able to find and download the image you specified. You may have made a typo in the image name; check your kubectl run command.

If the status is ContainerCreating, then all is well; Kubernetes is still downloading and starting the image. Just wait a few seconds and check again.

**30 | Chapter 2: First Steps with Kubernetes**

**Minikube**

If you don’t want to use, or can’t use, the Kubernetes support in Docker Desktop, there is an alternative: the well-loved Minikube. Like Docker Desktop, Minikube pro‐ vides a single-node Kubernetes cluster that runs on your own machine (in fact, in a virtual machine, but that doesn’t matter).

To install Minikube, follow these Minikube installation instructions. **Summary**

If, like us, you quickly grow impatient with wordy essays about why Kubernetes is so great, we hope you enjoyed getting to grips with some practical tasks in this chapter. If you’re an experienced Docker or Kubernetes user already, perhaps you’ll forgive the refresher course. We want to make sure that everybody feels quite comfortable with building and running containers in a basic way, and that you have a Kubernetes envi‐ ronment you can play and experiment with, before getting on to more advanced things.

Here’s what you should take away from this chapter:

• All the source code examples (and many more) are available in the demo reposi‐ tory that accompanies this book.

• The Docker tool lets you build containers locally, push them to or pull them from a container registry such as Docker Hub, and run container images locally on your machine.

• A container image is completely specified by a Dockerfile: a text file that contains instructions about how to build the container.

• Docker Desktop lets you run a small (single-node) Kubernetes cluster on your machine, which is nonetheless capable of running any containerized application. Minikube is another option.

• The kubectl tool is the primary way of interacting with a Kubernetes cluster, and can be used either *imperatively* (to run a public container image, for example, and implicitly creating the necessary Kubernetes resources), or *declaratively*, to apply Kubernetes configuration in the form of YAML manifests.

**Minikube | 31**

**CHAPTER 3 Getting Kubernetes**

Perplexity is the beginning of knowledge.

—Kahlil Gibran

Kubernetes is the operating system of the cloud native world, providing a reliable and scalable platform for running containerized workloads. But how should you run Kubernetes? Should you host it yourself? On cloud instances? On bare-metal servers? Or should you use a managed Kubernetes service? Or a managed platform that’s based on Kubernetes, but extends it with workflow tools, dashboards, and web inter‐ faces?

That’s a lot of questions for one chapter to answer, but we’ll try.

It’s worth noting that we won’t be particularly concerned here with the technical details of operating Kubernetes itself, such as building, tuning, and troubleshooting clusters. There are many excellent resources to help you with that, of which we partic‐ ularly recommend Kubernetes cofounder Brendan Burns’s book *Managing Kuber‐ netes: Operating Kubernetes Clusters in the Real World* (O’Reilly).

Instead, we’ll focus on helping you understand the basic architecture of a cluster, and give you the information you need to decide how to run Kubernetes. We’ll outline the pros and cons of managed services, and look at some of the popular vendors.

If you want to run your own Kubernetes cluster, we list some of the best installation tools available to help you set up and manage clusters. **Cluster Architecture**

You know that Kubernetes connects multiple servers into a *cluster*, but what is a clus‐ ter, and how does it work? The technical details don’t matter for the purposes of this book, but you should understand the basic components of Kubernetes and how they**33**

fit together, in order to understand what your options are when it comes to building or buying Kubernetes clusters.

**The Control Plane**

The cluster’s brain is called the *control plane*, and it runs all the tasks required for Kubernetes to do its job: scheduling containers, managing Services, serving API requests, and so on (see Figure 3-1).

*Figure 3-1. How a Kubernetes cluster works*

The control plane is actually made up of several components:

kube-apiserver

This is the frontend server for the control plane, handling API requests.

etcdThis is the database where Kubernetes stores all its information: what nodes exist,

what resources exist on the cluster, and so on.

kube-scheduler

This decides where to run newly created Pods.

kube-controller-manager

This is responsible for running resource controllers, such as Deployments.

cloud-controller-manager

This interacts with the cloud provider (in cloud-based clusters), managing resources such as load balancers and disk volumes.

The members of the cluster which run the control plane components are called *mas‐ ter nodes*.

**34 | Chapter 3: Getting Kubernetes**

**Node Components**

Cluster members that run user workloads are called *worker nodes* (Figure 3-2).

Each worker node in a Kubernetes cluster runs these components:

kubelet

This is responsible for driving the container runtime to start workloads that are scheduled on the node, and monitoring their status.

kube-proxy

This does the networking magic that routes requests between Pods on different nodes, and between Pods and the internet.

*Container runtime*

This actually starts and stops containers and handles their communications. Usu‐ ally Docker, but Kubernetes supports other container runtimes, such as rkt and CRI-O.

Other than running different software components, there’s no intrinsic difference between master nodes and worker nodes. Master nodes don’t usually run user work‐ loads, though, except in very small clusters (like Docker Desktop or Minikube).

*Figure 3-2. How the Kubernetes components fit together*

**High Availability**

A correctly configured Kubernetes control plane has multiple master nodes, making it *highly available*; that is, if any individual master node fails or is shut down, or one of the control plane components on it stops running, the cluster will still work properly. A highly available control plane will also handle the situation where the master nodes

**Cluster Architecture | 35**

are working properly, but some of them cannot communicate with the others, due to a network failure (known as a *network partition*).

The etcd database is replicated across multiple nodes, and can survive the failure of individual nodes, so long as a quorum of over half the original number of etcd repli‐ cas is still available.

If all of this is configured correctly, the control plane can survive a reboot or tempo‐ rary failure of individual master nodes.

**Control plane failure** A damaged control plane doesn’t necessarily mean that your applications will go down, although it might well cause strange and errant behavior.

For example, if you were to stop all the master nodes in your cluster, the Pods on the worker nodes would keep on running—at least for a while. But you would be unable to deploy any new containers or change any Kubernetes resources, and controllers such as Deployments would stop working.

Therefore, high availability of the control plane is critical to a properly functioning cluster. You need to have enough master nodes available that the cluster can maintain a *quorum* even if one fails; for production clusters, the workable minimum is three (see “The smallest cluster” on page 96).

**Worker node failure** By contrast, the failure of any worker node doesn’t really matter. Kubernetes will detect the failure and reschedule the node’s Pods somewhere else, so long as the con‐ trol plane is still working.

If a large number of nodes fail at once, this might mean that the cluster no longer has enough resources to run all the workloads you need. Fortunately, this doesn’t happen often, and even if it does, Kubernetes will keep as many of your Pods running as it can while you replace the missing nodes.

It’s worth bearing in mind, though, that the fewer worker nodes you have, the greater the proportion of the cluster’s capacity each one represents. You should assume that a single node failure will happen at any time, especially in the cloud, and two simulta‐ neous failures are not unheard of.

A rare, but entirely possible, kind of failure is losing a whole cloud *availability zone*. Cloud vendors like AWS and Google Cloud provide multiple availability zones in each region, each corresponding roughly to a single data center. For this reason, rather than having all your worker nodes in the same zone, it’s a good idea to distrib‐ ute them across two or even three zones.

**36 | Chapter 3: Getting Kubernetes**

**Trust, but verify** Although high availability should enable your cluster to survive losing one master, or a few worker nodes, it’s always wise to *actually test* this. During a scheduled mainte‐ nance window, or outside of peak hours, try rebooting a worker and see what hap‐ pens. (Hopefully, nothing, or nothing that’s visible to users of your applications.)

For a more demanding test, reboot one of the master nodes. (Managed services such as Google Kubernetes Engine, which we’ll discuss later in the chapter, don’t allow you to do this, for obvious reasons.) Still, a production-grade cluster should survive this with no problems whatsoever. **The Costs of Self-Hosting Kubernetes**

The most important decision facing anyone who’s considering running production workloads in Kubernetes is *buy or build?* Should you run your own clusters, or pay someone else to run them? Let’s look at some of the options.

The most basic choice of all is self-hosted Kubernetes. By *self-hosted* we mean that you, personally, or a team in your organization, install and configure Kubernetes, on machines that you own or control, just as you might do with any other software that you use, such as Redis, PostgreSQL, or Nginx.

This is the option that gives you the maximum flexibility and control. You can decide what versions of Kubernetes to run, what options and features are enabled, when and whether to upgrade clusters, and so on. But there are some significant downsides, as we’ll see in the next section.

**It’s More Work Than You Think**

The self-hosted option also requires the maximum resources, in terms of people, skills, engineering time, maintenance, and troubleshooting. Just setting up a working Kubernetes cluster is pretty simple, but that’s a long way from a cluster that’s ready for production. You need to consider at least the following questions:

• Is the control plane highly available? That is, if a master node goes down or becomes unresponsive, does your cluster still work? Can you still deploy or update apps? Will your running applications still be fault-tolerant without the control plane? (See “High Availability” on page 35.)

• Is your pool of worker nodes highly available? That is, if an outage should take down several worker nodes, or even a whole cloud availability zone, will your workloads stop running? Will your cluster keep working? Will it be able to auto‐ matically provision new nodes to heal itself, or will it require manual interven‐ tion?

**The Costs of Self-Hosting Kubernetes | 37**

• Is your cluster set up *securely*? Do its internal components communicate using TLS encryption and trusted certificates? Do users and applications have minimal rights and permissions for cluster operations? Are container security defaults set properly? Do nodes have unnecessary access to control plane components? Is access to the underlying etcd database properly controlled and authenticated?

• Are all services in your cluster secure? If they’re accessible from the internet, are they properly authenticated and authorized? Is access to the cluster API strictly limited?

• Is your cluster *conformant*? Does it meet the standards for Kubernetes clusters defined by the Cloud Native Computing Foundation? (See “Conformance Check‐ ing” on page 102 for details.)

• Are your cluster nodes fully *config-managed*, rather than being set up by impera‐ tive shell scripts and then left alone? The operating system and kernel on each node needs to be updated, have security patches applied, and so on.

• Is the data in your cluster properly backed up, including any persistent storage? What is your restore process? How often do you test restores?

• Once you have a working cluster, how do you maintain it over time? How do you provision new nodes? Roll out config changes to existing nodes? Roll out Kuber‐ netes updates? Scale in response to demand? Enforce policies?

Distributed systems engineer and writer Cindy Sridharan has estimated that it takes around a million dollars in engineer salary to get Kubernetes up and running in a production configuration from scratch (“And you still might not get there”). That fig‐ ure should give any technical leader food for thought when considering self-hosted Kubernetes.

**It’s Not Just About the Initial Setup**

Now bear in mind that you need to pay attention to these factors not just when set‐ ting up the first cluster for the first time, but for all your clusters for all time. When you make changes or upgrades to your Kubernetes infrastructure, you need to con‐ sider the impact on high availability, security, and so on.

You’ll need to have monitoring in place to make sure the cluster nodes and all the Kubernetes components are working properly, and an alerting system so that staff can be paged to deal with any problems, day or night.

Kubernetes is still in rapid development, and new features and updates are being released all the time. You’ll need to keep your cluster up to date with those, and understand how the changes affect your existing setup. You may need to reprovision your cluster to get the full benefit of the latest Kubernetes functionality.

**38 | Chapter 3: Getting Kubernetes**

It’s also not enough to read a few books or articles, configure the cluster the right way, and leave it at that. You need to test and verify the configuration on a regular basis— by killing a master node and making sure everything still works, for example.

Automated resilience testing tools such as Netflix’s Chaos Monkey can help with this, by randomly killing nodes, Pods, or network connections every so often. Depending on the reliability of your cloud provider, you may find that Chaos Monkey is unnec‐ essary, as regular real-world failures will also test the resilience of your cluster and the services running on it (see “Chaos Testing” on page 107).

**Tools Don’t Do All the Work for You**

There are tools—lots and lots of tools—to help you set up and configure Kubernetes clusters, and many of them advertise themselves as being more or less point-and- click, zero-effort, instant solutions. The sad fact is that in our opinion, the large majority of these tools solve only the easy problems, and ignore the hard ones.

On the other hand, powerful, flexible, enterprise-grade commercial tools tend to be very expensive, or not even available to the public, since there’s more money to be made selling a managed service than there is selling a general-purpose cluster man‐ agement tool.

**Kubernetes Is Hard**

Despite the widespread notion that it’s simple to set up and manage, the truth is that *Kubernetes is hard*. Considering what it does, it’s remarkably simple and well- designed, but it has to deal with very complex situations, and that leads to complex software.

Make no mistake, there is a significant investment of time and energy involved in both learning how to manage your own clusters properly, and actually doing it from day to day, month to month. We don’t want to discourage you from using Kubernetes, but we want you to have a clear understanding of what’s involved in running Kuber‐ netes yourself. This will help you to make an informed decision about the costs and benefits of self-hosting, as opposed to using managed services.

**Administration Overhead**

If your organization is large, with resources to spare for a dedicated Kubernetes clus‐ ter operations team, this may not be such a big problem. But for small to medium enterprises, or even startups with a handful of engineers, the administration overhead of running your own Kubernetes clusters may be prohibitive.

**The Costs of Self-Hosting Kubernetes | 39**

Given a limited budget and number of staff available for IT opera‐ tions, what proportion of your resources do you want to spend on administering Kubernetes itself? Would those resources be better used to support your business’s workloads instead? Can you oper‐ ate Kubernetes more cost-effectively with your own staff, or by using a managed service?

**Start with Managed Services**

You might be a little surprised that, in a Kubernetes book, we recommend that you don’t run Kubernetes! At least, don’t run it yourself. For the reasons we’ve outlined in the previous sections, we think that using managed services is likely to be far more cost-effective than self-hosting Kubernetes clusters. Unless you want to do something strange and experimental with Kubernetes that isn’t supported by any managed pro‐ vider, there are basically no good reasons to go the self-hosted route.

In our experience, and that of many of the people we interviewed for this book, a managed service is the best way to run Kubernetes, period.

If you’re considering whether Kubernetes is even an option for you, using a managed service is a great way to try it out. You can get a fully working, secure, highly avail‐ able, production-grade cluster in a few minutes, for a few dollars a day. (Most cloud providers even offer a free tier that lets you run a Kubernetes cluster for weeks or months without incurring any charges.) Even if you decide, after a trial period, that you’d prefer to run your own Kubernetes cluster, the managed services will show you how it should be done.

On the other hand, if you’ve already experimented with setting up Kubernetes your‐ self, you’ll be delighted with how much easier managed services make the process. You probably didn’t build your own house; why build your own cluster, when it’s cheaper and quicker to have someone else do it, and the results are better?

In the next section, we’ll outline some of the most popular managed Kubernetes serv‐ ices, tell you what we think of them, and recommend our favorite. If you’re still not convinced, the second half of the chapter will explore Kubernetes installers you can use to build your own clusters (see “Kubernetes Installers” on page 44).

We should say at this point that neither of the authors is affiliated with any cloud pro‐ vider or commercial Kubernetes vendor. Nobody’s paying us to recommend their product or service. The opinions here are our own, based on personal experience, and the views of hundreds of Kubernetes users we spoke to while writing this book.

**40 | Chapter 3: Getting Kubernetes**

Naturally, things move quickly in the Kubernetes world, and the managed services marketplace is especially competitive. Expect the features and services described here to change rapidly. The list presented here is not complete, but we’ve tried to include the services we feel are the best, the most widely used, or otherwise important. **Managed Kubernetes Services**

Managed Kubernetes services relieve you of almost all the administration overhead of setting up and running Kubernetes clusters, particularly the control plane. Effectively, a managed service means you pay for someone else (such as Google) to run the clus‐ ter for you.

**Google Kubernetes Engine (GKE)**

As you’d expect from the originators of Kubernetes, Google offers a fully managed Kubernetes service that is completely integrated with the Google Cloud Platform. Just choose the number of worker nodes, and click a button in the GCP web console to create a cluster, or use the Deployment Manager tool to provision one. Within a few minutes, your cluster will be ready to use.

Google takes care of monitoring and replacing failed nodes, auto-applying security patches, and high availability for the control plane and etcd. You can set your nodes to auto-upgrade to the latest version of Kubernetes, during a maintenance window of your choice.

**High availability** GKE gives you a production-grade, highly available Kubernetes cluster with none of the setup and maintenance overhead associated with self-hosted infrastructure. Everything is controllable via the Google Cloud API, using Deployment Manager,1 Terraform, or other tools, or you can use the GCP web console. Naturally, GKE is fully integrated with all the other services in Google Cloud.

For extended high availability, you can create *multizone* clusters, which spread worker nodes across multiple failure zones (roughly equivalent to individual data centers). Your workloads will keep on running, even if a whole failure zone is affected by an outage.

*Regional* clusters take this idea even further, by distributing multiple master nodes across failure zones, as well as workers.

1 Deployment Manager is Google’s command-line tool for managing cloud resources; not to be confused with

Kubernetes Deployments.

**Managed Kubernetes Services | 41**

**Cluster Autoscaling**

GKE also offers an attractive cluster autoscaling option (see “Autoscaling” on page 102). With autoscaling enabled, if there are pending workloads that are waiting for a node to become available, the system will add new nodes automatically to accommo‐ date the demand.

Conversely, if there is spare capacity, the autoscaler will consolidate Pods onto a smaller number of nodes and remove the unused nodes. Since billing for GKE is based on the number of worker nodes, this helps you control costs.

**GKE is best-of-breed** Google has been in the Kubernetes business longer than anybody else, and it shows. GKE is, in our opinion, the best managed Kubernetes service available. If you already have infrastructure in Google Cloud, it makes sense to use GKE to run Kubernetes. If you’re already established on another cloud, it needn’t stop you using GKE if you want to, but you should look first at managed options within your existing cloud pro‐ vider.

If you haven’t made a cloud provider decision yet, GKE is a persuasive argument in favor of choosing Google Cloud.

**Amazon Elastic Container Service for Kubernetes (EKS)**

Amazon has also been providing managed container cluster services for a long time, but until very recently the only option was Elastic Container Service (ECS), Amazon’s proprietary technology.

While perfectly usable, ECS is not as powerful or flexible as Kubernetes, and evidently even Amazon has decided that the future is Kubernetes, with the launch of Elastic Container Service for Kubernetes (EKS). (Yes, *EKS* ought to stand for *Elastic Kuber‐ netes Service*, but it doesn’t.)

It’s not quite as seamless an experience as Google Kubernetes Engine, so be prepared to do more of the setup work yourself. Also, unlike some competitors, EKS charges you for the master nodes as well as the other cluster infrastructure. This makes it more expensive, for a given cluster size, than either Google or Microsoft’s managed Kubernetes service.

If you already have infrastructure in AWS, or run containerized workloads in the older ECS service that you want to move to Kubernetes, then EKS is a sensible choice. As the newest entry into the managed Kubernetes marketplace, though, it has some distance to go to catch up to the Google and Microsoft offerings.

**42 | Chapter 3: Getting Kubernetes**

**Azure Kubernetes Service (AKS)**

Although Microsoft came a little later to the cloud business than Amazon or Google, they’re catching up fast. Azure Kubernetes Service (AKS) offers most of the features of its competitors, such as Google’s GKE. You can create clusters from the web inter‐ face or using the Azure az command-line tool.

As with GKE and EKS, you have no access to the master nodes, which are managed internally, and your billing is based on the number of worker nodes in your cluster.

**OpenShift**

OpenShift is more than just a managed Kubernetes service: it’s a full Platform-as-a- Service (PaaS) product, which aims to manage the whole software development life cycle, including continuous integration and build tools, test runner, application deployment, monitoring, and orchestration.

OpenShift can be deployed to bare-metal servers, virtual machines, private clouds, and public clouds, so you can create a single Kubernetes cluster that spans all these environments. This makes it a good choice for very large organizations, or those with very heterogeneous infrastructure.

**IBM Cloud Kubernetes Service**

Naturally, the venerable IBM is not to be left out in the field of managed Kubernetes services. IBM Cloud Kubernetes Service is pretty simple and straightforward, allow‐ ing you to set up a vanilla Kubernetes cluster in IBM Cloud.

You can access and manage your IBM Cloud cluster through the default Kubernetes CLI and the provided command-line tool, or a basic GUI. There are no real killer fea‐ tures that differentiate IBM’s offering from the other major cloud providers, but it’s a logical option if you’re already using IBM Cloud.

**Heptio Kubernetes Subscription (HKS)**

For large enterprises that want the security and flexibility of running clusters across multiple public clouds, Heptio Kubernetes Subscription (HKS) aims to provide just that.

Heptio has a solid brand in the Kubernetes world: it’s run by two of the cofounders of the Kubernetes project, Craig McLuckie and Joe Beda, and has produced many important open source tools, such as Velero (see “Velero” on page 208), and Sono‐ buoy (see “Conformance Testing with Sonobuoy” on page 104).

**Managed Kubernetes Services | 43**

**Turnkey Kubernetes Solutions**

While managed Kubernetes services are a good fit for most business requirements, there may be some circumstances in which using managed services isn’t an option. There is a growing class of *turnkey* offerings, which aim to give you a ready-to-use, production-grade Kubernetes cluster by just clicking a button in a web browser.

Turnkey Kubernetes solutions are attractive both to large enterprises (because they can have a commercial relationship with the vendor) and small companies with scarce engineering and operations resources. Here are a few of the options in the turnkey space.

**Stackpoint**

Stackpoint is advertised as “The simplest way to deploy a Kubernetes cluster to the public cloud,” in as few as three clicks. There are various price points available, start‐ ing from $50 a month, and Stackpoint offers unlimited-node clusters, high availabil‐ ity for master nodes and for etcd, and support for *federated* clusters, which can span multiple clouds (see “Federated clusters” on page 97).

As a compromise between self-hosted Kubernetes and fully managed services, Stack‐ point is an attractive option for companies that want to be able to provision and man‐ age Kubernetes from a web portal, but still run worker nodes on their own public cloud infrastructure.

**Containership Kubernetes Engine (CKE)**

CKE is another web-based interface for provisioning Kubernetes in the public cloud. It lets you get a cluster up and running with sensible defaults, or customize almost every aspect of the cluster for more demanding requirements. **Kubernetes Installers**

If managed or turnkey clusters won’t work for you, then you’ll need to consider some level of Kubernetes self-hosting: that is, setting up and running Kubernetes yourself on your own machines.

It’s very unlikely that you’ll deploy and run Kubernetes completely from scratch, except for learning and demo purposes. The vast majority of people use one or more of the available Kubernetes installer tools or services to set up and manage their clusters.

**44 | Chapter 3: Getting Kubernetes**

**kops**

kops is a command-line tool for automated provisioning of Kubernetes clusters. It’s part of the Kubernetes project, and has been around a long time as an AWS-specific tool, but is now adding beta support for Google Cloud, and support for other provid‐ ers is planned.

kops supports building high-availability clusters, which makes it suitable for produc‐ tion Kubernetes deployments. It uses declarative configuration, just like Kubernetes resources themselves, and it can not only provision the necessary cloud resources and set up a cluster, but also scale it up and down, resize nodes, perform upgrades, and do other useful admin tasks.

Like everything in the Kubernetes world, kops is under rapid development, but it’s a relatively mature and sophisticated tool that is widely used. If you’re planning to run self-hosted Kubernetes in AWS, kops is a good choice.

**Kubespray**

Kubespray (formerly known as Kargo), a project under the Kubernetes umbrella, is a tool for easily deploying production-ready clusters. It offers lots of options, including high availability, and support for multiple platforms.

Kubespray is focused on installing Kubernetes on existing machines, especially on- premise and bare-metal servers. However, it’s also suitable for any cloud environ‐ ment, including private cloud (virtual machines that run on your own servers).

**TK8**

TK8 is a command-line tool for provisioning Kubernetes clusters that leverages both Terraform (for creating cloud servers) and Kubespray (for installing Kubernetes on them). Written in Go (of course), it supports installation on AWS, OpenStack, and bare-metal servers, with support for Azure and Google Cloud in the pipeline.

TK8 not only builds a Kubernetes cluster, but will also install optional add-ons for you, including Jmeter Cluster for load testing, Prometheus for monitoring, Jaeger, Linkerd or Zipkin for tracing, Ambassador API Gateway with Envoy for ingress and load balancing, Istio for service mesh support, Jenkins-X for CI/CD, and Helm or Kedge for packaging on Kubernetes.

**Kubernetes The Hard Way**

Kelsey Hightower’s *Kubernetes The Hard Way* tutorial is perhaps best considered not as a Kubernetes setup tool or installation guide, but an opinionated walkthrough of the process of building a Kubernetes cluster which illustrates the complexity of the moving parts involved. Nonetheless, it’s very instructive, and it’s an exercise worth

**Kubernetes Installers | 45**

doing for anyone considering running Kubernetes, even as a managed service, just to get a sense of how it all works under the hood.

**kubeadm**

kubeadm is part of the Kubernetes distribution, and it aims to help you install and maintain a Kubernetes cluster according to best practices. kubeadm does not provi‐ sion the infrastructure for the cluster itself, so it’s suitable for installing Kubernetes on bare-metal servers or cloud instances of any flavor.

Many of the other tools and services we’ll mention in this chapter use kubeadm inter‐ nally to handle cluster admin operations, but there’s nothing to stop you using it directly, if you want to.

**Tarmak**

Tarmak is a Kubernetes cluster life cycle management tool that is focused on making it easy and reliable to modify and upgrade cluster nodes. While many tools deal with this by simply replacing the node, this can take a long time and often involves moving a lot of data around between nodes during the rebuild process. Instead, Tarmak can repair or upgrade the node in place.

Tarmak uses Terraform under the hood to provision the cluster nodes, and Puppet to manage configuration on the nodes themselves. This makes it quicker and safer to roll out changes to node configuration.

**Rancher Kubernetes Engine (RKE)**

RKE aims to be a simple, fast Kubernetes installer. It doesn’t provision the nodes for you, and you have to install Docker on the nodes yourself before you can use RKE to install the cluster. RKE supports high availability of the Kubernetes control plane.

**Puppet Kubernetes Module**

Puppet is a powerful, mature, and sophisticated general configuration management tool that is very widely used, and has a large open source module ecosystem. The offi‐ cially supported Kubernetes module installs and configures Kubernetes on existing nodes, including high availability support for both the control plane and etcd.

**Kubeformation**

Kubeformation is an online Kubernetes configurator that lets you choose the options for your cluster using a web interface, and will then generate configuration templates for your particular cloud provider’s automation API (for example, Deployment Man‐

**46 | Chapter 3: Getting Kubernetes**

ager for Google Cloud, or Azure Resource Manager for Azure). Support for other cloud providers is in the pipeline.

Using Kubeformation is perhaps not as simple as some other tools, but because it is a wrapper around existing automation tools such as Deployment Manager, it is very flexible. If you already manage your Google Cloud infrastructure using Deployment Manager, for example, Kubeformation will fit into your existing workflow perfectly. **Buy or Build: Our Recommendations**

This has necessarily been a quick tour of some of the options available for managing Kubernetes clusters, because the range of offerings is large and varied, and growing all the time. However, we can make a few recommendations based on commonsense principles. One of these is the philosophy of *run less software*.

**Run Less Software**

There are three pillars of the Run Less Software philosophy, all of which will help you manipulate time and defeat your enemies.

1. Choose standard technology

2. Outsource undifferentiated heavy lifting

3. Create enduring competitive advantage

—Rich Archbold

While using innovative new technologies is fun and exciting, it doesn’t always make sense from a business point of view. Using *boring* software that everybody else is using is generally a good bet. It probably works, it’s probably well-supported, and you’re not going to be the one taking the risks and dealing with the inevitable bugs.

If you’re running containerized workloads and cloud native applications, Kubernetes is the boring choice, in the best possible way. Given that, you should opt for the most mature, stable, and widely used Kubernetes tools and services.

*Undifferentiated heavy lifting* is a term coined at Amazon to denote all the hard work and effort that goes into things like installing and managing software, maintaining infrastructure, and so on. There’s nothing special about this work; it’s the same for you as it is for every other company out there. It costs you money, instead of making you money.

The *run less software* philosophy says that you should outsource undifferentiated heavy lifting, because it’ll be cheaper in the long run, and it frees up resources you can use to work on your core business.

**Buy or Build: Our Recommendations | 47**

**Use Managed Kubernetes if You Can**

With the *run less software* principles in mind, we recommend that you outsource your Kubernetes cluster operations to a managed service. Installing, configuring, maintaining, securing, upgrading, and making your Kubernetes cluster reliable is undifferentiated heavy lifting, so it makes sense for almost all businesses not to do it themselves:

*Cloud native* is not a cloud provider, it’s not Kubernetes, it’s not containers, it’s not a technology. It’s the practice of accelerating your business by not running stuff that doesn’t differentiate you.

—Justin Garrison

In the managed Kubernetes space, Google Kubernetes Engine (GKE) is the clear win‐ ner. While other cloud providers may catch up in a year or two, Google is still way ahead and will remain so for some time to come.

For companies that need to be independent of a single cloud provider, and want 24- hour-a-day technical support from a trusted brand, Heptio Kubernetes Subscription is worth looking at.

If you want managed high availability for your cluster control plane, but need the flexibility of running your own worker nodes, consider Stackpoint.

**But What About Vendor Lock-in?**

If you commit to a managed Kubernetes service from a particular vendor, such as Google Cloud, will that lock you in to the vendor and reduce your options in the future? Not necessarily. Kubernetes is a standard platform, so any applications and services you build to run on Google Kubernetes Engine will also work on any other certified Kubernetes provider’s system. Just using Kubernetes in the first place is a big step toward escaping vendor lock-in.

Does managed Kubernetes make you more prone to lock-in than running your own Kubernetes cluster? We think it’s the other way around. Self-hosting Kubernetes involves a lot of machinery and configuration to maintain, all of which is intimately tied in to a specific cloud provider’s API. Provisioning AWS virtual machines to run Kubernetes, for example, requires completely different code than the same operation on Google Cloud. Some Kubernetes setup assistants, like the ones we’ve mentioned in this chapter, support multiple cloud providers, but many don’t.

Part of the point of Kubernetes is to abstract away the technical details of the cloud platform, and present developers with a standard, familiar interface that works the same way whether it happens to be running on Azure or Google Cloud. As long as you design your applications and automation to target Kubernetes itself, rather than

**48 | Chapter 3: Getting Kubernetes**

the underlying cloud infrastructure, you’re as free from vendor lock-in as you can reasonably be.

**Use Standard Kubernetes Self-Hosting Tools if You Must**

If you have special requirements which mean that managed Kubernetes offerings won’t work for you, only then should you consider running Kubernetes yourself.

If that’s the case, you should go with the most mature, powerful, and widely used tools available. We recommend kops or Kubespray, depending on your requirements.

If you know that you’ll be staying with a single cloud provider long-term, especially if it’s AWS, use kops.

On the other hand, if you need your cluster to span multiple clouds or platforms, including bare-metal servers, and you want to keep your options open, you should use Kubespray.

**When Your Choices Are Limited**

There may be business, rather than technical, reasons, why fully managed Kubernetes services aren’t an option for you. If you have an existing business relationship with a hosting company or cloud provider that doesn’t offer a managed Kubernetes service, that will necessarily limit your choices.

However, it may be possible for you to use a turnkey solution instead, such as Stack‐ point or Containership. These options provide a managed service for your Kuber‐ netes master nodes, but connect them to worker nodes running on your own infrastructure. Since most of the administration overhead of Kubernetes is in setting up and maintaining the master nodes, this is a good compromise.

**Bare-Metal and On-Prem**

It may come as a surprise to you that being cloud native doesn’t actually require being *in the cloud*, in the sense of outsourcing your infrastructure to a public cloud provider such as Azure or AWS.

Many organizations run part or all of their infrastructure on bare-metal hardware, whether colocated in data centers or on-premises. Everything we’ve said in this book about Kubernetes and containers applies just as well to in-house infrastructure as it does to the cloud.

You can run Kubernetes on your own hardware machines; if your budget is limited, you can even run it on a stack of Raspberry Pis (Figure 3-3). Some businesses run a *private cloud*, consisting of virtual machines hosted by on-prem hardware.

**Buy or Build: Our Recommendations | 49**

*Figure 3-3. Kubernetes on a budget: a Raspberry Pi cluster (photo by David Merrick)* **Clusterless Container Services**

If you really want to minimize the overhead of running container workloads, there’s yet another level above fully managed Kubernetes services. These are so-called *clus‐ terless* services, such as Azure Container Instances or Amazon’s Fargate. Although there really is a cluster under the hood, you don’t have access to it via tools like kubectl. Instead, you specify a container image to run, and a few parameters like the CPU and memory requirements of your application, and the service does the rest.

**Amazon Fargate**

According to Amazon, “Fargate is like EC2, but instead of a virtual machine, you get a container.” Unlike ECS, there’s no need to provision cluster nodes yourself and then connect them to a control plane. You just define a task, which is essentially a set of instructions for how to run your container image, and launch it. Pricing is per-second based on the amount of CPU and memory resources that the task consumes.

It’s probably fair to say that Fargate makes sense for simple, self-contained, long- running compute tasks or batch jobs (such as data crunching) that don’t require much customization or integration with other services. It’s also ideal for build con‐ tainers, which tend to be short-lived, and for any situation where the overhead of managing worker nodes isn’t justified.

**50 | Chapter 3: Getting Kubernetes**

If you’re already using ECS with EC2 worker nodes, switching to Fargate will relieve you of the need to provision and manage those nodes. Fargate is available now in some regions for running ECS tasks, and is scheduled to support EKS by 2019.

**Azure Container Instances (ACI)**

Microsoft’s Azure Container Instances (ACI) service is similar to Fargate, but also offers integration with the Azure Kubernetes Service (AKS). For example, you can configure your AKS cluster to provision temporary extra Pods inside ACI to handle spikes or bursts in demand.

Similarly, you can run batch jobs in ACI in an ad hoc way, without having to keep idle nodes around when there’s no work for them to do. Microsoft calls this idea *serverless containers*, but we find that terminology both confusing (*serverless* usually refers to cloud functions, or functions-as-a-service) and inaccurate (there are servers; you just can’t access them).

ACI is also integrated with Azure Event Grid, Microsoft’s managed event routing ser‐ vice. Using Event Grid, ACI containers can communicate with cloud services, cloud functions, or Kubernetes applications running in AKS.

You can create, run, or pass data to ACI containers using Azure Functions. The advantage of this is that you can run any workload from a cloud function, not just those using the officially supported (*blessed*) languages, such as Python or JavaScript.

If you can containerize your workload, you can run it as a cloud function, with all the associated tooling. For example, Microsoft Flow allows even nonprogrammers to build up workflows graphically, connecting containers, functions, and events. **Summary**

Kubernetes is everywhere! Our journey through the extensive landscape of Kuber‐ netes tools, services, and products has been necessarily brief, but we hope you found it useful.

While our coverage of specific products and features is as up to date as we can make it, the world moves pretty fast, and we expect a lot will have changed even by the time you read this.

However, we think the basic point stands: it’s not worth managing Kubernetes clus‐ ters yourself if a service provider can do it better and cheaper.

In our experience of consulting for companies migrating to Kubernetes, this is often a surprising idea, or at least not one that occurs to a lot of people. We often find that organizations have taken their first steps with self-hosted clusters, using tools like

**Summary | 51**

kops, and hadn’t really thought about using a managed service such as GKE. It’s well worth thinking about.

More things to bear in mind:

• Kubernetes clusters are made up of *master nodes*, which run the *control plane*, and *worker nodes*, which run your workloads.

• Production clusters must be *highly available*, meaning that the failure of a master node won’t lose data or affect the operation of the cluster.

• It’s a long way from a simple demo cluster to one that’s ready for critical produc‐ tion workloads. High availability, security, and node management are just some of the issues involved.

• Managing your own clusters requires a significant investment of time, effort, and expertise. Even then, you can still get it wrong.

• Managed services like Google Kubernetes Engine do all the heavy lifting for you, at much lower cost than self-hosting.

• Turnkey services are a good compromise between self-hosted and fully managed Kubernetes. Turnkey providers like Stackpoint manage the master nodes for you, while you run worker nodes on your own machines.

• If you have to host your own cluster, kops is a mature and widely used tool that can provision and manage production-grade clusters on AWS and Google Cloud.

• You should use managed Kubernetes if you can. This is the best option for most businesses in terms of cost, overhead, and quality.

• If managed services aren’t an option, consider using turnkey services as a good compromise.

• Don’t self-host your cluster without sound business reasons. If you do self-host, don’t underestimate the engineering time involved for the initial setup and ongo‐ ing maintenance overhead.

**52 | Chapter 3: Getting Kubernetes**

**CHAPTER 4 Working with Kubernetes Objects**

I can’t understand why people are frightened of new ideas. I’m frightened of the old ones.—John Cage

In Chapter 2, you built and deployed an application to Kubernetes. In this chapter, you’ll learn about the fundamental Kubernetes objects involved in that process: Pods, Deployments, and Services. You’ll also find out how to use the essential Helm tool to manage application in Kubernetes.

After working through the example in “Running the Demo App” on page 30, you should have a container image running in the Kubernetes cluster, but how does that actually work? Under the hood, the kubectl run command creates a Kubernetes resource called a *Deployment*. So what’s that? And how does a Deployment actually run your container image? **Deployments**

Think back to how you ran the demo app with Docker. The docker container run command started the container, and it ran until you killed it with docker stop.

But suppose that the container exits for some other reason; maybe the program crashed, or there was a system error, or your machine ran out of disk space, or a cos‐ mic ray hit your CPU at the wrong moment (unlikely, but it does happen). Assuming this is a production application, that means you now have unhappy users, until some‐ one can get to a terminal and type docker container run to start the container again.

That’s an unsatisfactory arrangement. What you really want is a kind of supervisor program, which continually checks that the container is running, and if it ever stops,**53**