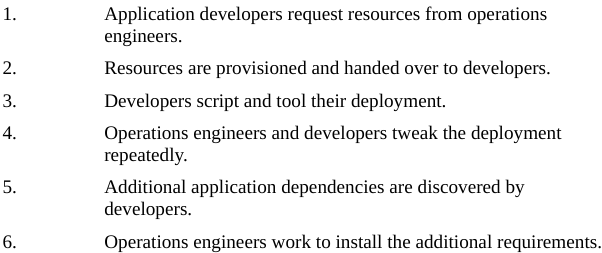
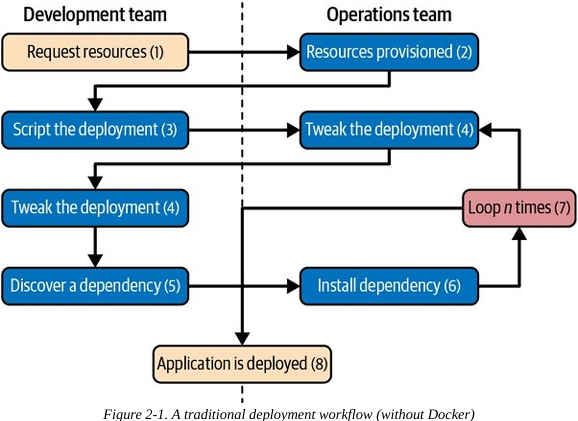
Chapter 2: The Docker Landscape

# 1. Process Simplification

-Traditionally, the cycle of getting an app to production often looks:







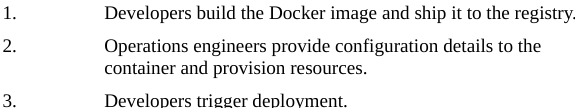
-Following traditional processes to deploy brand-new app into production can take a week for a complex new system. -> not productive, lots of effort and communication between teams of people, technically challenging and expensive, limit the kinds of innovation development teams will undertake in future.

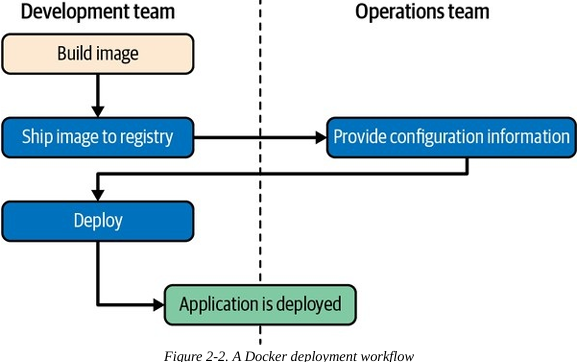
-Push-to-deploy systems like Heroku (<https://www.heroku.com/>) show what the world can look like if you are in control of your app and a majority of your dependencies. -> you might heard how much slower your internal system are compared with push-button solutions.

-Heroku is a whole env, not just a container engine. Docker provides a clean separation of responsibilities and encapsulation of dependencies, which results in a similar boost in productivity. It put devs in control of everything. Some tooling and orchestrators built on top of Docker (Kubernetes, Docker Swarm mode, Mesos) aim to replicate the simplicity of Heroku. However, a simple platform uses only Docker still provides all core process benefits.

-Docker adopt: “batteries included but removable”: its tools come with everything most people need to get the job done while still being built from interchangeable parts that can swapped and out to support custom solutions.

-Using an image repository, Docker allows the responsibility of building app image to be separated from deployment and operation of container. Dev teams can build their app with all of its dependencies, run it in development and test environment, and then just ship the exact same bundle of app and dependencies to production. Those bundles look the same from outside, operation engineers can then build or install standard tooling to deploy and run apps. That’s possible as Docker allows all of dependency issues to be discovered during development and test cycles. When app ready for 1st deploy, that work already been done. It doesn’t require many handoff between dev and op team, this can alleviate the need for anyone other than dev team to be involved in creation and deployment of a new service -> Simple, save lots of time, more robust software through testing of deployment environment before release.





# 2. Broad Support and Adoption

-Docker is well supported with the majority of large public clouds offering support

+Amazon Elastic Container Service (Amazon ECS), Amazon Elastic Kubernetes Service (Amazon EKS), Amazon Fargate, Amazon Elastic Beanstalk.

+Google App Engine (GAE), Google Kubernetes Engine, Read Hat OpenShift, IBM Cloud, Microsoft Azure.

-To gain some goodwill and support wider adoption in marketplace, Docker decided to help sponsor Open Container Initiative (OCI) in June 2015. The 1st full specification from that effort was released in July 2017 and was based in large part on version 2 of Docker image format. It’s now possible to apply for OCI certification for both container images and container runtimes.

-This is the primary high-level OCI-certified runtime:

+containerd: default high-level runtime in modern versions of Docker and Kubernetes.

-These lower-level OCI-certified runtimes can be used by containerd to manage and create containers:

+runc: used as default lower-level runtime by containerd

+crun: written in C and designed to be fast and have a small memory footprint.

+Kata Container: from Intel, Hyper, OpenStack Foundation is a virtualized runtime that can run a mix of containers and virtual machines

+gVisor: from Google, a sandboxed runtime, implemented entirely in user space.

+Nabla Containers: provide another sandboxed runtime designed to reduce the attack surface of Linux containers

# 3. Architecture

-Docker is a powerful technology that often indicates both tools and processes that come with a high level of complexity. But its fundamental structure is a simple client/server model. Several pieces are sitting behind Docker API (containerd, runc…) but the basic system interaction is a client talking over an API to server. Underneath it, Docker leverages kernel mechanisms.

## 3.1 Client/Server model

A diagram of a server

AI-generated content may be incorrect.

-Docker consists of 2 parts: the client and server/daemon. Optionally the 3rd component registry which store Docker images and their metadata.

+Server does the work of building, running, managing your containers. The Docker daemon can run on any number of servers in infrastructure, and a single client can address any number of servers.

+Clients drive all of communication, but Docker servers can talk directly to image registries when told to do so by client. Clients are responsible for telling servers what to do, and servers focusing on hosting and managing containerized apps.

+Docker has a **docker client** and **dockerd server**, the server then orchestrates a few other components behing the scenes on behalf of client, including containerd-shim-runc-v2.

-Each Docker host will have one Docker server running that can manage containers. You can use docker cmd tool to talk to server from the server itself or remote client.

## 3.2 Network Ports and Unix Sockets

-docker cmd tool and dockerd daemon can talk to each other over **Unix sockets** and **network ports**. Docker registered 3 ports for use by daemon and client:

+TCP port 2375 for unencrypted traffic

+TCP port 2376 for encrypted SSL connection

+TCP port 2377 for Docker Swarm mode.

-The default setting for Docker installer is only use Unix socket for communication with local Docker daemon. The Unix socket located in different paths on different OS, but in most cases: /var/run/docker.sock.

## 3.3 Robust Tooling

-Since Docker initial release, the tooling capabilities have been expanding thanks to efforts form Docker community:

+The tooling support building Docker images, basic deployment to individual Dockers daemons, a distributed mode Swarm mode, all functionality needed to manage a remote Docker server.

+Beside, community have focused on managing whole fleets (or clusters) of Docker servers and scheduling and orchestrating container deployments.

-Docker has also launched its own orchestration toolset: Compose (<https://github.com/docker/compose>), Docker Desktop (<https://www.docker.com/products/docker-desktop>), Swarm mode (<https://docs.docker.com/engine/swarm>) which create a cohesive deployment story.

-Docker provides both cmd tools and remote REST API. Cmd tool lends itself well to shell scripting, and anything client can do can also be done via REST API. Docker CLI is so well-known that many other Linux container CLI tools (podman (https://podman.io/), nerdctl (https://github.com/containerd/nerdctl)) mimic its arguments for compatibility and easy adoption.

## 3.4 Docker Command-Line Tool

-cmd tool docker is the main interface that most people will have with Docker. Some things you can do with cmd:

A close up of text

AI-generated content may be incorrect.

## 3.5 Docker Engine API

-Docker daemon has an API-what Docker cmd tool uses to communicate with daemon (<https://dockr.ly/2wxCHnx>)

-Robust implementation of Docker API libraries have emerged for all popular languages: <https://dockr.ly/2wxCHnx>

-Most of things you can do with Docker cmd are supported easily via API. 2 notable exceptions: running remote shells or executing the container in interactive mode.

## 3.6 Container Networking

-Most people run their containers in default configuration: **bridge mode**.

+Think each of your Linux containers like a host on a private network. Docker server acts as a virtual bridge, containers are clients behind it. A bridge is a network device that repeats traffic from one side to another. You can think of it like a mini virtual network, which each container acting like a host attached to that network.

A diagram of a network

AI-generated content may be incorrect.

-The actual implementation: each container has a virtual Ethernet interface connected to Docker bridge and an IP address allocated to the virtual interface. Docker lets you bind and expose individual or groups of ports on the host to the container so that the outside world can reach your container on those port. The traffic is managed by vpnkit

(<https://github.com/moby/vpnkit>)

-Docker allocates the private subnets from RFC 1918 private subnet block. It detects which network blocks are unused on host and allocates one of those to virtual network. That is bridged to host’s local network through an interface on server docker0. By default, all containers are on network together and can talk to one another directly. But to get to the host or the outside world, they go over docker0 virtual bridge interface.

# 4. Getting the Most from Docker

-To begin with, Docker’s architecture is aimed at apps that are stateless or where the state is externalized into data stores like databases or caches. Some good apps for beginning with Docker include: web frontends, backend APIs, short-running tasks like maintenance scripts that might be handled by cron.

## 4.1 Containers are not Virtual Machines

-Linux containers are not as virtual machines but as very lightweight wrappers around a single Unix process.

- Containers may come and go much more readily than a traditional VM

+VM are by design a stand-in for real hardware. As a real server is what they’re abstracting, VM are often long-lived in nature. Container might exists for months, or it created run a task for a minute and destroyed.

-If you run Docker on mac or Win, you are leveraging a Linux VM to run dockerd (Docker server). But in Linux, dockerd can run natively, no need for a VM.

A diagram of a container and a container

AI-generated content may be incorrect.

## 4.2 Limited Isolation

## 4.3 Containers are Lightweight

-Creating a new container can take up very little disk space (quick test: container created from an existing image takes 12 KB). A new VM might require hundreds, thousands megabyte. New container just a reference to a layered filesystem image and some metadata about configuration.

-By default, no copy of data is allocated to container. Containers are just processes on existing system that may only need to read information from disk->no need to copy any data for exclusive use of container.

## 4.4 Toward an Immutable Infrastructure

-Handle all app management simply by deploying and redeploying containers to server. When server need update (Docker daemon or Linux kernel), bring up a new server with changes, deploy here and decommission or reinstall the old one. Container-based Linux distributions Red Hat’s Fedora CoreOS designed around this principle (<https://getfedora.org/en/coreos>).

+Because the clean separation between deployment and configuration of servers, many container-based production systems use tools like HashiCorp’s Packer (<https://www.packer.io/intro/index.xhtml>) to build cloud virtual server images, then leverage Docker to avoid configuration management systems.

## 4.5 Stateless Applications

-Example of app that containerizes well is web app that keeps its state in database. Stateless apps are designed to answer a single self-contained request and have no need to track info between requests from one or more clients. If you bake the file configuration into images, it means you’ve limited the reusability of image and made it more challenging to deploy into different env.

-The process of containerizing: move configuration state into environment variables that can be passed to app at run time -> You apply configuration to container when it’s deployed.

## 4.6 Externalizing State

-How do you best store state when you need to? Docker support env variables, and they are stored in metadata that makes up a container configuration. It’s possible to store and retrieve app configuration inside an external datastore: Consul or PostgreSQL.

-Apps that need to store files face some challenges. Storing things to container’s filesystem is not performant, will be limited by space, and will not preserve state when a container is re-created. You can use Amazon Simple Storage Service (Amazon S3), OpenStack Switf, or local block store.

# 5. The Docker Workflow

## 5.1 Revision Control

-Doker 1st gives 2 forms of revision control: track filesystem layers that each Docker image is comprised of, and tagging system for those images

### 5.1.1 Filesystem layers

-Linux containers made up of stacked filesystem layers, each identified by a unique hash, where ach new set of changes made during build process is laid on top of previous changes. When you do a new build, you only have to rebuild the layers that follow the change you’re deploying->save time and bandwidth. Because of layering effect, and Linux containers include all dependencies, with Docker you can be more confident about the changes shipping to production.

-Docker image contains everything required to run app. When you change one line of code, by leveraging the build cache, Docker can ensure that only layers affected by code change are rebuilt.

### 5.1.2 Image tags

-To answer “What was the previous version?”, Docker has a built-in mechanism: image tagging a standard build step. You can leave multiple revision of app on server so that performing a rollback is trivial.

## 5.2 Building

-Docker cmd tool use build flag: consume a Dockerfile and produce Docker image. Each command in Dockerfile generates a new layer in image

-Docker image is a standardized artifact, all of tolling behind the build will be the same regardless of development language or based image or number of layers needed.

-docker image build: generate container image.

## 5.3 Testing

## 5.4 Packaging

-Docker tooling will have to deal with one kind of package: Docker image. Once a container is built, it can easily deployed on any system with a running Docker server one the same architecture.

## 5.5 Deploying

-Deployment are handled by kinds of tools in different shops: shell scripting, Capistrano, Fabric, Ansible

## 5.6 The Docker Ecosystem

### 5.6.1 Orchestration

-The 1st important category of tools that add functionality to core Docker distribution and Linux container experience contains orchestration and mass deployment tools. Fully automatic schedulers like Kubernetes, Apache Mesos with Marathon scheduler are powerful options take complete control of a pool of hosts on your behalf

### 5.6.2 Immutable atomic hosts

### 5.6.3 Additional tools