Aalto University
School of Electrical
Engineering

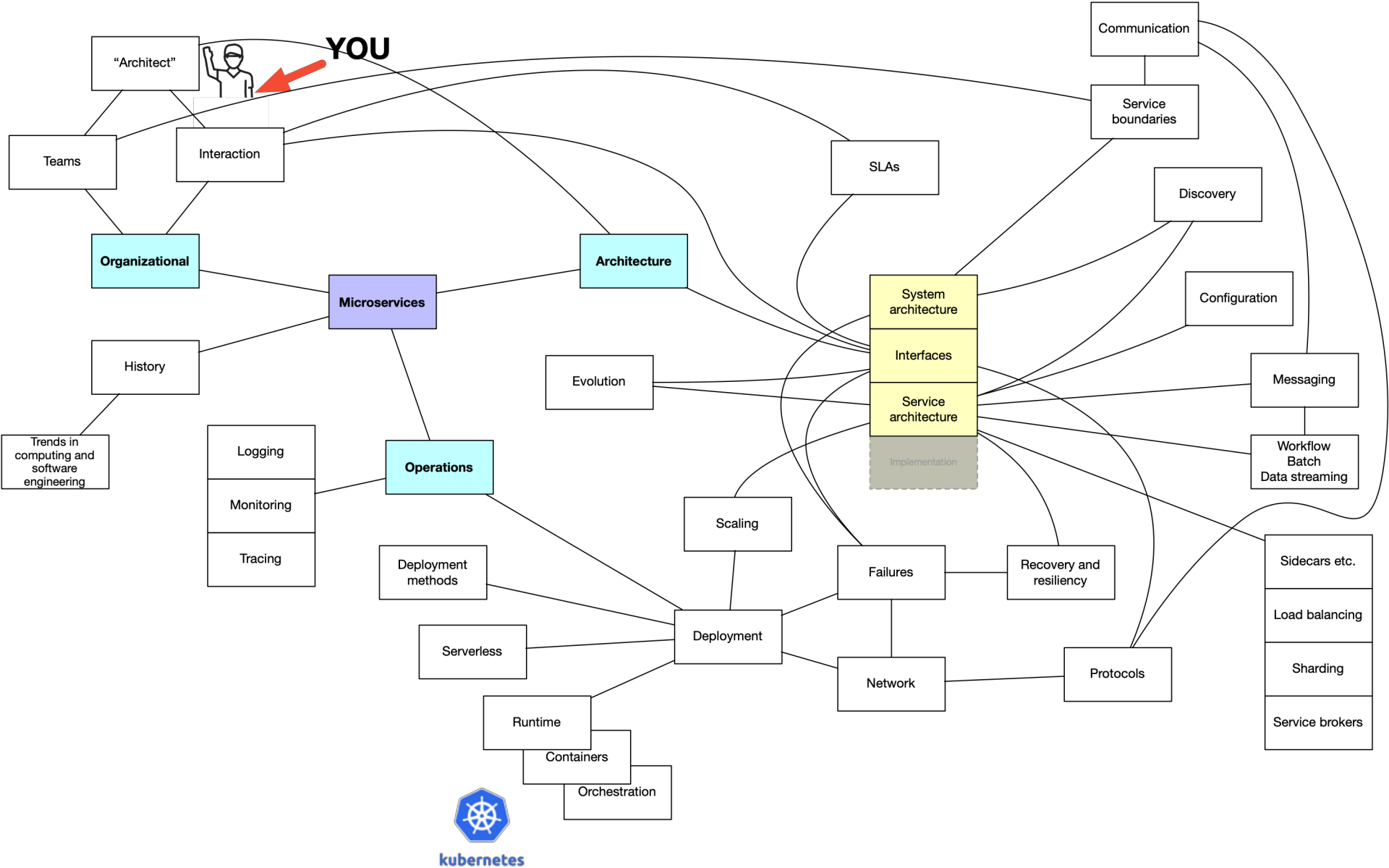
Course recap

4.4.2018

Santeri Paavolainen

Caveat emptor

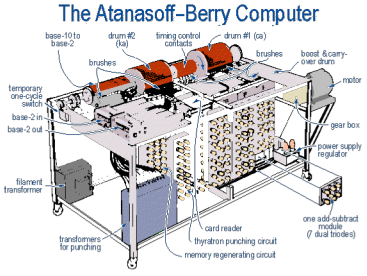
- **This set of slides covers only portion of the course material**
  - Most important, but not all
  - Focusing on refreshing the topics, not in details

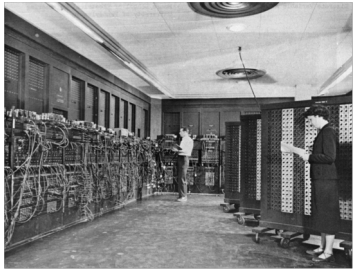

"Architect"
YOU
Teams
Interaction
Organizational
History
Trends in computing and software engineering
Microservices
Logging
Monitoring
Tracing
Operations
Deployment methods
Serverless
Runtime
Containers
Orchestration
kubernetes
Architecture
SLAs
Evolution
Scaling
Deployment
Communication
Service boundaries
Discovery
System architecture
Interfaces
Service architecture
Implementation
Configuration
Messaging
Workflow
Batch
Data streaming
Failures
Recovery and resiliency
Network
Protocols
Sidecars etc.
Load balancing
Sharding
Service brokers

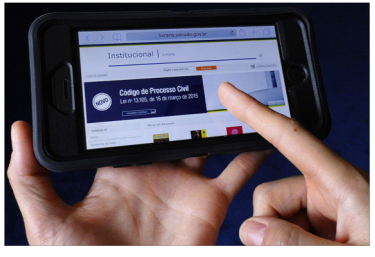
What does a software architect do?

- **In a larger company**
  - Draw fancy pictures ← **Communication**
  - Talk to lots of stakeholders ← **Communication**
  - Attend lots of meetings ← **Communication**
  - Talk to other engineers ← **Communication**
  - Draw lots of stuff on whiteboard ← **Communication**
- **In a start-up**
  - Less talking and less meetings ← **Communicate with actual customers**
  - Less people to catch your mistakes

Trends in computing







Genesis **Custom built** **Product** **Commodity**

Sources:Joi Ito (SAGE, CC BY 2.0), Arnold Reinhold (IBM 1401, CC BY-SA 3.0), Ben Franske (IBM S/360, CC BY 2.5), Veradrive (IBM PC/XT, CC BY 4.0), Senado Feredal (Smartphone, CC BY 2.0), Google (Google data center)

Scarcity

Few users

Little data

Abundance

World population

Big data

Then

Now

What is a microservice?

Microservices as an architectural design model

- **Loosely coupled architectures**
  - Parameterized configuration and service discovery
  - Independent component lifecycles
- **Fine grained component separation**
  - Identifying domains of logical responsibilities
- **Identifying and managing state**
  - Preference to purely stateless or purely stateful components

- **This is a high-level technology design viewpoint**

Microservices as implementation patterns

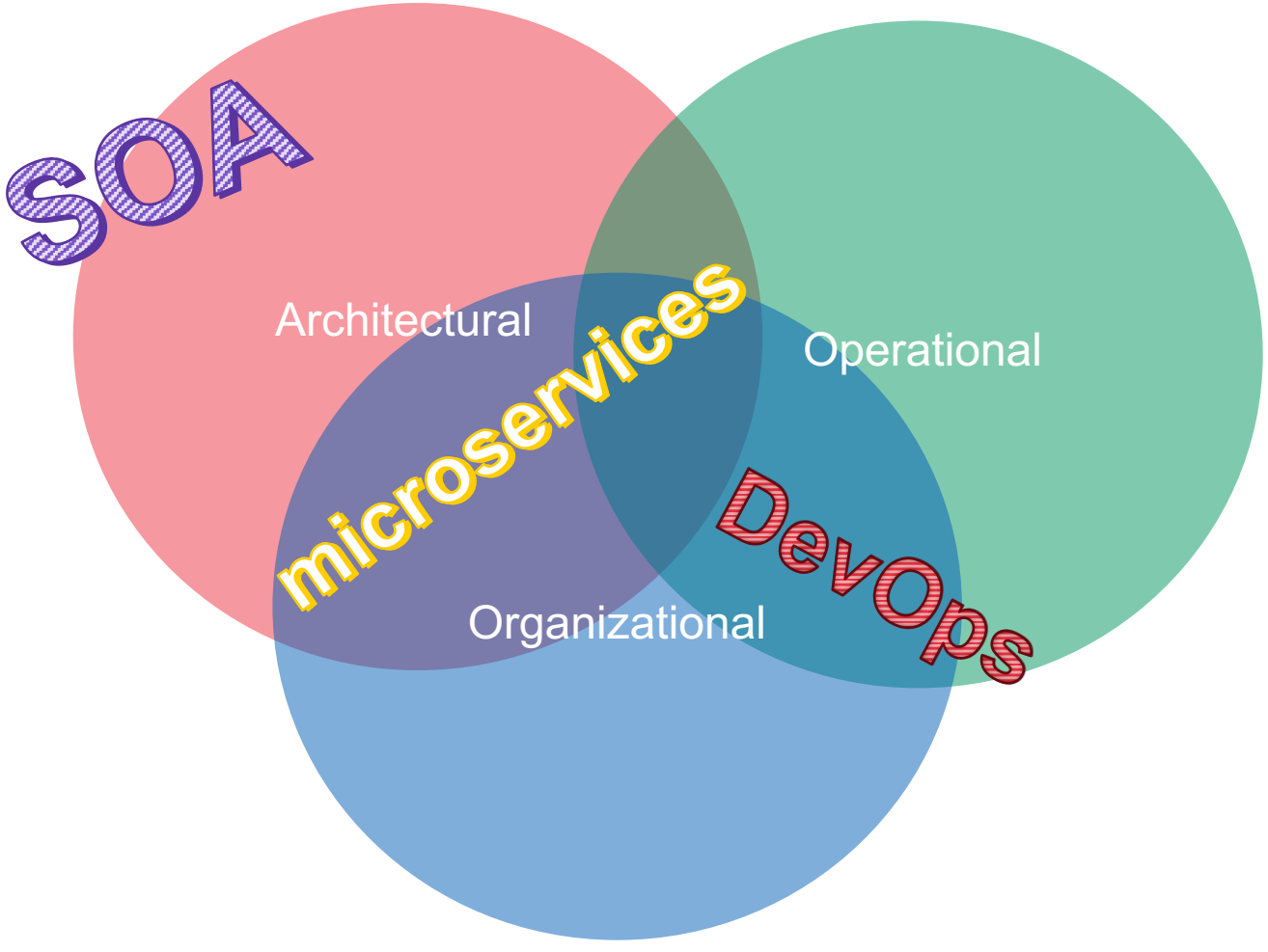
- **“Architecture astronauts” often overlook practical but important concerns**
  - Logging, tracing and monitoring
  - Edge cases such as cold restarts, bad nodes
  - Deployments and resource scaling
- **Operational and implementation patterns**
  - Logging sidecars, external services, distributed tracing
  - Blue/green deployments, gradual rollouts
  - Testing live systems

- **This is a practical / operational viewpoint**

Microservices as organizational structure

- **Conway's law**
  - "organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations."
  - Define system by organization, or organize by system design
- **Two-pizza rule for team size (Bezos)**
  - Minimize friction on internal communication
- **Formalize external interfaces**
  - Service contracts, SLAs → DevOps

- **This is a management viewpoint**


SOA
Architectural
Operational
microservices
DevOps
Organizational

Microservices are not systems

- **Systems comprise of multiple services**
  - This was true even decades ago, nothing new about microservices
- **Often multi-faceted**
  - Serving different types of users, different workloads
  - Overall goal is to support an organization's goals (business, academic, ..)

Why microservices?

Pros of microservices

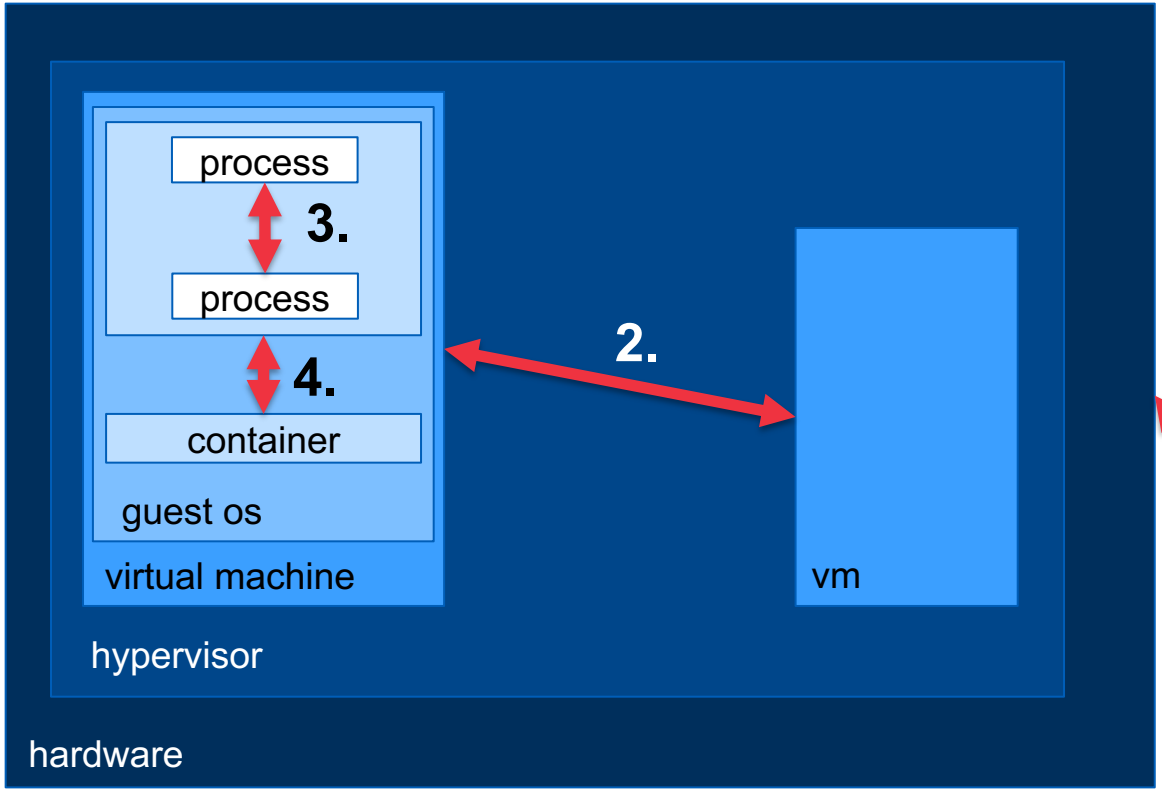
- **Helps managing large development organizations**
  - Clearer responsibilities, divisions of labor
  - Easier to scale at team and individual level
- **Increases development velocity**
  - Independent decisions in teams, formal dependencies
  - Intra-team communications more focused
- **“Product” viewpoint (vs. “project”)**
  - Easier to focus on customer needs than managing schedules

Cons of microservices

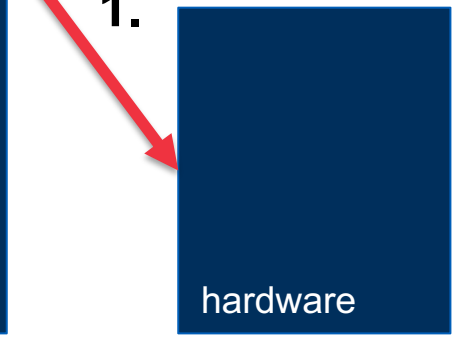
- **Increases development overhead**
  - Repetition of code, configuration etc.
  - In practice, requires investment in automation (CI/CD)
  - Debugging distributed systems notoriously difficult
- **Changes usage patterns and increases operational risks**
  - Distributed services put more load on the network (vs. local IPC)
  - Authority on infrastructure open to misuse and accidents
  - Security harder to monitor and enforce
- **Dependencies between services**
  - Configuration management and versioning require effort
  - Increased number of services leads to lower availability, higher variance of many service level metrics

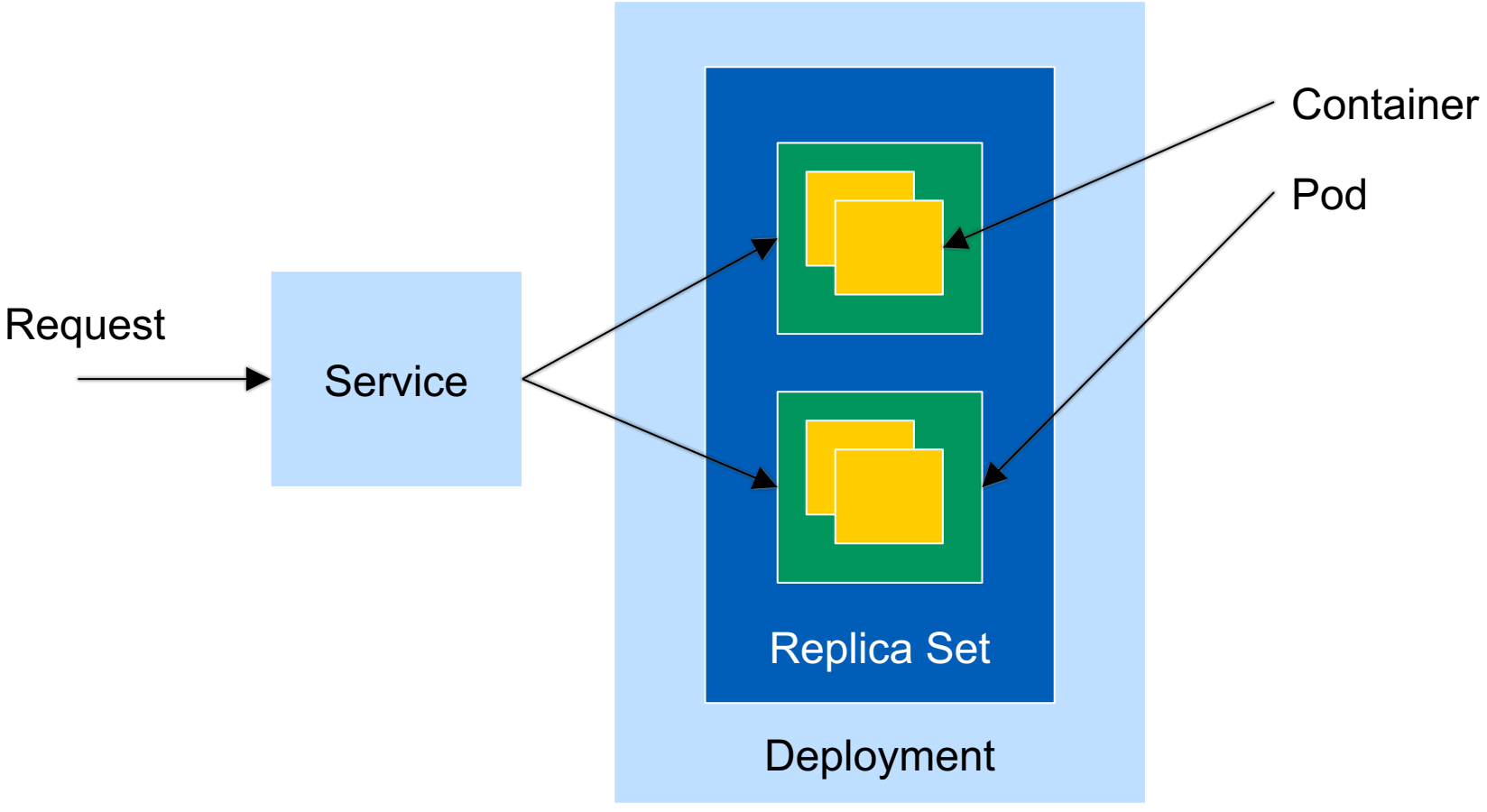
Containers: Docker and Kubernetes

Aalto University
School of Electrical
Engineering



1. Hardware isolation
2. Virtual machine isolation
3. Process isolation
4. Container isolation




Container
Pod
Request
Service
Replica Set
Deployment

Summary

- **Docker is a container build and execution framework**
  - Manages networking, volume mounts, registry push/pull, persistent container state, etc.
- **Docker's boundary is a single container**
  - No service orchestration in docker itself (yes in docker compose, but that's a separate solution)
- **Kubernetes widely used for container orchestration**
  - Manages pods, which can consist of multiple containers, and services which are exposed network ports and/or addresses

Microservices: Architecture

Aalto University
School of Electrical
Engineering

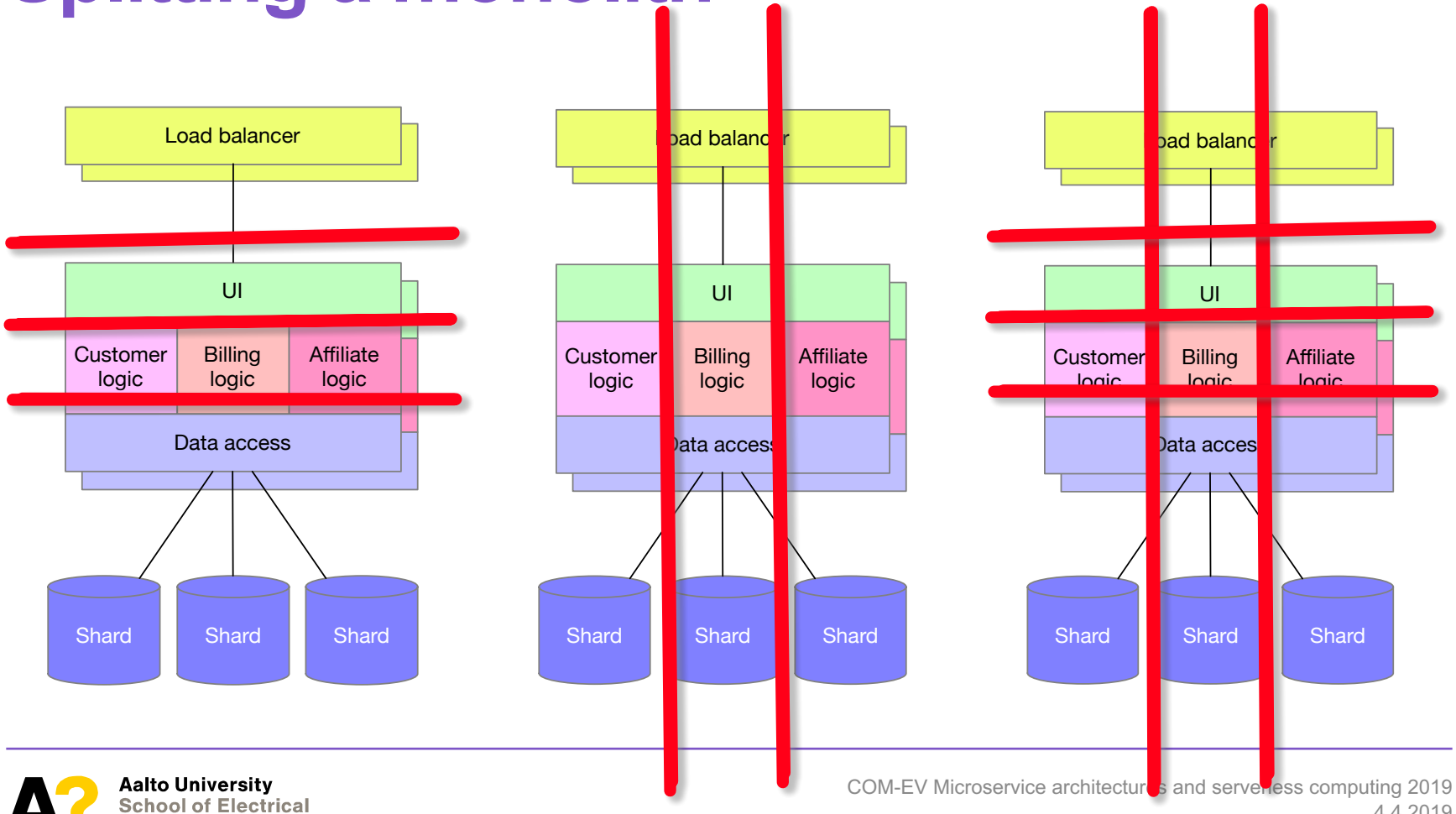
Service Orientation: The Idea

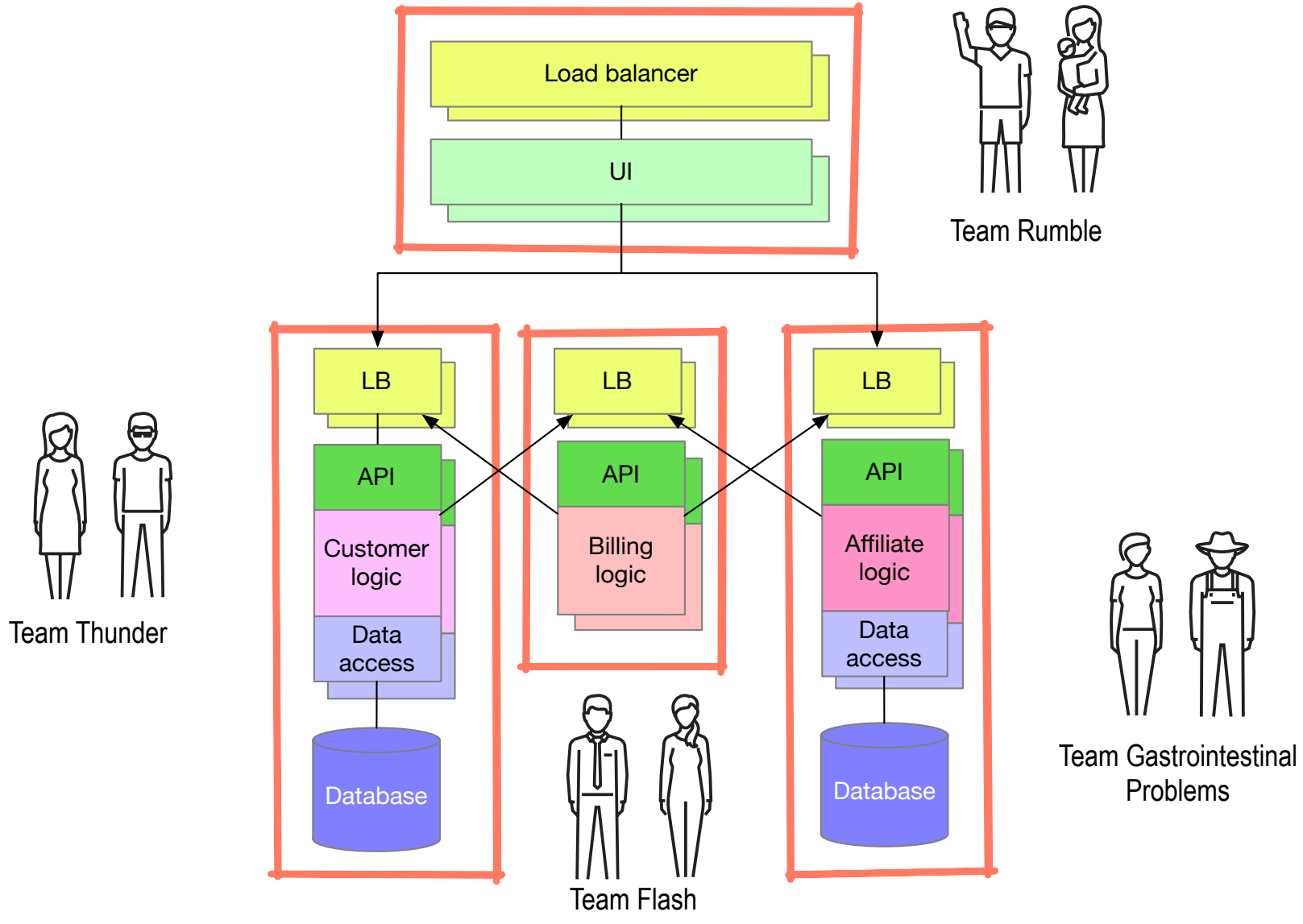
- **Define services by a logical boundaries**
  - Each service responsible for anything “inside”
  - Interaction via well-defined interfaces (APIs or other)

  - “Separation of concerns”

- **How to define a service boundary?**

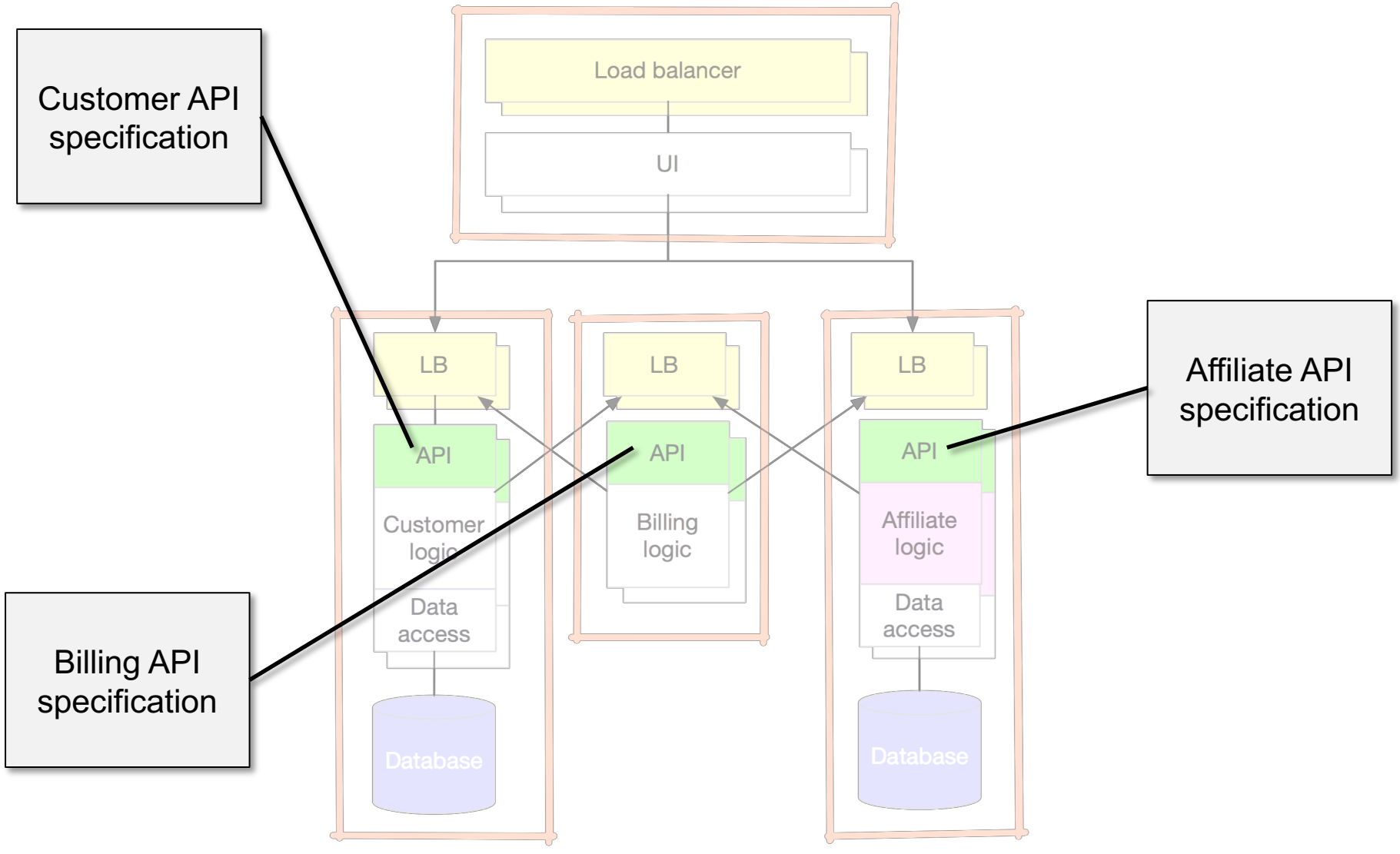
Splitting a monolith




Load balancer
UI
Team Rumble
LB
LB
LB
API
API
API
Customer logic
Billing logic
Affiliate logic
Data access
Data access
Database
Database
Team Thunder
Team Flash
Team Gastrointestinal Problems

Service description: Interface

- **How does a microservice interact with another?**
  - Service interface description
  - May be anything that allows control and data transfer
    - *REST is a practical default*
  - Formal interface definitions: Interface Definition Languages
    - *(WSDL and SOAP)*
    - *OpenAPI and Swagger for REST*
    - *gRPC and Thrift inherently IDL protocols*


Customer API specification
Billing API specification
Affiliate API specification
Load balancer
UI
LB
LB
LB
API
API
API
Customer logic
Billing logic
Affiliate logic
Data access
Data access
Database
Database

Service description: SLAs

- **Interfaces do not tell about non-functional requirements**
  - Availability
  - Reliability
  - Response time (distribution)
  - Security guarantees
  - Interface stability (obsolescence)

- **These are often highly situational**

The Big Kahuna

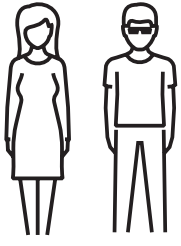
1M users
100k daily
99.9%
availability

Hi Thunder!
We need
99.95% and
800 r/s peak

Team Rumble

Hi old farts!
We need
99.5% and
30 req/s
sustained

Yep Rumble,
can do but it'll
cost you 200
donuts/day

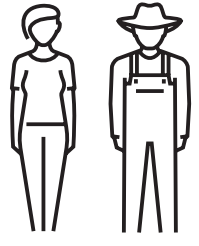


Team Thunder

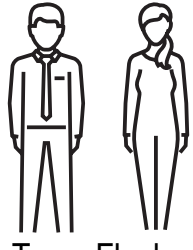
grumble grumble
Only if you can submit
requests using JCL

Either 99% at 50ms
or 99.99% at 5s?

Pssst Flash, what
can you do for 500
donuts/year?



Team Gastrointestinal
Problems



Team Flash

Network communication

Aalto University
School of Electrical
Engineering

Application protocols

- **Almost all service interactions occur at application level protocols**
  - HTTP and HTTPS primary (QUIC in the future?)
    - *HTTP(S) used to transport other application level protocols*
    - *SOAP, REST, …*
  - gRPC, Thrift, AQMP, etc.
- **Operate on top of TCP**
  - Sometime work around TCP issues (such as slow start, with Keep-Alive connections)
  - TCP is connection-oriented: connect → transmit → close
  - Usually client-server, e.g. specific listener address and port

Communication models

- **Synchronous response**
  - Request-response pattern
  - Reply expected immediately (after processing)
- **Asynchronous response**
  - Processing started by request
  - Immediate response provides a handle or identifier
  - Response methods
    - *Polling by client (known endpoint or part of response)*
    - *Callback from server (agreed-upon endpoint or part of request)*
    - *Response publish (message queue, pubsub, blackboard, …)*
- **Message-passing**
  - Request itself asynchronous

Failures

Failures in distributed systems

- **Rule of thumb:**
  - Everything fails all the time (randomly, when least expected)
- **See *Network is reliable* paper (hint: it is not)**

- **Microservice architectures fail more**
  - More components, more computers, more connections, more changes, more of everything
  - Risks of correlated failures can be either higher or lower than for monolithic systems
  - See first lecture slide how number of components affects reliability

Brewer's theorem's consequences

- **Hard partitions are generally rare**
  - Most of the time it is possible to achieve both consistency and availability
- **However, partitions do still occur**
  - Then you need to choose between availability and consistency
  - "Eventually consistent" mechanisms choose availability

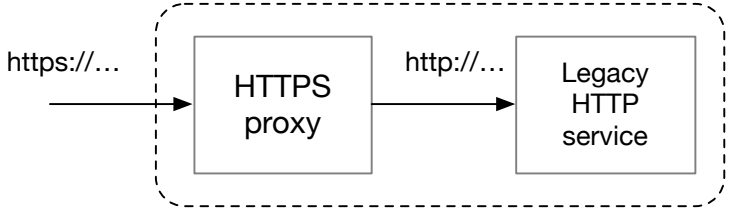
- **In large enough systems, something fails all the time**
- **Consideration in services — which is critical?**

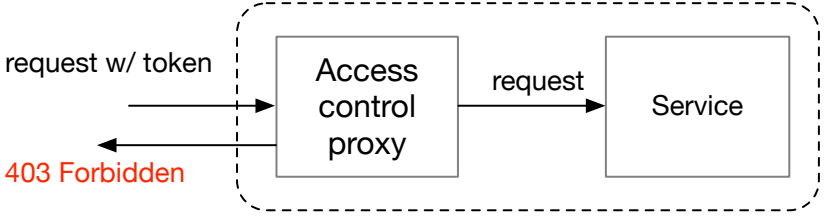
Single-node patterns

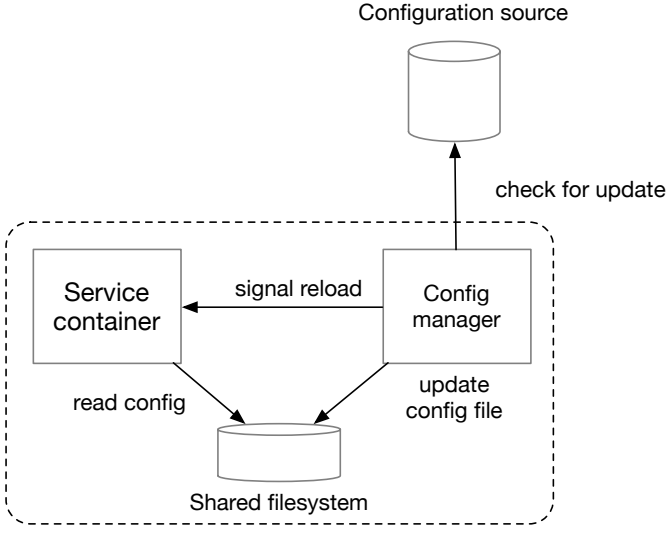
Aalto University
School of Electrical
Engineering

Sidecar examples

- **Adding HTTPS to legacy application**
- **Updating configuration**
- **Access control**

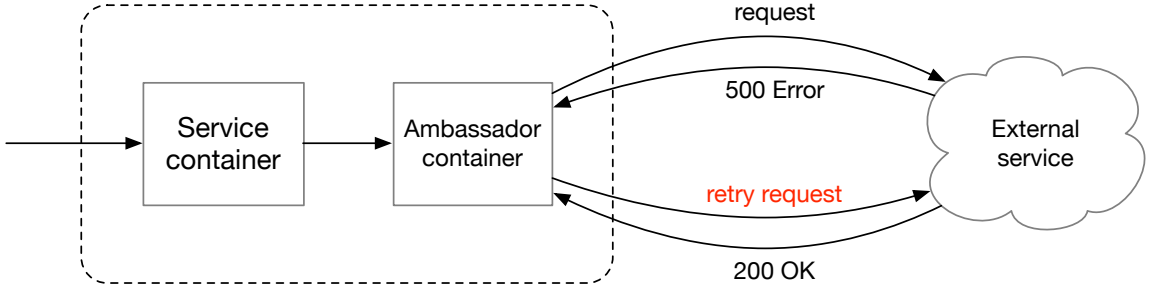


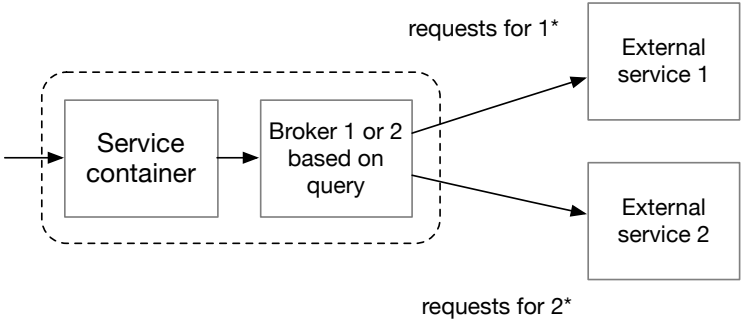


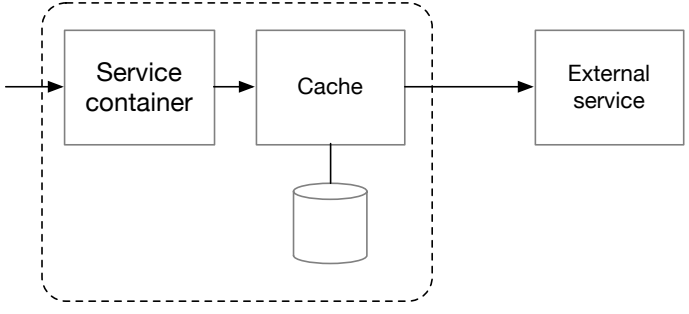


Ambassador examples

- Hiding 500s
- Service brokering
- Local caching

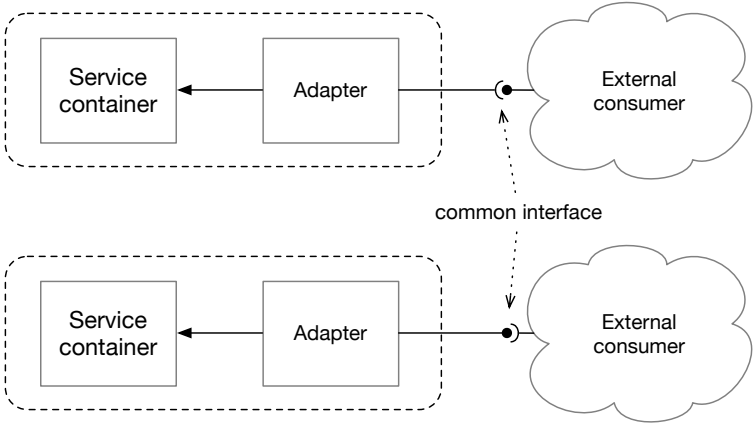






Adapter

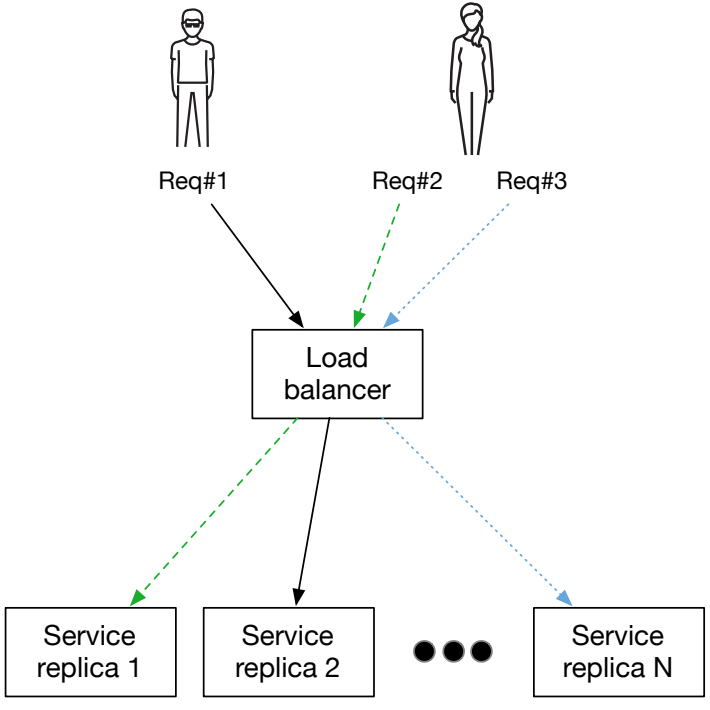
- **Sidecar pattern when someone else needs a specific interface**
  - Common interface used across the system such as logging, metrics, service health etc.
  - Not “core” service but supporting interfaces
- **Both push and pull interfaces**



Extending to multiple nodes

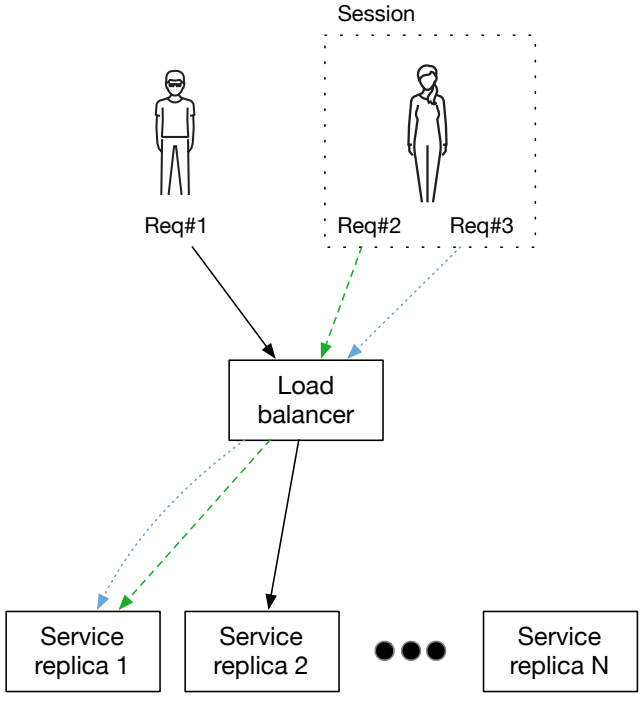
Load-balanced services

- **Multiple identical stateless services**
  - Send requests according to some policy (RR, random, LRU, ...)
  - Service is replicated, functionally identical portions duplicated



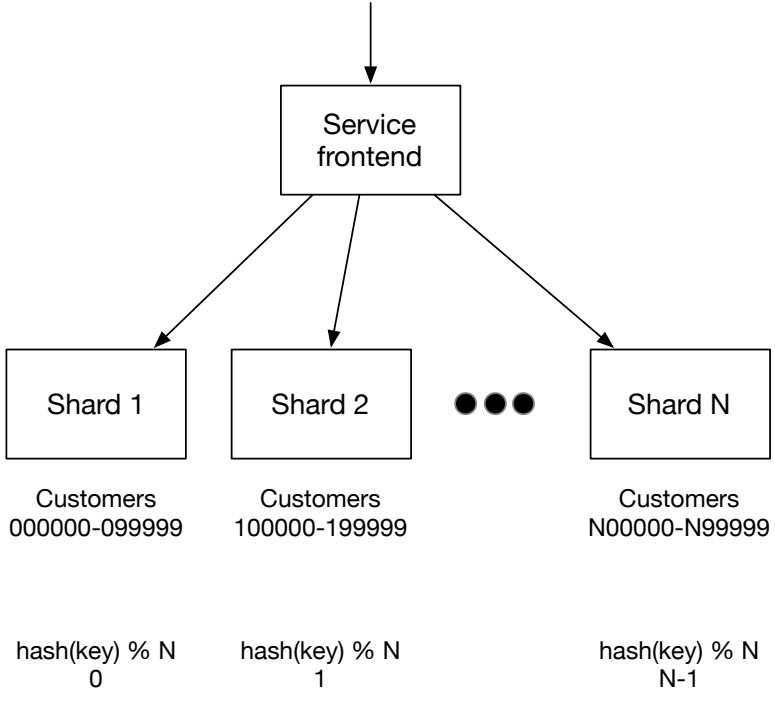
Load-balanced services

- **Multiple identical <u>stateful</u> services**
  - Identify a session key
  - Send request to backend identified by the session key
  - If not identified, use some policy (like before)
- **Problems**
  - Hot replica
  - Key redistribution

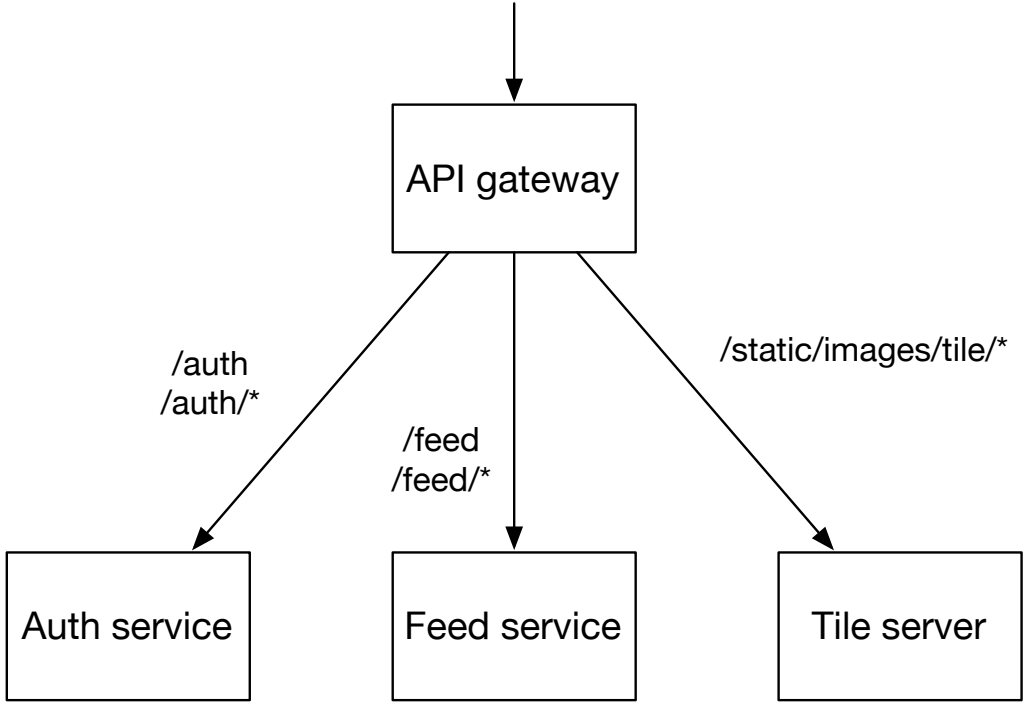


Sharding

- **Distribute requests to specific backend**
  - Use sharding function mapping a sharding key to shard index
  - Non-sequential keys hashed
  - Consistent sharding functions (why modulo is not?)



Service brokering



Asynchronous processing

Aalto University
School of Electrical
Engineering

“Workflow system”

- **System that orchestrates a flow of work**
  - Potentially across different systems (e.g. always in microservice architectures)

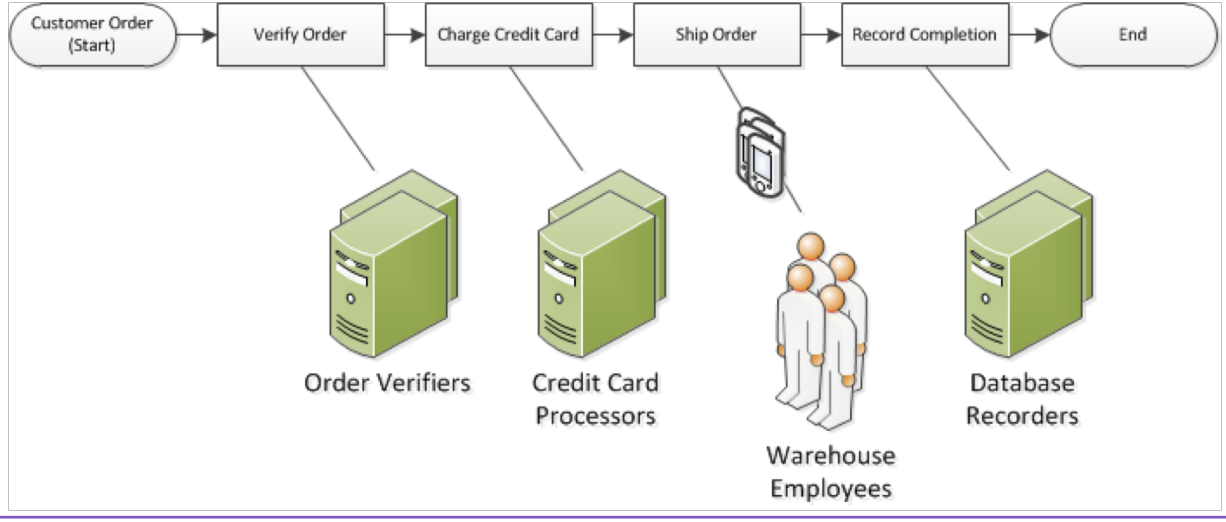


Image: AWS

Batch vs. stream processing

	Batch processing	Stream processing
Data scope	Queries or processing over all or most of the data in the dataset.	Queries or processing over data within a rolling time window, or on just the most recent data record.
Data size	Large batches of data.	Individual records or micro batches consisting of a few records.
Performance	Latencies in minutes to hours.	Requires latency in the order of seconds or milliseconds.
Analyses	Complex analytics.	Simple response functions, aggregates, and rolling metrics.

Source: AWS

Examples:

- Log ingestion
- Device sensors
- User interactions (game, website, mobile app, ...)
- News / social media feeds

Messaging

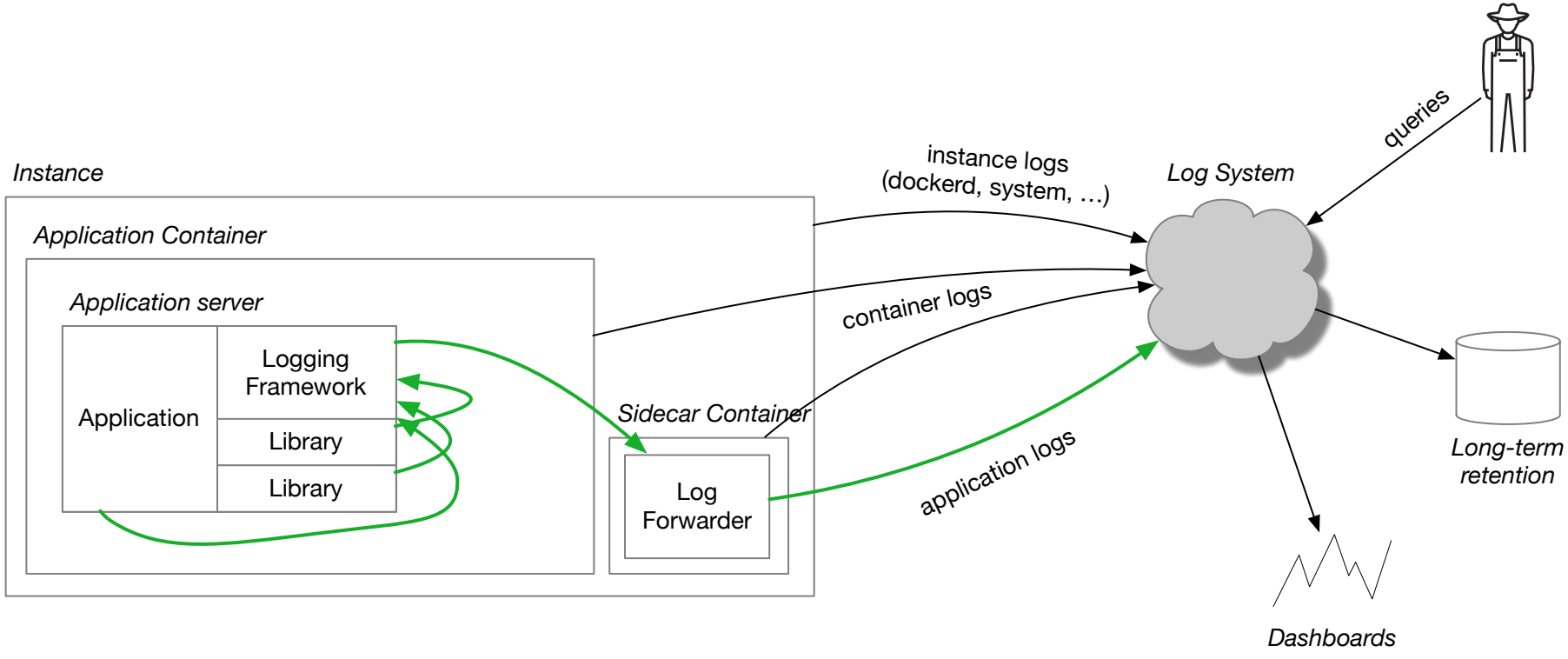
- **Messaging is exchange of asynchronous messages via a 3rd party**
  - Message queues: unordered / FIFOs, single message (1-1)
  - Publish/Subscribe (PubSub): Message fanout 1-N
  - Message bus: PubSub, but goes much into ESBs …
  - Specialized systems (Celery – task queue, e.g. asynchronous RPC, message priorities, …)
- **Lots of OSS and commercial solutions**
  - AWS SQS (FIFO) & SNS (PubSub), Apache ActiveMQ, RabbitMQ, … (lots and lots), also can use databases

Why asynchronous models?

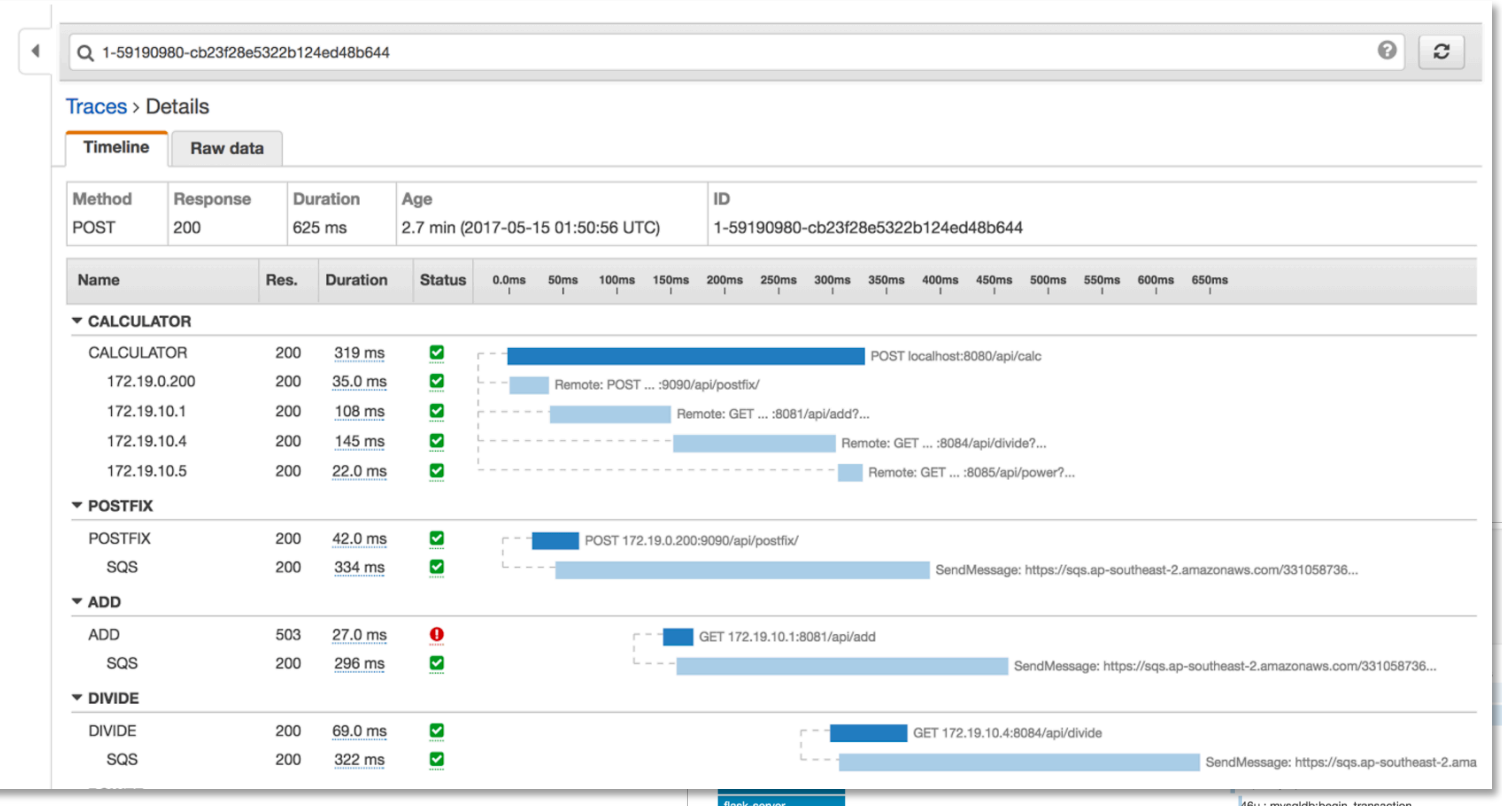
- **Splitting a big task to smaller, sequential pieces**
  - Easier to develop and debug each in isolation
  - Natural for microservice architectures to create service boundaries
- **Less prone to failures, easier to recover**
  - Management can be made HA and resilient
  - State transitions ~idempotent → no (big) problem re-running
- **Less sensitive to processing delays and load variations**
  - Not in path of synchronous processing (order fulfilment ~ days!)
  - Buffering, capacity scaling
- **Many business processes <u>are</u> workflow processes!**

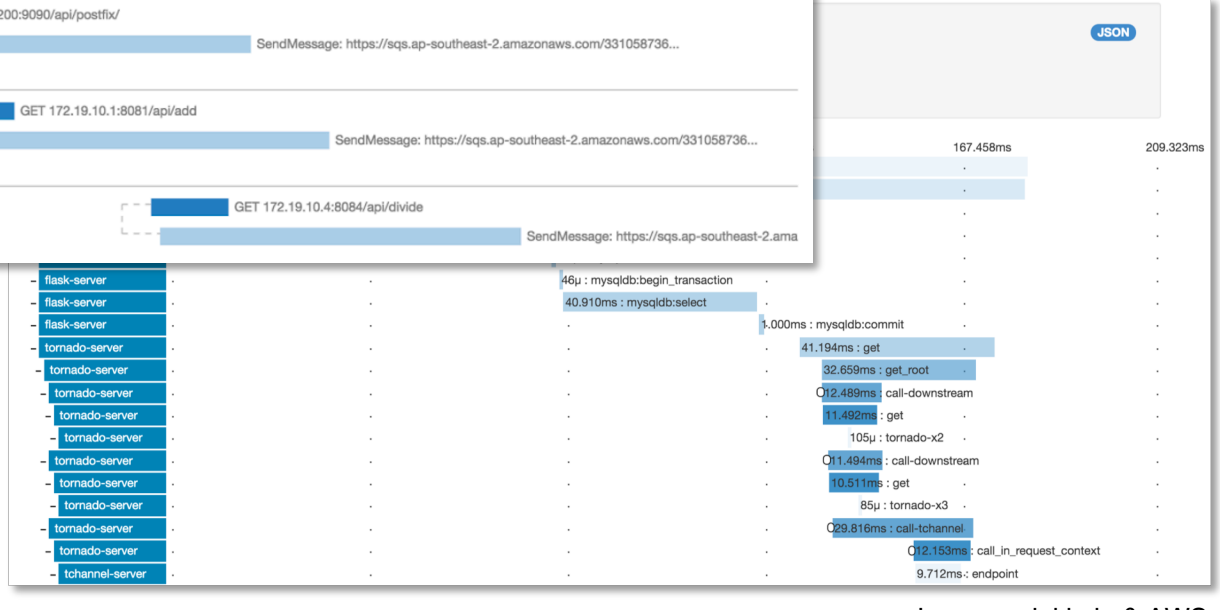
Logging, metrics and tracing

Aalto University
School of Electrical
Engineering


Instance
Application Container
Application server
Application
Logging Framework
Library
Library
Sidecar Container
Log Forwarder
instance logs (dockerd, system, ...)
container logs
application logs
Log System
queries
Long-term retention
Dashboards

Images: Elastic & NBSoftSolutions





Images: zipkin.io & AWS

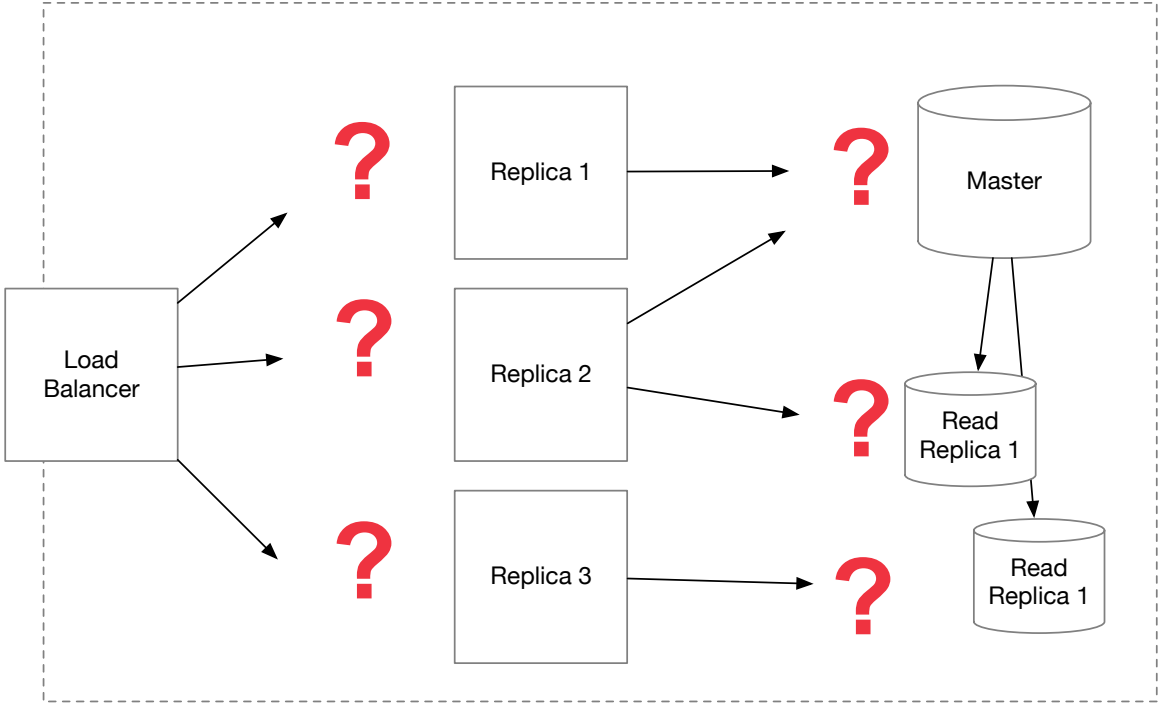
Summary

- **Which one you prefer:**
  - System you KNOW is hosed?
  - System which APPEARS to work?
- **Logging, metrics and tracing are tools for the FIRST one**
  - Identifying the problem **metrics**
  - Locating the problem **logging (tracing)**
  - Understanding the problem **logging**
  - After fix is rolled out, verifying that problem has gone away **all**

Service Discovery

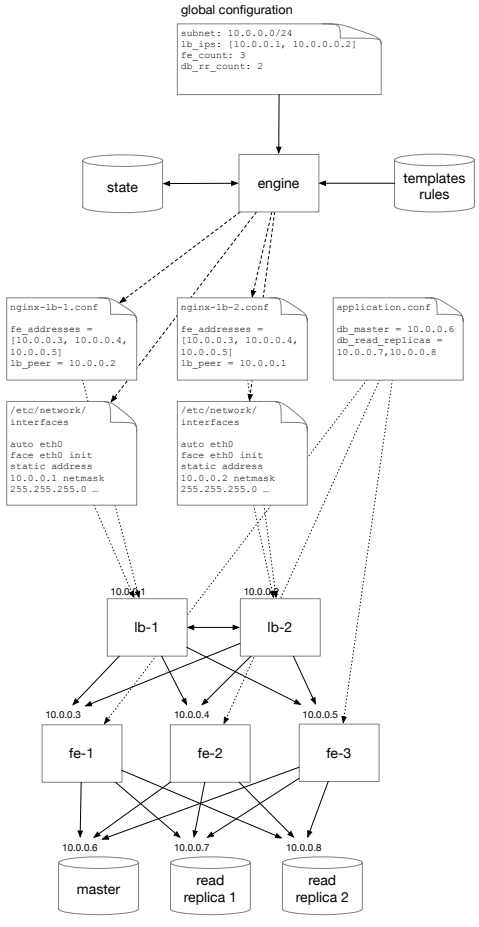
Aalto University
School of Electrical
Engineering

Independent components



Service injection

- **Extremely static case of discovery**
- **<u>Everything</u> is known at deployment**
  - IP addresses
  - Number of nodes in cluster
- **Methods**
  - Configuration templates
  - Environmental variables



Host-based discovery

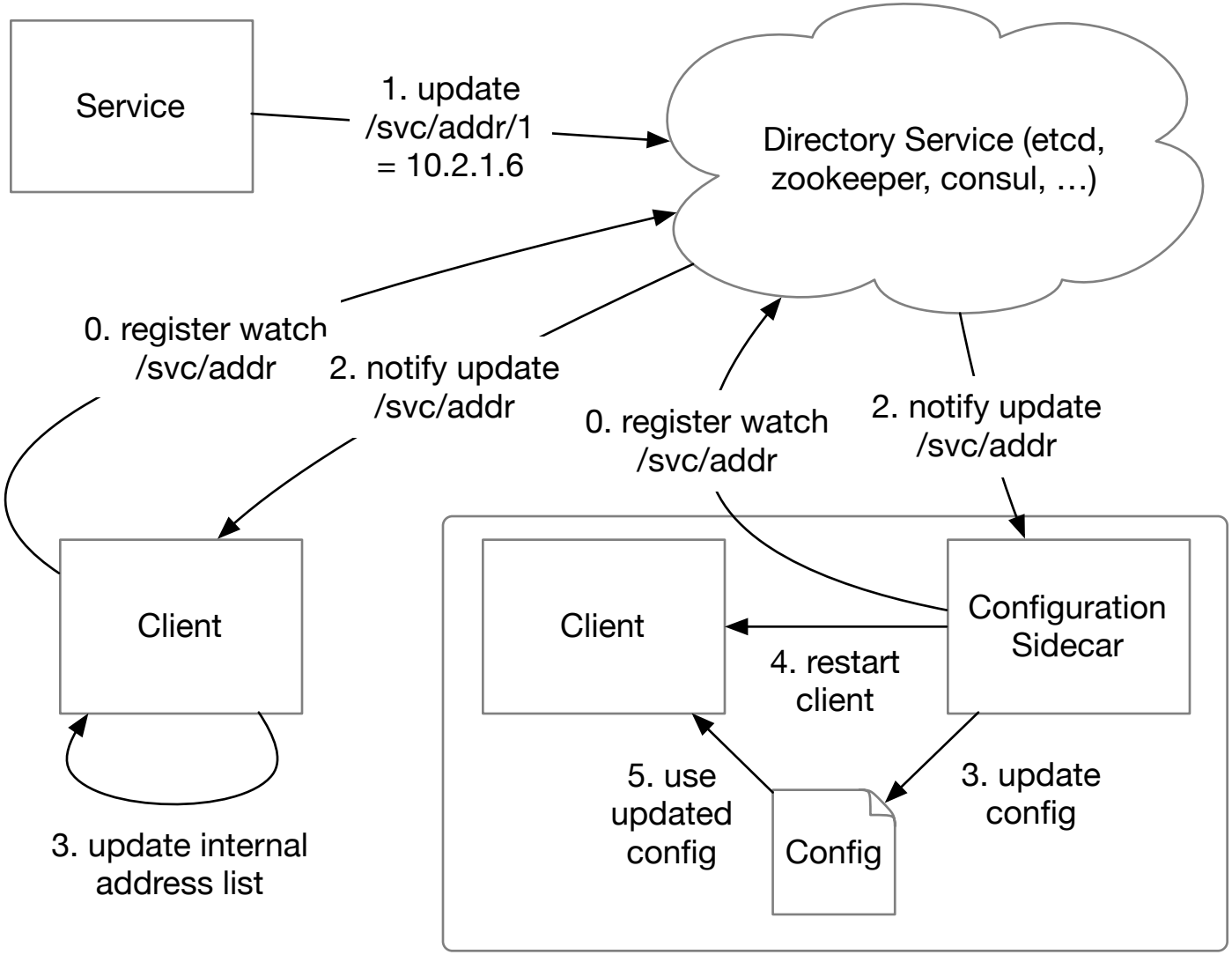
- **Idea: Distributed services over network**
  ➔ DNS built-in to almost everywhere ➔ why not use it?
- **Host-based discovery**
  - /etc/hosts (static = old, since dynamic mounts or rewriting)
  - Local DNS resolver
  - Cluster DNS
  - Integrated service discovery service with DNS

More service discovery patterns

- **Host-based via DNS easy to use**
  - Might require client-side understanding (multiple records)
  - Difficult to generalize to other uses (queue names etc.)
  - (Ports not so much a problem with private IPs and port remapping)

➔ **Generalized directory services**

  - etcd, ZooKeeper, Consul …
  - Requires client-side support: external configurator (sidecar?) or internal to application (integrate service client)
  - Users have complete control over key and value semantics


Service
1. update
/svc/addr/1
= 10.2.1.6
Directory Service (etcd,
zookeeper, consul, …)
0. register watch
/svc/addr
2. notify update
/svc/addr
0. register watch
/svc/addr
2. notify update
/svc/addr
Client
3. update internal
address list
Client
Configuration
Sidecar
4. restart
client
5. use
updated
config
Config
3. update
config

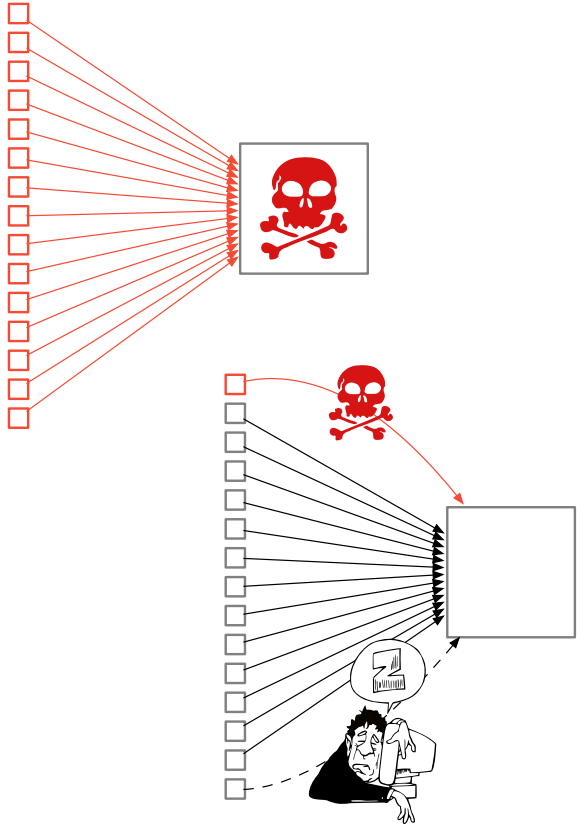
Failures

Overview

- **Already established that in a distributed system**
  - Services may fail at any time
  - Network may fail at any time
  - Services may delay response arbitrarily
  - Network may hang up arbitrarily
  - Services may produce unexpected responses for any request
  - (Client may fail at any time too, but let's ignore that for now)
- **What can we do about that?**

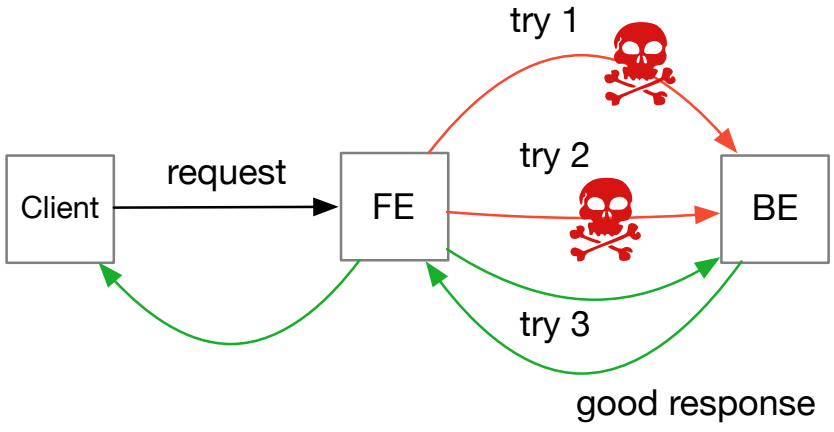
Failure types

- **Failure types (roughly)**
  - Server dies (application server crashes, server crashes)
  - Request fails (connection terminated, 500 Server Error, incorrect response, corrupted response)
  - Request hangs (response not completed)
- **May occur in combination**
  - Server dies → may look as a hang to client
  - Server dies → may result in 500 from proxy or a connection termination
  - Request hangs → eventual error response from timeout in proxy



Transparent retry

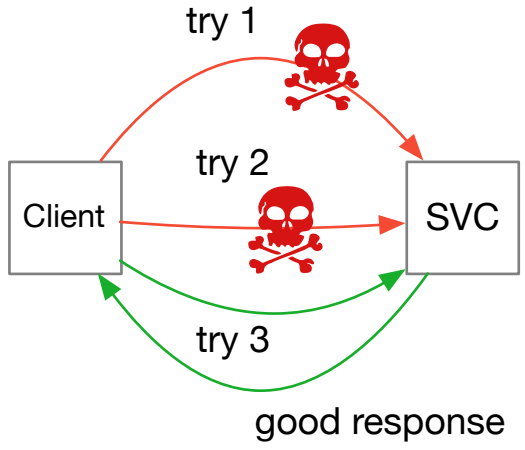
- **Incorporate retry logic at intermediary**
  - LB, reverse proxy, etc.
- **Most useful for transients**
- **Really only for idempotent requests**
  - GET or HEAD
- **Timeouts**
  - N tries with timeout T for each, max N * T seconds (N = 3, T = 10 → 30 seconds before definite failure)
- **Also useful for backoffs (429, 503)**
- **Does not help if clients aggressive**
  - RELOAD if >5s second page load

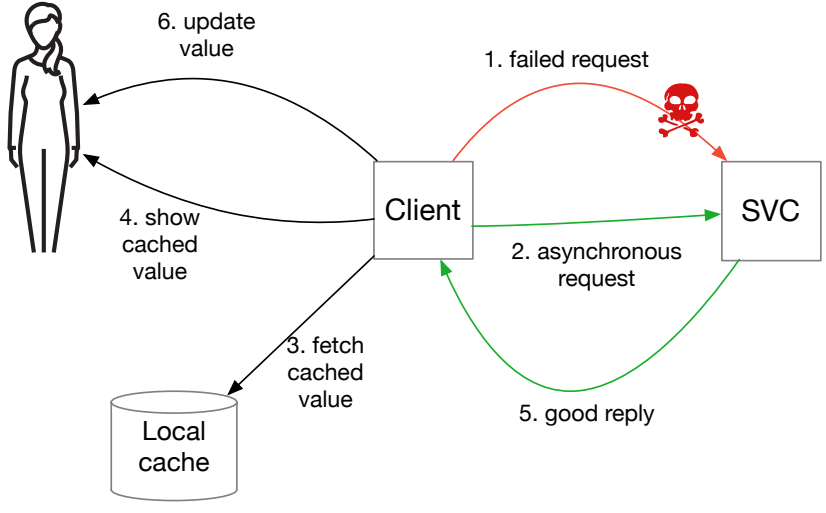


Note: BE may be multiple replicas, with different requests going to different nodes

Client-side retry

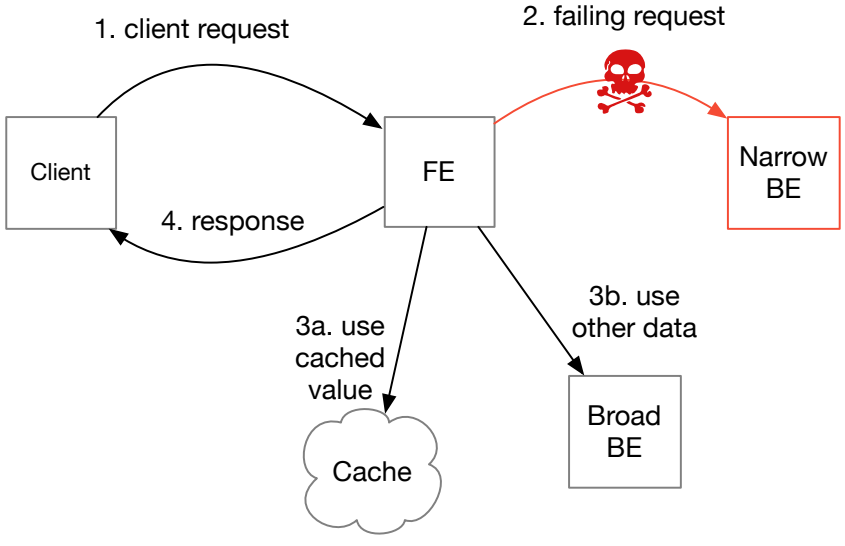
- **Return failures immediately to client**
  - Timeouts controlled by client
- **Client decides what to do**
  - Retry?
  - Use other service?
  - Use cached value?
  - Use cached value, but call asynchronously and update if successful?
  - Report error?





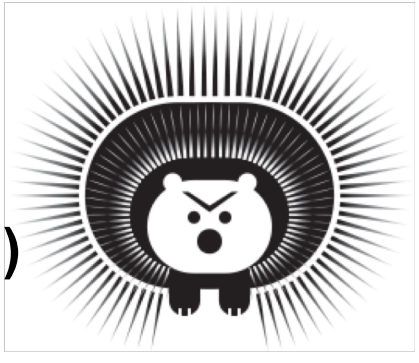
Fallbacks

- **Aspect of resilient computing**
  - Adaptive activities and responses based on environmental conditions
- **Use cached or other data**
  - "Old value" often better than "no value"
  - Broader data may be applicable too
    - *Per-user recommendations → general recommendations*
    - *Finland feed → Europe aggregate feed*
- **Applicable for all**
  - Services using other services, transparent proxies and client-side logic



Circuit breakers (software fuses)

- **Distributed system = many parties**
  - Share information about failures
  - Clients can react to failing services before using them
- **Circuit breaker**
  - Trip on failures
  - Use fallback if tripped
  - Some fuse reset policy
- **Hystrix! (originating from Netflix, where else?)**
  - Circuit breaker design pattern

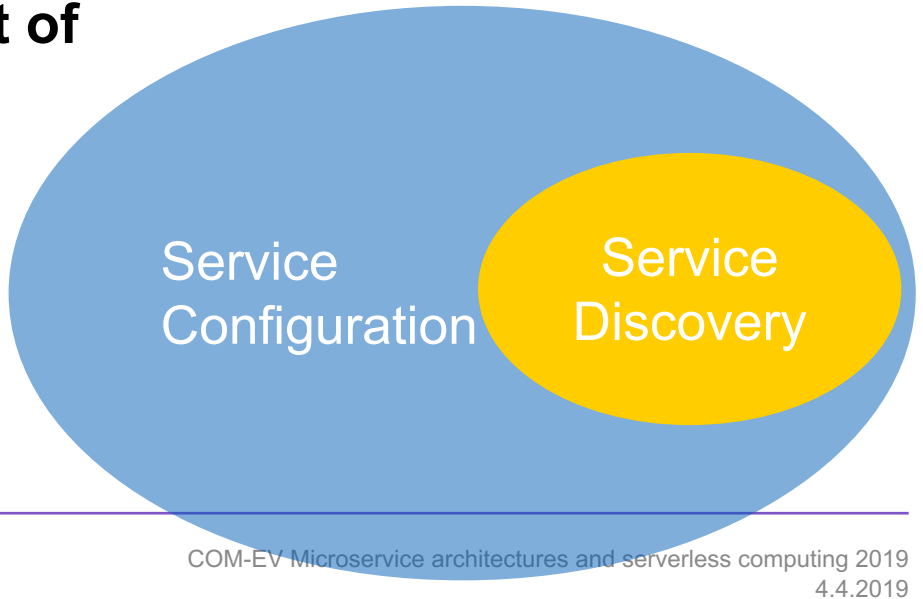


Service Configuration

Aalto University
School of Electrical
Engineering

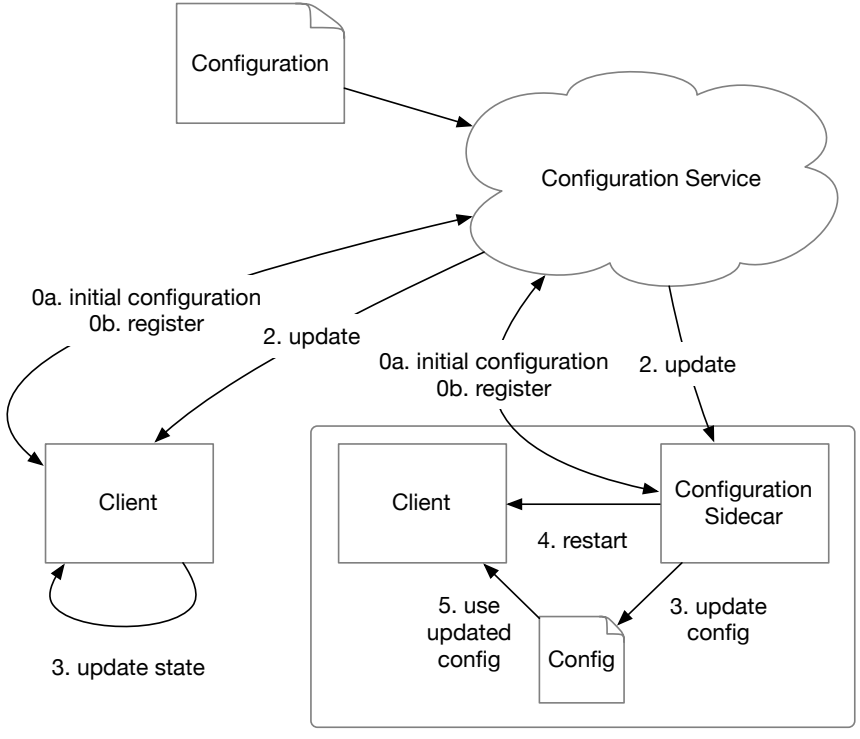
Previously …

- **Discussed service discovery**
  - How to “plumb” the pipes between services
  - Injection, host-based discovery, directory services
- **Discovery is just one aspect of <u>service configuration</u>**
  - E.g. not only about plumbing
  - Settings, secrets, …



Techniques pretty similar to discovery

- **Static configuration**
  - System deployment
  - Service start
- **Dynamic configuration**
  - Integrated into service
  - Sidecar managed



Dynamic configuration

- **Facebook and Google extreme examples**
  - Feature flags dynamically enable/disable functionality
    ```
    if (feature_x_enabled) { … } else { … }
    ```
  - Feature flags are dynamically configurable (via some directory)
  - Multivariate flags: on/off based on complex criteria
- **Potentially change large portions of service functionality without code changes or redeployment**
  - We'll come back to "dark launches" later on deployments
- **(Not without its own problems)**

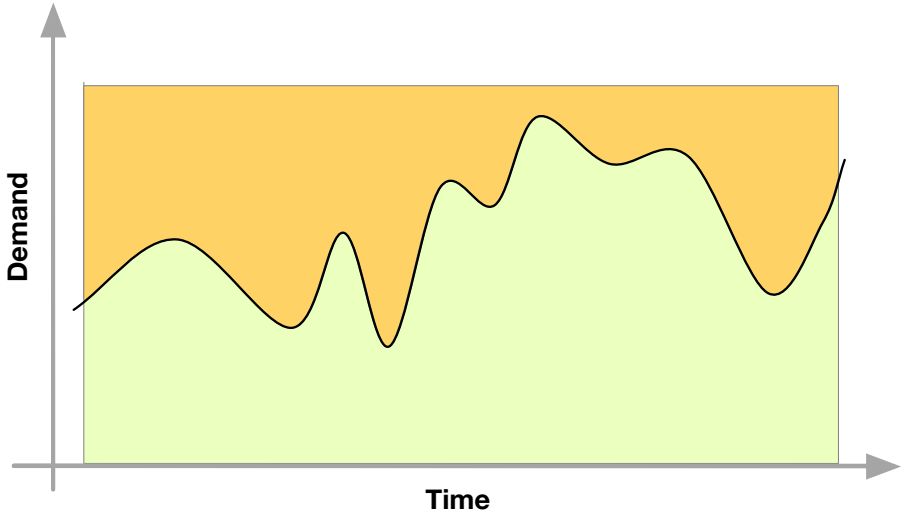
Secrets and sensitive information

- **What are ”secrets”?**
  - Cloud infrastructure and 3rd party service access keys
  - Keys used for HMAC and encryption (signed session token)
  - Passphrases for asymmetric cryptography private keys (e.g. TLS)
    - *For any other kind of keystore (Java, Bitcoin, …)*
  - On-disk encryption keys
- **“Secrets” are runtime information**
  - Should never go into actual service code or configuration
  - Injected only when service started, or pulled in as needed

Deployments

Scaling

- **Statically resourced systems applicable if**
  - Load pattern is predictable and not highly variable
- **Conversely many real-world problems don't fit this**
  - Daily variation (night / day)
  - Weekly variation (weekday / weekend)
  - Spikes and dips (black Friday, Christmas)
  - Long-term patterns (increased popularity, viral effects)
- **➔ Unused capacity ➔ $£€ lost**



Horizontal and vertical scaling

- **Vertical scaling (going up!): bigger box**
  - Increase instance size, increase disk allocation, ...
- **Horizontal scaling (going sideways!): more boxes**
  - Add 1 box ... add 1 box ... add 1 box ... repeat
- **Of course it is possible to use both simultaneously**

- **”Blast radius” describes area of impact of an failure**
  - “Larger instance” (vertical scaling) >> Lots of boxes
  - SPOF database’s blast radius is easily the whole system
    1-out-of-N stateful customer service affects 1/N customers

“Systems spend more of their life in operation than in development”

- M.T. Nygard, *Release It!*

General solutions

- **Stop-and-go deployment**
  - Stop the world
  - Update
  - Start the world
- **Service degradation**
  - Fallback services
  - Read-only mode
- **Non-stop deployments**
  - Blue-green deployments
  - Canaries etc.

- **Minimizing critical intervals**
  - Database techniques

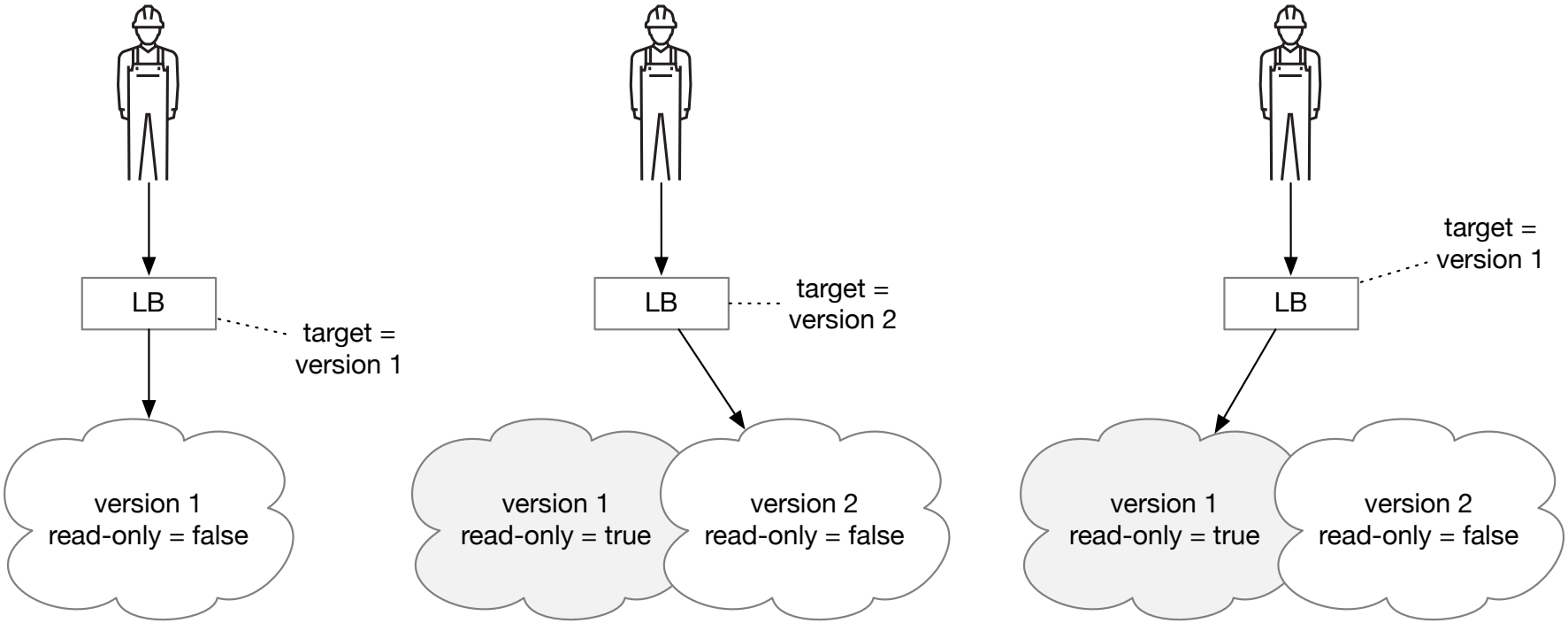
- **Minimizing affected users**
  - E.g. avoiding big bugs
  - Scientist
  - Multivariate feature flags

- **Later: Destructive changes**

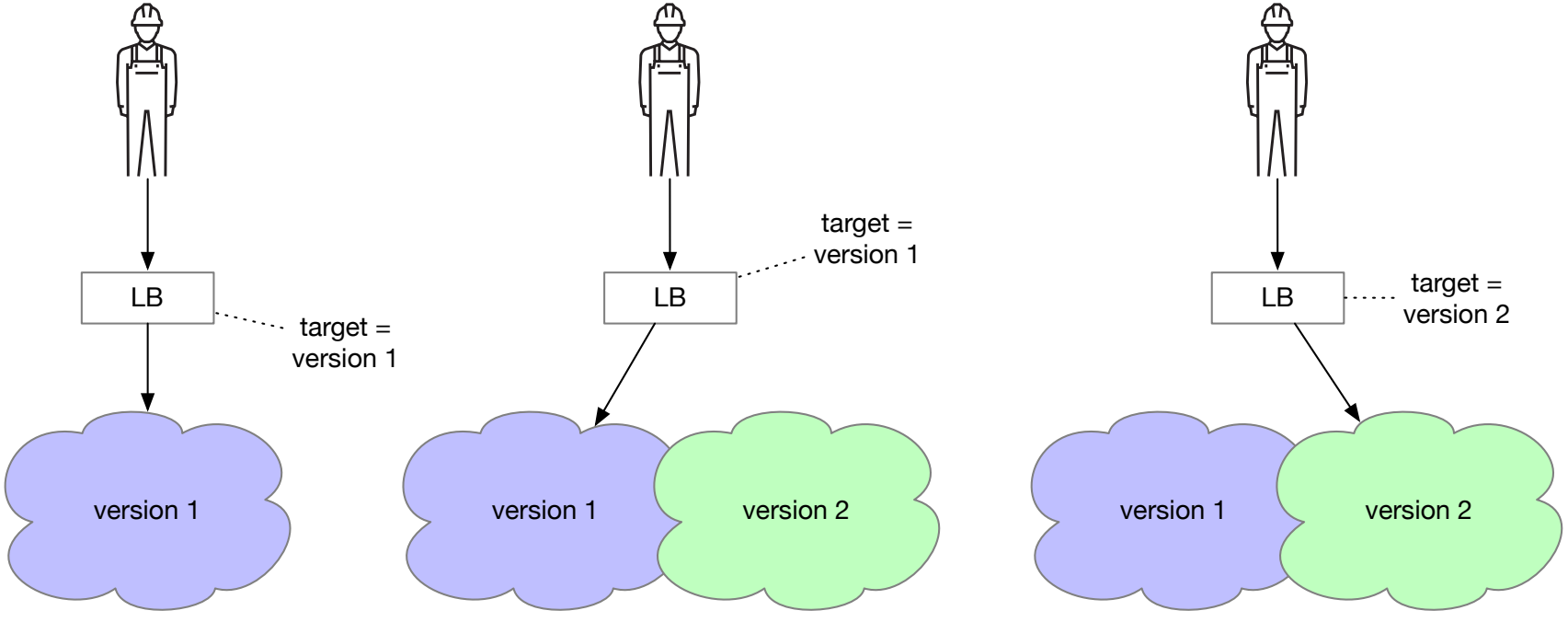
Stop-and-go deployment

- **Simplest**
  - Almost all problems during upgrade are related to state!
  - State in stable (not changing) state easiest to handle
- **All-or-nothing**
  - Difficult to test with small number of users (possible, but bad $)
  - Rollback affects also everything similarly (stop for rollback)
- **Any scripting tool with or without CI/CD works**
  - Shell scripts (used this with early EC2!)
  - Nowadays Puppet, Chef, Fabric, CloudFormation, Terraform, …
  - ”`kubectl delete -f old.yml; kubectl apply -f new.yml`”

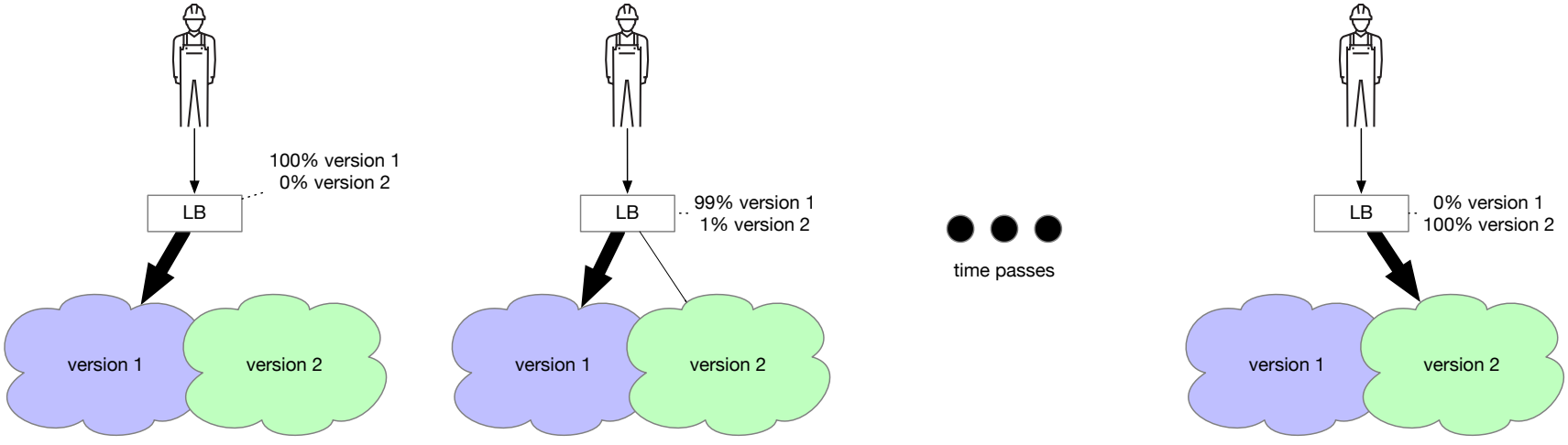
Read-only mode



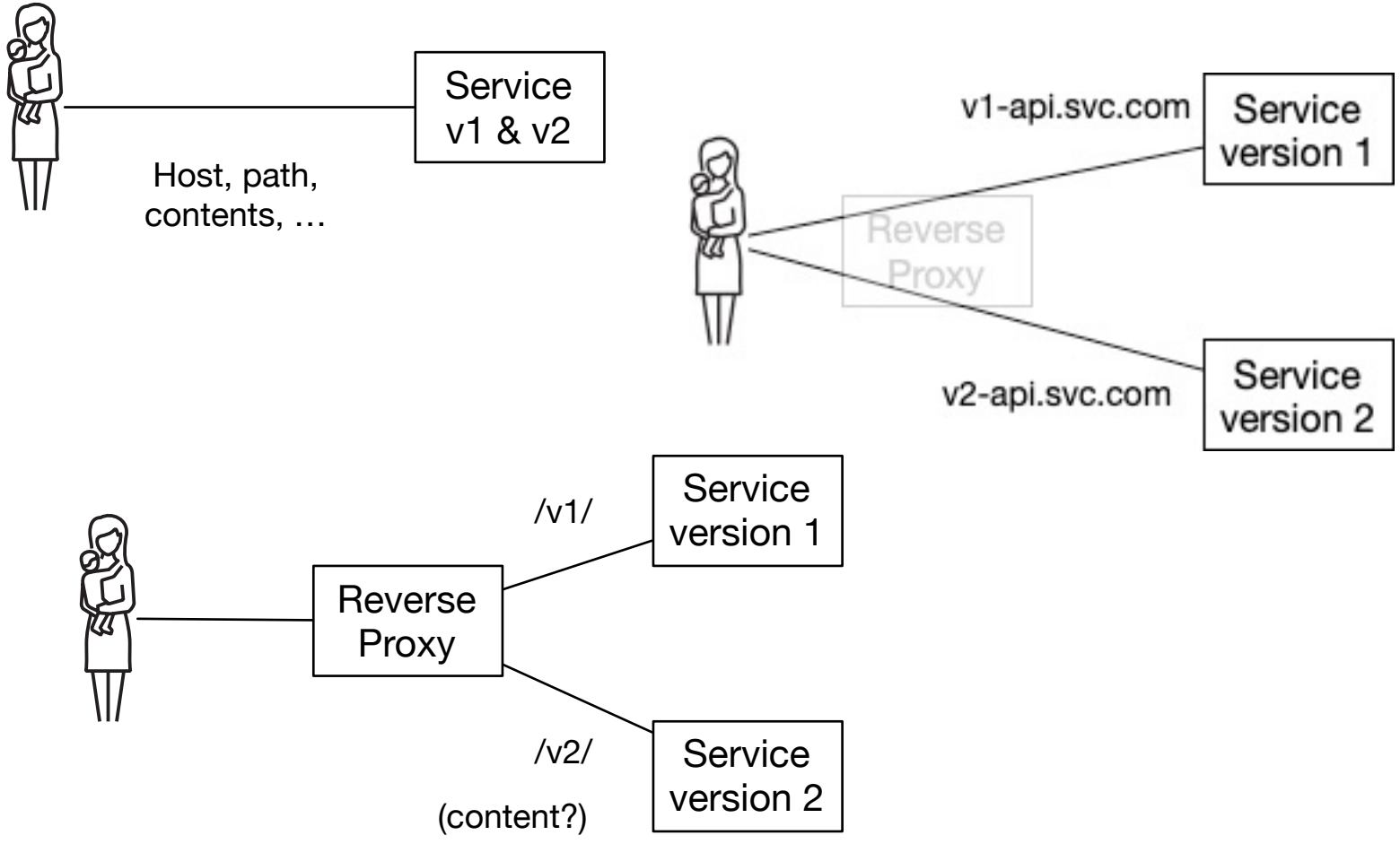
Blue-green deployment

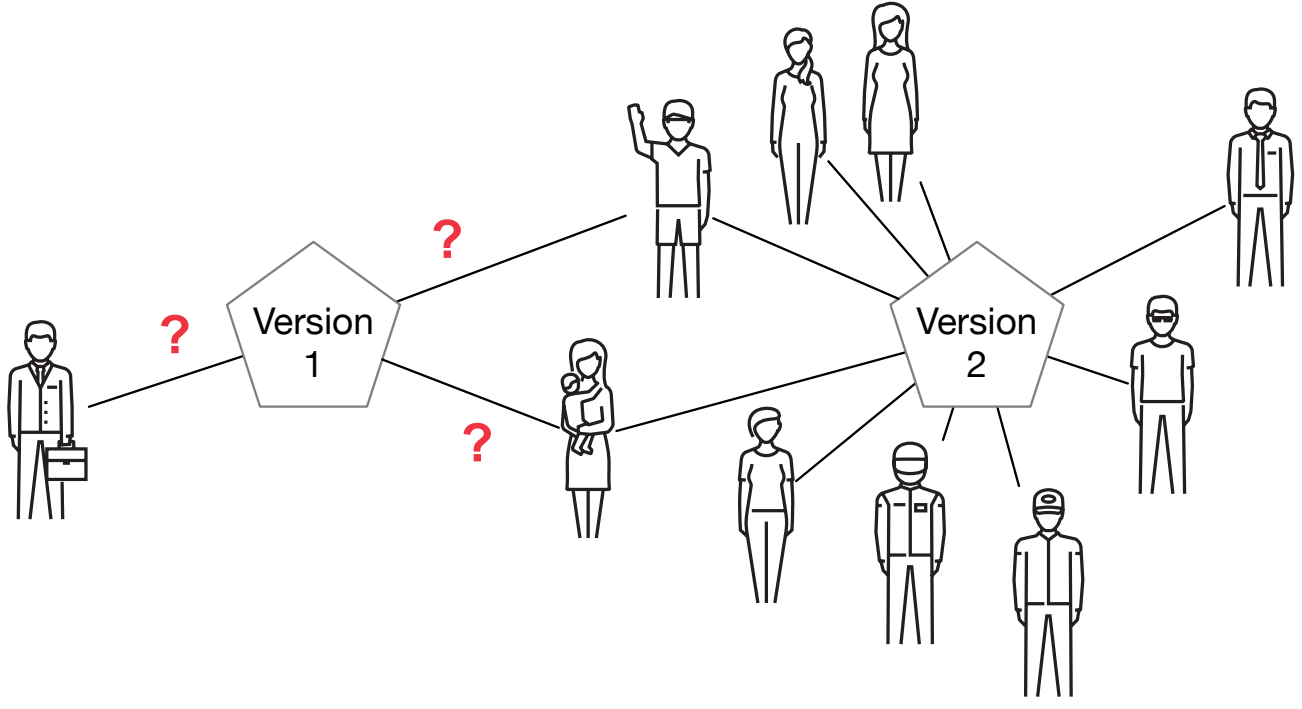


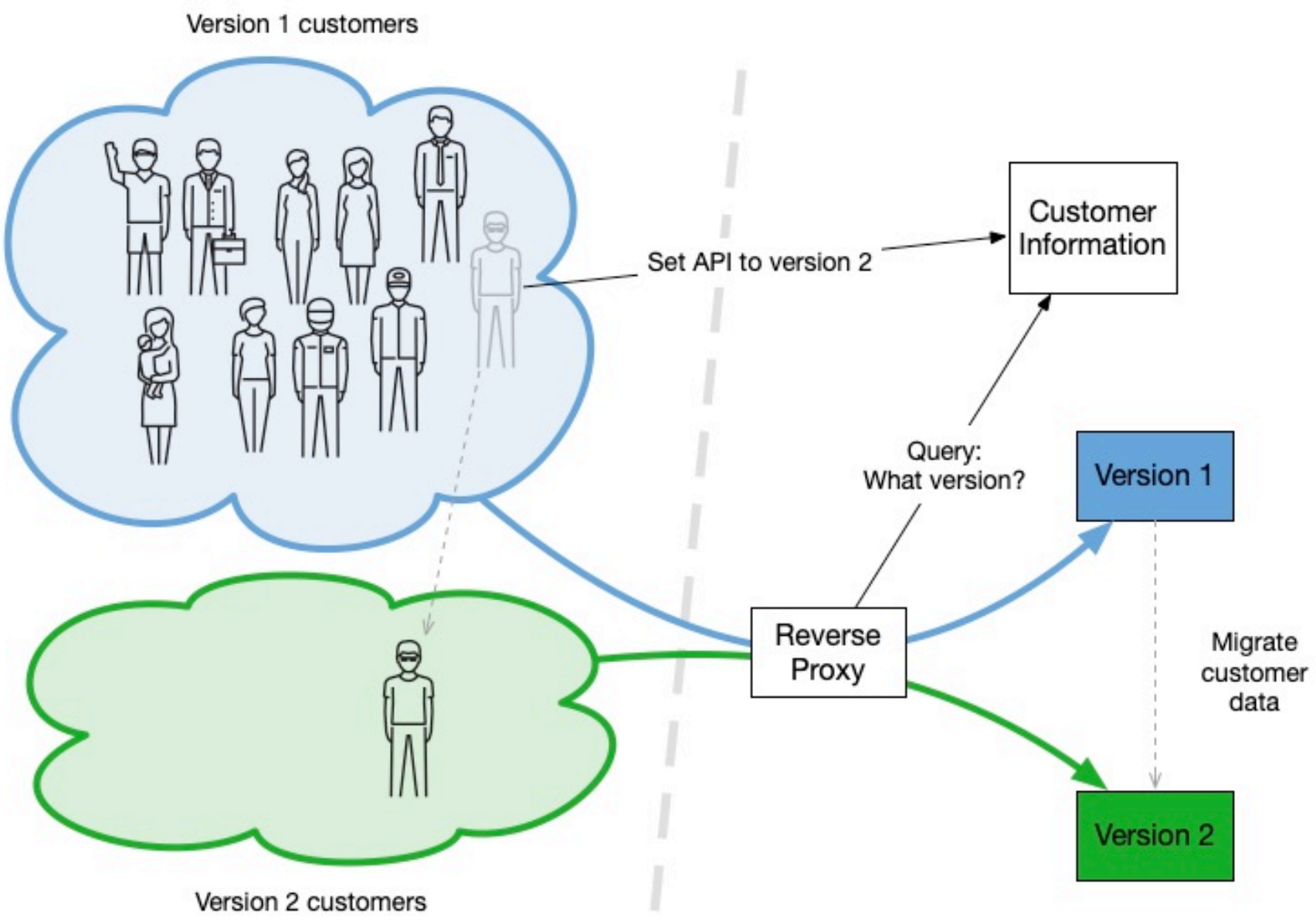
Gradual deployment (canary release)



Service evolution


Service
v1 & v2
Host, path,
contents, …
v1-api.svc.com
Service
version 1
Reverse
Proxy
v2-api.svc.com
Service
version 2
/v1/
Service
version 1
Reverse
Proxy
/v2/
(content?)
Service
version 2


Version
1
Version
2
?
?
?


Version 1 customers
Set API to version 2
Customer Information
Query: What version?
Version 1
Reverse Proxy
Migrate customer data
Version 2
Version 2 customers

Migration

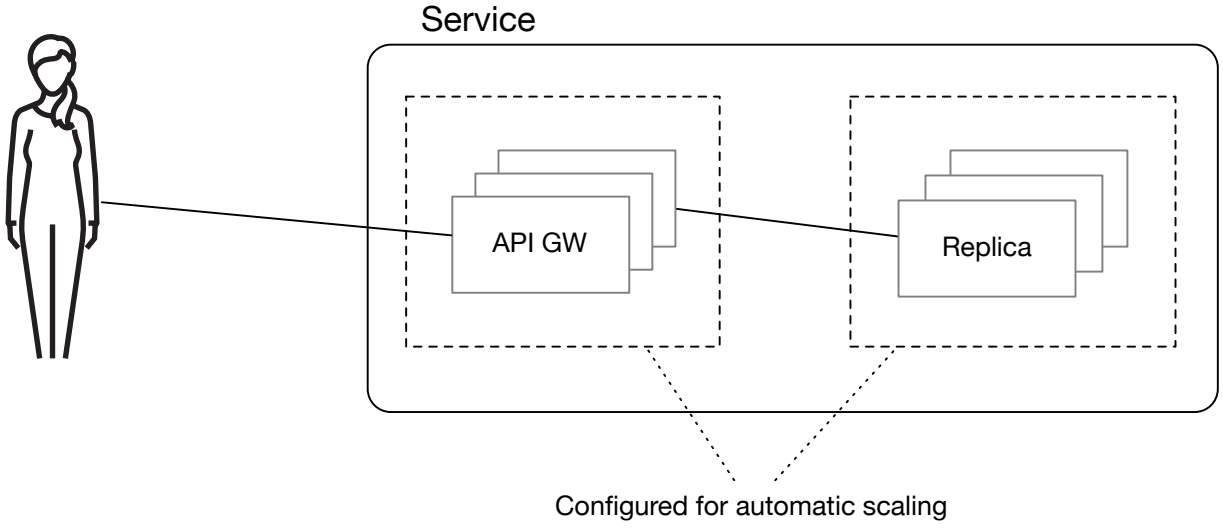
- **Idea: Run new and version in parallel but each customer in only one**
  - Migrate customers to the new version
  - Customers get to choose when to migrate
- **Pros**
  - Removes need of explicit versioning from interface
  - Version to be used becomes part of the customer configuration
  - Migration on customer's own pace
- **Cons**
  - Requires explicit customer information (+ no SLA on anonymous APIs)
  - Schema changes and data migration (Rollback? Lots of data to transfer?)
  - Gateway has to know which version to use (dynamic)

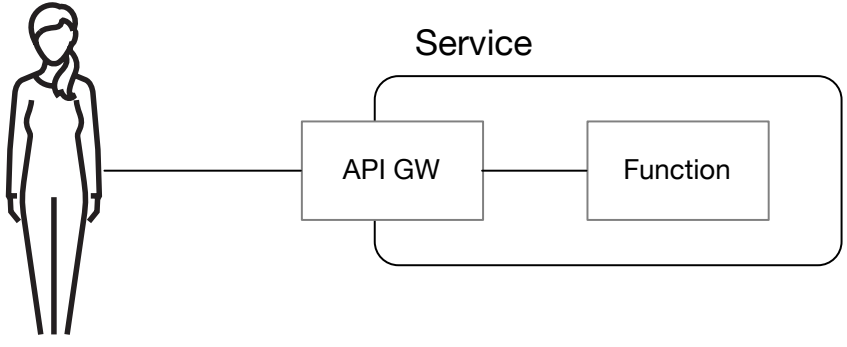
Serverless computing

Aalto University
School of Electrical
Engineering

Serverless

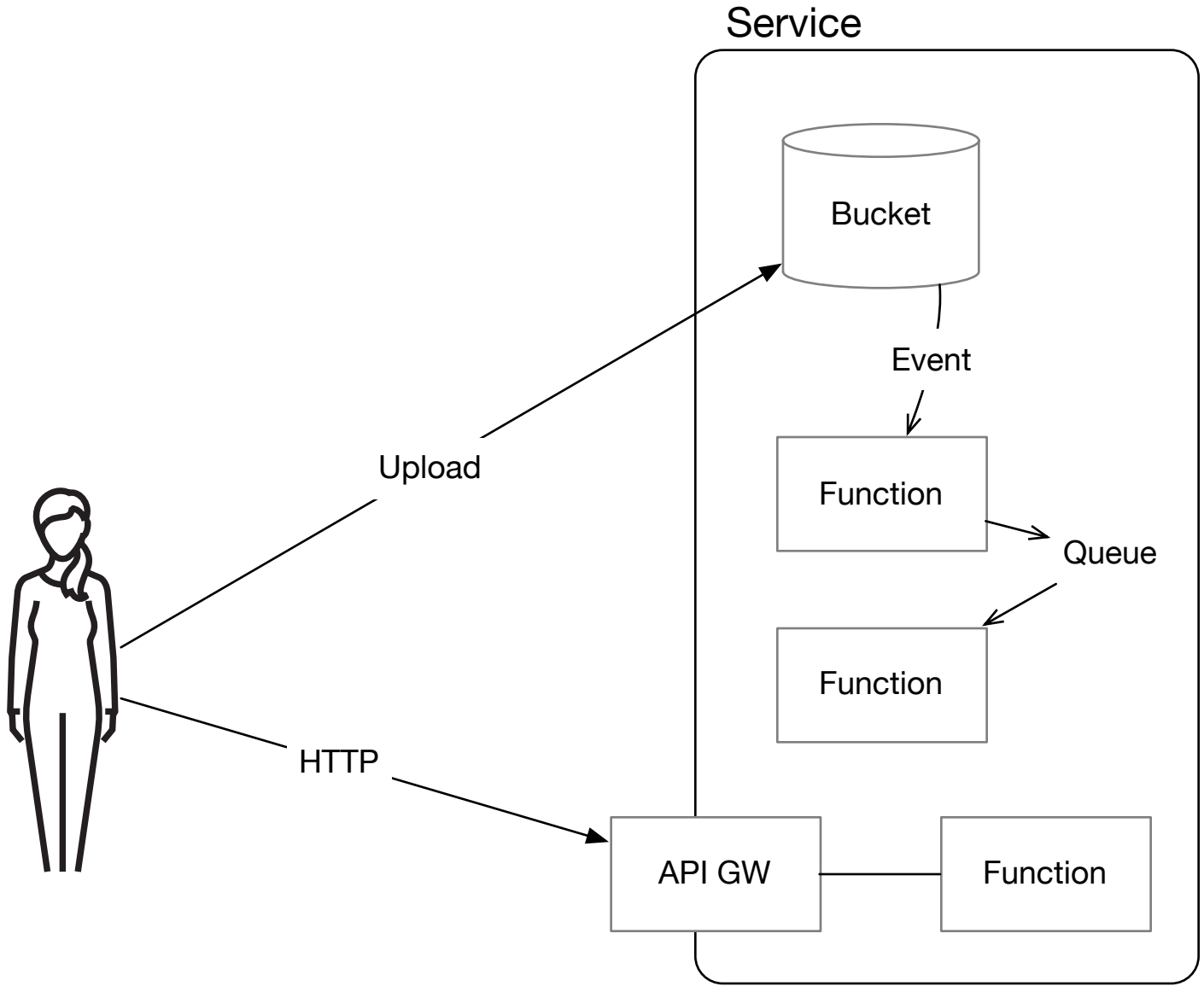
- **“Serverless” (or Function-as-a-Service, FaaS)**
  - There is always some hardware somewhere (servers)
  - Operates at a function or a single service level (one or more “endpoints”)
  - ”Someone else” is responsible for
    - *Providing hardware*
    - *Scaling up and down as needed*
    - *Handling log collection*


Service
API GW
Replica
Configured for automatic scaling


Service
API GW
Function

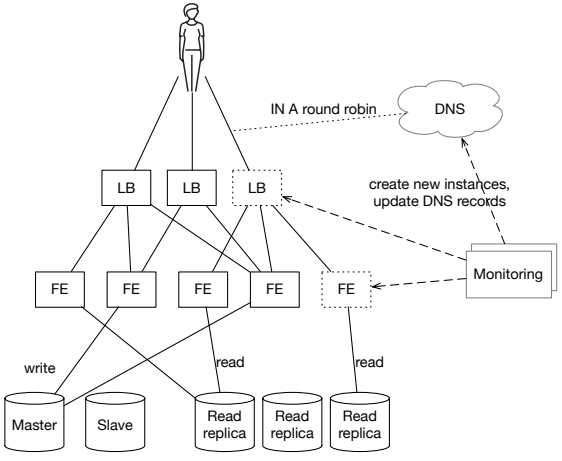
Event model

- **Serverless uses an event model**
  - For HTTP, receives a request event
- **Many other event sources**
  - Data streaming
  - Messages from queues
  - IaaS internal events (like bucket upload complete)
  - Chimes (e.g. cron triggers)

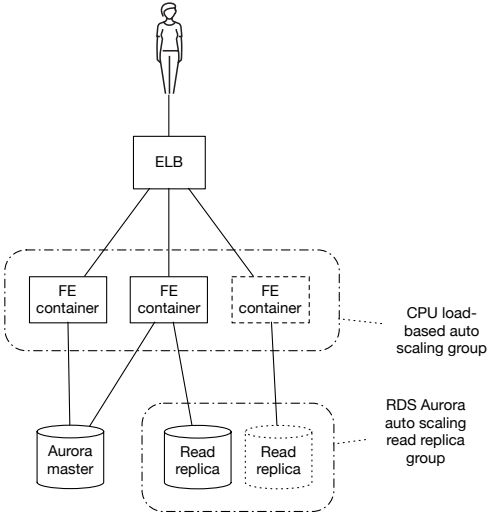

Service
Bucket
Event
Function
Queue
Function
Upload
HTTP
API GW
Function

Quiz problems

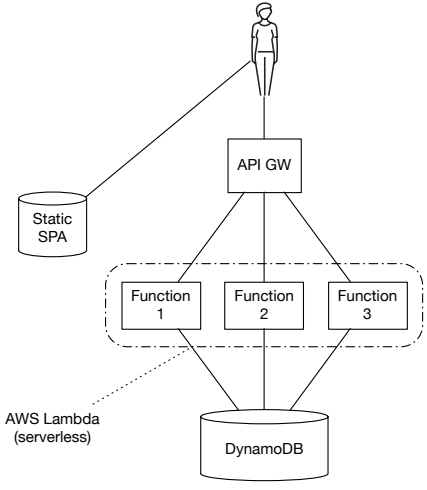
Problem 1: Scalable service


IN A round robin
DNS
LB
LB
LB
create new instances,
update DNS records
Monitoring
FE
FE
FE
FE
FE
write
read
read
Master
Slave
Read
replica
Read
replica
Read
replica

A

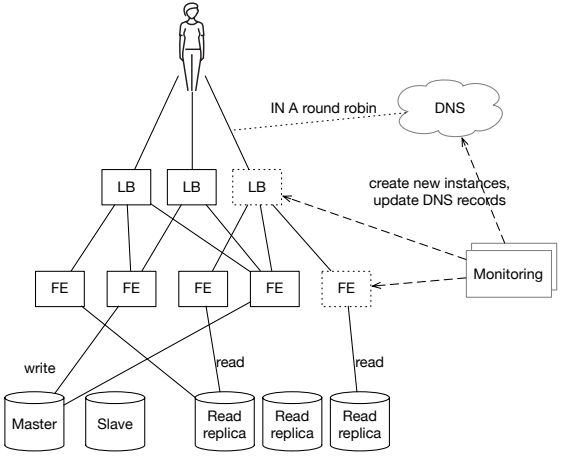

ELB
FE
container
FE
container
FE
container
CPU load-
based auto
scaling group
RDS Aurora
auto scaling
read replica
group
Aurora
master
Read
replica
Read
replica

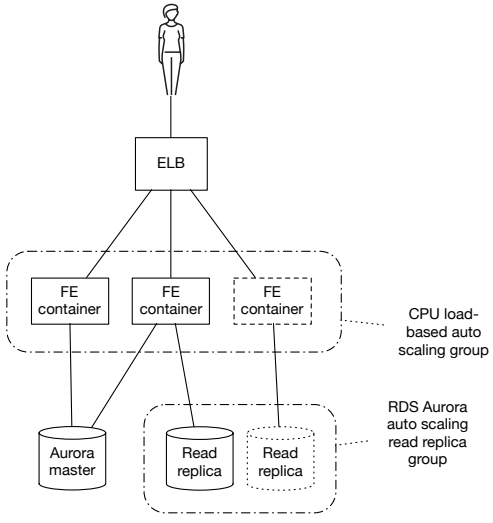
B

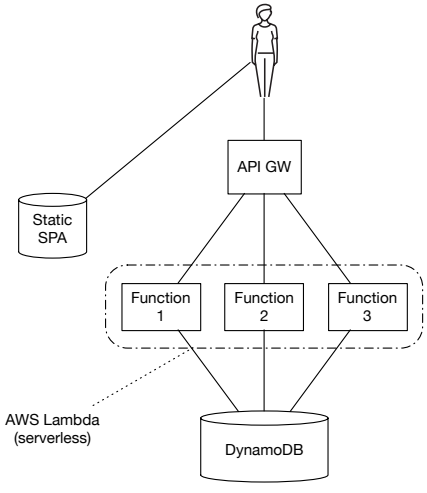

API GW
Static
SPA
Function
1
Function
2
Function
3
AWS Lambda
(serverless)
DynamoDB

C

Problem 2: Why one over another?







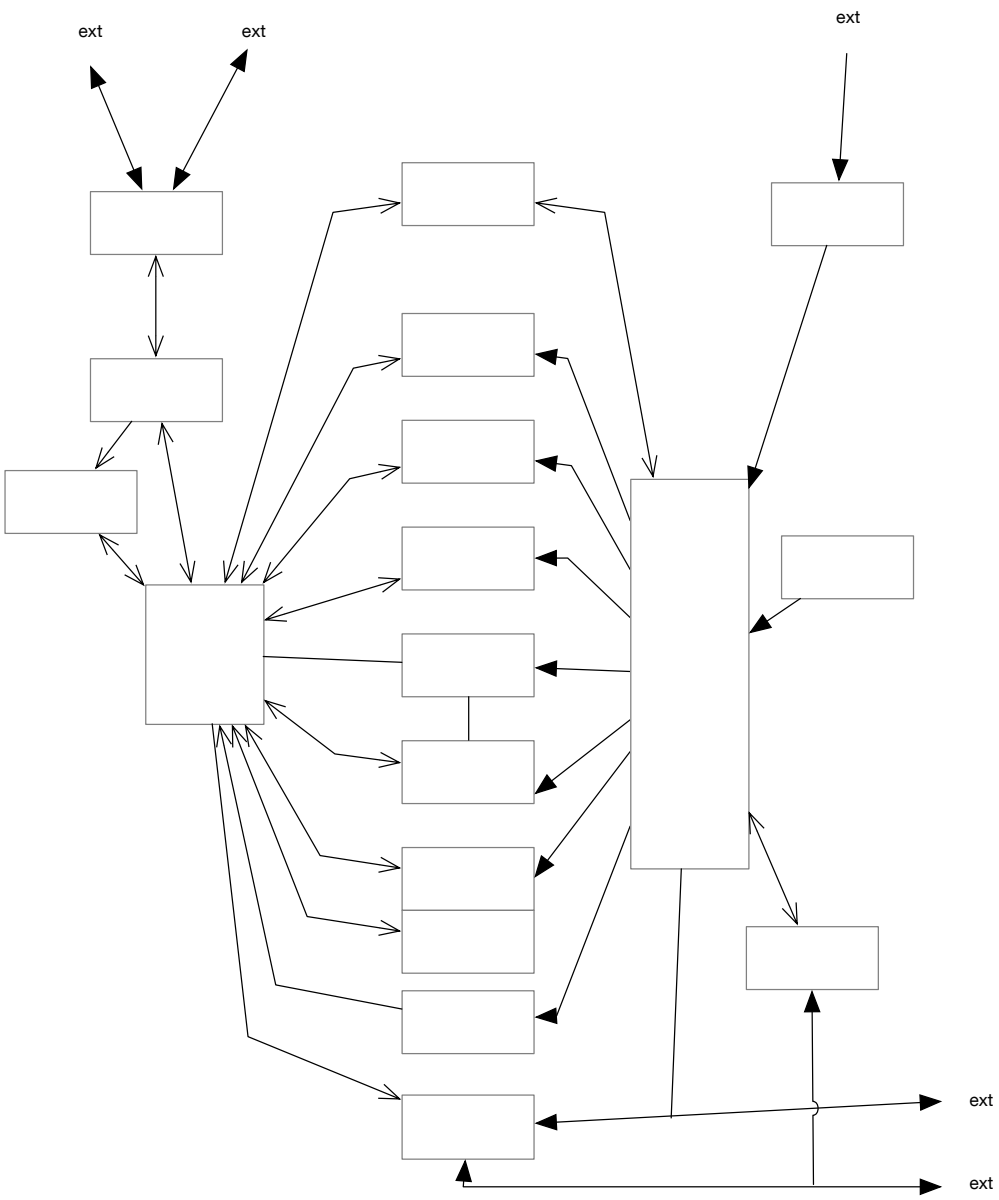
Good when?

Bad when?

Problem 3: Large data

whiteboard

Problem 4: What does this do?


ext
ext
ext
ext
ext

Exam

- **Potential question types**
  - Concepts and definitions
  - Comparison
  - Design problem
  - Evaluation
  - Selection
  - …
- **Thursday 11.4. at 16:30, TU7 / TUAS**