



Universidad de Oviedo



Allocation



SOFTWARE
ARCHITECTURE

Course 2020/2021

Jose E. Labra Gayo

Allocation

Relationship between Software and its environment
Where does each component run?

Infrastructure?

Deployment?



Allocation

Packaging, distribution and deployment

Software computation options

Execution environments

Continuous delivery and deployment pipeline

Software in production

Software in production patterns

Software in production testing

Logging & Monitoring

Incidents & post-mortem

Chaos engineering

Packaging, distribution and deployment

Packaging

Create an executable from source code

A typical package consists of:

- Compiled code

 - Even for interpreted languages:

 - Transpiled, obfuscated & minimized

- Configuration files

 - Environment variables

 - Credentials, etc.

- Libraries & dependencies

- User manuals & docs

- Installation scripts



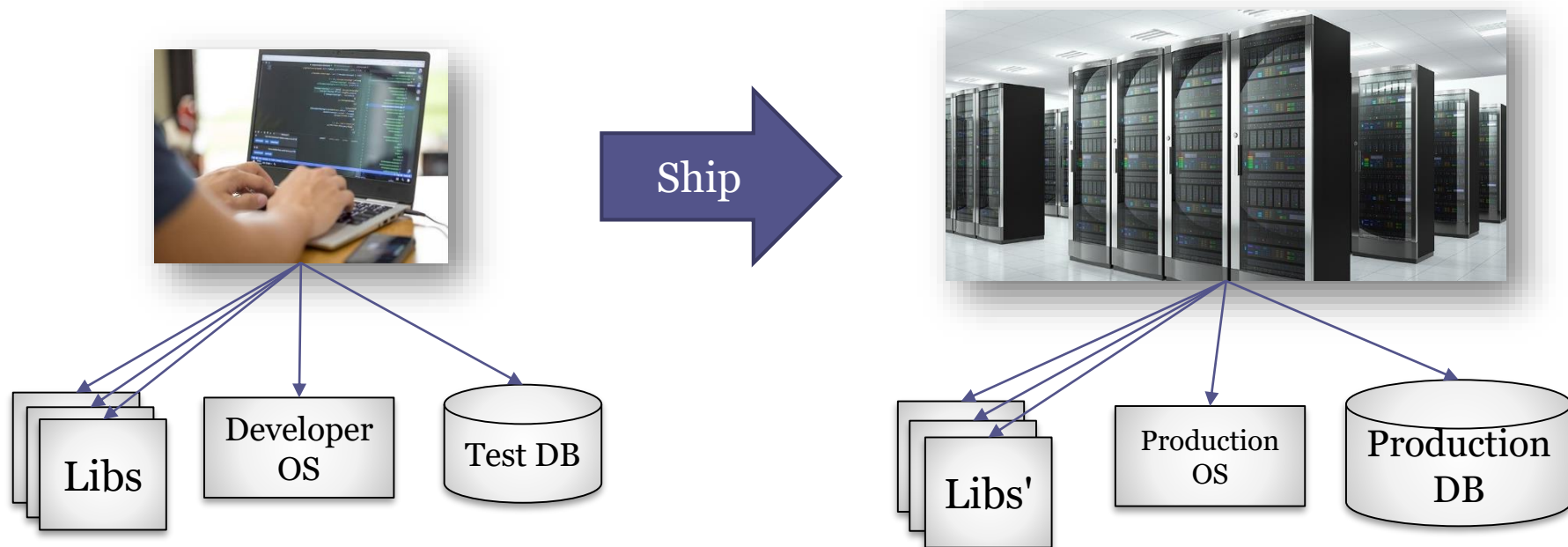
The problem of shipping software

Most software is not standalone

Lots of dependencies

Libraries, shared libraries, operating system libraries, ...

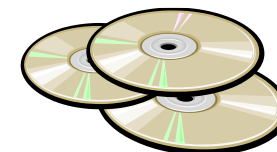
Developer's environment \neq Production environment



Distribution channels

Physical distribution

CDs, DVDs, ...



Web based

Downloads, FTP, ...



Application markets

Linux packages

App stores:

AppStore,

Google Play,

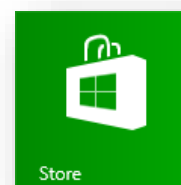
Windows Store



App Store



Google play



Store

Deployment

Deployment view

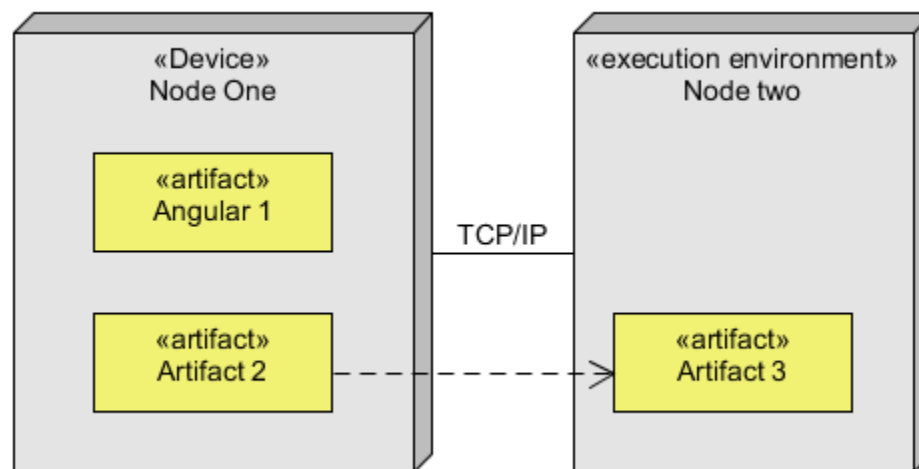
UML has deployment diagrams

Artifacts associated with computational nodes

2 types of nodes:

Device node

Execution environment node



Software computing options

On-premises

Cloud computing:

Edge computing

Fog computing

On premises computing

Software run *in the building*

Client's computers/data center

Advantages

More control on hardware environment

Upgrades, customization

Security

When it is well configured

Challenges

Requires hardware investment

Which hardware is required?

Return of inversion?

Maintenance costs

Also costs on licenses, space,...

Sys. admin. skills required



Cloud computing

Computer resources on demand

Software as a service (SaaS)

Advantages

- No initial investment
- Less expensive
- Affordable access to expensive hardware
- No need for sys. admins. skills

Challenges

- Security
- Dependency on cloud providers
- Varying costs (possible surprises)
- Requires configuration skills



Pets vs cattle metaphor

In the old way of doing things, we treat our servers like pets, for example Bob the mail server. If Bob goes down, it's all hands on deck. The CEO can't get his email and it's the end of the world.

In the new way, servers are numbered, like cattle in a herd. For example, www001 to www100. When one server goes down, it's taken out back, shot, and replaced on the line.



"Pet" server

- Unique and indispensable
- GUI driven
- Hand crafted
- Reserved
- Scale-up
- ...



"Cattle" servers

- Disposable, one of the herd
- API driven
- Automated
- On demand
- Scale-out
- ...

More info: <http://cloudscaling.com/blog/cloud-computing/the-history-of-pets-vs-cattle/>

Edge computing

Computing done at customer devices

Connected devices process data closer to where it is created

Example: IOTs, Connected cars, ...

Advantages

- Faster response (real time)
- Micro data storage
- On-premises visualization
- Independency (no network involved)

Challenges

- Less computing power
- No access to required data
- Embedded systems development



Fog computing

Computating at intermediate nodes

Local Area Network

Advantages

Local network
Control response

Challenges

Complexity
Security

Cloud layer

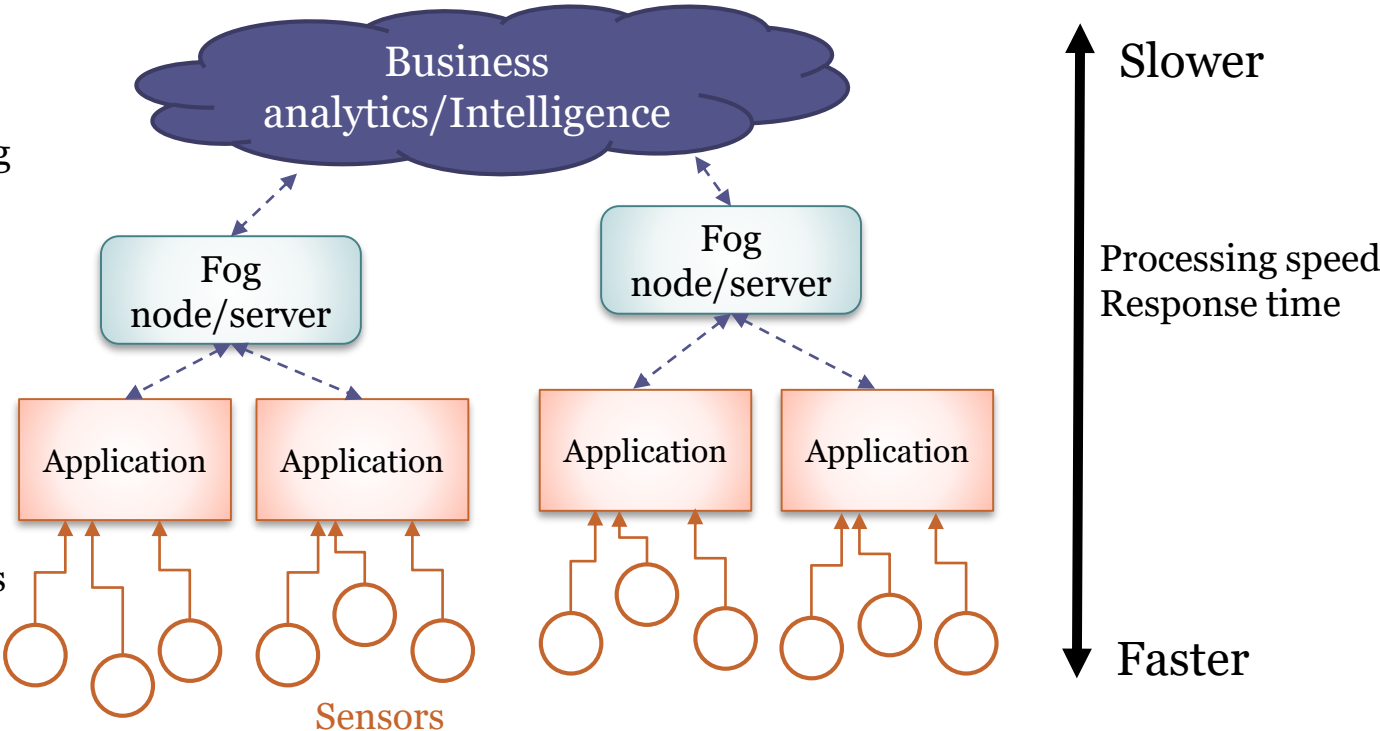
Data centers
Big data processing
Data warehousing

Fog layer

Local network
Control response

Edge layer

Real time
Embedded systems



Execution environments

Where will the software run?

Which dependencies does it have?

Operating systems

Shared libraries

Several options

Physical Hosts

Virtual machines

Containers



Physical hosts

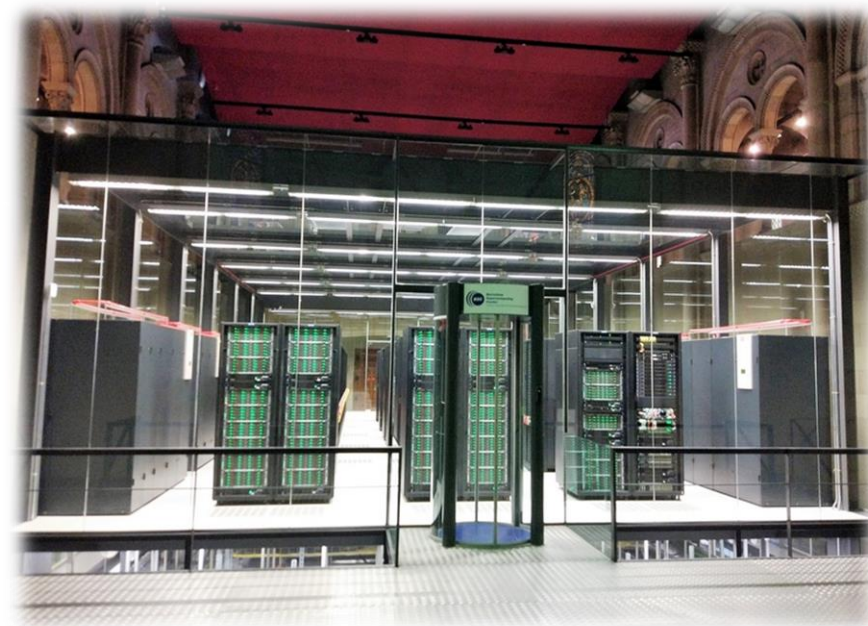
Lots of possibilities

Commodity computer

Super-computers

Server farms

End-user devices



The MareNostrum 4 supercomputer (2017)

Source: Wikipedia

Advantages

Control
Performance

Challenges

Reliability
Portability

System Virtual machines

Isolated emulation of a real machine

Virtual hardware emulator

Run multiple operating systems in a single machine

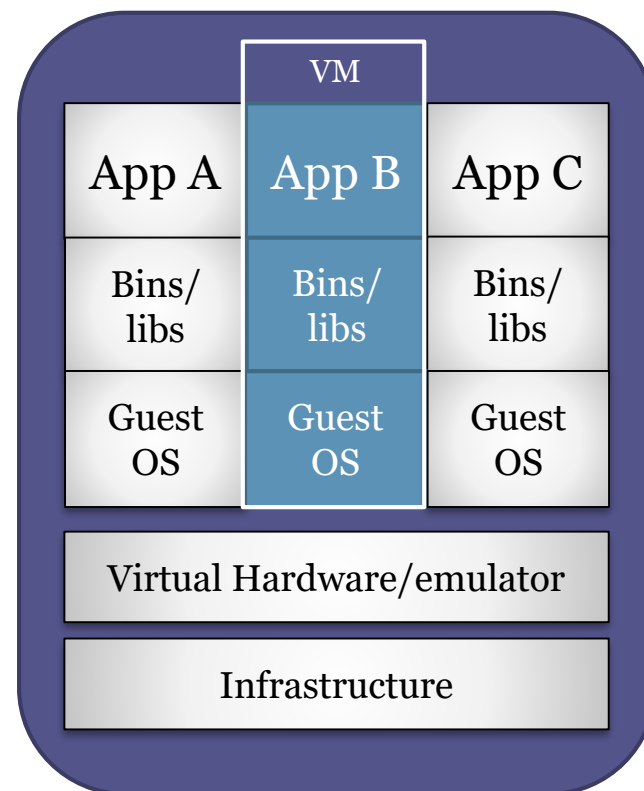
Examples: VMWare, Virtualbox, ...



Virtual machines

Running apps on VMs

Require Guest operating system + libraries



Advantages

- Portability
- Isolation
- Emulate whole machines

Challenges

- Resource consumption
- Startup times
- Less performance than bare-metal
- Can take a lot of space
- Each VM requires its own guest OS

Containers & docker

Operating system level virtualization

- Multiple isolated servers run on a single server

- The same OS kernel implements the *guest* servers

- Requires full process isolation at OS kernel

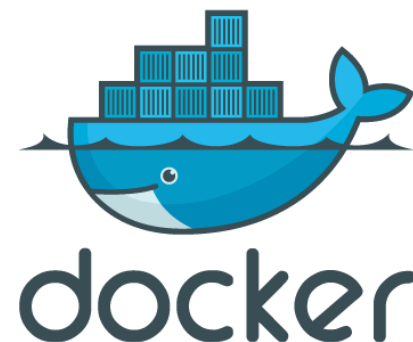
Docker (started in 2011) supports containers

Several parts

- Specification for container descriptions (images)

- Platform that runs containers

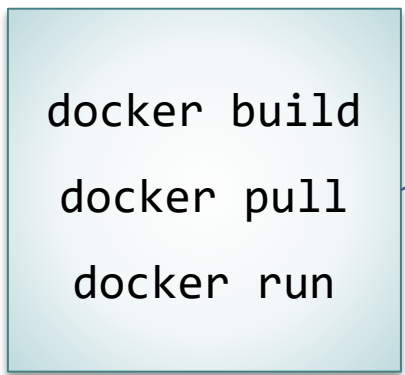
- Container registry (Docker-hub)



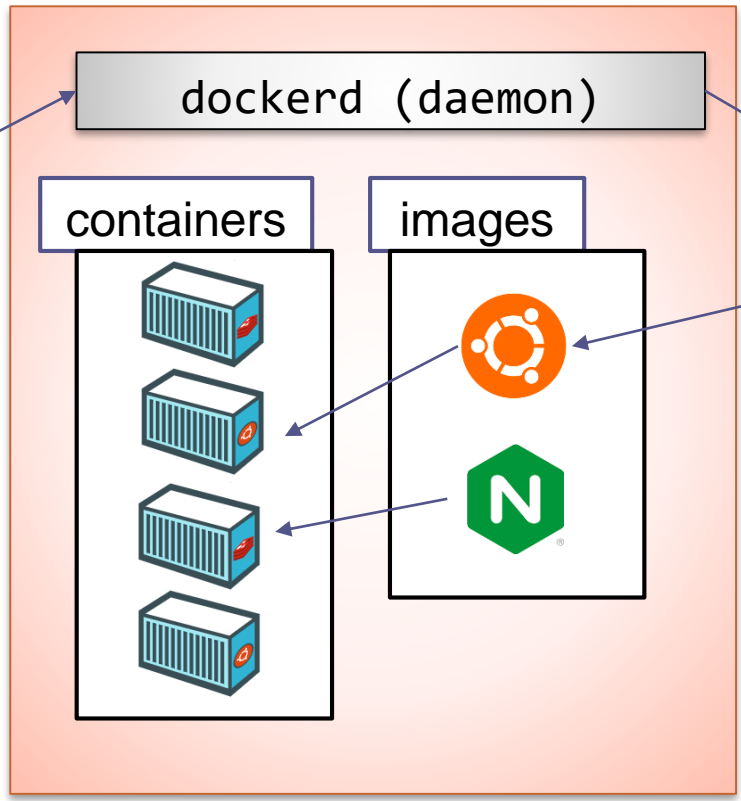
Docker high-level architecture

Client-server architecture

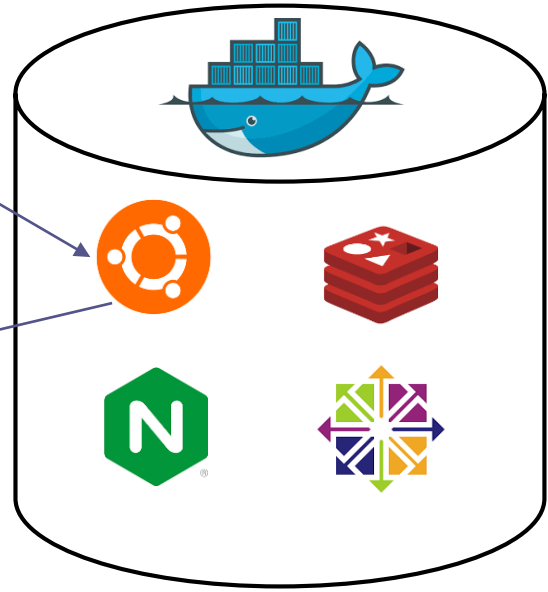
Docker client



DOCKER_HOST (server)



Docker registry



Docker hub

Docker images

Container image = read-only template with instructions to create a running container

DSL language

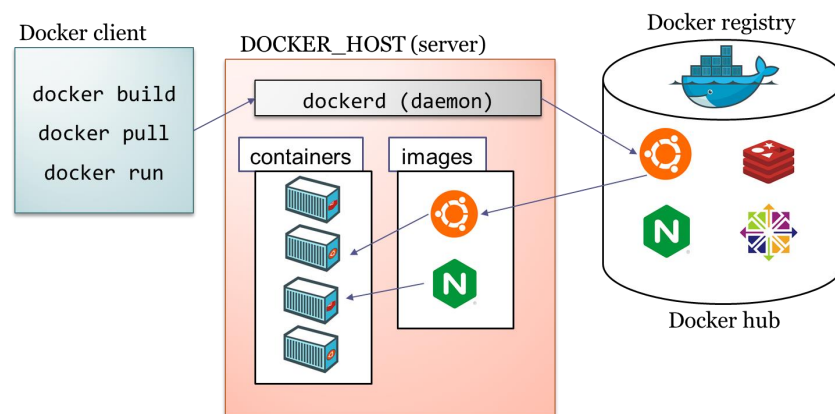
Typically described in a *Dockerfile*

Layered architecture

An image is usually *based on* another image + some customization

Each instruction creates a layer in the image

Lower layers can be reused



Docker containers

A runnable instance of an image

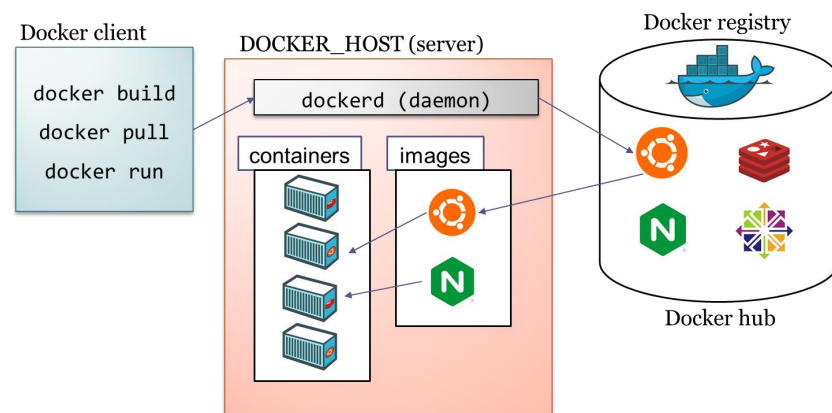
Containers are usually isolated

From other containers

From the host machine

It is possible to configure isolation

Data volumes, network, ...

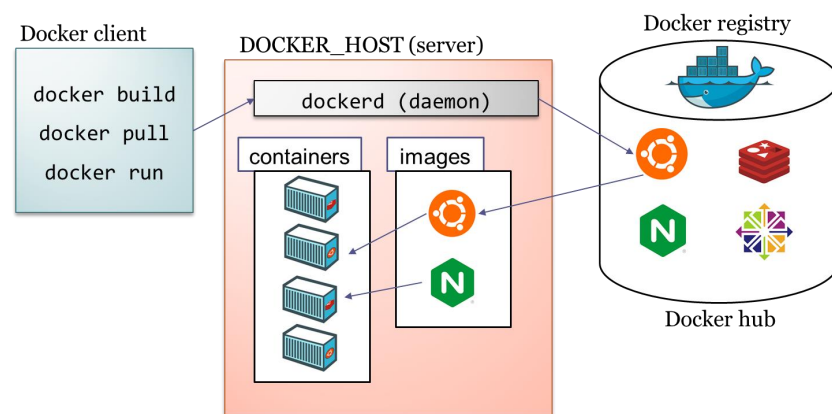


Docker registry

A Database of container images

Docker Hub is a public registry (used by default)

It is possible to use private registries

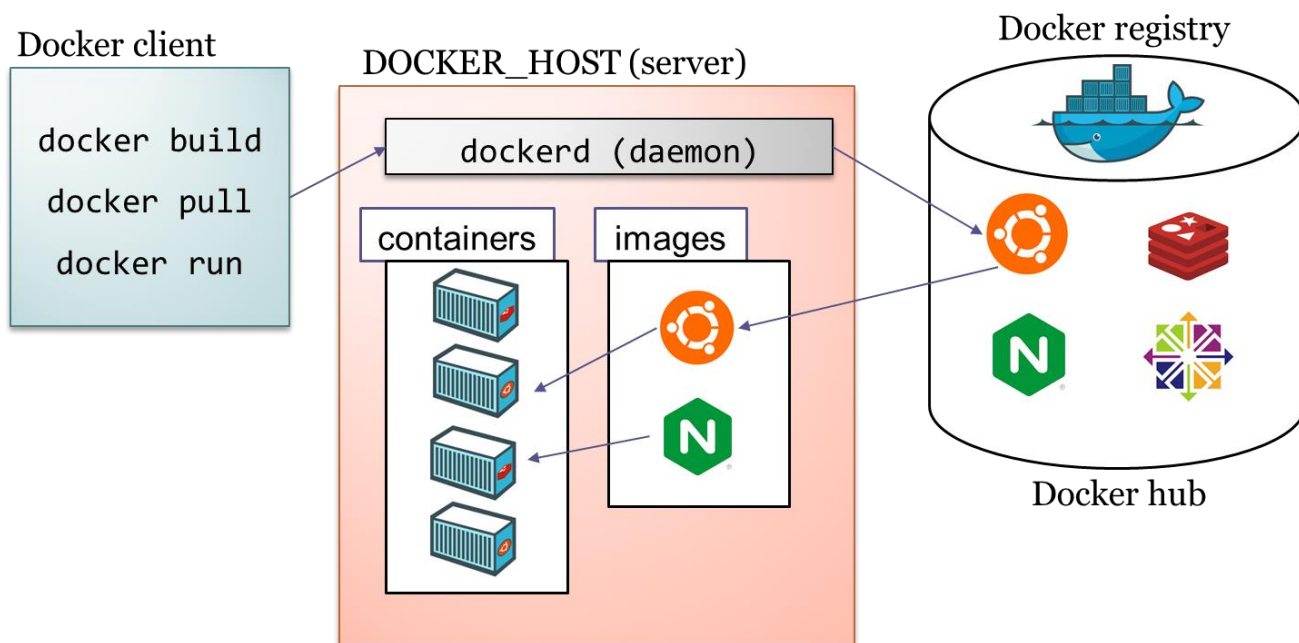


Docker client

docker command

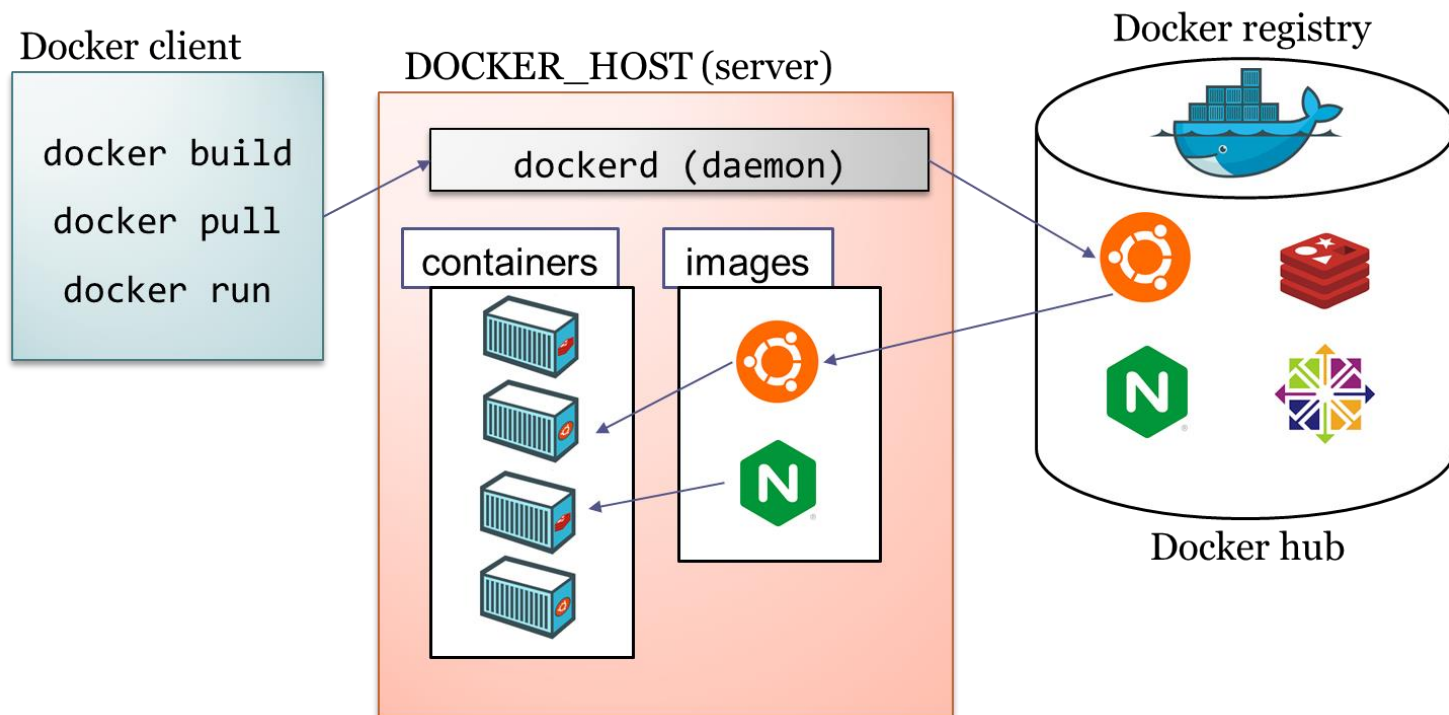
Communicates with the docker daemon using the API

Typical commands: docker pull, docker run, ...



Docker daemon

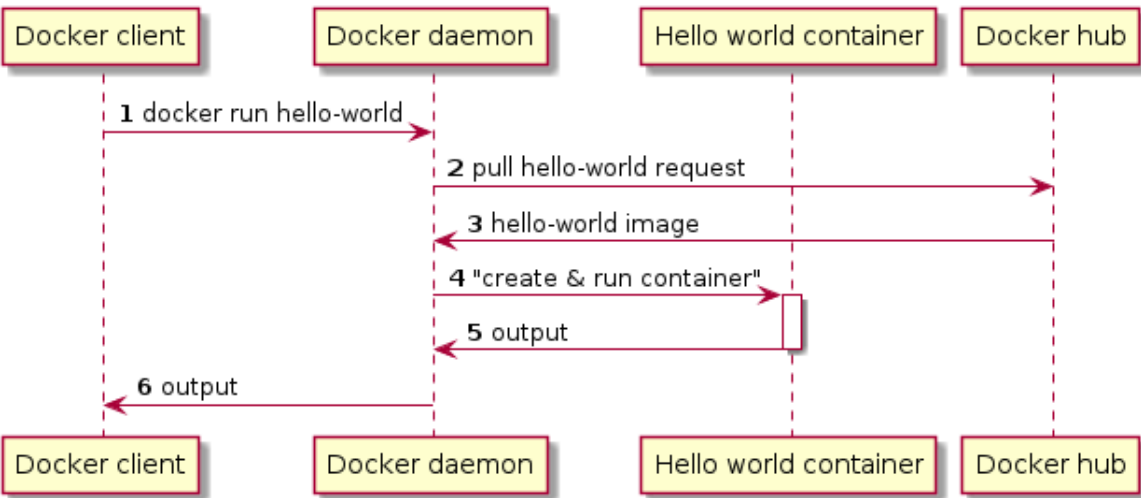
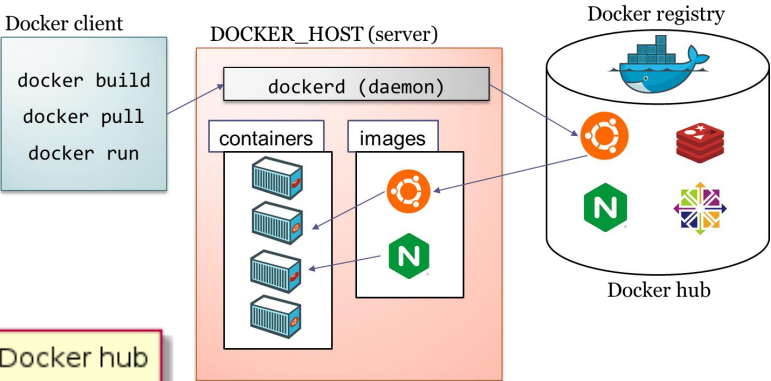
The docker daemon (dockerd) listens to API requests manages images and containers
It can also communicate with other daemons



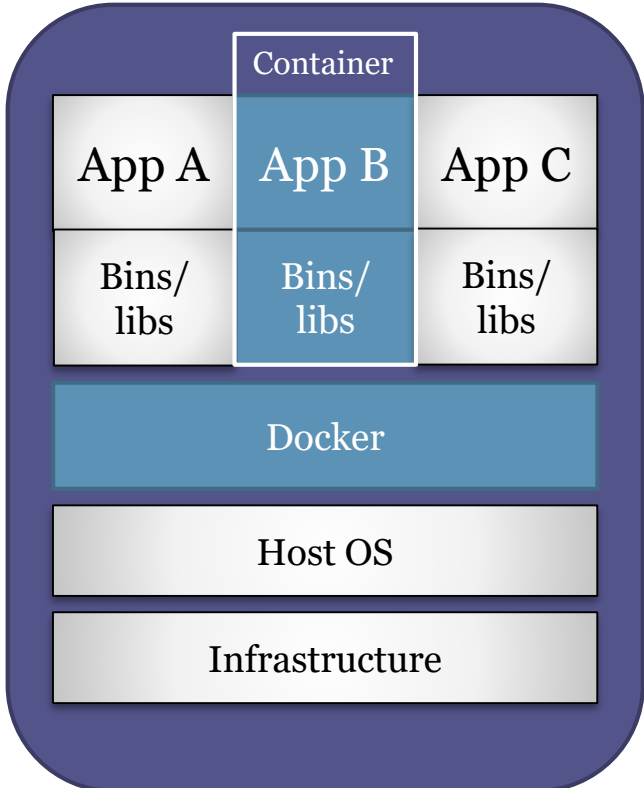
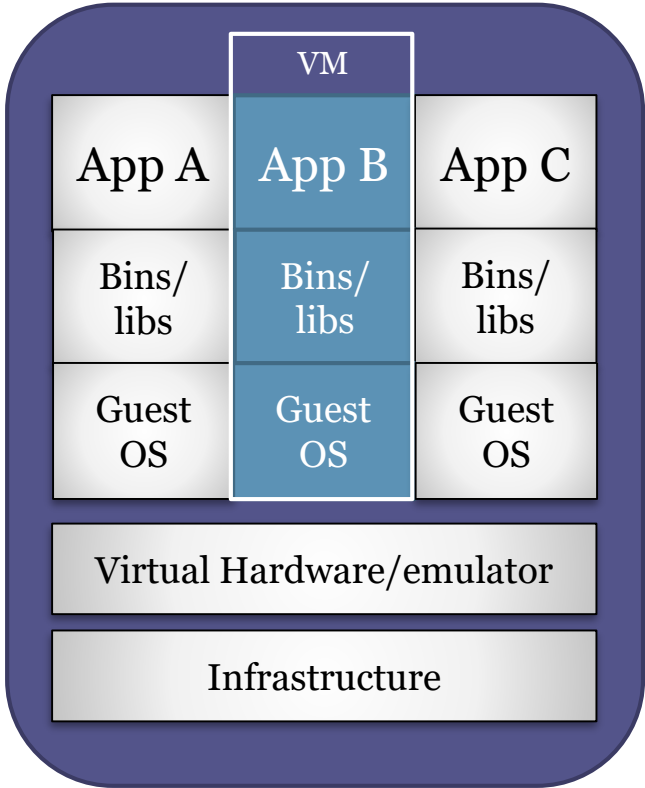
Docker example

Sequence diagram for hello-world example

```
$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
1b930d010525: Pull complete
Digest: sha256:f9dfddf63636d84ef479d645ab5885156ae030f...
Status: Downloaded newer image for hello-world:latest
```



Virtual machines vs Containers



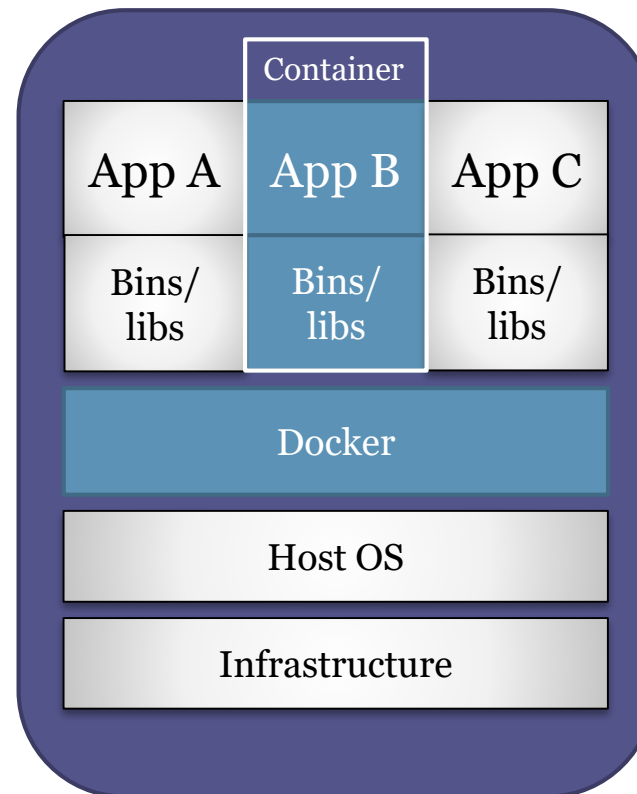
Containers consequences

Advantages

- Consistency & portability
- Easy to deploy
- Isolation
- Performance
 - Less space than VMs
 - 1000s of containers
- Immutable architecture
- Declarative configuration
- Infrastructure as code
- Automation

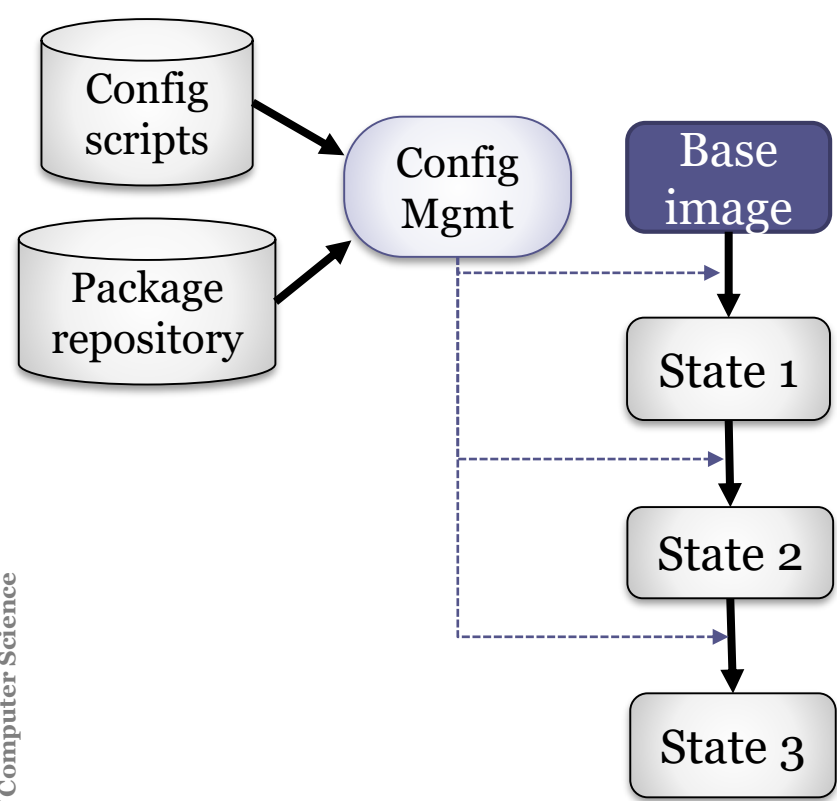
Challenges

- Orchestration
- Persistence more complex
- Graphical applications
- Platform-dependent (Linux)

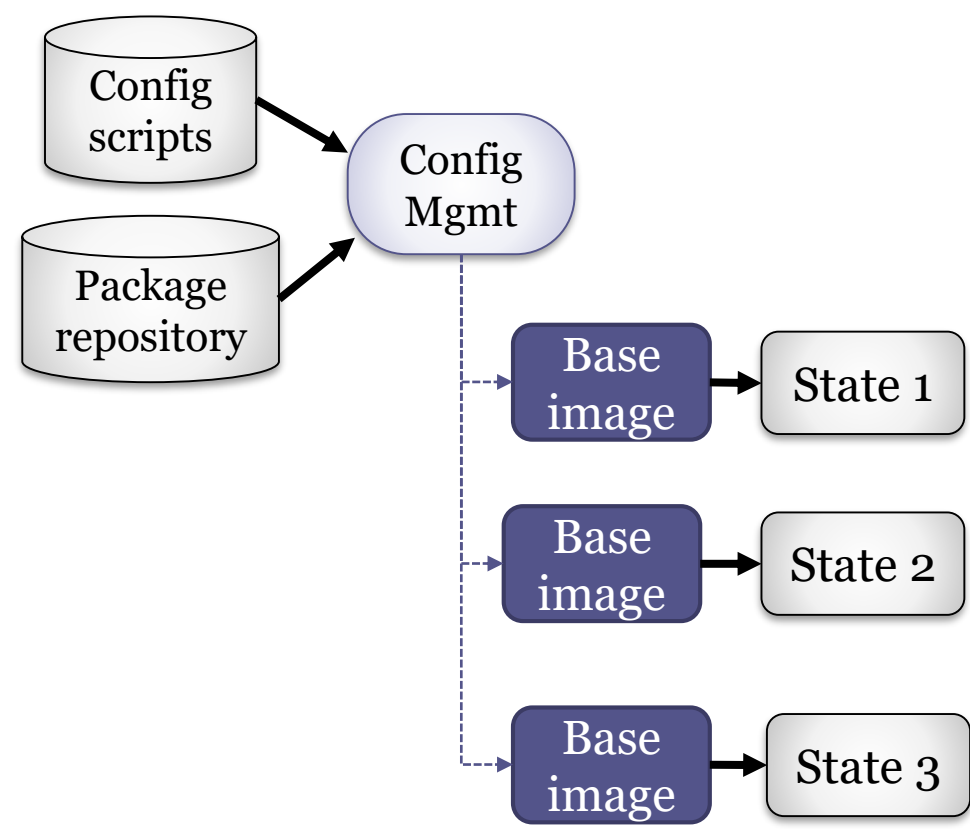


Mutable vs Immutable infrastructure

Mutable infrastructure



Immutable infrastructure



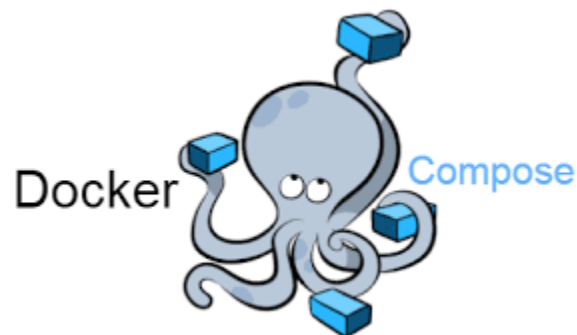
Container management

Docker-compose = tool to define and run multi-container apps

YAML configuration file (docker-compose.yml)

With a single command, create and start all the services from a multi-container configuration

Docker-compose usually works in a single host



Container orchestration

Automatically manage clusters of containers

Typical features:

Load balancing, Container lifecycles, provisioning...

Kubernetes

Initially developed by Google, donated to CNCF



Framework for distributed systems

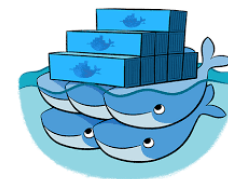
Clusters consists of pods, deployments and services

Available in most cloud providers

Docker swarm

Developed by Docker

It can be considered a "mode" of running docker



Deployment



Deployment pipeline

Automated implementation of an application's build, deploy, test and release process

Goals

Create runtime environments on demand

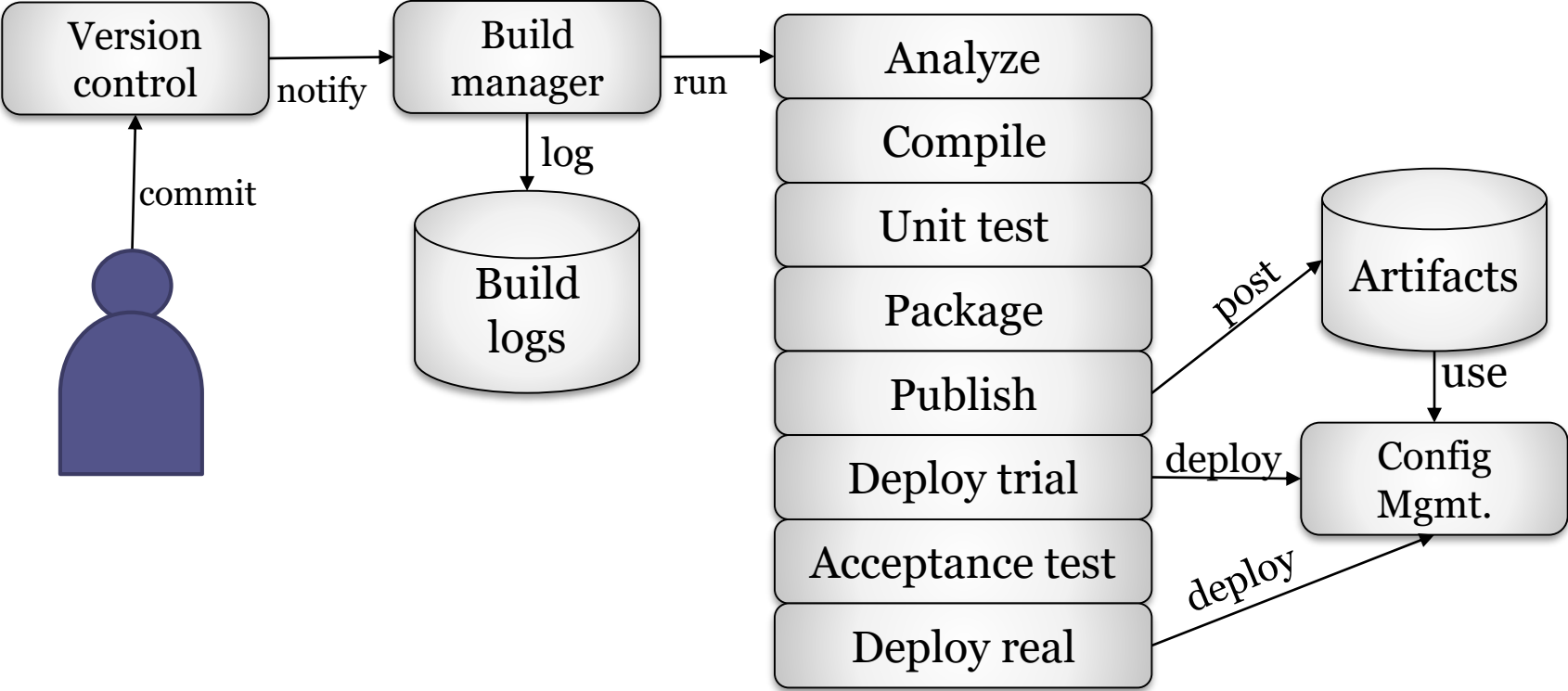
Fast, reliable, repeatable and predictable outcomes

Consistent environments in staging and production

Establish fast feedback loops to react upon

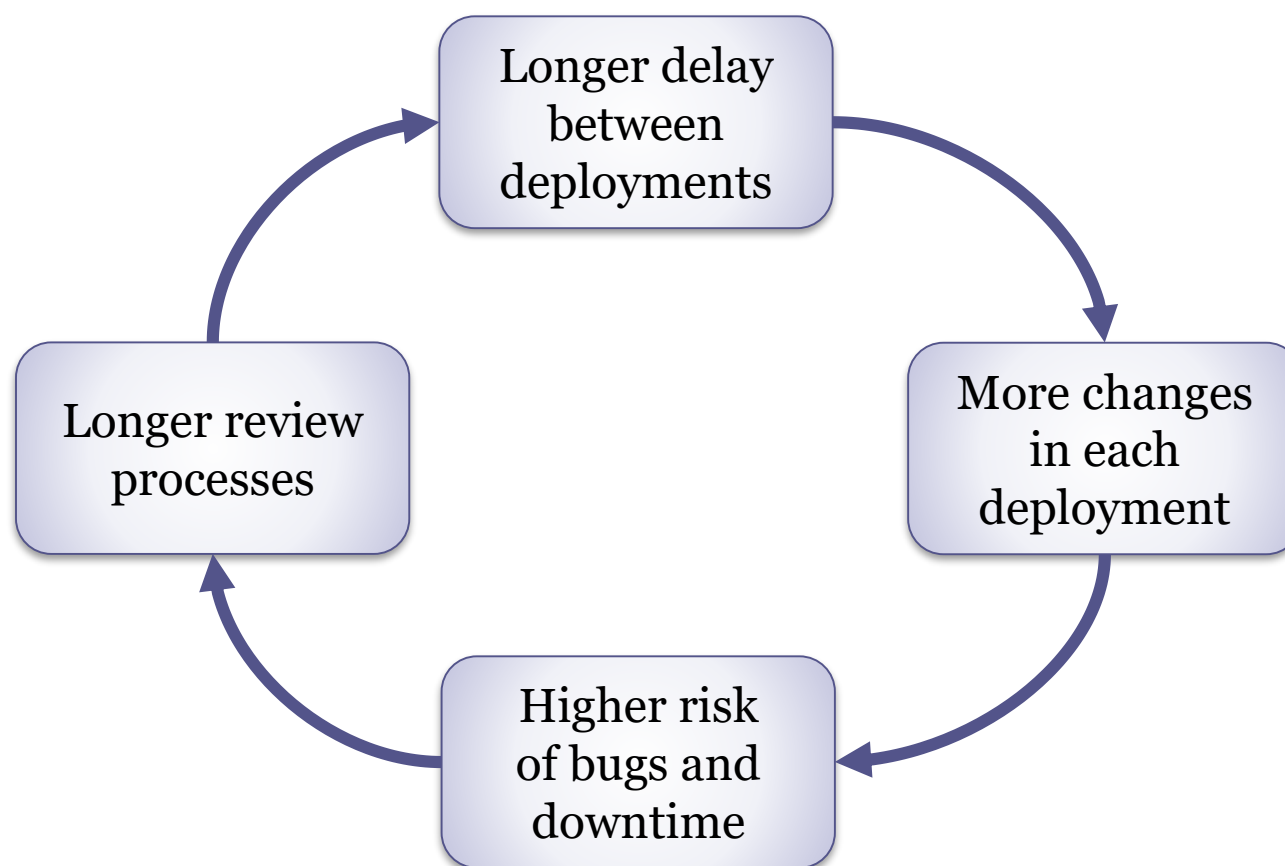
Make release days riskless, almost boring

Deployment pipeline



Manual deployment

Vicious circle of deployment size and risk



Continuous deployment

"If it hurts do it more often"

In the limit: "Do everything continuously"

Run the full pipeline in every commit

Final stage: deployment in production

Possibilities

- Confirmation by some human before going to production

- Automatic deployment to production

- Deployment to production marked by some tags

Trade-off

- Cost of moving slower vs cost of error in deployment

Continuous deployment

Patterns

Infrastructure as code

Keep everything in Version Control

Code

Configuration

Data

Align development and operations (DevOps)

Tools:

Ansible, Chef, Puppet,...

Best practices: 12 factors (next slide)

12 factor <https://12factor.net/>

I. Codebase One codebase tracked in revision control, many deploys

II. Dependencies Explicitly declare and isolate dependencies

III. Config Store config in the environment

IV. Backing services Treat backing services as attached resources

V. Build, release, run Strictly separate build and run stages

VI. Processes Execute the app as one or more stateless processes

VII. Port binding Export services via port binding

VIII. Concurrency Scale out via the process model

IX. Disposability Maximize robustness with fast startup and graceful shutdown

X. Dev/prod parity Keep development, staging, and production as similar as possible

XI. Logs Treat logs as event streams

XII. Admin processes Run admin/management tasks as one-off processes

Software in production



Quality attributes in production

Configurability

Customize system without re-compiling it

Observability

Possibility to monitor the internal state of a system

Availability

Probability that a system is working at time t

Stability

Produce availability despite faults and errors

Reliability

Probability that a system produces correct outputs over some time t

Configurability

Lots of configurable properties

Hostnames, port numbers, filesystem locations, ID numbers, usernames, passwords, etc.

Config files = interface between developers and operators

Should be human-readable and machine processable

Examples: XML, JSON, YAML, ...

Can contain sensitive information

Separated from source code



Logging

Logging is ubiquitous and easy to generate

White-box technology (integrated in source code)

They show activity and can easily persist

Human-readable

Log locations

Separate logs from source code

Logging levels

Find a good balance for logging between too noisy/silent

Anything marked as "ERROR" or "SEVERE" should require action

Remember: disable debug logs in production



Monitoring

Monitoring: Observe the behaviour at runtime while software is running

Time-series database systems

Time-series visualizations and dashboards

Prometheus, Graphite, Grafana, Datadog, Nagios, ...

Health checks

Profiling: Measure performance of a software while it is running



Data in production

High availability and data replication

Ensure backup and restore

Database schemas in control version

Change requests

Data migration

Data purging

Sensible data in production

Inaccessible to developers

Encrypted

...



System problems

Fault:

Incorrect internal state (not necessarily observable)

Initiated by some defect or injection

Error:

Observable incorrect operation

Failure:

Loss of availability. System unresponsive

Chain reactions



Law of large systems

Large systems exist in a state of continuous partial failure

Corollary:

"Everything is working" is the anomaly

Important:

Don't propagate faults



Source: "Airplane" film

<https://www.imdb.com/title/tt0080339/>

In-production patterns

Load balancing

Timeouts

Circuit breakers

Bulkheads

Steady state

Fail fast

Handshaking

Test harnesses

Decoupling middleware

Create backpressure

Governor



Some libraries: <https://resilience4j.readme.io/>

Load balancing

Distribute requests accross a pool of instances

Goal:

Serve all requests correctly in shortest feasible time

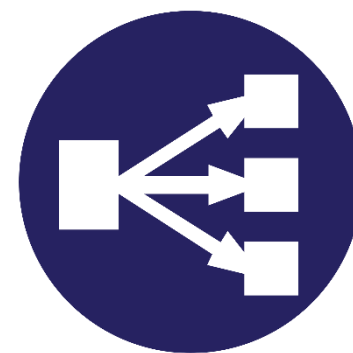
Decisions to take:

Load balancing algorithms

What health checks to do on instances

What to do when no pool members are available

Hardware/Software load balancers



Timeouts

Add a time limiter to other services requests

Provide fault isolation

A problem in some other service does not have to become your problem

Timeouts usually followed by retries

It may make things worse

The situation may not recover automatically

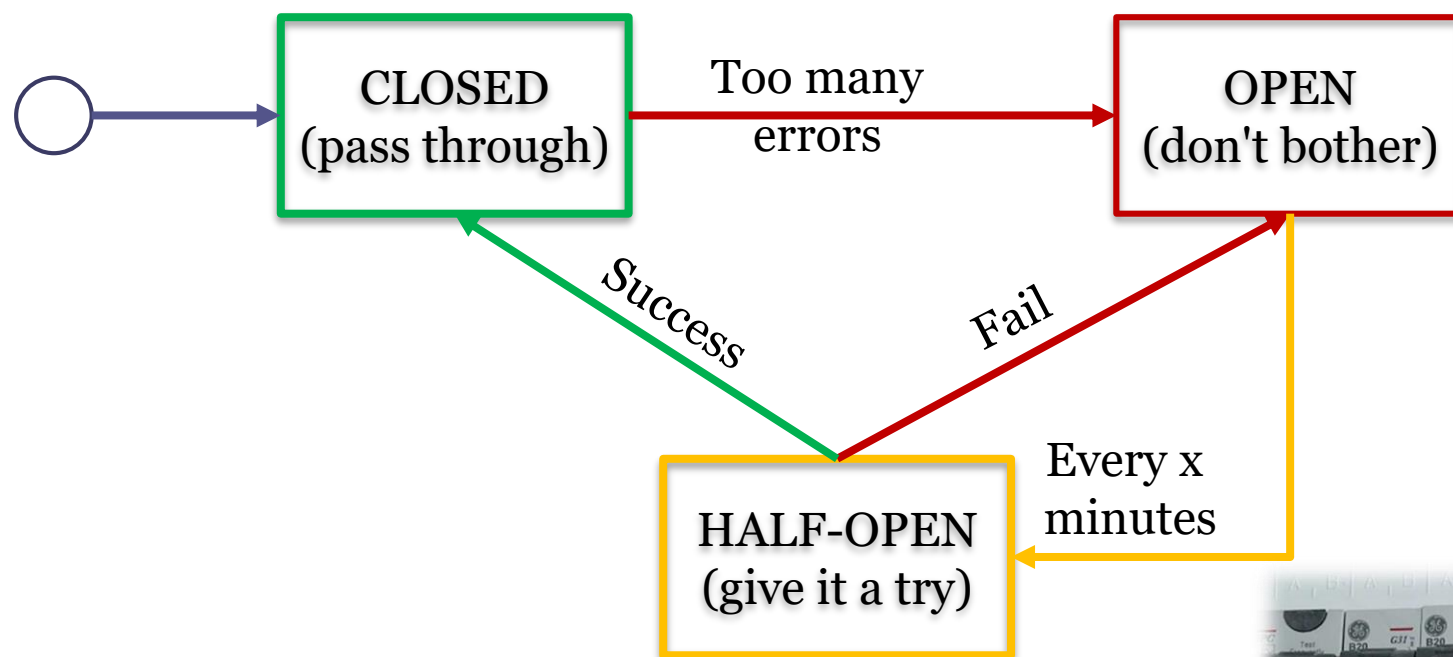
The consumer waits more time

Sometimes, just failing is better



Circuit breaker

Inspired by electrical fuses

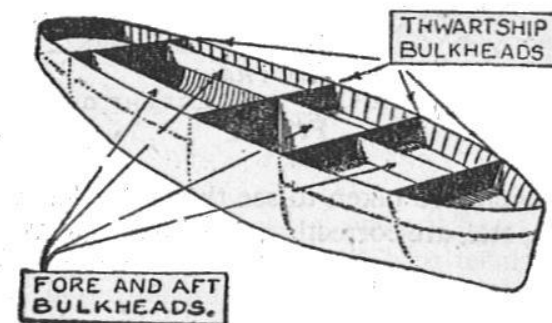
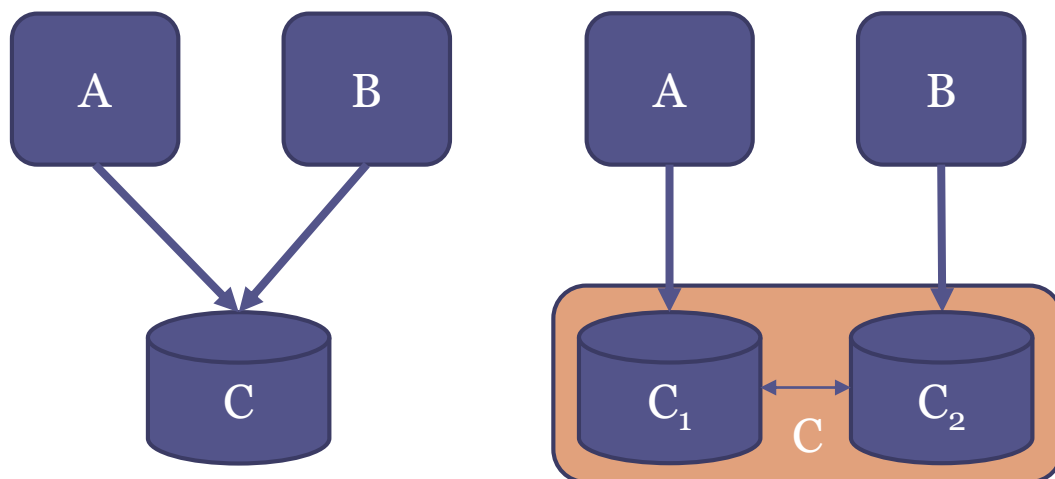


Bulkheads

"Contain damage" (save part of the ship)

If a component breaks, the system still works

Example: replicate instances in the cloud



Steady state

"Nothing is infinite"

Keep system resources constant

Avoid human intervention for cleanup

Examples:

Data purging

Log files

In-memory caching



Fail fast

Don't make consumers wait for a failure response

Reserve resources before starting work

Don't do useless work

Verify integration points early

Check all resources are available before start

Basic input validation

Shed load

Refuse new requests when load is too high



"Check ingredients before cooking"

Let it crash

"Crash components to save systems"

Inspired by Erlang's error handling

If a component can't do what it has to do, let it crash

Let some other component do the recovery

Do not program defensively

Conditions

Create boundaries

A component crashes in isolation

Fast replacement

Supervision

Reintegration



Handshaking

"Agree before doing"

Cooperative demand control

Both clients and server agree

The server should can reject incoming work

Services provide "health check" query

Load balancers check health before directing a request to some instance



Create backpressure

Backpressure = *resistance opposing desired flow of data*

Input is coming faster than we can output

Create safety by slowing down producers

Strategies

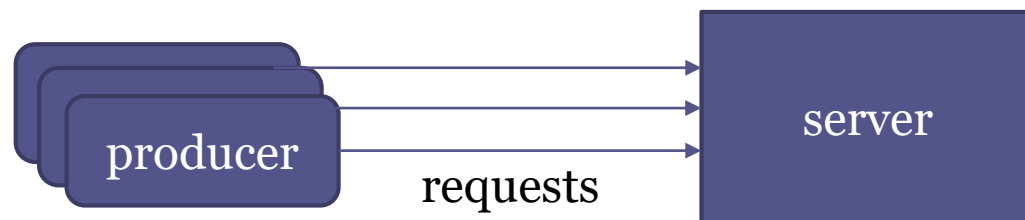
Control de producer (slow down producers)

Buffer (accumulate incoming data temporarily)

Unbounded buffers can be very dangerous

Drop

Not always acceptable to lose data



Governor

Create governors to slow the rate of actions

When automation goes wrong, can do it very quickly

Avoid force multiplier

Slow things down to allow human intervention

Apply resistance in the unsafe direction

Examples: shutdowns, deleting instances, ...

Consider a response curve



Source: [Steam Engine Centrifugal Governor](#)

Test harnesses

"Be evil when testing"

Create test harnesses that check most failure modes

Emulate out-of-spec failures

Stress the caller

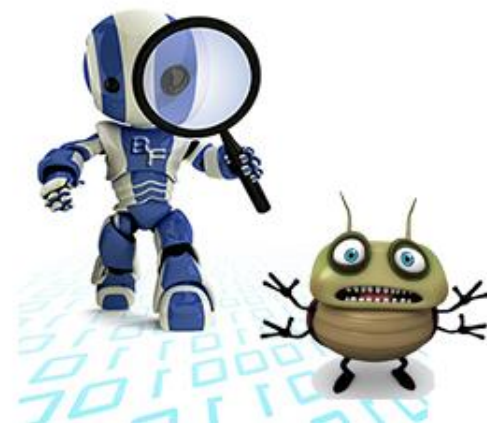
Produce slow responses, no responses, garbage responses

Shared harnesses can be reused

Example: killer services

Related with Chaos engineering

[See later]



Chaos engineering

Started by Netflix in 2010 (Chaos Monkey)

Test distributed systems

Break things on purpose

Failure injection testing

Ensure that one instance failure doesn't affect the system

Antifragility and resilience



<https://github.com/Netflix/chaosmonkey>

In-production antipatterns

Integration points

Chain reactions

Cascading failures

Users

Blocked threads

Self-denial attacks

Scaling effects

Unbalanced capacities

Dogpile

Force multiplier

Slow responses

Unbounded result sets



Testing in production

Progressive delivery

Reduce *blast radius* of new deployments

Enable experimentation

Some techniques

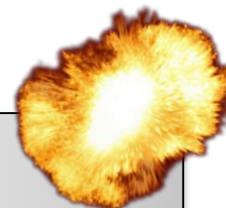
Canary releases

Feature toggles

A/B testing and multi-armed bandits

Blast radius of a deployment:

Who is impacted? What functionality? How many locations? ...

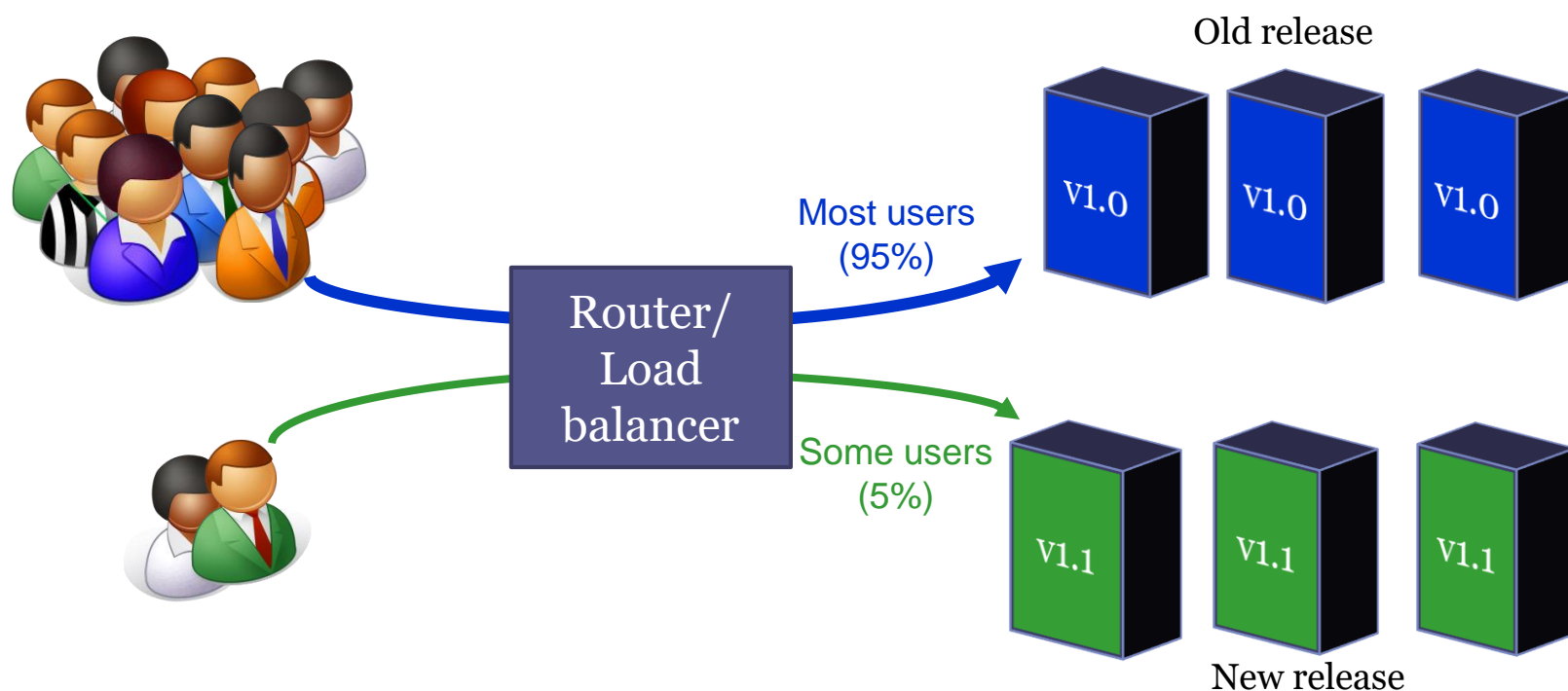


Canary release

Introduce new releases by slowly rolling out the change to small subset of users

Infrastructure driven (router/load balancers)

Blue-Green deployment

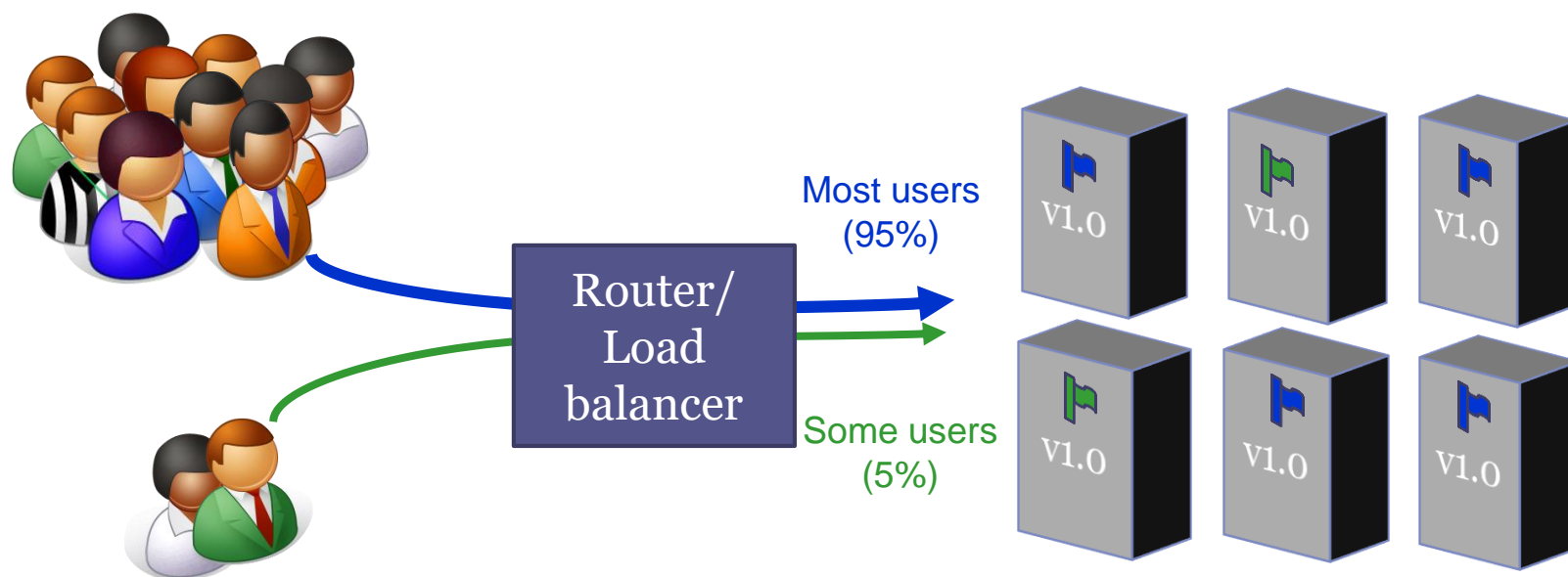


Feature toggles

Also known as *feature flags*, *feature bits*

Modify system behaviour without changing code

Decouple deployment from release



Types of tests

A/B testing:

Also known as split testing, bucket testing

Controlled experiment to test some hypothesis

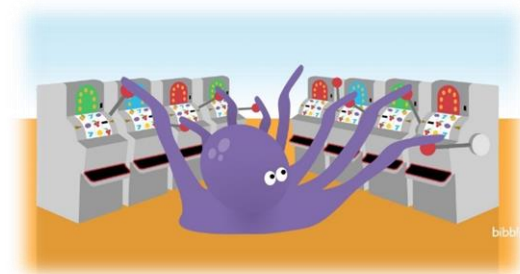
Divide users in groups

Problem: Bad alternatives shown to groups of users during experiment

Multi-armed bandits

Dynamic traffic allocation

Bad alternatives get less users during time



Load & stress testing

Load testing

Test performance under load

Example: simulate multiple users accessing concurrently

Stress testing

Load raised beyond normal usage patterns to test system's response

Check upper bounds

What happens when limit is reached

Several tools

JMeter, Gatling



Incidents & post-mortem

Resolve and review incident

Ensure team view it as **blameless**

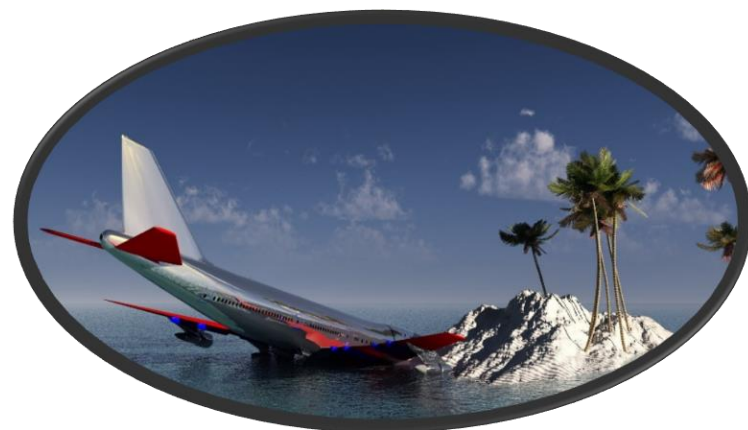
Create post-mortem report

- Incident details

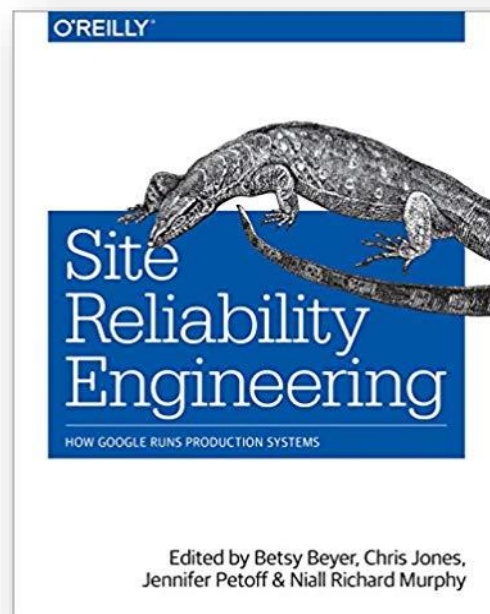
- Root Cause Analysis

- Timeline and actions taken to resolve it

Identify preventive measures



End of presentation



Free online