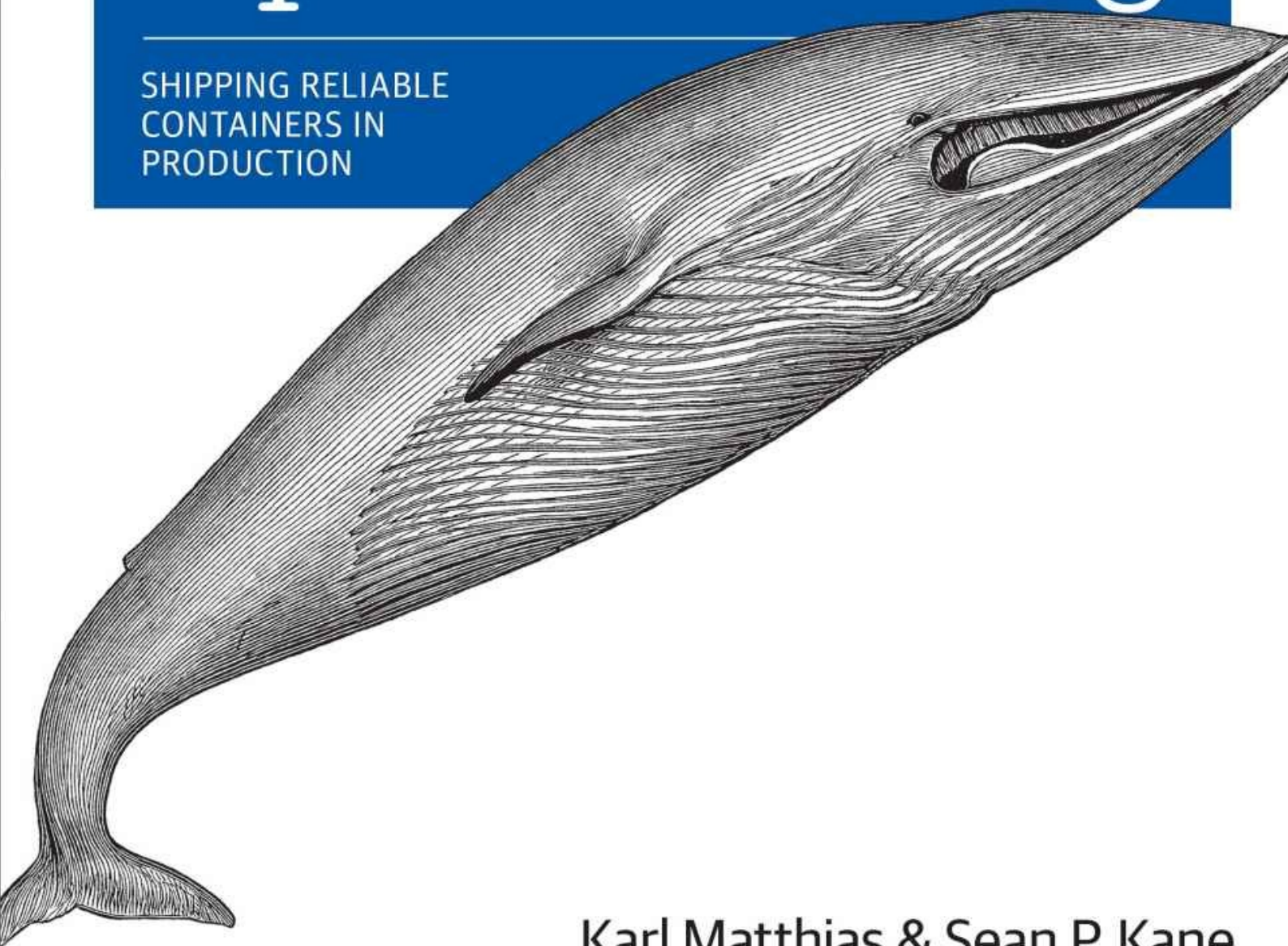


O'REILLY®

# Docker Up & Running

SHIPPING RELIABLE  
CONTAINERS IN  
PRODUCTION



Karl Matthias & Sean P. Kane

---

# Docker: Up and Running

Karl Matthias and Sean P. Kane

# Docker: Up and Running

---

by Karl Matthias and Sean P. Kane

Copyright © 2015 Karl Matthias, Sean P. Kane. 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://safaribooksonline.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or [corporate@oreilly.com](mailto:corporate@oreilly.com).

- Editor: Brian Anderson
- Production Editor: Melanie Yarbrough
- Copyeditor: Gillian McGarvey
- Proofreader: Sonia Saruba
- Indexer: Wendy Catalano
- Interior Designer: David Futato
- Cover Designer: Ellie Volkhausen
- Illustrator: Rebecca Demarest
- June 2015: First Edition

## Revision History for the First Edition

- 2015-06-09: First Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781491917572> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Docker: Up and Running*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

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.

978-1-491-91757-2

[LSI]

## **Dedication**

---

For my wife and children, who make everything worth it. And my parents, who pointed me towards the beautiful intersection between logic and passion.

— Sean P. Kane

For my Mom, who got me to read, and my Dad, who read to me. And for my wife and daughters, who are my bedrock.

— Karl Matthias

## Foreword

---

“*Everything old is new again*” is a commonly heard phrase that has described everything from fashion, to politics, to technology. It is also an apt statement when it comes to Linux containers, and I would expand upon it to say, “Everything old is new again — and nonetheless exciting.”

Containers have been available for many years in Linux distributions but they’ve seldom been used because of the complexity required to build something that worked. Thus historically, Linux container implementations have been purpose-built with a single objective in mind, which made additional requirements like scaling and portability challenging — if not impossible — to implement.

Enter Docker, which has created phenomenal momentum in unlocking the value of Linux containers by combining a standardized packaging format with ease of use, to turn processes that were once esoteric and incomprehensible into consumable capabilities for developers and operations teams. Docker, in a sense, has created a Renaissance for Linux containers, driving an ever-growing wave of interest and possibility, leading to rapid adoption of the technology. It’s helping technology teams everywhere realize the benefits of application portability, simplified integration, and streamlined development as promised by Linux containers for some time but historically trapped behind layers of complexity.

Through Docker, Linux containers have catapulted into an elite club of truly disruptive technologies with the power to transform the IT landscape, related ecosystems, and markets. In the wake of this emergence rises a wave of innovation that demonstrates Linux containers’ potential to dramatically change application delivery across a variety of computing environments and platforms while leveraging a spectrum of technical skill sets.

*Innovation* doesn’t necessarily mean the introduction of a completely new, world-altering technology. Like many of its predecessors, Docker’s success stands on the shoulder of giants. It builds on years of technological innovation and Linux evolution that now provides the core capabilities which Docker makes easy to use. The maturity of the Linux capabilities exploited by Docker can now be replicated in other operating systems, allowing Docker to function beyond its Linux roots.

Docker is facilitating a disruptive change in the minds of technology professionals. It has reshaped views on which aspects of application development and delivery, as well as infrastructure management should be considered “table stakes” versus complexity that requires technology or process solutions. As is typical for the early adoption phase of any disruptive technology, these perspective changes aim at what’s right in front of us, often oversimplifying and ignoring relevant aspects — but the potential for Docker and Linux containers goes much deeper than simply redefining development. It is redefining the very nature of the application itself.

The obvious impact of Docker and the ease of use it brings to Linux containers is the possibility to redefine the organizational divide between business, application development, and IT infrastructure teams. In a sense, Docker provides a tangible technology for implementing DevOps, which is the merger (or at least an armistice) between the often competing teams of development and operations. Containerization modernizes IT environments and, at an organizational level, allows for “proper” ownership of the technology stack and processes, reducing handovers and the costly change coordination that comes with them.

Docker’s role as both a packaging format for the application *and* a unifying interface and

methodology enables the application team to own the Docker-formatted container image, including all dependencies, while allowing operations to retain infrastructure ownership. With a standardized container infrastructure in place, the IT organization can then focus on building and managing deployments, meeting their security standards, automation needs, skill levels and ultimately cost profile, all without losing the ability to hold the application team accountable for the security and cost impact of their code that is deployed inside the container.

Docker also brings with it greater efficiencies of scale and performance — by shrinking application footprints through Docker-formatted containers, system-level dependencies are reduced to a bare minimum, often dozens-to-hundreds of megabytes in size. Compare this to traditional virtual machine images, which typically consume gigabytes of storage...but when you factor in performance, it goes beyond simply being innovative and becomes truly disruptive.

Starting a container takes milliseconds — quite a difference compared to the minutes most users experience with virtual machines. Deploying container images is faster if less data needs to travel over networks and storage fabrics, so modern, elastic applications with frequent state changes and dynamic allocation of resources can be built far more efficiently if the rollout of changes can happen extremely quickly and resource needs can be fulfilled in real time.

But perhaps the greatest innovation and most significant impact delivered by Docker and Linux containers is the fundamental change to application consumption. The monolithic application stack as we know it can be broken into dozens or even hundreds of tiny, single-minded applications that, when woven together, perform the same function as the traditional application. The benefit, however, is that these pieces can be rewritten, reused, and managed far more efficiently than monolithic applications, delivering a truly composite application built entirely of microservices.

Containers represent the way forward for the application development world, but it's critical that we do not lose sight of the old as we bring in the new. Docker and Linux containers are not without challenges. Management, security, and certification are three of the most glaring challenges to enterprise adoption, and these concerns are not so dissimilar from more traditional applications. Obviously, containers must be deployed on a secure host, but, more importantly, container security will be defined by what is in a given container — is it free of vulnerabilities, malware, and known exploits? Having the appropriate signature on a given containerized application, from a trusted, certified source goes a long way towards effectively answering these questions.

Additionally, management is paramount when it comes to the new world of containerized applications and Docker. The potential for sheer sprawl with containers is exponentially greater than that of virtual machines. Managing all of the containers is one challenge, but just as important, much like security, will be managing the content inside these containers. How are updates and rollbacks handled? What about orchestration? What will define “sprawl”? When do containers need to be retired or archived, else spend a life of limbo on an aging server? These too are questions that the enterprise must see addressed before *mission critical* can be applied to containerized applications.

These challenges aside, Linux containers do represent a fundamental shift in how the enterprise world creates, consumes, and manages applications. While the traditional monolithic application is likely to remain (lots of old technology survives as a legacy still today), containers have huge potential to modernize the operational model for these traditional applications and, alongside new, container-

based applications, allow for incredible flexibility, portability, and efficiency across the datacenter and hybrid clouds.

---

Sean and Karl have worked with Linux containers (and Docker) for years, and have, in this book, captured what the IT world needs to know about Docker and the *container Renaissance*. Leveraging their insights, the authors provide a solid overview of how Docker actually works in the real world and how developers and other IT professionals can best utilize Docker and Linux containers in a way that makes sense for them and their organization.

Lars Herrmann, General Manager for Enterprise Linux, Enterprise Virtualization and Container Strategy, Red Hat

# Preface

---

This book will get you to the point where you have a running Docker environment and steer you towards good choices for a production environment. Along the way we'll explore building applications for Docker, testing, deploying, and debugging a running system. We'll stop by to see a few of the orchestration tools and platforms in the Docker ecosystem. And we'll round out with guidance on security and best practices for your container environment.

## Who Should Read This Book

This book is intended for anyone who is looking to solve the complex workflow problems involved in developing and deploying software to production at scale. If you're interested in Docker, Linux containers, DevOps, and large, scalable, software infrastructures, then this book is for you.

## Why Read This Book?

Today there are many conversations, projects, and articles on the Internet about Docker. So why should you devote precious hours to reading this book?

Even though there is a lot of information out there, Docker is a new technology and it is evolving very quickly. Even during the time that it took us to write this book, Docker, Inc., released four versions of Docker plus a few major tools into their ecosystem. Getting your arms around the scope of what Docker provides, understanding how it fits into your workflow, and getting integration right are not trivial tasks. Few companies or engineering teams have been running it in production for more than a year.

We have worked for over a year and a half building and operating a production Docker platform within the Site Engineering team at New Relic. We implemented Docker in production only months after its release and can share with you some of the experience we gained from evolving our production platform over the last year and a half. The goal is for you to enjoy the wins while avoiding many of the bumps in the road that we experienced. Even though the online documentation for the Docker project is useful, we attempt to give you a bigger picture and expose you to many of the best practices that we have learned along the way.

When you finish this book, you should have enough information to understand what Docker is, why it's important, how to get it running, how to deploy your applications with it, and be armed with a clear view of what you can do to get to production. It will hopefully be a quick trip through an interesting technology with some very practical applications.

## Navigating This Book

This book is organized as follows:



- Chapters [1](#) and [2](#) provide an introduction to Docker, and explain what it is and how you can use it.
- [Chapter 3](#) takes you through the steps required to install Docker.
- Chapters [4](#) through [6](#) dive into the Docker client, images, and containers, exploring what they are and how you can work with them.
- Chapters [7](#) and [8](#) discuss the flow for getting your containers into production and debugging them.
- [Chapter 9](#) delves into deploying containers at scale in public and private clouds.
- [Chapter 10](#) dives into advanced topics that require some familiarity with Docker and can be important as you start to use Docker in your production environment.
- [Chapter 11](#) explores some of the core concepts that have started to solidify in the industry about how to design the next generation of Internet-scale production software.
- [Chapter 12](#) wraps everything up and ties it with a bow. It includes a summary of what you have and how it should help you improve the way you deliver and scale software services.

We realize that many people don't read technical books front to back and that something like the preface is incredibly easy to skip, but if you're still with us, here is a quick guide to some different approaches to reading this book:

- If you are new to Linux containers, start at the beginning. The first two chapters are intended to get your head around the basics of Docker and Linux containers, including what they are, how they work, and why you should care.
- If you want to jump right in and install and run Docker on your workstation, then dive right into Chapters [3](#) and [4](#), which show you how to install Docker, create and download images, run containers, and much more.
- If you are already using Docker for development but need some help getting it into production, consider starting with Chapters [7](#) through [10](#), which delve into deploying and debugging containers and many other advanced topics.
- If you are a software or platform architect, you might find [Chapter 11](#) an interesting place to investigate, as we dive into some of the current thinking about designing containerized applications and horizontally scalable services.

## Conventions Used in This Book

The following typographical conventions are used in this book:

### *Italic*

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

### *Constant width*

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

### ***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 determined by context.

#### **TIP**

This element signifies a tip or suggestion.

#### **NOTE**

This element signifies a general note.

#### **WARNING**

This element indicates a warning or caution.

## **Safari® Books Online**

### **NOTE**

*Safari Books Online* is an on-demand digital library that delivers expert **content** in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of **plans and pricing** for **enterprise, government, education**, and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, 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, Course Technology, and hundreds **more**. For more information about Safari Books Online, please visit us **online**.

## **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 <http://bit.ly/docker-up-and-running>.

To comment or ask technical questions about this book, send email to [bookquestions@oreilly.com](mailto:bookquestions@oreilly.com).

For more information about our books, courses, conferences, and news, see our website 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

We'd like to send a heartfelt thanks to the many people who helped make this book possible:

- Nic Benders, Bjorn Freeman-Benson, and Dana Lawson at New Relic, who went far above and beyond in supporting this effort, and who ensured that we had time to pursue it.
- Laurel Ruma at O'Reilly who initially reached out to us about writing a Docker book, and Mike Loukides who helped get everything on track.
- Gillian McGarvey and Melanie Yarbrough, for their efforts copyediting the manuscript, and helping it appear like we were actually paying attention in our high school English classes. 464 commas added and counting...
- Wendy Catalano, who helped us ensure that the Index was useful to all of our readers.
- A special thanks to our editor, Brian Anderson, who ensured that we knew what we were getting into, and guided us along every step of the way.
- All of our peers at New Relic, who have been along for the whole Docker ride and provided us with much of the experience that's reflected here.
- World Cup Coffee, McMenamins Ringlers Pub, and Old Town Pizza in Portland, OR, who kindly let us use their tables and power long after our dishes were empty.

- Our draft reviewers, who helped ensure that we were on the right track at various points throughout the writing process: ~~Ksenia Burlachenko, who gave us our very first review as well as a full tech review~~, Andrew T. Baker, Sébastien Goasguen, and Henri Gomez.
- A special callout is due to Alice Goldfuss and Tom Offermann who gave us detailed and consistently useful feedback.
- Our families, for being supportive and giving us the required quiet time when we needed it.
- And finally to everyone else who encouraged us, gave us advice, or supported us in any way throughout this process.

# Chapter 1. Introduction

---

## The Birth of Docker

Docker was first introduced to the world — with no pre-announcement and little fanfare — by Solomon Hykes, founder and CEO of dotCloud, in a five-minute **lightning talk** at the Python Developers Conference in Santa Clara, California, on March 15, 2013. At the time of this announcement, only about 40 people outside dotCloud been given the opportunity to play with Docker.

Within a few weeks of this announcement, there was a surprising amount of press. The project was quickly open-sourced and made publicly available on **GitHub**, where anyone could download and contribute to the project. Over the next few months, more and more people in the industry started hearing about Docker and how it was going to revolutionize the way software was built, delivered, and run. And within a year, almost no one in the industry was unaware of Docker, but many were still unsure what it was exactly, and why people were so excited about.

Docker is a tool that promises to easily encapsulate the process of creating a distributable artifact for any application, deploying it at scale into any environment, and streamlining the workflow and responsiveness of agile software organizations.

## The Promise of Docker

While ostensibly viewed as a virtualization platform, Docker is far more than that. Docker's domain spans a few crowded segments of the industry that include technologies like KVM, Xen, OpenStack, Mesos, Capistrano, Fabric, Ansible, Chef, Puppet, SaltStack, and so on. There is something very telling about the list of products that Docker competes with, and maybe you've spotted it already. For example, most engineers would not say that virtualization products compete with configuration management tools, yet both technologies are being disrupted by Docker. The technologies in that list are also generally acclaimed for their ability to improve productivity and that's what is causing a great deal of the buzz. Docker sits right in the middle of some of the most enabling technologies of the last decade.

If you were to do a feature-by-feature comparison of Docker and the reigning champion in any of these areas, Docker would very likely look like a middling competitor. It's stronger in some areas than others, but what Docker brings to the table is a feature set that crosses a broad range of workflow challenges. By combining the ease of application deployment tools like Capistrano and Fabric, with the ease of administrating virtualization systems, and then providing hooks that make workflow automation and orchestration easy to implement, Docker provides a very enabling feature set.

Lots of new technologies come and go, and a dose of skepticism about the newest rage is always healthy. Without digging deeper, it would be easy to dismiss Docker as just another technology that solves a few very specific problems for developers or operations teams. If you look at Docker as a virtualization or deployment technology alone, it might not seem very compelling. But Docker is much more than it seems on the surface.

It is hard and often expensive to get communication and processes right between teams of people, even in smaller organizations. Yet we live in a world where the communication of detailed information between teams is increasingly required to be successful. A tool that reduces the complexity of that communication while aiding in the production of more robust software would be a big win. And that's exactly why Docker merits a deeper look. It's no panacea, and implementing Docker well requires some thought, but Docker is a good approach to solving some real-world organizational problems and helping enable companies to ship better software faster. Delivering a well-designed Docker workflow can lead to happier technical teams and real money for the organization's bottom line.

So where are companies feeling the most pain? Shipping software at the speed expected in today's world is hard to do well, and as companies grow from one or two developers to many teams of developers, the burden of communication around shipping new releases becomes much heavier and harder to manage. Developers have to understand a lot of complexity about the environment they will be shipping software into, and production operations teams need to increasingly understand the internals of the software they ship. These are all generally good skills to work on because they lead to a better understanding of the environment as a whole and therefore encourage the designing of robust software, but these same skills are very difficult to scale effectively as an organization's growth accelerates.

The details of each company's environment often require a lot of communication that doesn't directly build value in the teams involved. For example, requiring developers to ask an operations team for *release 1.2.1* of a particular library slows them down and provides no direct business value to the company. If developers could simply upgrade the version of the library they use, write their code, test with the new version, and ship it, the delivery time would be measurably shortened. If operations people could upgrade software on the host system without having to coordinate with multiple teams of application developers, they could move faster. Docker helps to build a layer of isolation in software that reduces the burden of communication in the world of humans.

Beyond helping with communication issues, Docker is opinionated about software architecture in a way that encourages more robustly crafted applications. Its architectural philosophy centers around atomic or throwaway containers. During deployment, the whole running environment of the old application is thrown away with it. Nothing in the environment of the application will live longer than the application itself and that's a simple idea with big repercussions. It means that applications are not likely to accidentally rely on artifacts left by a previous release. It means that ephemeral debugging changes are less likely to live on in future releases that picked them up from the local filesystem. And it means that applications are highly portable between servers because all state has to be included directly into the deployment artifact and be immutable, or sent to an external dependency like a database, cache, or file server.

This leads to applications that are not only more scalable, but more reliable. Instances of the application container can come and go with little repercussion on the uptime of the frontend site. These are proven architectural choices that have been successful for non-Docker applications, but the design choices included in Docker's own design mean that Dockerized applications will follow these best practices by requirement and that's a good thing.

## Benefits of the Docker Workflow

---

It's hard to cohesively group into categories all of the things Docker brings to the table. When implemented well, it benefits organizations, teams, developers, and operations engineers in a multitude of ways. It makes architectural decisions simpler because all applications essentially look the same on the outside from the hosting system's perspective. It makes tooling easier to write and share between applications. Nothing in this world comes with benefits and no challenges, but Docker is surprisingly skewed toward the benefits. Here are some more of the things you get with Docker:

*Packaging software in a way that leverages the skills developers already have.*

Many companies have had to create positions for release and build engineers in order to manage all the knowledge and tooling required to create software packages for their supported platforms. Tools like rpm, mock, dpkg, and pbuilder can be complicated to use, and each one must be learned independently. Docker wraps up all your requirements together into one package that is defined in a single file.

*Bundling application software and required OS filesystems together in a single standardized image format.*

In the past, you typically needed to package not only your application, but many of the dependencies that it relied on, including libraries and daemons. However, you couldn't ever ensure that 100 percent of the execution environment was identical. All of this made packaging difficult to master, and hard for many companies to accomplish reliably. Often someone running Scientific Linux would resort to trying to deploy a community package tested on Red Hat Linux, hoping that the package was close enough to what they needed. With Docker you deploy your application along with every single file required to run it. Docker's layered images make this an efficient process that ensures that your application is running in the expected environment.

*Using packaged artifacts to test and deliver the exact same artifact to all systems in all environments.*

When developers commit changes to a version control system, a new Docker image can be built which can go through the whole testing process and be deployed to production without any need to recompile or repackage at any step in the process.

*Abstracting software applications from the hardware without sacrificing resources.*

Traditional enterprise virtualization solutions like VMware are typically used when people need to create an abstraction layer between the physical hardware and the software applications that run on it, at the cost of resources. The hypervisors that manage the VMs and each VM running kernel use a percentage of the hardware system's resources, which are then no longer available to the hosted applications. A container, on the other hand, is just another process that talks directly to the Linux kernel and therefore can utilize more resources, up until the system or quota-based limits are reached.

When Docker was first released, Linux containers had been around for quite a few years, and many of the other technologies that it is built on are not entirely new. However, Docker's unique mix of strong architectural and workflow choices combine together into a whole that is much more powerful than the sum of its parts. Docker finally makes Linux containers, which have been around for more than a decade, approachable to the average technologist. It fits containers relatively easily into the existing

workflow and processes of real companies. And the problems discussed above have been felt by so many people that interest in the Docker project has been accelerating faster than anyone could have reasonably expected.

In the first year, newcomers to the project were surprised to find out that Docker wasn't already production-ready, but a steady stream of commits from the open source Docker community has moved the project forward at a very brisk pace. That pace seems to only pick up steam as time goes on. As Docker has now moved well into the 1.x release cycle, stability is good, production adoption is here, and many companies are looking to Docker as a solution to some of the serious complexity issues that they face in their application delivery processes.

## What Docker Isn't

Docker can be used to solve a wide breadth of challenges that other categories of tools have traditionally been enlisted to fix; however, Docker's breadth of features often means that it lacks depth in specific functionality. For example, some organizations will find that they can completely remove their configuration management tool when they migrate to Docker, but the real power of Docker is that although it can replace some aspects of more traditional tools, it is usually compatible with them or even augmented by combining with them, as well. In the following list, we explore some of the tool categories that Docker doesn't directly replace but that can often be used in conjunction to achieve great results:

### *Enterprise Virtualization Platform (VMware, KVM, etc.)*

A container is not a virtual machine in the traditional sense. Virtual machines contain a complete operating system, running on top of the host operating system. The biggest advantage is that it is easy to run many virtual machines with radically different operating systems on a single host. With containers, both the host and the containers share the same kernel. This means that containers utilize fewer system resources, but must be based on the same underlying operating system (i.e., Linux).

### *Cloud Platform (Openstack, CloudStack, etc.)*

Like Enterprise virtualization, the container workflow shares a lot of similarities on the surface with cloud platforms. Both are traditionally leveraged to allow applications to be horizontally scaled in response to changing demand. Docker, however, is not a cloud platform. It only handles deploying, running, and managing containers on pre-existing Docker hosts. It doesn't allow you to create new host systems (instances), object stores, block storage, and the many other resources that are typically associated with a cloud platform.

### *Configuration Management (Puppet, Chef, etc.)*

Although Docker can significantly improve an organization's ability to manage applications and their dependencies, it does not directly replace more traditional configuration management. Dockerfiles are used to define how a container should look at build time, but they do not manage the container's ongoing state, and cannot be used to manage the Docker host system.

### *Deployment Framework (Capistrano, Fabric, etc.)*



Docker eases many aspects of deployment by creating self-contained container images that encapsulate all the dependencies of an application and can be deployed, in all environments, without changes. However, Docker can't be used to automate a complex deployment process by itself. Other tools are usually still needed to stitch together the larger workflow automation.

### *Workload Management Tool (Mesos, Fleet, etc.)*

The Docker server does not have any internal concept of a cluster. Additional orchestration tools (including Docker's own Swarm tool) must be used to coordinate work intelligently across a pool of Docker hosts, and track the current state of all the hosts and their resources, and keep an inventory of running containers.

### *Development Environment (Vagrant, etc.)*

Vagrant is a virtual machine management tool for developers that is often used to simulate server stacks that closely resemble the production environment in which an application is destined to be deployed. Among other things, Vagrant makes it easy to run Linux software on Mac OS X and Windows-based workstations. Since the Docker server only runs on Linux, Docker provides boot2docker and Docker Machine to allow developers to quickly launch Linux-based Docker machines on various platforms. Boot2docker and Docker Machine are sufficient for many standard Docker workflows, but they don't provide the breadth of features found in Vagrant.

Wrapping your head around Docker can be challenging when you are coming at it without a strong frame of reference. In the next chapter we will lay down a broad overview of Docker, what it is, how it is intended to be used, and what advantages it brings to the table when implemented with all of this in mind.

# Chapter 2. Docker at a Glance

---

Before you dive into configuring and installing Docker, a quick survey is in order to explain what Docker is and what it can bring to the table. It is a powerful technology, but not a tremendously complicated one. In this chapter, we'll cover the generalities of how Docker works, what makes it powerful, and some of the reasons you might use it. If you're reading this, you probably have your own reasons to use Docker, but it never hurts to augment your understanding before you dive in.

Don't worry — this shouldn't hold you up for too long. In the next chapter, we'll dive right into getting Docker installed and running on your system.

## Process Simplification

Docker can simplify both workflows and communication, and that usually starts with the deployment story. Traditionally, the cycle of getting an application to production often looks something like the following (illustrated in [Figure 2-1](#)):

1. Application developers request resources from operations engineers.
2. Resources are provisioned and handed over to developers.
3. Developers script and tool their deployment.
4. Operations engineers and developers tweak the deployment repeatedly.
5. Additional application dependencies are discovered by developers.
6. Operations engineers work to install the additional requirements.
7. Loop over steps 5 and 6  $N$  more times.
8. The application is deployed.

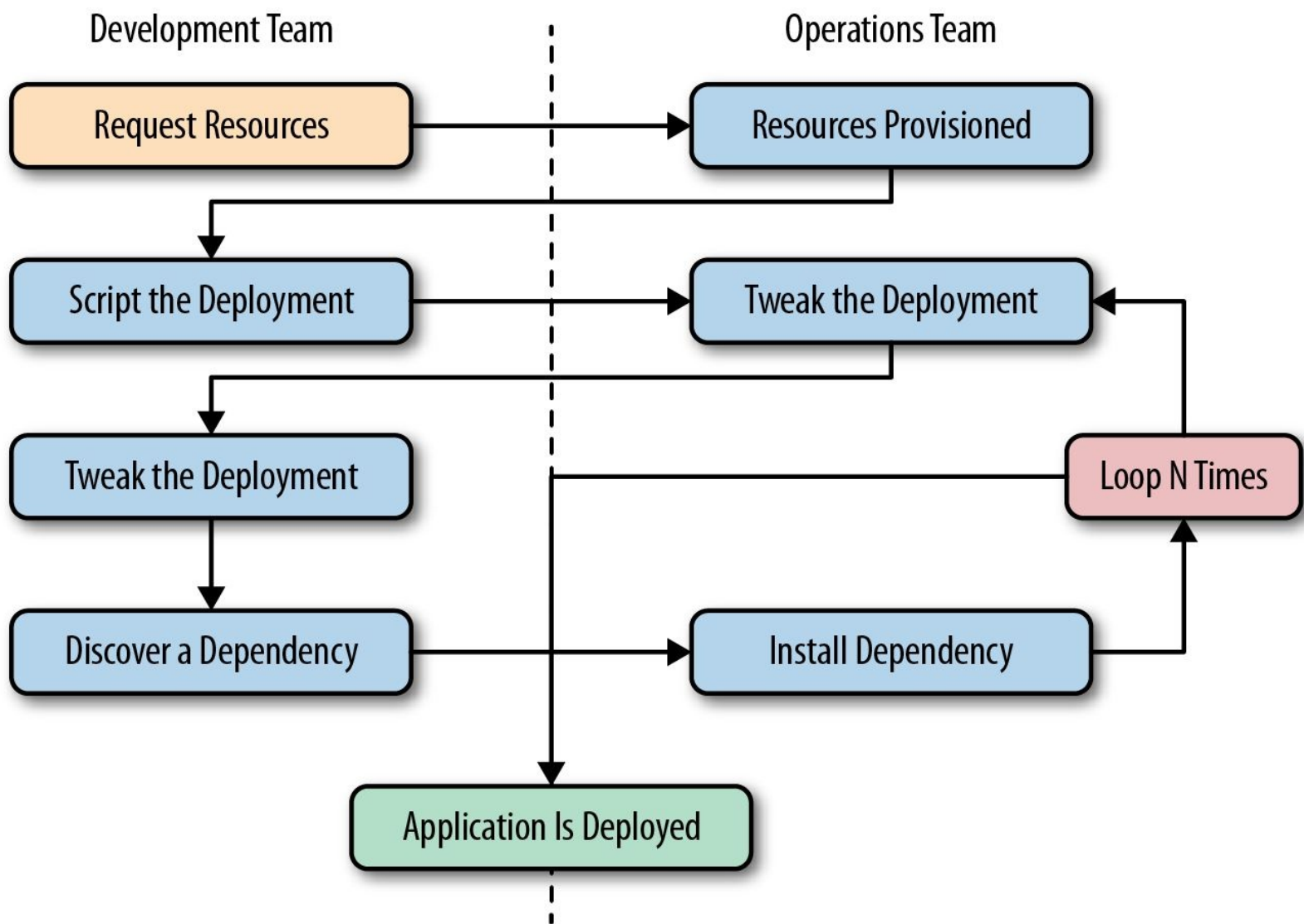


Figure 2-1. A traditional deployment workflow (without Docker)

Our experience has shown that deploying a brand new application into production can take the better part of a week for a complex new system. That's not very productive, and even though DevOps practices work to alleviate some of the barriers, it often requires a lot of effort and communication between teams of people. This process can often be both technically challenging and expensive, but even worse, it can limit the kinds of innovation that development teams will undertake in the future. deploying software is hard, time-consuming, and requires resources from another team, then developers will often build everything into the existing application in order to avoid suffering the new deployment penalty.

Push-to-deploy systems like **Heroku** have shown developers what the world can look like if you are in control of most of your dependencies as well as your application. Talking with developers about deployment will often turn up discussions of how easy that world is. If you're an operations engineer you've probably heard complaints about how much slower your internal systems are compared with deploying on Heroku.

Docker doesn't try to be Heroku, but it provides a clean separation of responsibilities and encapsulation of dependencies, which results in a similar boost in productivity. It also allows even more fine-grained control than Heroku by putting developers in control of everything, down to the O

distribution on which they ship their application.

As a company, Docker preaches an approach of “batteries included but removable.” Which means that they want their tools to come with everything most people need to get the job done, while still being built from interchangeable parts that can easily be swapped in and out to support custom solutions.

By using an image repository as the hand-off point, Docker allows the responsibility of building the application image to be separated from the deployment and operation of the container.

What this means in practice is that development teams can build their application with all of its dependencies, run it in development and test environments, and then just ship the exact same bundle of application and dependencies to production. Because those bundles all look the same from the outside, operations engineers can then build or install standard tooling to deploy and run the applications. The cycle described in [Figure 2-1](#) then looks somewhat like this (illustrated in [Figure 2-2](#)):

1. Developers build the Docker image and ship it to the registry.
2. Operations engineers provide configuration details to the container and provision resources.
3. Developers trigger deployment.

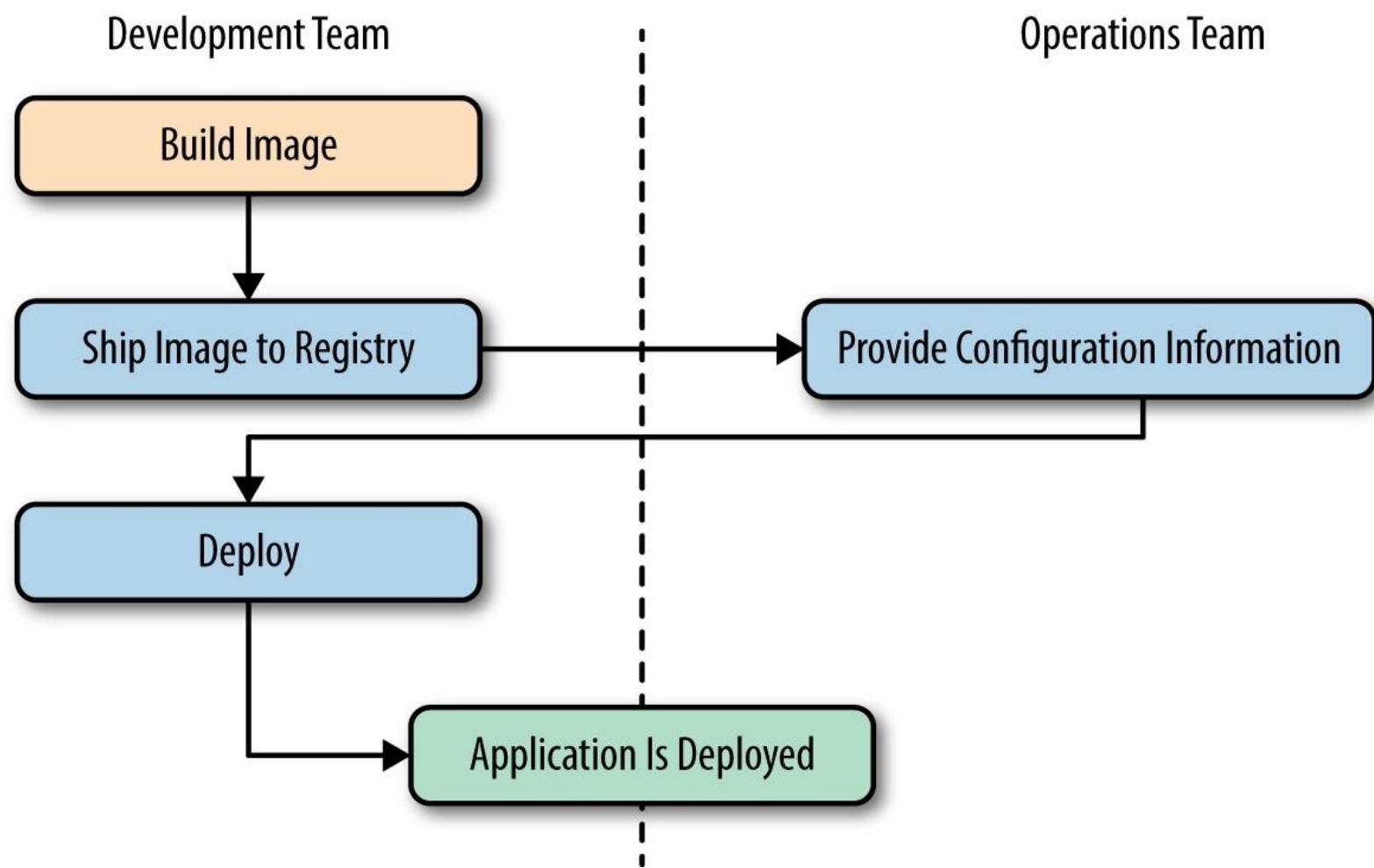


Figure 2-2. A Docker deployment workflow

This is possible because Docker allows all of the dependency issues to be discovered during the

development and test cycles. By the time the application is ready for first deployment, that work is done. And it usually doesn't require as many handovers between the development and operations teams. That's a lot simpler and saves a lot of time. Better yet, it leads to more robust software through testing of the deployment environment before release.

## Broad Support and Adoption

Docker is increasingly well supported, with the majority of the large public clouds announcing at least some direct support for it. For example, Docker runs on AWS Elastic Beanstalk, Google AppEngine, IBM Cloud, Microsoft Azure, Rackspace Cloud, and many more. At DockerCon 2014, Google's Eric Brewer announced that Google would be supporting Docker as its primary internal container format. Rather than just being good PR for these companies, what this means for the Docker community is that a lot of money is starting to back the stability and success of the Docker platform.

Further building its influence, Docker's containers are becoming the common format among cloud providers, offering the potential for "write once, run anywhere" cloud applications. When Docker released their `libswarm` development library at DockerCon 2014, an engineer from Orchard demonstrated deploying a Docker container to a heterogeneous mix of cloud providers at the same time. This kind of orchestration has not been easy before, and it seems likely that as these major companies continue to invest in the platform, the support and tooling will improve correspondingly.

So that covers Docker containers and tooling, but what about OS vendor support and adoption? The Docker client runs directly on most major operating systems, but because the Docker server uses Linux containers, it does not run on non-Linux systems. Docker has traditionally been developed on the Ubuntu Linux distribution, but most Linux distributions and other major operating systems are now supported where possible.

Docker is barely two years old, but with such broad support across many platforms, there is a lot of hope for its continued growth into the future.

## Architecture

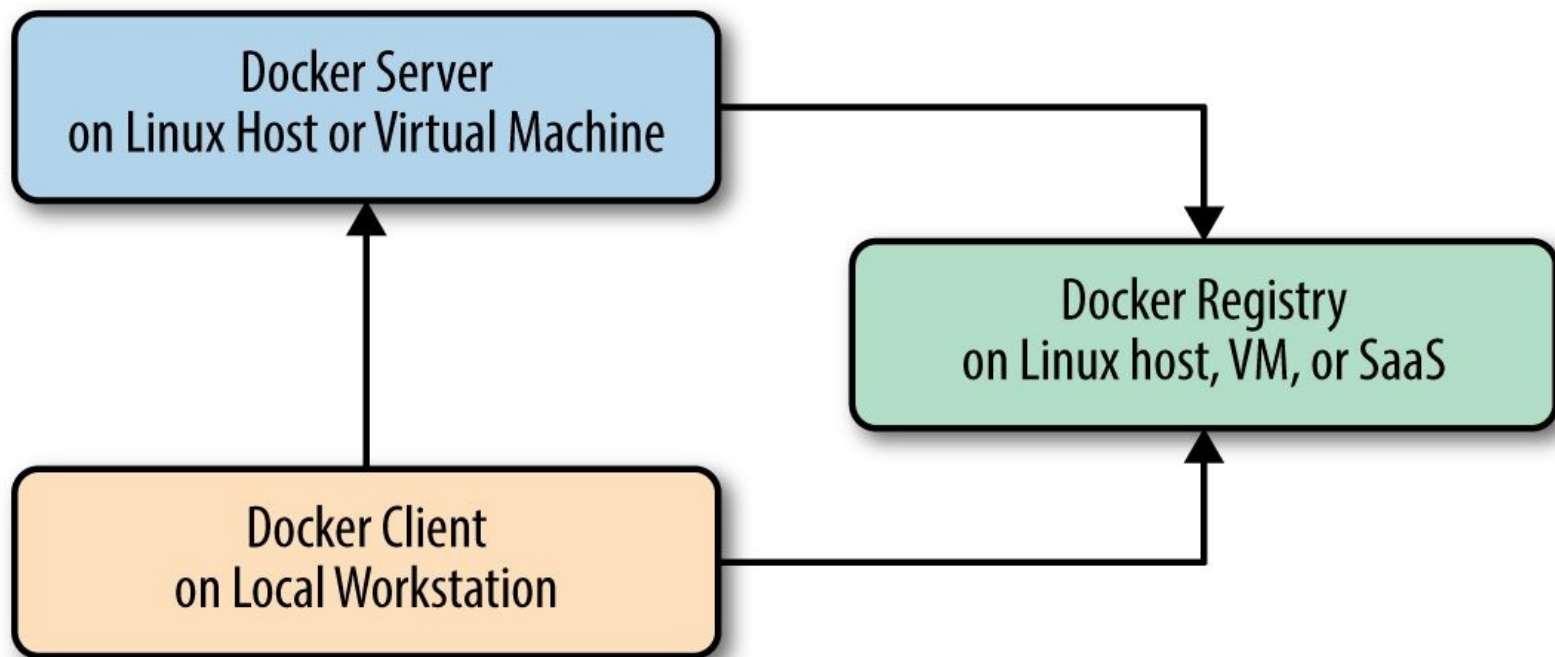
Docker is a powerful technology, and that often means something that comes with a high level of complexity. But the fundamental architecture of Docker is a simple client/server model, with only one executable that acts as both components, depending on how you invoke the `docker` command.

Underneath this simple exterior, Docker heavily leverages kernel mechanisms such as iptables, virtual bridging, cgroups, namespaces, and various filesystem drivers. We'll talk about some of these in [Chapter 10](#). For now, we'll go over how the client and server work and give a brief introduction to the network layer that sits underneath a Docker container.

### Client/Server Model

Docker consists of at least two parts: the client and the server/daemon (see [Figure 2-3](#)). Optionally there is a third component called the registry, which stores Docker images and metadata about those images. The server does the ongoing work of running and managing your containers, and you use the

client to tell the server what to do. The Docker **daemon** can run on any number of servers in the infrastructure, and a single client can address any number of servers. Clients drive all of the communication, but Docker servers can talk directly to image registries when told to do so by the client. Clients are responsible for directing servers what to do, and servers focus on hosting containerized applications.



*Figure 2-3. Docker client/server model*

Docker is a little different in structure from some other client/server software. Instead of having separate client and server executables, it uses the same binary for both components. When you install Docker, you get both components, but the server will only launch on a supported Linux host. Launching the Docker server/daemon is as simple as running `docker` with the `-d` command-line argument, which tells it to act like a daemon and listen for incoming connections. Each Docker host will normally have one Docker daemon running that can manage a number of containers. You can then use the `docker` command-line tool `client` to talk to the server.

## Network Ports and Unix Sockets

The `docker` command-line tool and `docker -d` daemon talk to each other over network sockets. You can choose to have the Docker daemon listen on one or more TCP or Unix sockets. It's possible, for example, to have Docker listen on both a local Unix socket and two different TCP ports (encrypted and nonencrypted). On many Linux distributions, that is actually the default. If you want to only be able to access Docker from the local system, listening only on the Unix socket would be the most secure option. However, most people want to talk to the `docker` daemon remotely, so it usually listens on at least one TCP port.

The original TCP port that `docker` was configured to use was 4243, but that port was never registered and in fact was already used by other tools such as the Mac OS X backup client `CrashPlan`. As a result

Docker registered its own TCP port with IANA and it's now generally configured to use TCP port 2375 when running un-encrypted, or 2376 when handling encrypted traffic. In Docker 1.3 and later, the default is to use the encrypted port on 2376, but this is easily configurable. The Unix socket is located in different paths on different operating systems, so you should check where yours is located. If you have strong preferences, you can usually specify this at install time. If you don't, then the defaults will probably work for you.

## Robust Tooling

Among the many things that have led to Docker's growing adoption is its simple and powerful tooling. This has been expanding ever wider since its initial release by Docker, and by the Docker community at large. The tooling that Docker ships with supports both building Docker images and basic deployment to individual Docker daemons, as well as all the functionality needed to actually manage a remote Docker server. Community efforts have focused on managing whole fleets (or clusters) of Docker servers and the scheduling and orchestrating of container deployments. Docker has also launched its own orchestration toolset, including **Compose** (previously known as Fig), **Machine**, and **Swarm**, which promises to eventually create a cohesive deployment story across environments.

Because Docker provides both a command-line tool and a remote web API, it is easy to add additional tooling in any language. The command-line tool lends itself well to scripting, and a lot of power can easily be leveraged with simple shell script wrappers around the command-line tool.

## Docker Command-Line Tool

The command-line tool `docker` is the main interface that most people will have with Docker. This is a **Go program** that compiles and runs on all common architectures and operating systems. The command-line tool is available as part of the main Docker distribution on various platforms and also compiles directly from the Go source. Some of the things you can do with the Docker command-line tool include, but are not limited to:

- Build a container image.
- Pull images from a registry to a Docker daemon or push them up to a registry from the Docker daemon.
- Start a container on a Docker server either in the foreground or background.
- Retrieve the Docker logs from a remote server.
- Start a command-line shell inside a running container on a remote server.

You can probably see how these can be composed into a workflow for building and deploying. But the Docker command-line tool is not the only way to interact with Docker, and it's not necessarily the most powerful.

## Application Programming Interface (API)

Like many other pieces of modern software, the Docker daemon has a remote API. This is in fact what the Docker command-line tool uses to communicate with the daemon. But because the API is documented and public, it's quite common for external tooling to use the API directly. This enables all manners of tooling, from mapping deployed Docker containers to servers, to automated deployments, to distributed schedulers. While it's very likely that beginners will not initially want to talk directly to the Docker API, it's a great tool to have available. As your organization embraces Docker over time, it's likely that you will increasingly find the API to be a good integration point for this tooling.

Extensive documentation for the [API](#) is on the Docker site. As the ecosystem has matured, robust implementations of Docker API [libraries](#) have begun to appear for many popular languages. We've used the Go and Ruby libraries, for example, and have found them to be both robust and rapidly updated as new versions of Docker are released.

Most of the things you can do with the Docker command-line tooling is supported relatively easily via the API. Two notable exceptions are the endpoints that require streaming or terminal access: running remote shells or executing the container in interactive mode. In these cases, it's often easier to use the command-line tool.

## Container Networking

Even though Docker containers are largely made up of processes running on the host system itself, they behave quite differently from other processes at the network layer. If you think of each of your Docker containers as behaving on the network like a host on a private network, you'll be on the right path. The Docker server acts as a virtual bridge and the containers are clients behind it. A bridge is just a network device that repeats traffic from one side to another. So you can think of it like a mini virtual network with hosts attached.

The implementation is that each container has its own virtual Ethernet interface connected to the Docker bridge and its own IP address allocated to the virtual interface. Docker lets you bind ports on the host to the container so that the outside world can reach your container. That traffic passes over a proxy that is also part of the Docker daemon before getting to the container. See [Chapter 10](#) for more detailed information.

Docker allocates the private subnet from an unused [RFC 1918](#) private subnet block. It detects which network blocks are unused on startup and allocates one to the virtual network. That is bridged to the host's local network through an interface on the server called `docker0`. This means that all of the containers are on a network together and can talk to each other directly. But to get to the host or the outside world, they go over the `docker0` virtual bridge interface. As we mentioned, inbound traffic goes over the proxy. This proxy is fairly high performance but can be limiting if you run high throughput applications in containers. We talk more about this as well as other networking topics in [Chapter 10](#), and offer some solutions.

There is a dizzying array of ways in which you can configure Docker's network layer, from allocating your own network blocks to configuring your own custom bridge interface. People often run with the default mechanisms, but there are times when something more complex or specific to your application is required. You can find much more detail about Docker networking in its [documentation](#), and we



- [download No Ordinary Joe: The Autobiography of the Greatest British Boxer of Our Time](#)
- [read Yeah! Yeah! Yeah!: The Beatles, Beatlemania, and the Music that Changed the World pdf, azw \(kindle\), epub](#)
- [Morgoth's Ring: The Later Silmarillion, Part 1 \(The History of Middle-Earth, Book 10\) pdf, azw \(kindle\), epub](#)
- [Clinical Microbiology Made Ridiculously Simple \(3rd Edition\) here](#)
- [download online LSD, mein Sorgenkind for free](#)
  
- <http://redbuffalodesign.com/ebooks/Naked-Pictures-of-Famous-People.pdf>
- <http://creativebeard.ru/freebooks/Yeah--Yeah--Yeah---The-Beatles--Beatlemania--and-the-Music-that-Changed-the-World.pdf>
- <http://drmurphreesnewsletters.com/library/The-Accounting-Game--Basic-Accounting-Fresh-from-the-Lemonade-Stand.pdf>
- <http://www.mmastyles.com/books/Fatal-Error--Repairman-Jack--Book-14-.pdf>
- <http://twilightblogs.com/library/LSD--mein-Sorgenkind.pdf>