qaware.de

QA|WARE
SOFTWARE ENGINEERING

Cloud Computing 2024 / 2025 TH Rosenheim Summary

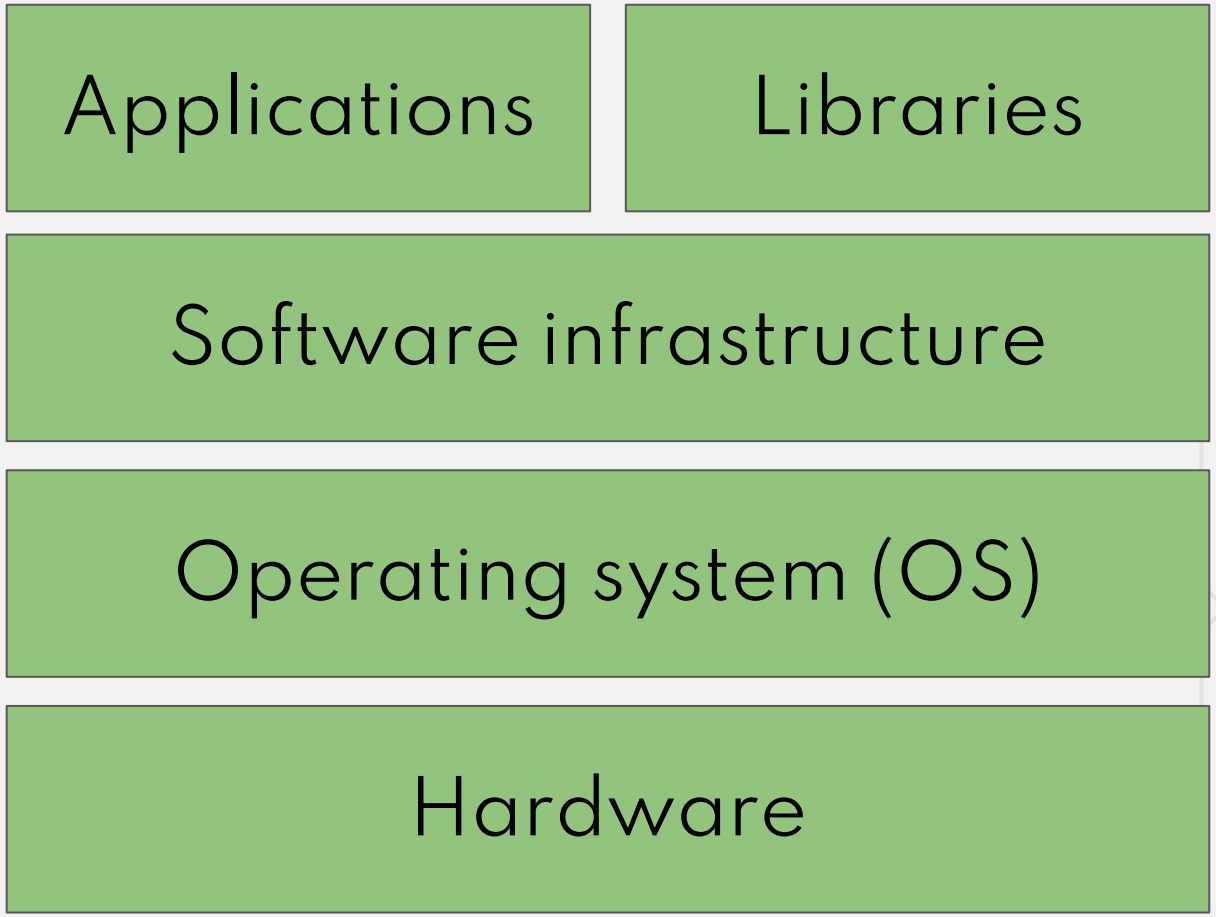
Franz Wimmer | franz.wimmer@qaware.de
Lukas Buchner | lukas.buchner@qaware.de

Evaluation

Introduction

At its core, cloud computing is about a shallower level of integration in system development and operation.

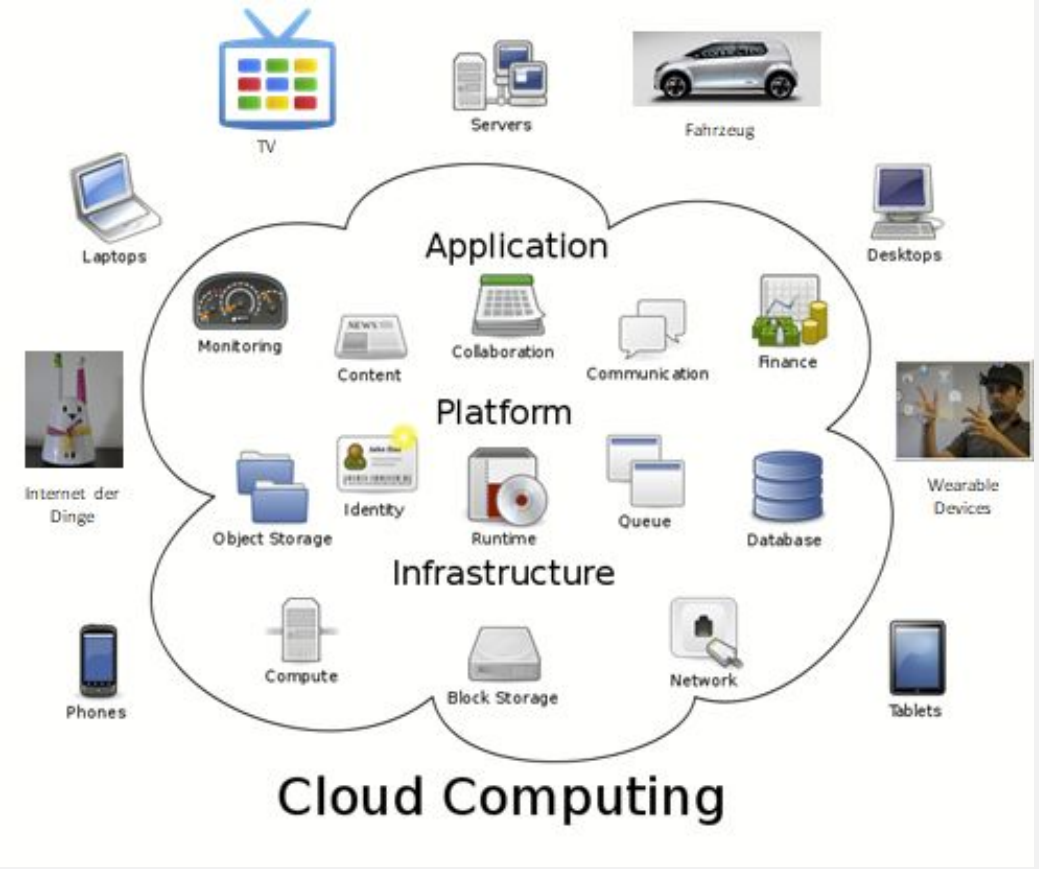
QA|WARE



IT resources from the cloud that can be consumed on demand.

“computation may someday be organized as a public utility”, John McCarthy, 1961

The cloud is dynamic, elastic and omnipresent.



<u>The most important properties of Cloud Computing:</u>

- **X as a Service**: On-demand character; provision of computing capacity, platform services and applications on request and in real time.
- **Resource pools**: Availability of seemingly unlimited resources that process requests in a distributed manner.
- **Elasticity**: Dynamic allocation of additional resources as needed (self-adaptation). No more capacity planning necessary from the user's point of view.
- **Pay as you go model**: Economy of Scale. The costs scale with the benefits.
- **Omnipresence:** Access to the cloud via the internet and from a wide range of end devices (via standard protocols).

The 5 Commandments of the Cloud

- Everything fails all the time
- Focus on MTTR, not on MTTF
- Respect the eight fallacies of distributed computing
- Scale out, not up
- Treat resources as cattles, not as pets

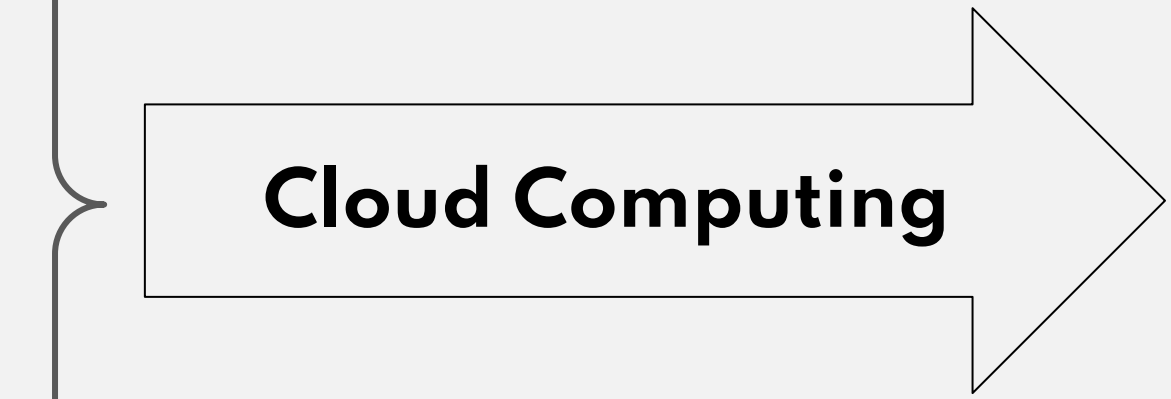
Quelle: https://de.wikipedia.org/wiki/Zehn_Gebote

Bildquelle: https://twitter.com/deniseyu21/status/1035199199443386369

Cloud computing is not a surprise, but has been created on the shoulders of giants.

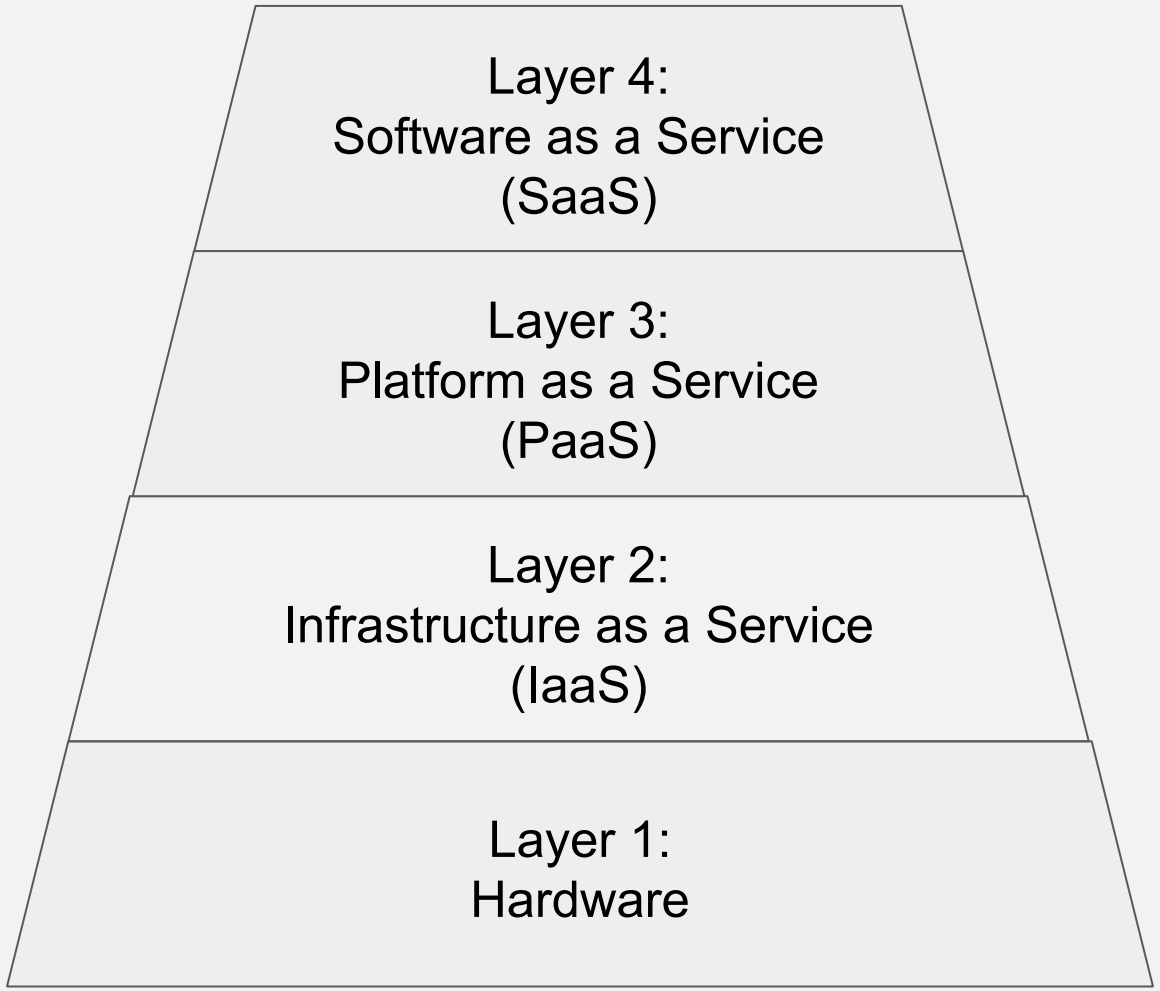
QA|WARE

- **Virtualization** (Virtual machines, hardware virtualization)
- **Commoditization of hardware**
- **Commoditization of the internet** (broadband access, various end devices)
- **Distributed processing** (remote protocols, distributed memory and storage, grid/cluster computing, peer-to-peer)
- **High performance computing** (Parallel algorithms, parallel computers, GPU computing, NoSQL databases)



The layered model of cloud computing: From metal to application.

QA|WARE



- Customizable software services
- CaaS: Component as a service (e.g. Google Charts)
- BaaS: Backend as a Service
- Transparent Updates

Target Group: Users

- Platform services
- Application programming interfaces (APIs)
- Abstraction of technical infrastructure

Target group: Developers

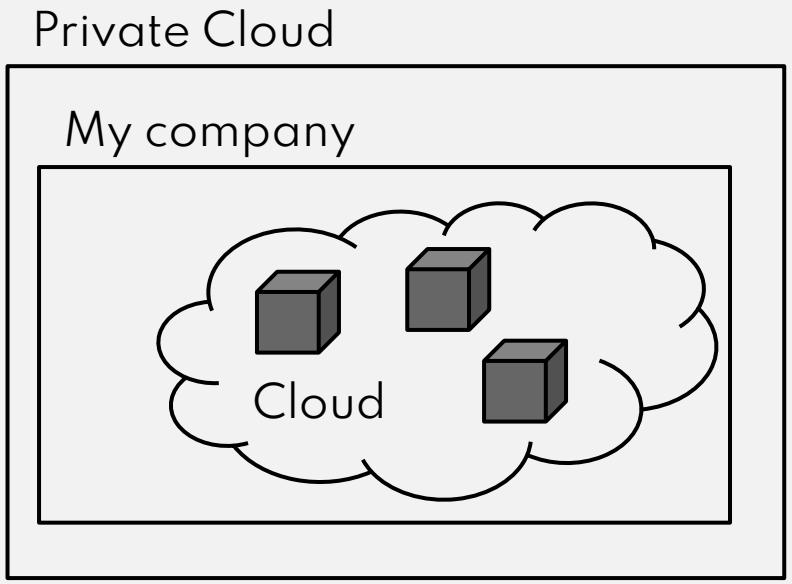
- Elasticity
- Virtual resourcepools
- Technical infrastructure: Machines, Servers (DNS, DHCP, LB, NAS, …)

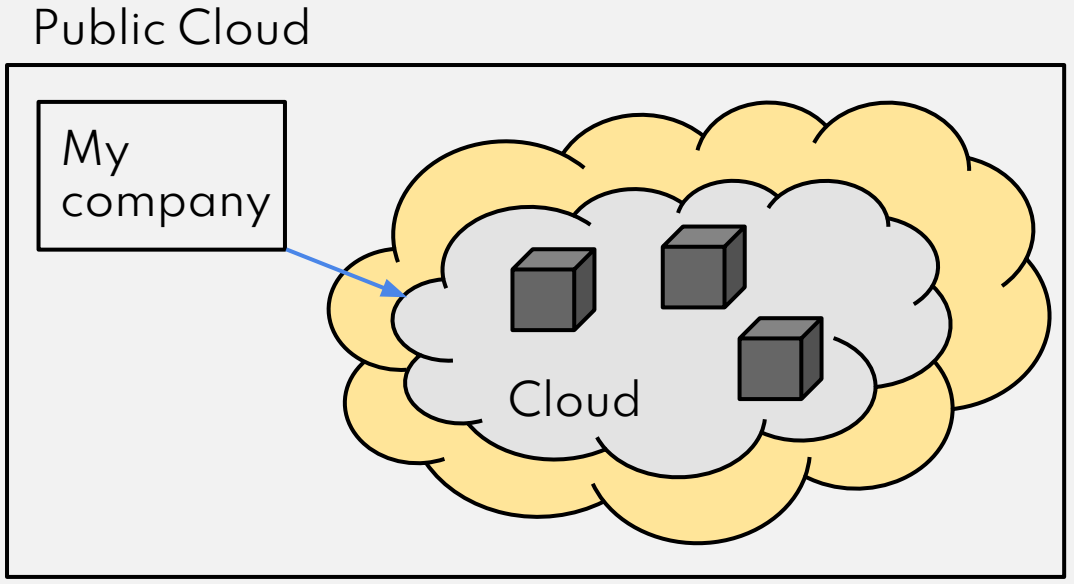
Target group: Operations

- Computers
- Network
- Storage

https://www.youtube.com/watch?v=M988_fsOSWo

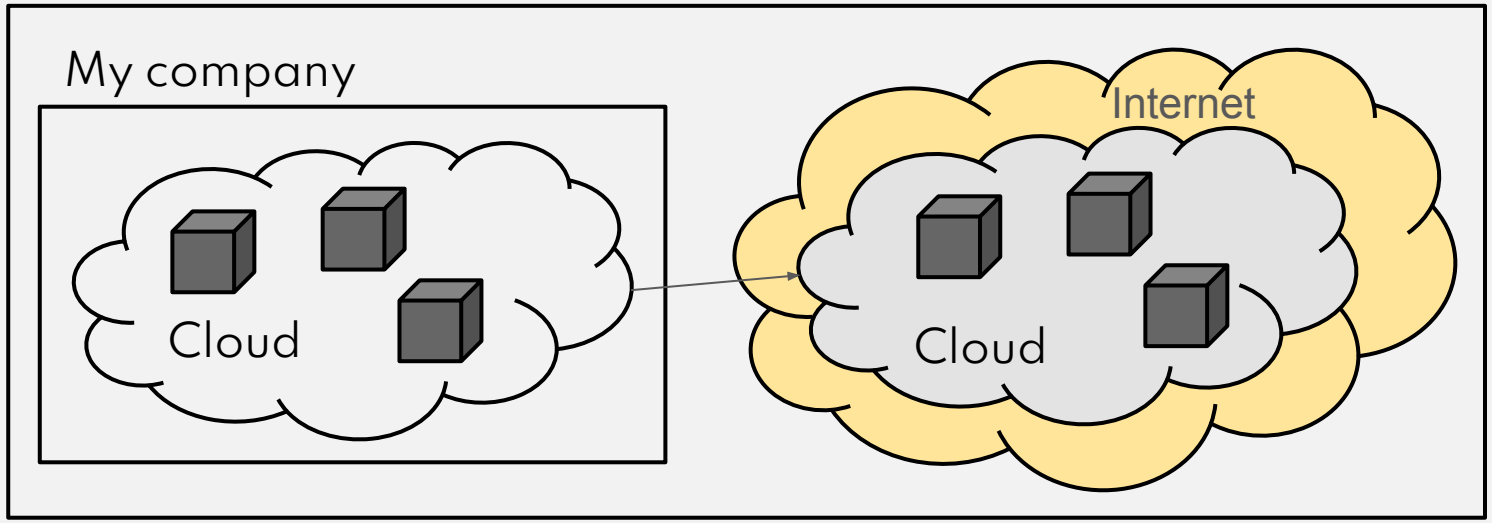
Public and private clouds.



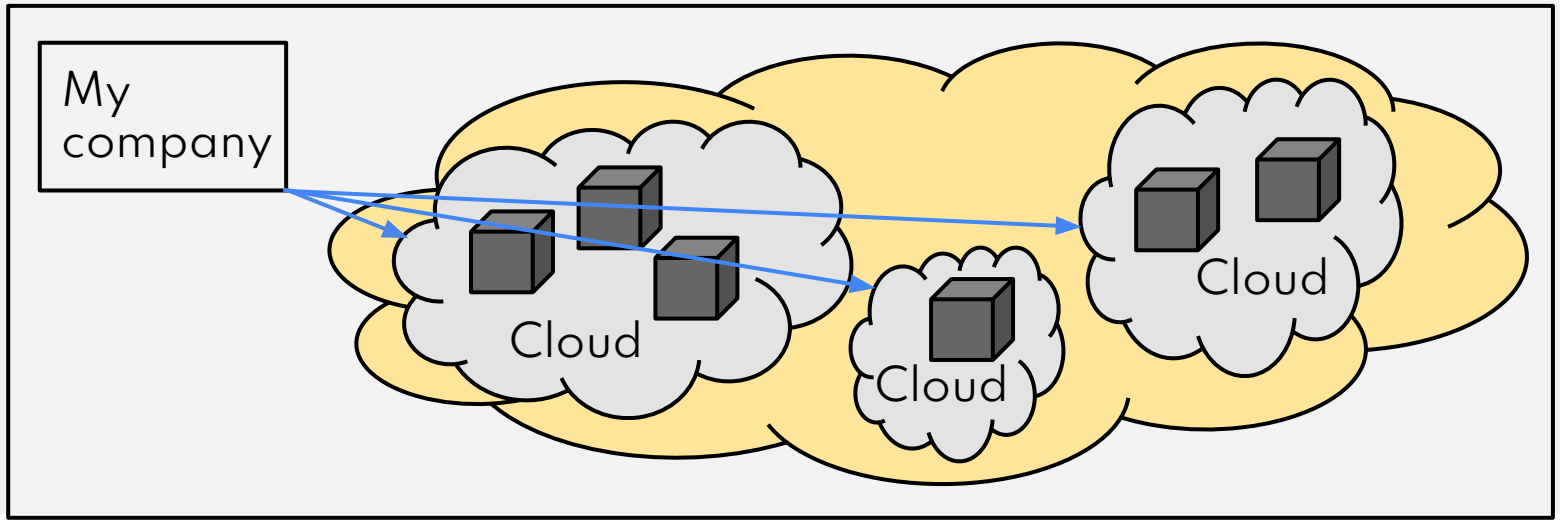


Hybrid and multi Clouds.

Hybrid Cloud

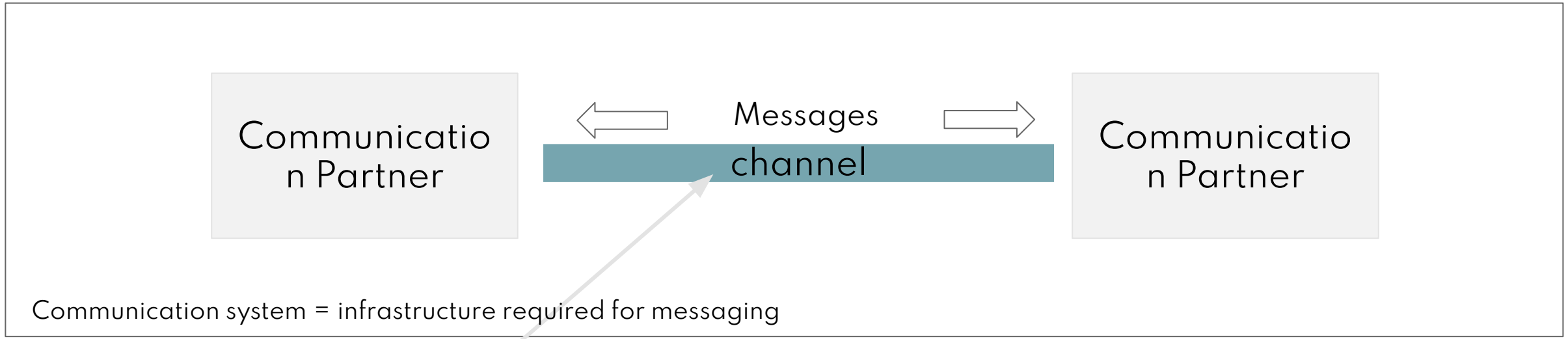


Multi Cloud



Communication

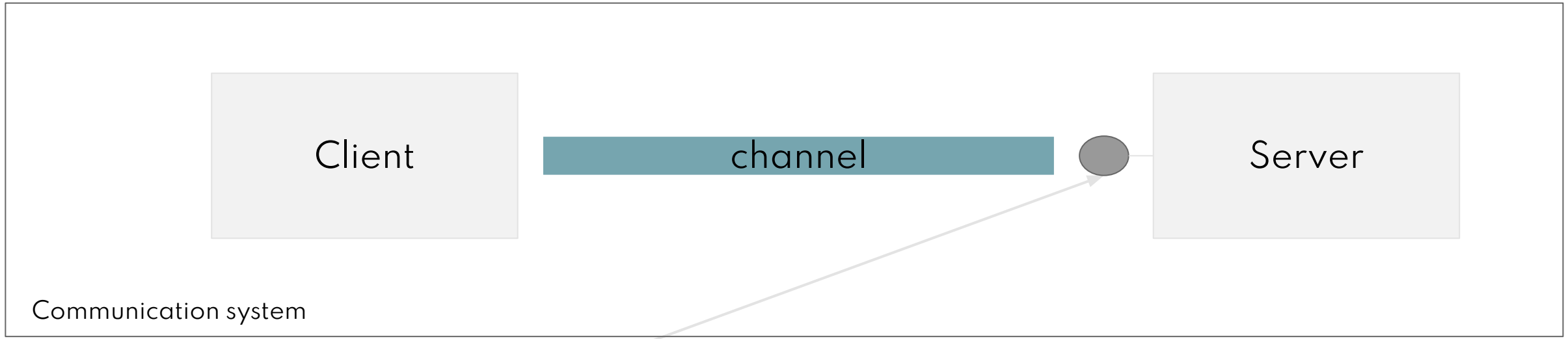
A general communication model for the internet - adapted from Shannon/Weaver.



Typical properties of a channel:

- direction
- data format
- synchronous/asynchronous
- reliability and guarantees
- security
- performance (latency, bandwidth)
- overhead (payload / total load)

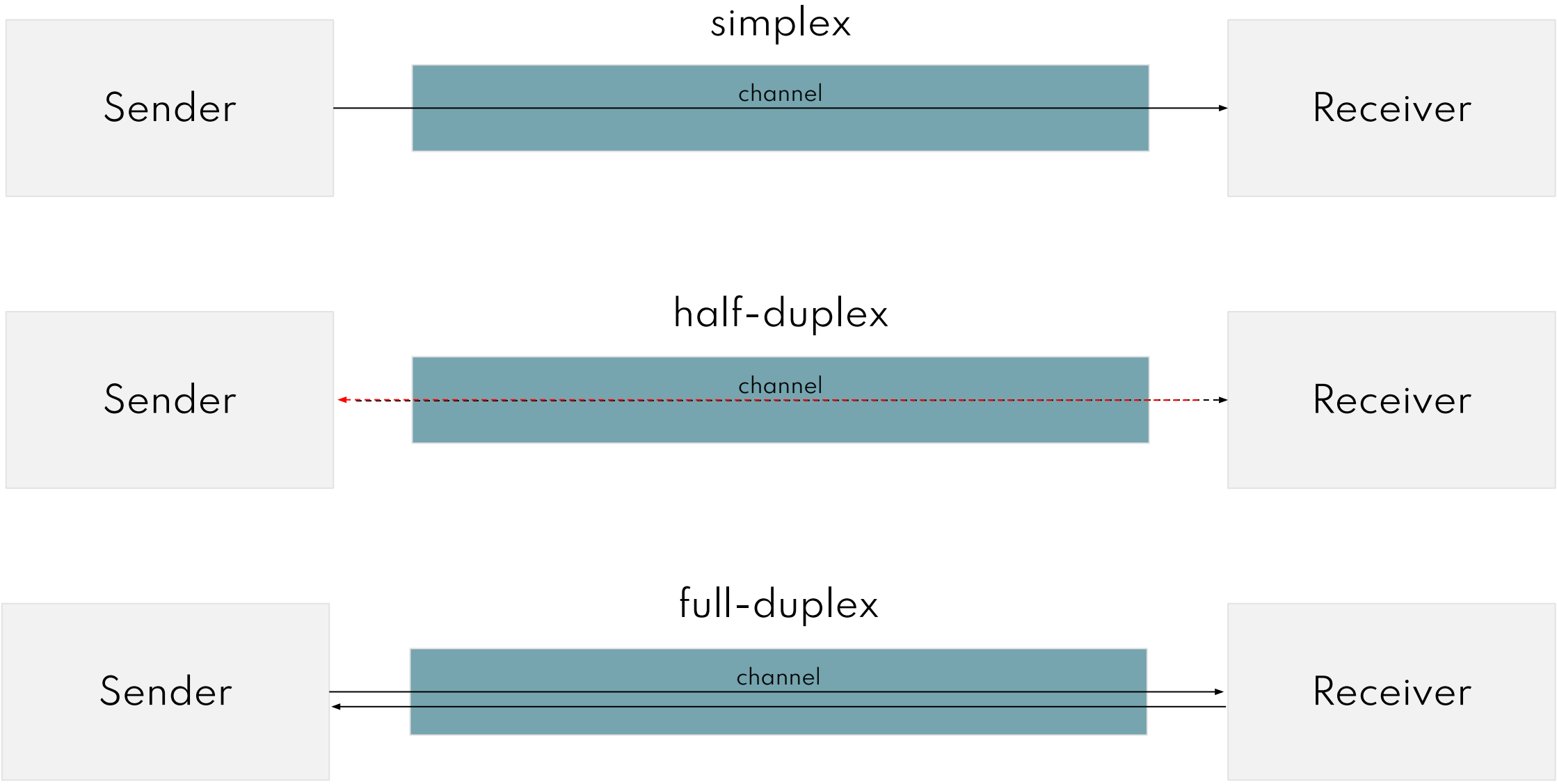
Service-orientation in a communication system: client-server-communication via services



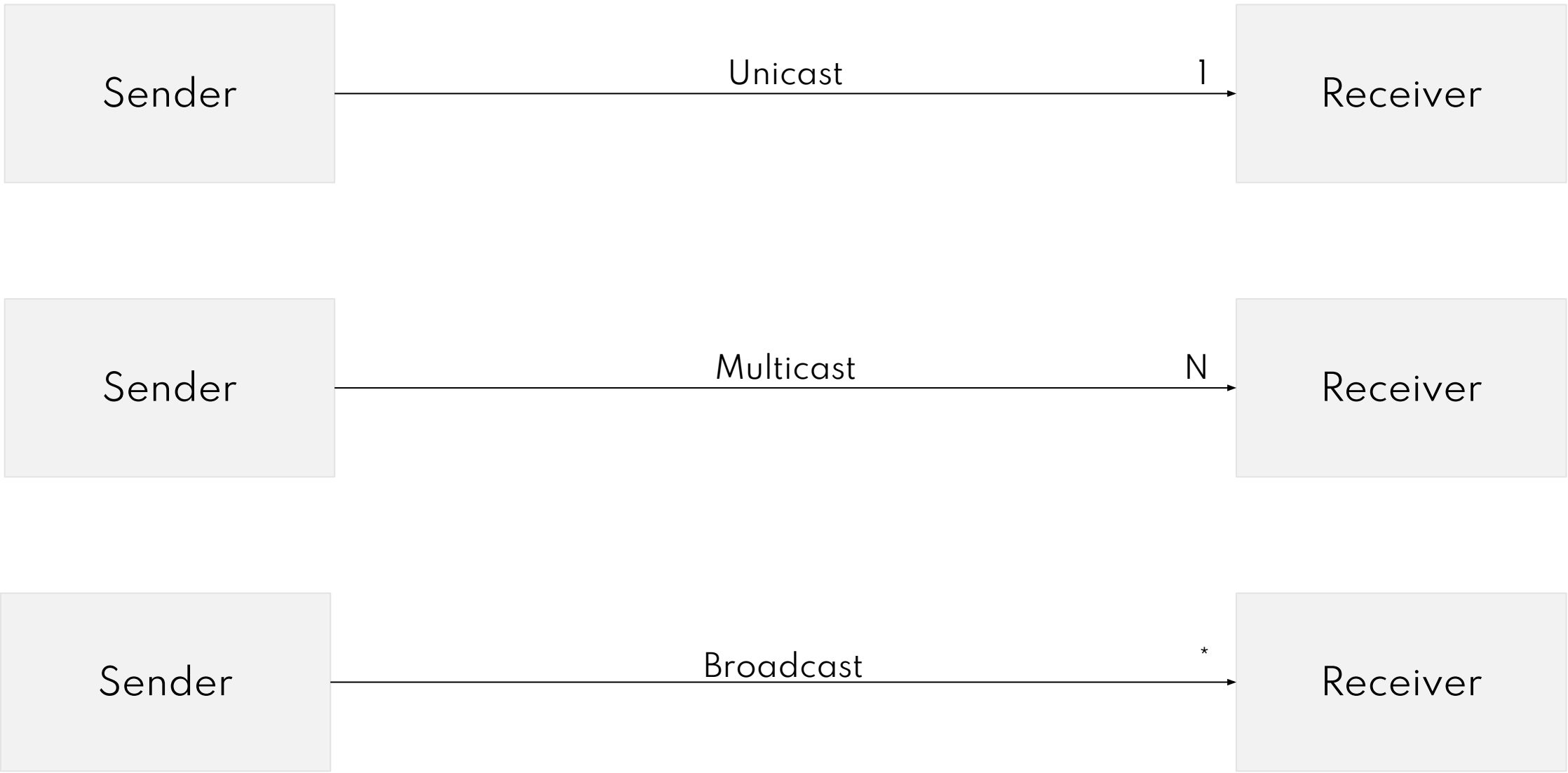
A **service** is a functionality provided through a defined interface.
Each service is defined by a service interface.

A service interface is a contract between the user and the provider regarding the syntax and semantics of service usage and optionally includes guarantees regarding the Quality of Service.

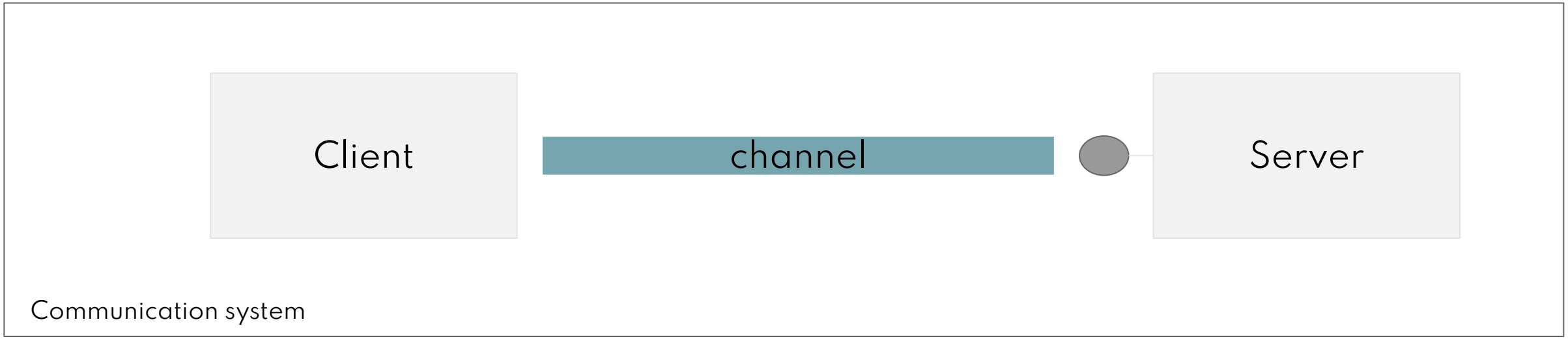
Usage patterns of a channel (Direction)



Cardinality of message receivers



Who initiates the communication?



Client starts communication

Request-Response

requests response from server

Unidirectional notification

Push

server starts communication

Peer-to-Peer

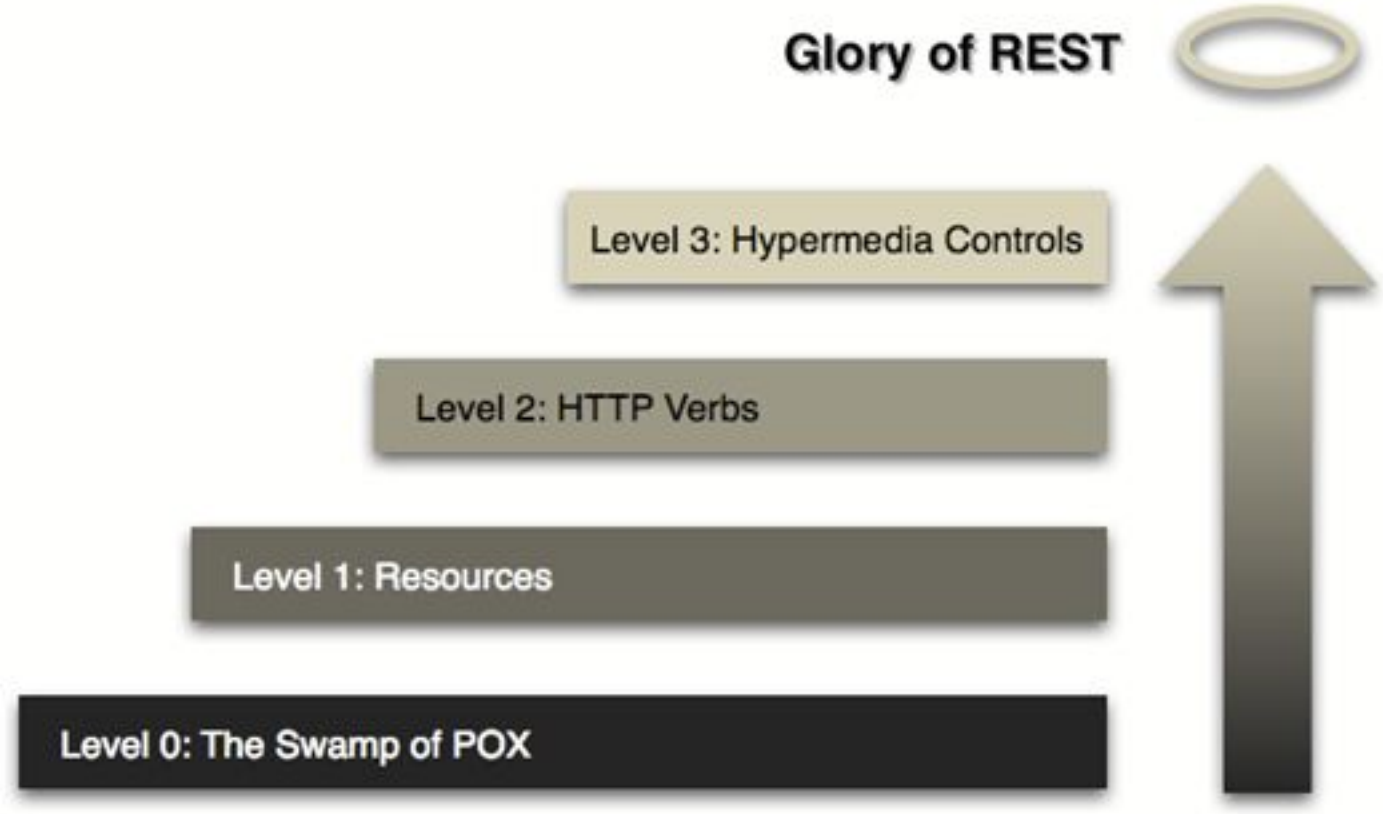
Both communication partners may start the communication.

REST is a paradigm for application services based on the HTTP protocol

Basic properties:

- **Everything is a resource**: A resource is uniquely addressable via a URI, has one or more representations (XML, JSON, any MIME type), and can link to other resources via hyperlink. Resources are, wherever possible, hierarchically navigable.
- **Uniform interfaces**: Services are based on HTTP methods (POST = create, PUT = update or create, DELETE = delete, GET = retrieve). Errors are reported through HTTP codes. Thus, services have standardized semantics and a stable syntax.
- **Stateless**: The communication between server and client is stateless. A state is maintained on the client only through URIs.
- **Connectivity**: Based on mature and ubiquitous infrastructure: The web infrastructure with effective caching and security mechanisms, powerful servers, and, for example, web browsers as clients.

The REST maturity model



https://martinfowler.com/articles/richardsonMaturityModel.html

What do **updates** for orders look like?

http://www.service.de/customers/4711/orders
Input: Order per PUT
Output: HTTP-Code (200, 500) +
Link to changed Order

http://www.service.de/customers/4711/orders
Input: Order per PUT
Output: HTTP-Code (200, 500)

http://www.service.de/customers/4711/orders
Input: Order per POST
Output: UpdateOrderConfirmation (OK)

http://www.service.de/customers/updateOrderService
Input: UpdateOrderRequest per POST
Output: UpdateOrderConfirmation (OK)

REST-API with Java's JAX-RS

Request Path

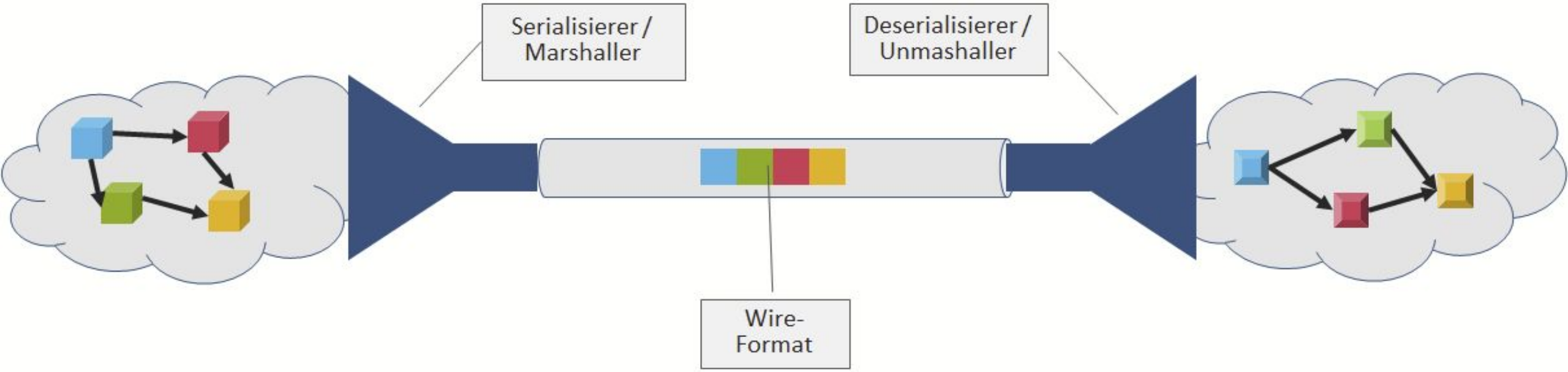
HTTP Verb

HTTP Content-Type

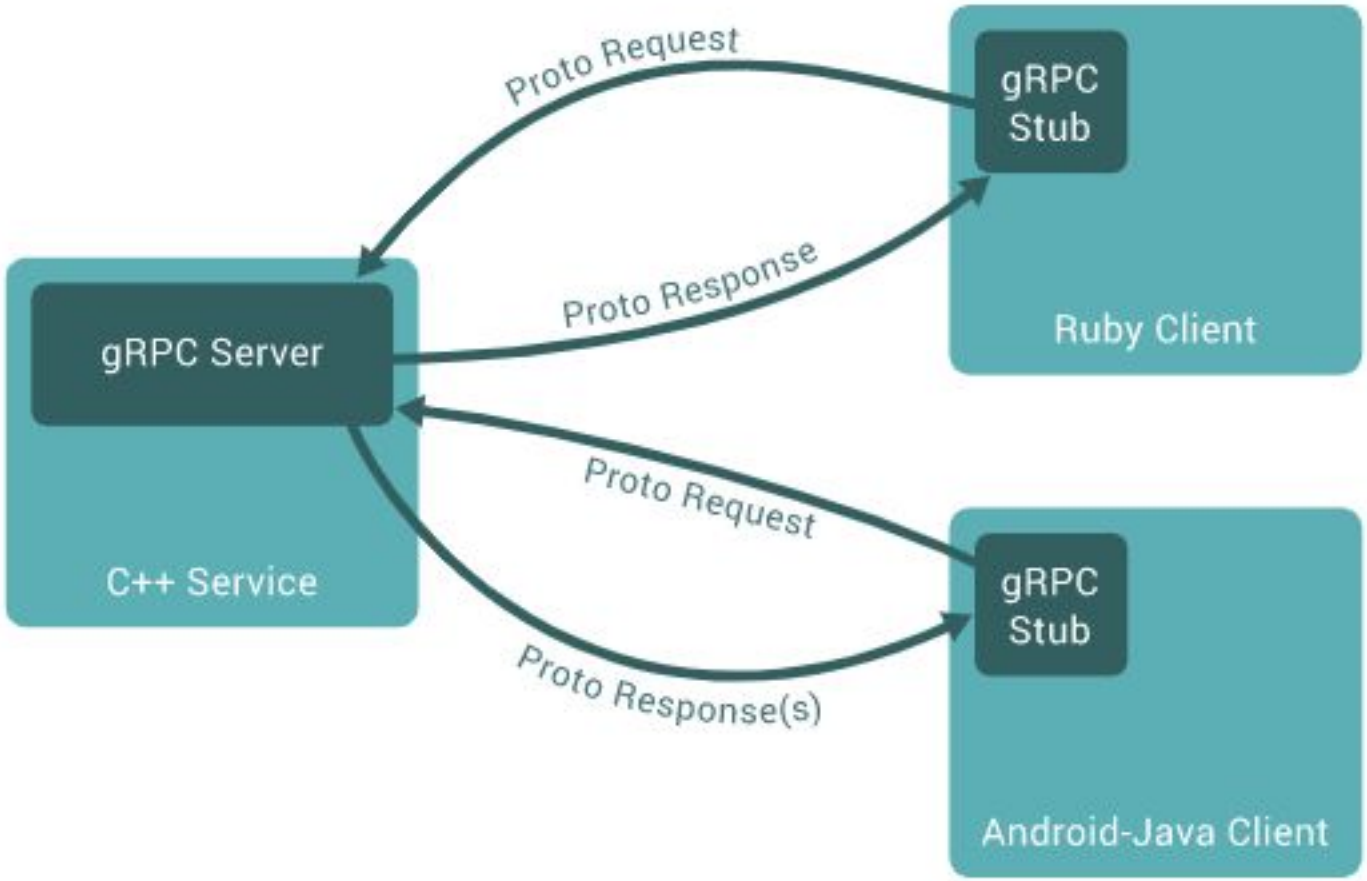
```
@Path("/hello/{name}")
public class HelloWorldResource {

    @GET
    @Produces("application/json")
    public ResponseMessage getMessage(
            @DefaultValue("Hallo") @QueryParam("salutation") String salutation,
            @PathParam("name") String name) throws IOException {
        ResponseMessage response = new ResponseMessage(new Date().toString(), salutation + " " + name);
        return response;
    }

}
```

Usually serialization & deserialization is needed

Serialization/Marshalling is the process of converting an object, data structure, or complex data format (such as a Python object, Java object, etc.) into a format that can be easily stored or transmitted and later reconstructed. The resulting data can be stored in files, sent over a network, or communicated between different systems.



gRPC



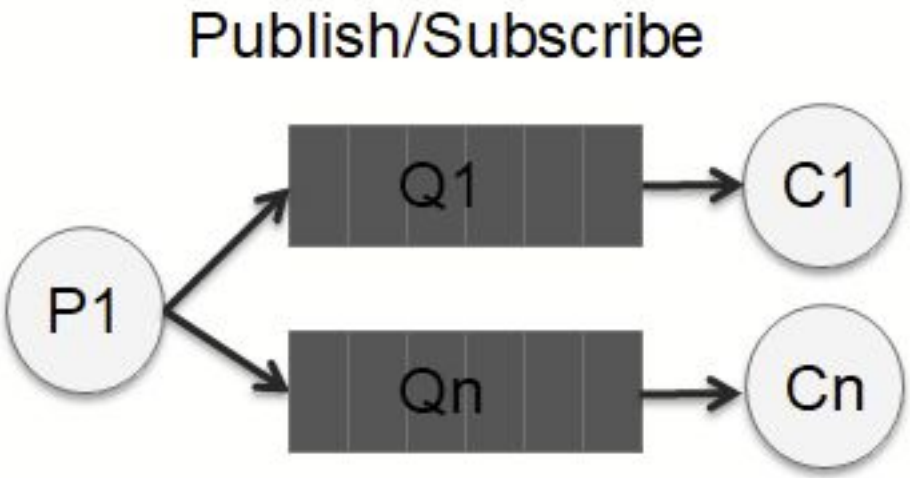
https://grpc.io/docs/what-is-grpc/introduction/

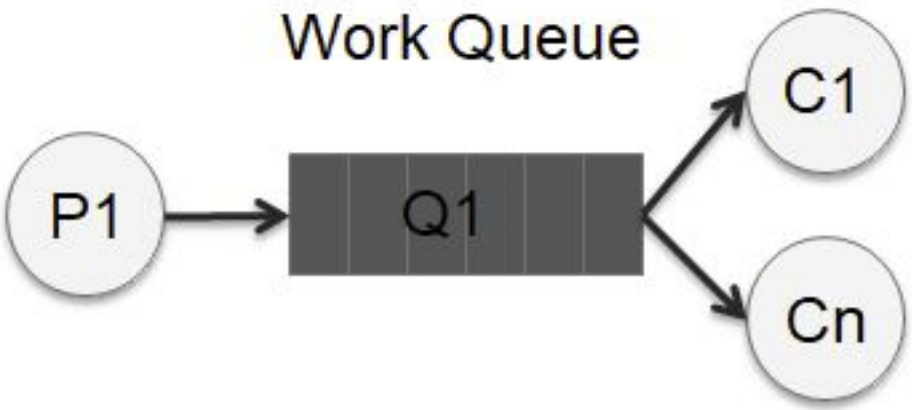
Resilient and asynchronous messaging

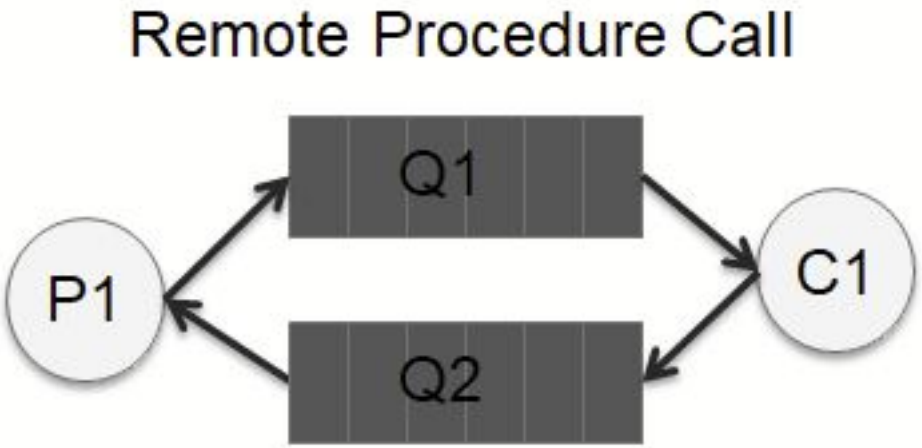
Decouples Producer and Consumer

- Service interface: application format
- Message broker: no restrictions on the format
- Advantages
  - The sending and receiving times can be spaced apart as long as needed
  - Horizontal scalability: Can be delivered to multiple consumers
  - Load balancing: e.g., delivery to the consumer with the least workload
  - Handling peak loads: Message delivery can be delayed if consumers are overloaded
- Configuration options:
  - TTL (Time to Live) of the message
  - Delivery guarantee (at least once, exactly once, no guarantee)
  - Transactionality
  - Message priority
  - Order compliance

Messaging Patterns







Virtualization

Virtualization types

Virtualization is representative of several fundamentally different concepts and technologies.

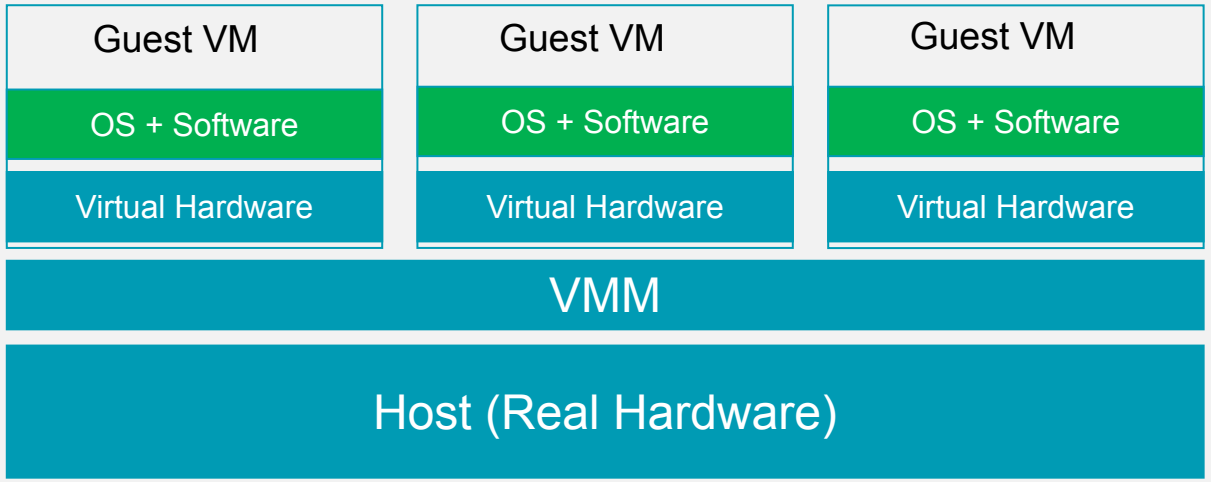
Virtualization of hardware infrastructure

1. Emulation
2. Full virtualization (Type 2 virtualization)
3. Para virtualization (Type 1 virtualization)

Virtualization of software infrastructure

4. Operating system virtualization (*Containerization*)
5. Application virtualization (*Runtime*)

Hardware virtualization



- **Hardware virtualization** divides the resources of a computer system so that they can be used by multiple independent operating system instances.

- The requirements of the operating system instances are intercepted by the virtualization software (**virtual machine monitor, VMM**) and implemented on the actual hardware.

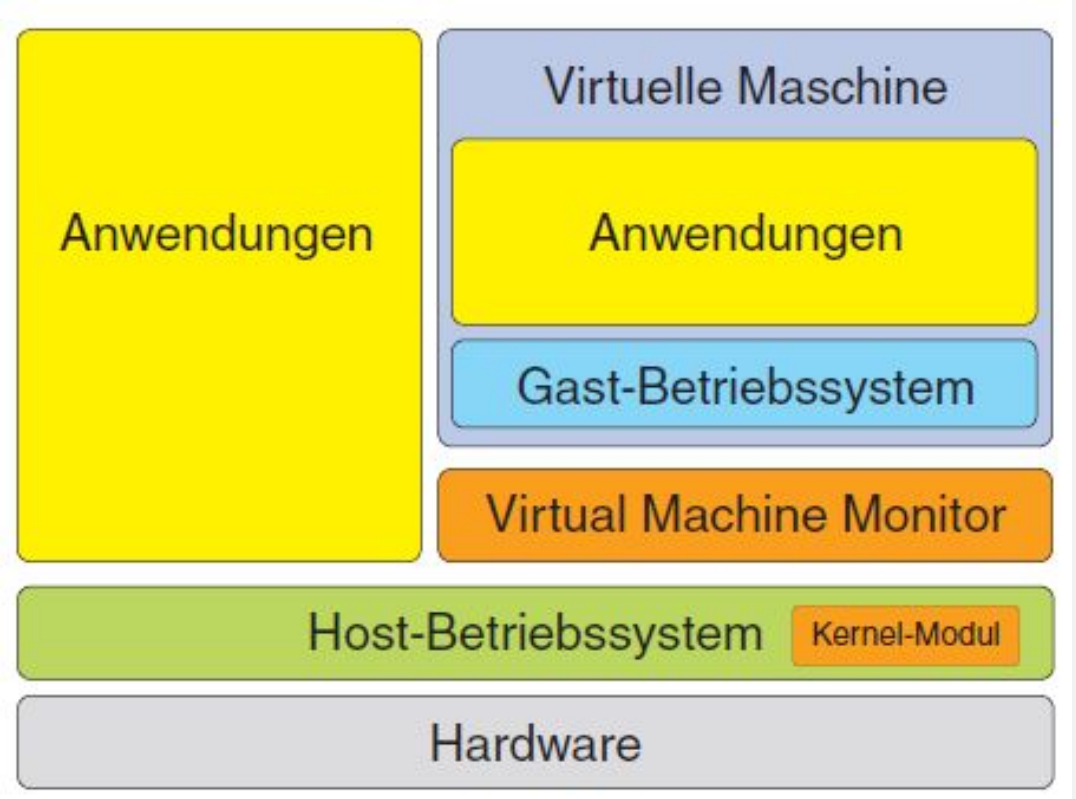
Host
- The computer that runs one or more virtual machines and provides the necessary hardware resources.

Guest
- A runnable / running virtual machine

VMM (Virtual Machine Monitor)
- The control software for managing guests and host resources

Hardware virtualization: Full virtualization

- Each guest operating system has its own virtual computer with virtual resources such as CPU, main memory, disk drives, network cards, etc.
- The VMM runs as an application hosted by the host operating system (type 2 hypervisor)
- The VMM distributes the computer's hardware resources among the VMs
- The VMM partially emulates hardware that is not designed for simultaneous access by multiple operating systems (e.g. network cards, graphics cards)
- Performance loss: 5-10%.



Hardware virtualization: para-virtualization

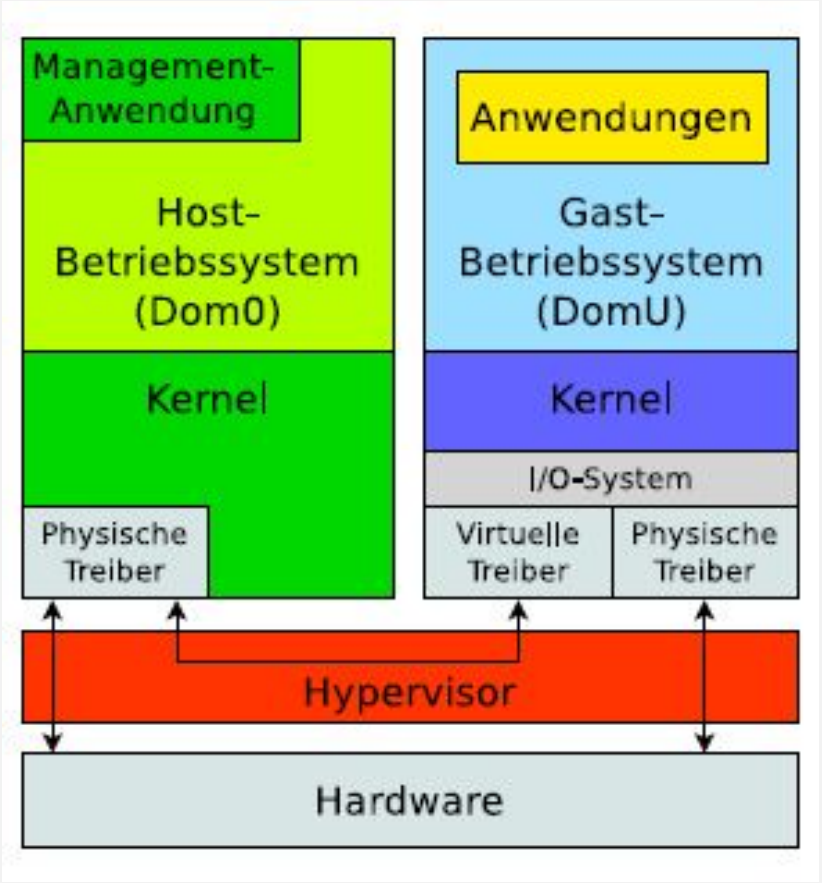
The hypervisor runs directly on the available hardware. It is therefore an operating system that is exclusively designed for virtualization.

Virtual drivers must be added to the guest operating system in order for it to interact with the hypervisor.

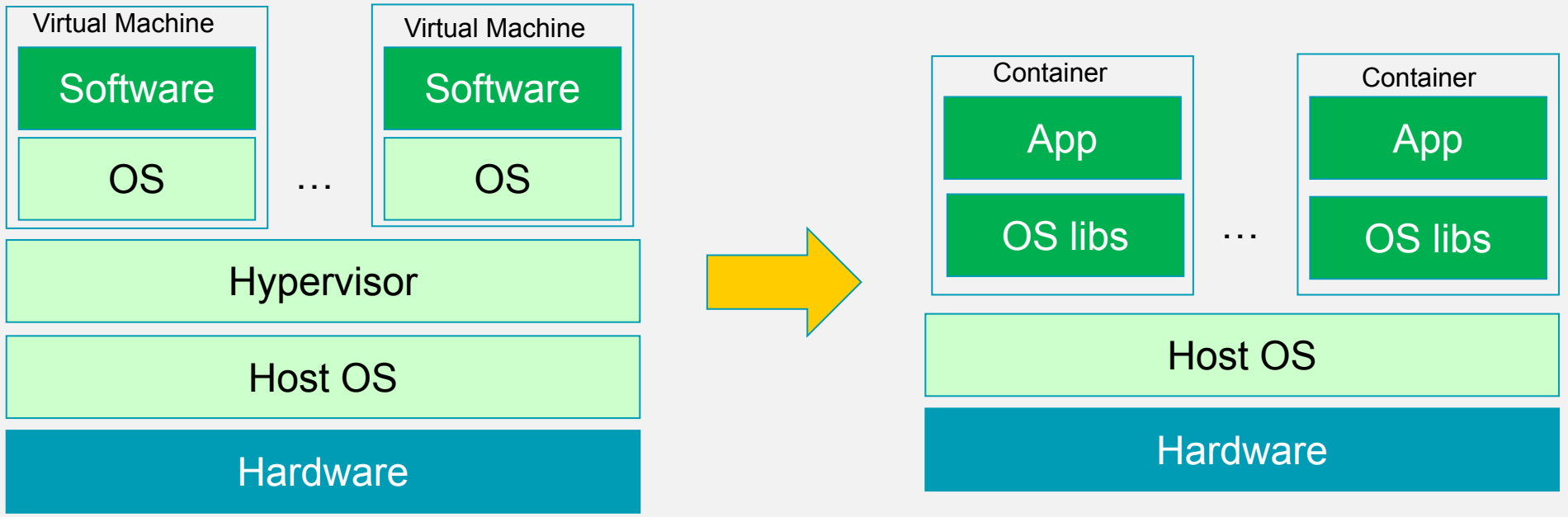
No directly low-level virtualized hardware resources (CPU, RAM, etc.) are available to the guest operating system, but an API is available for use by the virtual drivers.

> Supported operating systems and hardware variants from the guest's point of view are limited per hypervisor implementation.
> The hypervisor uses the drivers of a host operating system to access the real hardware. This eliminates the need for the hypervisor to implement its own drivers.

Performance loss: 2-3%



Operating system virtualization



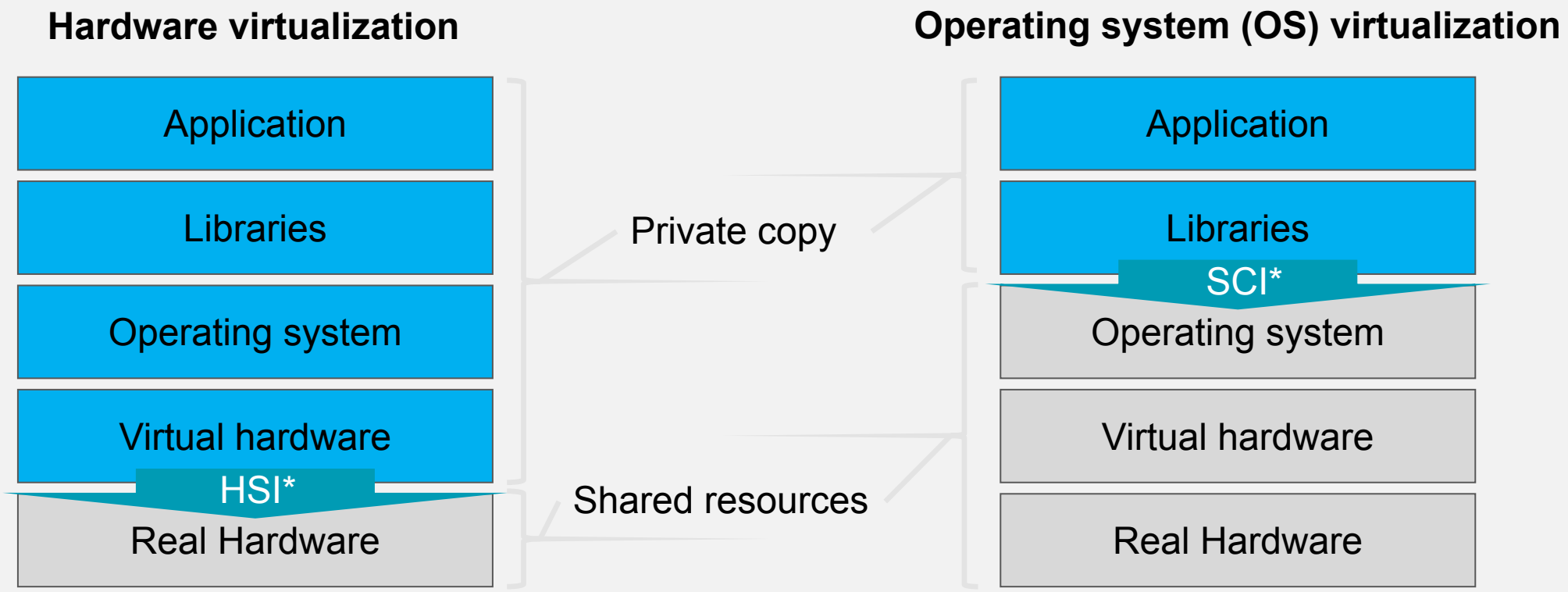
Lightweight virtualization approach: There is no hypervisor. Each app runs directly as a process in the host operating system. However, this is maximally isolated by corresponding OS mechanisms (e.g. Linux LXC).

- Isolation of the process through kernel namespaces (regarding CPU, RAM and disk I/O) and containments
- Isolated file system
- Separate network interface

CPU/RAM overhead generally not measurable (~ 0%)

Startup time = start duration for the first process

Hardware- vs. Operating system virtualization

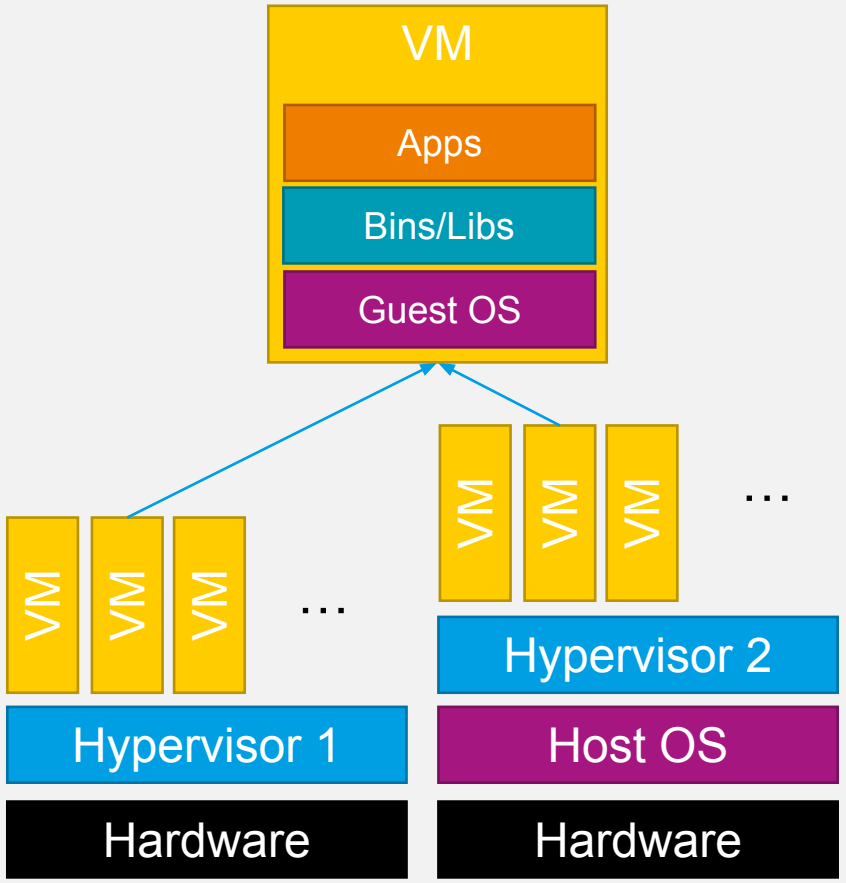


- Better insulation
- Higher security

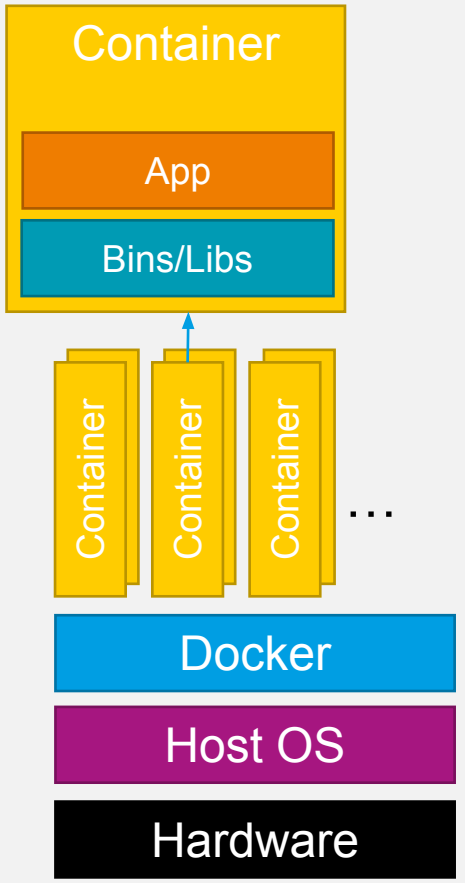
- Smaller private copy volume
- Lower overhead
- Faster startup time

*) HSI = Hardware Software Interface
SCI = System Call Interface

Containerization with Docker

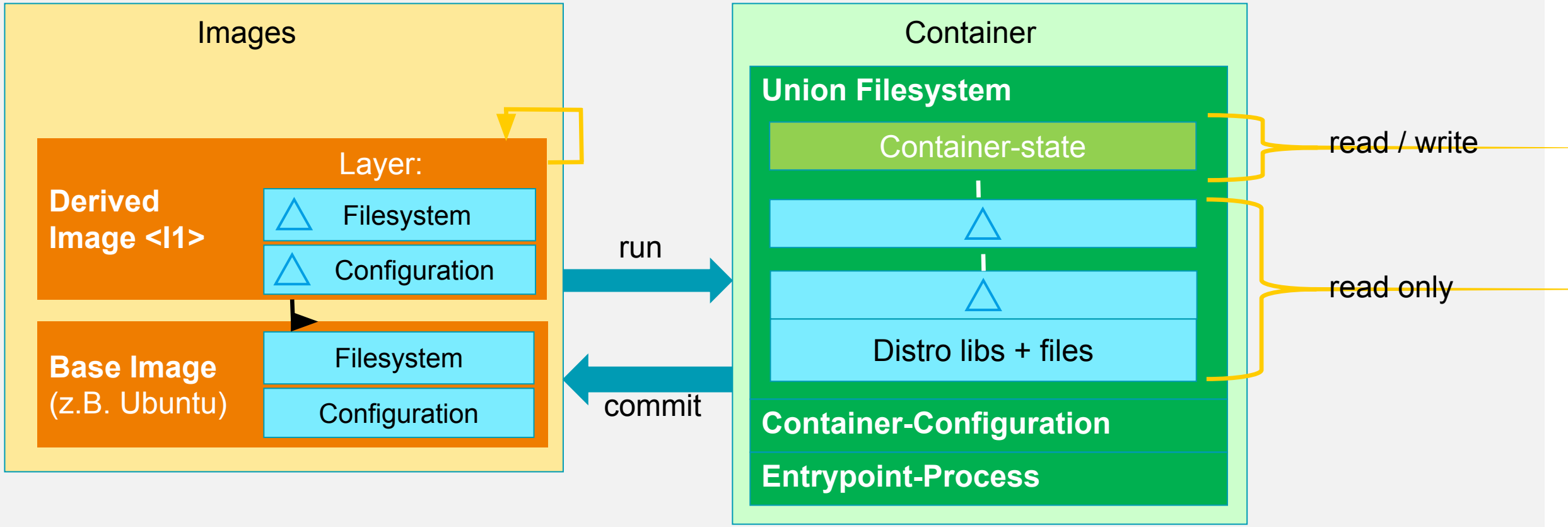


Type 1 / Type 2 Virtualization



Containerization

At the center of Docker are images and containers.

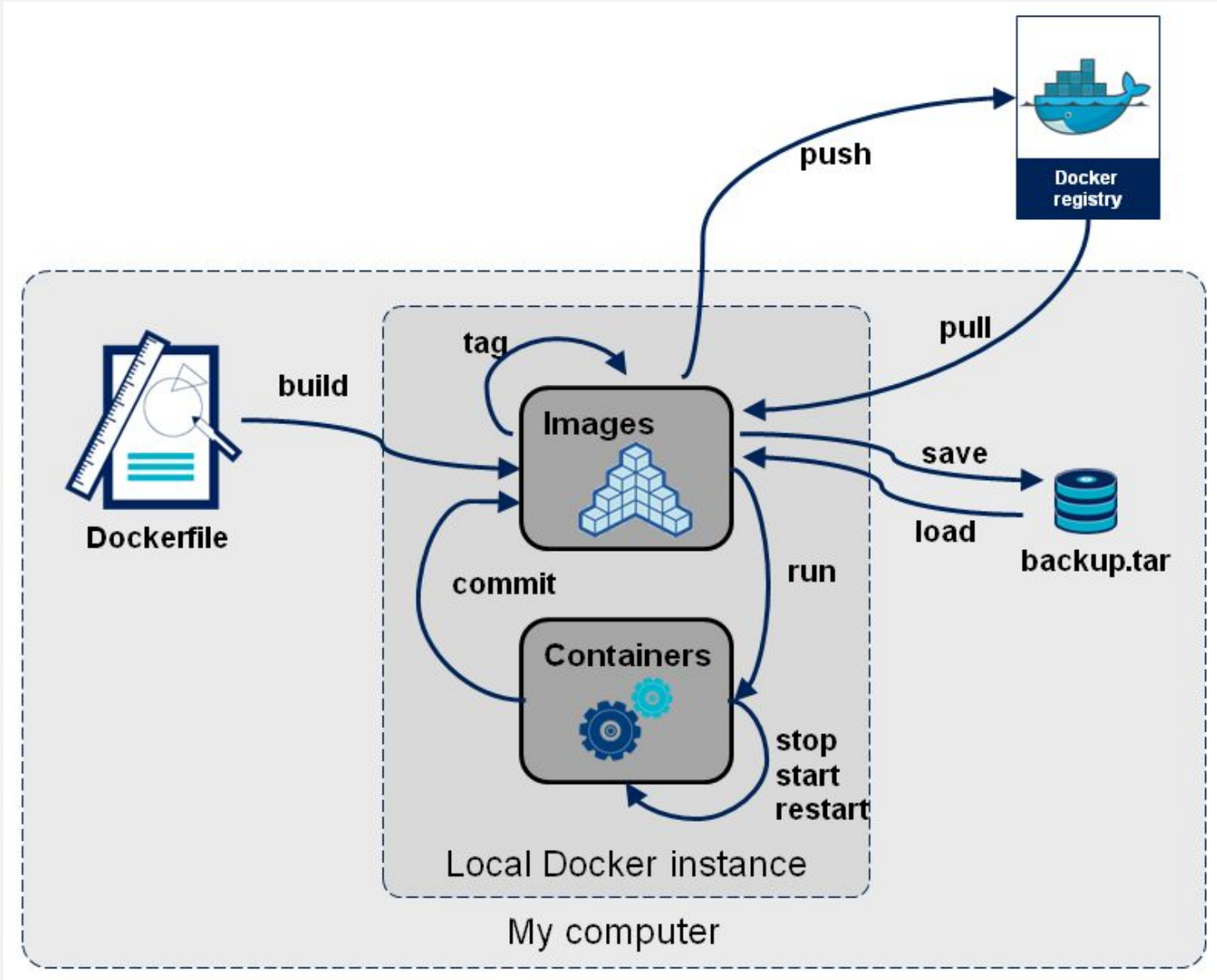


stationary and transportable state

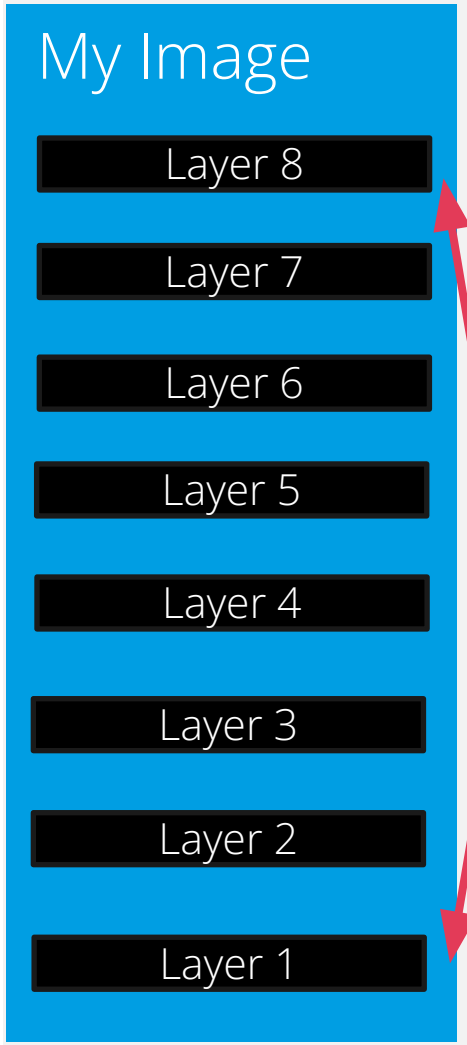
Running state

A container runs as long as its entrypoint process is in the foreground. Docker remembers the container state..

The Docker Workflow.



The Dockerfile defines Structure and content of the image.



Never use latest. Antipattern

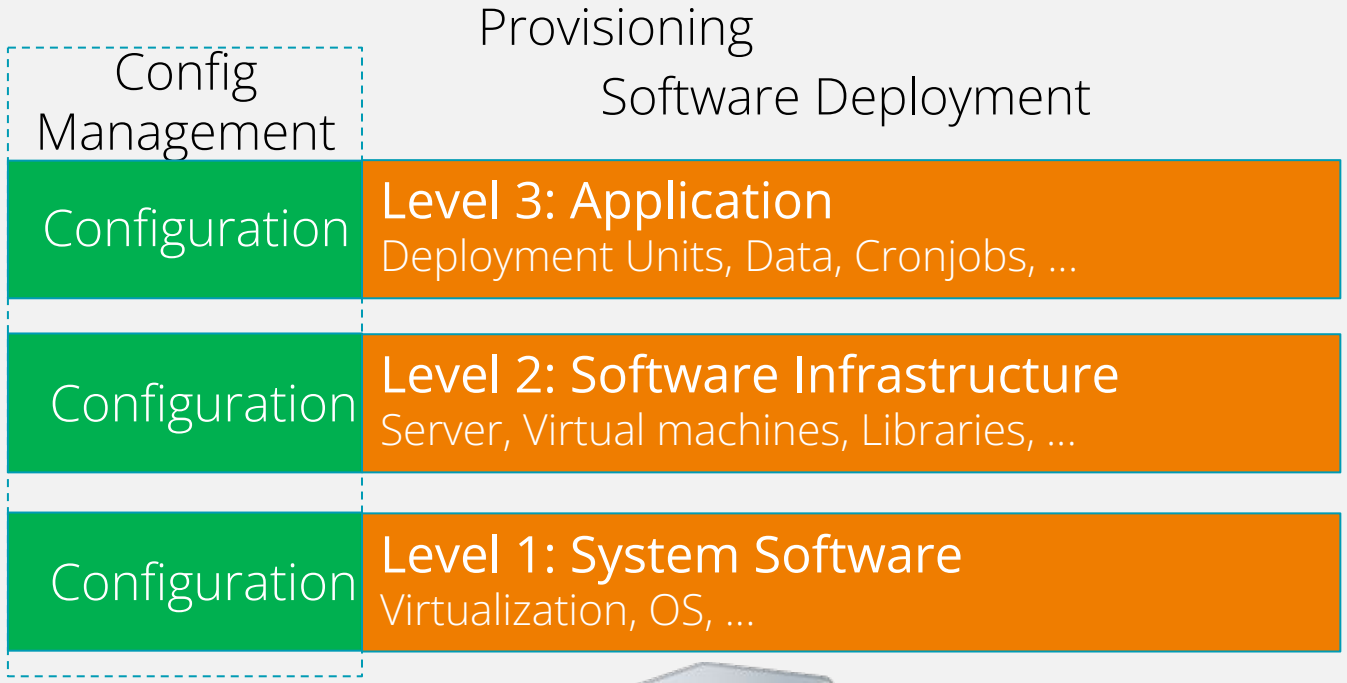
```
FROM qaware/alpine-k8s-ibmjava8:8.0-3.10
LABEL maintainer="QAware GmbH <qaware-oss@qaware.de>"

RUN mkdir -p /app

COPY build/libs/zwitscher-service-1.0.1.jar /app/zwitscher-service.jar
COPY src/main/docker/zwitscher-service.conf /app/

ENV JAVA_OPTS -Xmx256m

EXPOSE 8080
ENTRYPOINT ["java", "-jar", "/app/zwitscher-service.jar"]
```

Dockerfile commands

Element	Meaning
FROM <image-name>	Sets to base image (where the new image is derived from)
MAINTAINER <author>	Document author
RUN <command>	Execute a shell command and commit the result as a new image layer (!)
ADD <src> <dest>	Copy a file into the containers. <src> can also be an URL. If <src> refers to a TAR-file, then this file automatically gets un-tared.
VOLUME <container-dir> <host-dir>	Mounts a host directory into the container.
ENV <key> <value>	Sets an environment variable. This environment variable can be overwritten at container start with the –e command line parameter of `docker run`.
ENTRYPOINT <command>	The process to be started at container startup
CMD <command>	Parameters to the entrypoint process if no parameters are passed with `docker run`
WORKDIR <dir>	Sets the working dir for all following commands
EXPOSE <port>	Informs Docker that a container listens on a specific port and this port should be exposed to other containers
USER <name>	Sets the user for all container commands

Provisioning

Provisioning takes place on three different levels and in four stages.



Hardware

- Computers
- Memory
- Storage
- Network equipment
- ...

Running Software!

4 Application Provisioning

3 **Server Provisioning**
Provision of the necessary software infrastructure for the application.

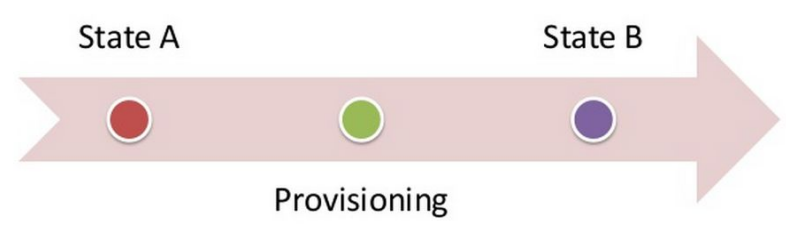
2 **Bootstrapping**
Providing the operating environment for the software infrastructure.

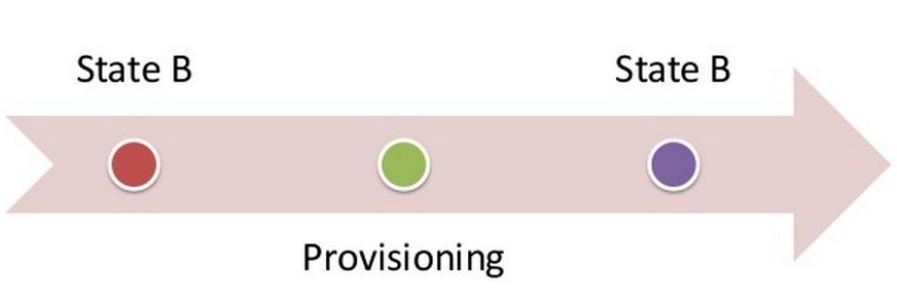
1 **Bare Metal Provisioning**
Initialisierung einer physikalischen Hardware für den Betrieb.

Conceptual considerations for provisioning.

QA|WARE

System status := The totality of software, data and configurations on a system.
Provisioning := Transfer from a system's current state to a target state.





What a provisioning mechanism has to do:

1. Determine the initial state
2. Check preconditions
3. Determine state-changing actions
4. Execute state-changing actions
5. Check postconditions and reset state if necessary

Guarantees

Idempotency: The ability of an action to produce the same result whether it is performed once or multiple times.

Consistency: After the actions have been carried out, the system state is consistent, regardless of whether individual, several or all actions have failed.

The new lightness of being.

Old Style

Any state

1. Determine initial state
2. Check preconditions
3. Determine actions that change the state
4. Execute actions that change the state
5. Check postconditions and, if necessary, reset the state

Target state

New Style

„Immutable Infrastructure / Phoenix Systems“

Base state

1. ~~Determine initial state~~
2. ~~Check preconditions~~
3. ~~Determine actions that change the state~~
4. Execute actions that change the state
5. Check postconditions ~~and, if necessary, reset the state~~

Target state

Provisioning with Dockerfile and Docker Compose

Deployment layers

Level 3: Application
Deployment units, Data, Cronjobs, ...

Level 2: Software Infrastructure
Server, Virtual Machines, Libraries, ...

Level 1: System Software
Virtualization, OS, ...

Docker Image Build Chain

Application Image
(z.B. www.qaware.de)

Server Image
(z.B. NGINX)

Base Image
(z.B. Ubuntu)

4 **Application Provisioning**
DockerFile & Docker Compose

3 **Server Provisioning**
Dockerfile

2 **Bootstrapping**
Docker Pull Base Image

1 **Bare Metal Provisioning**
Install Docker Daemon

Provisioning with Ansible

Deployment layers

Level 3: Application
Deployment units, Data, Cronjobs, ...

Level 2: Software Infrastructure
Server, Virtual Machines, Libraries, ...

Level 1: System Software
Virtualization, OS, ...

Docker image or VM chain

Application Image
(z.B. www.qaware.de)

Server Image
(z.B. NGINX)

Base Image
(z.B. Ubuntu)

Application Provisioning
Ansible or Ansible Container

Server Provisioning
Ansible or Ansible Container

Bootstrapping
Install SSH Daemon & Python

Bare Metal Provisioning
Install OS

Ansible – Concepts & terms

Description of the machines via IP, short names or URLs

- Inventory
- Modules
- Tasks
- Roles
- Playbook

```
[webserver]
my-web-server.example.com
my-other-web-server.example.com

[appserver-master]
app1-master absible_ssh_host=myapp.example.net
httpsports=9090
app2-master absible_ssh_host=myapp2.example.net
httpsports=9091

[appserver-slaves]
app1-slave absible_ssh_host=myapp3.example.net
httpsports=9090
app2-slave absible_ssh_host=myapp4.example.net
httpsports=9091
```

Definition of variables for individual hosts or groups

Groups combine several machines

Packer terminology (https://www.packer.io/docs/terminology)

Artifacts
- The result of a packer build, e.g. a folder of files or a set of AMI IDs

Builds
- Tasks that create an image for a specific platform

Builders
- create a specific image type
- e.g. VirtualBox, Amazon EC2, Docker

Commands
- Subcommands that can be executed with packer, e.g. packer build

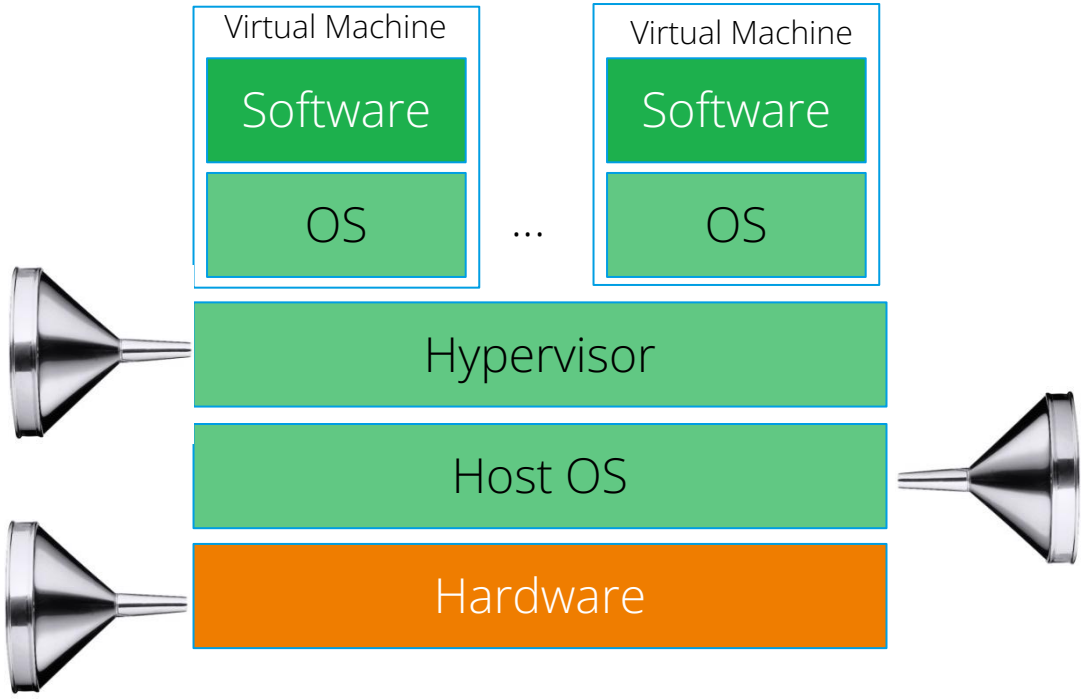
Post processors
- Create new artifacts from existing artifacts (e.g. compression, tagging, publishing)

Provisioners
- Install and configure software in a running instance before creating a static artifact from it

Templates
- JSON Files, that configure the Packer Build

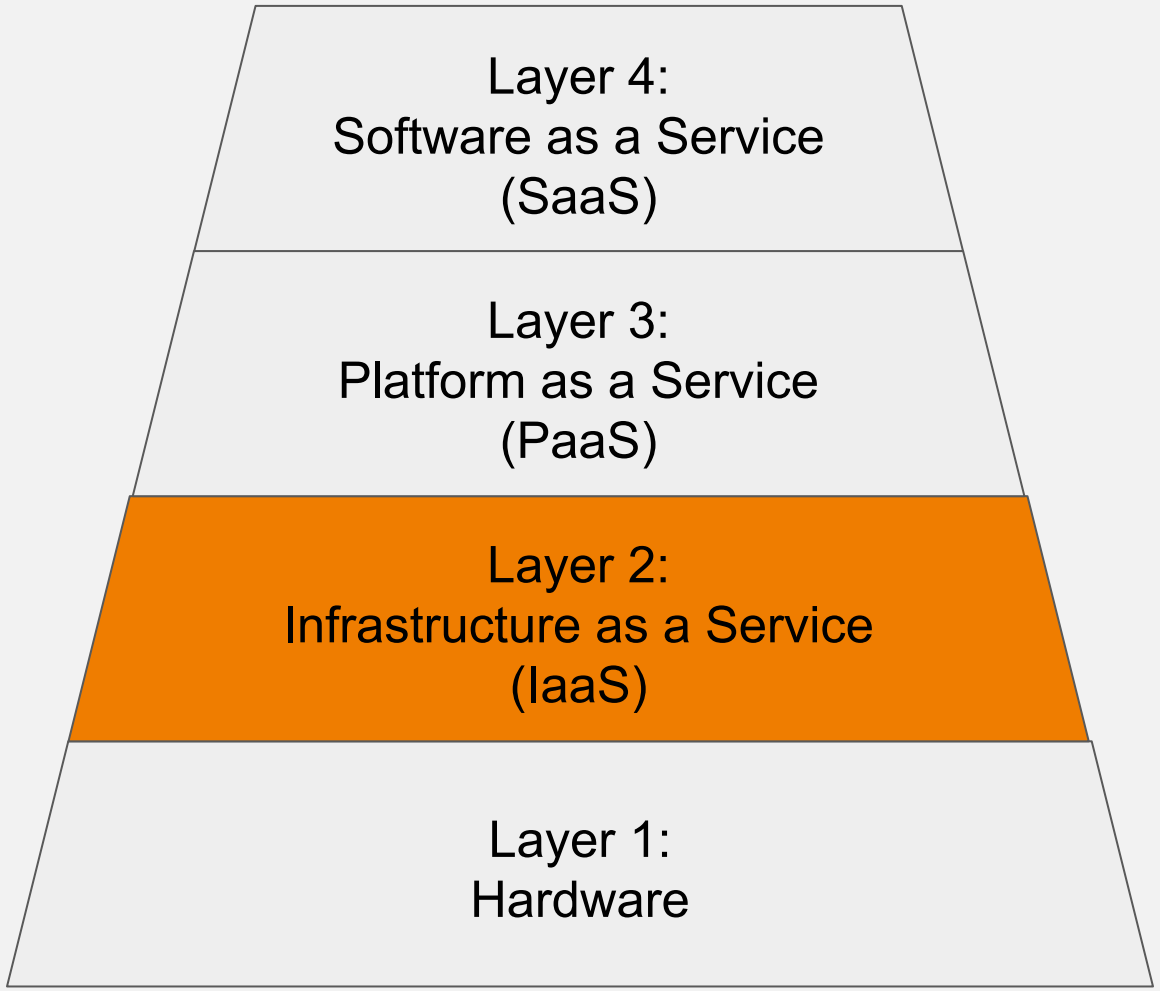
IaaS

How to get software onto infrastructure/hardware?



The layered model of cloud computing: From metal to application.

QA|WARE



- Customizable software services
- CaaS: Component as a service (e.g. Google Charts)
- BaaS: Backend as a Service
- Transparent Updates

Target Group: Users

- Platform services
- Application programming interfaces (APIs)
- Abstraction of technical infrastructure

Target group: Developers

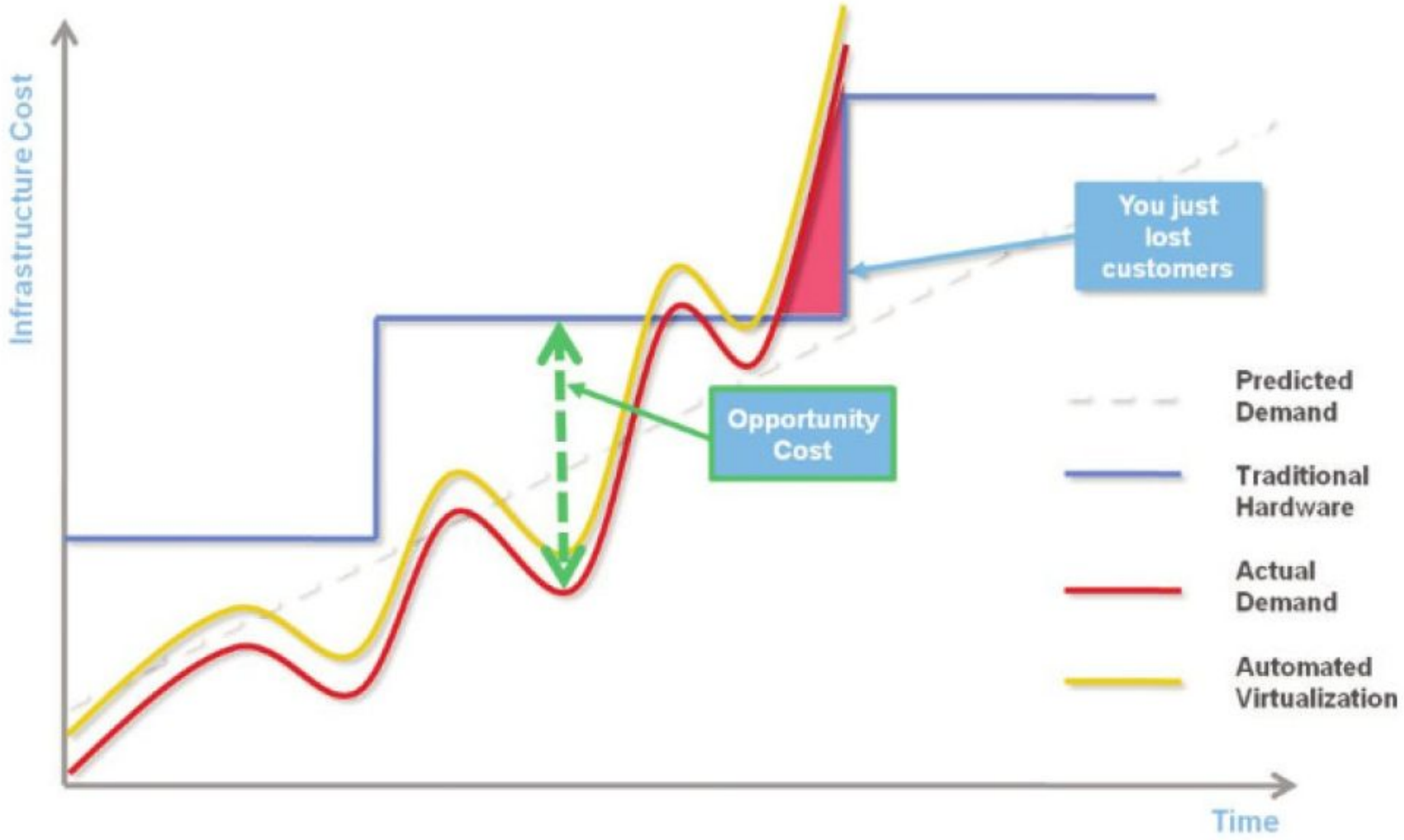
- Elasticity
- Virtual resourcepools
- Technical infrastructure: Machines, Servers (DNS, DHCP, LB, NAS, …)

Target group: Operations

- Computers
- Network
- Storage

https://www.youtube.com/watch?v=M988_fsOSWo

Classical operations are expensive in times of dynamic demand



Source: Amazon Web Services

Definition IaaS

> IaaS refers to a business model that, contrary to the traditional purchasing of computing infrastructure, provides for renting and releasing it as needed.

Properties of an IaaS-Cloud:

- **Resource-Pools**: Availability of seemingly unlimited resources, that process requests in a distributed manner
- **Elasticity**: Dynamic allocation of additional resources based on demand
- **Pay-as-you-go Modell**: Only pay for what you actually use

Resource-Types in an IaaS-Cloud:

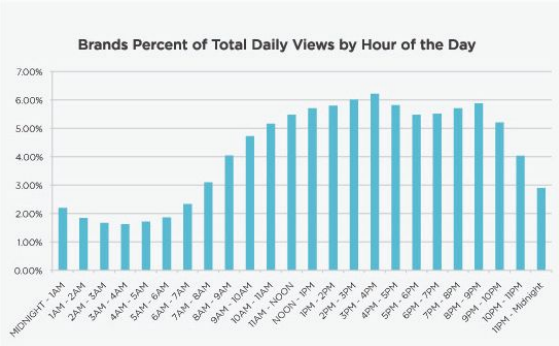
- **Compute**: Compute-Nodes with CPU & RAM
- **Storage**: Storage-capacity via mountable filesystems, block storage or database services.
- **Network**: Network and network-services like DNS, DHCP, VPN, CDN and Load Balancer.

Infrastructure-services in an IaaS-Cloud:

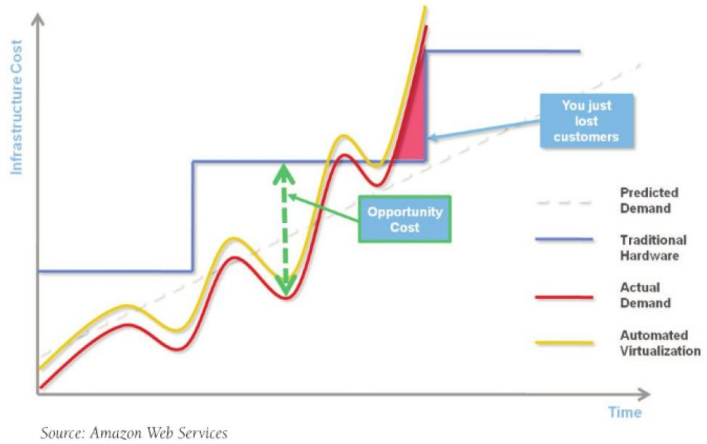
- **Monitoring**
- **Resource-Management**

Scalability: Effects

- **Daily and seasonal effects:** midday peak, prime-time peak, weekend peak, Christmas, Valentine's Day, Mother's Day, etc. (predictable load peaks)

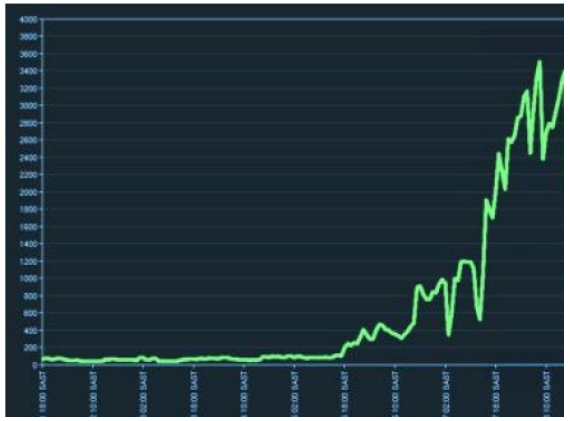


- Continuous growth

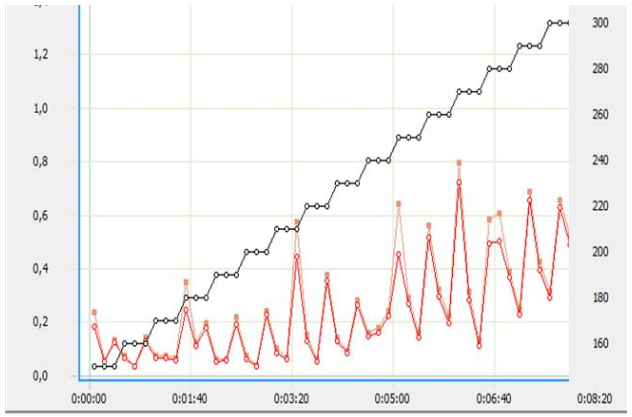


Source: Amazon Web Services

- **Special effects**: z.B. Slashdot-Effekt (unpredictable load peaks)



- **Temporary Platforms**: Projects, Tests, Batch…



Kinds of elasticity

Demand elasticity: Allocated resources increase/decrease with demand.

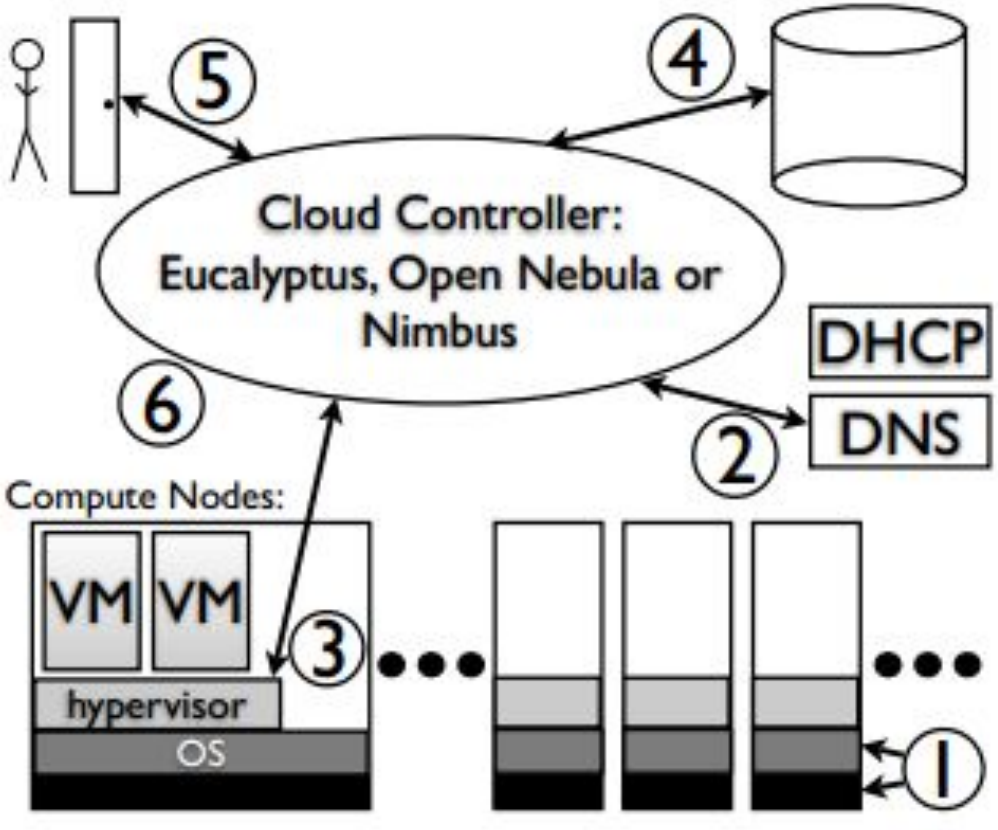
> **Pseudo-elasticity:** Quick setup. Short cancellation period.
>
> **Real-time elasticity:** Allocation and release of resources within seconds. Automated process with manual triggers or on schedule.
>
> **Self-adaptive elasticity:** Automatic allocation and release of resources in real-time based on rules and metrics.

Supply elasticity: Allocated resources increase/decrease with supply.

> This is typical behavior of a grid: all available machines are allocated.
> Variants are also available where one can bid for free resources.

Income elasticity: Allocated resources increase/decrease with income or budget.

An IaaS reference architecture.

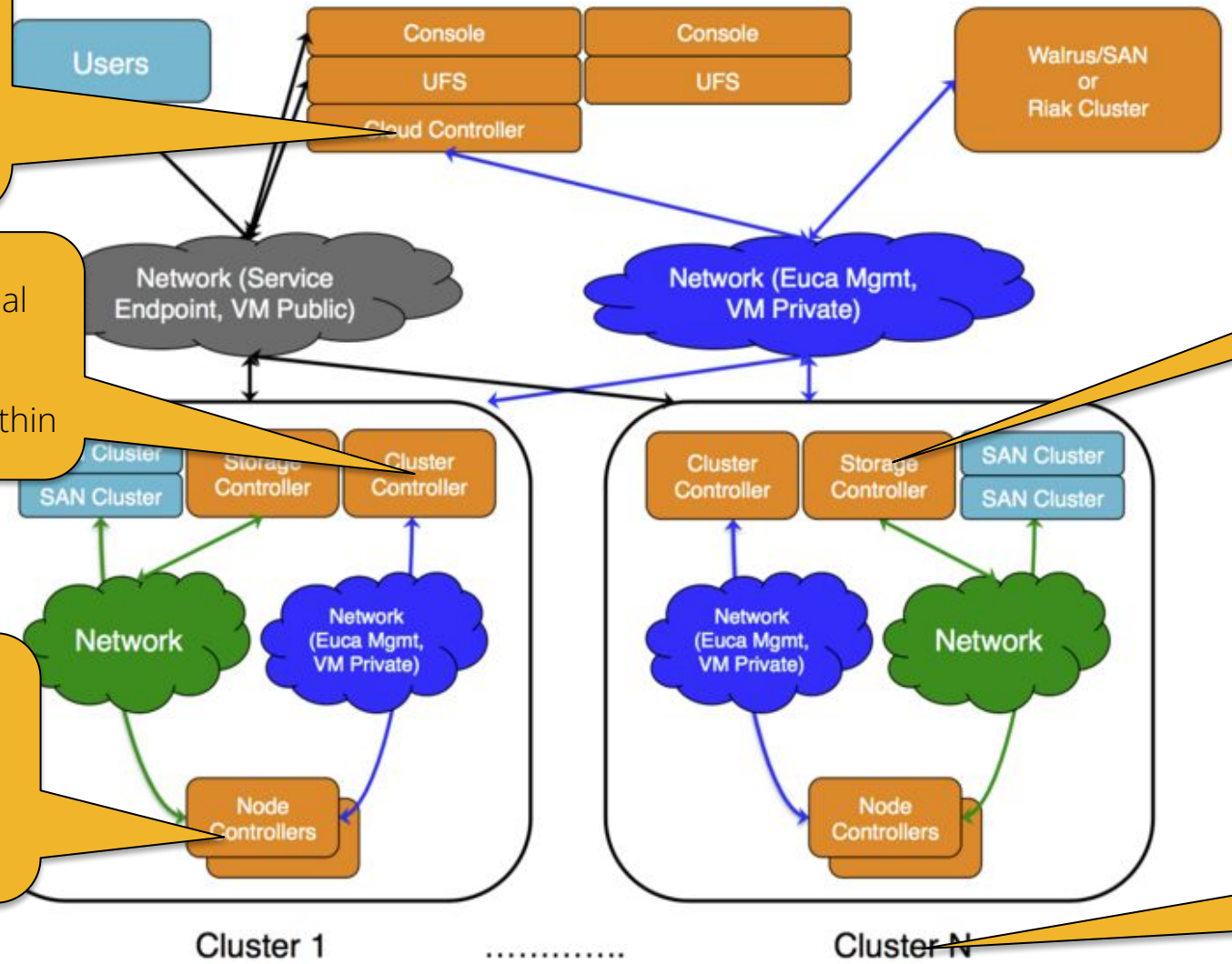


1. Hardware and OS
2. Virtual network and network services
3. Virtualization
4. Storage and Image management
5. Management interface for admins and users
6. Cloud Controller for tenant specific management of Cloud-Ressourcen

Peter Sempolinski and Douglas Thain,
“A Comparison and Critique of Eucalyptus, OpenNebula and Nimbus”,
IEEE International Conference on Cloud Computing Technology and Science, 2010.

Internal architecture of an IaaS-Cloud based on Eucalyptus.

Cloud Controller
Central control unit of the cloud.
Aggregates information from the CCs and distributes and manages all virtual resources.
Offers a central storage service (Walrus/S3), among other things, for virtual images.



Cluster Controller.
Controls the distribution of virtual images on nodes based on NC (Node Controller) information.
Manages the private network within the cluster.

Storage Controller.
Manages the virtual cluster storage and mount points

Node Controller.
Each hypervisor (cloud node) has a node controller.
Provides information on the available resources.

Cluster
colocated servers and networking infrastructure

Infrastructure as Code

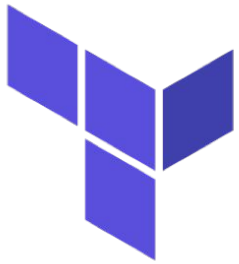
Provisioning and managing entire data centers—not just individual virtual machines
Distinction from configuration management (e.g., Ansible):

- Explicit creation and destruction of the infrastructure of a (virtual) data center
- Immutable infrastructure, instead of continuously modifying existing resources
- Typically declarative rather than imperative
- First introduced for the cloud in 2010 with AWS CloudFormation

Advantages:

- Versioning of the data center, enabling easy staging and rollbacks
- Accelerated delivery of infrastructure changes
- Consistency across environments
- Provides security and auditability of infrastructure in code
- Reusable and modular
- Enables collaboration through code management

Infrastructure as Code with Terraform



HashiCorp
Terraform

"Write, Plan, and Create
Infrastructure as Code"

Since 2014, open source by Hashicorp
Supports around 40 cloud providers
Also integrates with database systems, monitoring, and infrastructure software such as Kubernetes
Wide selection of plugins and reusable modules
Declarative configuration language
Commercial extensions available

Cloud-Architecture

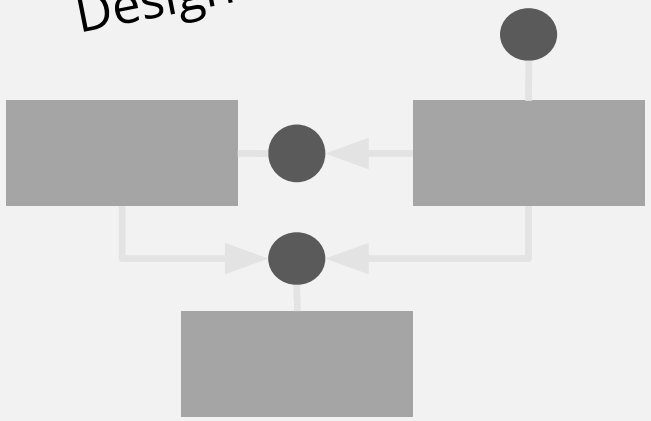
Cloud Native Application Development: Components All Along the Software Lifecycle

QAWARE

DESIGN → BUILD → RUN

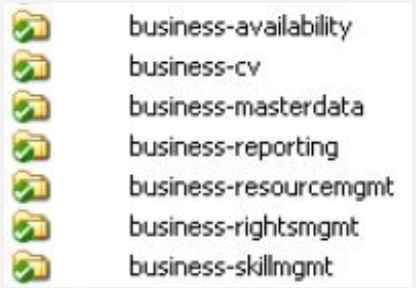
NEW!

Design Components



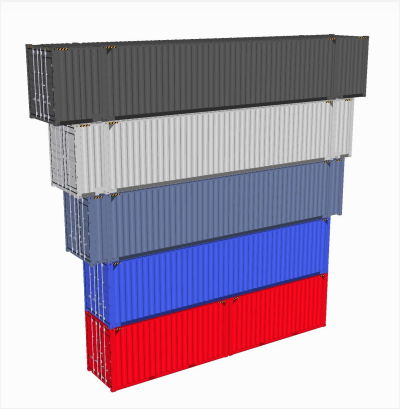
1:1

Dev Components



n:1

Ops Components



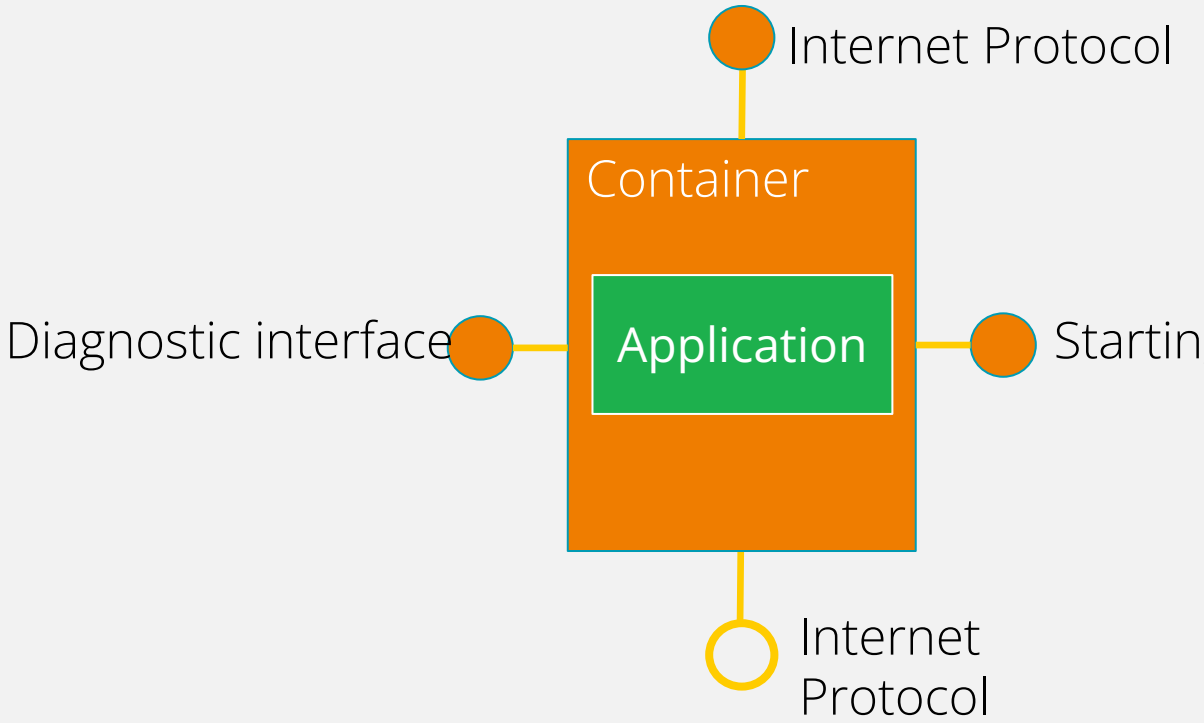
- Complexity unit
- Data integrity unit
- Coherent and cohesive features unit
- Decoupled unit

- Planning unit
- Team assignment unit
- Knowledge unit
- Development unit
- Integration unit

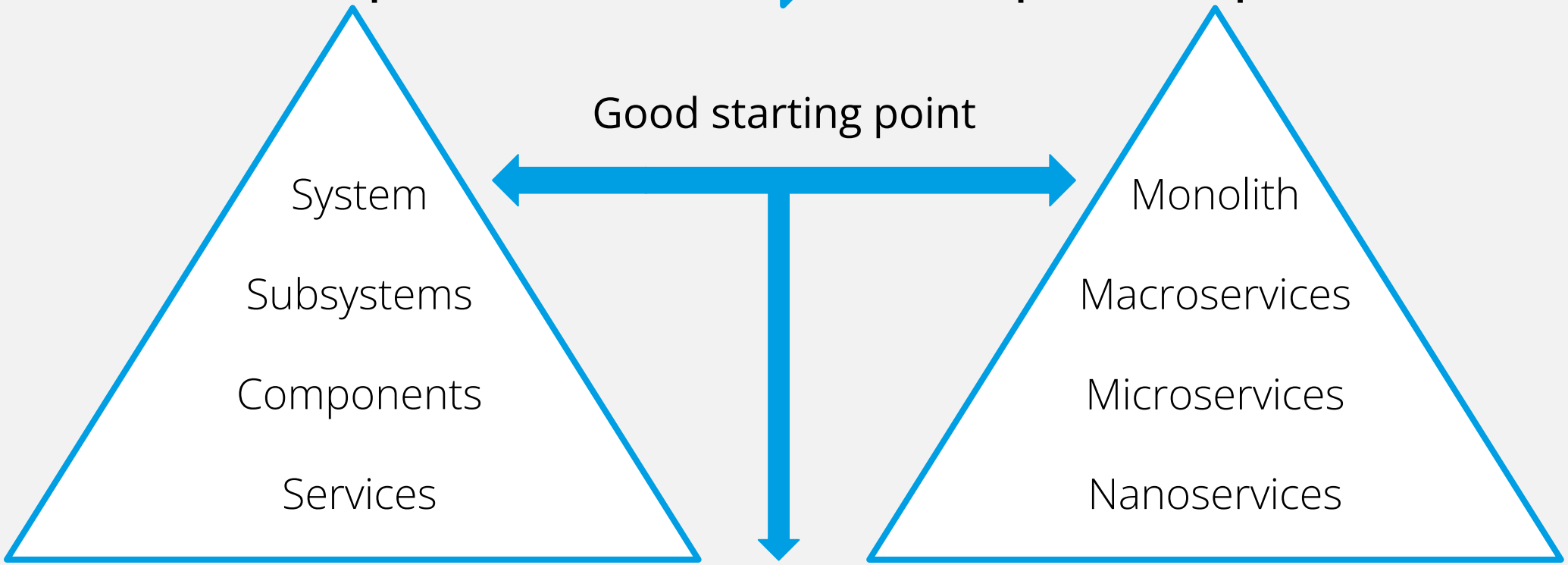
- Release unit
- Deployment unit
- Runtime unit (crash, slow-down, access)
- Scaling unit

The anatomy of an operation component

QA|WARE



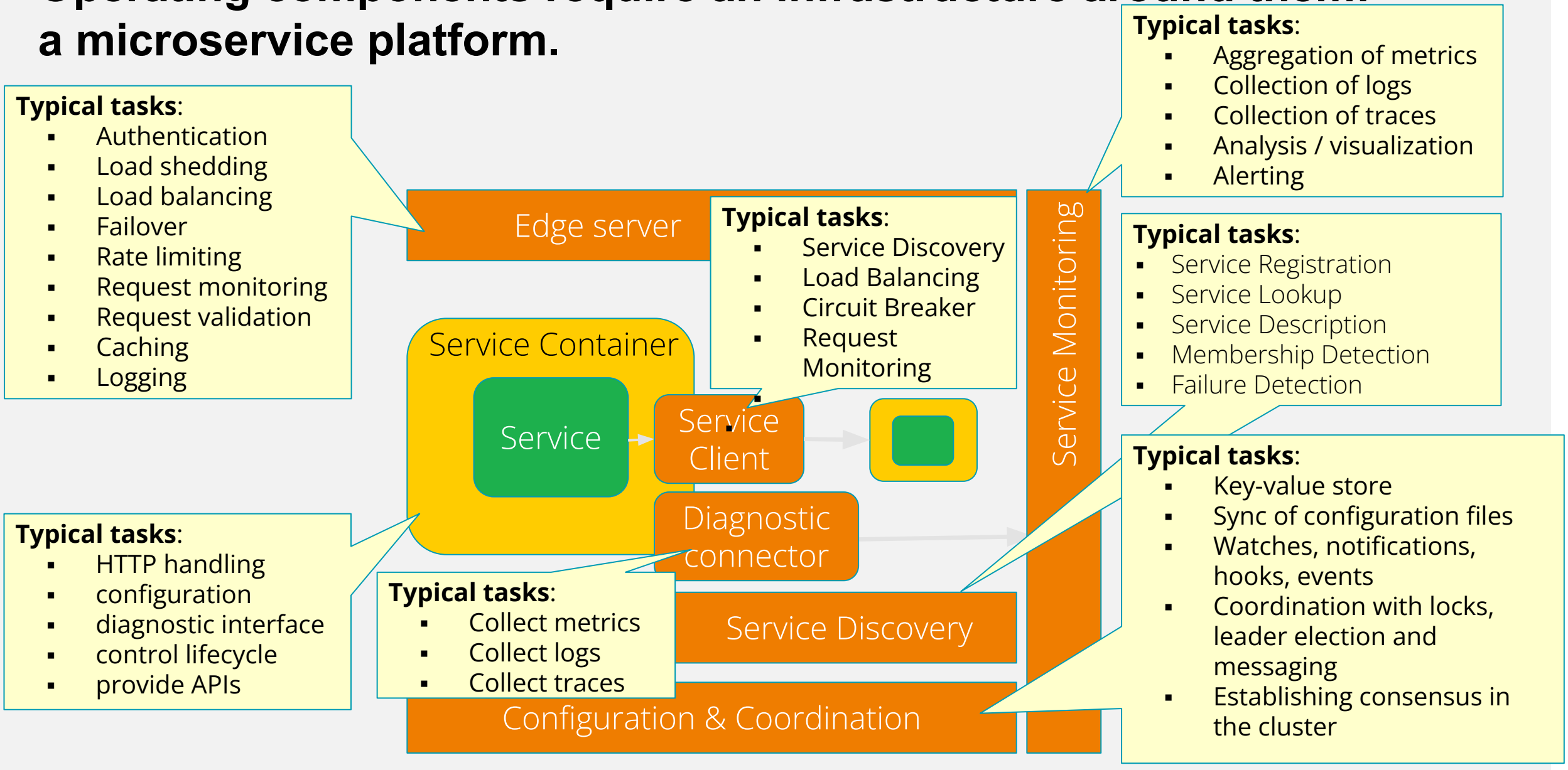
Dev Components ?:1 Ops Components



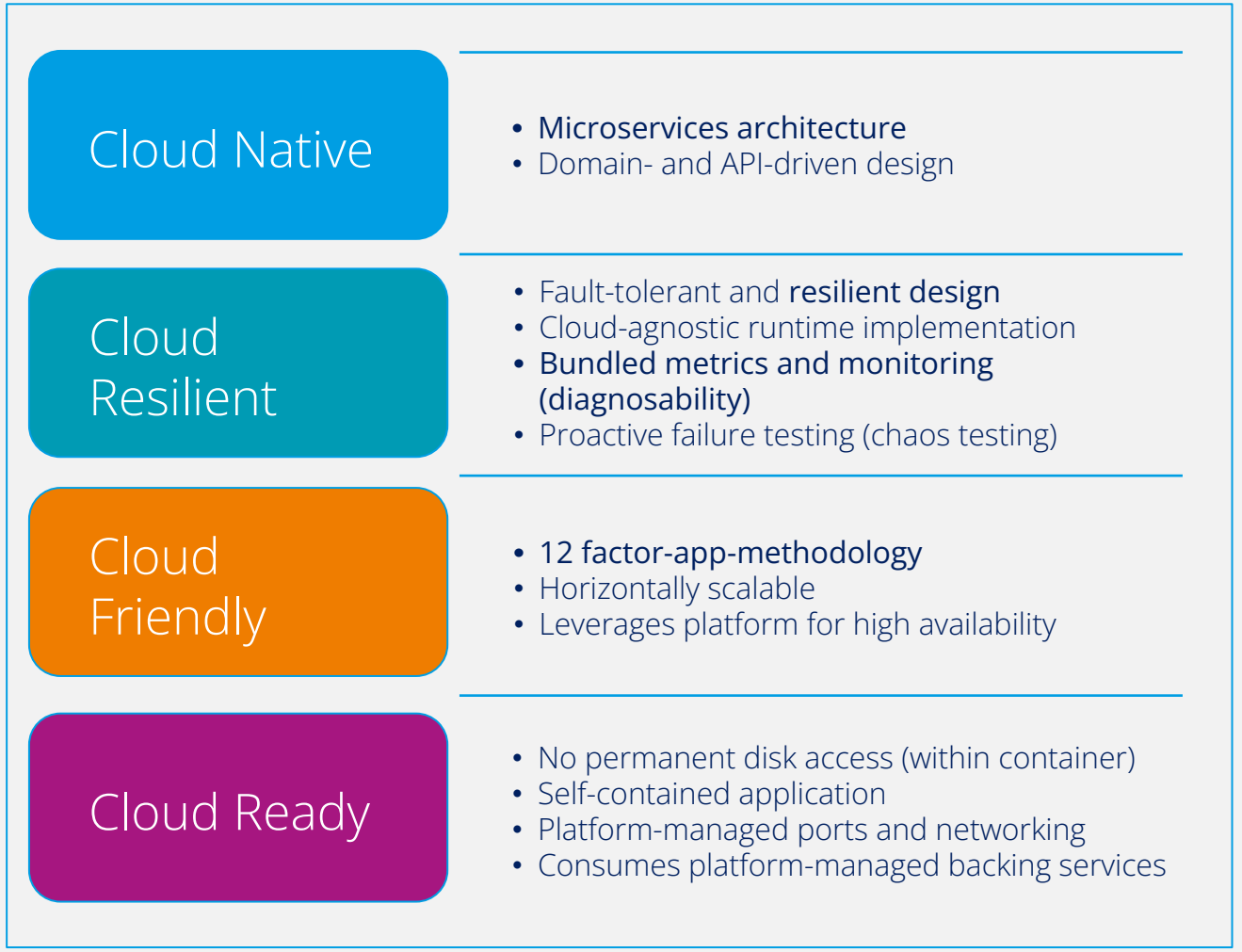
Decomposition Trade-Offs

+ More flexible to scale	− Distribution debt: Latency
+ Runtime isolation (crash, slow-down, ...)	− Increasing infrastructure complexity
+ Independent releases, deployments, teams	− Increasing troubleshooting complexity
+ Higher utilization possible	− Increasing integration complexity

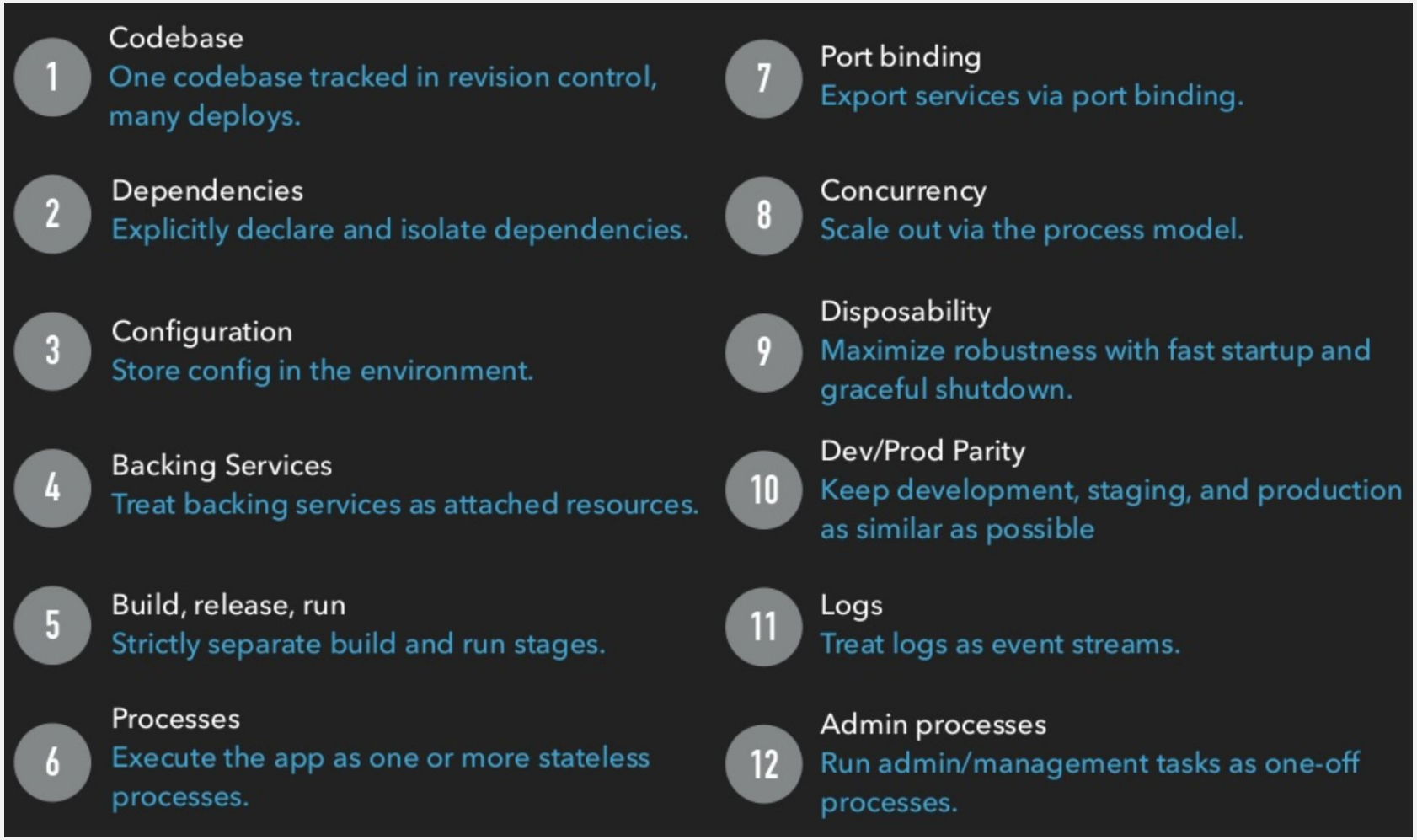
Operating components require an infrastructure around them: a microservice platform.



The Cloud Native Application Maturity Model

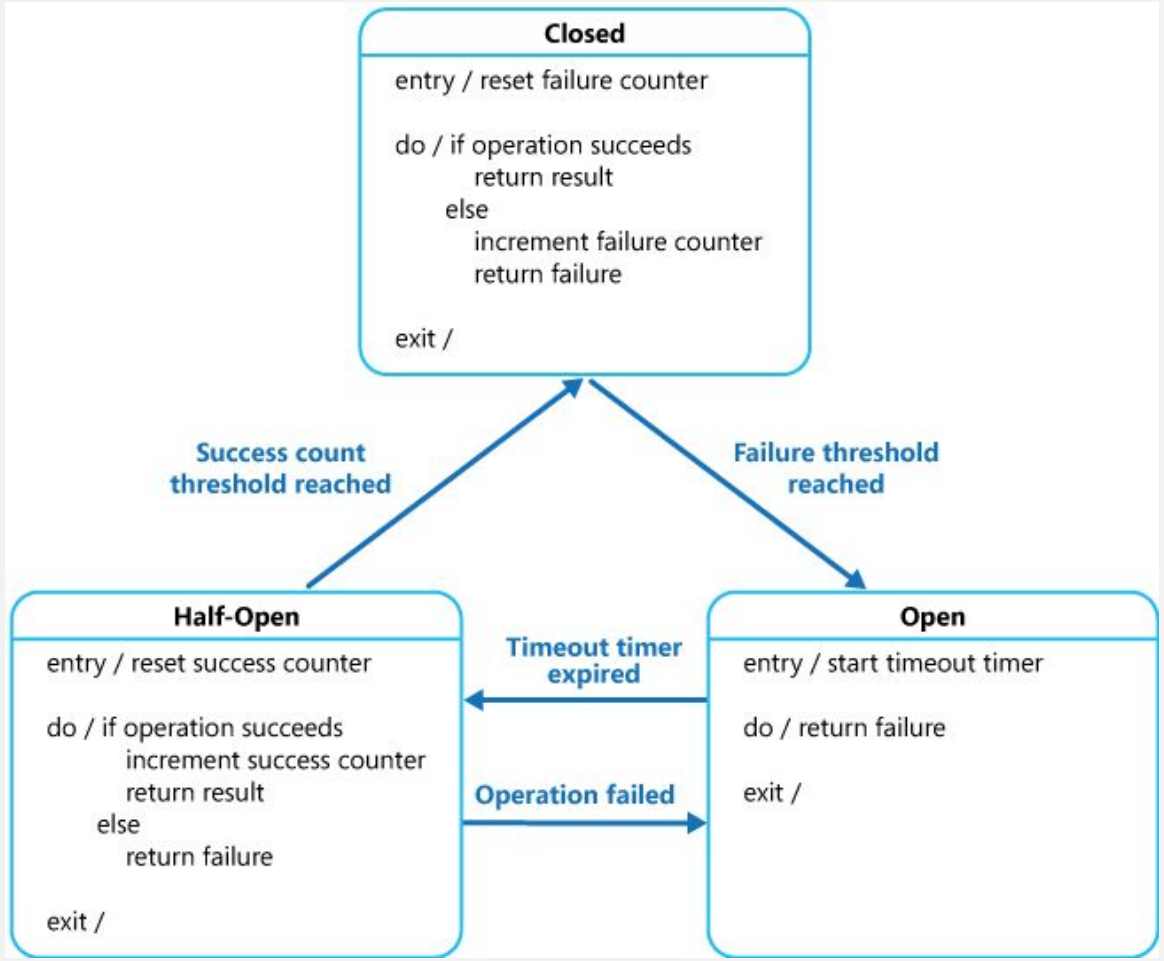


12 Factor App



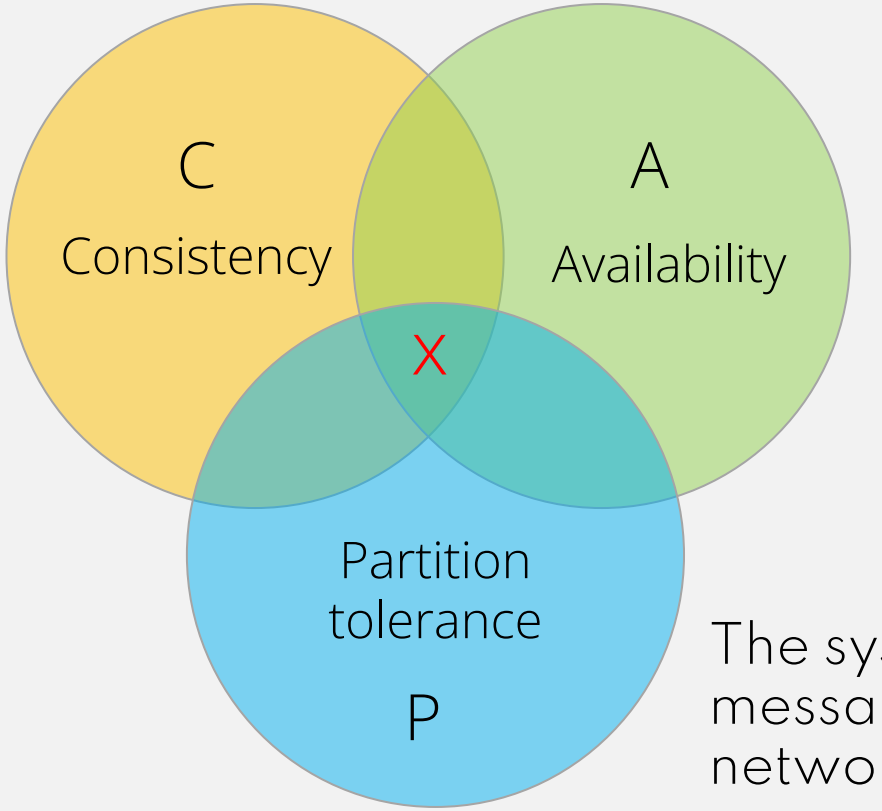
https://12factor.net/de/
https://www.slideshare.net/Alicanakku1/12-factor-apps

Resilience pattern: circuit breaker



Source: https://learn.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker
Further patterns: https://learn.microsoft.com/en-us/azure/well-architected/reliability/design-patterns

The CAP Theorem



All nodes see the same data at the same time. All copies are always the same.

The system will continue to operate even if individual nodes fail. Failures of nodes and channels do not prevent the surviving nodes from functioning.

The system also works in the case of lost messages. The system can handle the network being divided into nodes in several partitions that do not communicate with each other.

In the case of a network partition, you have to choose between consistency and availability; you can't have both.

Gossip protocols for high availability

Basis: A network of agents with their own state

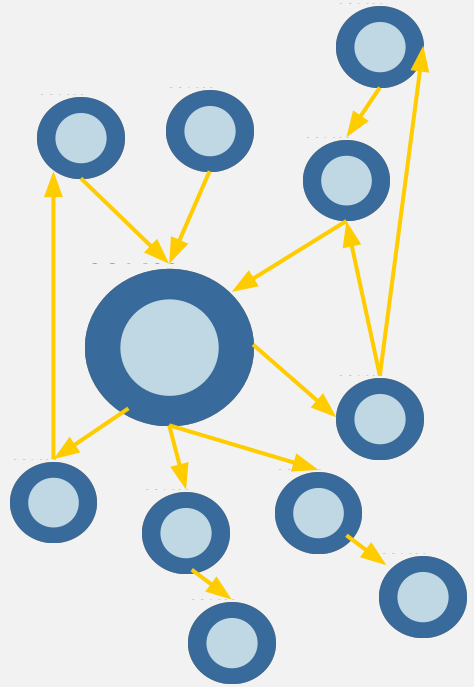
- Agents distribute a gossip stream
- Message: Source, content / state, timestamp
- Messages are sent periodically at fixed intervals
- to a certain number of other nodes (fanout)

Viral spread of the gossip stream

- Nodes that are in a group with me will definitely receive a message
- The top x% of nodes that send me messages will receive a message

Messages that are trusted are adopted into the local state if

- the same message has been heard from several pages
- the message comes from nodes that the agent trusts
- no more current message is available



Protocols for distributed consensus: consistent, but not highly available, in contrast to gossip protocols

Basis: network of agents

Principle: it is sufficient if the state is consistent on a simple majority of the nodes and the remaining nodes recognize their inconsistency.

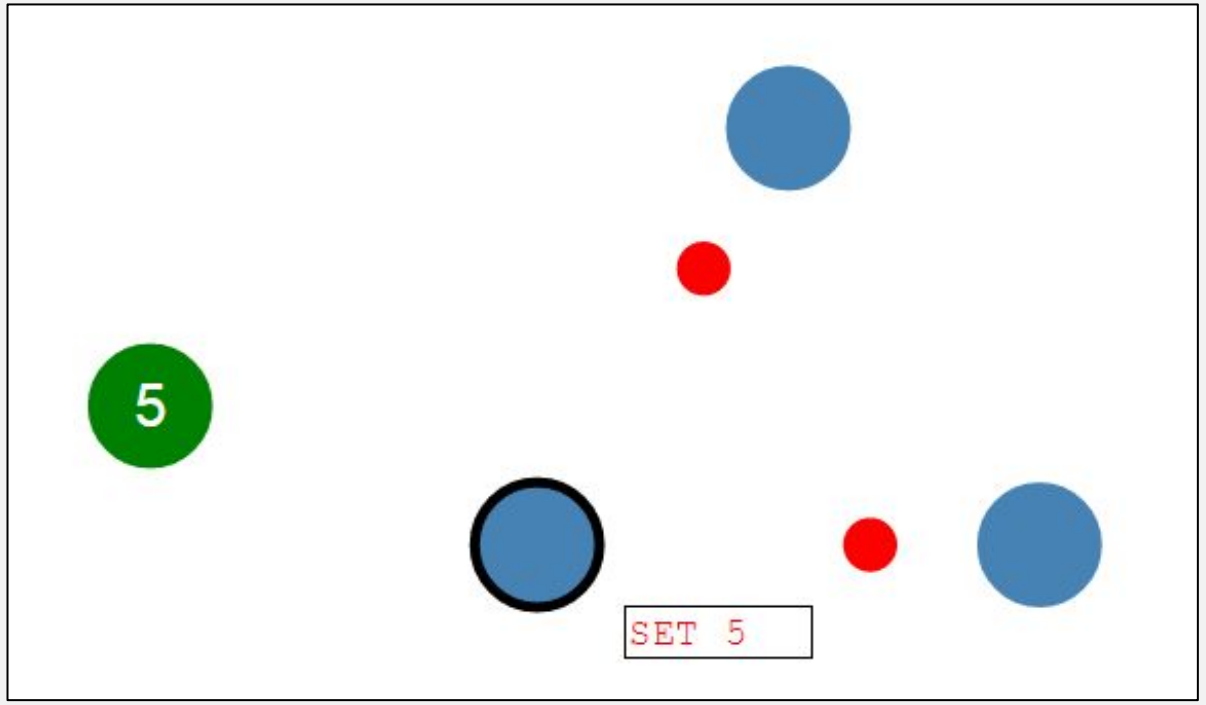
Procedure:

- the network agrees on a leader agent by a simple majority – initially and if the leader agent is not accessible. A partition in the minority cannot elect a leader agent.
- All changes are routed through the leader agent. The leader agent periodically distributes change messages to all other agents via multicast.
- If a simple majority of agents acknowledge the change message, the change is activated in the leader and (via message) also in the agents that have acknowledged. Otherwise, the state is assumed to be inconsistent.

Specific consensus protocols: **Raft**, **Paxos**

The Raft Consensus Protocol

Ongaro, Diego; Ousterhout, John (2013).
"In Search of an Understandable Consensus Algorithm".



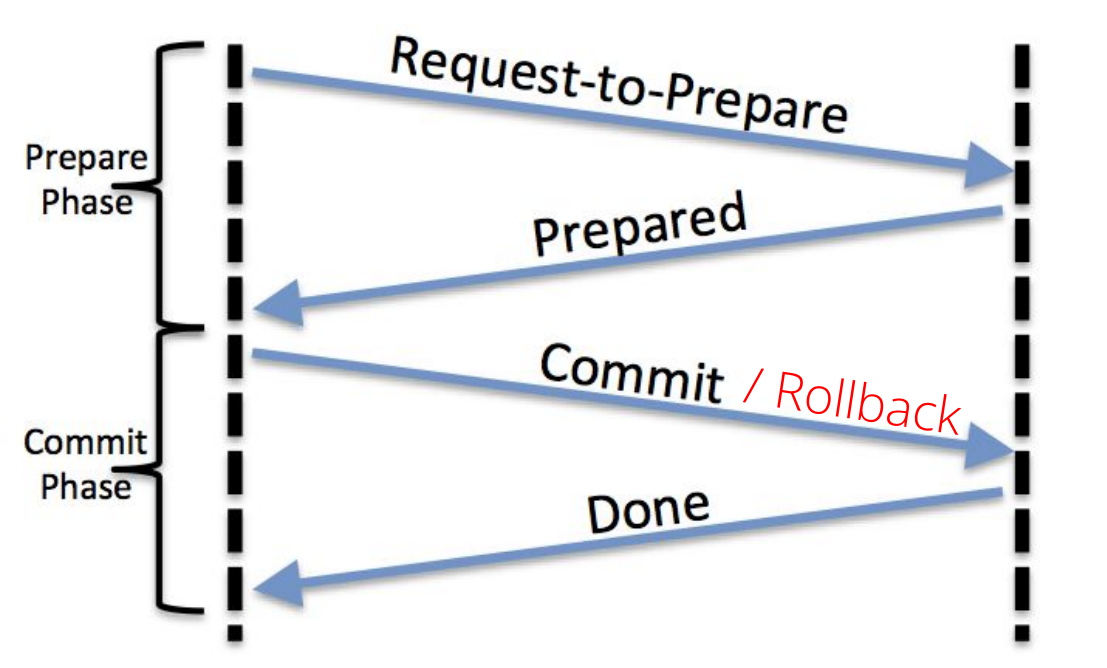
http://thesecretlivesofdata.com/raft

https://raft.github.io/

If strict consistency across all nodes is necessary, the 2-phase commit protocol (2PC) remains.

QA|WARE

A transaction coordinator distributes the changes and activates them only when all have been approved. Otherwise, the changes are undone.



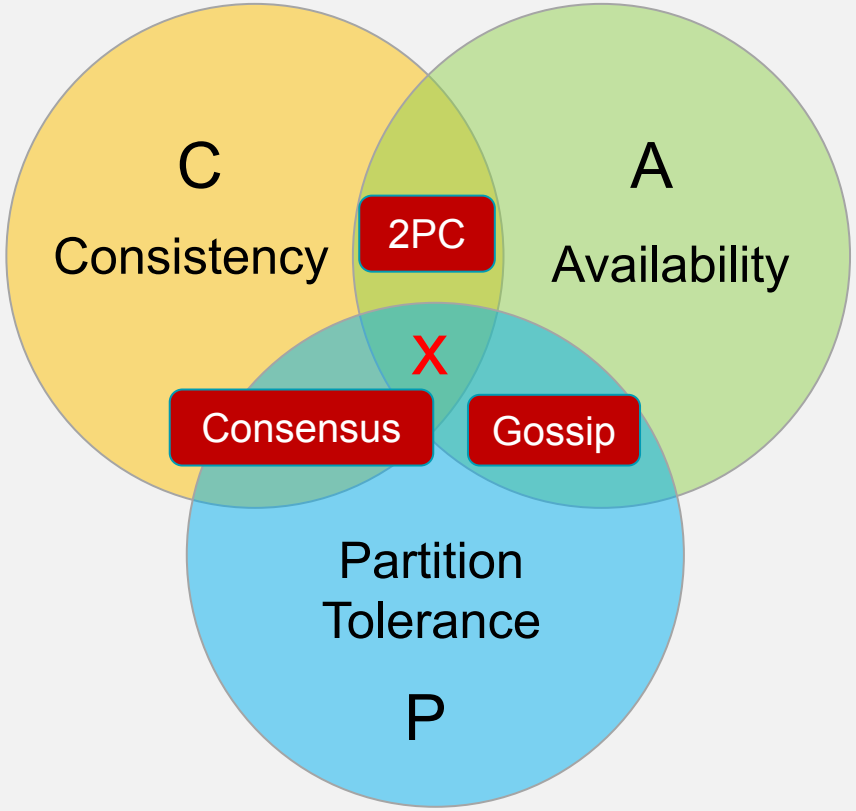
Advantage:

- All nodes are consistent with each other.

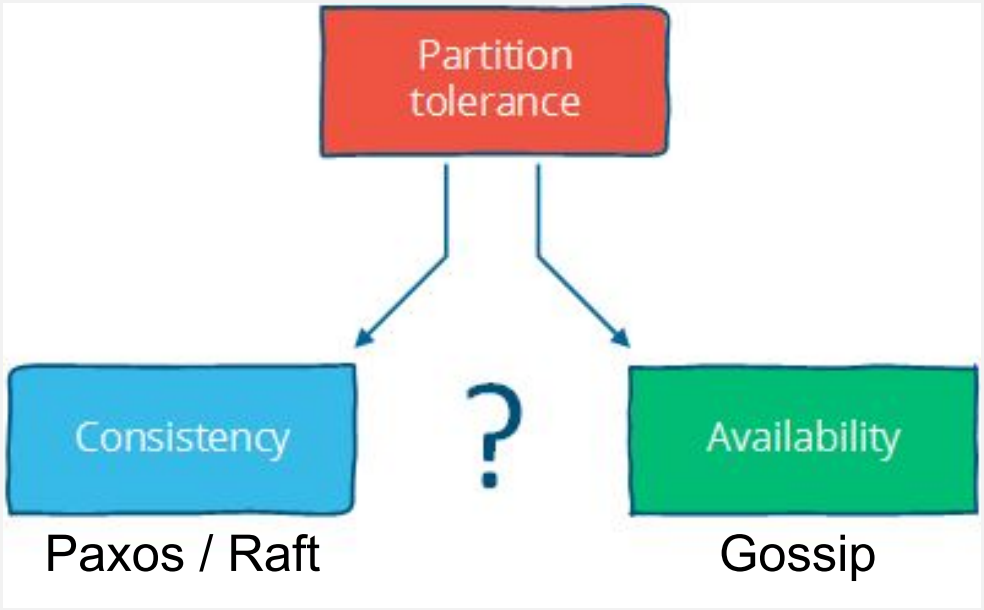
Disadvantages:

- Time-consuming, since all nodes must always agree.
- The system no longer works once the network is partitioned.

The presented protocols and the CAP theorem

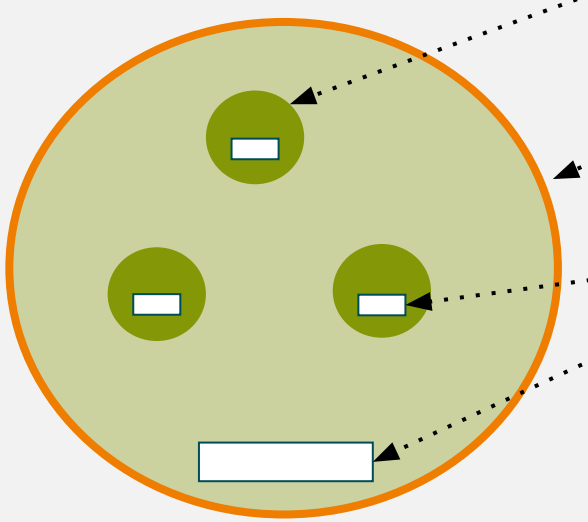


In the cloud, partitions have to be assumed.
This makes the decision binary:
Between consistency and availability.



Orchestration

Terminology



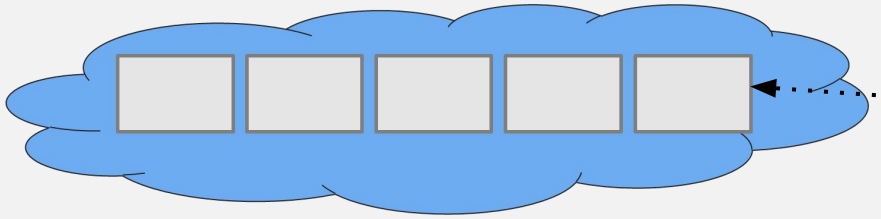
Task: Atomic calculation including execution rule.

Job: Set of tasks with common execution goal. The set of tasks is usually represented as a DAG with tasks as nodes and execution dependencies as edges.

Properties: Properties of the tasks and jobs that are relevant for execution, such as:

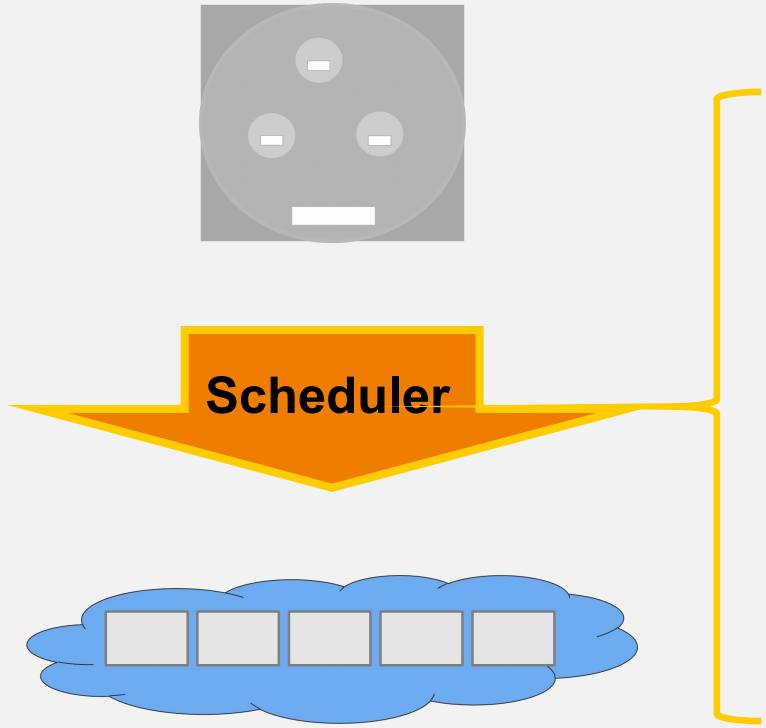
- Task: execution time, priority, resource consumption
- Job: task dependencies, execution time

Scheduler: Execution of tasks on the available resources, taking into account the properties and given **scheduling objectives** (e.g. fairness, throughput, resource utilization). A scheduler can be **preemptive**, i.e. it can interrupt the execution of tasks and restart them.



Resources: Clusters of computers with CPU, RAM, HDD and network resources. A computer makes its resources temporarily available for the execution of one or more tasks (slot). The parallel execution of tasks is isolated from each other.

Duties of a cluster scheduler:



Cluster Awareness: Know the currently available resources in the cluster (nodes including available CPUs, available RAM and hard disk space, and network bandwidth). Also respond to elasticity.

Job Allocation: To determine and allocate the appropriate amount of resources for a specific period of time to perform a service.

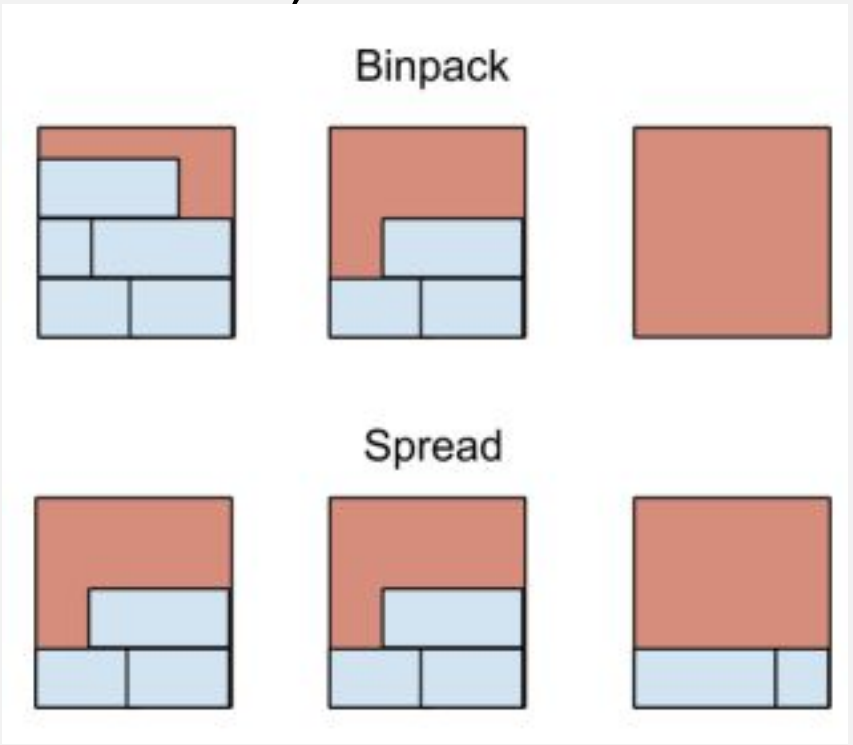
Job Execution: Reliably perform a service while isolating and monitoring it.

Simple scheduling algorithms

- Optimize the scheduling of tasks often in exactly one dimension (e.g. CPU utilization) or a few dimensions (CPU utilization and RAM).

Popular algorithms:

- Binpack (Fit First)
- Spread (Round Robin)



http://www.binpacking.4fan.cz
http://container-solutions.com/using-binpack-with-docker-swarm

Cluster Orchestration

Objective: Run an application that is distributed across multiple operational components (containers) on multiple nodes.

Introduces abstractions for running applications with their services in a large cluster.

Orchestration is not a static, one-time activity like provisioning, but a dynamic, continuous activity.

Orchestration has the potential to automate all standard operational procedures of an application.

Blueprint of the application that describes the desired operating state of the application: operating components (containers), their operating requirements, and the offered and required interfaces.

Cluster Orchestrator

- **Control activities in the cluster:**
- **Starting containers on nodes (scheduler)**
- **Linking containers**
- **...**

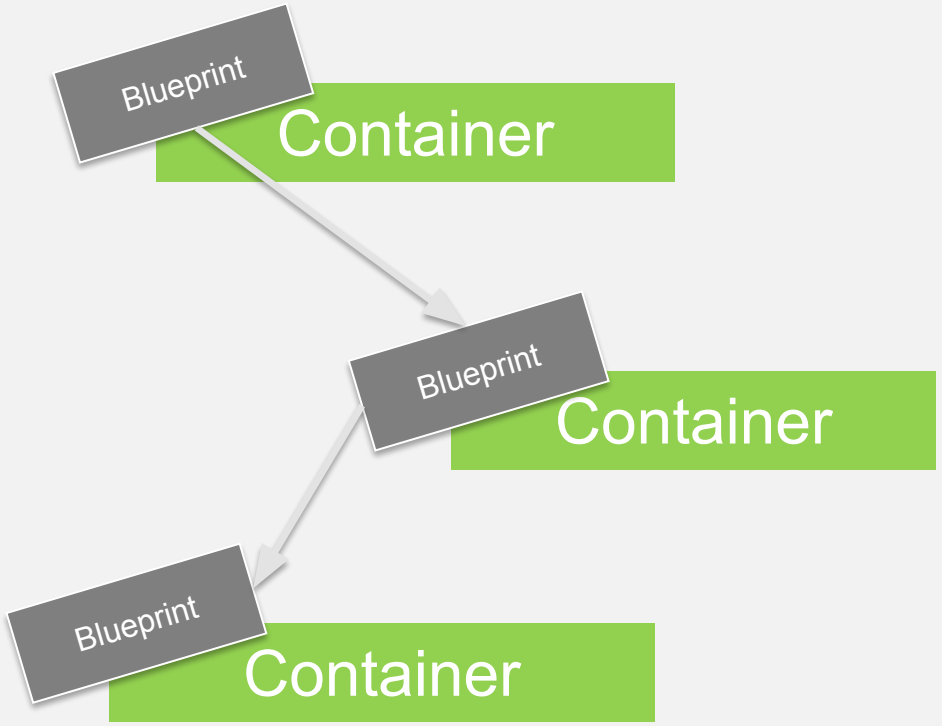
A cluster orchestrator automates many operational tasks for applications on a cluster. (1 / 2):

- Scheduling of containers with application-specific constraints (e.g. deployment and start orders, grouping, ...)
- Establishing the necessary network connections between containers.
- Providing persistent storage for stateful containers.
- (Auto-)scaling of containers.
- Rescheduling of containers in the event of an error (auto-healing) or to optimize performance.
- Container logistics: management and provision of containers.

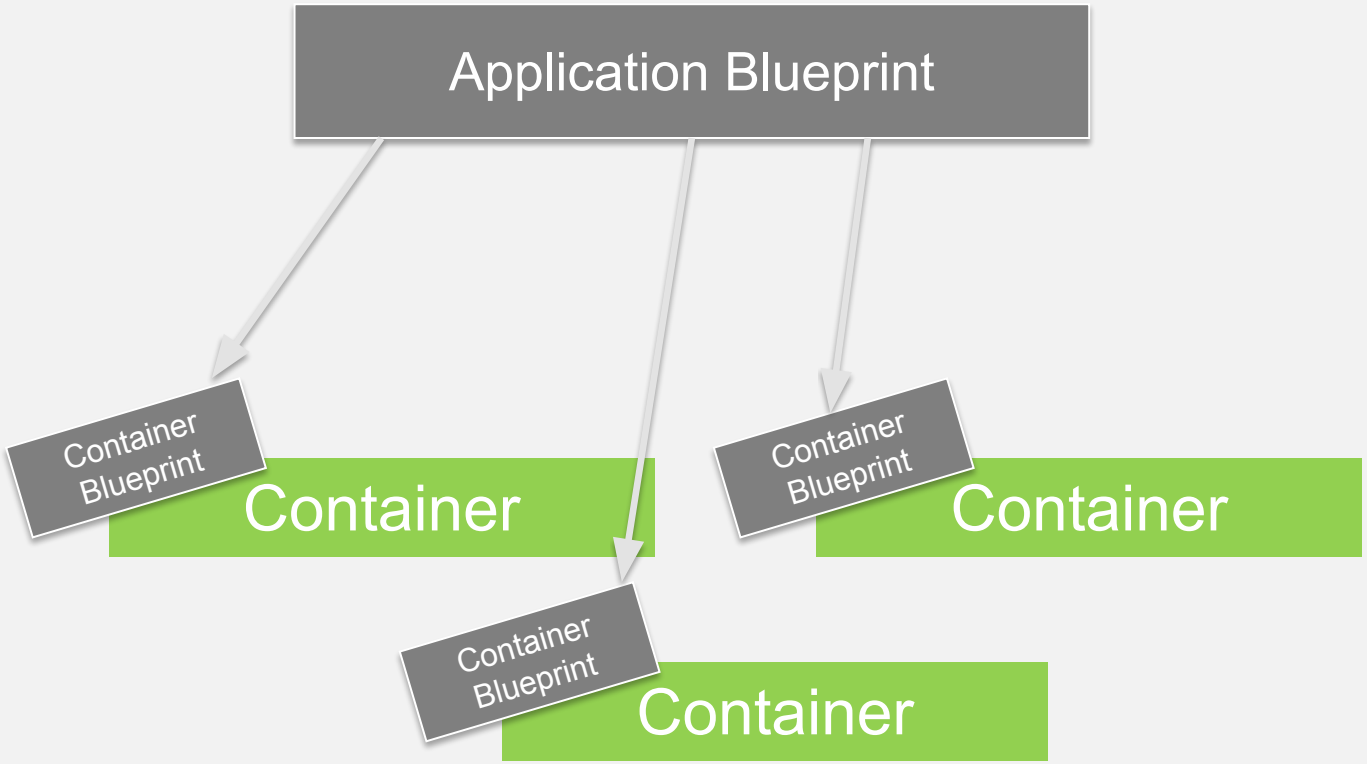
A cluster orchestrator automates many operational tasks for applications on a cluster. (2 / 2):

- Package management: administration and provision of applications.
- Provision of administration interfaces (remote API, command line).
- Service management: service discovery, naming, load balancing.
- Automation for rollout workflows such as Canary Rollout.
- Monitoring and diagnosis of containers and services.

1-level vs. 2-level orchestration



1-level Orchestration
(Container Graph)



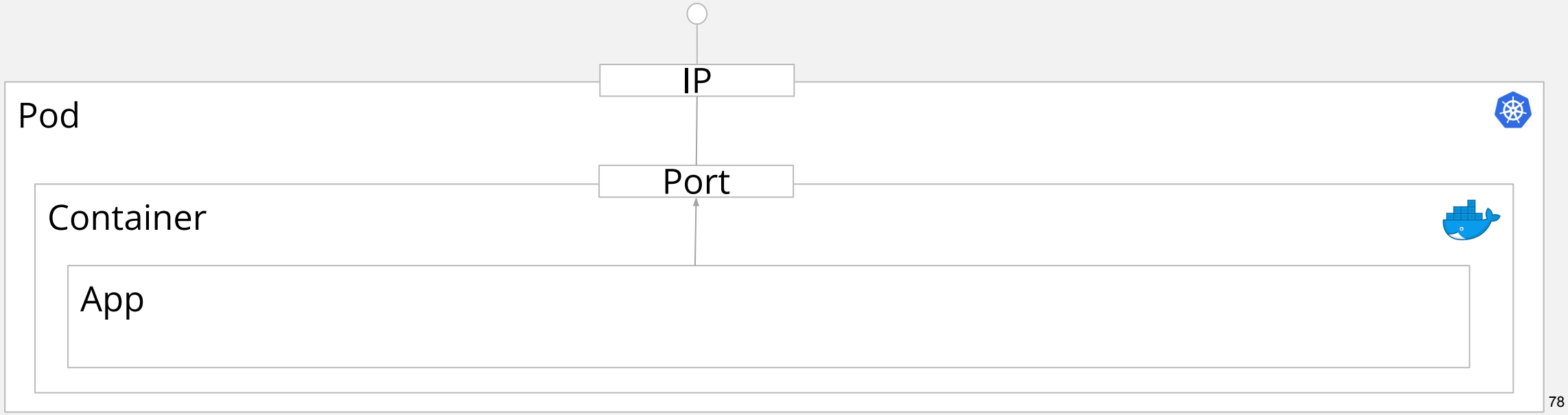
2-level Orchestration
(Container repository with a centralized building instruction)

Important Kubernetes Concepts

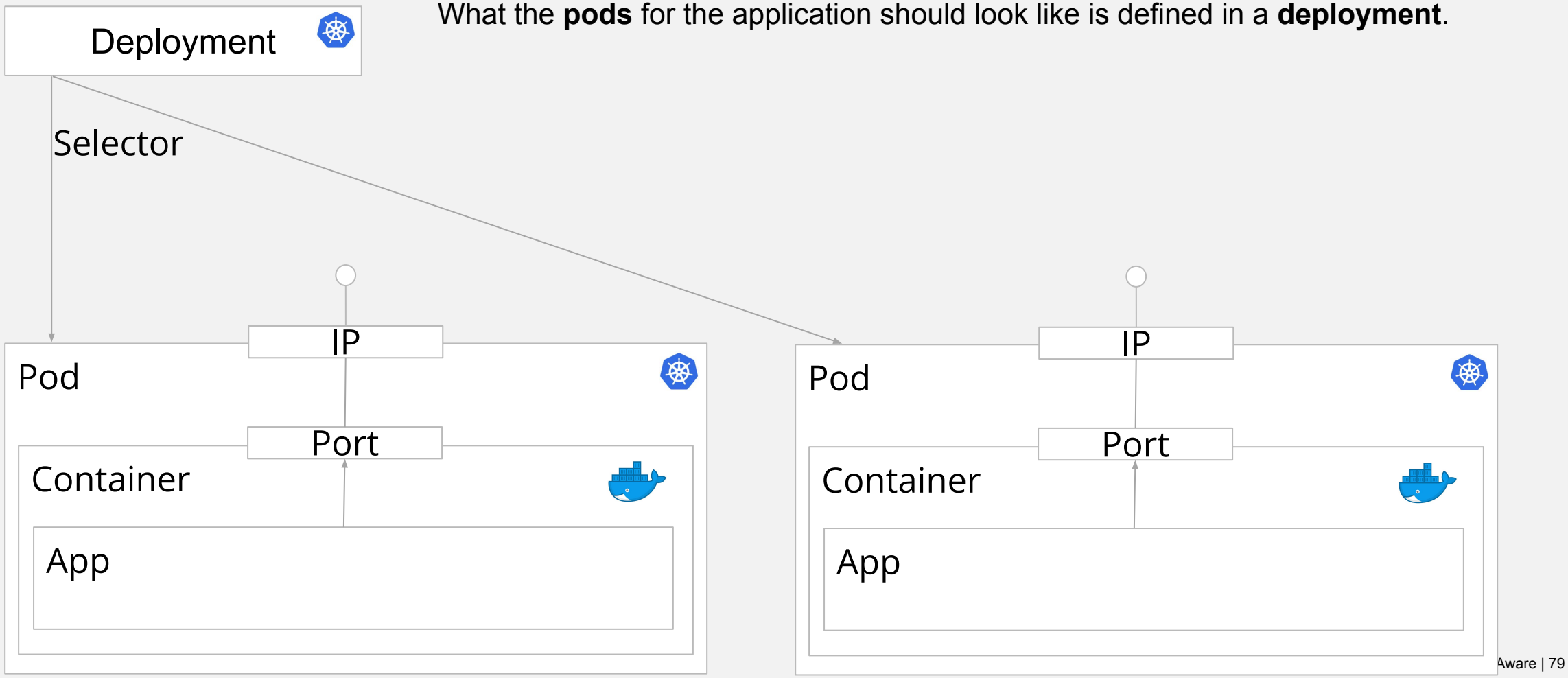
The basic building block is your **application**.

The **application** is in a **container** (see lecture “Virtualization”). **Containers** open **ports** to the outside.

In Kubernetes, **containers** are grouped into **pods**. **Pods** have one **IP address** facing outwards.



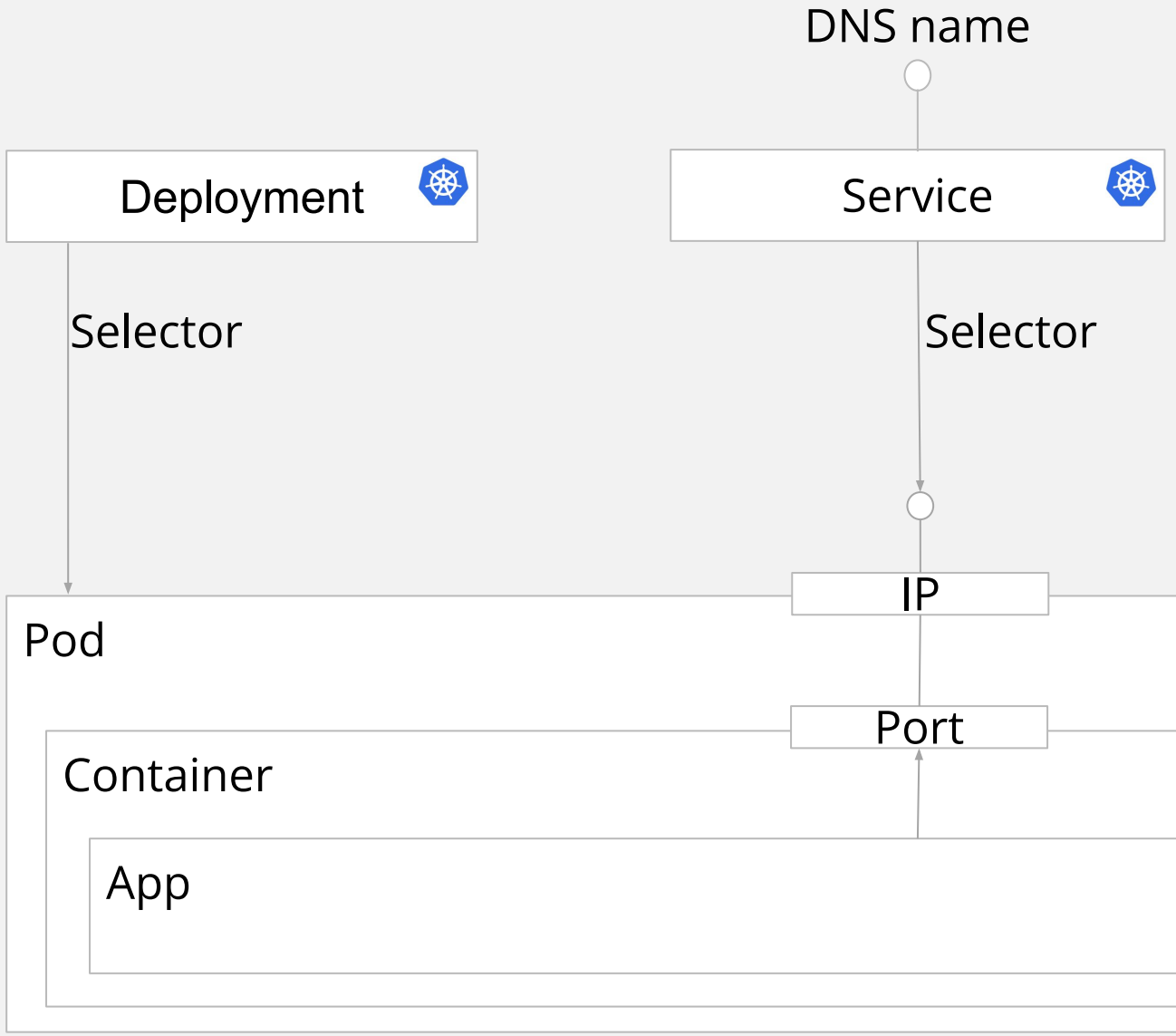
Important Kubernetes Concepts



Deployment: Definition


```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-service
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: helloservice
    spec:
      containers:
      - name: hello-service
        image: "hitchhikersguide/zwitscher-service:1.0.1"
        ports:
        - containerPort: 8000
        env:
        - name:
          value: zwitscher-consul
```

Important Kubernetes Concepts



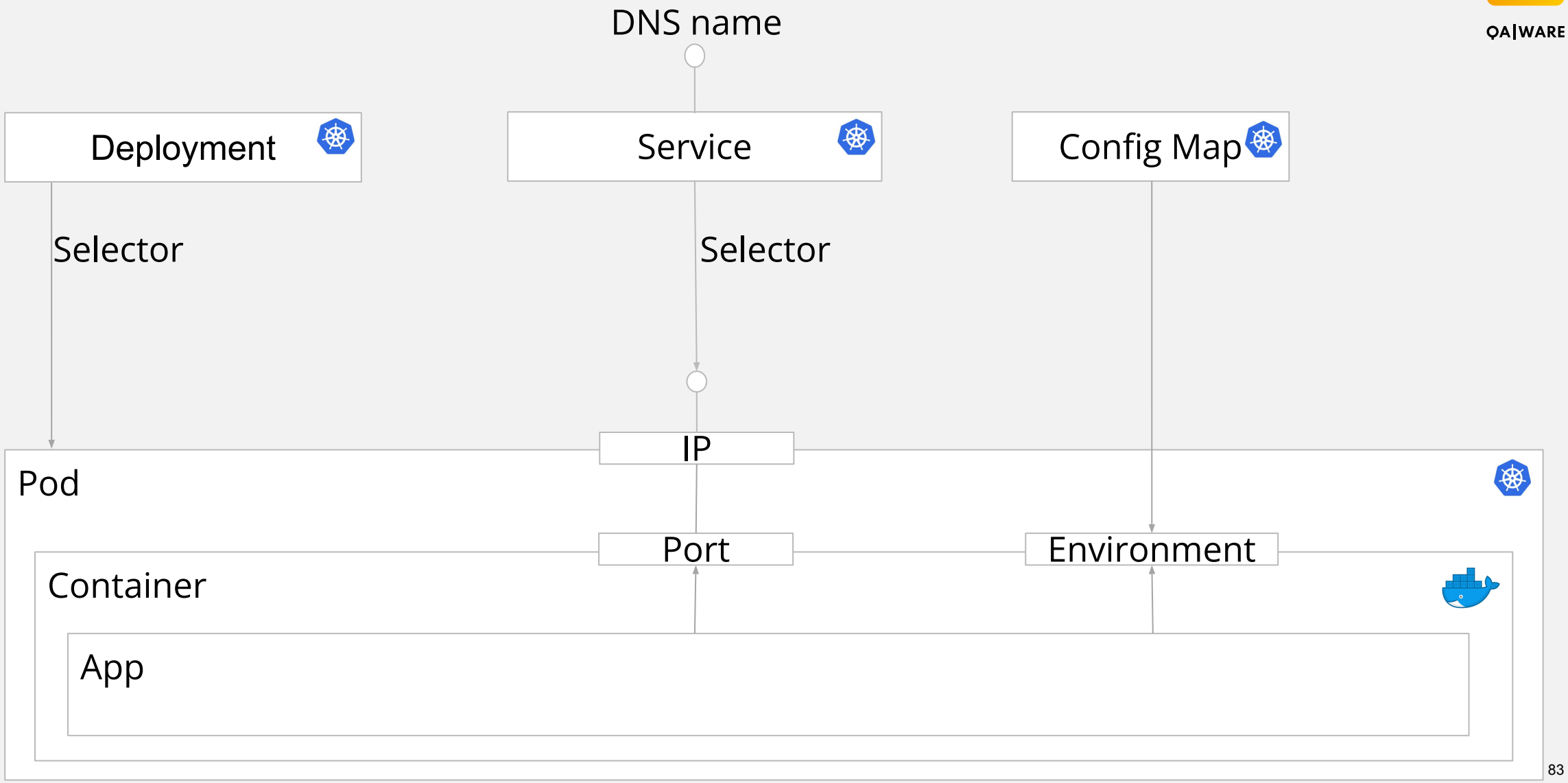
Pods get a random **hostname**. How can I access it in the **cluster** now?

A **service** assigns a **DNS name** for a set of **pods**.

Service: Definition


```yaml
apiVersion: v1
kind: Service
metadata:
  name: hello-service
  labels:
    app: helloservice
spec:
  # use NodePort here to be able to access the port on each node
  # use LoadBalancer for external load-balanced IP if supported
  type: NodePort
  ports:
  - port: 8080
  selector:
    app: helloservice
```

Important Kubernetes Concepts



Configuration: Config Maps (1)

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: game-demo
data:
  # property-like keys; each key maps to a simple value
  player_initial_lives: "3"
  ui_properties_file_name: "user-interface.properties"
```

Configuration: Config Maps (2)

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: configmap-demo-pod
spec:
  containers:
    - name: demo
      image: alpine
      command: ["sleep", "3600"]
      env:
        # Define the environment variable
        - name: PLAYER_INITIAL_LIVES # Notice that the case is different here
                                     # from the key name in the ConfigMap.
          valueFrom:
            configMapKeyRef:
              name: game-demo           # The ConfigMap this value comes from.
              key: player_initial_lives # The key to fetch.
        - name: UI_PROPERTIES_FILE_NAME
          valueFrom:
            configMapKeyRef:
              name: game-demo
              key: ui_properties_file_name
```

Resource Constraints


```yaml
resources:
  # Define resources to help K8S scheduler

  # CPU is specified in units of cores
  # Memory is specified in units of bytes

  # required resources for a Pod to be started
  requests:
    memory: "128M"
    cpu: "0.25"

  # the Pod will be throttled (CPU) or restarted (memory)
  # if limits are exceeded
  limits:
    memory: "192M"
    cpu: "0.5"
```

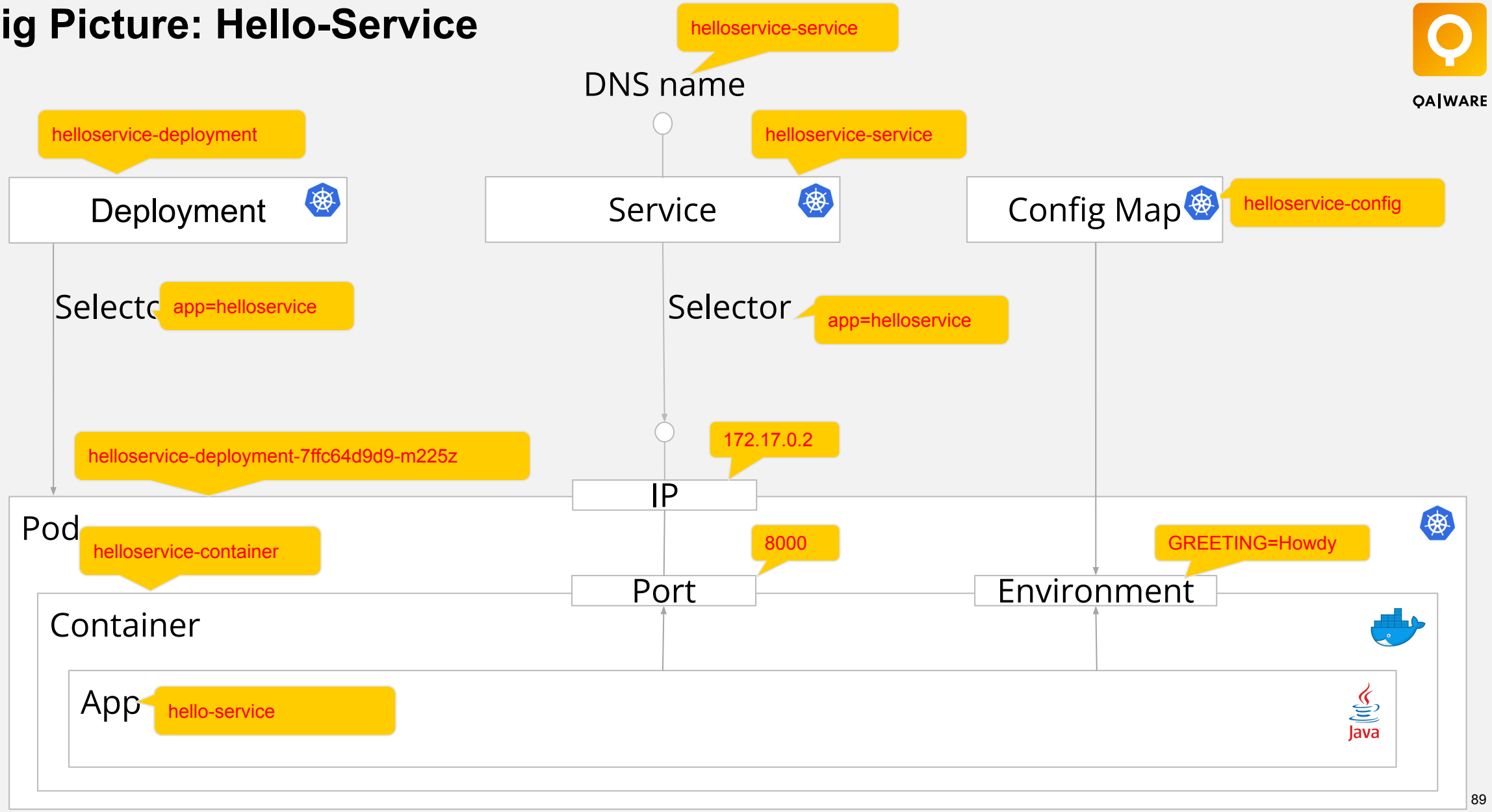
Liveness and Readiness Probes


```
# container will receive requests if probe succeeds
readinessProbe:
  httpGet:
    path: /admin/info
    port: 8080
  initialDelaySeconds: 30
  timeoutSeconds: 5

# container will be killed if probe fails
livenessProbe:
  httpGet:
    path: /admin/health
    port: 8080
  initialDelaySeconds: 90
  timeoutSeconds: 10
```

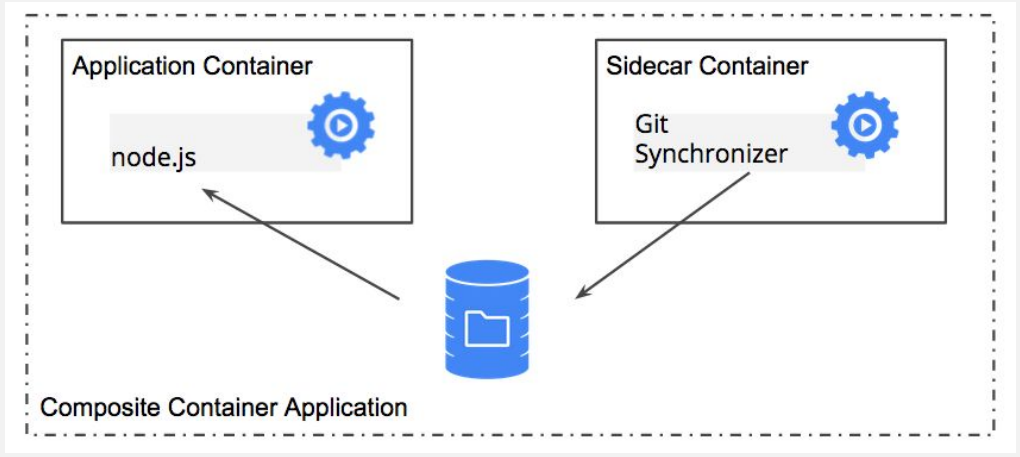
Startup Probe


```yaml
# other probes will start after startup probe succeeds
startupProbe:
  httpGet:
    path: /admin/health
    port: liveness-port
  failureThreshold: 30
  periodSeconds: 10
```

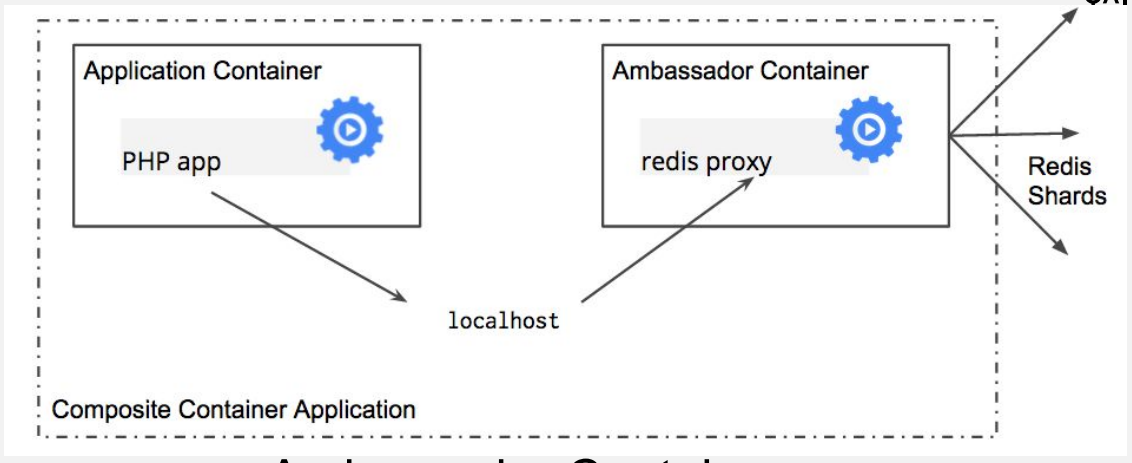
Big Picture: Hello-Service



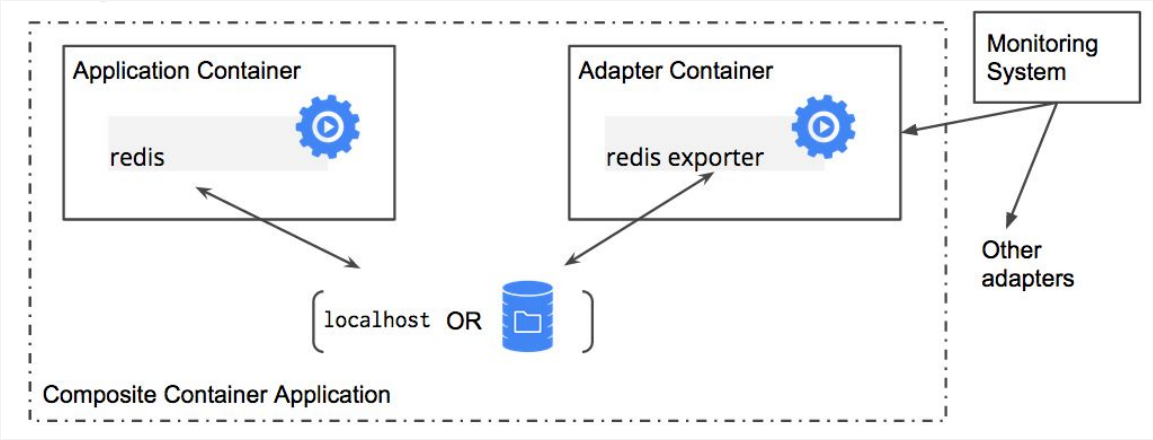
Orchestration patterns
Separation of Concerns with modular containers



Sidecar Container



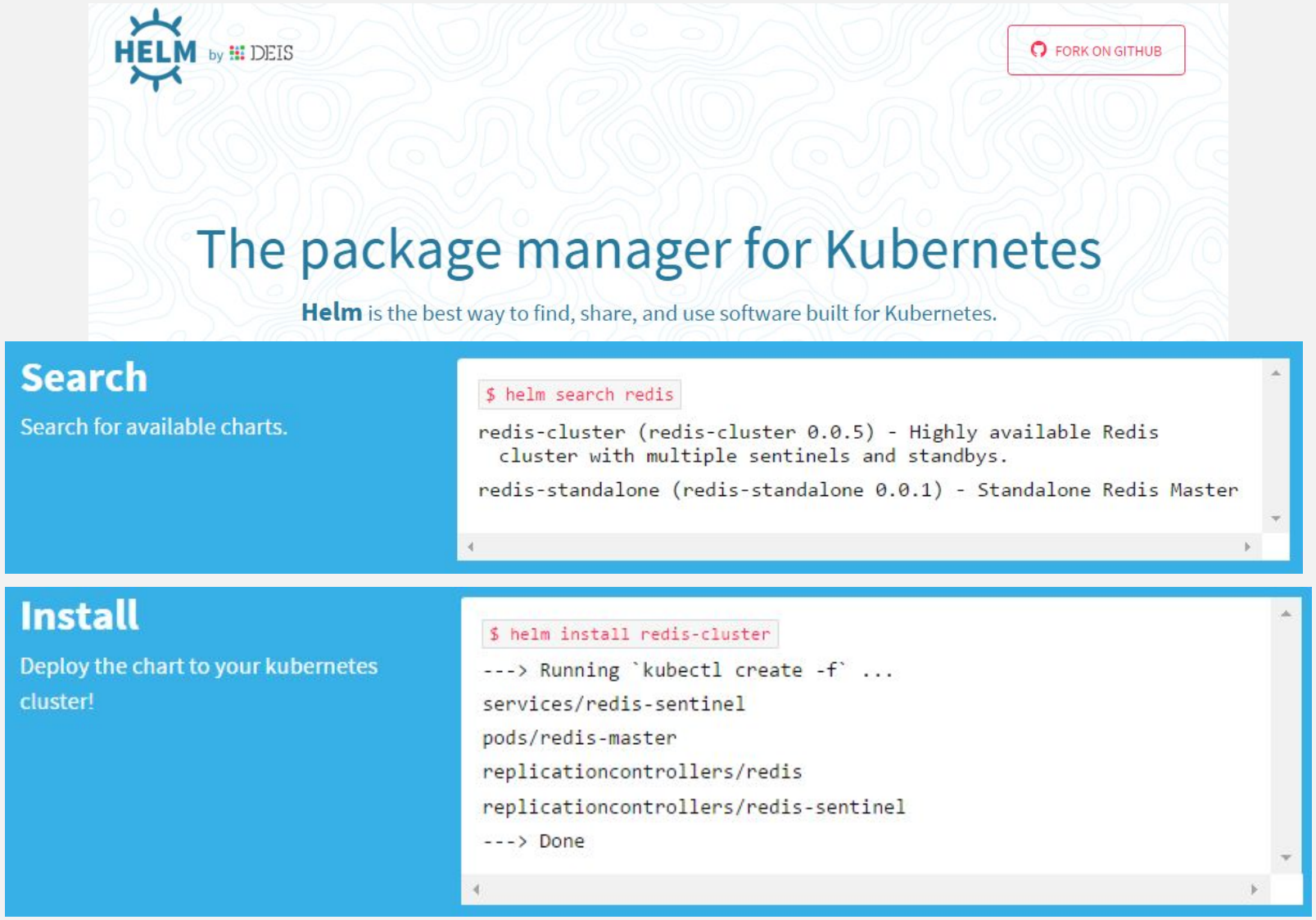
Ambassador Container



Adapter Container

Quelle: http://blog.kubernetes.io/2015/06/the-distributed-system-toolkit-patterns.html

Helm: Application package management for Kubernetes.



Source: https://github.com/helm/helm

Service Meshes

What is a Service Mesh?

*In software architecture, a **service mesh** is a dedicated infrastructure layer for facilitating service-to-service communications between services or microservices using a proxy.*

A dedicated communication layer can provide numerous benefits, such as providing observability into communications, providing secure connections or automating retries and backoff for failed requests.

A Service Mesh offers…

Traffic Management

- Load Balancing
- Service Discovery
- Routing
- Retries
- Timeouts
- Circuit Breaker

Security

- Encryption
- Authentication and Authorization
- Rate Limiting

Observability

- Metrics
- Tracing
- (Logging)

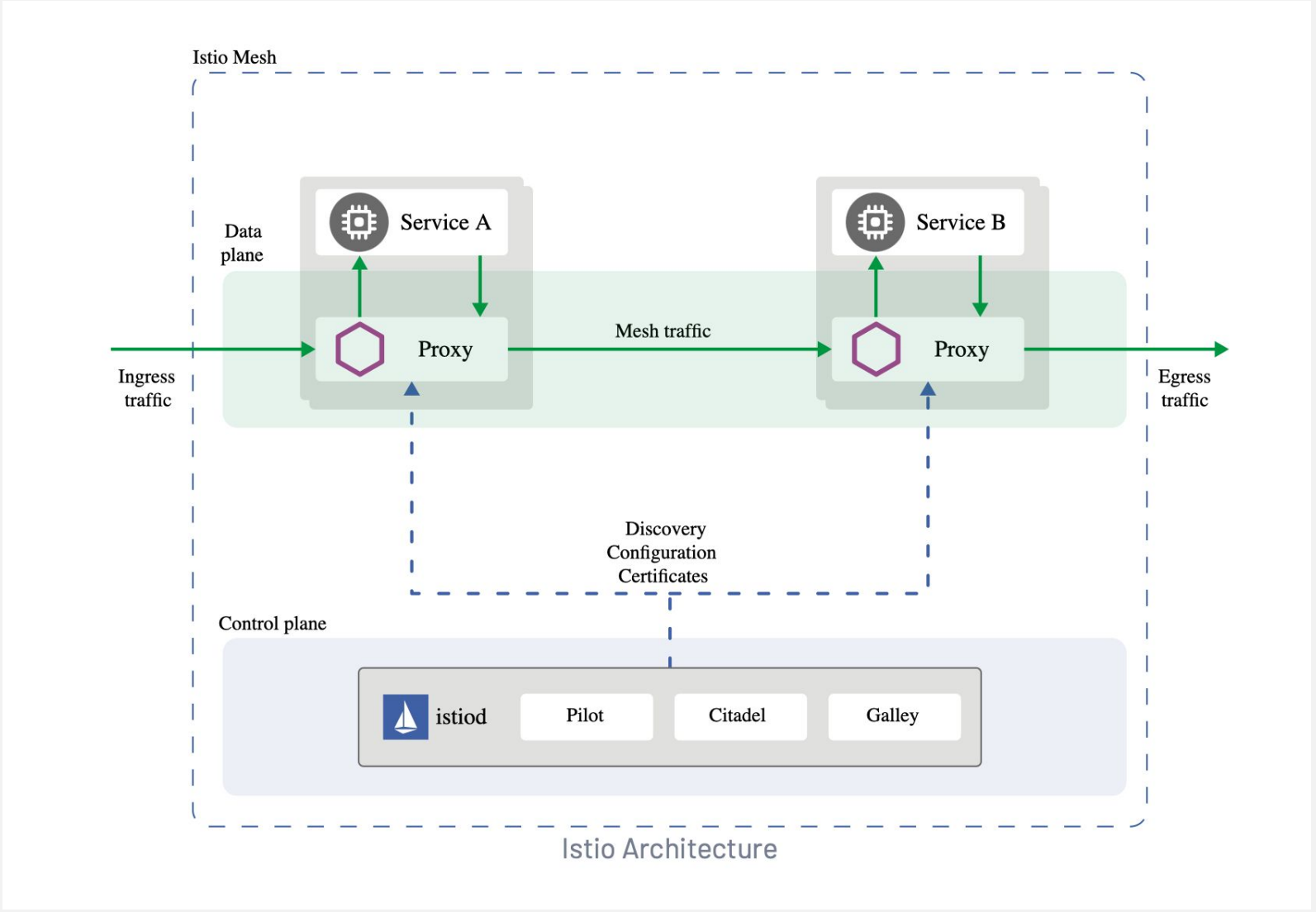
High Level architecture of Service Meshes

Control Plane:

- contains management components, acts as the “brain” of the Service Meshes.
- typical components include Service Discovery & Traffic rules, Configuration Stores & APIs, Identity & Credential Management

Data Plane:

- contains the actual workloads
- includes a proxy component that implements the service mesh’s features



Istio Architecture

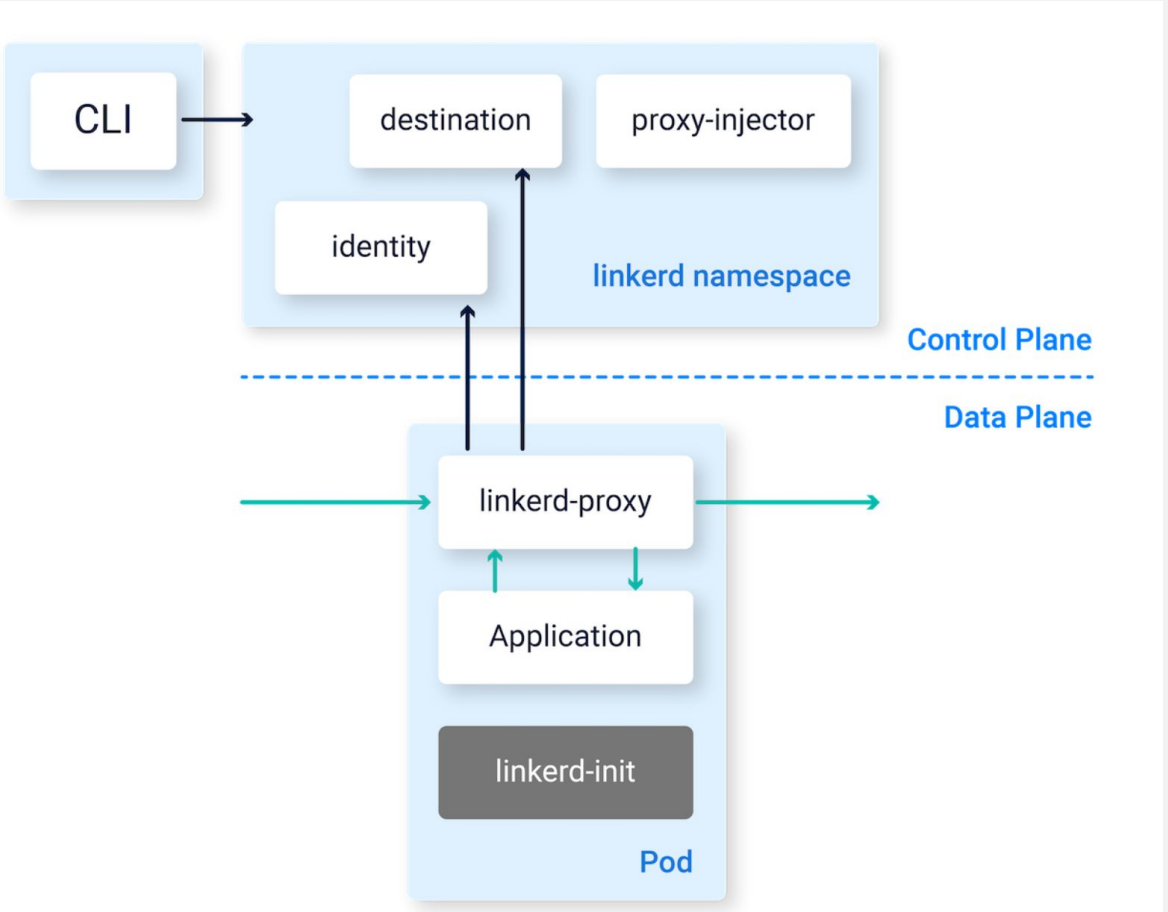
Data plane architectures - Sidecar

Sidecar-Pattern:

Each application pod contains an additional container, the “sidecar”.

Sidecars in Linkerd Service Mesh:

- Each pod in the mesh automatically is injected with a sidecar proxy
- Inbound and outbound traffic is routed through the sidecar proxy via IP tables rules. The rules are set either through the *linkerd-init* startup container or implemented via a CNI plugin.
- The sidecar proxy implements the mesh functionality.



Linkerd’s architecture

Data plane architectures - Sidecar

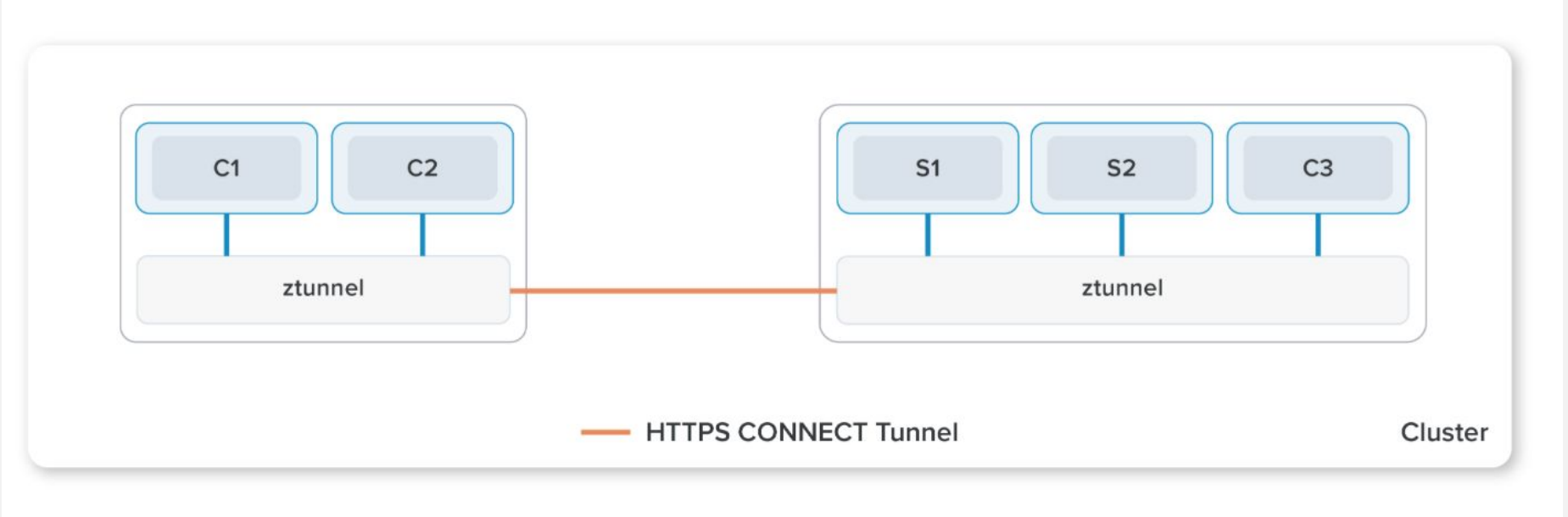
Advantages:

- **Resilience:** No single point of failure – as there is one proxy per application pod
- **Uniformity:** Traffic is routed uniformly through the proxy. This is identical for all applications
- **Multi-tenancy:** There is no shared infrastructure in the data plane between different parties
- **Scalability:** Proxy scales horizontally with the application
- **Zero code changes:** The application can be meshed without code modifications

Disadvantages:

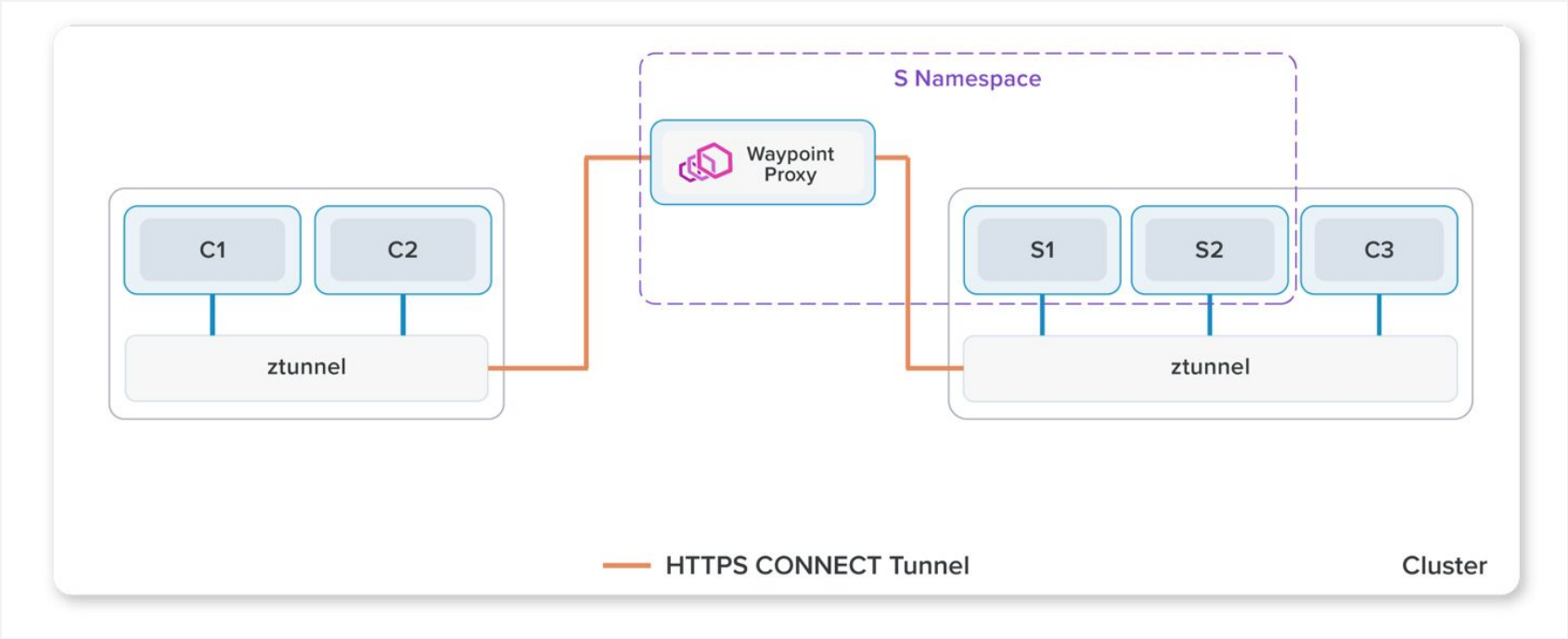
- **Resource Overhead:** Every pod in the mesh automatically gets a proxy injected, which in turn consumes resources like CPU and RAM.
- **Higher Latency:** All traffic is routed through proxies, requiring more hops. Therefore, Latency increases, which can be problematic depending on the application.
- **Resource Management:** The sidecar proxy runs as a container in Kubernetes and is subject to typical resource constraints. These need to be chosen and maintained carefully to ensure optimal resource utilization.
- **Security:** A new infrastructure component is deployed, potentially increasing the attack surface.

Data plane architectures - Ambient Mesh



Only supports Layer 4 features. As a result the implementation is much simpler and way more performant.

Data plane architectures - Ambient Mesh



In case Layer 7 features are required, envoy proxies are deployed.
All Traffic directed via the zTunnels through the proxy..

Data plane architectures - Ambient Mesh

Advantages:

- **Scalability**: Waypoint proxy scales horizontally, independently of applications
- **Zero code changes**: The application can be meshed without code modifications
- **Resource efficiency:** Expensive Envoy proxies are only used when L7 features are needed. In total, fewer proxies are deployed overall, as one waypoint proxy can be used per namespace.
- **Flexibility:** Can theoretically be deployed alongside the sidecar model

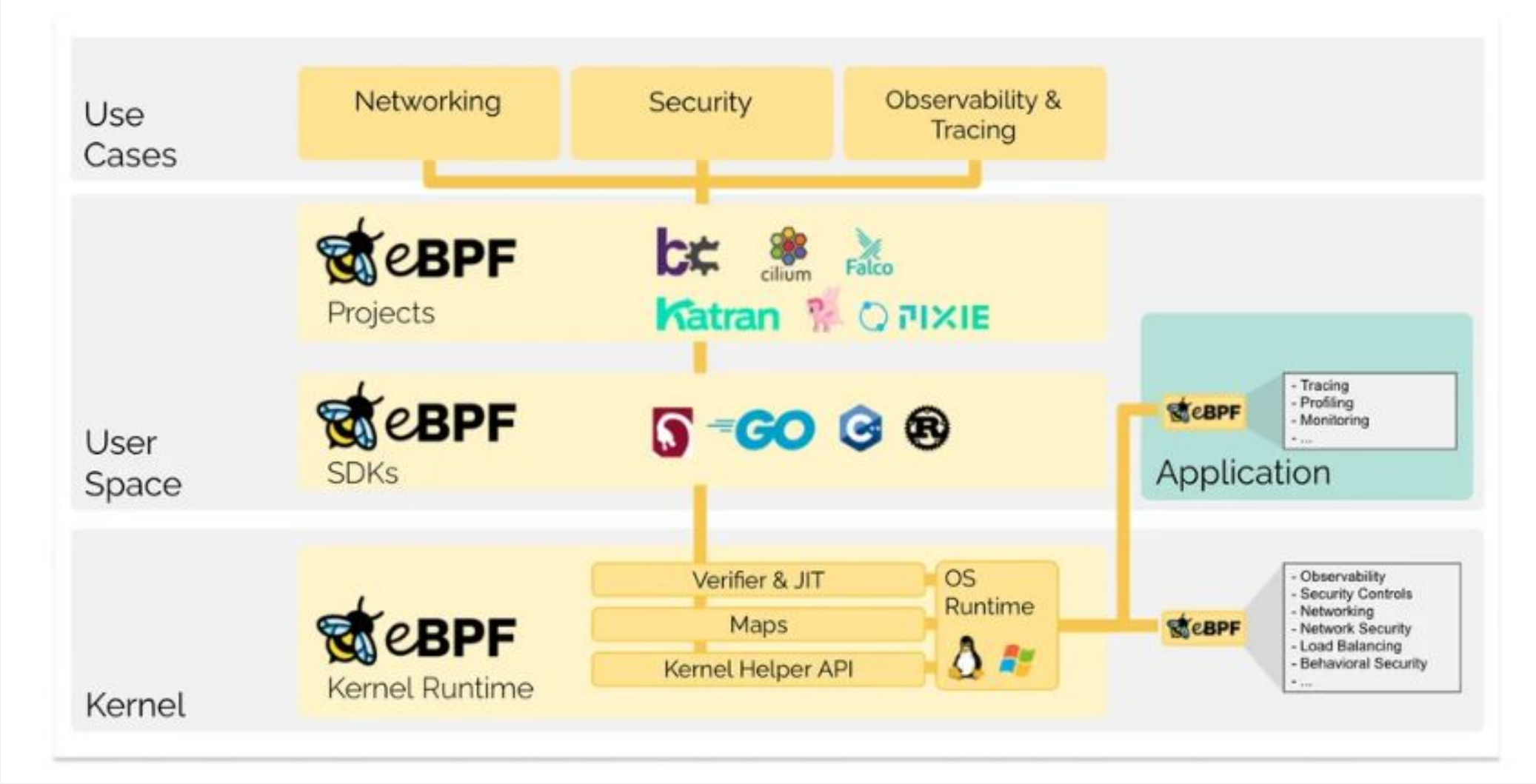
Disadvantages:

- **Potentially even higher latency:** When L7 features are needed, traffic is routed through a waypoint proxy. However, the proxy may no longer be on the same node as the application.

Evolution: extended berkeley packet filter (eBPF)

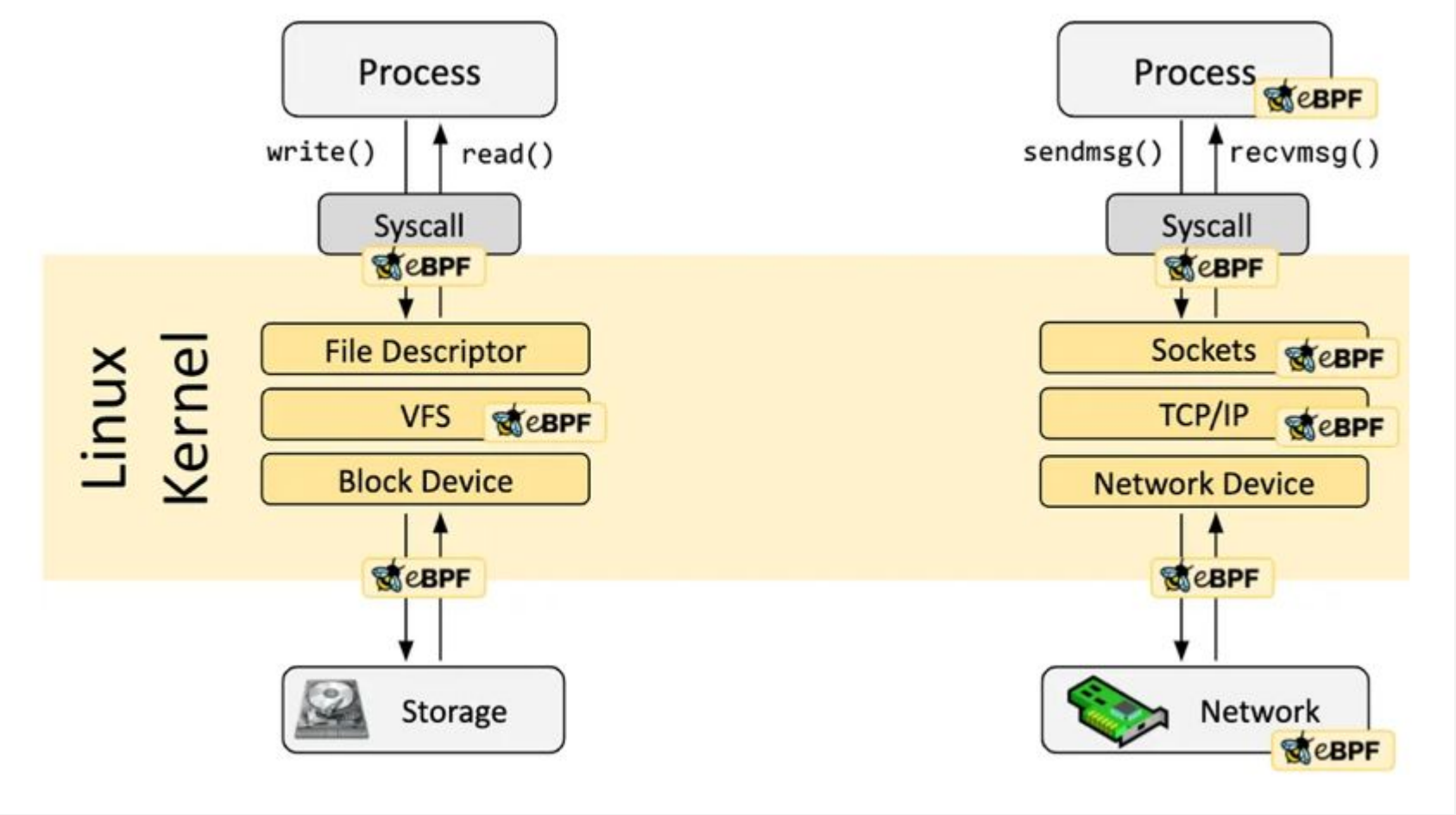
- Historically derived from the Berkeley Packet Filter (BPF).
- BPF was primarily a network tool.
- eBPF has significantly extended BPF, so the acronym essentially no longer makes sense.
- eBPF allows the user to dynamically extend the kernel with programs.
  - These programs run in a sandbox within the kernel.
  - They are JIT-compiled and must pass a verification engine in the kernel.
  - These programs run event-driven and subscribe to existing hooks in the kernel, e.g., system calls.
  - If there is no existing hook, a program can still be attached to relatively freely chosen points using a kProbe/uProbe.

Evolution: extended berkeley packet filter (eBPF)



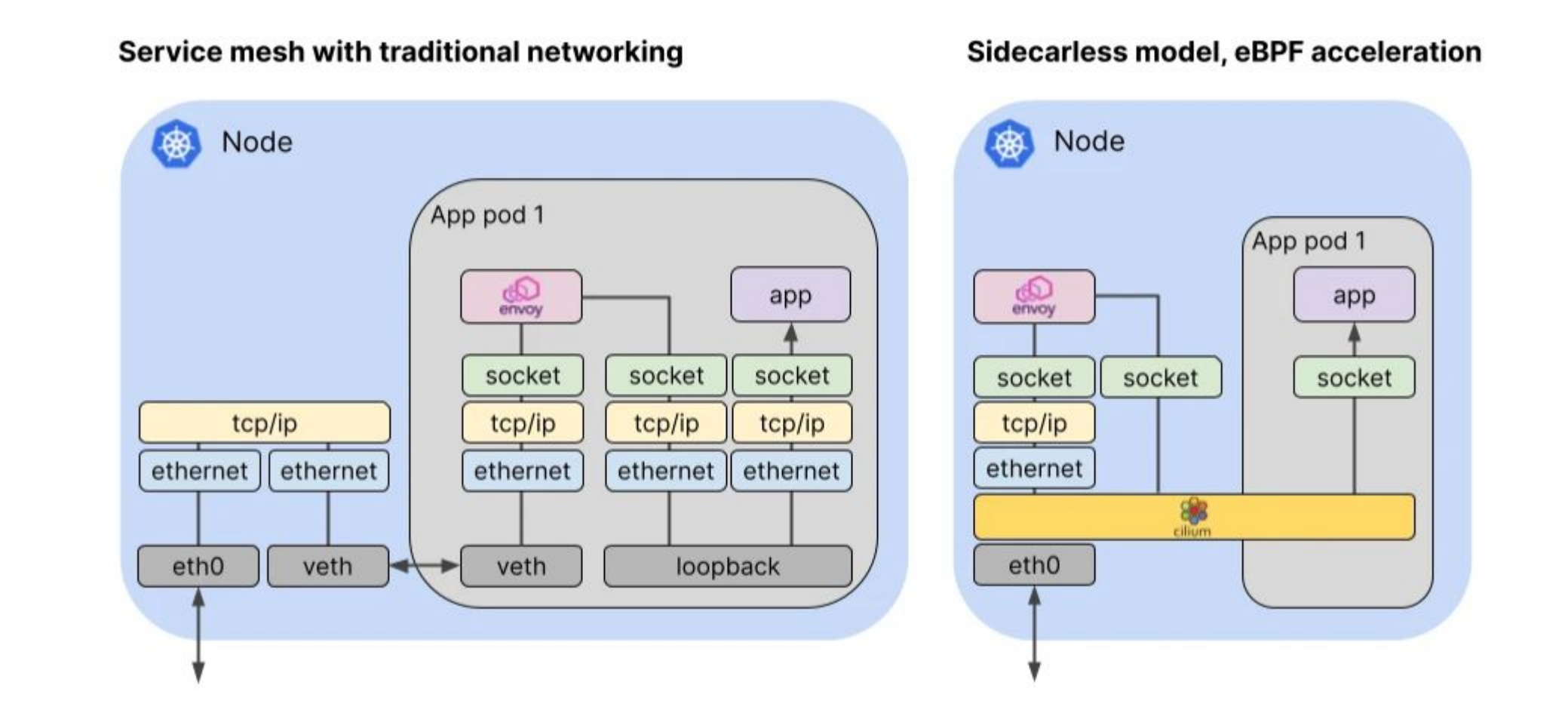
Quelle: https://ebpf.io/what-is-ebpf/#what-is-ebpf

Evolution: extended berkeley packet filter (eBPF)



Quelle: https://ebpf.io/what-is-ebpf/#hook-overview

Data plane architectures - eBPF based



Quelle: https://cilium.io/blog/2021/12/01/cilium-service-mesh-beta/

Data plane architectures - eBPF based

Advantages:

- **Zero code changes:** The application can be meshed without code modifications
- **Resource efficiency:** Performance is superior to the previous model
- **Flexibility:** Many features can be implemented in eBPF. Only some L7 features still require an L7 proxy

Disadvantages:

- **Complexity:** New, challenging technology. How do I debug this as a user?

CI / CD

What is DevOps?

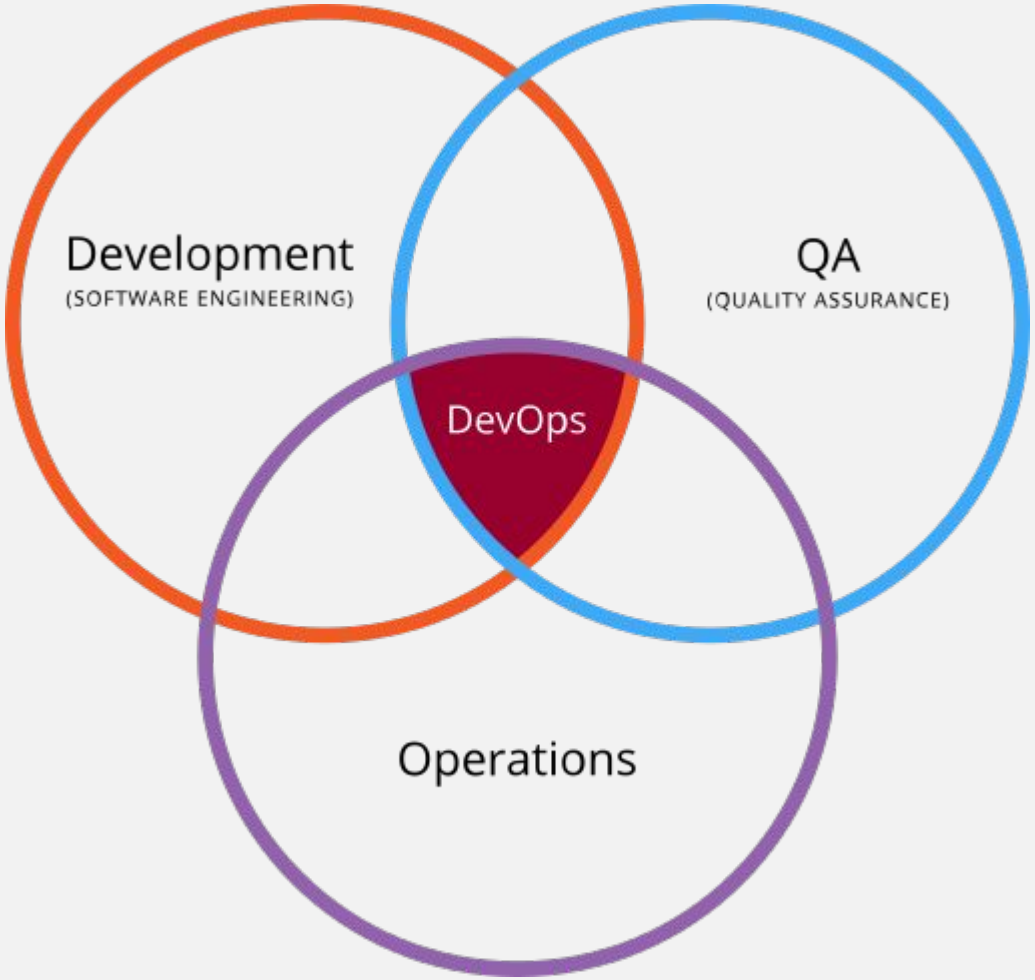
QA|WARE

DevOps is the improved integration of **development** and **operations** through greater **collaboration** and **automation**, aiming to **deploy** changes to production more quickly and keep the **MTTR** (Mean Time to Recovery) low. **DevOps** is therefore a **culture**.

MVP + Feature Stream
Per Feature:

- Minimal manual post-commit effort until PROD
- Diagnosability of a feature's success
- Ability to disable/roll back the feature

DevOps connects DEVelopment, OPerations and Quality Assurance



„Venn diagram showing DevOps“ by Rajiv.Pant/Wylve; Creative Commons 3.0

Continuous X

Continuous Integration (CI)

- All changes are immediately integrated into the current development branch and tested.
- This ensures continuous testing of compatibility between changes.

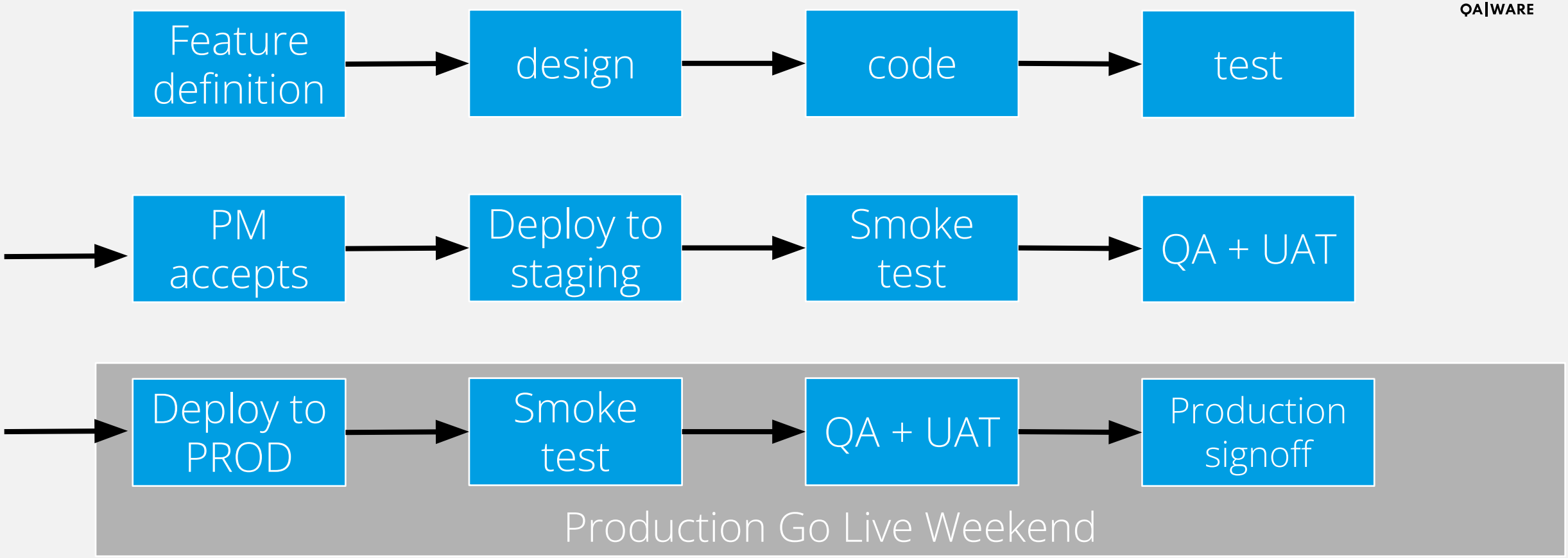
Continuous Delivery (CD)

- The code *can* be deployed at any time.
- However, it does not have to be deployed immediately.
- This means the code must (ideally) always build, be tested, and debugged when necessary.

Continuous Deployment

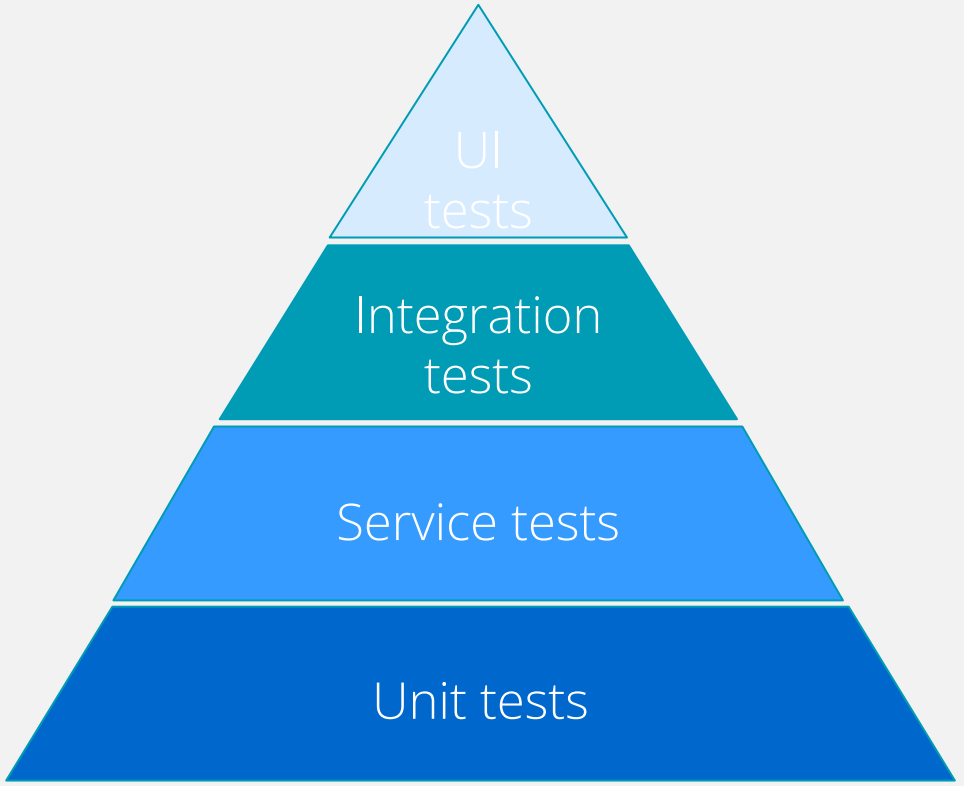
- Every stable change is deployed to production.
- Part of the quality testing takes place directly in production.
- The ability to handle errors must be in place (e.g., Canary Release, see later).

Example: Pipeline without Continuous Delivery



https://www.scaledagileframework.com/continuous-delivery-pipeline/

Example of a test pyramid that aids early feedback on errors

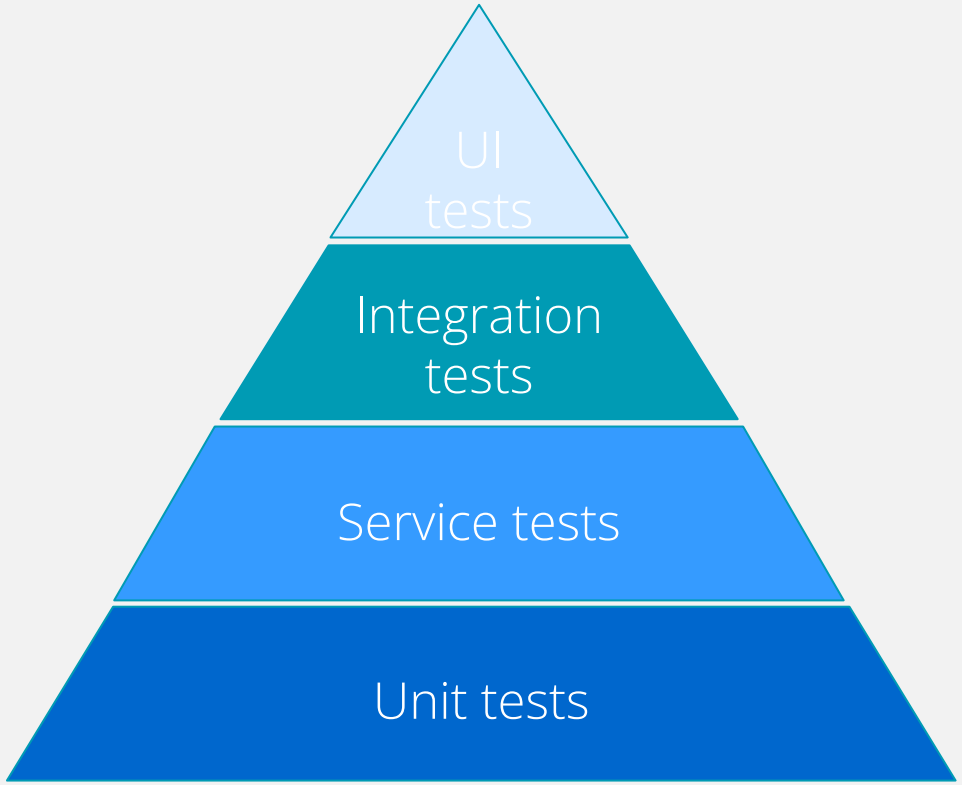


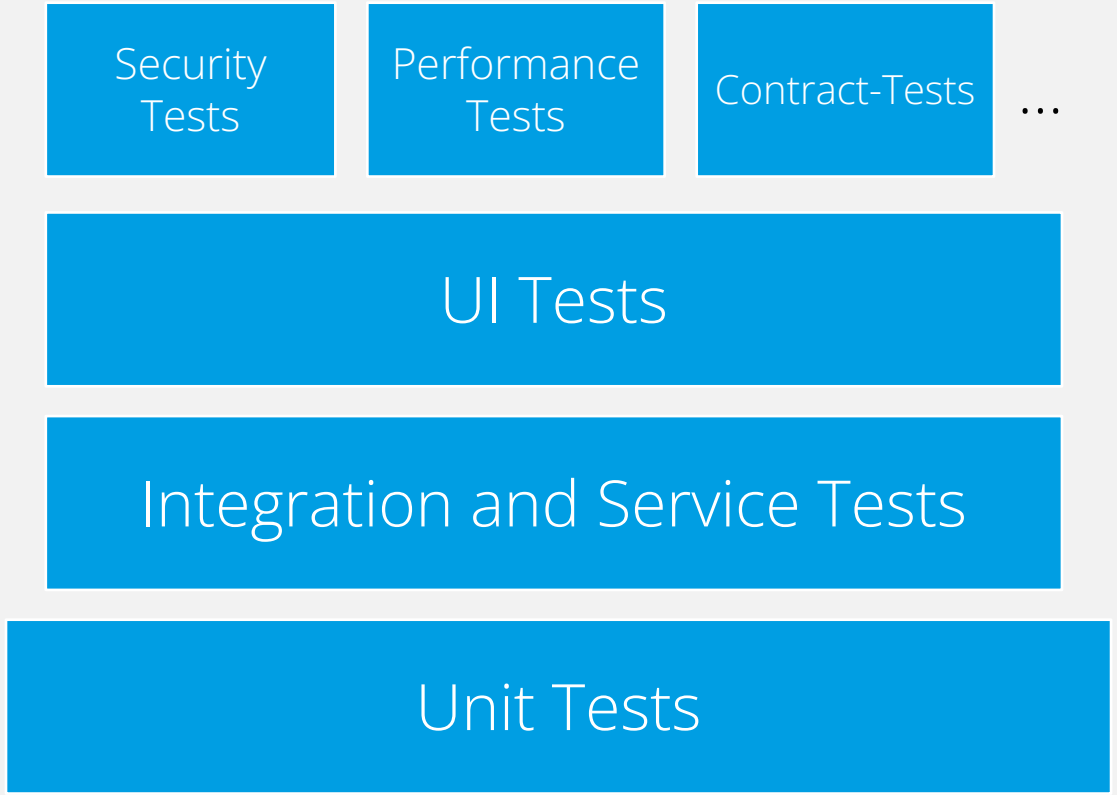
Unit Tests:
Classic unit tests (e.g., JUnit, Mockito).
Service Tests:
Tests for a single microservice, including REST controllers and client calls (e.g., JUnit, Spring MVC Tests, Wiremock).
Mocks for other microservices are necessary (e.g., using Wiremock).
Integration Tests:
Tests the integration of multiple services and their interactions (e.g., JUnit, Spring MVC Tests).
Performance Tests:
Checks for significant performance changes (e.g., Gatling).
UI Tests:
Tests UI functionality and its interaction with the backend (e.g., Selenium, Protractor).

All tests should be executed as often as possible—ideally with every commit!

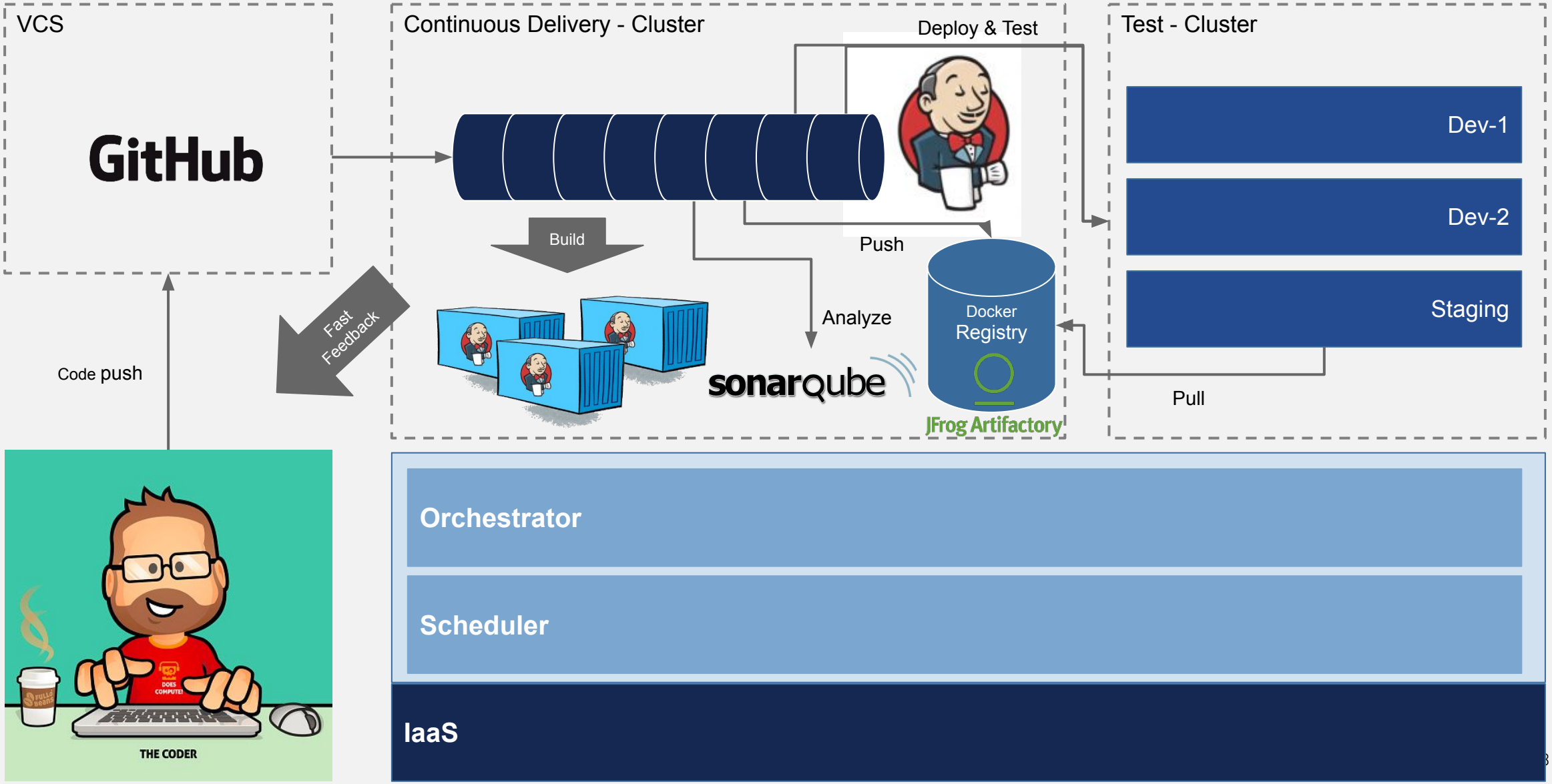
All layers should run in an automated fashion.
The pyramid may also come in different shapes:

QA|WARE





Example architecture of continuous delivery pipeline



GitOps

The Idea

1 Declarative

A system managed by GitOps must have its desired state expressed declaratively.

2 Versioned and Immutable

Desired state is stored in a way that enforces immutability, versioning and retains a complete version history.

3 Pulled automatically

Software agents automatically pull the desired state declarations from the source.

4 Continuously reconciled

Software agents continuously observe actual system state and attempt to apply the desired state.

Advantages gained out of the box

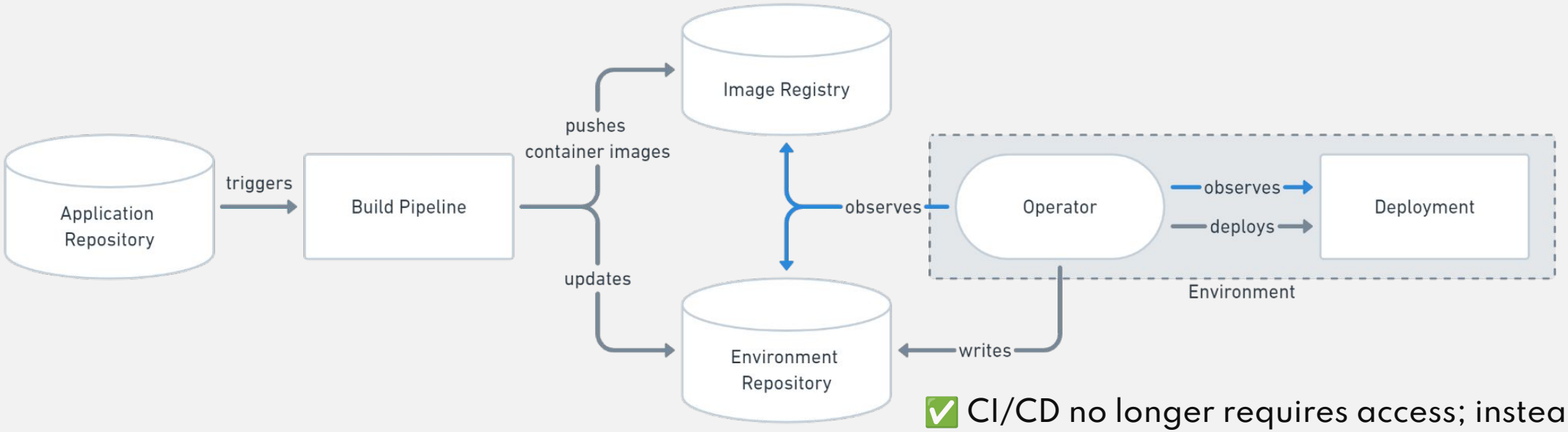
- Ideally allows restoring any system state in history, e.g., for easy rollbacks.
- Enforces pipelines.
- Provides transparency for changes; in the case of Git, it also shows who made changes.
- Ensures that the system state does not deviate from the desired state.

GitOps in the K8s ecosystem

ArgoCD

Flux

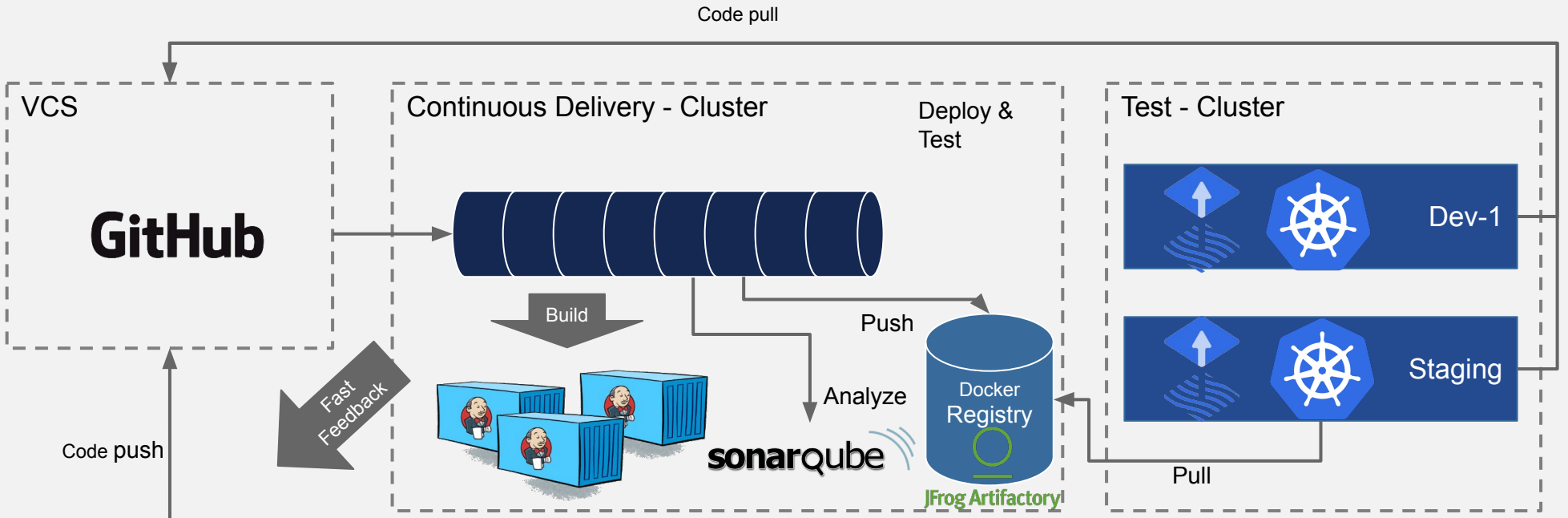
Pull-based deployments with GitOps



✅ CI/CD no longer requires access; instead, the operator accesses the environment repo and registry.
✅ The operator attempts to establish the desired cluster state, including resource deletion.
❌ More complex

https://www.gitops.tech/#what-is-gitops

GitOps with Flux



Differences in the architecture

CI/CD pipeline no longer requires cluster access

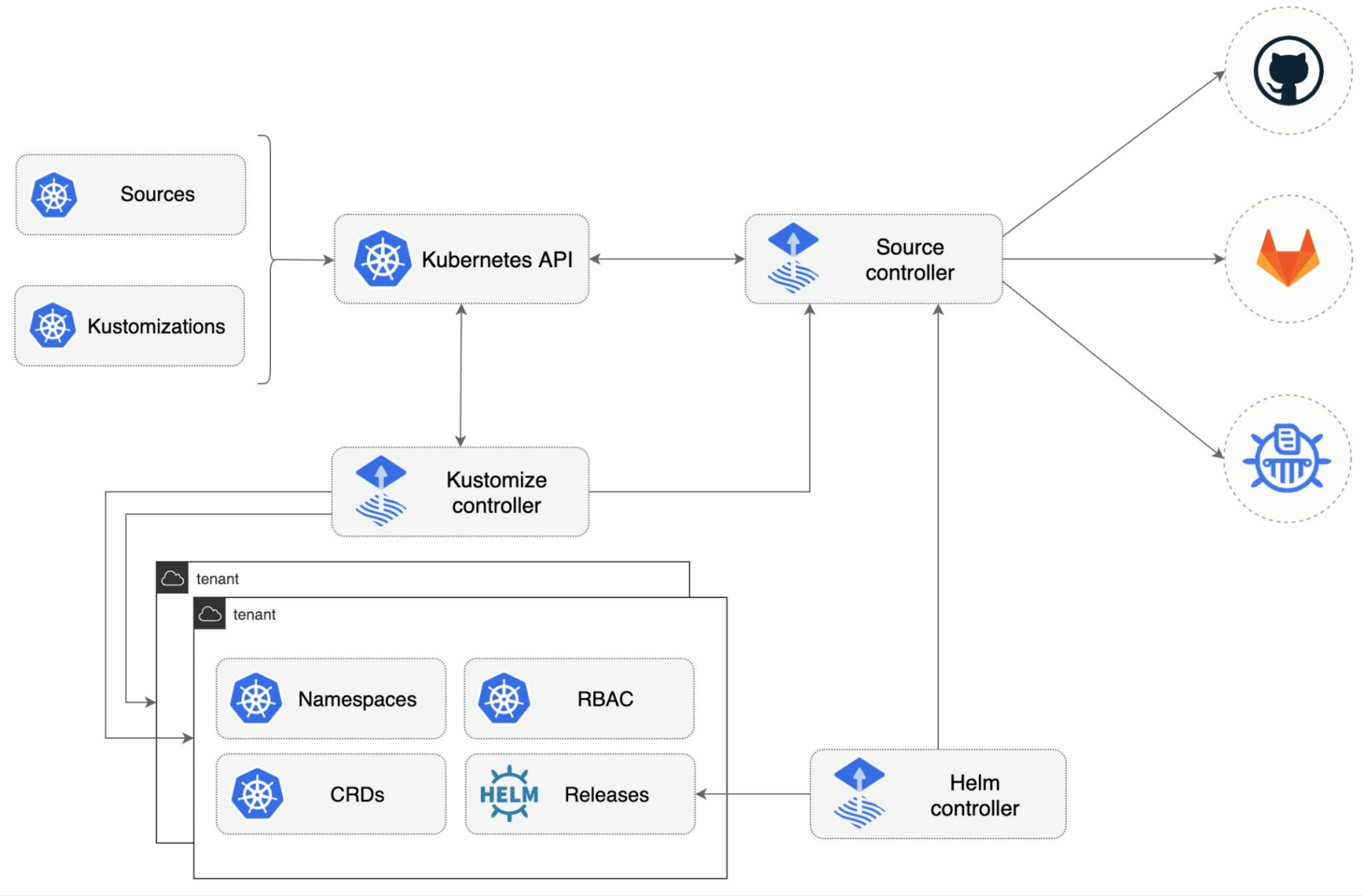
=> Improved security, as no high-privilege credentials are needed for the pipeline.

Clusters now actively pull source repositories that contain the desired state.

Clusters reconcile their state, meaning the current state is compared with the desired state, and the cluster attempts to reach the desired state.

=> Clusters automatically converge toward the desired state.

Flux in detail



https://fluxcd.io/flux/components/

Flux in detail

Source Controller

Connects to various sources, e.g., Git repositories, OCI repositories, etc.
Provides the downloaded packages to other controllers.

Helm Controller

Offers Helm integration for Flux.
In combination with the Source Controller, Helm charts can be automatically downloaded, installed, and updated.

Kustomize Controller

Uses the Git repositories managed by the Source Controller to execute Kustomize and apply the configured resources to the cluster.
Tracks all resources, enabling automated garbage collection of unused resources.

Notification Controller

Provides insights into Flux events, such as successes or errors.
Includes integrations with chat systems.

Configure a source


```yaml
apiVersion: source.toolkit.fluxcd.io/v1
kind: GitRepository
metadata:
  name: podinfo
  namespace: default
spec:
  interval: 5m0s
  url: https://github.com/stefanprodan/podinfo
  ref:
    branch: master
```

Configures a GitRepository as a source in the Source Controller.
Clones the master branch from the URL and makes it available to other controllers as a tar.gz file.
Polls the URL every 5 minutes to check for changes.

Configure a kustomization


```yaml
apiVersion: kustomize.toolkit.fluxcd.io/v1
kind: Kustomization
metadata:
  name: podinfo
  namespace: default
spec:
  interval: 10m
  targetNamespace: default
  sourceRef:
    kind: GitRepository
    name: podinfo
  path: "./kustomize"
  prune: true
  timeout: 1m
```

Creates a Kustomization in the Kustomize Controller that references the GitRepository from the Source Controller via **sourceRef**.

The Kustomize Controller retrieves the source code from the Source Controller and uses Kustomize (a Kubernetes tool) to apply everything under the specified **path**.

Observability

A system can be easily diagnosed if you can easily recognize and correct healthy and unhealthy conditions.

A diagnosable system has a

- short Mean Time to Detect (MTTD)
- short Mean Time to Repair (MTTR)

and thus to a short period of time in which errors remain undetected and even exist.

Diagnosability: A structured approach is necessary to avoid influencing the system too much.

1. **Get an overview:**
   a. What are the central components, e.g. login, etc.?
   b. What are the supporting components, e.g. batch and loader jobs, etc.?
2. **Identify error limits:**
   a. Internal error limits: layers/use cases
   b. External error limits: incoming and outgoing calls
3. **Define error classes:**
   a. Severity levels: operation still possible, no impact on customer, etc.
   b. Impact: certain functions are not available to customers, etc.
4. **Determine the runtime data that is necessary for detection:**
   a. Uniformity: data has the same meaning; uniform data formats are defined, etc.
   b. Clearly defined: it is clear what the metric means, e.g. CPU load
5. **Define actions:**
   a. Alerting: who is notified and when
   b. Create playbooks for errors: How do I get the data, etc.

Logs

- **Logs** are an important part of any IT system.
- In the cloud, we need to look at logs from many different services at the same time.
- Every system has logs, but they are always in a different **format.**

- Example: **Quarkus** webservice:

```
2023-01-10 20:56:42,122 DEBUG [io.qua.mic.run.bin.ver.VertxHttpServerMetrics]
(vert.x-eventloop-thread-11) requestRouted null HttpRequestMetric [initialPath=/q/metrics,
currentRoutePath=null, templatePath=null, request=io.vertx.core.http.impl.Http1xServerRequest@512b1fd6]
```

- Example: **Nginx** webserver:

```
- 2a02:c207:3005:5132::1 - - [10/Jan/2023:00:00:13 +0100] 0.000 repo.saturn.codefoundry.de "POST
/api/v4/jobs/request HTTP/1.1" 957 204 0 "-" "gitlab-runner 15.6.1 (15-6-stable; go1.18.8; linux/amd64)"
```

Logging. Please be structured. Please be well-considered. /1

- Define a log format and ensure that all services use the same format.
- Define a diagnostic context = information that helps in the event of an error, e.g.
  - TraceId
  - UserId
  - SessionId
- Use structured logging (e.g. JSON)
  - Simplifies handling in the analysis
- Use asynchronous logging (provided that the recipients support it) to prevent blockages
  - If all of this is sent over TCP etc. and then filtered and sorted later, the order doesn't matter 😎

Logging. Please be structured. Please be well-considered. /2

- Don't log too much, but log every
  - exception,
  - every error,
  - every meaningful piece of information …
  - … but not multiple times.
- Use log levels. To do this, define which category should have which level, e.g.:
  - WARN: Errors that affect a single request but not the stability of the service.
  - ERROR: Errors that affect the stability of the service.

Metrics: Basics

Metrics are measurements that reflect the current state of the system.

Examples:

- **CPU** load
- Size of the **JVM Heap**
- Number of **calls to an interface**

The metrics are provided by the system to be monitored.

Metrics can also have metadata.

Metrics: Basics

Grafana Alloy pulls metrics from all configured targets every **15 seconds**.

The transport is usually done via HTTP: **GET /metrics**

No aggregation is done here yet. The consumer is responsible for that.

```
# HELP process_open_fds Number of open file descriptors.
# TYPE process_open_fds gauge
process_open_fds 8
# HELP process_start_time_seconds Start time of the process since unix epoch in seconds.
# TYPE process_start_time_seconds gauge
process_start_time_seconds 1.65566242485e+09
# HELP process_virtual_memory_bytes Virtual memory size in bytes.
# TYPE process_virtual_memory_bytes gauge
process_virtual_memory_bytes 1.782685696e+09

# HELP http_server_requests_seconds
# TYPE http_server_requests_seconds summary
http_server_requests_seconds_count{method="GET",status="200",uri="/tle",} 1.0
http_server_requests_seconds_sum{method="GET",status="200",uri="/tle",} 8.183356641
# HELP http_server_requests_seconds_max
# TYPE http_server_requests_seconds_max gauge
http_server_requests_seconds_max{method="GET",status="200",uri="/tle",} 8.183356641
```

Metric Types

Metrics have different types to visualize data:

- **Counter**

  A number that can either be incremented or reset.
- **Gauge**

  A metric that can change in either direction.
- **Histogram**

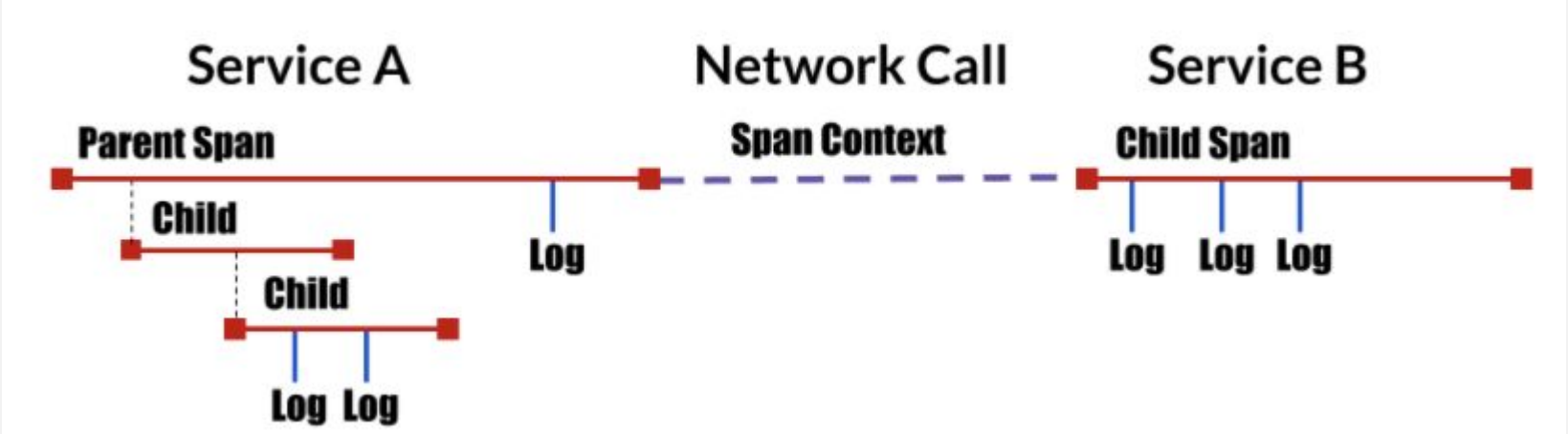
  Values in “buckets” - e.g. the duration of requests.
- **Summary**

  Similar to histogram, summarizes values.

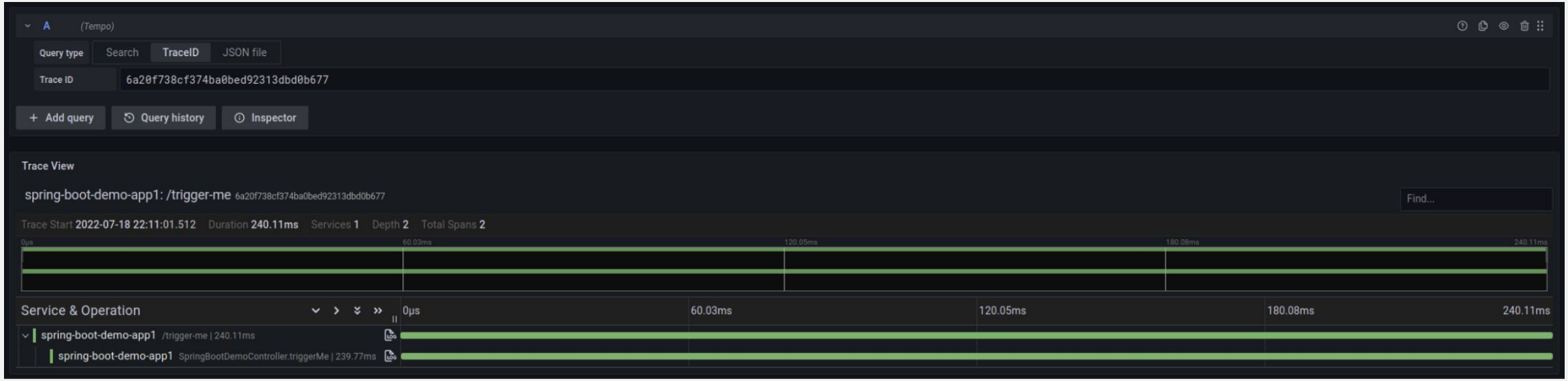
Distributed Tracing

- Distributed Tracing: Technology for tracing calls and processes in distributed software systems.
- Today's considerations go back to the paper: Dapper, a Large-Scale Distributed Systems Tracing Infrastructure
- Idea: Each service forwards information from the caller to the next service and back again. This results in a directed graph.
  - Each service must be instrumented. Otherwise there will be a gap.
  - The smallest unit is called a span. A span has a duration and descriptive information. A trace is a set of 1..n spans.
- Many implementations exist for different programming languages and technologies.
  - https://opentelemetry.io/docs/languages/
  - https://opentelemetry.io/docs/zero-code/

Distributed Tracing



Traces: Foundation



Traces visualize which paths a request has taken through the microservices……

… and where an error may have occurred!

Architecture

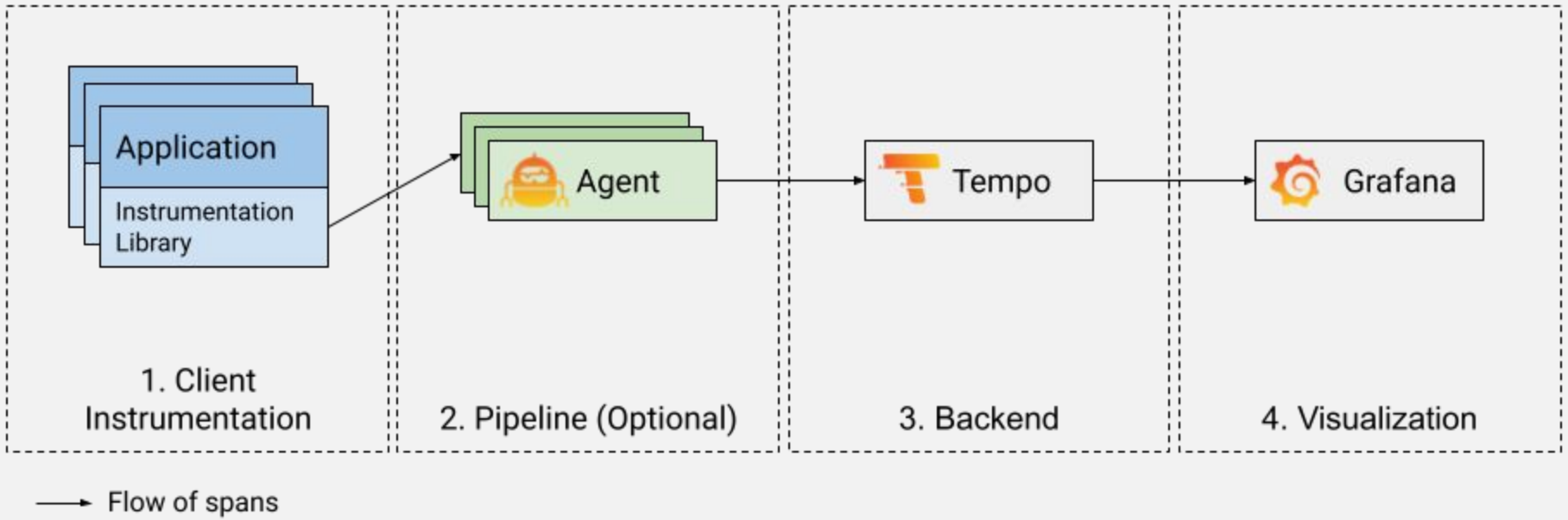
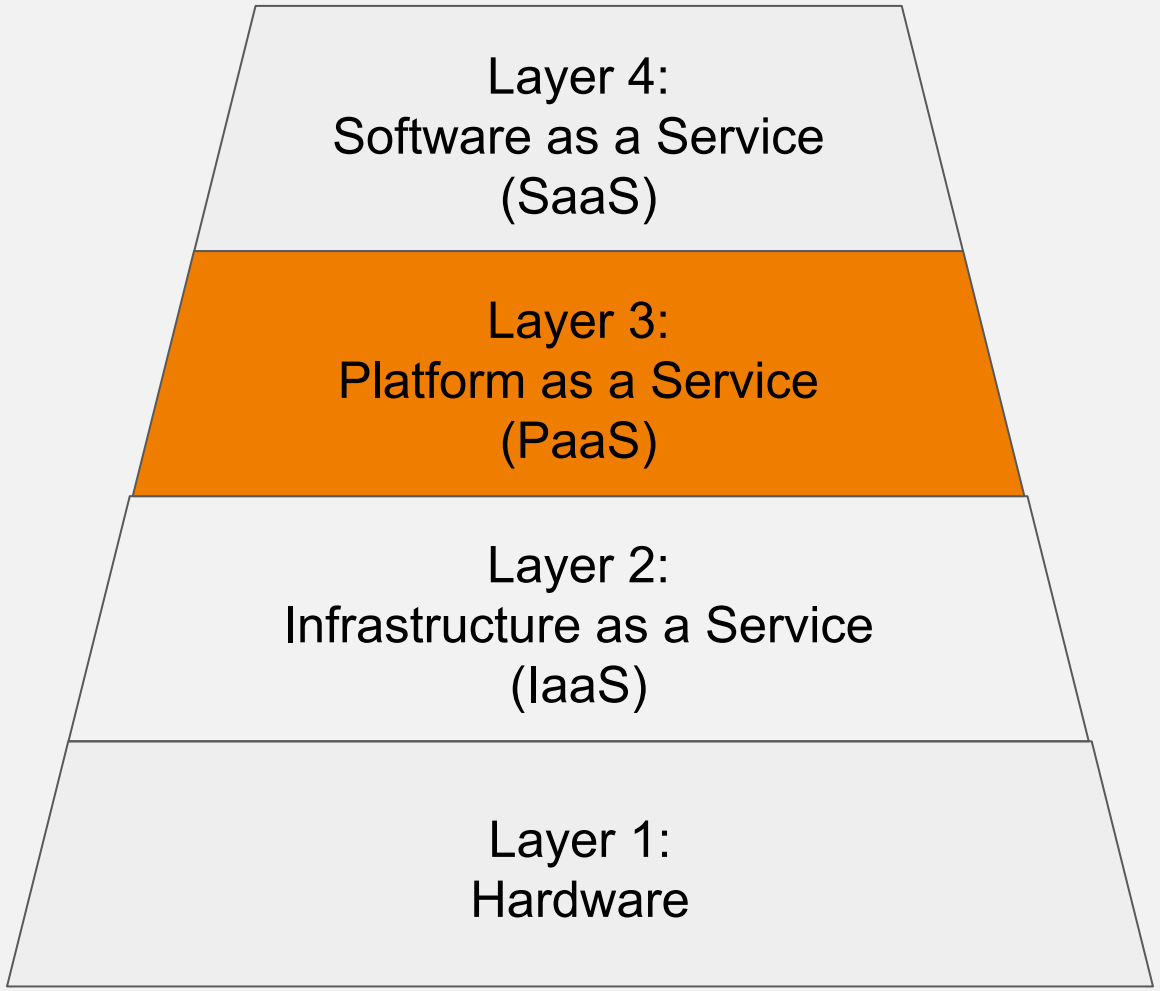


Bild: https://grafana.com/docs/tempo/latest/getting-started/

PaaS

The layered model of cloud computing: From metal to application.



- Customizable software services
- CaaS: Component as a service (e.g. Google Charts)
- BaaS: Backend as a Service
- Transparent Updates

Target Group: Users

- Platform services
- Application programming interfaces (APIs)
- Abstraction of technical infrastructure

Target group: Developers

- Elasticity
- Virtual resourcepools
- Technical infrastructure: Machines, Servers (DNS, DHCP, LB, NAS, …)

Target group: Operations

- Computers
- Network
- Storage

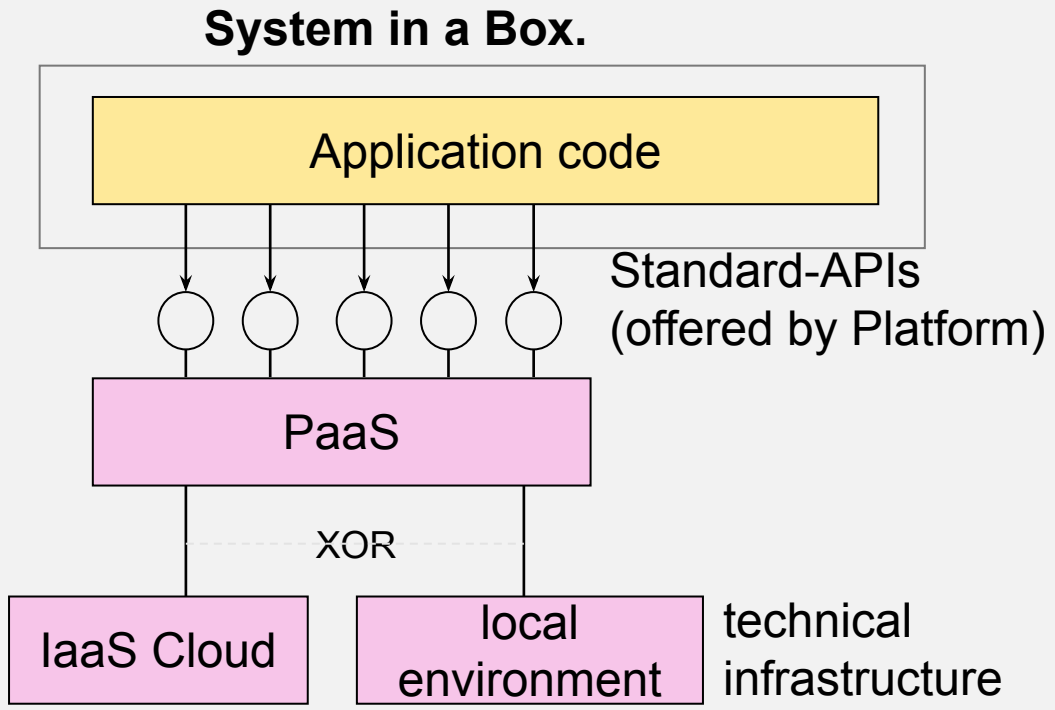
https://www.youtube.com/watch?v=M988_fsOSWo

Idea: Platform-as-a-Service offers an ad-hoc Development- and Operations Platform.

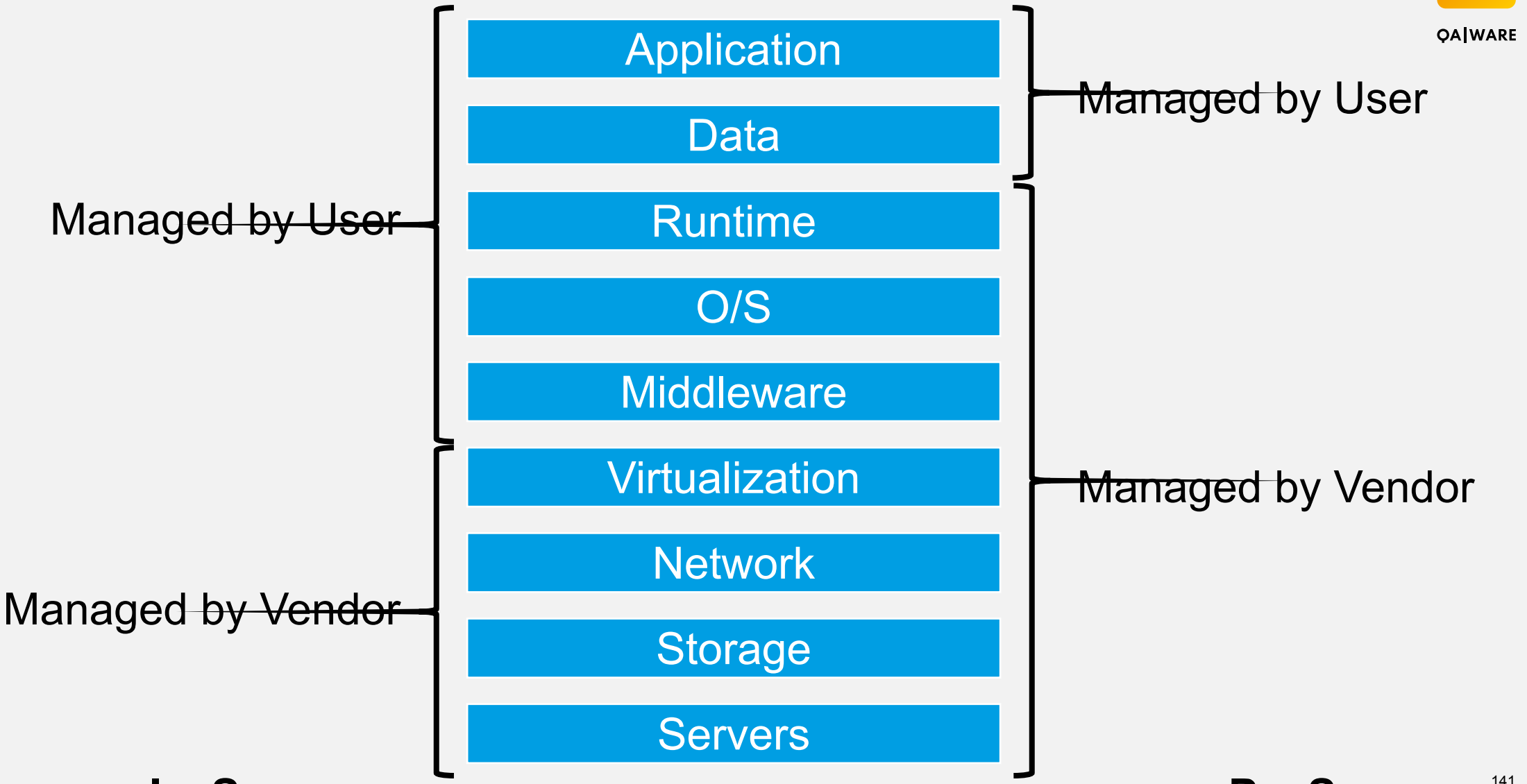
The application is deployed either as an application package or as source code. No image with technical infrastructure is required.

The application only interacts with programming or access interfaces of its runtime environment.

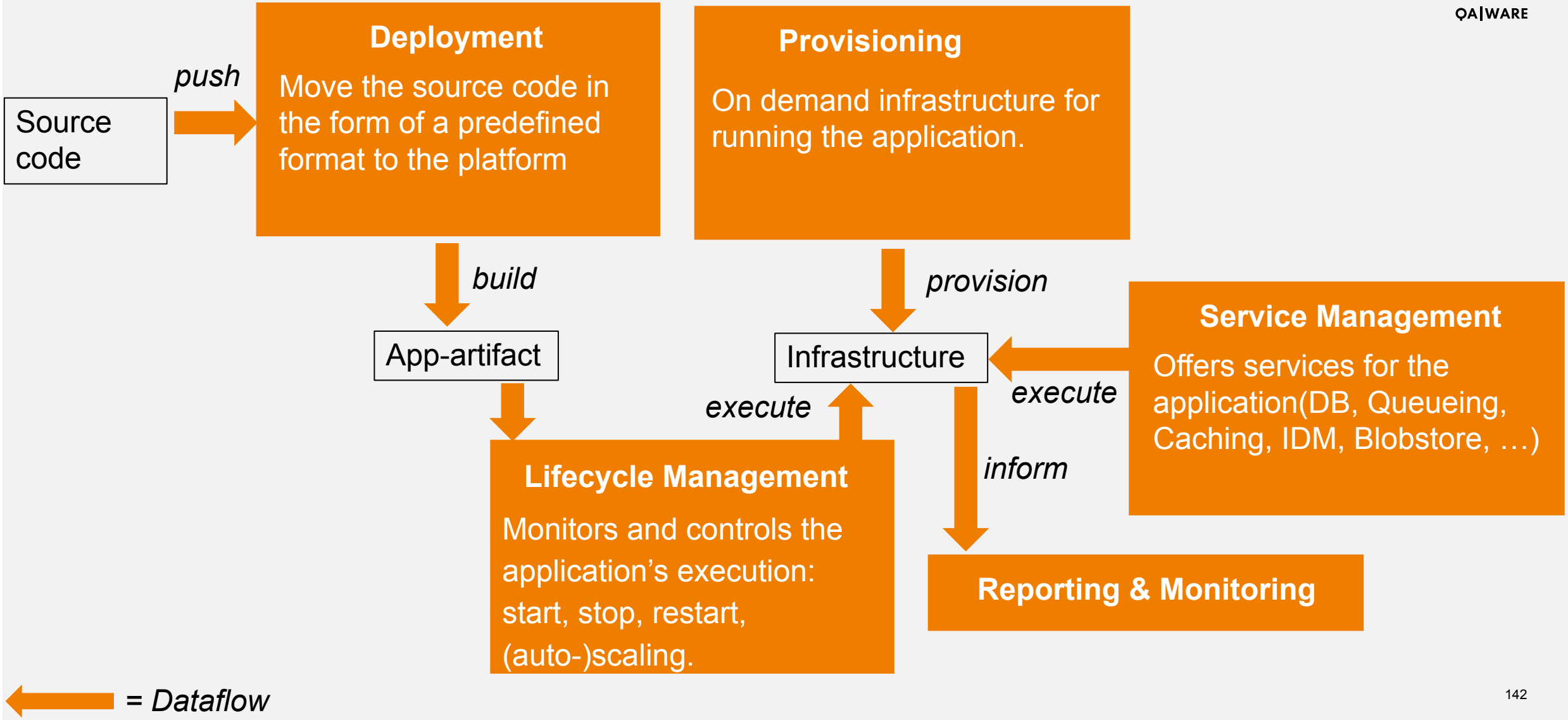
“Engine and operating system should not matter…”
The application is automatically scaled.



IaaS vs. PaaS


Application
Data
Runtime
O/S
Middleware
Virtualization
Network
Storage
Servers
Managed by User
Managed by Vendor
Managed by User
Managed by Vendor
IaaS
PaaS

The functional building blocks of a PaaS Cloud.



Serverless Computing

Serverless Computing - Definition

Serverless computing is a cloud computing execution model in which the cloud provider dynamically manages the allocation of machine resources. Pricing is based on the actual amount of resources consumed by an application, rather than on pre-purchased units of capacity.[1] It is a form of utility computing.

Serverless computing still requires servers, hence it's a misnomer.[1] The name "serverless computing" is used because the server management and capacity planning decisions are completely hidden from the developer or operator. Serverless code can be used in conjunction with code deployed in traditional styles, such as microservices. Alternatively, applications can be written to be purely serverless and use no provisioned services at all.[2]

wikipedia.org

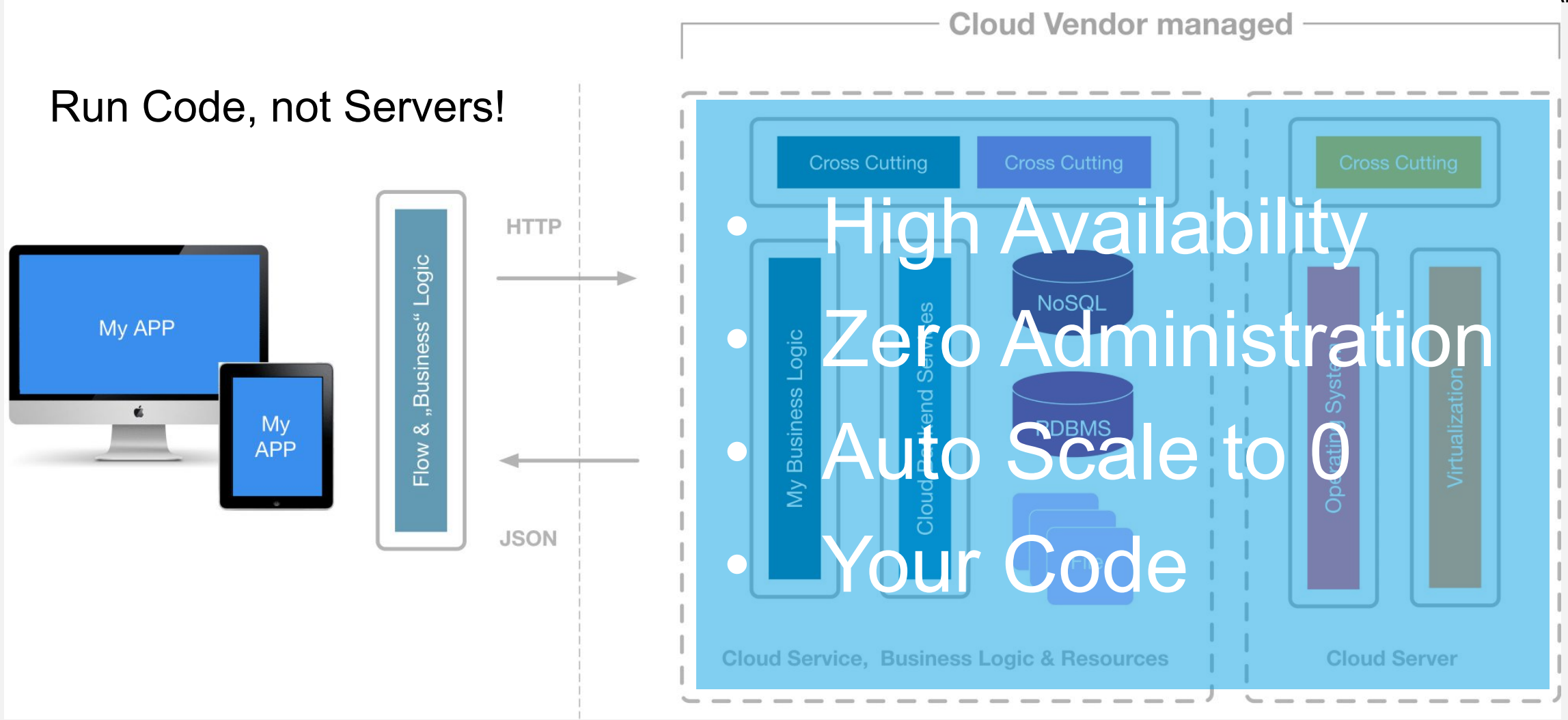
Serverless computing allows you to build and run applications and services without thinking about servers. Serverless applications don't require you to provision, scale, and manage any servers. You can build them for nearly any type of application or backend service, and everything required to run and scale your application with high availability is handled for you.

amazon.com

Serverless Computing – Overview & Comparison to PaaS

- Serverless computing is often referred to as Function as a Service (FaaS).
    - FaaS is a specialized form of PaaS.
    - Deployment and operations are handled by the cloud provider, making a FaaS platform similar to PaaS.
- A key difference from 'traditional' PaaS platforms:
    - The provider does not guarantee that a single function is constantly deployed. Often, the function is only loaded/deployed when needed.
- Fine-grained administration of a PaaS is unnecessary.
- Developers in general do not need to worry about the runtime environment (e.g., building containers). However, not true for all languages and runtimes.
- The primary architectural style of FaaS is event-driven architecture (EDA).
  The largest providers are Amazon with AWS Lambda, Microsoft with Azure Functions, and Google with App Engine.

Serverless application architecture



Serverless Computing – Advantages & Disadvantages

Advantages:

- **Cost:** Since individual functions are only deployed when used, this is often more cost-effective than running servers continuously.
- **Productivity:** Individual functions can be written, deployed, and updated very quickly.
- **Performance:** Individual functions can be scaled very granularly.

Disadvantages:

- **Performance:** Since functions may only be loaded on demand, significant fluctuations in execution time can occur.
- **Debugging:** Beyond error messages and log output, there are fewer diagnostic options available, making debugging and profiling the application more difficult.


QA|WARE
Q & A