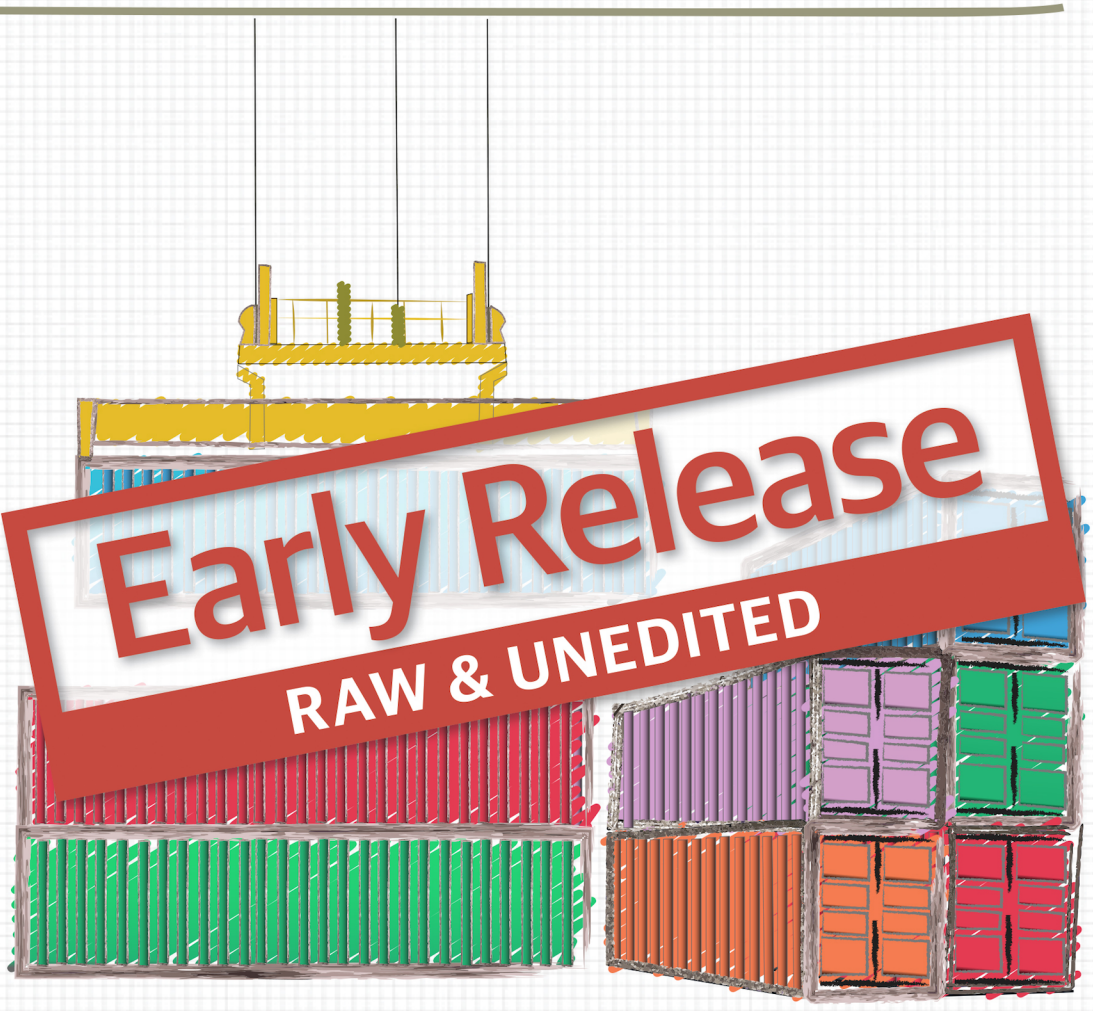
Docker in the Trenches

SUCCESSFUL PRODUCTION DEPLOYMENT



Written by: Joe Johnston, Antoni Batchelli, Justin Cormack
John Fiedler, Milos Gajdos

BLEEDING EDGE PRESS

Docker in the Trenches Early Release

Successful Production Deployment

Joe Johnston, Antoni Batchelli, Justin Cormace, John Fiedler, Milos Gajdos

Docker in the Trenches Early Release

ISBN 9781939902221
Published by: Bleeding Edge Press, Santa Rosa, CA 95404
Title: Docker in the Trenches
Authors: Joe Johnston, Antoni Batchelli, Justin Cormace, John Fiedler, Milos Gajdos
Editor: Troy Mott
Copy Editor: Christina Rudloff
Cover Design: Bob Herbstman
Website: bleedingedgepress.com

Table of Contents

Preface

Docker is the new sliced bread of infrastructure. Few emerging technologies compare to how fast it swept the DevOps and infrastructure scenes. In less than two years, Google, Amazon, Microsoft, IBM, and nearly every cloud provider announced support for running Docker containers. Dozens of Docker related startups were funded by venture capital in 2014 and early 2015. Docker, Inc., the company behind the namesake open source technology, was valued at about $1 billion USD during their Series D funding round in Q1 2015.

Companies large and small are converting their apps to run inside containers with an eye towards service oriented architectures (SOA) and microservices. Attend any DevOps meet-up from San Francisco to Berlin or peruse the hottest company engineering blogs, and it appears the ops leaders of the world now run on Docker in the cloud.

No doubt, containers are here to stay as crucial building blocks for application packaging and infrastructure automation. But there is one thorny question that nagged this book's authors and colleagues to the point of motivating another Docker book:

Who is actually using Docker in production?

Or more poignantly, how does one navigate the hype to successfully address real world production issues with Docker? This book sets out to answer these questions through a mix of interviews, end-to-end production examples from real companies, and referable topic chapters from leading DevOps experts. Although this book contains useful examples, it is not a copy-and-paste "how-to" reference. Rather, it focuses on the practical theories and experience necessary to evaluate, derisk and operate bleeding-edge technology in production environments.

As authors, we hope the knowledge contained in this book will outlive the code snippets by providing a solid decision tree for teams evaluating how and when to adopt Docker related technologies into their DevOps stacks.

Running Docker in production gives companies several new options to run and manage server-side software. There are many readily available use cases on how to use Docker, but few companies have publicly shared their full-stack production experiences. This book is a compilation of several examples of how the authors run Docker in production as well as a select group of companies kind enough to contribute their experience.

Who is this book for?

Readers with intermediate to advanced DevOps and ops backgrounds will likely gain the most from this book. Previous experience with both the basics of running servers in production as well as the basics of creating and managing containers is highly recommended. Many books and blog posts already cover individual topics related to installing and running Docker, but few resources exist to weave together the myriad and sometimes forehead-to-wall-thumping concerns of running Docker in production. But fear not, if you enjoyed the movie Inception, you will feel right at home running containers in VMs on servers in the cloud.

Why Docker?

The underlying container technology used by Docker has been around for many years before **dotcloud**, the Platform-as-a-Service startup, pivoted to become Docker as we now know it. Before dotCloud, many notable companies like **Heroku** and **Iron.io** were running large scale container clusters in production for added performance benefits over virtual machines. Running software in containers instead of virtual machines gave these companies the ability to spin up and down instances in seconds instead of minutes, as well as run more instances on fewer machines.

So why did Docker take off if the technology wasn't new? Mainly, ease of use. Docker created a unified way to package, run, and maintain containers from convenient CLI and HTTP API tools. This simplification lowered the barrier to entry to the point where it became feasible--and fun--to package applications and their runtime environments into self-contained images rather than into configuration management and deployment systems like Chef, Puppet, Capistrano, etc.

Fundamentally, Docker changed the interface between developer and DevOps teams by providing a unified means of packaging the application and runtime environment into one simple Dockerfile. This radically simplified the communication requirements and boundary of responsibilities between devs and DevOps.

Before Docker, epic battles raged within companies between devs and ops. Devs wanted to move fast, integrate the latest software and dependencies, and deploy continuously. Ops were on call and needed to ensure things remained stable. They were the gatekeepers of what ran in production. If ops was not comfortable with a new dependency or requirement, they often ended up in the obstinate position of restricting developers to older software to ensure bad code didn't take down an entire server.

In one fell swoop, Docker changed the roll of DevOps from a "mostly say no" to a "yes, if it runs in Docker" position where bad code only crashes the container, leaving other services unaffected on the same server. In this paradigm, DevOps are effectively responsible for providing a PaaS to developers, and developers are responsible for making sure their code

runs as expected. Many teams are now adding developers to PagerDuty to monitor their own code in production, leaving DevOps and ops to focus on platform uptime and security.

Development vs. production

For most teams, the adoption of Docker is being driven by developers wanting faster iterations and release cycles. This is great for development, but for production, running multiple Docker containers per host can pose security challenges covered in the [Security chapter](10 Security.md). In fact, almost all conversations about running Docker in production are dominated by two concerns that separate development environments from production: 1) orchestration and 2) security.

Some teams try to mirror development and production environments as much as possible. This approach is ideal but often not practical due to the amount of custom tooling required or the complexity of simulating cloud services (like AWS) in development.

To simplify the scope of this book, we cover use cases for deploying code but leave the exercise of determining the best development setup to the reader. As a general rule, always try to keep production and development environments as similar as possible and use a continuous integration / continuous deliver (CI/CD) system for best results.

What we mean by Production

Production means different things to different teams. In this book, we refer to production as the environment that runs code for real customers. This is in contrast to development, staging, and testing environments where downtime is not noticed by customers.

Sometimes Docker is used in production for containers that receive public network traffic, and sometimes it is used for asynchronous, background jobs that process workloads from a queue. Either way, the primary difference between running Docker in production vs. any other environment is the additional attention that must be given to security and stability.

A motivating driver for writing this book was the lack of clear distinction between actual production and other envs in Docker documentation and blog posts. We wagered that four out of five Docker blog posts would recant (or at least revise) their recommendations after attempting to run in production for six months. Why? Because most blog posts start with idealistic examples powered by the latest, greatest tools that often get abandoned (or postponed) in favor of simpler methods once the first edge case turns into a showstopper. This is a reflection on the state of the Docker technology ecosystem more than it is a flaw of tech bloggers.

Bottom line, production is hard. Docker makes the work flow from development to production much easier to manage, but it also complicates security and orchestration.

To save you time, here are the cliff notes of this book.

All teams running Docker in production are making one or more concessions on traditional security best practices. If code running inside a container can not be fully trusted, a one-to-one container to virtual machine topology is used. The benefits of running Docker in production outweigh security and orchestration issues for many teams. If you run into a tooling issue, wait a month or two for the Docker community to fix it rather than wasting time patching someone else's tool. Keep your Docker setup as minimal as possible. Automate everything. Lastly, you probably need full-blown orchestration (Mesos, Kubernetes, etc.) a lot less than you think.

Batteries included vs. composable tools

A common mantra in the Docker community is "batteries included but removable." This refers to monolithic binaries with many features bundled in as opposed to the traditional Unix philosophy of smaller, single purpose, pipeable binaries.

The monolithic approach is driven by two main factors: 1) desire to make Docker easy to use out of the box, 2) golang's lack of dynamic linking. Docker and most related tools are written in Google's **Go programming language**, which was designed to ease writing and deploying highly concurrent code. While Go is a fantastic language, its use in the Docker ecosystem has caused delays in arriving at a pluggable architecture where tools can be easily swapped out for alternatives.

If you are coming from a Unix sysadmin background, your best bet is to get comfortable compiling your own stripped down version of the `docker` daemon to meet your production requirements. If you are coming from a dev background, expect to wait until Q3/Q4 of 2015 before Docker plugins are a reality. In the meantime, expect tools within the Docker ecosystem to have significant overlap and be mutually exclusive in some cases.

In other words, half of your job of getting Docker to run in production will be deciding on which tools make the most sense for your stack. As with all things DevOps, start with the simplest solution and add complexity only when absolutely required.

As of May, 2015, Docker, Inc., released **Compose**, **Machine**, and **Swarm** that compete with similar tools within the Docker ecosystem. All of these tools are optional and should be evaluated on merit rather than assumption that the tools provided by Docker, Inc., are the best solution.

Another key piece of advice in navigating the Docker ecosystem is to evaluate each open source tool's funding source and business objective. Docker, Inc., and **CoreOS** are frequently releasing tools at the moment to compete for mind and market share. It is best to wait a few months after a new tool is released to see how the community responds rather than switch to the latest, greatest tool just because it seems cool.

What not to dockerize

Last but not least, expect to not run everything inside a Docker container. Heroku-style **12 factor** apps are the easiest to Dockerize since they do not maintain state. In an ideal microservices environment, containers can start and stop within milliseconds without impacting the health of the cluster or state of the application.

There are startups like **ClusterHQ** working on Dockerizing databases and stateful apps, but for the time being, you will likely want to continue running databases directly in VMs or bare metal due to orchestration and performance reasons.

Any app that requires dynamic resizing of CPU and memory requirements is not yet a good fit for Docker. There is work being done to allow for dynamic resizing, but it is unclear when this will become available for general production use. At the moment, resizing a container's CPU and memory limitations requires stopping and restarting the container.

Also, apps that require high network throughput are best optimized without Docker due to Docker's use of iptables to provide NAT from the host IP to container IPs. It is possible to disable Docker's NAT and improve network performance, but this is an advanced use case with few examples of teams doing this in production.

Authors

Joe Johnston is a full-stack developer, entrepreneur, and advisor to various startups and enterprises in San Francisco, and is focused on Docker, microservices, and HTML5 apps.

Justin Cormack is a consultant especially interested in the opportunities for innovation made available by open source software, the web development model, and the cloud, and is a NETBSD committer.

John Fiedler is the Director of Engineering at RelateIQ.

Antoni Batchelli is the Vice President of Engineering at PeerSpace and Founder of PalletOps.

Milos Gajdos is a consultant, UK Ministry of Justice, and is an Infrastructure Tsar at Infrahackers Ltd.

Getting Started 1

The first task of setting up a Docker production system is to understand the terminology in a way that helps visualize how components fit together. As with any rapidly evolving technology ecosystem, it's safe to expect over ambitious marketing, incomplete documentation, and outdated blog posts that lead to a bit of confusion about what tools do what job.

Rather than attempting to provide a unified thesaurus for all things Docker, we'll instead define terms and concepts in this chapter that remain consistent throughout the book. Often, our definitions are compatible with the ecosystem at large, but don't be too surprised if you come across a blog post that uses terms differently.

In this chapter, we'll introduce the core concepts of running Docker, and containers in general, in production without actually picking specific technologies. In subsequent chapters, we'll cover real-world production use cases with details on specific components and vendors.

Terminology

Image vs. Container

- Image is the filesystem snapshot or tarball.
- Container is what we call an image when it is run.

Containers vs. Virtual Machines

- VMs hold complete OS and application snapshots.
- VMs run their own kernel.
- VMs can run OSs other than Linux.
- Containers only hold the application, although the concept of an application can extend to an entire Linux distro.
- Containers share the host kernel.
- Containers can only run Linux, but each container can contain a different distro and still run on the same host.

CI/CD: Continuous Integration / Continuous Delivery

System for automatically building new images and deploying them whenever application new code is committed or upon some other trigger.

Host Management

The process for setting up--provisioning--a physical server or virtual machine so that it's ready to run Docker containers.

Orchestration

This term means many different things in the Docker ecosystem. Typically, it encompasses scheduling and cluster management but sometimes also includes host management.

In this book we use *orchestration* as a loose umbrella term that encompasses the process of scheduling containers, managing clusters, linking containers (discovery), and routing network traffic. Or in other words, orchestration is the controller process that decides where containers should run and how to let the cluster know about the available services.

Scheduling

Deciding which containers can run on which hosts given resource constraints like CPU, memory, and IO.

Discovery

The process of how a container exposes a service to the cluster and discovers how to find and communicate with other services. A simple use case is a web app container discovering how to connect to the database service.

Docker documentation refers to *linking* containers, but production grade systems often utilize a more sophisticated discovery mechanism.

Configuration Management

Configuration management is often used to refer to pre-Docker automation tools like Chef and Puppet. Most DevOps teams are moving to Docker to eliminate many of the complications of configuration management systems.

In many of the examples in this book, configuration management tools are only used to provision hosts with Docker and very little else.

Development to Production

This book focuses on Docker in production--non-development environments, which means we will spend very little time on configuring and running Docker in development. But since all servers run code, it is worth a brief discussion on how to think about application code in a Docker vs non-Docker system.

Unlike traditional configuration management systems like Chef, Puppet, Ansible, etc., Docker is best utilized when application code is pre-packaged into a Docker image. The image typically contains all the application code as well as any runtime dependencies and system requirements. Configuration files containing database credentials and other secrets are often added to the image at runtime rather than being built into the image.

Some teams choose to manually build Docker images on dev machines and push them to image repositories that are used to pull images down onto production hosts. This is the simple use case. It works, but is not ideal due to workflow and security concerns.

A more common production example is to use a CI/CD system to automatically build new images whenever application code or Dockerfiles change.

Multiple Ways to Use Docker

Over the years, technology has changed significantly from physical servers to virtual servers to clouds with platform-as-a-service (PaaS) environments. Docker images can be used in current environments without heavy lifting or with completely new architectures. It is not necessary to immediately migrate from a monolithic application to a service oriented architecture to use Docker. There are many use cases which allow for Docker to be integrated at different levels.

A few common Docker uses:

- Replacing code deployment systems like Capistrano with image-based deployment.
- Safely running legacy and new apps on the same server.
- Migrating to service oriented architecture over time with one toolchain.
- Managing horizontal scalability and elasticity in the cloud or on bare metal.
- Ensuring consistency across multiple environments from development to staging to production.
- Simplifying developer machine setup and consistency.

Migrating an app's background workers to a Docker cluster while leaving the web servers and database servers alone is a common example of how to get started with Docker. Another example is migrating parts of an app's REST API to run in Docker with a Nginx proxy in front to route traffic between legacy and Docker clusters. Using techniques like these allows teams to seamlessly migrate from a monolithic to a service oriented architecture over time.

Today's applications often require dozens of third-party libraries to accelerate feature development or connect to third-party SaaS and database services. Each of these libraries introduces the possibility of bugs or dependency versioning hell. Then add in frequent library changes and it all creates substantial pressure to deploy working code consistently without failure on infrastructure.

Docker's golden image mentality allows teams to deploy working code--either monolithic, service oriented, or hybrid---in a way that is testable, repeatable, documented, and consistent for every deployment due to bundling code and dependencies in the same image. Once an image is built, it can be deployed to any number of servers running the Docker daemon.

Another common Docker use case is deploying a single container across multiple environments, following a typical code path from development to staging to production. A container allows for a consistent, testable environment throughout this code path.

As a developer, the Docker model allows for debugging the exact same code in production on a developer laptop. A developer can easily download, run, and debug the problematic production image without needing to first modify the local development environment.

What to Expect

Running Docker containers in production is difficult but achievable. More and more companies are starting to run Docker in production everyday. As with all infrastructure, start small and migrate over time.

Why is Docker in production difficult?

A production environment will need bulletproof deployment, health checks, minimal or zero downtime, the ability to recover from failure (rollback), a way to centrally store logs, a way to profile or instrument the app, and a way to aggregate metrics for monitoring. Newer technologies like Docker are fun to use but will take time to perfect.

Docker is extremely useful for portability, consistency, and packaging services that require many dependencies. Most teams are forging ahead with Docker due to one or more pain points:

- Lots of different dependencies for different parts of an app.
- Support of legacy applications with old dependencies.
- Workflow issues between devs and DevOps.

Out of the teams we interviewed for this book, there was a common tale of caution around trying to adopt Docker in one fell swoop within an organization. Even if the ops team is fully ready to adopt Docker, keep in mind that transitioning to Docker often means pushing the burden of managing dependencies to developers. While many developers are begging for this self-reliance since it allows them to iterate faster, not every developer is capable or interested in adding this to their list of responsibilities. It takes time to migrate company culture to support a good Docker workflow.

The Stack 2

Every production Docker setup includes a few basic architectural components that are universal to running server clusters--both containerized and traditional. In many ways, it is easiest to initially think about building and running containers in the same way you are currently building and running virtual machines but with a new set of tools and techniques.

1. Build and snapshot an image.
2. Upload image to repository.
3. Download image to a host.
4. Run the image as a container.
5. Connect container to other services.
6. Route traffic to the container.
7. Ship container logs somewhere.
8. Monitor container.

Unlike VMs, containers provide more flexibility by separating hosts (bare metal or VM) from applications services. This allows for intuitive improvements in building and provisioning flows, but it comes with a bit of added overhead due to the additional nested layer of containers.

The typical Docker stack will include components to address each of the following concerns:

- Build system
- Image repository
- Host management
- Configuration management
- Deployment
- Orchestration
- Logging
- Monitoring

Build System

- How do images get built and pushed to the image repo?
- Where do Dockerfiles live?

There are two common ways to build Docker images:

1. Manually build on a developer laptop and push to a repo.
2. Automatically build with a CI/CD system upon a code push.

The ideal production Docker environments will use a CI/CD (Configuration Integration / Continuous Deployment) system like Jenkins, Codeship, etc. to automatically build images when code is pushed. Once the container is built, it is sent to an image repo where the automated test system can download and run it.

Image Repository

- Where are Docker images stored?

The current state of Docker image repos is less than reliable, but getting better every month. Docker's **hosted image repo hub** is notoriously unreliable, requiring additional retries and failsafe measures. Most teams will likely want to run their own image repo on their own infrastructure to minimize network transfer costs and latencies.

Host Management

- How are hosts provisioned?
- How are hosts upgraded?

Since Docker images contain the app and dependencies, host management systems typically just need to spin up new servers, configure access and firewalls, and install the Docker daemon.

Services like Amazon's **EC2 Container Service** eliminate the need for traditional host managment.

Configuration Management

- How do you define clusters of containers?
- How do you handle run time configuration for hosts and containers?
- How do you manage keys and secrets?

As a general rule, avoid traditional configuration management as much as possible. It is added complexity that often breaks. Use tools like **Ansible**, **SaltStack**, **Chef**, **Puppet**, etc.

only to provision hosts with the Docker daemon. Try to get rid of reliance on your old configuration management systems as much as possible and move towards self-configured containers using the discovery and clustering techniques in this book.

Deployment

- How do you get the container onto the host?

There are two basic methods of image deployment:

1. **Push** - deployment or orchestration system pushes an image to the relevant hosts.
2. **Pull** - image is pulled from image repo in advance or on demand.

Orchestration

- How do you organize containers into clusters?
- What servers do you run the containers on?
- How do you schedule server resources?
- How do you run containers?
- How do you route traffic to containers?
- How do you enable containers to expose and discover services?

Orchestration = duct tape. At least most of the time.

There are many early stage, full-featured container orchestration systems like **Docker Swarm**, **Kubernetes**, **Mesos**, **Flynn**, etc. These are often overkill for most teams due to the added complexity of debugging when something goes wrong in production. Deciding on what tools to use for orchestration is often the hardest part of getting up and running with Docker.