Building Cloud-Native Applications

Slides: http://bit.ly/buildcnapps

Agenda - Day 1

Workshop runs from 9:00 AM to 5:00 PM

- 30 min coffee breaks (10:30AM and 1:30PM)
- 1h lunch break (12:30PM)

Multiple sections - theory + exercises

- Introduction to Cloud-Native
- Cloud-Native Building Blocks
- Kubernetes
- APIs

Introduction

- I am Peter (@pjausovec)
- Software Engineer at Oracle
- Working on "cloud-native" stuff
- Books:
  - Cloud Native: Using Containers, Functions, and Data to Build Next-Gen Apps
  - SharePoint Development
  - VSTO For Dummies
- Courses:
  - Kubernetes Course (https://startkubernetes.com)
  - Istio Service Mesh Course (https://learnistio.com)

Introduction to Cloud-Native

Understanding Cloud-Native

"... natively utilizies service and infrastructure from cloud computing providers..."

"... approach to build & run apps that exploit the advantages of the cloud computing model"

"... describes container-based environments... deployed as microservices and managed on elastic infrastructure through agile DevOps, continuous delivery workflows."

"... build, run, and improve apps based on well-known techniques and technologies for cloud computing."

"... collection of small, independent, and loosely coupled services."

CNCF Definition

Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructures, and declarative APIs exemplify this approach.

These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil.

https://github.com/cncf/toc/blob/master/DEFINITION.md

Pets vs. cattle

Pets

- Treat your infrastructure like pets
- Give them names, IP addresses, ...
- Care of them, keep them updated

Cattle

- Everything is just a number
- No attachment
- If something goes wrong, you replace it

Understanding Cloud-Native

Apart from focusing on business logic, you will realize the following when building cloud-native applications for the first time:

- I am dealing with services running across multiple machines
- I am dealing with network and communication between these services

Challenges

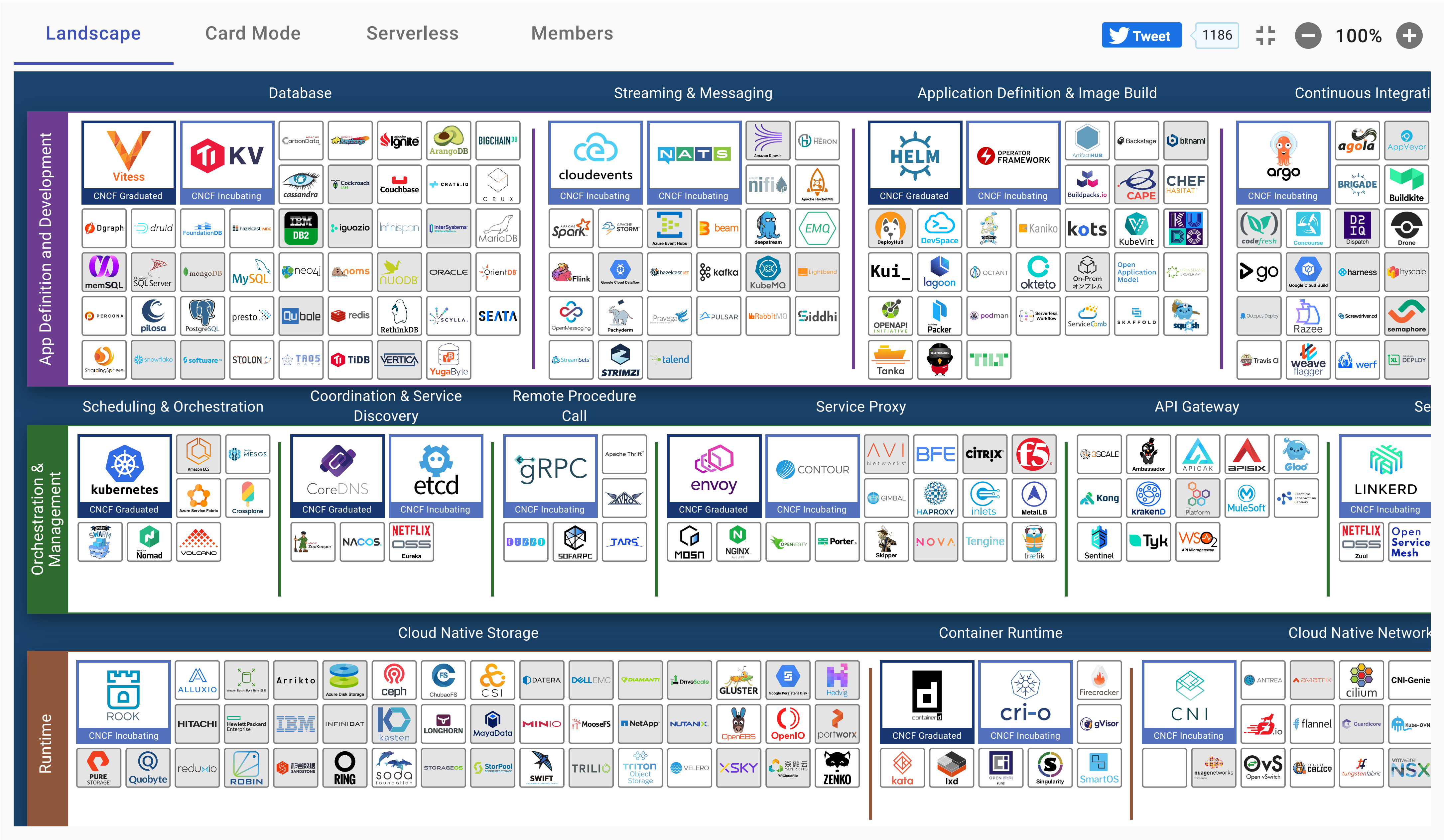
What are distributed systems?

Cloud-native apps are distributed systems*

*computers connected through a network and appearing as a single computer

Challenges

New technologies and tools


Landscape
Card Mode
Serverless
Members
Tweet
1186
100%
Database
Streaming & Messaging
Application Definition & Image Build
Continuous Integrati
App Definition and Development
Vitess
CNCF Graduated
TiKV
CNCF Incubating
CarbonData
Hadoop
Ignite
ArangoDB
BIGCHAINDB
cassandra
Cockroach
Couchbase
CRATE.IO
CRUX
Dgraph
druid
FoundationDB
hazelcast
IBM DB2
iguazio
Infinispan
InterSystems
MariaDB
memSQL
SQL Server
mongoDB
MySQL
neo4j
noms
NUODB
ORACLE
OrientDB
PERCONA
pilosa
PostgreSQL
presto
Qubole
redis
RethinkDB
SCYLLA
SEATA
ShardingSphere
snowflake
software
STOLON
TAOS
TiDB
VERTICA
YugaByte
cloudevents
CNCF Incubating
NATS
CNCF Incubating
Amazon Kinesis
HERON
nifi
Apache RocketMQ
Spark
STORM
Azure Event Hubs
beam
deepstream
EMQ
Flink
Google Cloud Dataflow
hazelcast
kafka
KubeMQ
Lightbend
OpenMessaging
Pachyderm
Pravega
PULSAR
RabbitMQ
Siddhi
StreamSets
STRIMZI
talend
HELM
CNCF Graduated
OPERATOR FRAMEWORK
CNCF Incubating
ArtifactHUB
Backstage
bitnami
Buildpacks.io
CAPE
CHEF HABITAT
DeployHub
DevSpace
Kaniko
kots
KubeVirt
KUDO
Kui
lagoon
OCTANT
okteto
On-Prem
Open Application Model
OPENAPI INITIATIVE
Packer
podman
Serverless Workflow
ServiceComb
SKAFFOLD
squash
Tanka
TILT
argo
CNCF Incubating
agola
AppVeyor
BRIGADE
Buildkite
codefresh
Concourse
D2IQ Dispatch
Drone
go
Google Cloud Build
harness
hyscale
Octopus Deploy
Razee
Screwdriver.cd
semaphore
Travis CI
weave flagger
werf
DEPLOY
Scheduling & Orchestration
Coordination & Service Discovery
Remote Procedure Call
Service Proxy
API Gateway
Se
Orchestration & Management
kubernetes
CNCF Graduated
Amazon ECS
MESOS
Azure Service Fabric
Crossplane
SWARM
Nomad
VOLCANO
CoreDNS
CNCF Graduated
etcd
CNCF Incubating
ZooKeeper
NACOS
NETFLIX OSS
Eureka
gRPC
CNCF Incubating
Apache Thrift
AVRO
DUBBO
SOFARPC
TARS
envoy
CNCF Graduated
CONTOUR
CNCF Incubating
AVI Networks
BFE
CITRIX
f5
GIMBAL
HAPROXY
inlets
MetalLB
MOSN
NGINX
OPENRESTY
Porter
Skipper
NOVA
Tengine
traefik
3SCALE
Ambassador
APIOAK
APISIX
Gloo
Kong
KrakenD
Platform
MuleSoft
Sentinel
Tyk
WSO2
API Microgateway
LINKERD
CNCF Incubating
NETFLIX OSS
Zuul
Open Service Mesh
Cloud Native Storage
Container Runtime
Cloud Native Network
Runtime
ROOK
CNCF Incubating
ALLUXIO
Amazon Elastic Block Store (EBS)
Arrikto
Azure Disk Storage
ceph
ChubaoFS
CSI
DATERA
DELL EMC
DIAMANTI
DriveScale
GLUSTER
Google Persistent Disk
Hedvig
HITACHI
Hewlett Packard Enterprise
IBM
INFINIDAT
kasten
LONGHORN
MayaData
MINIO
MooseFS
NetApp
NUTANIX
OpenEBS
OpenIO
portworx
PURE STORAGE
Quobyte
reduxio
ROBIN
SANDSTONE
RING
soda foundation
STORAGEOS
StorPool
SWIFT
TRILIO
Triton Object Storage
VELERO
XSKY
YRCloudFile
ZENKO
containerd
CNCF Graduated
cri-o
CNCF Incubating
Firecracker
gVisor
kata
lxd
OPEN runc
Singularity
SmartOS
CNI
CNCF Incubating
ANTREA
aviatrix
cilium
CNI-Genie
flannel
Guardicore
Kube-OVN
nuagenetworks
Open vSwitch
CALICO
tungstenfabric
vmware NSX

Challenges

Patterns for building cloud-native apps

Cloud-Native vs. Traditional Architectures

Stateful vs. Stateless

- State stored with the compute instance
- Load balancers using sticky sessions
- What happens on reboot or crash 💥

Service orchestration vs. Service choreography

- Multiple services orchestrated to work as one, using sync communication
- Choreography = uses eventing system

Dealing with failures

- Minimize failures vs. expect and deal with them

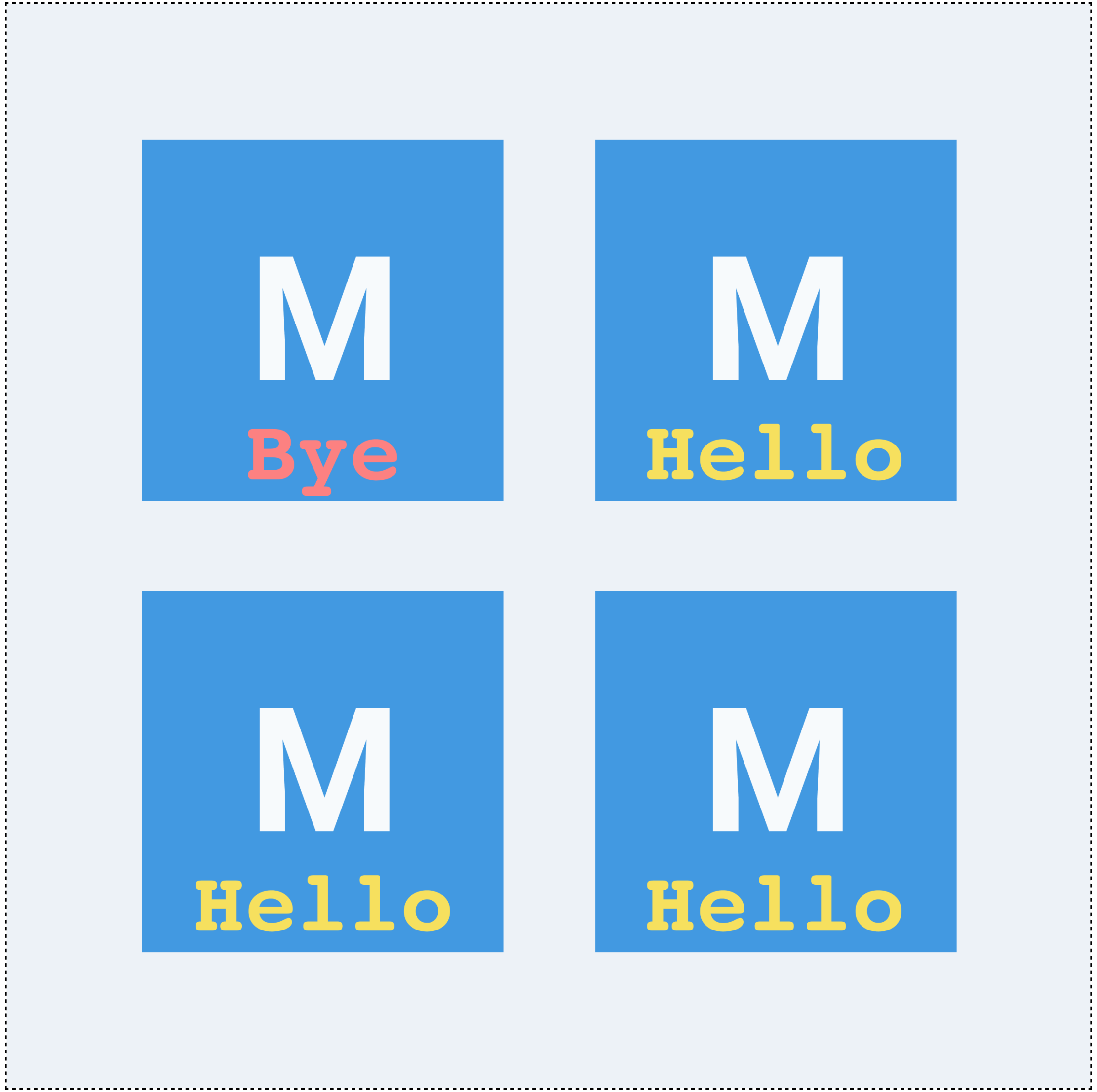
CAP Theorem

Consistency

High Availability

Partition Tolerance


M
Hello
M
Hello
M
Hello
M
Hello


M
Bye
M
Hello
M
Hello
M
Hello

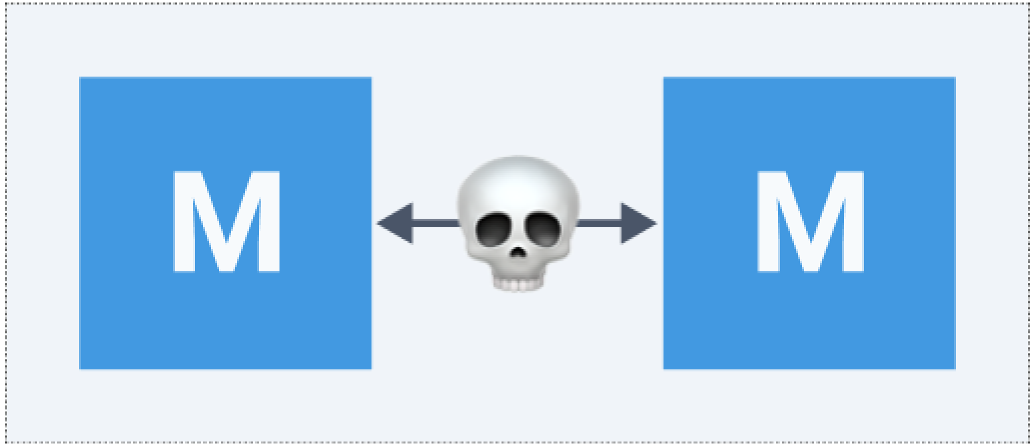
Consistency

Every node in the system provides the most recent state and nodes never return an outdated state


M
Hello
Hello
M
Hello
Hello

Availability

System is available, even though not all nodes are available

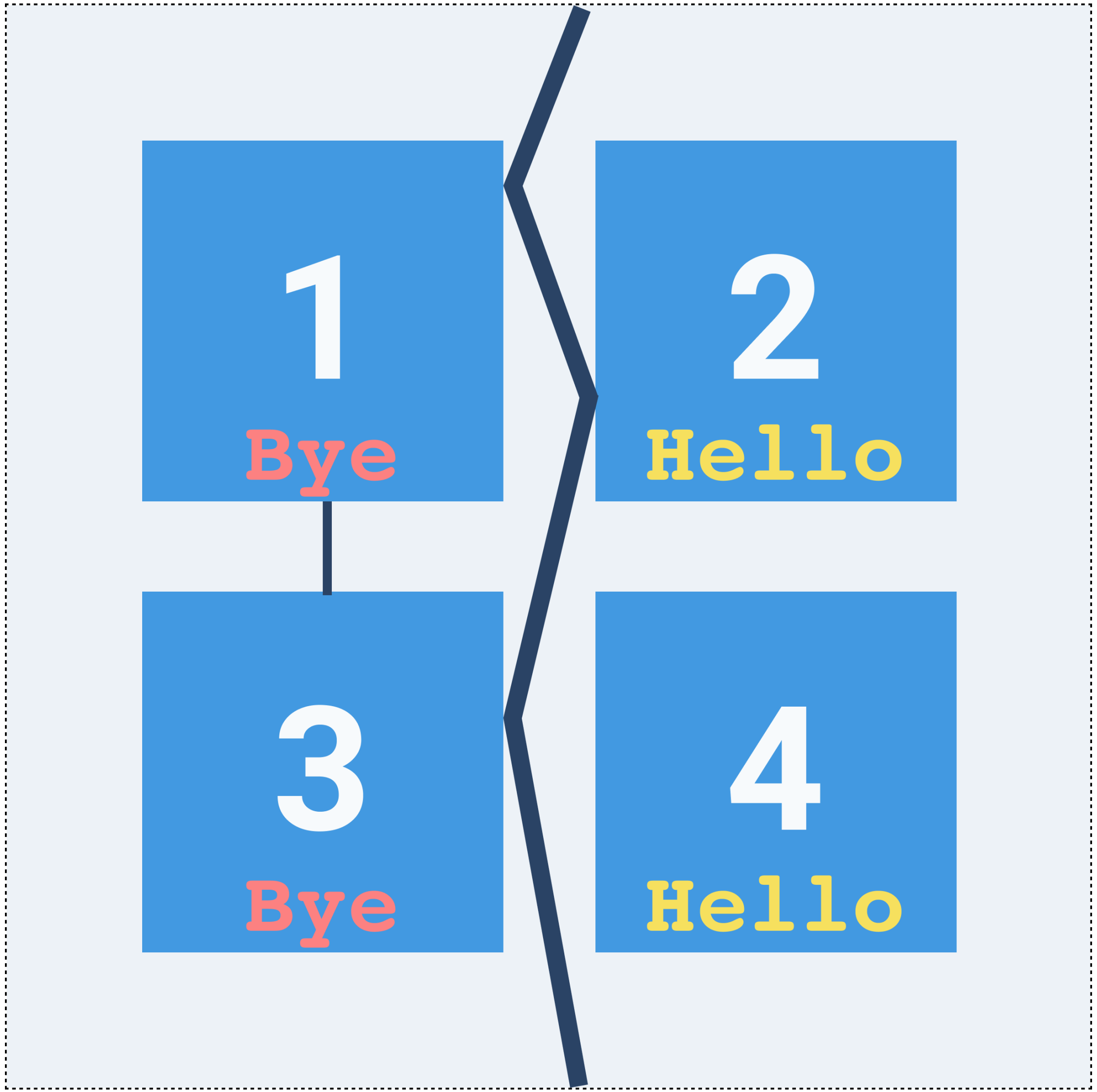

M
M

Partition tolerance

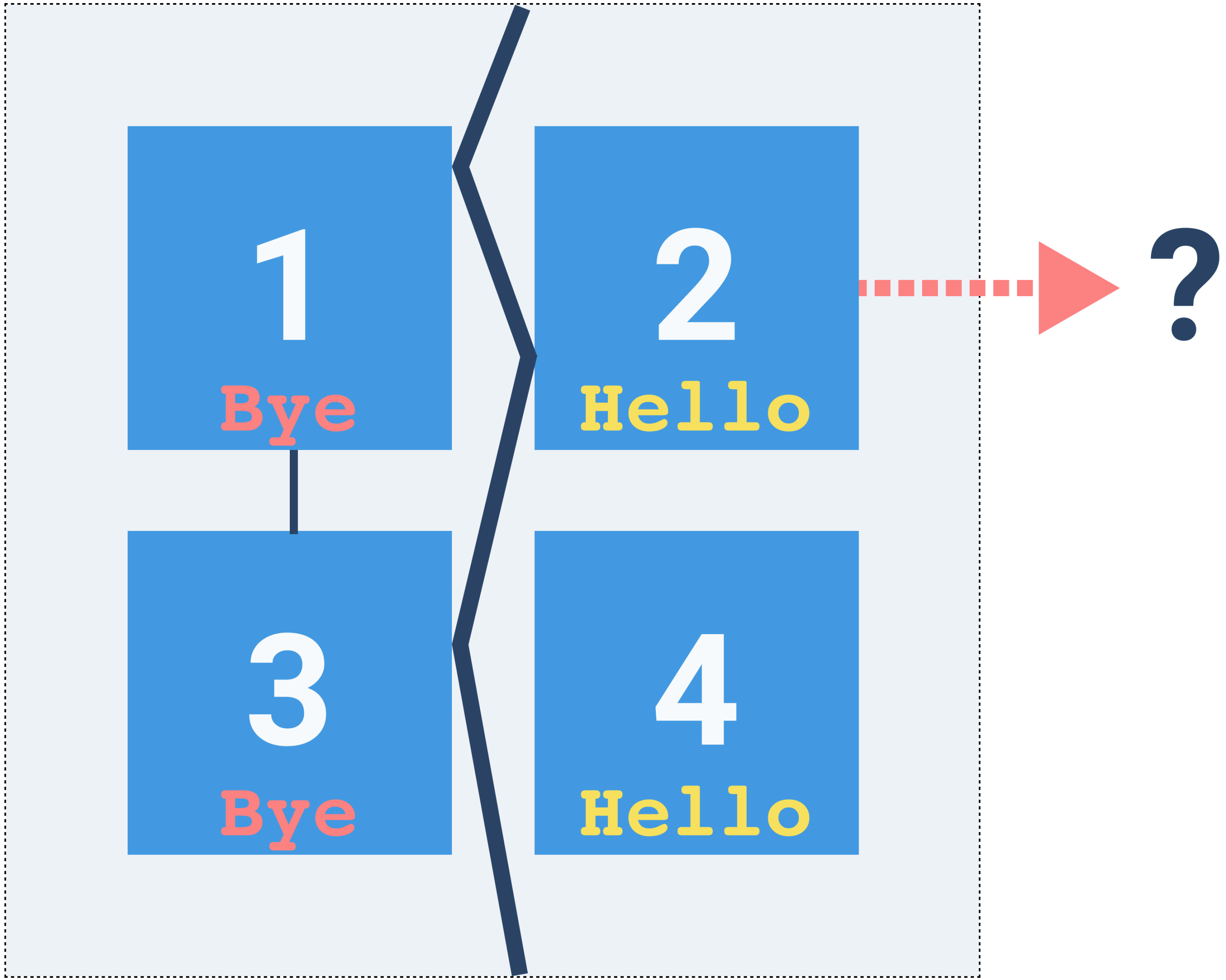
System is up and running, even though connections between nodes are severed

You can only have 2 of the 3 properties

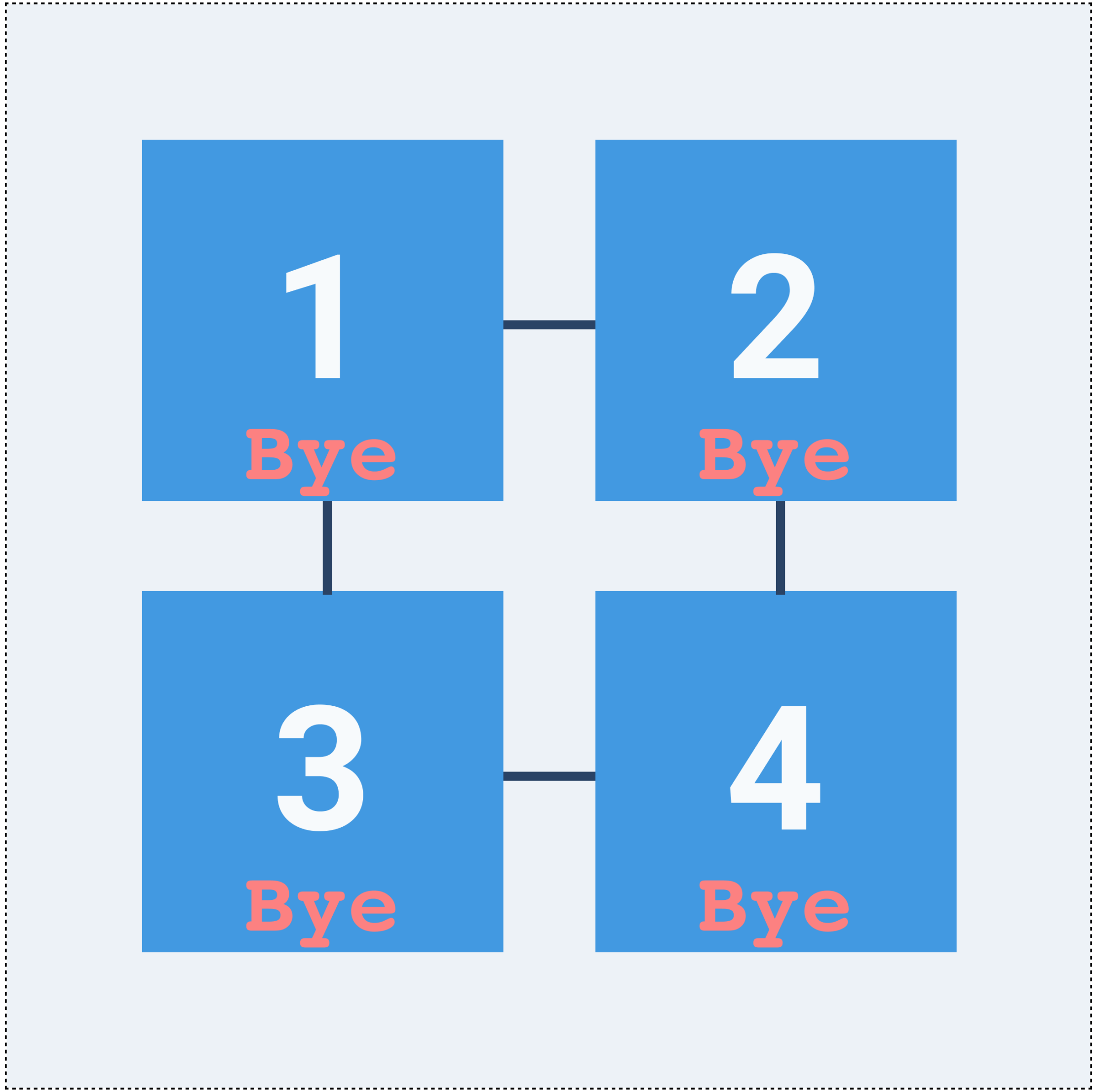
Partition Tolerant + Available



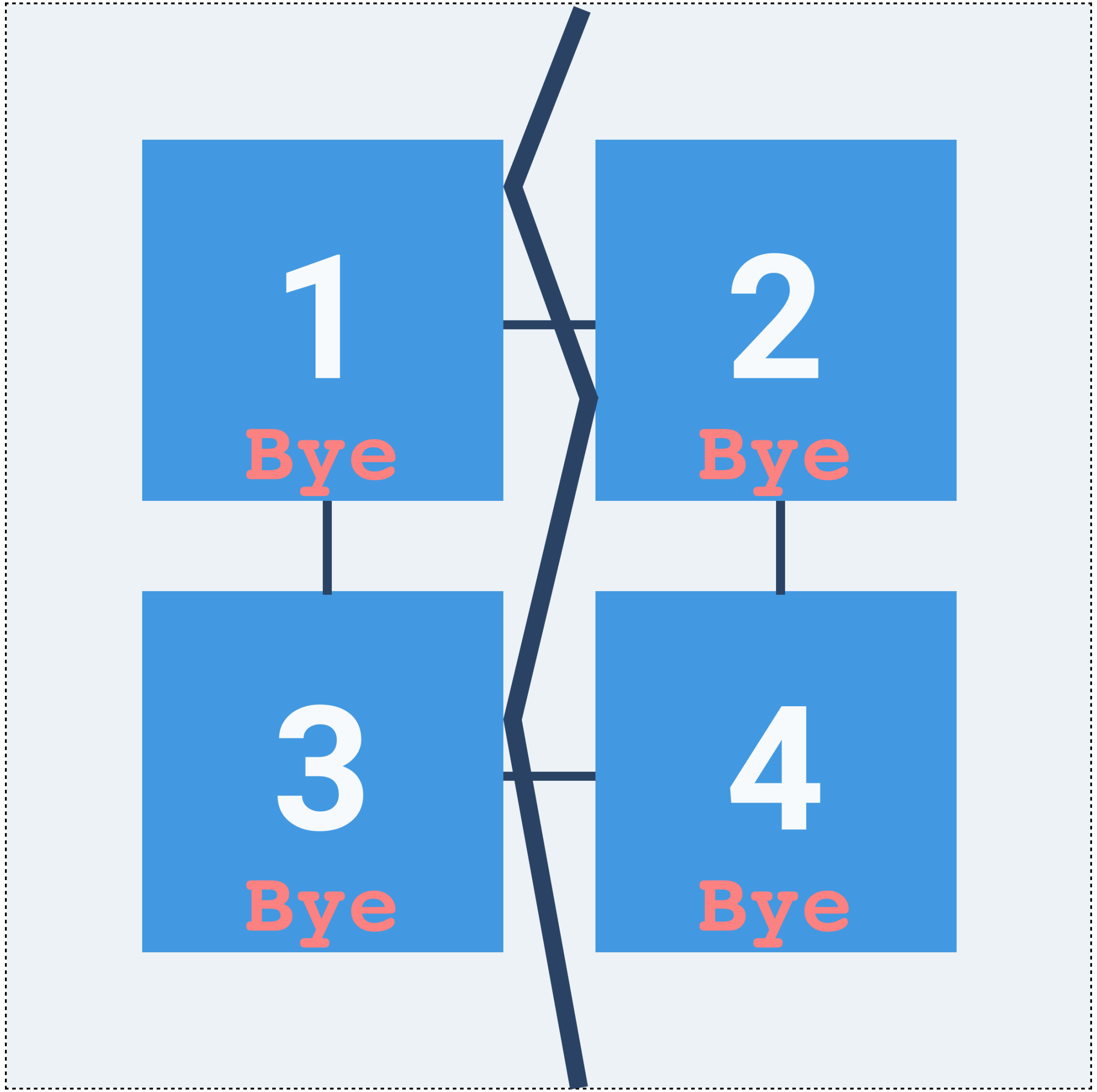
Partition Tolerant + Consistent



Consistent + Available



Consistent + Available



CAP Theorem

- Make compromises
- Partitions will always exist
- Optimize for Consistency or Availability

Fallacies of Distributed Systems

8 Fallacies of Distributed Systems

1. Network is reliable
2. Latency is zero
3. Infinite bandwidth
4. Network is secure
5. Topology does not change
6. There is one administrator
7. Transport cost is zero
8. Network is homogeneous

Network is reliable

Latency* is zero

*how much time goes by until data is received

Infinite bandwidth*

*maximum throughput

Network is secure

Topology does not change

There is one administrator

Transport cost is zero

Network is homogeneous*

*of the same or a same kind

The Twelve-Factor App

The Twelve-Factor App

- Introduced by engineers at Heroku
- Derived from best practices for app development in the cloud
- Cloud deveopment evolved since, but principles still apply

https://www.12factor.net/

CODEBASE

One codebase tracked in revision control, many deploys

DEPENDENCIES

Explicitly declare and isolate dependencies

CONFIGURATION

Store configuration in the environment

BACKING SERVICES

Treat backing services as attached resources

BUILD, RELEASE, RUN

Strictly separate build and run stages

PROCESSES

Execute the app in one or more stateless processes

PORT BINDING

Export services via port binding

CONCURRENCY

Scale out via the process model

DISPOSABILITY

Maximize robustness with fast startup and graceful shutdown

DEV/PROD PARITY

Keep development, staging, and production as similar as possible

or use one environment...

LOGS

Treat logs as event streams

ADMIN PROCESSES

Run admin and management tasks as one-off processes

Cloud-Native Building Blocks

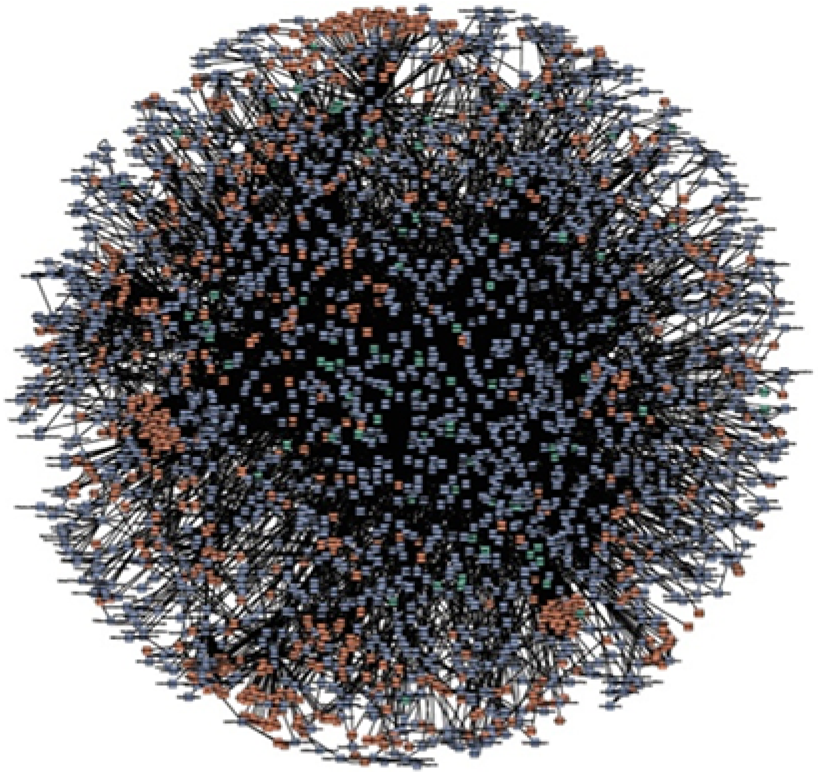
Microservices vs. Containers vs. Functions

- Microservices = architectural style
- Functions & Containers = technologies serving a particular purpose

Understand how to best use functions & containers, together with eventing/messaging technologies to design, develop and operate cloud-native microservices-based applications

Microservices

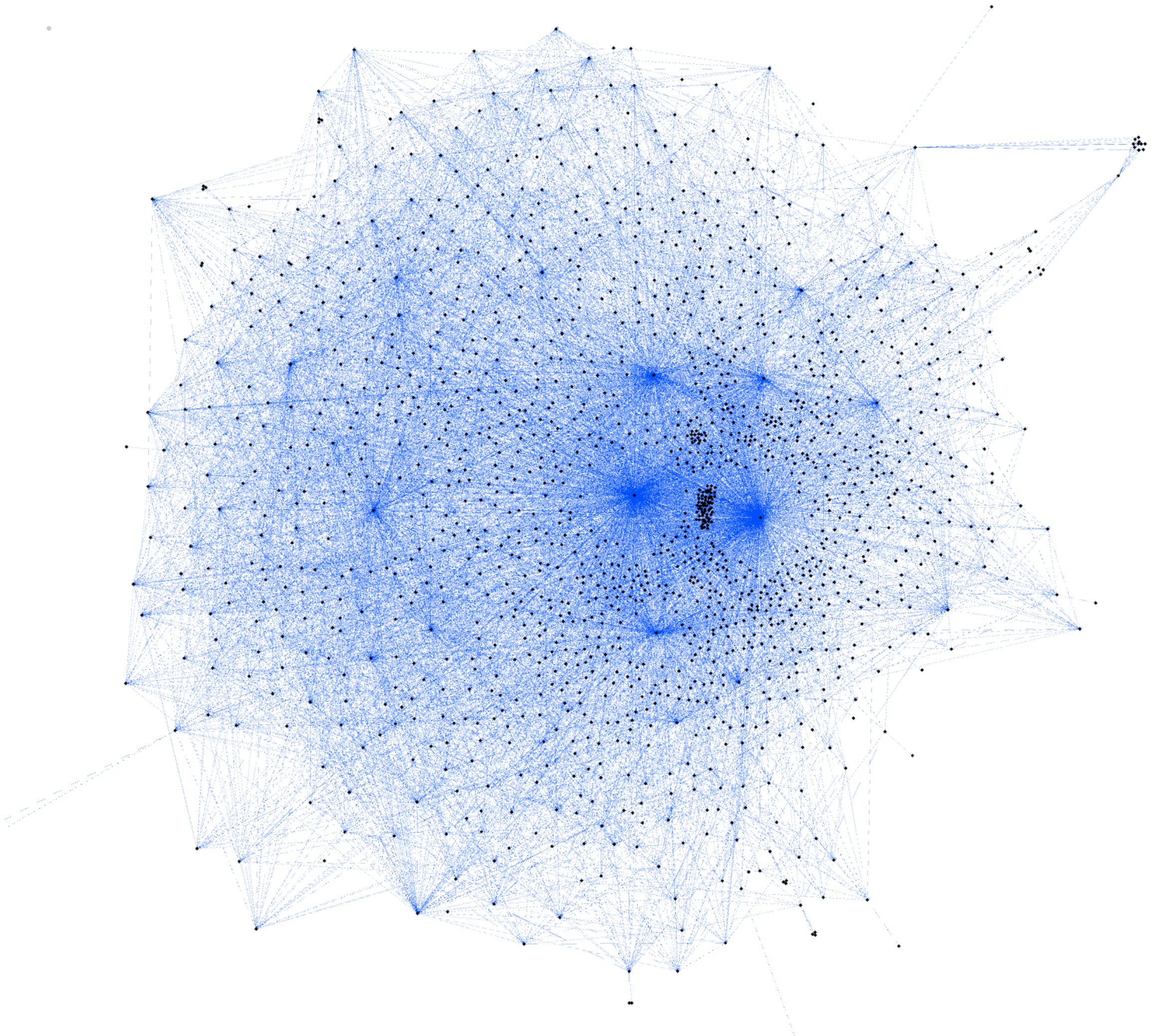
- Service-oriented architecture
- Loosely coupled services
- Organized around business capability



amazon.com

NETFLIX

https://www.appcentrica.com/the-rise-of-microservices/



https://twitter.com/JackKleeman/status/1190354757308862468

Microservices

- Smaller code bases
- Managed by independent teams
- Independently deployable
- Single, well-defined task
- Communication through APIs
- Own tests, builds, data, deployments

Benefits

- Fast(er) verification, deployment, and releases
- Easier to deliver new value
- Use the best tools/frameworks/languages for the job
- Move quicker, faster ramp up time, focus on smaller piece
- One rotten apple won't "poison" other apples
- Able to scale services at different rates
- Easier to measure and observe individual services, specific functionality

Challenges

- Complexity - fallacies of distributed systems
- Decentralized data makes transactions difficult - need to use different approaches to data management
- Performance - network adds overhead
- Lack of tools for development and testing
- Versioning, backward and forward compatibility
- Inconsistent naming, types, values, etc. when logging and monitoring
- Service dependency management
- Service availability

Serverless Computing

Serverless Computing

Functions as a Service (FaaS)

- Run in stateless, event-triggered, ephemeral containers
- Code is running without managing servers or long-lived server apps

Backend as a Service (BaaS)

- Used by single-page web apps, mobile apps
- Uses 3rd party cloud-hosted services
- Highly scalable

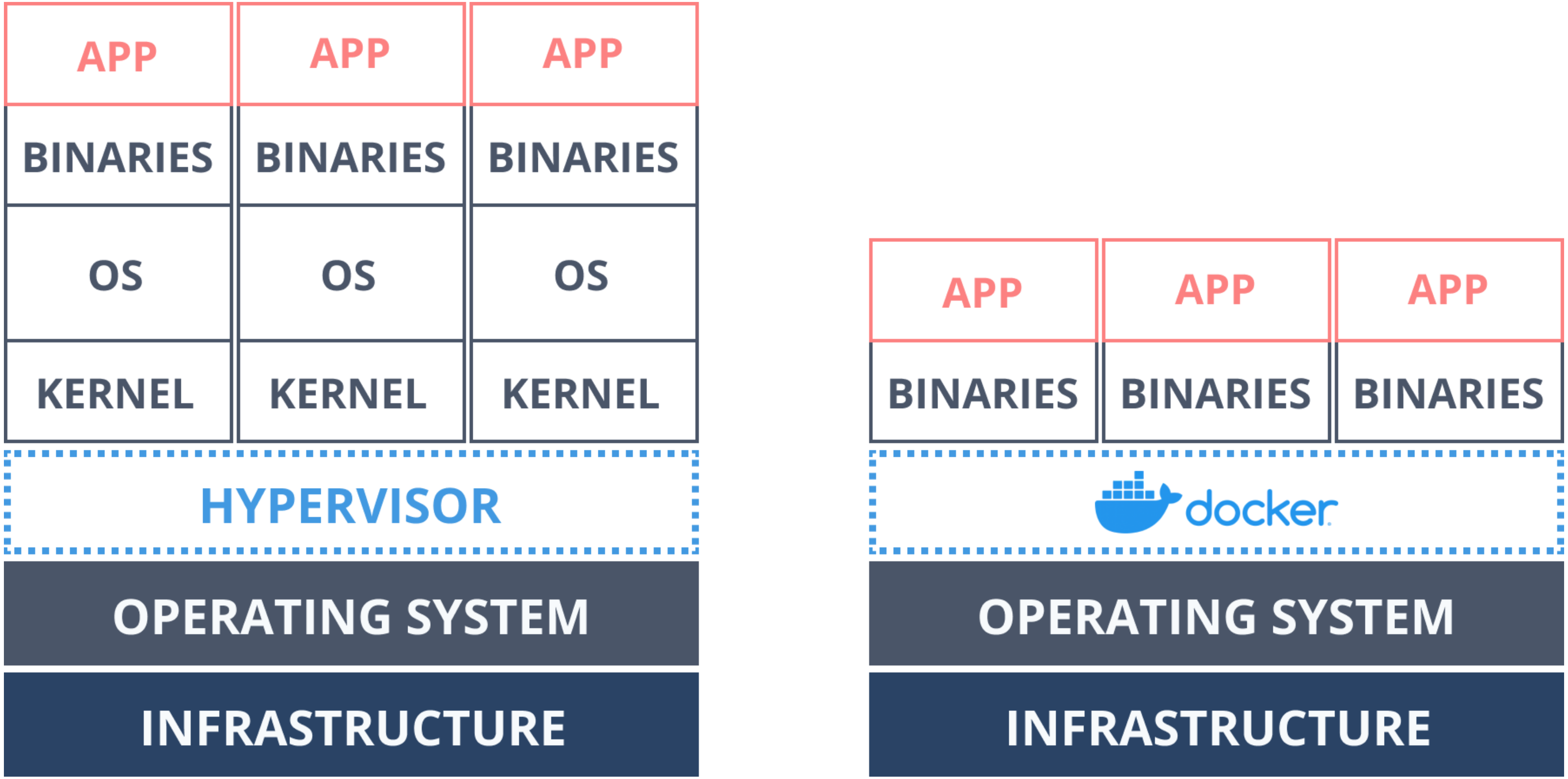
Serverless Computing

Serverless application

- Scale and infrastructure managed by the cloud provider
- Auto-scaling is based on the load
- Event-driven programming model
- Pay per execution (CPU time consumed)
- Highly available

Containers

- "Docker containers"
- Linux containers (LXC)
  - Namespaces and control groups
- Slice up the OS, so it can run securely multiple applications
  - Namespaces: allows OS to be sliced up and create isolated workspaces
  - Control groups: gives fine-grained control over resource utilization


APP
APP
APP
BINARIES
BINARIES
BINARIES
OS
OS
OS
KERNEL
KERNEL
KERNEL
HYPERVISOR
OPERATING SYSTEM
INFRASTRUCTURE
APP
APP
APP
BINARIES
BINARIES
BINARIES
docker
OPERATING SYSTEM
INFRASTRUCTURE

Functions (FaaS)

- Function = unit of work
- Triggered by events, emitted by other functions or services
- Developers can focus on code, no need to worry about infrastructure
- Use for short-lived, independent tasks

Functions as a Service (FaaS)

- AWS Lambda, Azure functions, Google Cloud Functions, Oracle Functions

Functions (FaaS) vs. Containers

Functions (FaaS)

- Does one thing
- Just code
- Respond to one kind of event
- Scales down to 0

Containers

- Does more than one thing
- Declared with e.g. Dockerfile
- Can respond to more than one kind of event
- Long running

Function Scenarios

Parallel execution scenarios

- Functions don't need to communicate with another functions
- Generating things, updating records, map-reduce functions, batch processing

IoT

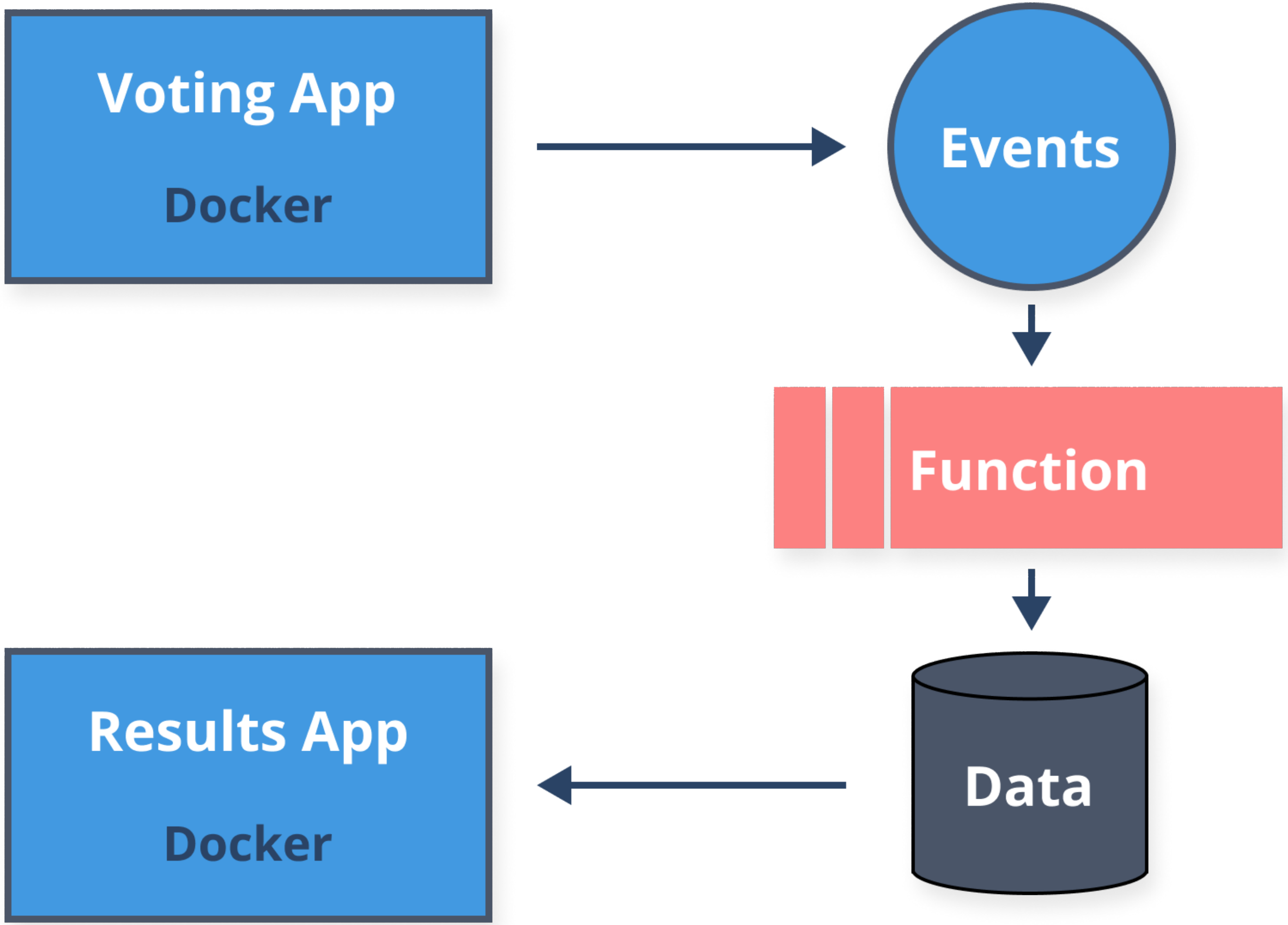
- For orchestration tasks: message → IoT hub → function

Full applications

- Azure Durable Functions
- AWS step functions

Considerations for Using Functions

- Limited lifetime of a function
  - Not suited for long-running tasks
- No usage of specialized hardware
- Stateless and not directly network addressable
- Local development/debugging
- Economics


Voting App
Docker
Events
Function
Data
Results App
Docker

What is Docker?

Docker Engine (daemon) + CLI

Dockerfile

```
FROM ubuntu:18.04
WORKDIR /app
COPY hello.sh /app
RUN chmod +x hello.sh
RUN apt-get update
RUN apt-get install curl -y
CMD ["./hello.sh"]
```

Docker image

- Collection of layers from `Dockerfile` (one layer per command)
- Layers are stacked on top of each other
- Each layer is a delta from the layer before it
- All layers are read-only

Docker image

Image names

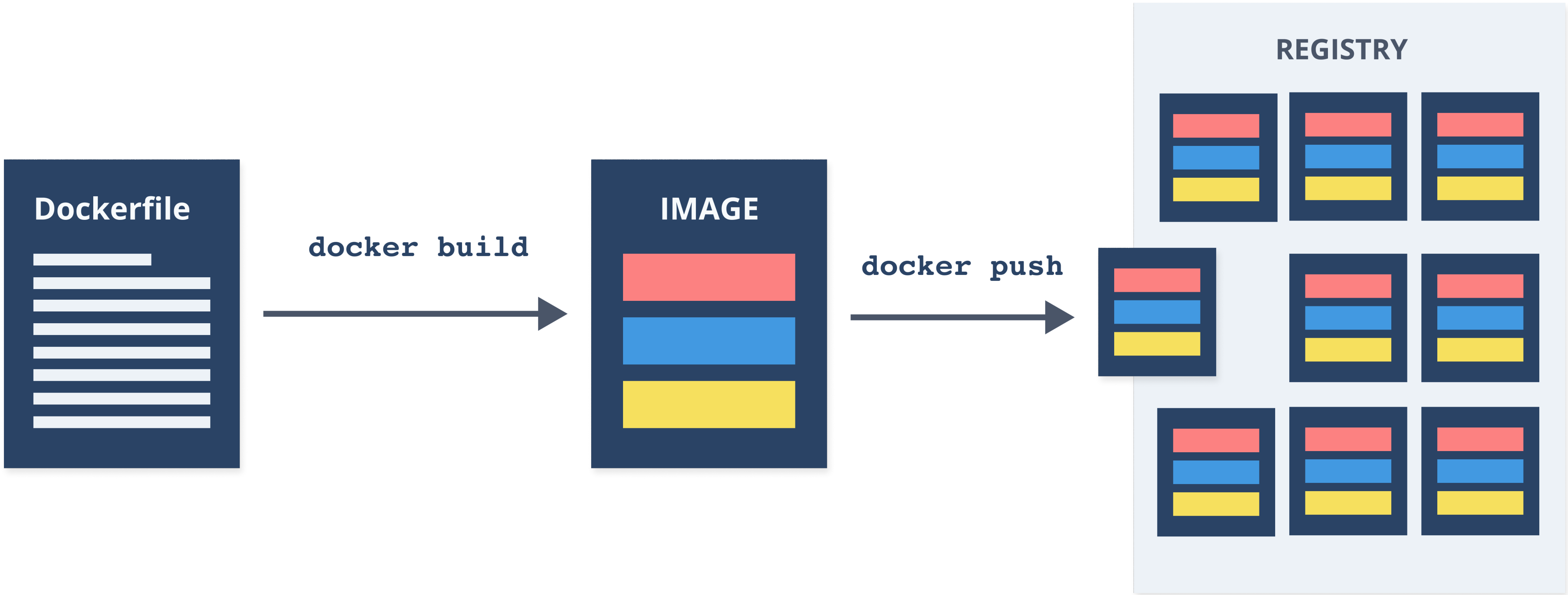
- Image = repository + image name + tag

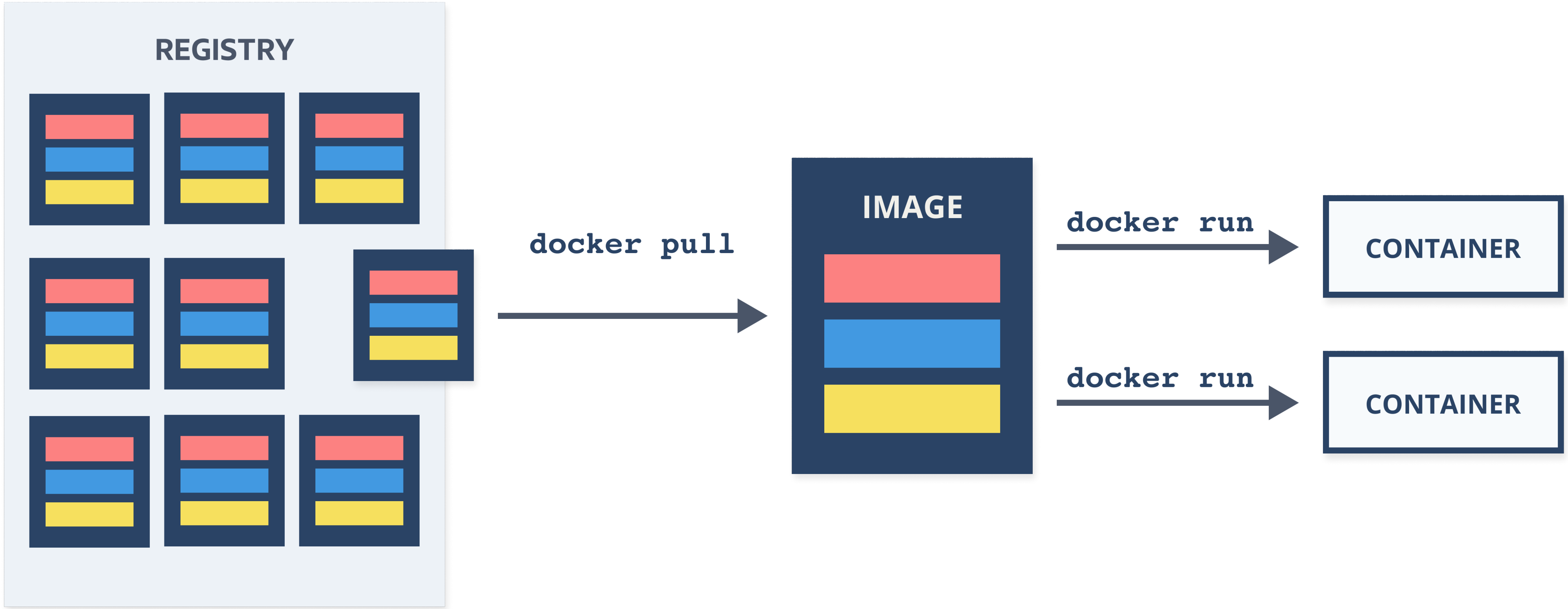
```
mycompany/hello-world:1.0.1
```

- All images get a default tag called *latest*
- Tag = version or variant of an image

Docker Registry

- Place to store your Docker images
  - Public and private repositories
  - Docker Hub (https://hub.docker.com)
  - Every cloud provider has its own
- You can also store images locally, on your Docker host


Dockerfile
docker build
IMAGE
docker push
REGISTRY


REGISTRY
docker pull
IMAGE
docker run
CONTAINER
docker run
CONTAINER

Exercises - Docker

https://github.com/peterj/velocity-berlin-2019

Container Orchestration

- Provision and deploy containers onto nodes
- Resource management/scheduling containers
- Health monitoring
- Scaling
- Connect to networking
- Internal load balancing

Kubernetes Overview

- Most popular choice for cluster management and scheduling container-centric workloads
- Open source project for running and managing containers

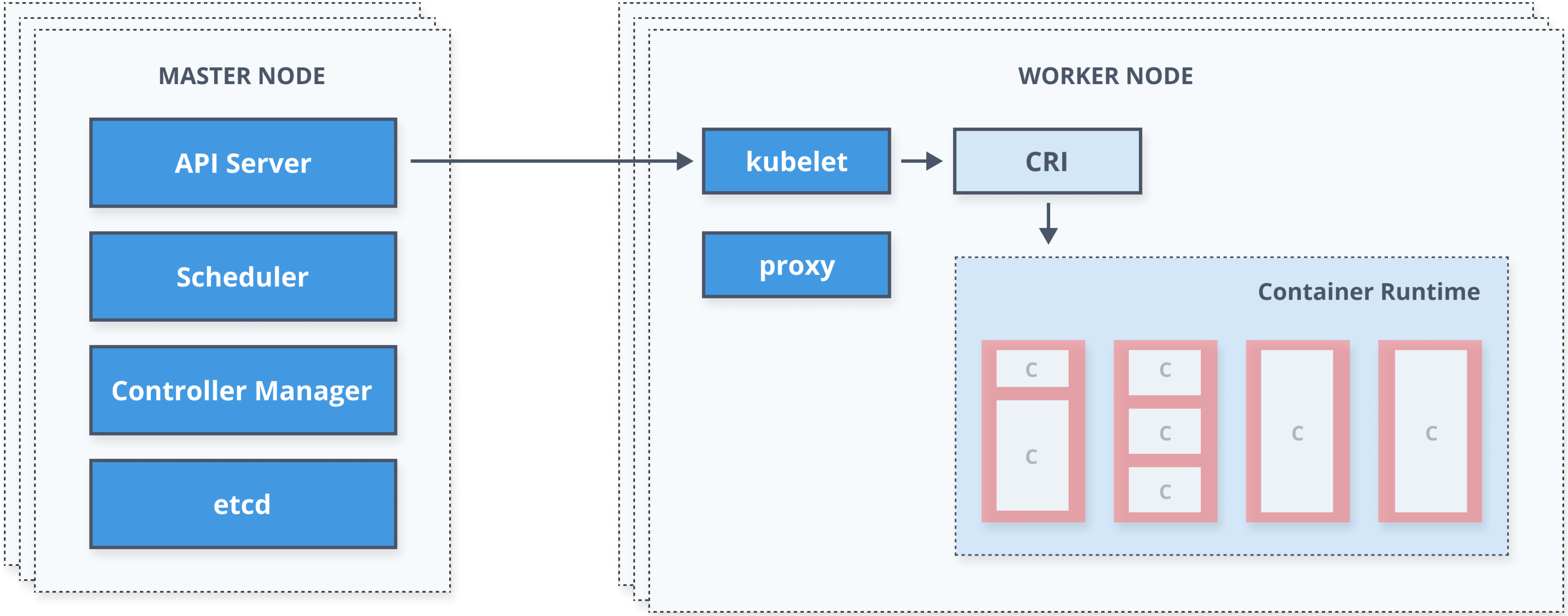
Definitions

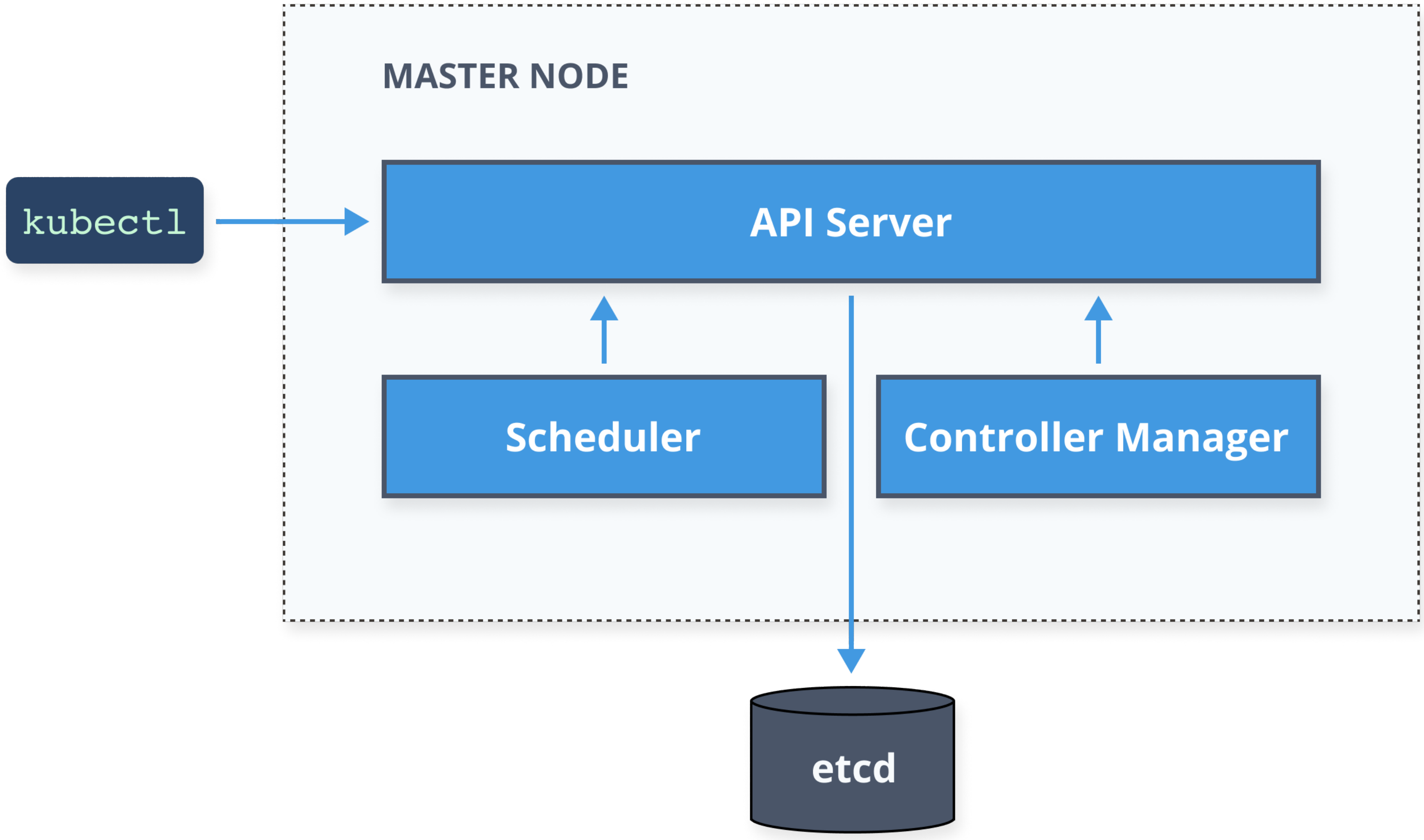
Portable, extensible, open-source platform for managing containerized workloads and services

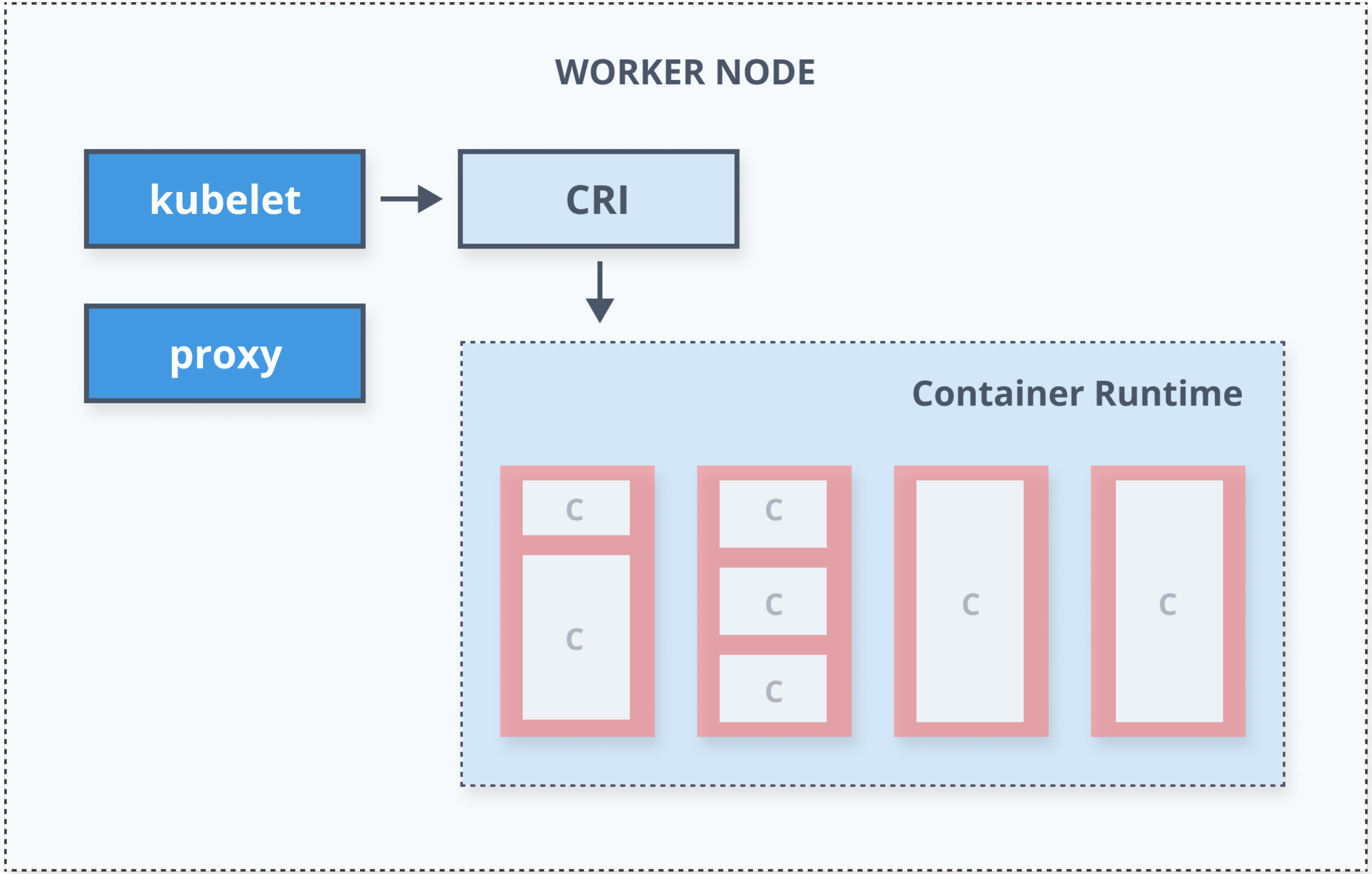
Container-orchestration system for automating application deployment, scaling, and management

https://kubernetes.io

Kubernetes Architecture




MASTER NODE
kubectl
API Server
Scheduler
Controller Manager
etcd


WORKER NODE
kubelet
CRI
proxy
Container Runtime
C
C
C
C
C
C
C

Kubernetes Building Blocks (1/2)

Pods

- Collection of containers that share storage, network, volumes
- All containers scaled together as a unit
- Unique IP

ReplicaSets

- Controller for pods
- Allows scaling pods (up and down)

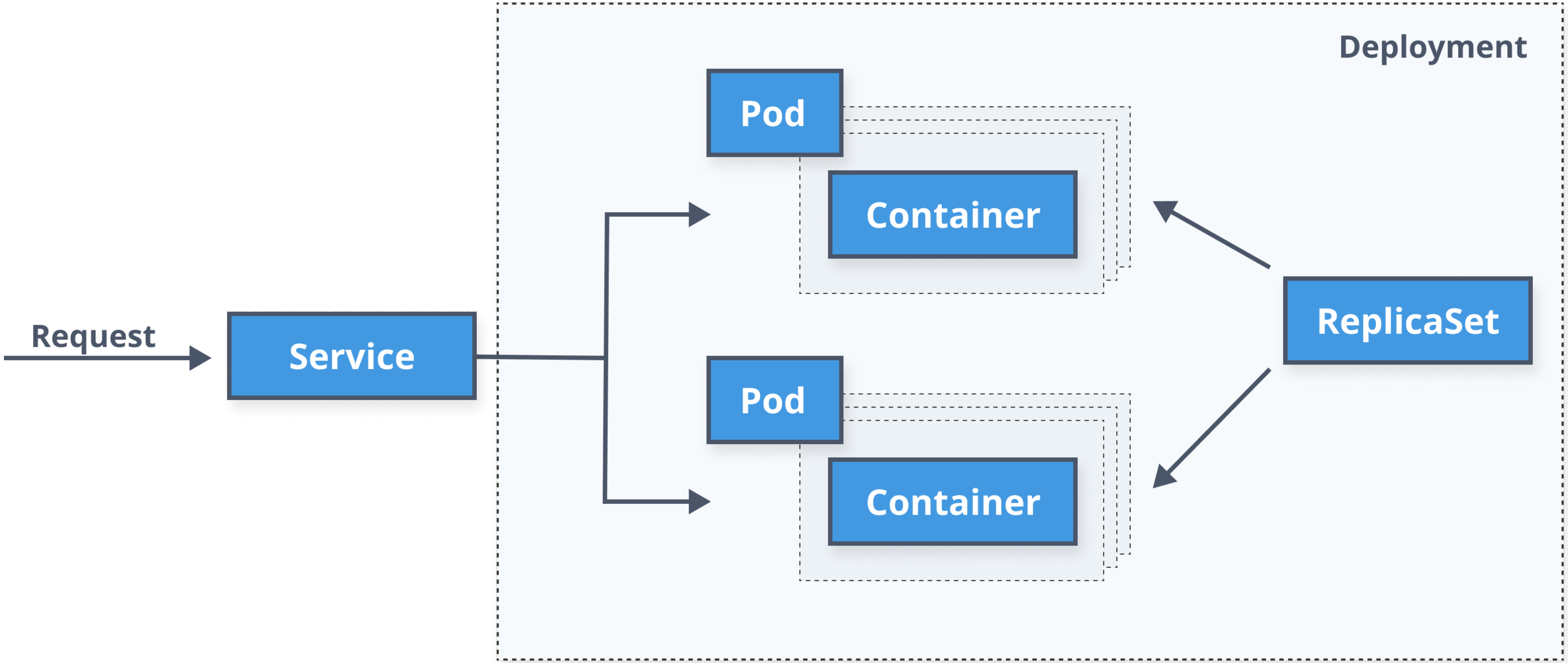
Kubernetes Building Blocks (2/2)

Deployments

- Manages updates, does controlled roll-out

Services

- Defines a logical set of pods and endpoint to access them
- Manages a list of endpoints (pod IPs)


Deployment
Pod
Container
Pod
Container
ReplicaSet
Request
Service

Configuration and Secret Management

- Store configuration in the environment
- Design your services so you can easily add/remove config settings
- Come up with a configuration schema for all your services
  - Simplifies testing
- Use files if number of settings is too big
  - `./myservice --arg1 --arg2 --arg3 ...`
  - `./myservice config.json`
- Kubernetes: `ConfigMap`

Configuration and Secret Management

- Secret = anything with sensitive info
  - passwords, API keys, certificates
- Kubernetes: `Secret`
- Consider secret management solutions (*HashiCorp Vault, Microsoft Key Vault)

https://learn.hashicorp.com/vault

https://azure.microsoft.com/en-us/services/key-vault/

Configuration and Secret Management

- Kubernetes: *Helm
- Manage, install, upgrade a collection of templatized YAML files as a single unit
- Uses `values.yaml` for per-environment deployments

https://helm.sh

Exercises - Kubernetes

https://github.com/peterj/velocity-berlin-2019

Designing Cloud-Native Applications

Approach

Five key areas

- Operational excellence
- Security
- Reliability and availability
- Scalability and cost

Operational Excellence (1/2)

Automate everything (enviroments, deployments)

- Infrastructure as Code (IaC)
- Track changes to your environment
- Minimize errors during provisioning and deployment
- Terraform, Azure Resource Manager, AWS CloudFormation

Monitor everything

- Learn about your application and environment and how it's being used
- Consistent monitoring across the stack

Operational Excellence (2/2)

Document everything

- OpenAPI spec
- Automatically generate documentation

Design for failure

- Failures will happen
- Testing for failures

Make incremental (and reversible) changes

Security

*9 out of 10 cybersecurity professionals are troubled by cloud security issues (esp. data loss & breaches)

Security

*9 out of 10 cybersecurity professionals are troubled by cloud security issues (esp. data loss & breaches)

However...

Security

*9 out of 10 cybersecurity professionals are troubled by cloud security issues (esp. data loss & breaches)

However...

Cloud environments are safer than most on-premises environments

Security

*9 out of 10 cybersecurity professionals are troubled by cloud security issues (esp. data loss & breaches)

However…

Cloud environments are safer than most on-premises environments

BUT

Security

*9 out of 10 cybersecurity professionals are troubled by cloud security issues (esp. data loss & breaches)

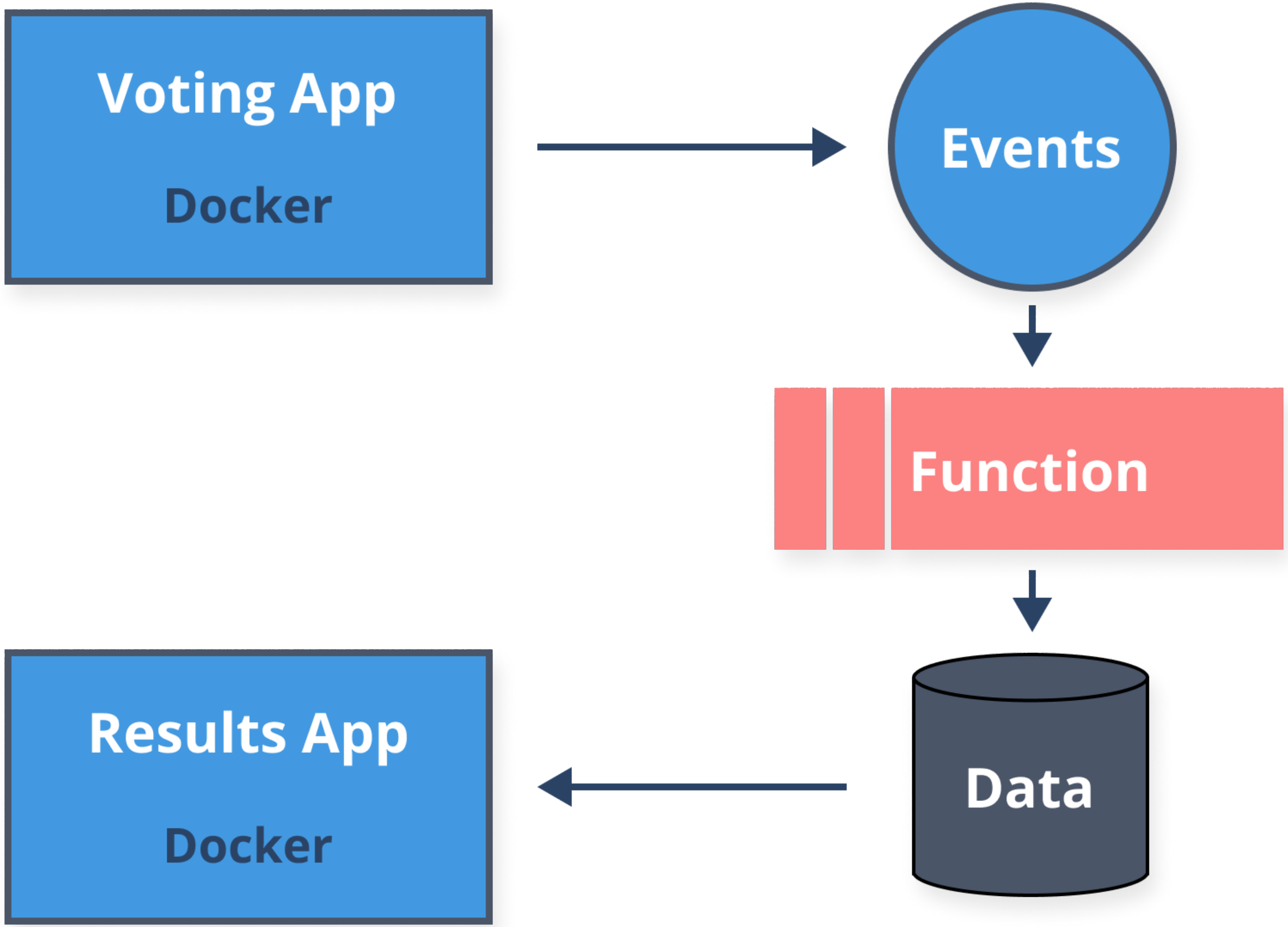
However...

Cloud environments are safer than most on-premises environments

BUT

That doesn't mean you can ignore the security

*https://pages.cloudpassage.com/rs/857-FXQ-213/images/2018-Cloud-Security-Report%20%281%29.pdf


Voting App
Docker
Events
Function
Data
Results App
Docker

Defense-in-depth approach (1/2)

Source code

- Secure (private) repository (track and audit access)
- Vulnerability checks as part of the continuous integration

Container image

- Image contains the bare-minimum needed

Container registry

- Use private registry (track and audit access)
- Image vulnerability scanning (e.g. Twistlock)

Defense-in-depth approach (2/2)

Pods

- Images pulled from approved registries only
- Use pod security policies to control volumes, priviledged containers, host ports, networking, …

Cluster/Orchestrator

- Secure access to the cluster
- Enable RBAC (Role-based access control)
- Enable audit logs

Reliability and availability

- Reliability: App still works in an acceptable way, even in the presence of failures
- Can recover from failures
- Retries, timeouts, circuit breakers
- Testing is a must
- Availability: App is available for a certain amount of time

Scalability and cost

- How to scale in a cost-efficient way?
- More nodes in the cluster?
  - Horizontal node autoscalers (can be slow)
  - Burst into container as a service?
- Experiment during development to find a better solution

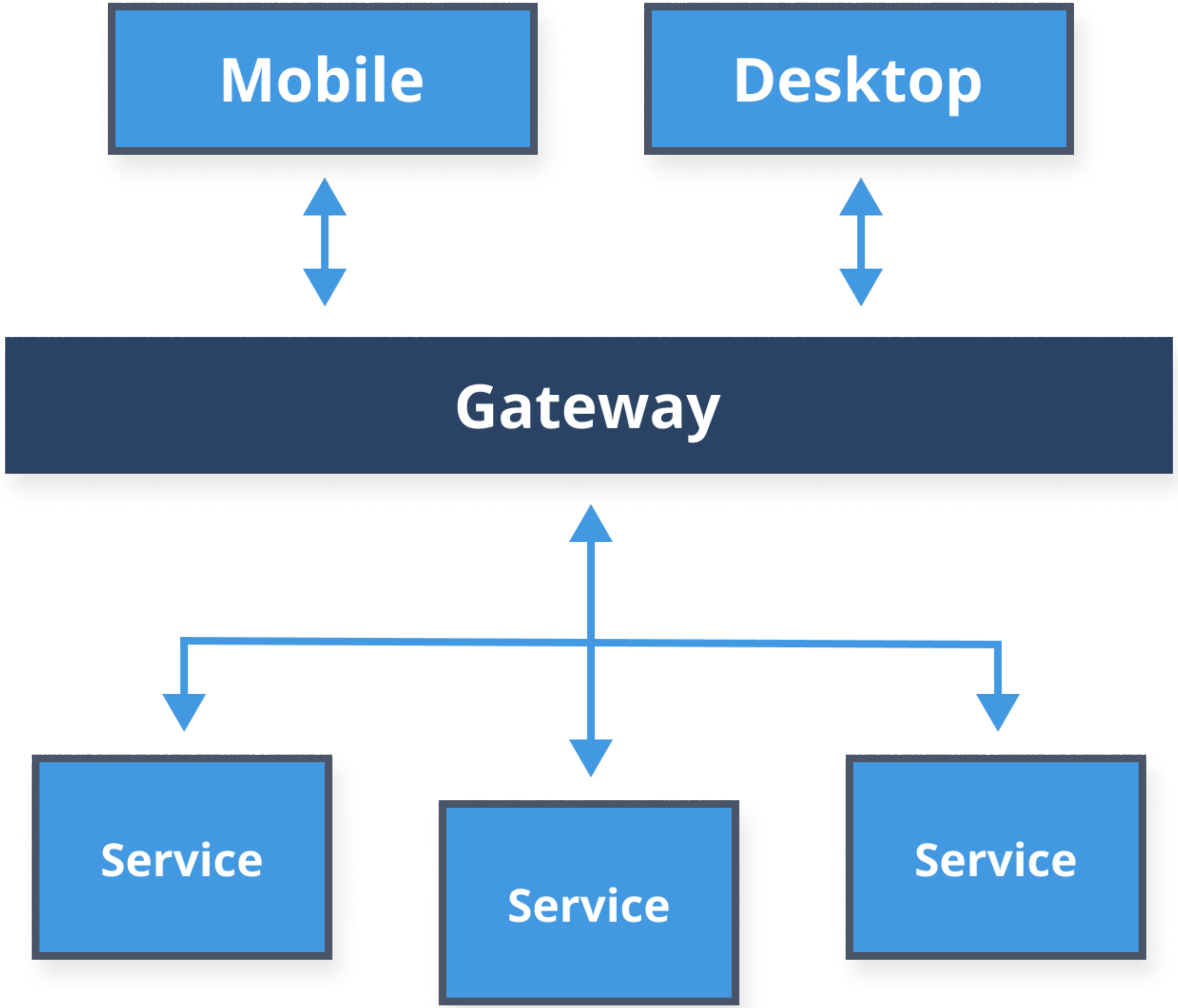
API Gateway Pattern

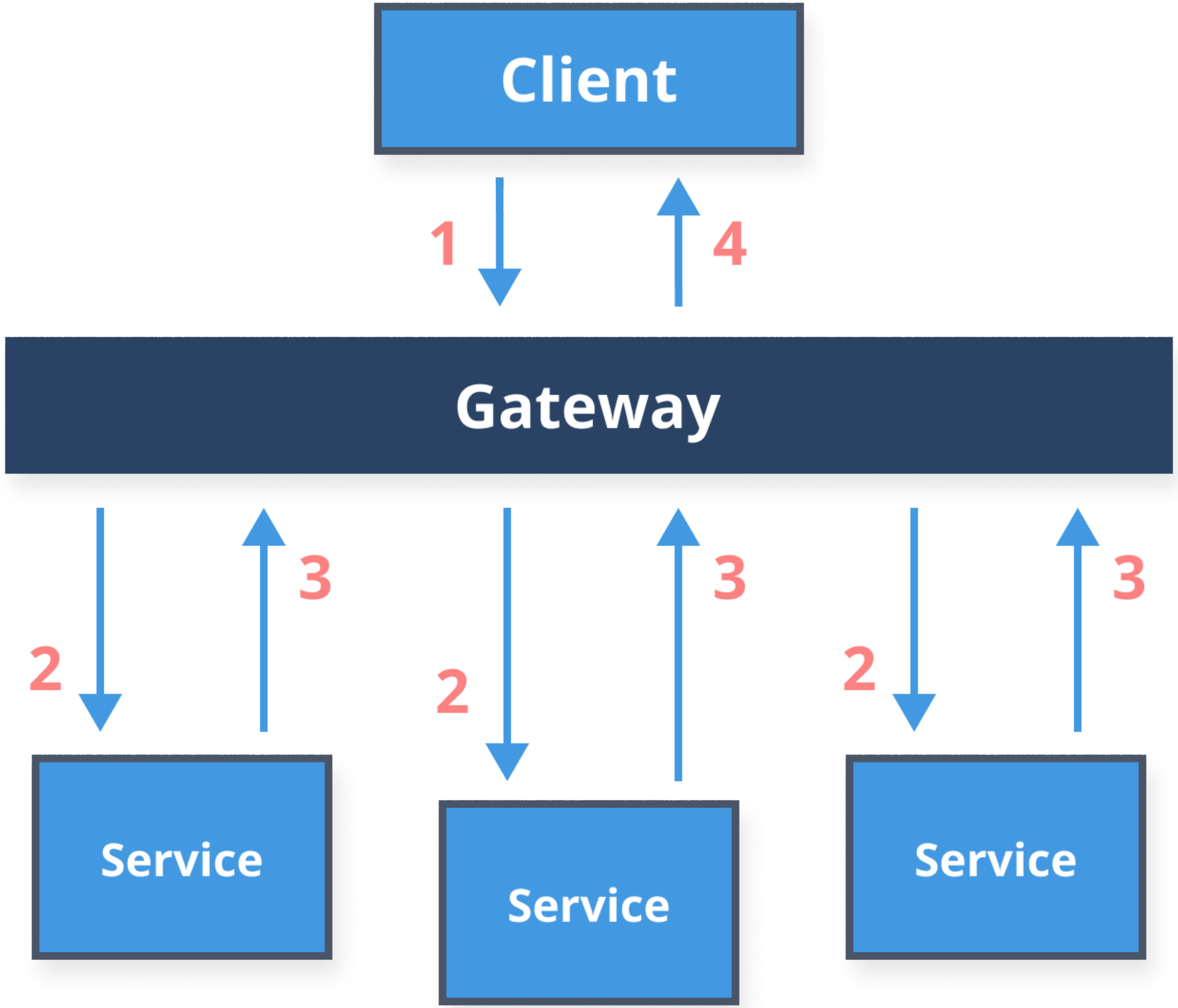
API Gateway Pattern

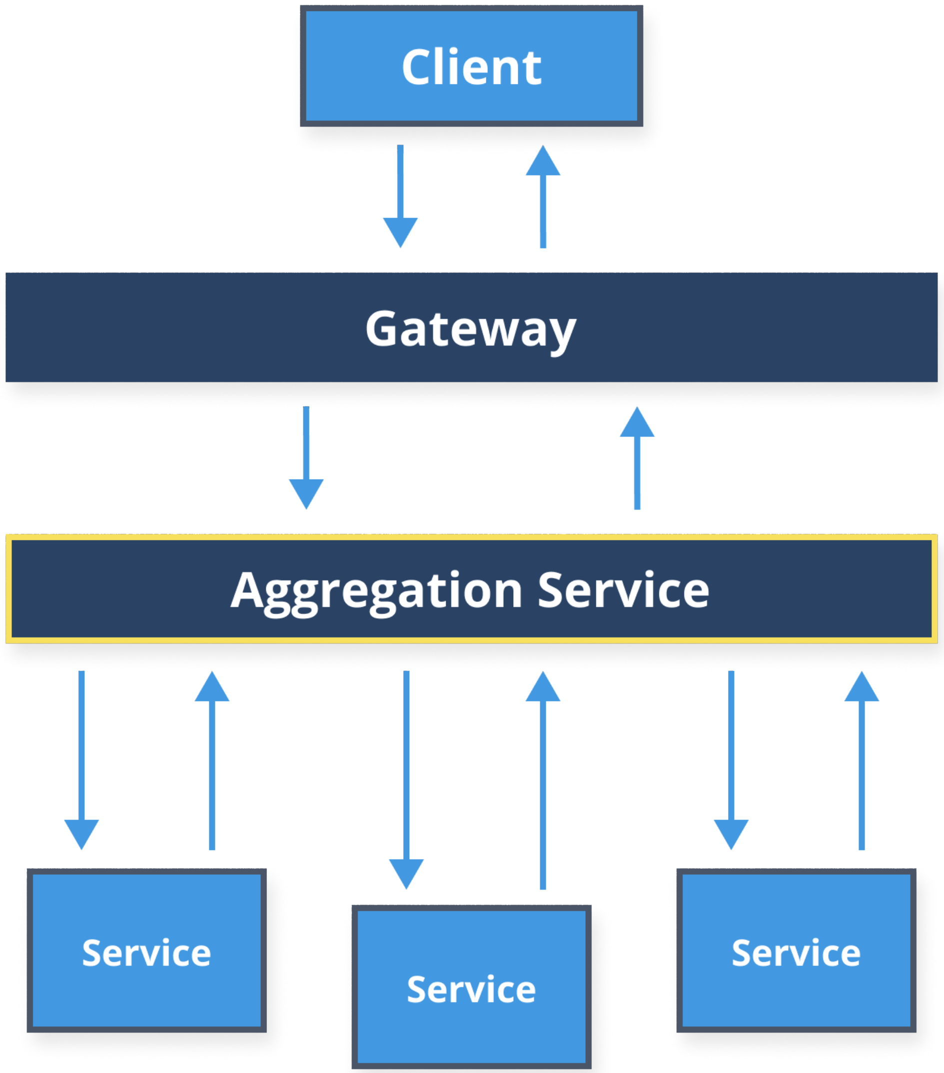
- Single entry point
- Handles incoming requests:
  - Routing
  - Aggregating
  - Offloading

Use cases:

- SSL termination/authentication
- Caching
- Rate limiting, retry policies, circuit breakers
- Compression


Mobile
Desktop
Gateway
Service
Service
Service


Client
1
4
Gateway
2
3
2
3
2
3
Service
Service
Service


Client
Gateway
Aggregation Service
Service
Service
Service

API Gateways

Reverse proxy server (*Nginx, HAProxy, Envoy*)

- Support load balancing, SSL and L7 routing
- High performance, extendable

Managed service/other API gateways

- Azure Application Gateway
- AWS API Gateway
- Apigee
- Gloo
- Kong
- Ambassador ...

If using service mesh, you can use ingress/egress controllers

API Design and Versioning

API Design and Versioning

- API is the communication interface between services
- Properly document and version your APIs
- Use standard protocols
- Transparent API evolution

API Design and Versioning

The knot ($ for clients)

- Clients tied to single version of the API
- When API changes, all consumers need to upgrade

Point-to-point ($ for maintainers)

- Keep all API versions running
- Clients migrate when they want to

Compatible versioning ($, but efficient)

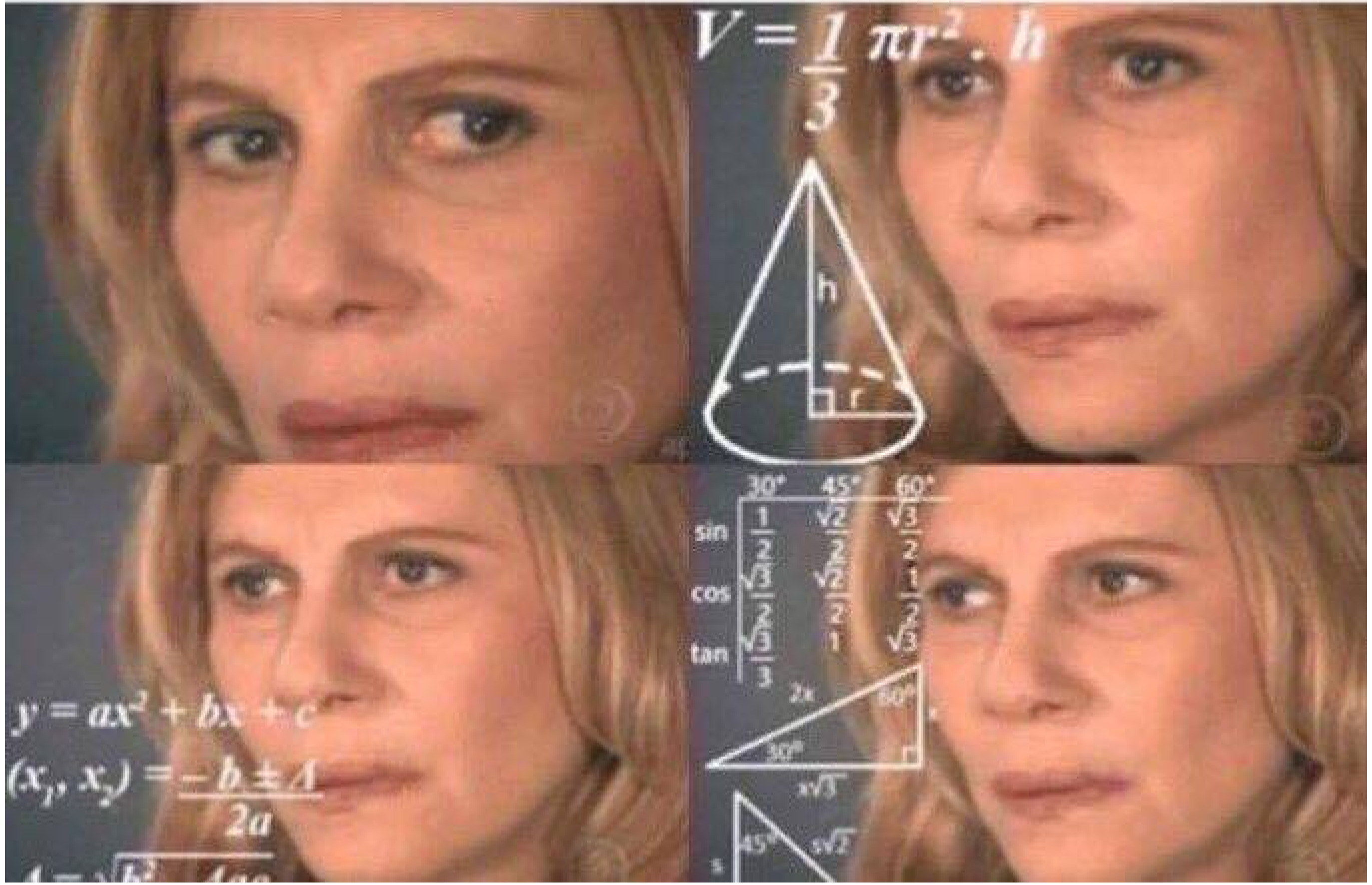
- All clients talk to the same API version
- New versions are backward compatible - deprecate old versions

https://web.archive.org/web/20180202134605/https://www.ebpml.org/blog2/index.php/2013/11/25/understanding-the-costs-of-versioning

$$Cost = S_1 + \sum_{i=2}^{i=n} \left(V_i + \sum_{j=1}^{j=i-1} (U+T)_j \right)$$

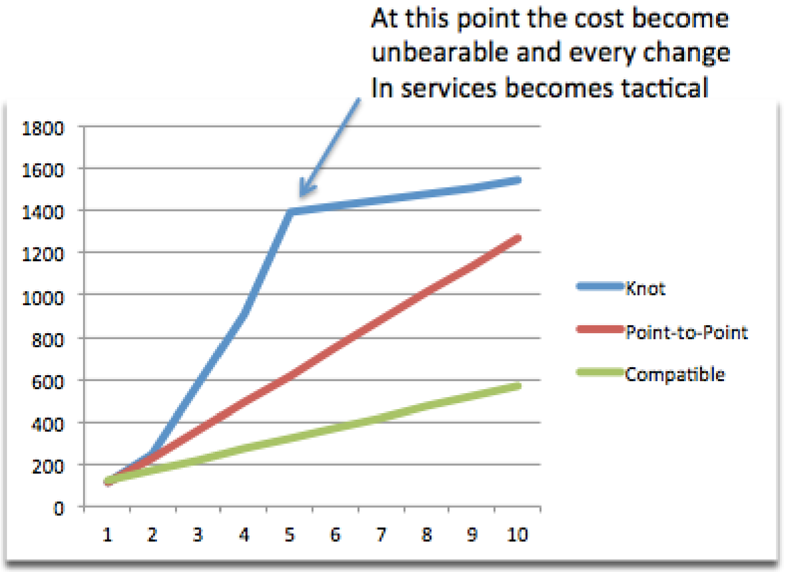
$$Cost = \sum_{i=1}^{i=n} S_i + \sum_{j=1}^{j=i-1} (U+T)_j$$

$$Cost = S_1 + T_1 + A_1 + \sum_{i=2}^{i=n} (V_i + C_i)$$



V = 1/3 πr². h
h
r
30° 45° 60°
sin
cos
tan
y = ax² + bx + c
(x₁, x₂) = −b ± Δ / 2a
2x
60°
30°
x√3
45°
s√2
s

Compatible versioning wins



Versions	1	2	3	4	5
Knot	110	250	580	910	1390
Point-to-Point	110	230	360	490	620
Compatible	120	170	220	270	320
Gains (Comp. vs P2P)	-9%	26%	39%	45%	48%

API Design and Versioning

- No specific versioning with REST
- No clear/best approach

Ideas

- No versioning: `api/users/123`
- Global/URI versioning: `/api/v1/users/123`, `/api/v2/users/123`
- Query string versioning: `/api/users/123?version=3`
- Header based: add a header e.g. `api-version: 3`
- Mime-based approach: `Accept: application/vnd.example.users.v2+json`

API change management is what matters

API Design and Versioning

Compatible versioning strategy

- APIs are backward compatible, no need to maintain different API versions
- Don't version resources, relations between them or the API itself
- Version message formats and API documentation
- Breaking changes: create new resource or use content negotiation

Backward Compatibility

New service version that supports features of an older version

- Provide sensible defaults
- Never rename existing fields or remove them
- Never make optional things required
- Mark old API endpoints as obsolete if not used anymore
- Test the combination of new/existing service version by passing old messages

Forward compatibility

Service can accept and gracefully handle requests for a later version of itself

- Ignore any additional fields
- Don't throw errors

Exercises - APIs

https://github.com/peterj/velocity-berlin-2019

Building Cloud-Native Applications

Slides: http://bit.ly/buildcnapps

Agenda - Day 2

Workshop runs from 9:00 AM to 5:00 PM

- 30 min coffee breaks (10:30AM and 1:30PM)
- 1h lunch break (12:30PM)

Multiple sections - theory + exercises

- Service communcation
- Developing, testing, and operating cloud-native apps
- Service mesh

Service Communication

Smart endpoints and dumb pipes

Basic, async communication over complex integration platform

Service Communication

External communication (North-South traffic)

- Communication from/to external services

Internal communication (East-West traffic)

- Service-to-service communication (e.g. within a cluster)

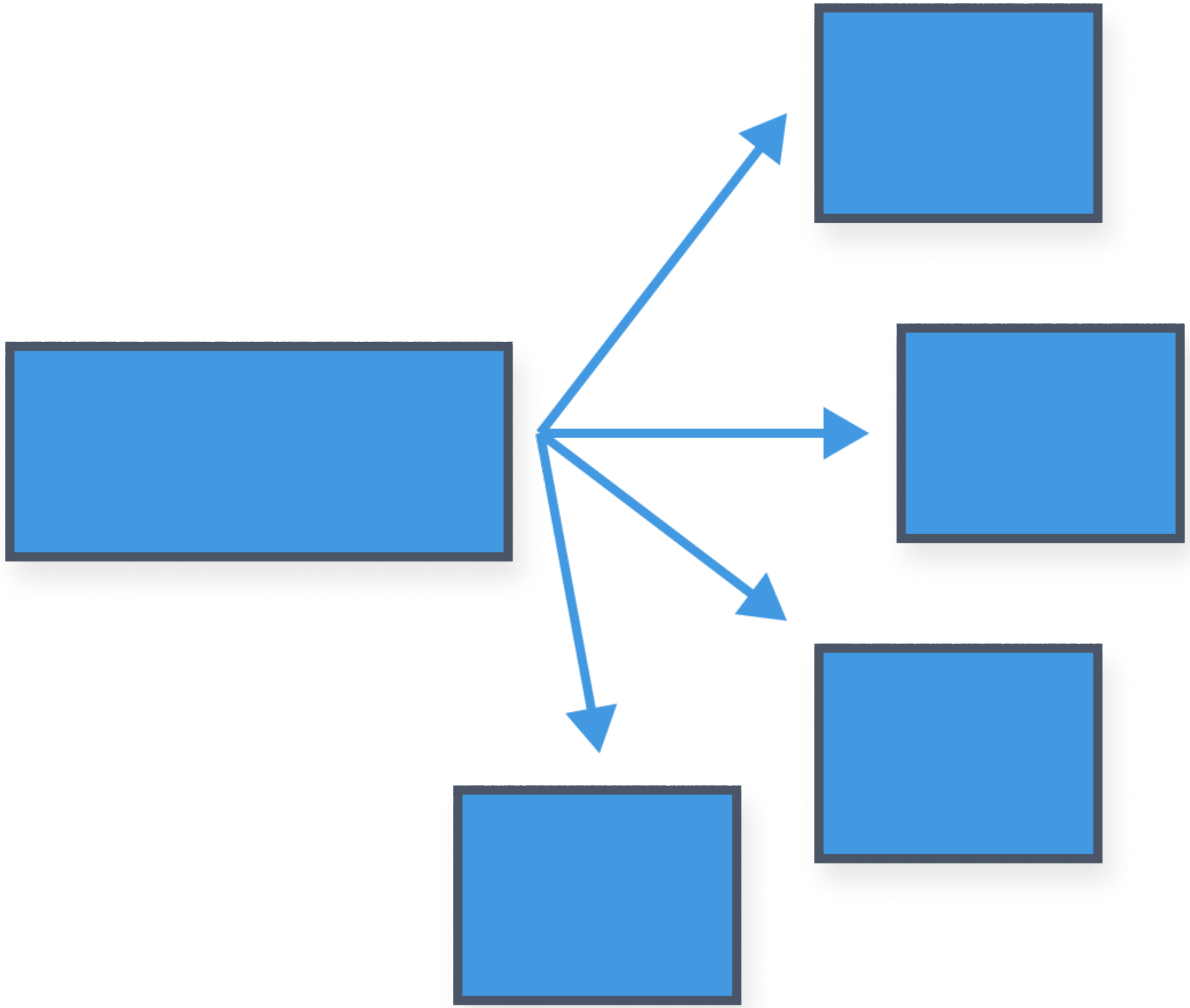
Synchronous and asynchronous

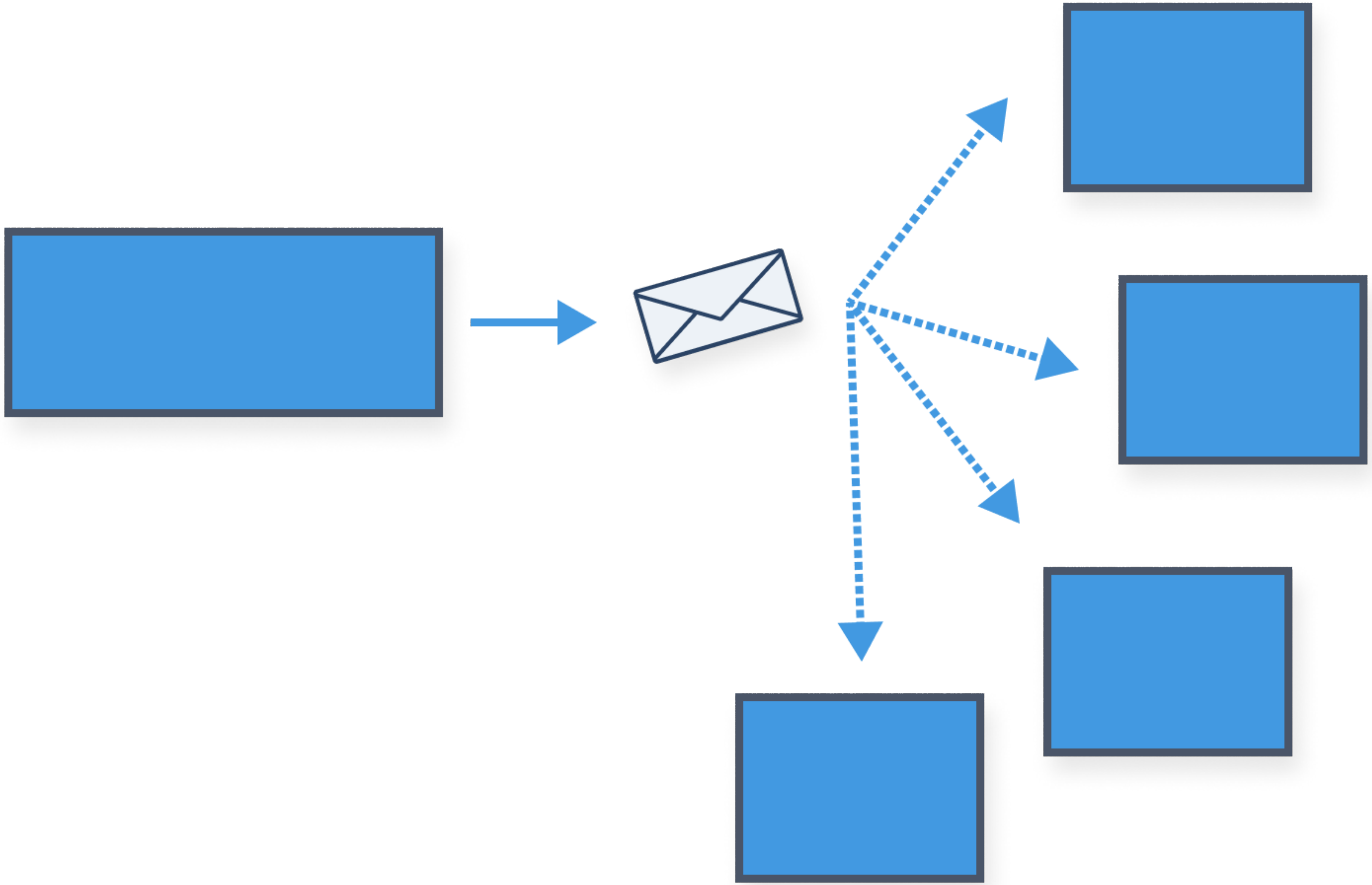
Synchronous and asynchronous

One receiver and multiple receivers

Integrating services

- Minimize the communication between internal services
- Try not to depend on sync communication
- Use async between services (propagate data asynchronously)
- Orchestration vs. choreography





Protocols (1/2)

HTTP

- Textual protocol
- Most popular, not the most performant

HTTP/2

- Binary protocol
- Designed for low latency
- More efficient data transfer on the wire

Protocols (2/2)

WebSockets

- Persistent connection between client/server
- Based on HTTP
- Low-latency, for transferring large volumes of data

gRPC

- Binary format, small payloads
- Uses HTTP/2 as transport protocol
- Uses protocol buffers - define & serialize structured data into binary format

Messaging Protocols

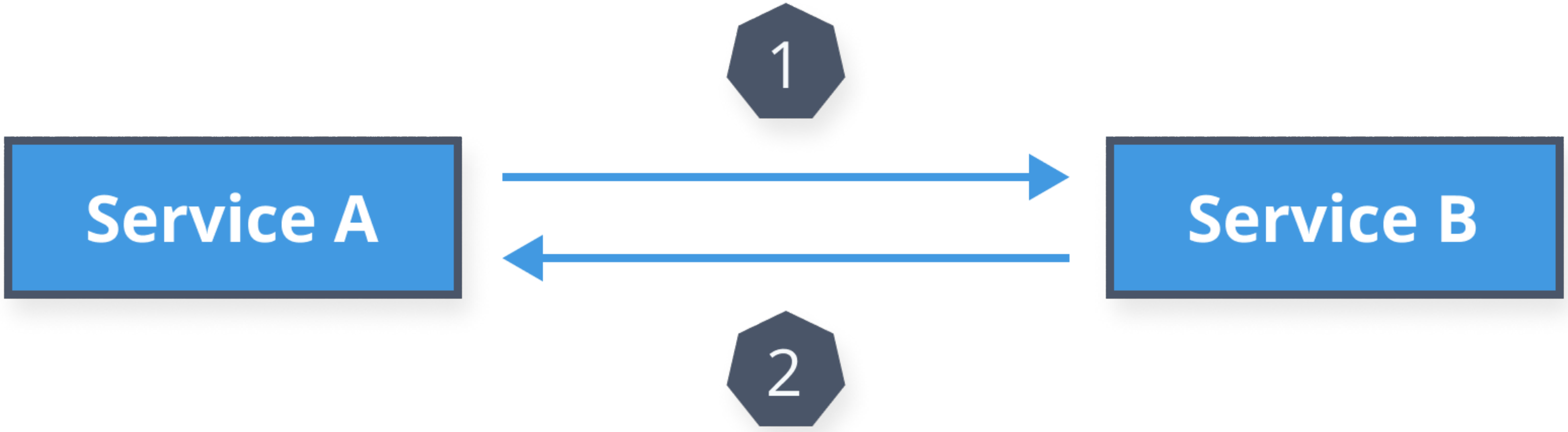
Message Queue Telemetry Transport (MQTT)

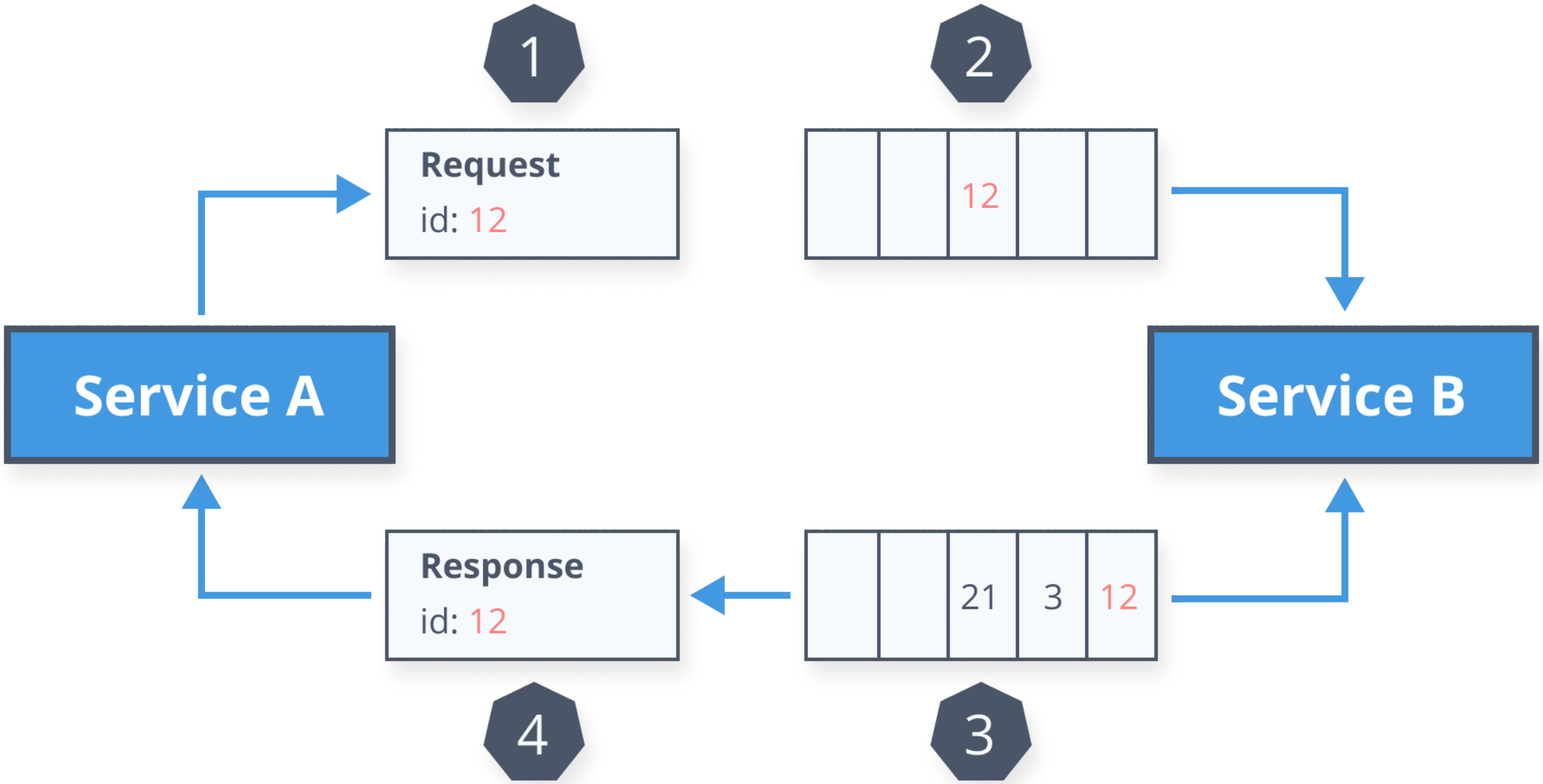
- Simple and lightweight binary protocol
- Designed for low-bandwidth/high-latency environments (e.g. dial-up lines, embedded systems)
- Focuses on Pub/Sub messaging, offers delivery guarantees

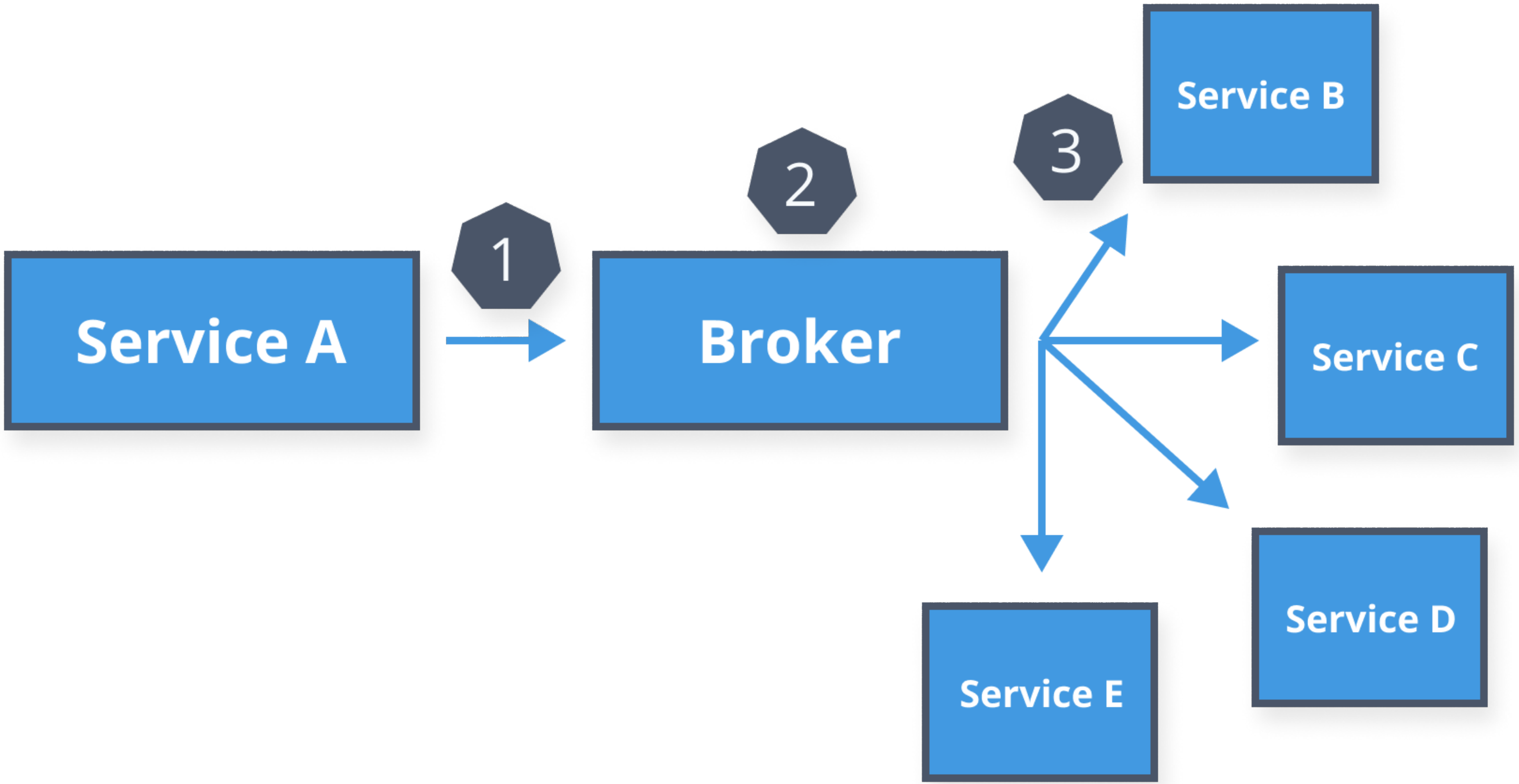
Advanced Message Queuing Protocol (AMQP)

- Binary protocol with rich set of features
  - Reliable queuing, topic-base pub/sub, routing, security, transactions
- Battle-tested and proven to be reliable

both use WebSockets over TCP


1
Service A
Service B
2


1
2
Request
id: 12
12
Service A
Service B
Response
id: 12
21
3
12
4
3



Service A
1
Broker
2
3
Service B
Service C
Service D
Service E

Publisher/Subscriber - Considerations

- Message order is not guaranteed (default)
  - Design for idempotent operations
- If ordering is needed:
  - Use messaging systems ordering functionality
  - Priority queue pattern
- Use poison message queue (for errors/crashes)

Service Communication - Idempotency (1/2)

Run an operation multiple times, without changing the result

- Messages can be received and processed more than once
  - Retry policies, failures etc.

Two approaches:

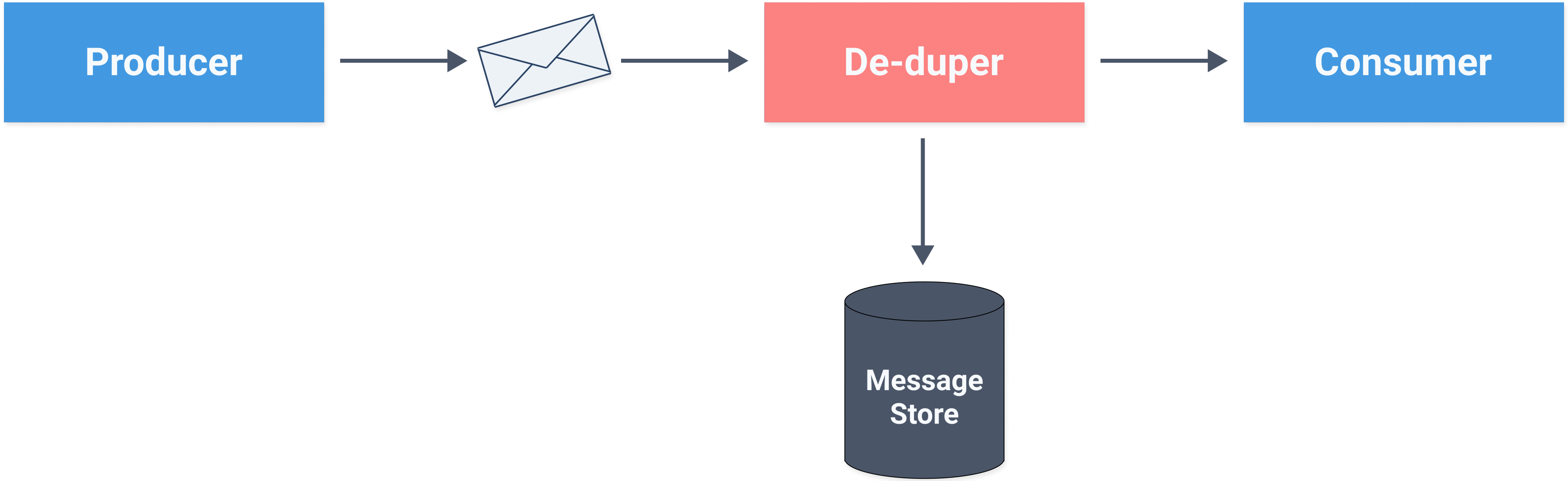
- *Exactly-once* approach is hard
- Use *at-least-once* approach

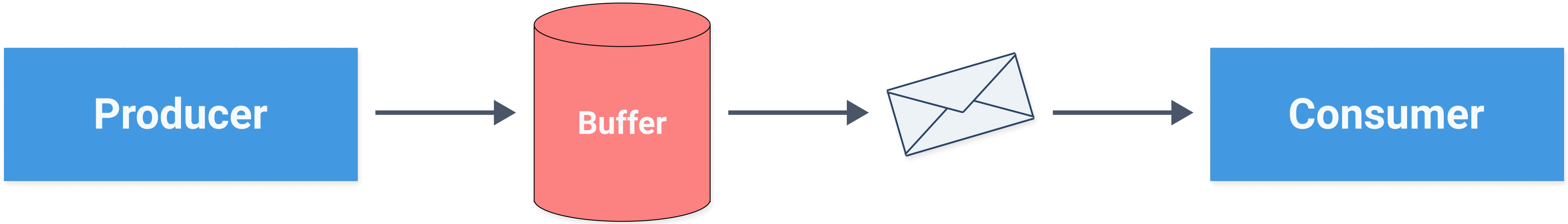
Service Communication - Idempotency (2/2)

- Natural idemopotency = no need to do anything special
- Not naturally idempotent:
  - Add unique identifier to the message
  - Service checks if the message was processed or not

🙀

What if you can't enforce idempotency?


Producer
De-duper
Consumer
Message Store


Producer
Buffer
Consumer

Service Communication - Serialization

JSON

- Readable, self-contained
- Large memory footprint
- Expensive serialization/deserialization with a lot of data

Protobuf

- Binary format - needs a generator
- Schema defined in `.proto` files

Exercises - Communication

https://github.com/peterj/velocity-berlin-2019

Data in Cloud-Native Applications

Data in Cloud-Native Applications

- Cost of storing data has decreased
  - Cheaper to keep vast amounts of data (2¢/GB/month)
- Managed/serverless* data services lowered the operational overhead of DB systems
- Easier to spread data across different storage types
- Decentralization data is encouraged, each service has its own datastore

*usage-based billing (data stored and processed)

Data in Cloud-Native Applications

- Managed services to store, process, and analyze data
- Use polyglot persistence, data partitioning, and caching
- Embrace eventual consistency (use strong consistency when necessary)
- Deal with data distributed across multiple datastores

Focus on building your applications, not provisioning/managing data systems

---

Data Storage Systems

- Objects, files, and disks
- Databases
    - Key/value store
    - Document
    - Relational
    - Graph
    - Column-family
    - Time-series
    - Search
- Streams and queues
- Blockchain

Data Storage Systems - Objects, files, and disks

Object/blob storage

- Use it with files, cloud provider API support needed

File storage

- Network Attached Storage (NAS) support is needed
- Services needd shared access to files

Disk/Block storage

- Services that require presistent local storage disks

Data Storage Systems - Databases (1/3)

Key/Value

- Hash table, stores a value under a unique key
- Values can be retried using the key, or part of the key
- Performance of reads/writes depends on a good key selection
- Inexpensive and very scalable

Document

- Stores a document (value) by a primary key
- Document (value) needs to conform to defined structure
- Documents map nicely to objects in programming languages
- Schemas are not enforced (*schema on read*)
  - Apps consuming the data need to know how to work with the data returned

Data Storage Systems - Databases (2/3)

Relational

- Most popular and commonly used, very mature
- Data organized into tables (rows and columns)
- Relationships between tables can be enforced by the database system
- Strict schema (*schema on write*)
- Good with data that contains a lot of relationships
  - *many-to-many* relationships are hard with document DBs, but simple in relational

Graph

- Data stored in edges and nodes
- Works well for analyzing relationships between entities
- Graph data can be stored in other DBs, however traversals are complex

Data Storage Systems - Databases (3/3)

Column-family

- Data in rows and columns (tabular data with rows and columns)
- Columns divided into groups = column families
  - Set of logically related columns, retrieved and manipulated as a unit

Time-series

- Optimized for time, storing values based on time
- Support for very high number of small writes
- Good for telemetry data, IoT sensors, app/system counters

Search

- Used to search information in other datastores/services
- Indexes large volumes of data

Data Storage Systems - Blockchain

- Records stored in an immutable* way
- Records grouped in a block → added to the chain
- Blocks are chained using hashing, to ensure they are not tampered with

*unable to be changed

Data Storage Systems - Selecting a Datastore

Functional Requirements

- What type of data do you need to store?
- How will the data be consumed and written?
- How large are the items placed in the datastore?
- How much storage capacity do you need? Do you anticipate partitioning the data?
- Do you need support for complex relationships?
- Strong consistency or eventual consistency?
- Do you need a fixed or strongly enforced schema?
- Do you need full-text search, indexing?
- Do you plan to fire events on data changes?

Data Storage Systems - Selecting a Datastore

Non-functional Requirements

- What experience does your team have?
- Do you need support?
- What are your performance/reliability requirements?
- Do you need backup/restore features?
- How about data replication across multiple regions/zones?
- On-premise or multiple cloud providers?

Data Storage Systems - Selecting a Datastore

Management and Cost

- Is there a managed data storage available?
- Any restrictions on licensing types?
- Any preferences on proprietary vs. OSS license?
- What is the overall cost of using the service?

Data Storage Systems - Selecting a Datastore

- **357** different databases*
- Major driving factor: skillset of the team
- Significant overhead for managing data systems
  - Deploying simple DB is easy, but consider patching, upgrades, perf tuning, backups, ...
- Tooling availability

*https://db-engines.com/en/ranking

Data in Multiple Datastores

- Introduces data management challenges
- Traditional transactions not possible
- Distributed transactions adversly affect the performance & scale

Challenges

- Consistency across datastores
- Analysis of data
- Backup and restore

Change Data Capture

- Stream of data change events (change log)
- Exposed through API → trigger functions on events

Change Data Capture - Use cases

- Notifications
- Materialized views
- Cache invalidation
- Auditing
- Search
- Analytics and change analytics
- Archive
- Legacy systems

Transactions

Scenario

User uploads a picture file with description and name. Application needs to write the file to object storage and data to a document database.

Problem

File gets written, but writing to the database fails. We end up with orphaned file.

Solution

Treat both writes as a transaction - one fails, both should fail.

Distributed Transactions

Transaction that spans over multiple databases and services

How to keep the transaction atomic*?

- One step fails, how do we roll back?

How to handle concurrent requests?

- Data is being persisted and read at the same time - do you return old data or new?

*atomicity of transaction=all steps completed, or no steps completed

Distributed Transactions

Two-Phase Commit

Pros

- *prepare* and *commit* phase
- Transaction coordinator
- Guarantees an atomic transaction
- Read-write isolation - changes on records not available until coordinatior commits them

Cons

- Slow, dependent on the transaction coordinator
- Database row locking → **deadlocks**
- Doesn't scale
- Not recommended for microservices

Distributed Transactions

Eventual Consistency and Compensation / Saga

Each service publishes an event when data is updated. Other services subscribe to events. When event is received, service updates its data

Pros

- Each microservice focuses on its own transaction
- No DB lock required
- Highly scalable under heavy load

Cons

- No read isolation
- Hard to debug and maintain if a lot of services are involved

Scaling Data

- No logic in DBs (stored procedures/triggers)
  - Makes it harder to scale
- Replicate:
  - Cache, materialized view, read-replica
- Partition:
  - Horizontally (sharding)
  - Vertically based on the model
  - Functionally based on features

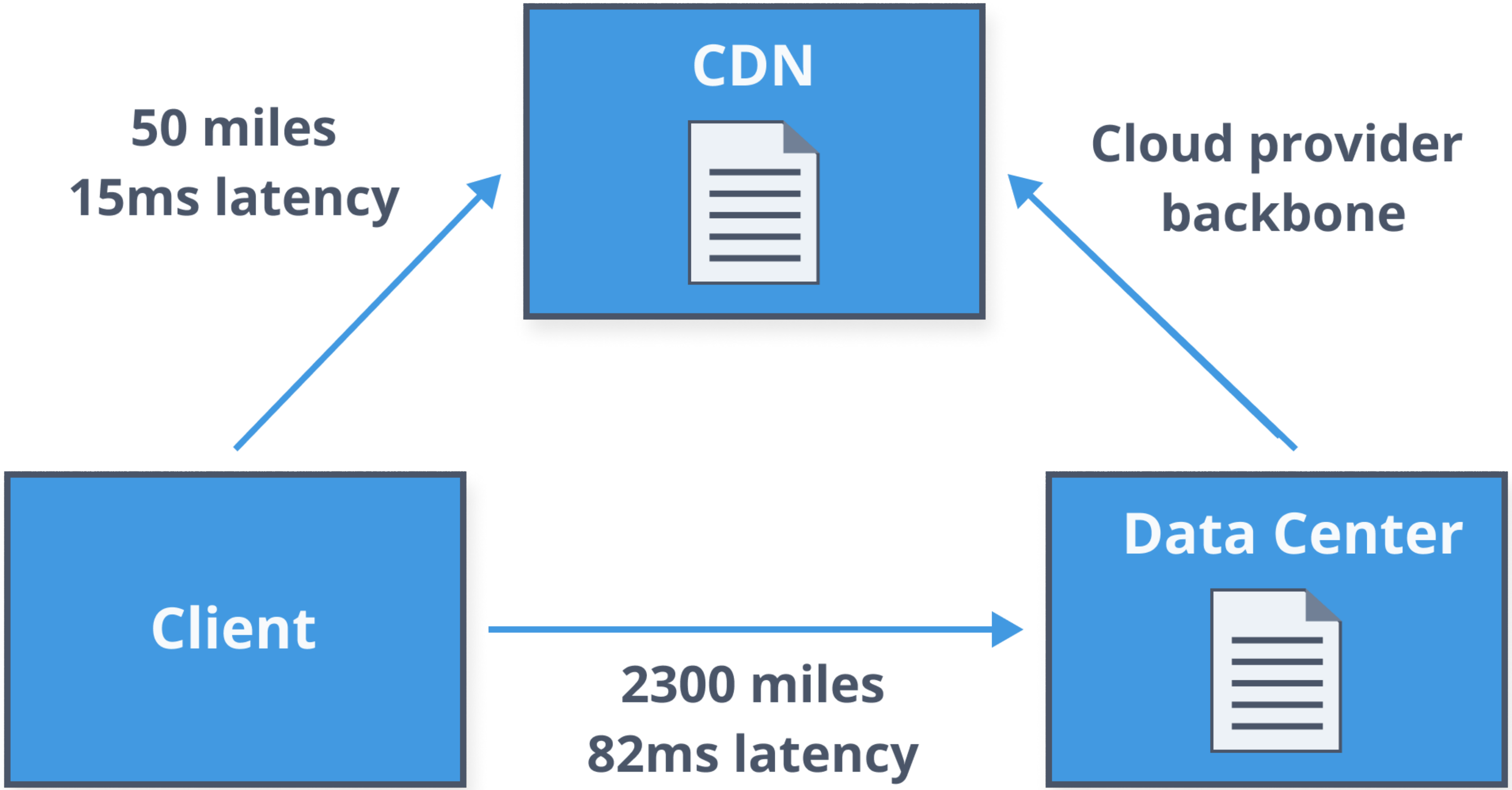
Content Delivery Network (CDN)

A group of geographically distributed data centers

- Used for caching static content closer to consumers
- Reduces the latency between the consumer and the data

Use cases

- Improve website loading times
- Speed up downloads & updates
- Increase content availability/redundancy
- Speed up file uploads (e.g. Amazon CloudFront)


CDN
50 miles
15ms latency
Cloud provider
backbone
Client
Data Center
2300 miles
82ms latency

Content Delivery Network (CDN)

- Configured with an expiration date-time (TTL)
  - Upon expiry, data is reloaded from origin
- Manual expiry = add hash/version to the content (e.g. `/image1.jpg?v=1`)
- Explicitly expire the cache through API or management console

Databases on Kubernetes

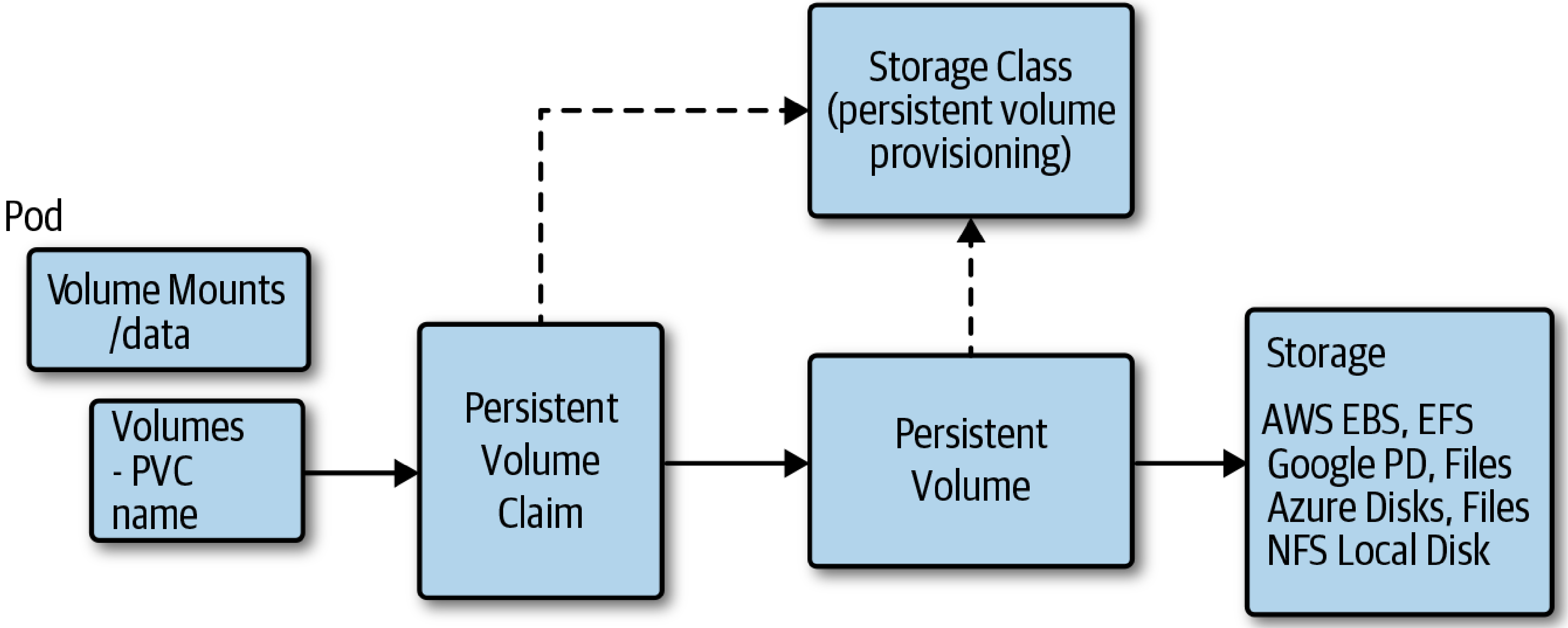
Running a stateful workload is much different than stateless services

- Use **stateful sets** and **persistent volumes**
- Use operators* instead - can simplify deployment and management of data systems

*https://www.operatorhub.io

Databases on Kubernetes

- Durable volumes needed
  - Different lifecycle as containers
- Mount into pods:
  - Persistent volumes (PV)
  - Persistent volume claims (PVC)
  - Storage classes (underlying storage providers)


Storage Class
(persistent volume
provisioning)
Pod
Volume Mounts
/data
Volumes
- PVC
name
Persistent
Volume
Claim
Persistent
Volume
Storage
AWS EBS, EFS
Google PD, Files
Azure Disks, Files
NFS Local Disk

Databases on Kubernetes

Designed to address problems of running stateful services in Kubernetes

- Manages the deployment and scaling of pods
- Guarantees the order and uniqueness of pods
- Each pod has a persistent identifier (e.g. `mongo-0`, `mongo-1` ...)
- PV and PVC for each pod

Exercises - Working with Data

https://github.com/peterj/velocity-berlin-2019

Developing Cloud-Native Applications

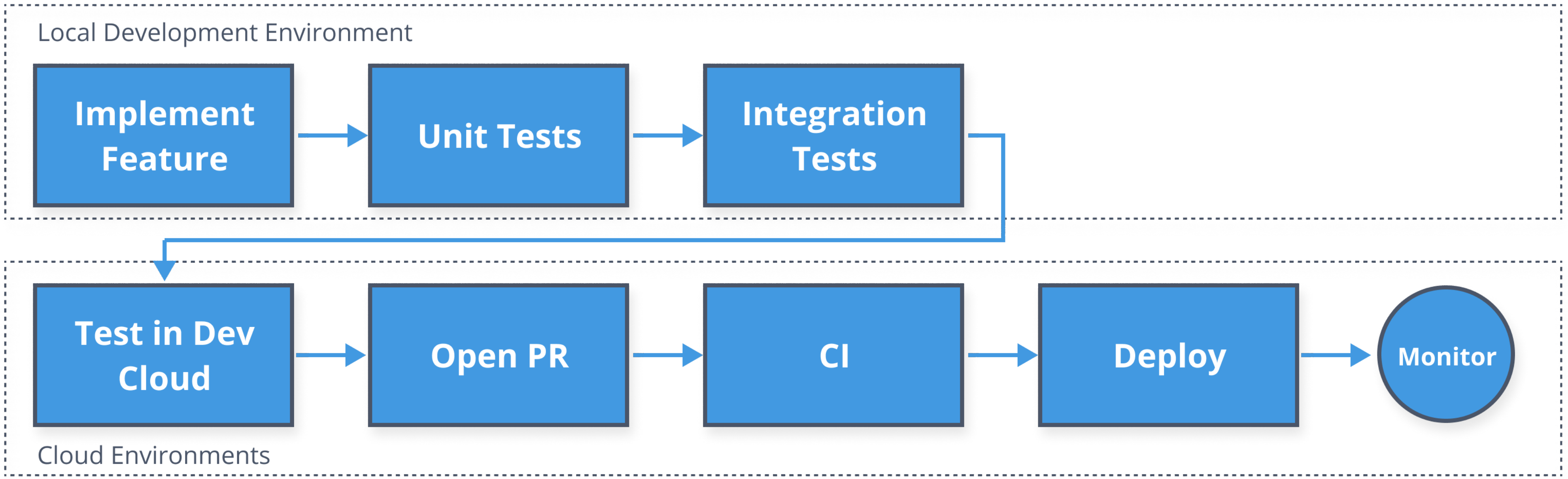
Development Environments and Tools

- Traditionally set up on local development machines/VMs
- Enables you to quickly iterate, test, debug changes
- Microservices & serverless makes this difficult
    - Hard/impossible to run entire application on local machine

Development Environments and Tools

Consider the following

- Does the code need to run in the cluster?
- Are you running your cluster locally or in the cloud?
- Where do you make/commit changes - locally or in the cloud?
- Any dependencies that need to run in the cloud?
- Team distribution - would you benefit from a collaborative dev environment?


Local Development Environment
Implement Feature
Unit Tests
Integration Tests
Test in Dev Cloud
Open PR
CI
Deploy
Monitor
Cloud Environments

Development Tools (1/2)

Docker Compose

- Define and run multiple containers locally
- Easily bring up more complex development environments

Minikube

- Run a single-node Kubernetes cluster in a VM locally on your machine
- Good for experimenting

Docker for Mac/Windows

- Similar to Minikube

Kubernetes can be resource heavy, different ways to access LoadBalancer type services

Development Tools (2/3)

KSync

- Replicates local files to containers running in a remote cluster
- Allows quick iteration without re-building, push and updating the containers

Skaffold

- Deploy code changes to a local/remote cluster
- Automatically builds an image and pushes it to a cluster on code changes

Development Tools (3/3)

Telepresence

- Wires local containers into a remote cluster
- Makes your local machine "*part of the cluster*"

Azure DevSpaces

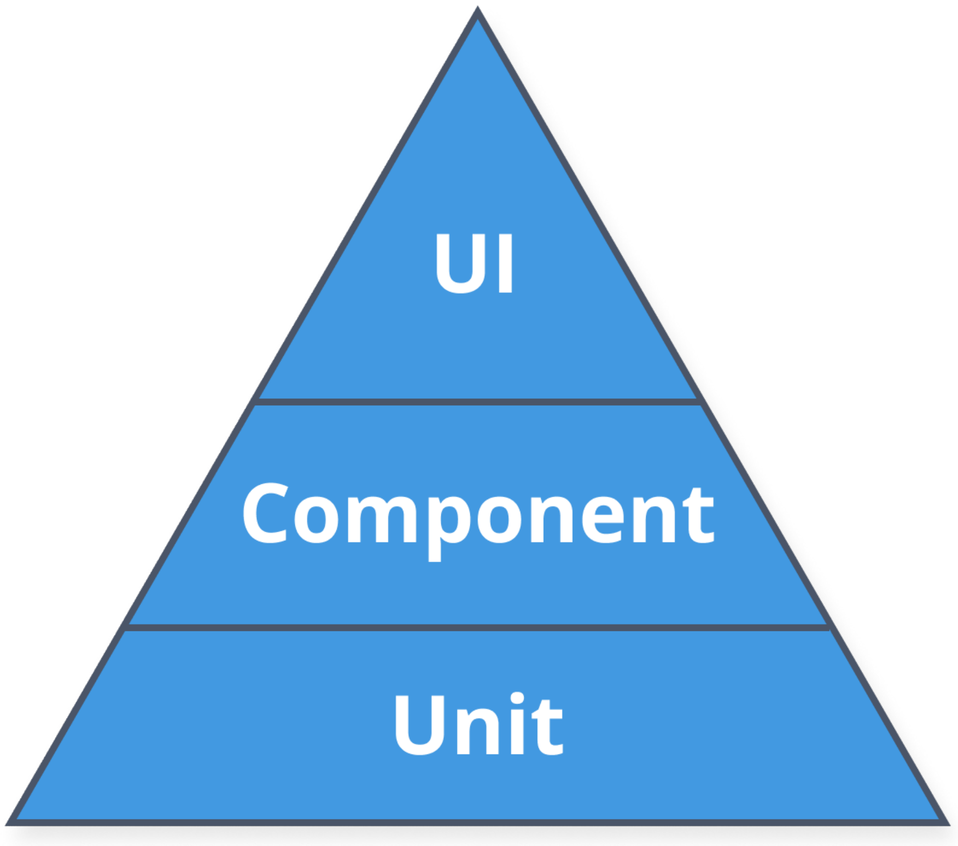
- Develop and run containers services in isolation on AKS
- Allows teams to share a cluster

Testing Cloud-Native Applications

Testing

- Can't survive with manual tests
- Reliable, repeatable and automated tests are essential
- Use test doubles (mocks, fakes and stubs) for dependencies
    - **MOCK** = object use to test the calls it receives
    - **FAKE** = object with working implementation (different from a real implementation)
    - **STUB** = object that returns prepopulated data

Test Automation Pyramid



Testing

Unit tests

- Fast to execute
- First line of defense

Service Tests / Component-level tests

- Test interactions between services/components

User interface tests

- End-to-end tests
- (Relatively) slow to execute and costly to write/maintain
- Cruical for usability/accessibility

Testing

- Acceptance tests
- Smoke tests
- Integration tests
- Security
  - Penetration tests
  - Fuzz tests
- Performance tests
  - Load tests
- Usability tests
- Chaos tests
- Canary tests
- A/B tests
- ...

Testing

When and how often to execute tests?

- Unit: during development, before every merge/check-in
- Service/Component: before/after every merge/check-in
- Integration: before deployments
- Canary: run continuously*
- Security: automated, part of integration/canary tests if possible

---

- UI: on UI changes, consider automating if UI heavy
- Performance: get baselines manually, consider automating for repeatable numbers
- A/B: as needed, make sure you have clear goals and metrics defined
- Chaos: as needed, for operational readiness, to catch potential prod issues

* can be costly to maintain

Testing Environments

Dev → Test → Staging → Production

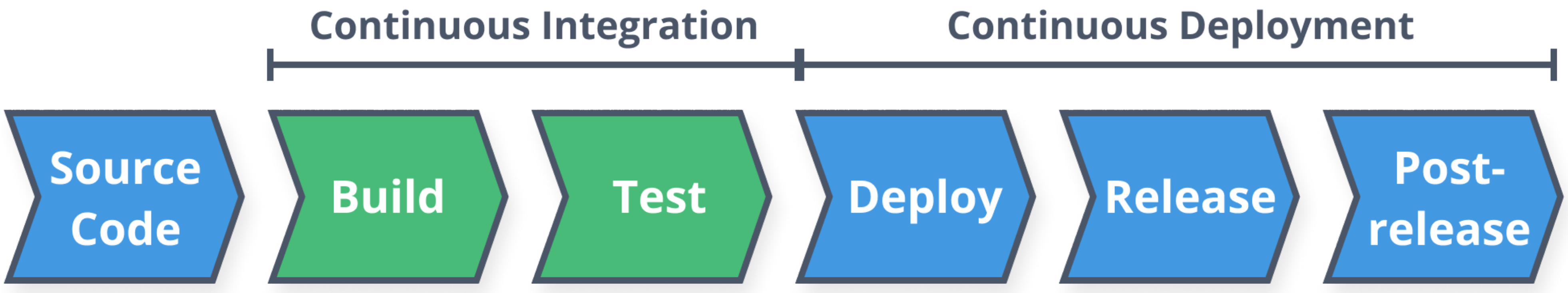
- Mimic production environment (as closely as possible)
- How to keep environments in sync?
- What's the cost for all this?
- Are you running staging in all regions, just like production

How about production only?*

* always consult your co-workers, management, teams before beginning this exercise program. This information is not intended to be taken lightly and without any considerations. Consult with your team to design an appropriate testing environment. If you experience any issues or difficulties...

Operating Cloud-Native Applications

CI/CD


Continuous Integration
Continuous Deployment
Source Code
Build
Test
Deploy
Release
Post-release

CI - Source Code Control

- Repository where your code lives
- Source of truth for your code/configuration
- Branching strategies
- Mono or multi-repo

Branching Strategies

Trunk-based Development

- Everyone works on a single branch - trunk (*master*)
- No need for long-lived feature branches with smaller teams
- Release from the trunk - use 'fix forward' strategy
  - Short-lived release branches that get deleted
- As soon as commit/branch is merged to master, you release

Branching Strategies

Git Flow Strategy

- Designed around releases
- Start with `master` and `develop` branch
- Use feature, release, hotfix branches (off `develop`)
  - Helps with feature/release/hotfix tracking
- Once release is complete:
  - merge to `develop`
  - merge to `master` + tag with a version

CI - Source Code Control

Mono-repo vs. Multi-repo

You will be solving similar problems, regardless where your code is

- Depends on the number of services
- Decide how to do dependency management, isolate service
- How about build breaks?
- More complicated to do independent deploys with mono-repo
- Single tag/version for all services?
- Hard to cleanly define ownership

CI - Build and Test Stage

- Build the code
- Run the tests:
  - Unit/Component
  - Linters
  - Static analysis
  - ...
- Version/tag the generated artifact
  - Use Git commit checksum hash + build number (`ed3ee93-0.1.0`)

Result: built and versioned artifact*

*Docker image or serverless function package

CD - Deploy Stage

- No source code beyond this point
  - Images, packaged artifacts, config/deploy templates
- Automatically triggered by successful CI
- Prepare everything needed and place the artifact into staging
- Run tests (canary) + monitor the services

CD - Release Stage

You need enough data from previous stages to feel comfortable releasing the service into production

Production traffic → new service

- Swap stage & production deployment slots
- Redirect a % of the production traffic to deployed services

Monitor and observe released versions

- Integrate with alarm system

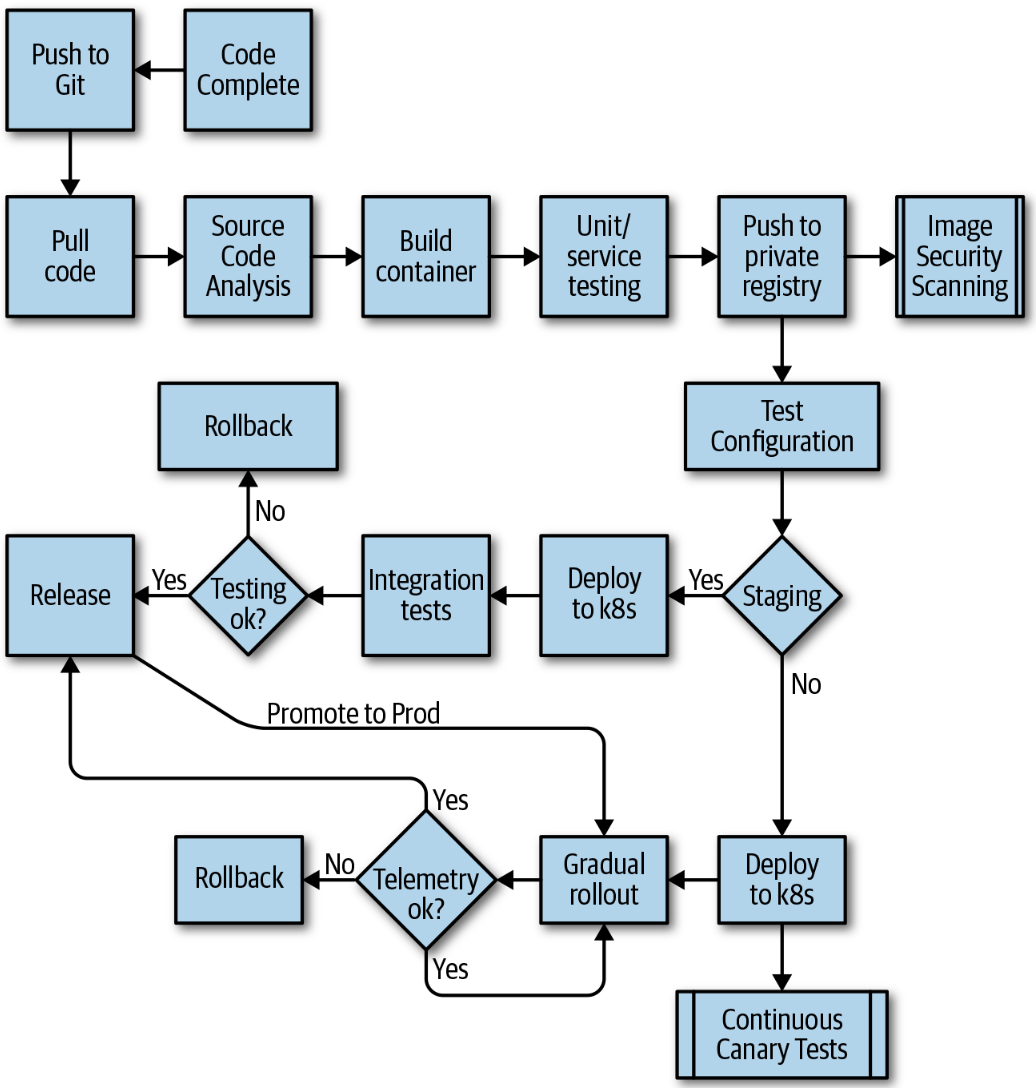
Rollback to previous version OR keep increasing to 100% (fix-forward)

- Usually done manually
- Could be automated

CD - Post-Release Stage

Part of testing in production/operating the application

- Continous service monitoring
- Investigating incidents/errors
  - Alerting/monitoring systems
- Doing chaos testing



Push to Git
Code Complete
Pull code
Source Code Analysis
Build container
Unit/ service testing
Push to private registry
Image Security Scanning
Test Configuration
Rollback
No
Release
Yes
Testing ok?
Integration tests
Deploy to k8s
Yes
Staging
No
Promote to Prod
Yes
Rollback
No
Telemetry ok?
Gradual rollout
Deploy to k8s
Yes
Continuous Canary Tests

Monitoring and Observability

Monitoring is used to assess and report on the overall health of a system or services using metrics

- **Error rate**
  - Rate of failing requests (e.g. `HTTP 500`)
- **Incoming request rate**
  - How much traffic is coming into the system (HTTP requests/second)
- **Latency**
  - Time it took to process a request
- **Utilization**
  - Usage of different pieces of the system (e.g. CPU, memory, disk usage)

Monitoring and Observability

Using monitoring you should be able to tell which part of your system is broken and why is it broken

- Come up with the set of metrics before your first release
- Define:
  - when to continue the release (`5 minutes, 1 hour, 2 hours, ...`)
  - when to rollback (`more than 1% change in negative direction`)

Tools:

- Prometheus (collecting metrics from services)
- Grafana (visualizing metrics)

Monitoring and Observability

Observability captures everything that monitoring doesn't using traces

- Granular details and insights into services
- Helps you debug services more efficiently

Monitoring tells you something is wrong, observability helps you dig deeper and investigate

Tools:

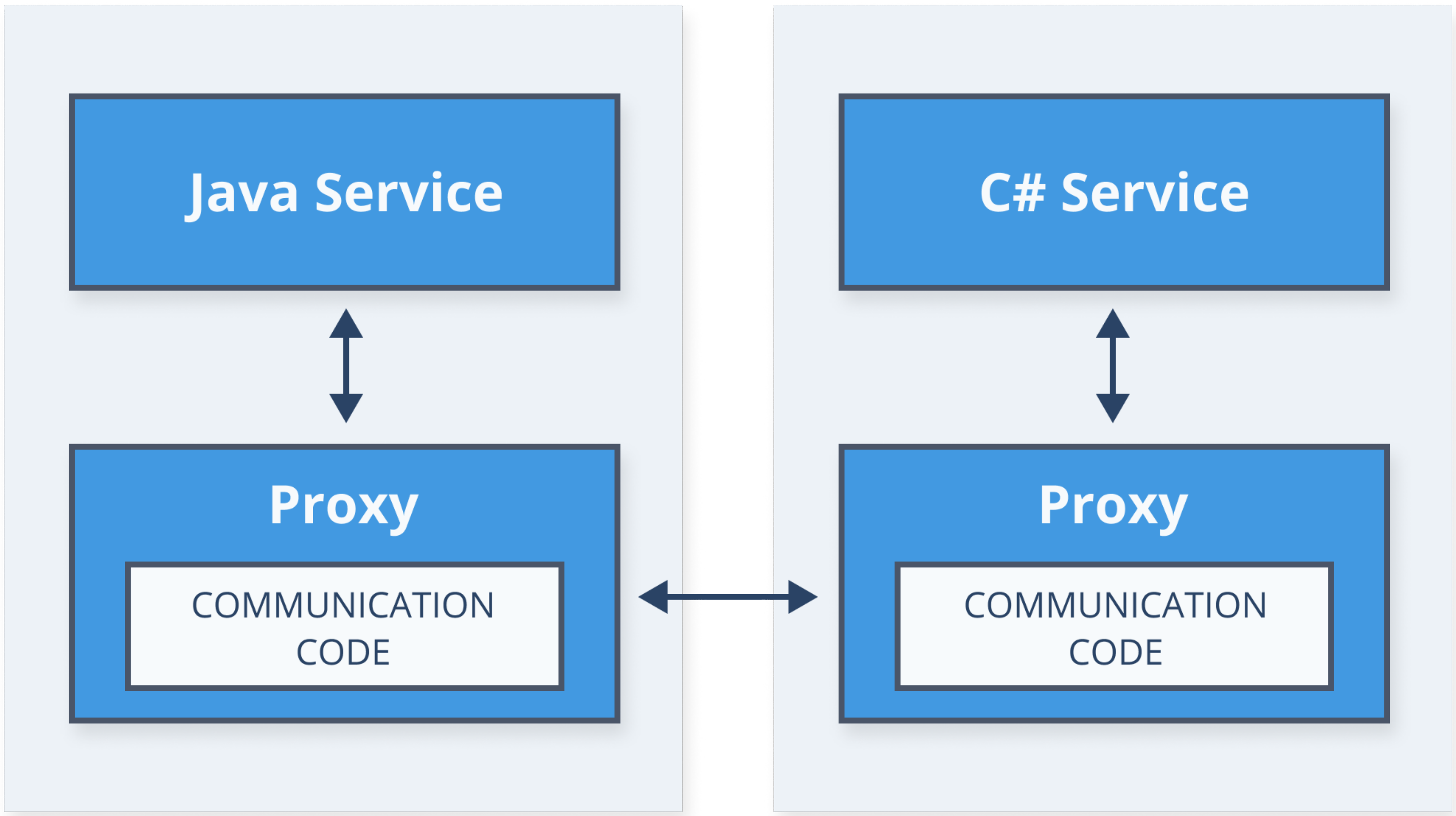
- Jaeger
- Zipkin

Exercises - Development, Testing, and Operations

https://github.com/peterj/velocity-berlin-2019

Service Mesh

Dedicated infrastructure layer for managing service-to-service communication to make it manageable, visible, and controlled


Java Service
Proxy
COMMUNICATION CODE
C# Service
Proxy
COMMUNICATION CODE

Service Mesh - Architecture

Data plane (proxies)

- Run next to each service instance (or one per host)
- Intercept all incoming/outgoing requests (`iptables`)
- Configure on how to handle traffic
- Emits metric

Control plane

- Validates rules
- Translates high-level rules to proxy configuration
- Updates the proxies/configuration
- Collects metrics from proxies

Service Mesh - Features

Traffic Management 🚦

- Percentage based traffic routing: `X% to v1, Y% to v2`
- Request based routing: headers, URIs, scheme, method, ...

Resiliency Features 🌊

- Retries
- Timeouts
- Circuit breakers

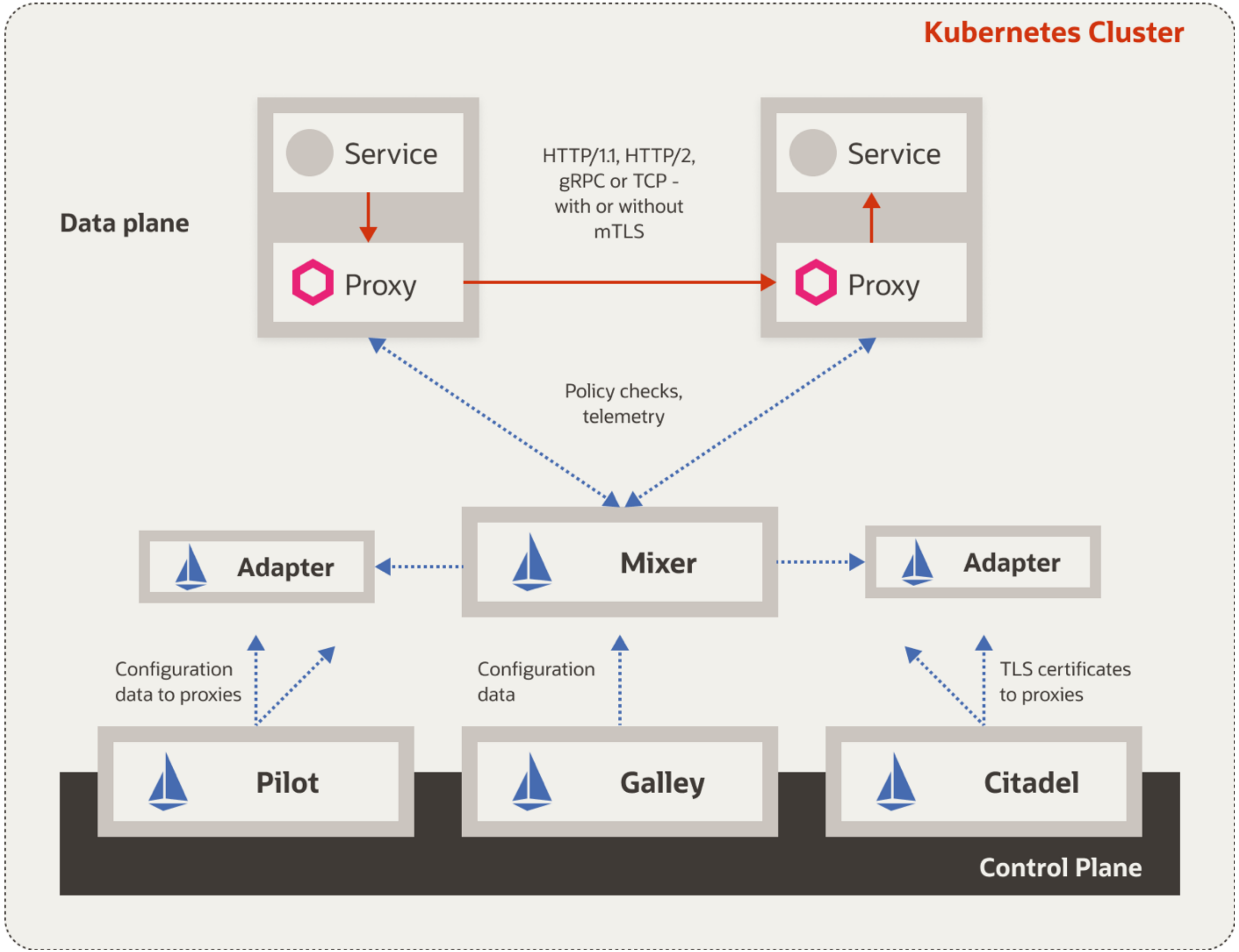
Service Mesh - Features

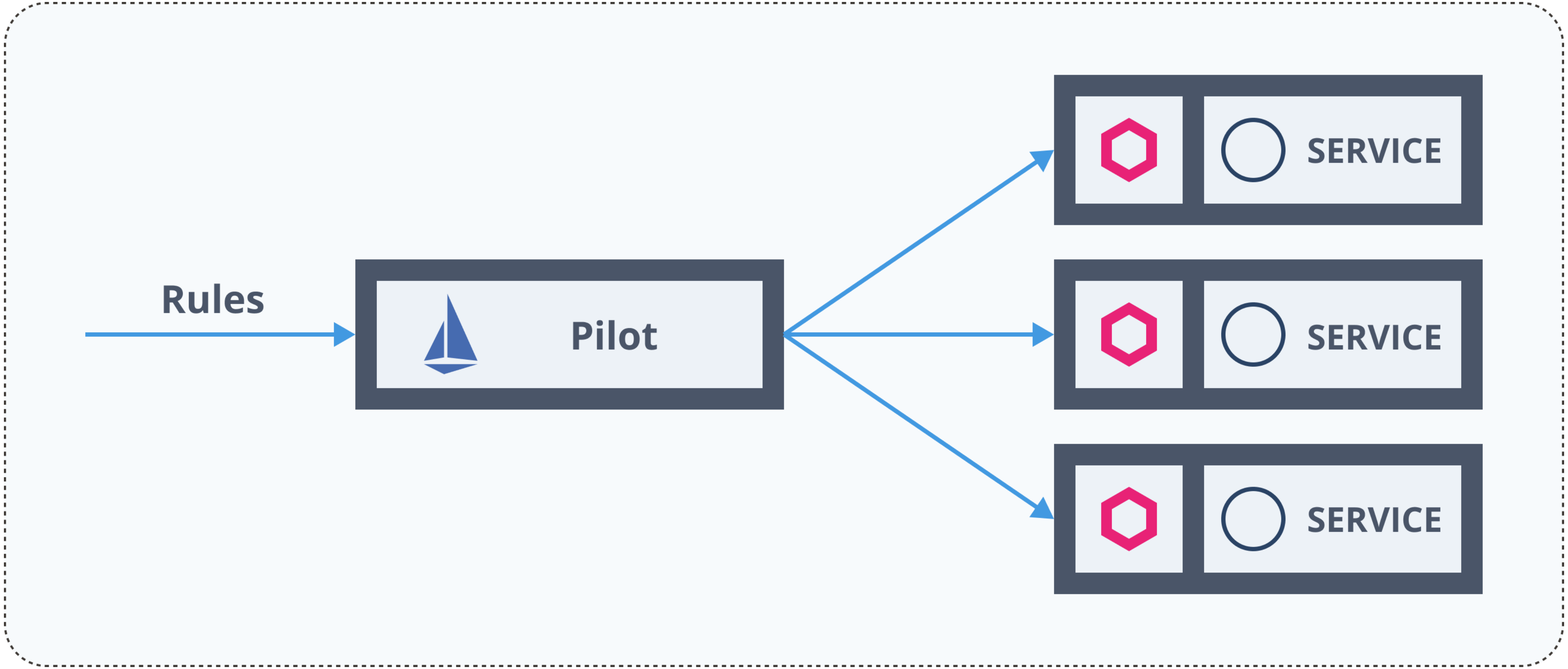
Security

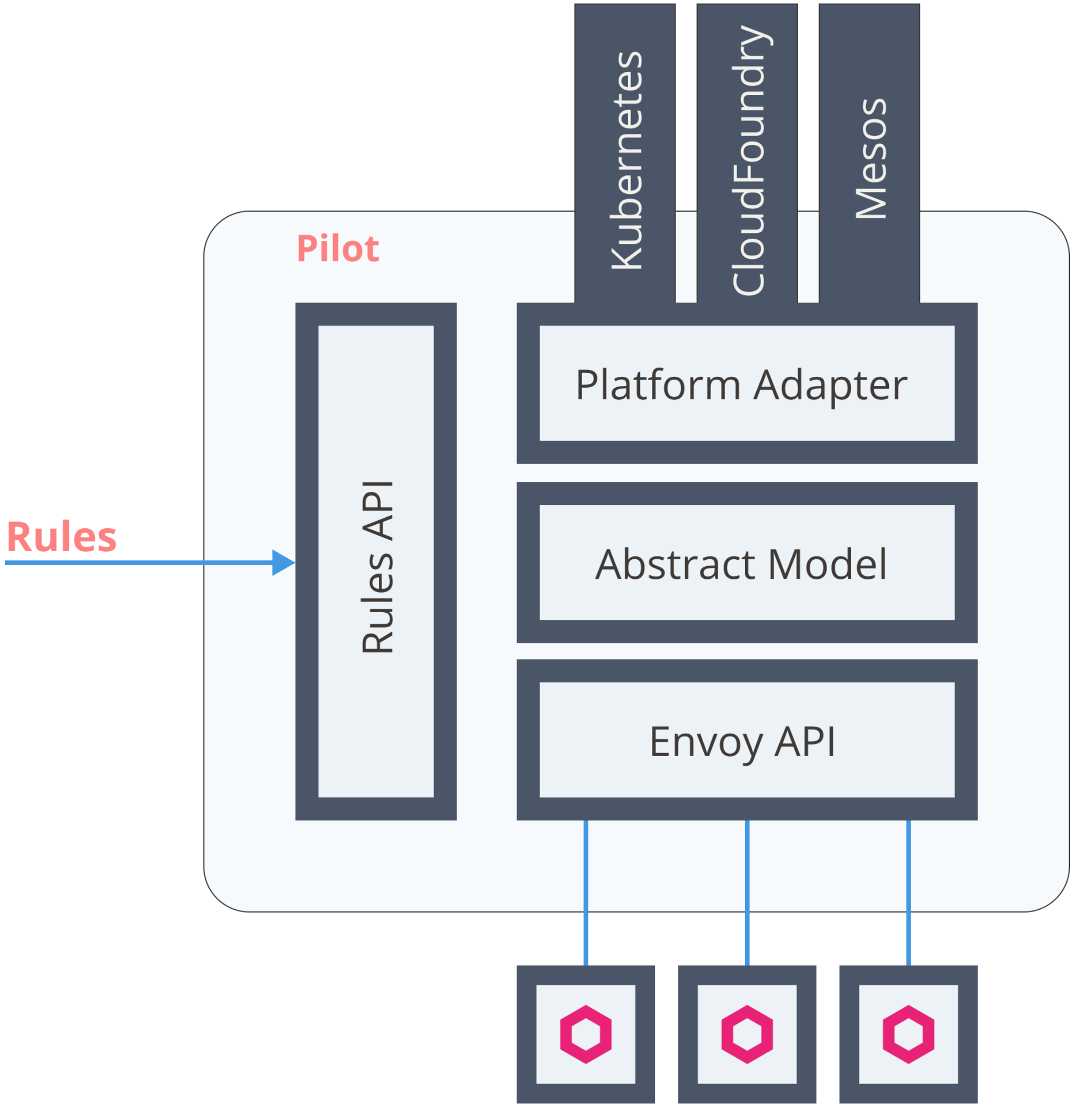
- Secure communication between services (mutual TLS)
- Identity + cert for each service
- Cert lifecycle managed by the proxy
- Access control (namespace, service, method level)

Tracing and Monitoring

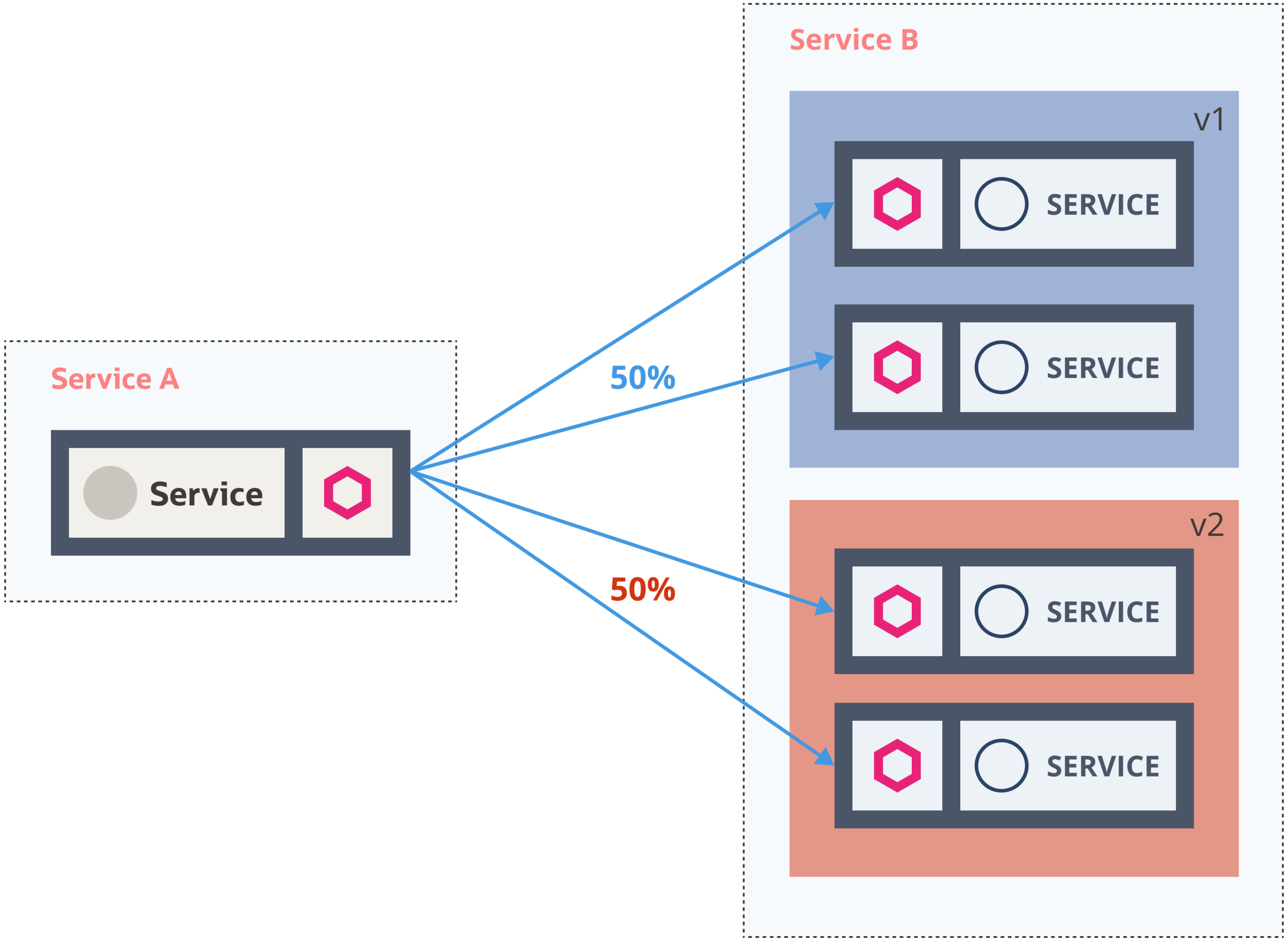
- Proxies collect metrics automatically (requests, durations, sizes, response codes,...)
- Visibility into service communication without code changes
- Tools: Grafana, Jaeger, Kiali


Kubernetes Cluster
Data plane
Service
Proxy
HTTP/1.1, HTTP/2, gRPC or TCP - with or without mTLS
Service
Proxy
Policy checks, telemetry
Adapter
Mixer
Adapter
Configuration data to proxies
Configuration data
TLS certificates to proxies
Pilot
Galley
Citadel
Control Plane


Rules
Pilot
SERVICE
SERVICE
SERVICE



Kubernetes
CloudFoundry
Mesos
Pilot
Platform Adapter
Rules
Rules API
Abstract Model
Envoy API


Service A
Service
50%
50%
Service B
v1
SERVICE
SERVICE
v2
SERVICE
SERVICE

Service Mesh - Istio

Traffic Management Resources

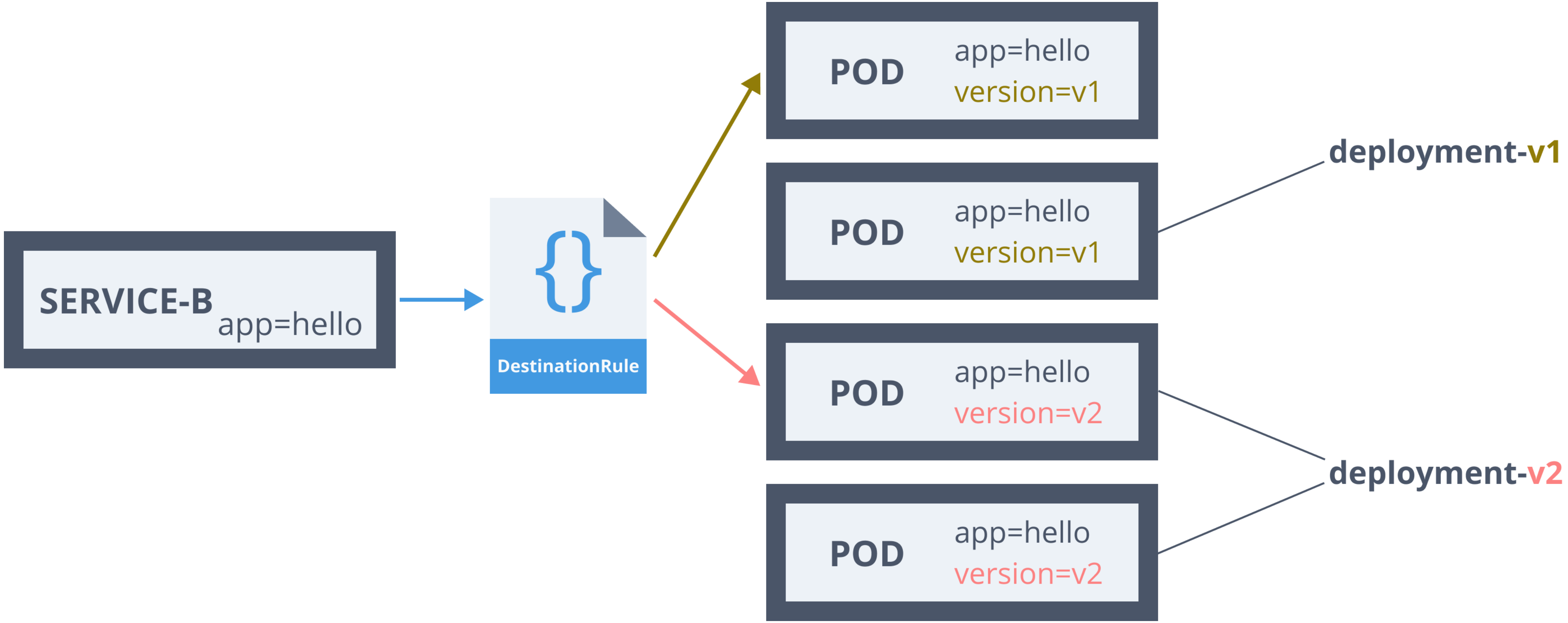
- Gateway
- VirtualService
- DestinationRule
- ServiceEntry
- Sidecar

Service Mesh - Virtual Service

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: service-b
spec:
  hosts:
    - service-b.default.svc.cluster.local
  http:
    - route:
        - destination:
             host: service-b
             subset: v1
           weight: 98
        - destination:
            host: service-b
            subset: v2
          weight: 2
```

Service Mesh - Destination Rule

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: service-b
spec:
  host: service-b.default.svc.cluster.local
  subsets:
    - name: v1
      labels:
        version: v1
    - name: v2
      labels:
        version: v2
  trafficPolicy:
    tls:
      mode: ISTIO_MUTUAL
```


SERVICE-B
app=hello
DestinationRule
POD
app=hello
version=v1
POD
app=hello
version=v1
POD
app=hello
version=v2
POD
app=hello
version=v2
deployment-v1
deployment-v2

Service Mesh - Service Entry

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: movie-db
spec:
  hosts:
      - api.themoviedb.org
  ports:
      - number: 443
        name: https
        protocol: HTTPS
   resolution: DNS
   location: MESH_EXTERNAL
```

Service Mesh - Gateway

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: gateway
spec:
  selector:
    istio: ingressgateway
  servers:
    - port:
        number: 80
        name: http
        protocol: HTTP
      hosts:
      - "hello.example.com"
```

Service Mesh - Sidecar

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: Sidecar
metadata:
  name: default
  namespace: prod-us-west-1
spec:
  egress:
    - hosts:
        - 'prod-us-west-1/*'
        - 'prod-apis/*'
        - 'istio-system/*'
```

Service Mesh - Traffic Management

- Define subsets in DestinationRule
- Define route rules in VirtualService
- Define one or more destinations with weights

Resiliency

Ability to recover from failures and continue to function

Return the service to a fully functioning state after failure

Resiliency

High availability

- Healthy
- No significant downtime
- Responsive
- Meeting SLAs

Disaster recovery

- Design can't handle the impact of failures
- Data backup & archiving

Resiliency Strategies

- Load Balancing
- Timeouts and retries
- Circuit breakers and bulkhead pattern
- Data replication
- Graceful degradation
- Rate limiting

Service Mesh - Timeouts

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
    name: service-b
spec:
    hosts:
    - service-b.default.svc.cluster.local
    http:
    - route:
        - destination:
            host: service-b
            subset: v1
       timeout: 5s
```

Service Mesh - Retries

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: service-b
spec:
  hosts:
    - service-b.default.svc.cluster.local
  http:
    - route:
        - destination:
            host: service-b
            subset: v1
      retries:
        attempts: 3
        perTryTimeout: 3s
```

Service Mesh - Circuit Breakers

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: service-b
spec:
  host: service-b.default.svc.cluster.local
   trafficPolicy:
     tcp:
       maxConnections: 1
     http:
       http1MaxPendingRequests: 1
       maxRequestsPerConnection: 1
     outlierDetection:
       consecutiveErrors: 1
       interval: 1s
       baseEjectionTime: 3m
       maxEjectionPercent: 100
```

Service Mesh - Delays

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: service-b
spec:
  hosts:
    - service-b.default.svc.cluster.local
  http:
    - route:
        - destination:
            host: service-b
            subset: v1
       fault:
         delay:
             percentage: 50
         fixedDelay: 2s
```

Service Mesh - Aborts

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: service-b
spec:
  hosts:
    - service-b.default.svc.cluster.local
  http:
    - route:
        - destination:
            host: service-b
            subset: v1
       fault:
         abort:
           percentage: 50
         httpStatus: 404
```

Exercises - Service Mesh

https://github.com/peterj/velocity-berlin-2019

Thank you

Please rate the session!

Slides: http://bit.ly/buildcnapps

Exercises: https://github.com/peterj/velocity-berlin-2019

Contact:

- @pjausovec
- peterj.dev

Table of Contents

Day 1

Day 2

Slides

http://bit.ly/buildcnapps