

# Quarkus Basics

Training



**PUZZLE ITC**  
changing IT for the better

# Nice to meet you



Raffael Hertle  
Cloud Architect  
[rafael@rokt.cloud](mailto:rafael@rokt.cloud)



Christof Lüthi  
Kafka Messaging Engineer  
[luethi@puzzle.ch](mailto:luethi@puzzle.ch)

# Agenda - Day 1

- Microservices architecture
- Quarkus introduction
- MicroProfile specification
- RESTful microservices with Quarkus

# Agenda - Day 2

- 8 fallacies of distributed computing
- Cloud patterns
- Continuous integration and delivery
- Event driven architecture and messaging
- Observability – Metrics and Tracing
- Writing your own Quarkus extension

# Day 1

# Agenda - Day 1

- Microservices architecture
- Quarkus introduction
- MicroProfile specification
- RESTful microservices with Quarkus

# Microservices

- What are microservices
- Why and when to use them
- Advantages / Disadvantages
- Approach to migrate

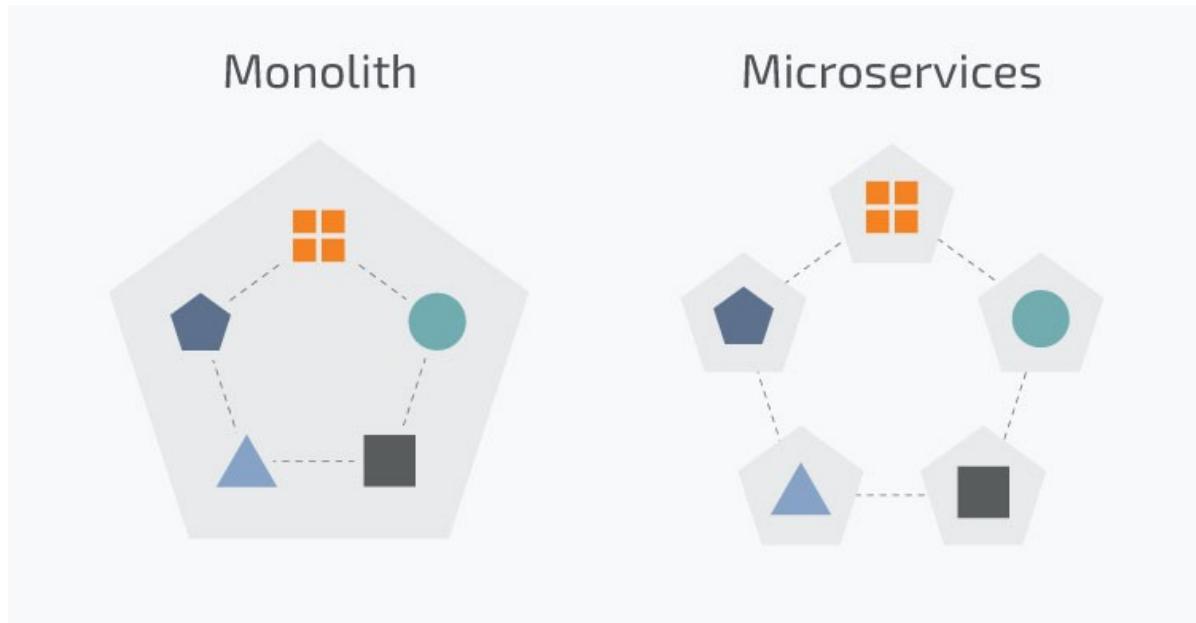
# Monolithic Architecture



# Monolithic Architecture

- Single code-base
- Single unit deployable
- Independent from other applications
- All domains or business processes in one application

# Microservices Architecture



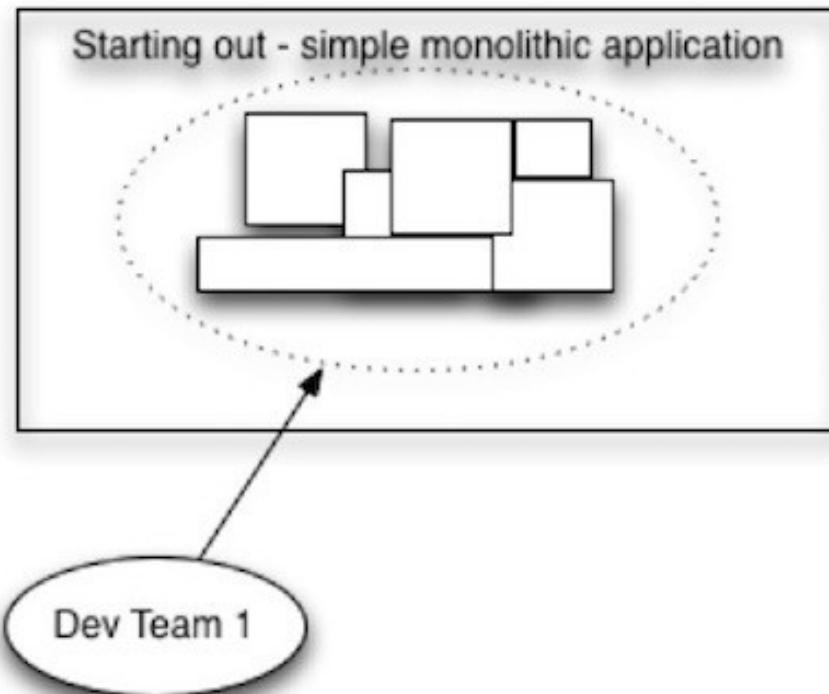
# Microservices Architecture

- Small autonomous applications
- Independent life cycles
- Microservice for single responsibility / domain
- Loosely coupled
- Code base per domain / business process

# Microservices Architecture

- When to choose which architecture?

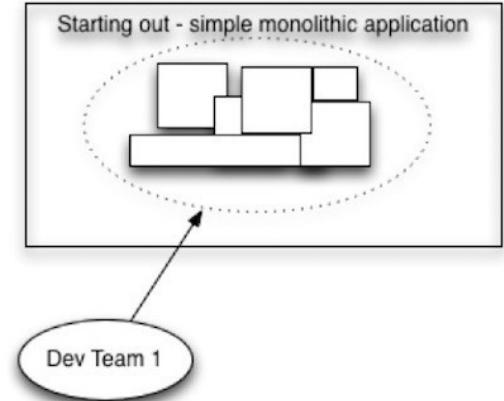
# New Application



# Advantages of monoliths

## Simple architecture

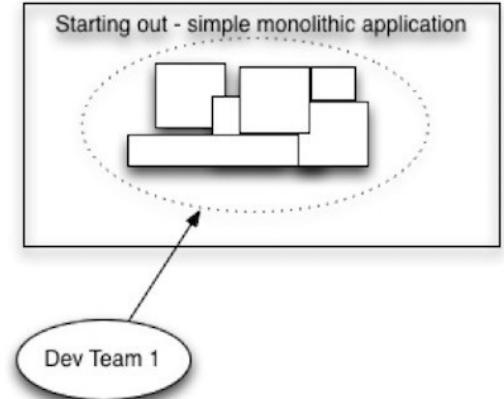
- Everything is local
- High productivity
- Limited attack vectors
- Easy testing
- Performance matches requirements



# Advantages of monoliths

## Team

- Dedicated team
- Independent releasing
- Features can be released fast
- No dependencies to other teams
- Devs have strong application knowledge

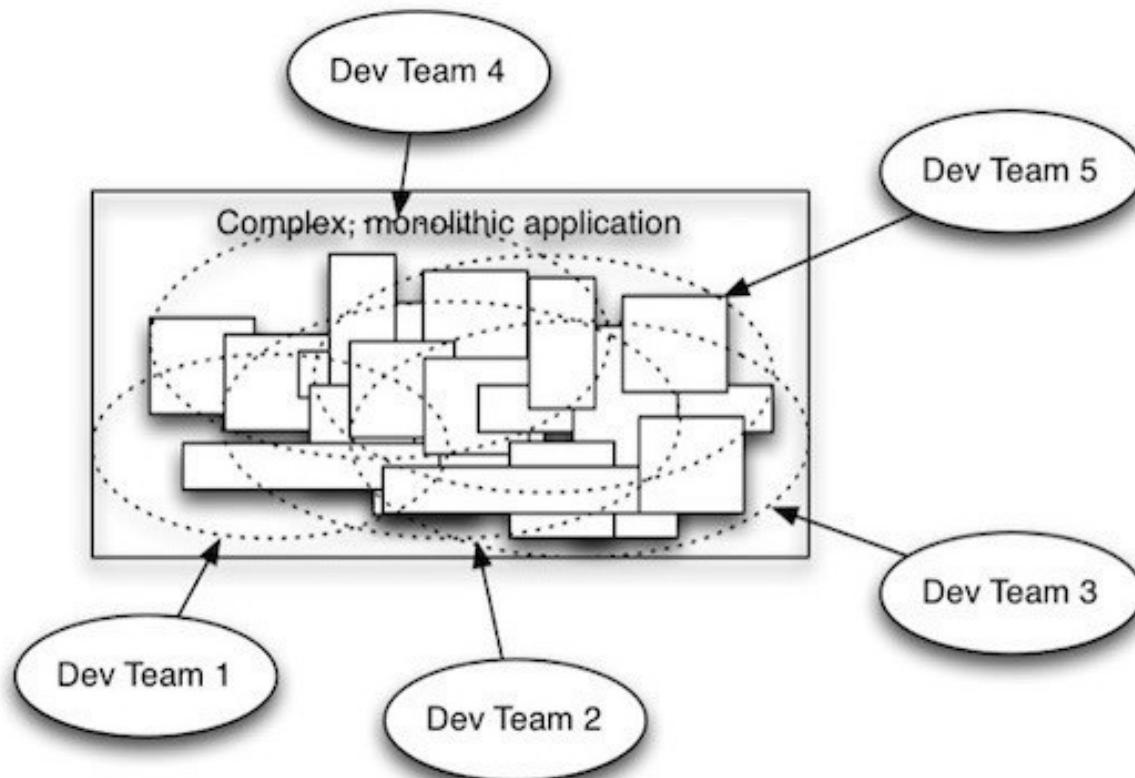


# But then...

- Application is a big success
- Users increase
- Traffic increases dramatically
- New features
- Dev team grows



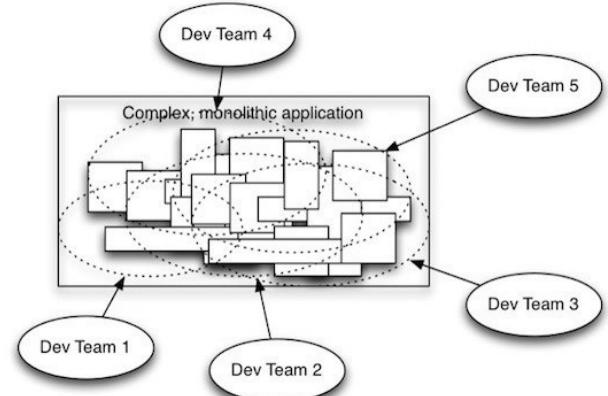
# Application and complexity grows



# Disadvantages of monoliths

## Complex architecture

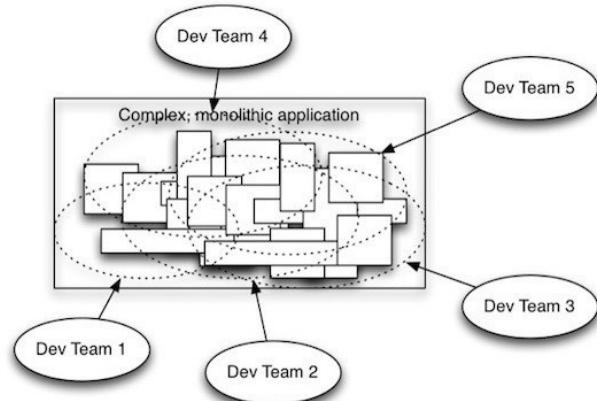
- Architectural changes are difficult
- Impact of code change are hard to estimate
- Keeping up code quality needs extra effort
- Newer technologies are hard to pickup



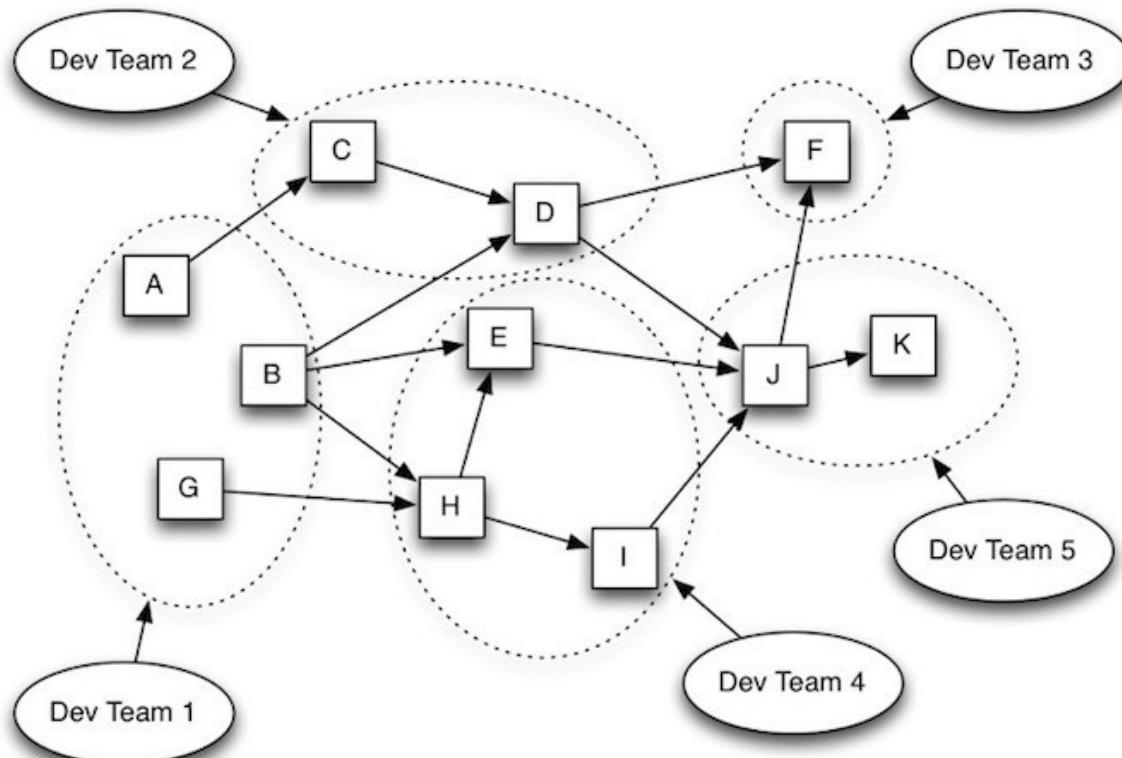
# Disadvantages of monoliths (cont.)

## Team

- Teams need to be coordinated
- Code changes may collide
- Release planning required
- Feature freeze and test cycles
- Devs have limited knowledge
- Productivity drops



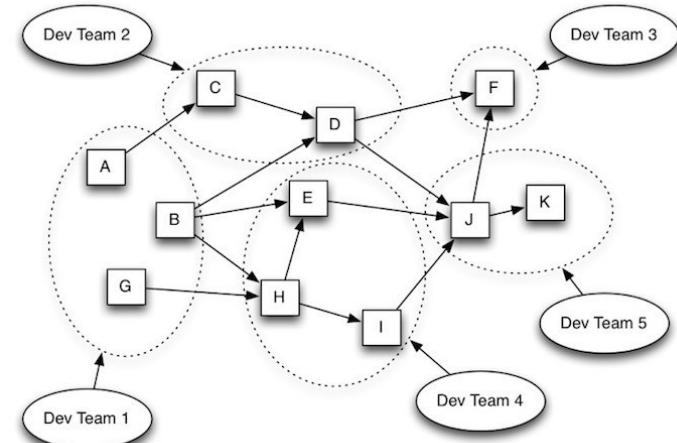
# Ok, now what?



# Advantages of microservices

## Architecture

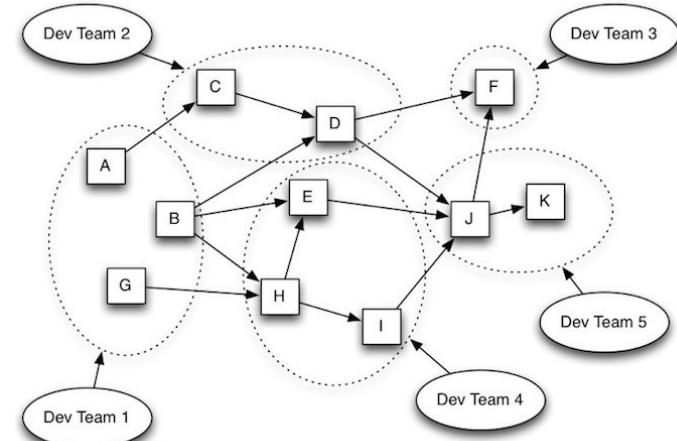
- Independent modules
- Defined boundaries (APIs, Events)
- Loosely coupled (if done right)
- Polyglot (what best fits the task)



# Advantages of microservices

## Team

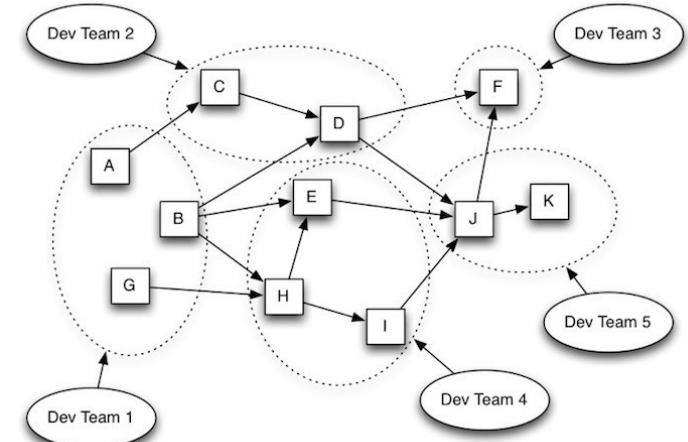
- Teams work independently
- Need to agree to defined boundaries
- Independent within their microservice
- Easier onboarding due limited scope



# Advantages of microservices

## Deployment and Runtime

- Deploy independently
- Easier scaling of single components
- Bugs may be local only



# But ...



# Disadvantages of microservices

## Architecture

- Everything is local does not hold anymore
- Data is distributed, no foreign keys across boundaries
- Keeping data consistent needs extra effort
- Communication and error handling needs extra effort
- Changing the agreed boundaries may be hard

# Disadvantages of microservices (cont.)

## Deployment and Runtime

- Harder troubleshooting with multiple instances
- Root cause detection can be hard
- More attack vectors

# Short Recap

## Microservices

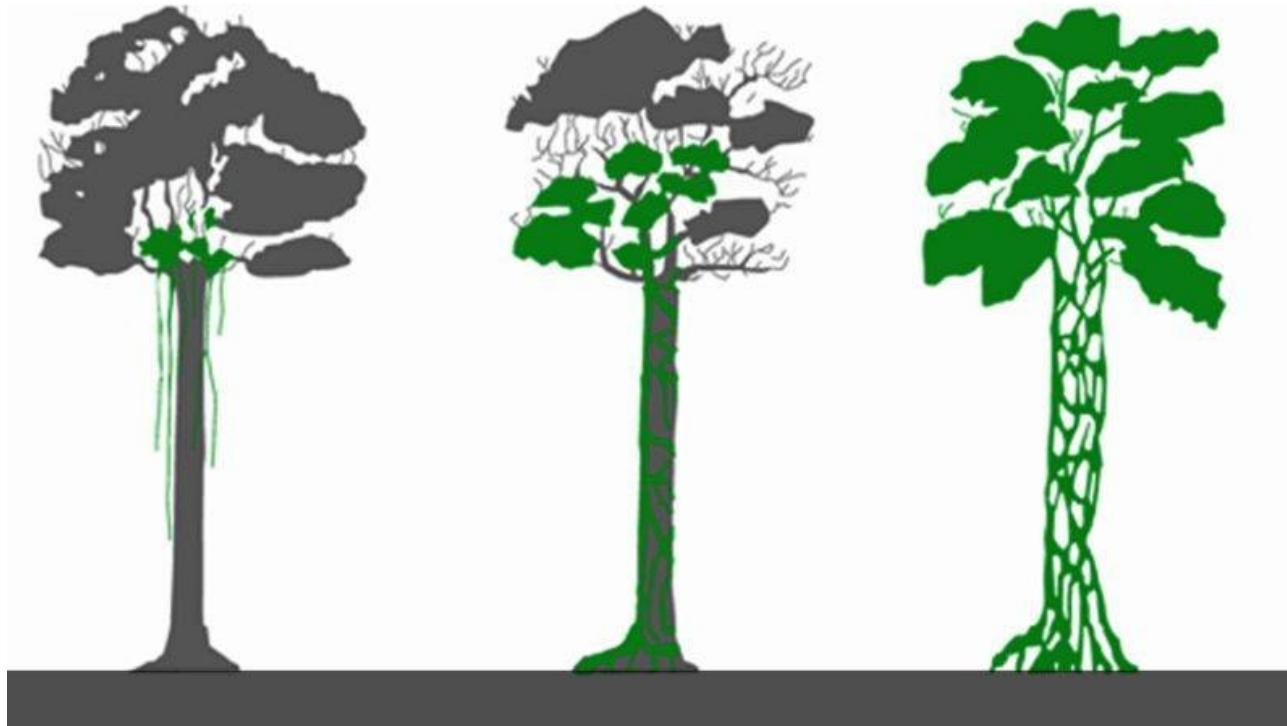
- Lead to modularity
- Developers are enforced to respect boundaries
- Enable teams to work and release independently
- Can be replaced as long as boundary is untouched

But they introduce technical complexity

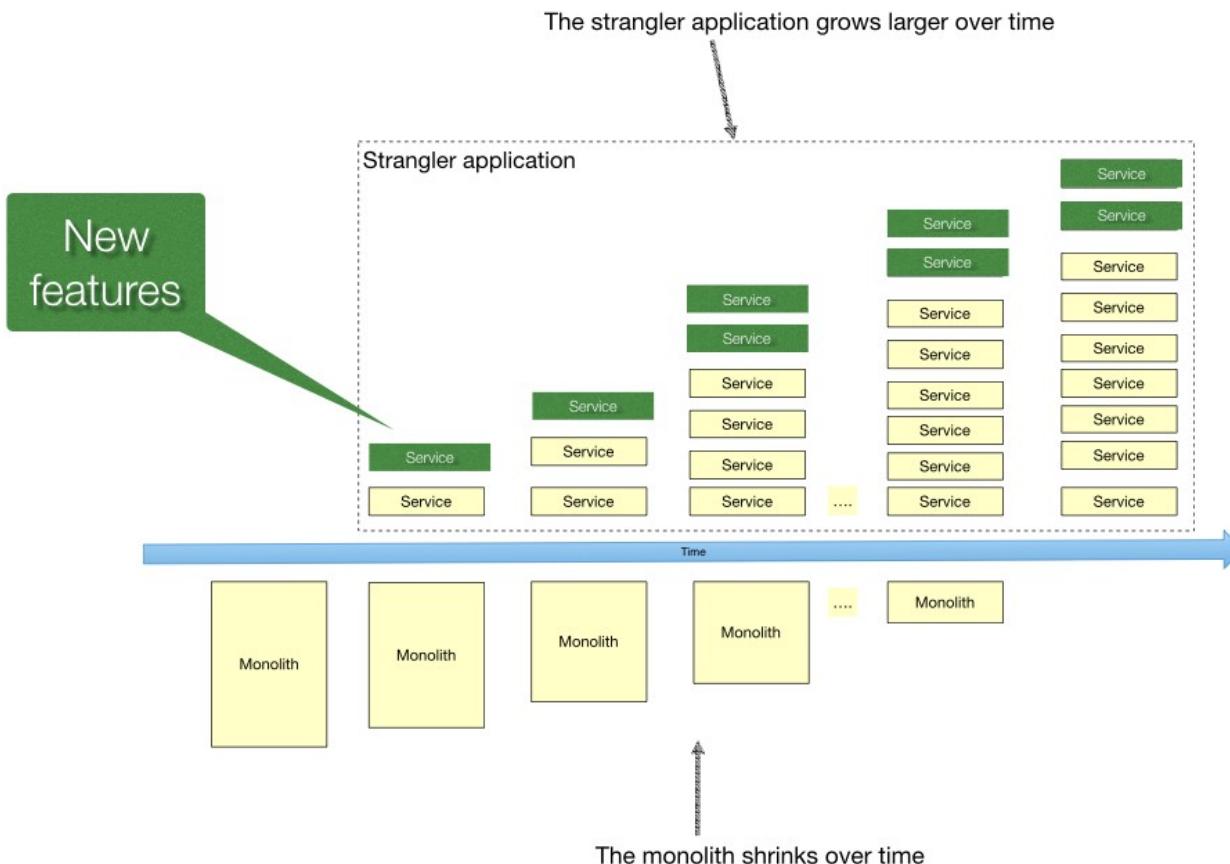
# Microservices Architecture

- How to migrate from monolithic application?

# Strangler pattern



# Strangler pattern



# Agenda - Day 1

- Microservices architecture
- Quarkus introduction
- MicroProfile specification
- RESTful microservices with Quarkus

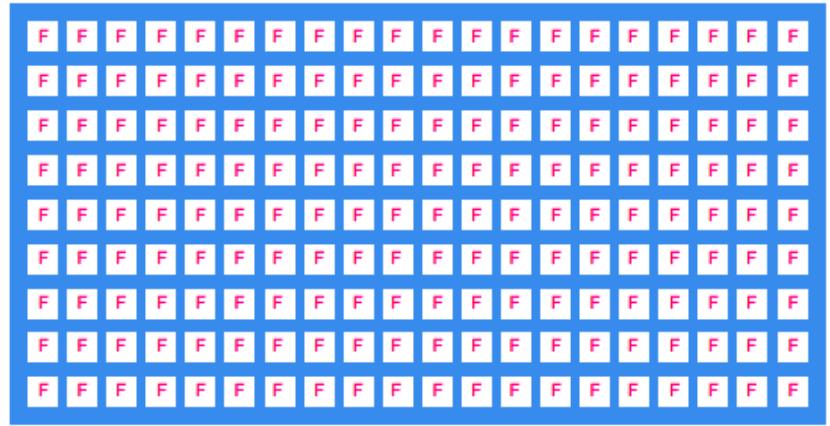
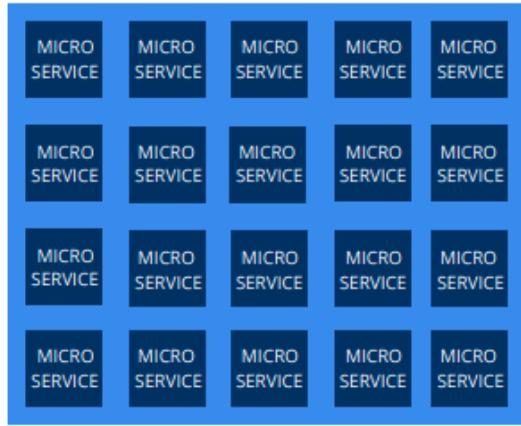
# What is Quarkus?

«Full-stack, Kubernetes-native Java framework made for JVM and native compilation», Red Hat

«Toolkit and Framework for writing Java, Kotlin and Scala applications», Peter Palaga 2020

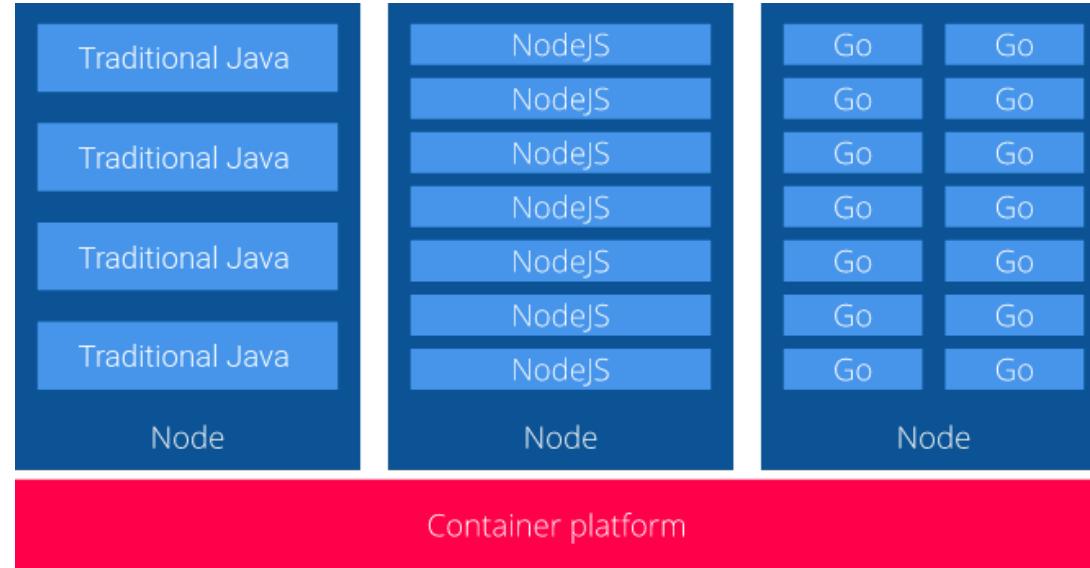
«Build time augmentation Toolkit», Peter Palaga 2021

# Why Quarkus?



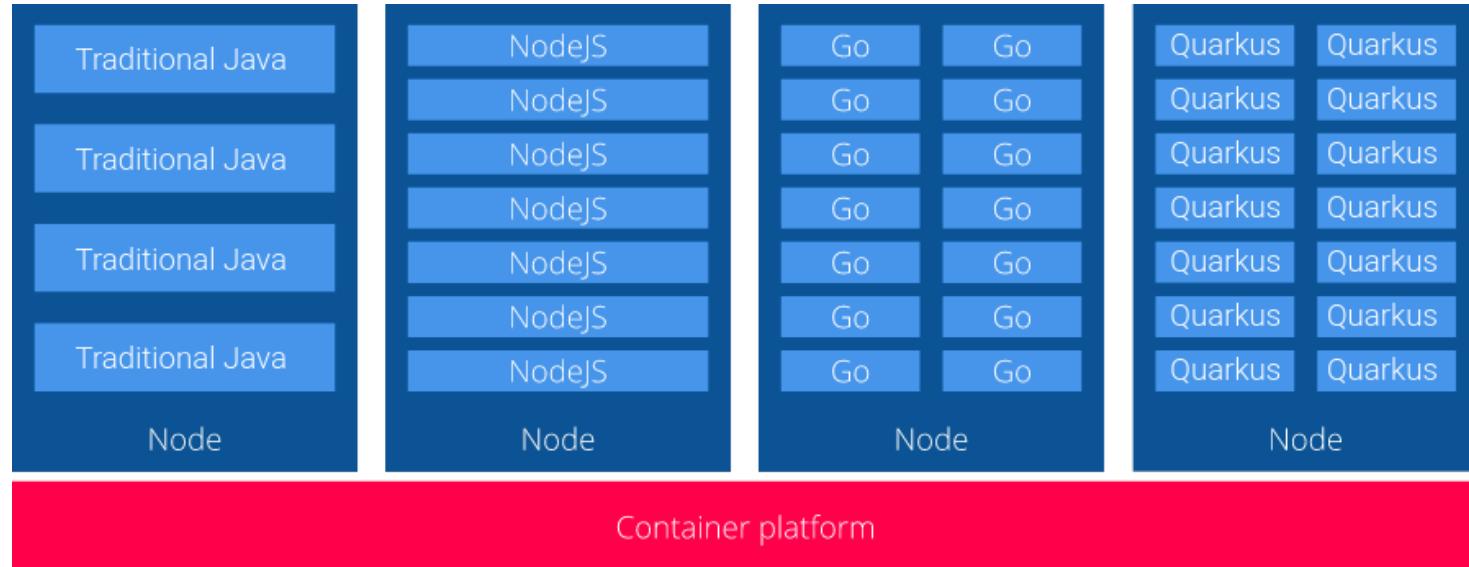
- 1 monolith → 20 Microservices → 200 functions
- Big → Small
- Long living → short living

# Why Quarkus? – Java and containers



- Startup overhead
- Resource overhead

# Why Quarkus?



- Optimizing Java for containers
- Enabling Java to become an effective platform for serverless, cloud and Kubernetes environments

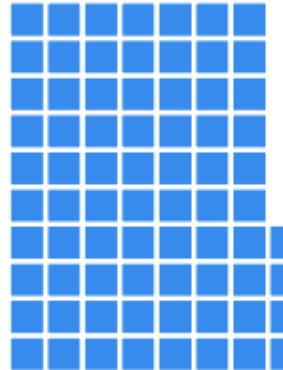
# Show me numbers!

REST



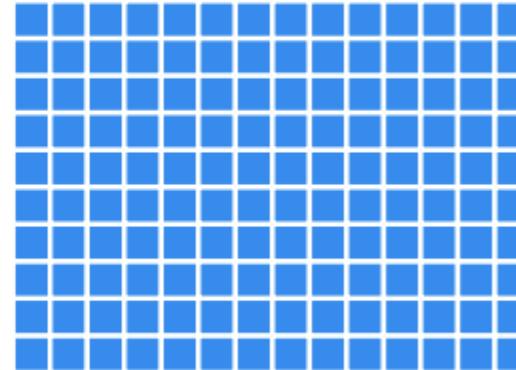
Quarkus + GraalVM

**13 MB**



Quarkus + OpenJDK

**74 MB**



Traditional Cloud-Native Stack

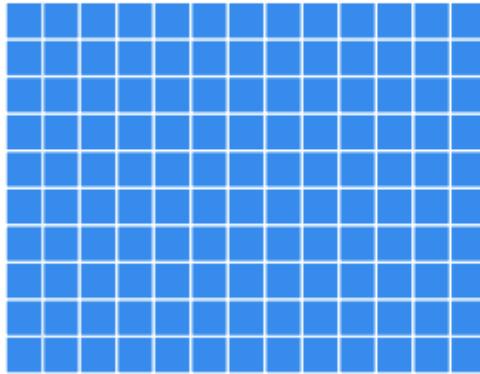
**140 MB**

# Show me numbers!

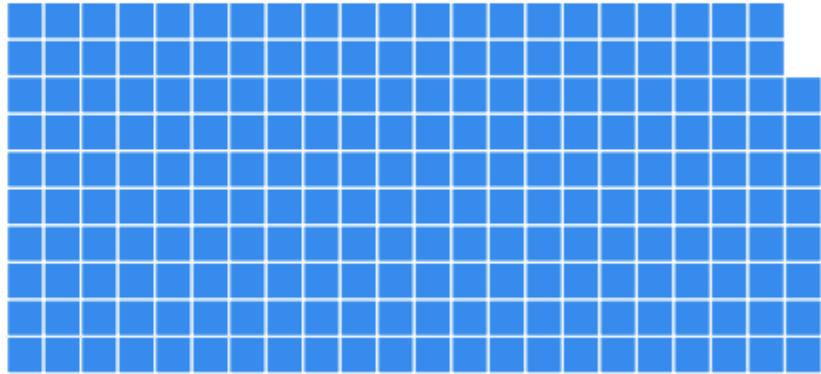
REST + CRUD



Quarkus + GraalVM  
**35 MB**



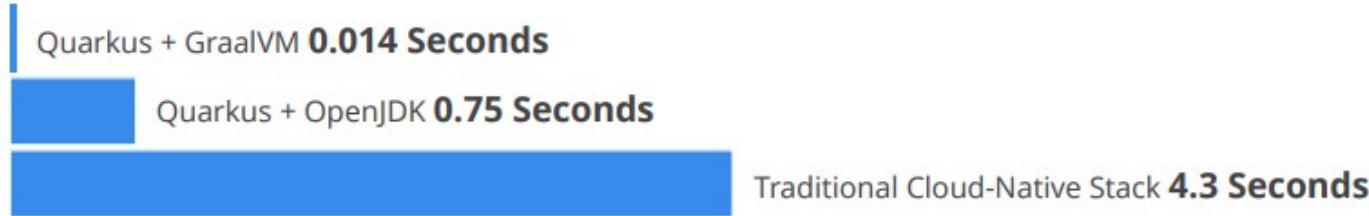
Quarkus + OpenJDK  
**130 MB**



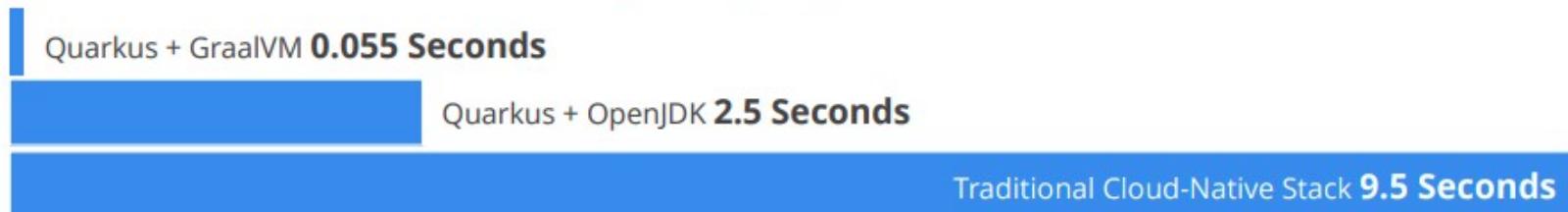
Traditional Cloud-Native Stack  
**218 MB**

# Show me numbers!

## REST

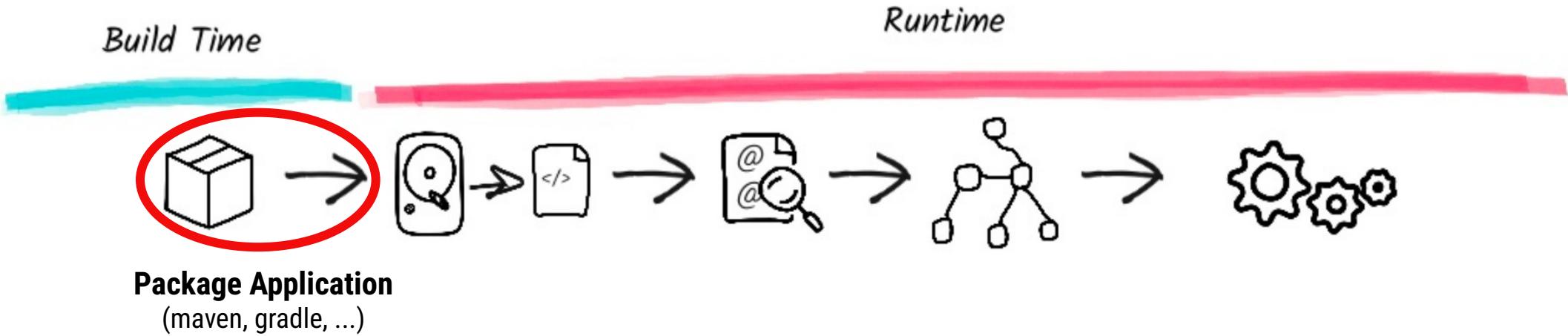


## REST + CRUD

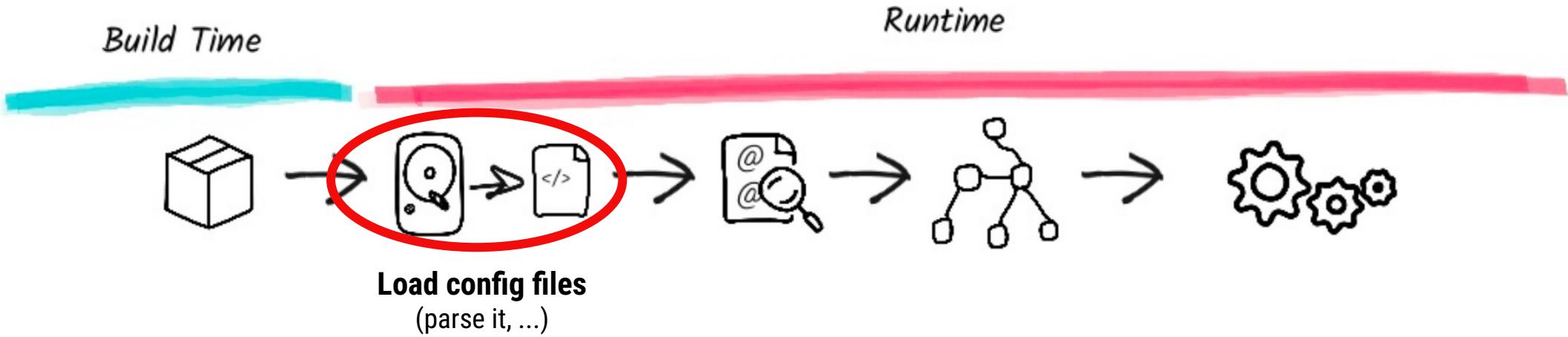


Time to first response

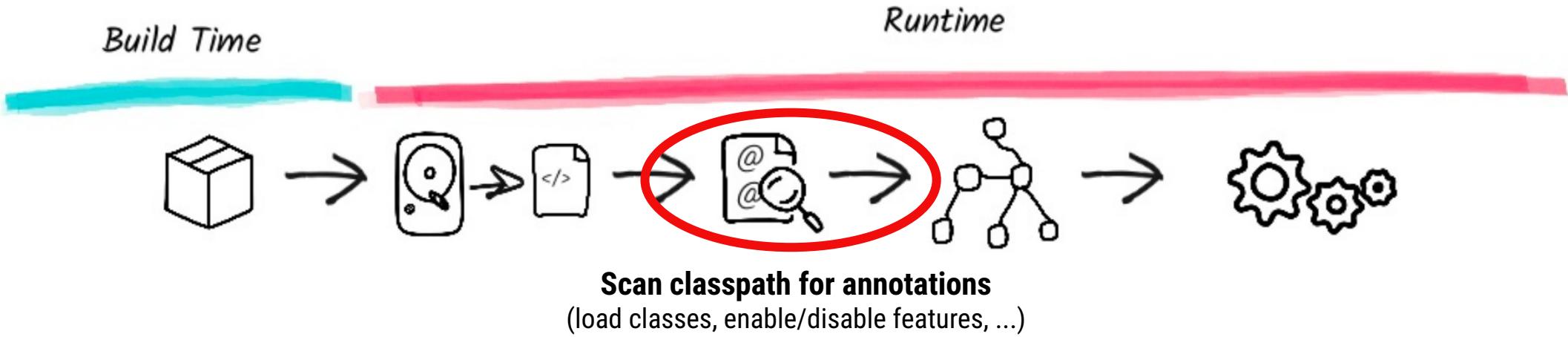
# How a framework starts



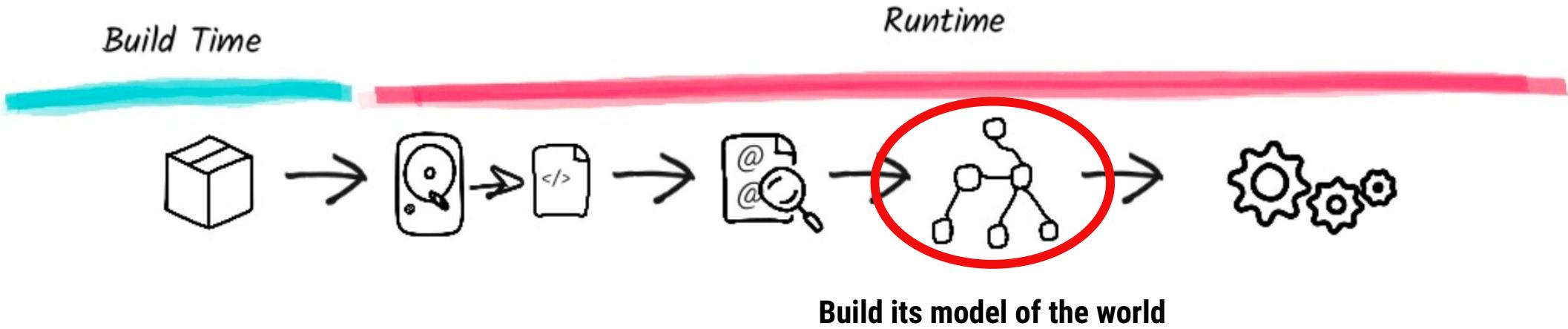
# How a framework starts



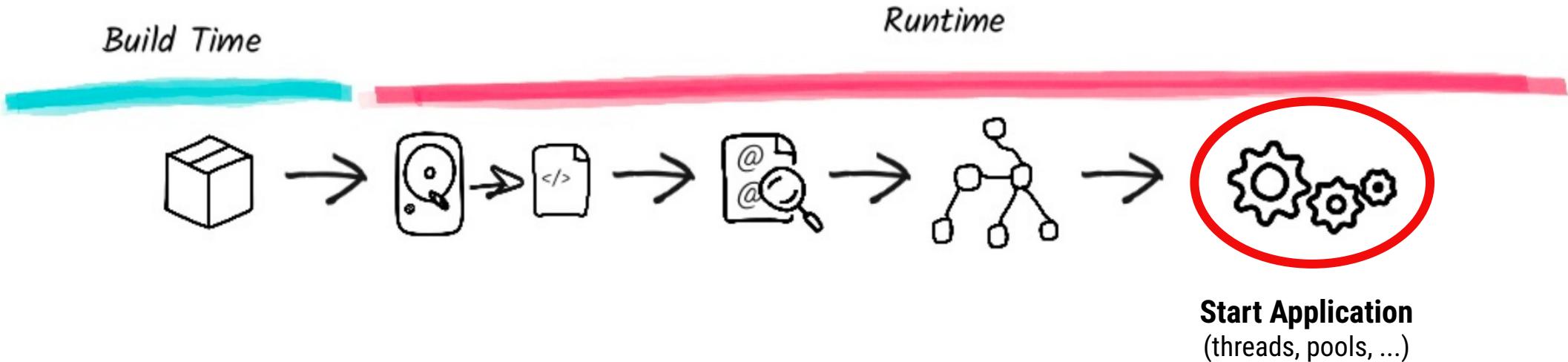
# How a framework starts



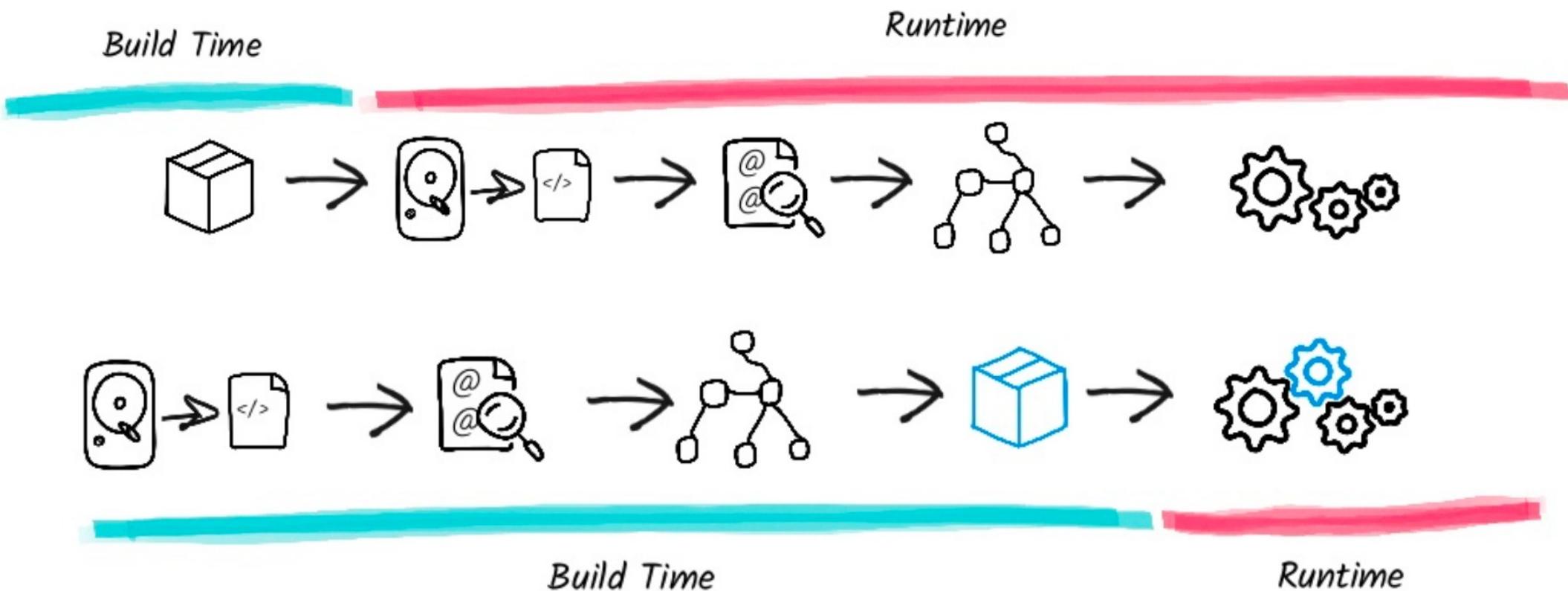
# How a framework starts



# How a framework starts



# How Quarkus does it



# Traditional vs Quarkus

## Traditional

- Many classes only run during boot
  - Xml parsers, annotation lookup, ...
  - Later unused and occupy memory

## Quarkus

- Built time approach
- As much work as possible during build
- Built time augmentation
  - Extensions
- Output: recorded wiring bytecode

# Quarkus Build

«The end result of a Quarkus build  
should be just enough bytecode to start  
the services that your application  
requires»

*Edson Yanaga, Red Hat*

# Configuration phases

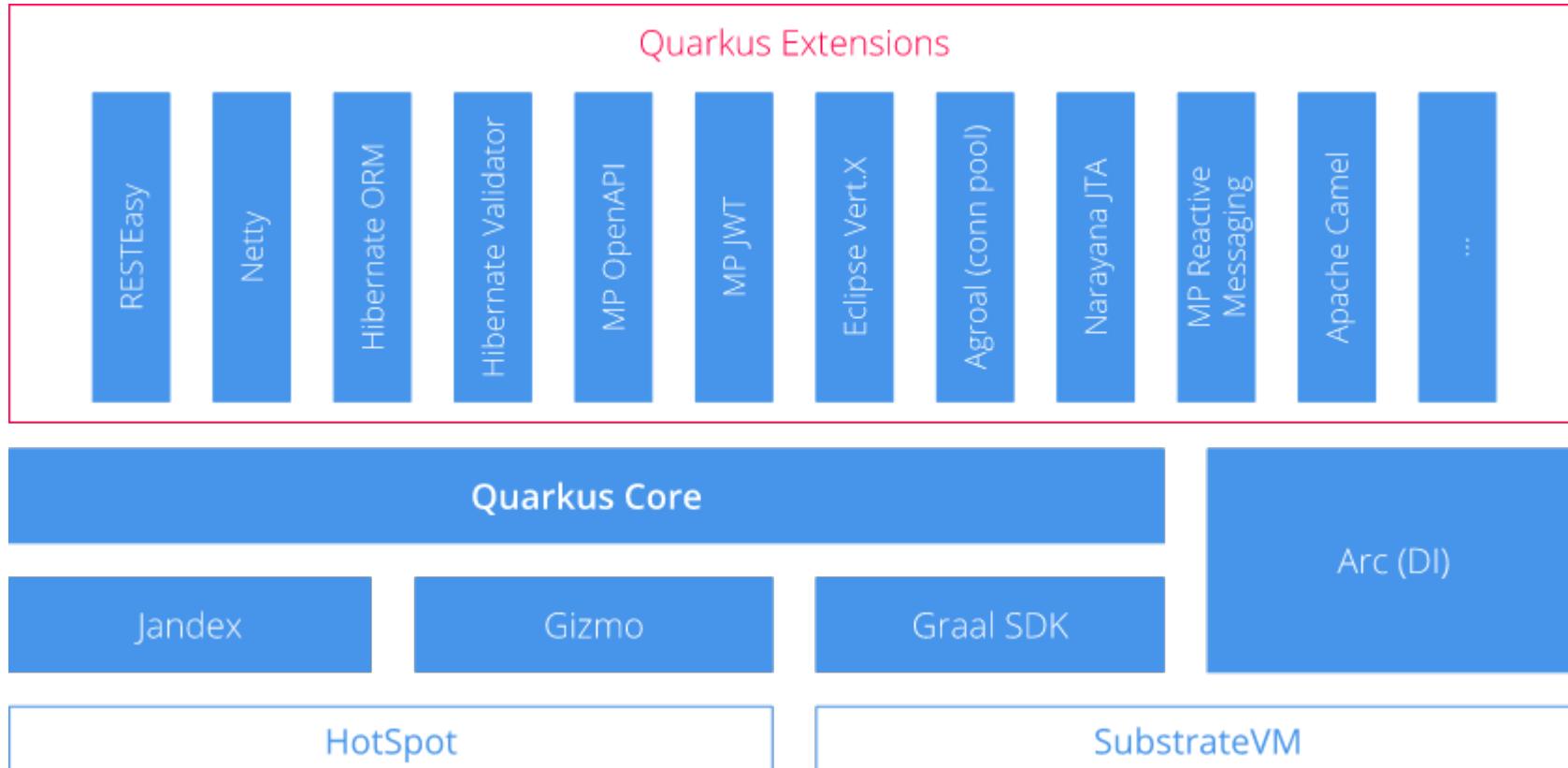
- Quarkus needs to make decisions at build time
- Different config phases
  - Build time
  - Build and runtime fixed
  - Bootstrap
  - Run Time

# Configuration phases

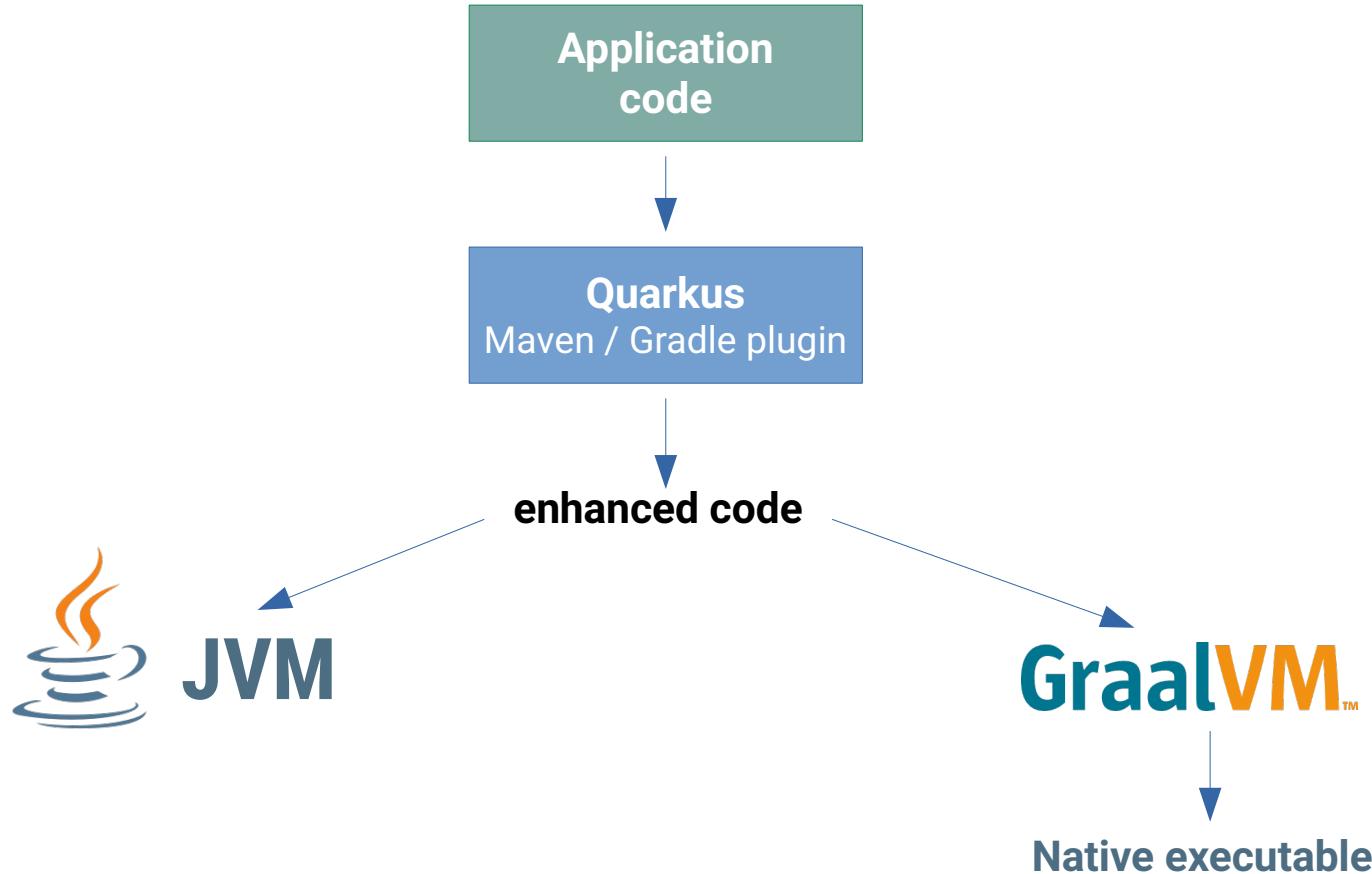
- Not all properties may be changed at runtime
- Some properties will need a rebuild
- Check «[All Configuration Options](#)» of Quarkus

Datasource configuration	Type	Default
<b>quarkus.datasource.db-kind</b> The kind of database we will connect to (e.g. h2, postgresql...).	string	
<b>quarkus.datasource.health.enabled</b> Whether or not an health check is published in case the smallrye-health extension is present. This is a global setting and is not specific to a datasource.	boolean	true
<b>quarkus.datasource.metrics.enabled</b> Whether or not datasource metrics are published in case a metrics extension is present. This is a	boolean	false
<b>quarkus.datasource.username</b> The datasource username	string	
<b>quarkus.datasource.password</b> The datasource password	string	

# Quarkus structure



# Quarkus modes



# Downside of GraalVM

- Closed world assumption
  - Everything has to be known at build-time
- No dynamic classloading
- No native VM Interfaces (JVM tool Interface, Java Management Extensions JMX)
- No agents (Profilers, Tracing, ...)
- No Java debugger
  - Native debugger GDB

<https://www.graalvm.org/22.1/reference-manual/native-image/Limitations/>

# Downside of GraalVM

- Require Registration
  - Reflection limited (@RegisterForReflection, reflection-config.json)
  - Dynamic Proxies (Proxy Config files)
  - Resources (resources-config.json)
- Static Inits
  - Bytecode recorded at build time
  - Stored in executable
  - No file handles, sockets, threads

<https://www.graalvm.org/22.1/reference-manual/native-image/Limitations/>

# Extensions and standards

Extend Quarkus framework with custom functionality

## Standards

Java EE (Servlet, JAX-RS, CDI, JPA, Bean Validation, ...)

MicroProfile (Health, Metrics, Rest Client, Fault-Tolerance, ...)

Spring (Web, Boot, Data, Scheduled,...)

## Databases & Tooling

PostgreSQL, MySQL, MariaDB, MS SQL Server, H2, MongoDB, Neo4j, Flyway, Liquibase ...

## 3rdParty libraries and frameworks

Netty, Vert.x, Apache Camel, Caffein, Keycloak, Elasticsearch, Infinispan, Debezium, ...

# Extensions and standards

## Quarkus Core Repository

[github.com/quarkusio](https://github.com/quarkusio) (~150 extensions)

## Independent sources

[github.com/apache/camel-quarkus](https://github.com/apache/camel-quarkus)

[github.com/datastax/cassandra-quarkus](https://github.com/datastax/cassandra-quarkus)

[github.com/debezium/debezium-quarkus-outbox](https://github.com/debezium/debezium-quarkus-outbox)

...

## Quarkiverse: Incubator & Community extensions

logging-json, cxf, github-api, freemarker, google-cloud-services, ...

[code.quarkus.io](https://code.quarkus.io) provides a big list of extensions

# Developer Joy

Easy to start & fast bootstrap

```
mvn io.quarkus:quarkus-maven-plugin:3.4.3:create \
-DprojectGroupId=ch.puzzle \
-DprojectArtifactId=quarkus-getting-started \
-DclassName="ch.puzzle.quarkustechlab.GreetingResource" \
-Dpath="/hello"
```

```
cd quarkus-getting-started
```

```
mvn quarkus:dev
```

```
curl localhost:8080/hello
Hello from RESTEasy Reactive
```

# Developer Joy

## Live Reload

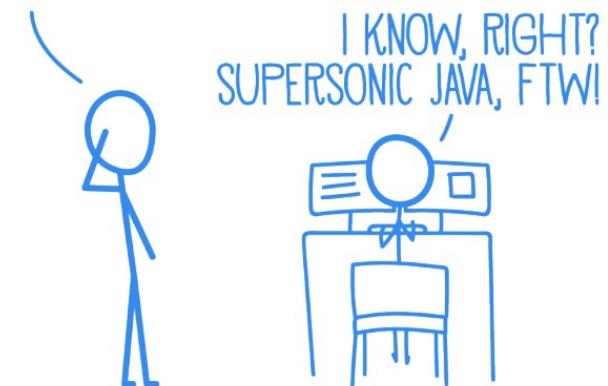
```
mvn quarkus:dev

curl localhost:8080/hello
Hello from RESTEasy Reactive

sed -i -e 's/RESTEasy/QuarkusTraining/g' \
.../GreetingResource.java \
&& curl localhost:8080/hello
Hello Quarkus

-> Live reload total time: 0.638s
```

WAIT.  
SO YOU JUST SAVE IT,  
AND YOUR CODE IS RUNNING?  
AND IT'S JAVA?!



# Developer Joy – Dev UI

<http://localhost:8080/q/dev> ... or hit «d» in your quarkus:dev console

The screenshot shows the Quarkus Dev UI interface. On the left, there's a sidebar with icons for Extensions, Configuration, Continuous Testing, Dev Services, and Build Metrics. The main area has tabs for Extensions, Configuration, and Testing. The Extensions tab is active, displaying a list of installed extensions: ArC, RESTEasy Reactive, and Eclipse Vert.x. ArC is described as "Build time CDI dependency injection" with sections for Beans (36), Observers (3), and Removed components (69). RESTEasy Reactive is described as "A Jakarta REST implementation utilizing build time processing and Vert.x. This extension is not compatible with the quarkus-resteasy extension, or any of the extensions that depend on it." It includes sections for Endpoint scores, Exception Mappers, and Parameter converter providers. Eclipse Vert.x is described as "Write reactive applications with the Vert.x API". At the bottom, the Testing tab is active, showing a log window with the following content:

```
2023-11-15 10:56:09,522 INFO (Quarkus Main Thread)
--_/_\ \ / / / _ | / _\ \ / / / / / /
-/ / / / / / / | / , _ , < / / / ^ \ \
--\_\ \_\ / / | / / / / / / \_\ / /
2023-11-15 10:56:09,602 INFO [io.qua.boo.run.Timing] (Quarkus Main Thread) quarkus-getting-started 1.0.0-SNAPSHOT on JVM (powered by Quarkus 3.5.1) started in 1.621s. Listening on: http://localhost:8080
2023-11-15 10:56:09,605 INFO [io.qua.boo.run.Timing] (Quarkus Main Thread) Profile dev activated. Live Coding activated.
2023-11-15 10:56:09,605 INFO [io.qua.boo.run.Timing] (Quarkus Main Thread) Installed features: [cdi, resteasy-reactive, smallrye-context-
```

# Developer Joy - DevServices

- Spins up required services
  - Most common Databases, Kafka, OIDC (Keycloak)
  - Enabled when testing or running in dev mode
  - Zero configuration required
  - Quarkus uses TestContainers (docker required)
  - For Kafka Services Quarkus uses Redpanda (Kafka API compatible → ~1sec startup time)
- How do we get this?
  - Add relevant extension (database type, kafka or OIDC)
  - Do not configure a database url, kafka.bootstrap.servers or auth-server-url for dev profile

→ Quarkus does the magic for you

# Developer Joy – Quarkus CLI Tooling

- Quarkus CLI
  - Replacement for maven or gradle commands
  - Create or add extensions, build and start application
  - Installation using jbang (See [CLI-Tooling Guide](#))

```
$ quarkus create app ch.puzzle.quarkustechlab:quarkus-cli-tooling-demo

$ quarkus extension list --installable --search rest-client
Current Quarkus extensions available:

★ ArtifactId                                Extension Name
★ quarkus-rest-client                         REST Client Classic

$ quarkus extension add quarkus-rest-client
[SUCCESS]      Extension io.quarkus:quarkus-rest-client has been installed

$ quarkus dev
```

# Developer Joy

- Fast start (easy bootstrap)
- Live reload
- Easy Configuration
- Documentation & Guides
- Development UI
  - Change configurations, Console log, Extension Integration
- DevServices
- Quarkus CLI

# Agenda - Day 1

- Microservices architecture
- Quarkus introduction
- MicroProfile specification
- RESTful microservices with Quarkus

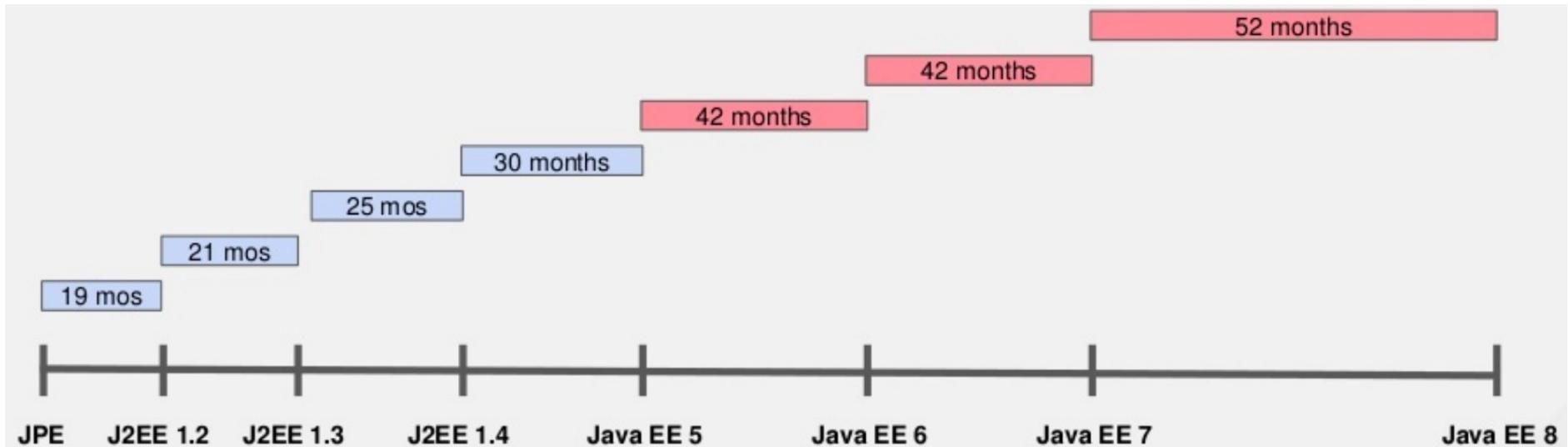
# What is MicroProfile?

«MicroProfile is an open-source community specification for Enterprise Java microservices», [microprofile.io](http://microprofile.io)

«A community of individuals, organizations, and vendors collaborating within an open source Eclipse Foundation Working Group to bring microservices to the Enterprise Java community», [microprofile.io](http://microprofile.io)



# Why MicroProfile?



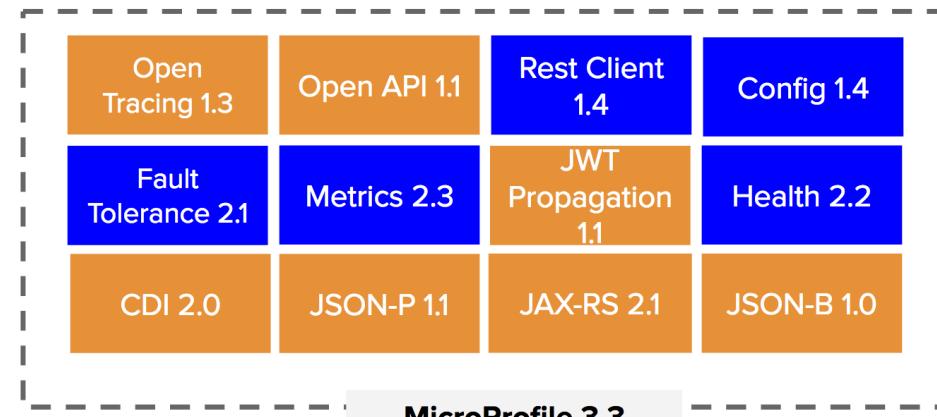
# MicroProfile 1.0 ...

September 2016



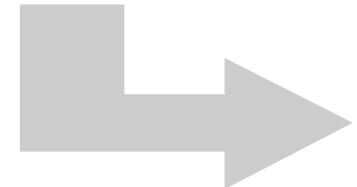
MicroProfile 1.0

February 2020

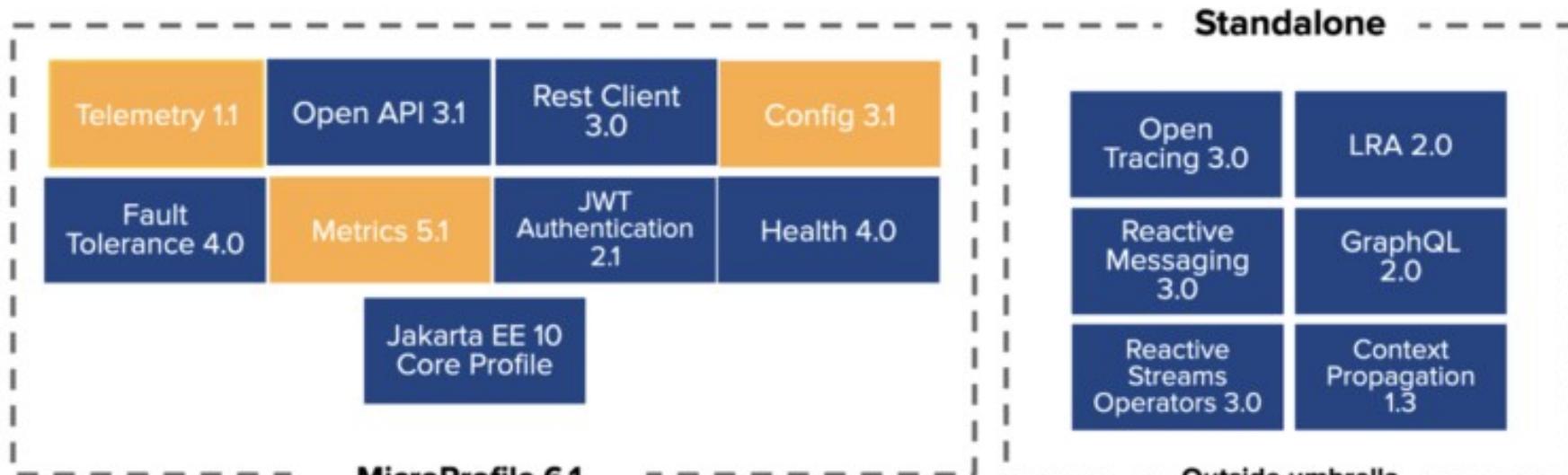


MicroProfile 3.3

- = New
- = Updated
- = No change from last release (MicroProfile 3.2)



# MicroProfile 6.1 (Current)



= New

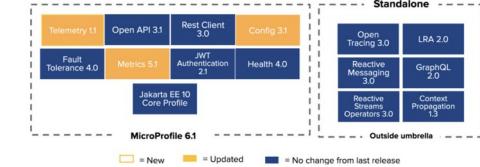


= Updated



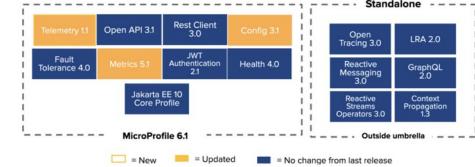
= No change from last release

# MicroProfile Configuration



- Easy and flexible way for application config
- Extendable (write your own config provider)
- Order (highest priority first)
  - System properties
  - Environment variables
  - File named .env in working directory
  - File application.properties placed in \$PWD/config
  - File application.properties (from src/main/resources)
  - File META-INF/microprofile-config.properties
- Details about configuration? → [MP Config 3.1](#)

# MicroProfile Configuration



## Injection @ConfigProperty

```
@Inject @ConfigProperty(name = "greeting.suffix", defaultValue="!")  
String suffix;
```

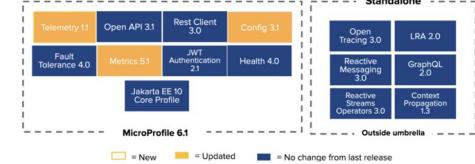
## Programmatically

```
ConfigProvider.getConfig().getValue("greeting.suffix", String.class);
```

## With @ConfigProperty

```
@ConfigProperties(prefix = "greeting")  
public interface GreetingConfiguration {  
  
    @ConfigProperty(defaultValue = "!"")  
    String getSuffix();  
}
```

# MicroProfile Rest Client



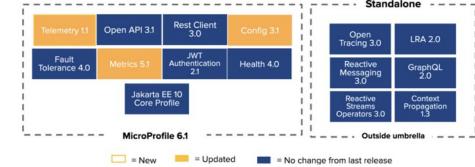
- Make typesafe calls to Rest APIs
- Defined as Java Interface
- Details about configuration? → [Rest Client 3.0](#)

```
@RegisterRestClient(configKey="my-client")
public interface MyServiceClient {
    @GET
    @Path("/greet")
    Response greet();
}
```

With `@ConfigProperty`

```
my-client/mp-rest/url=https://my-greeting-service/rest
my-client/mp-rest/readTimeout=xy
my-client/mp-rest/proxyAddresses=xy (MP 4.0)
```

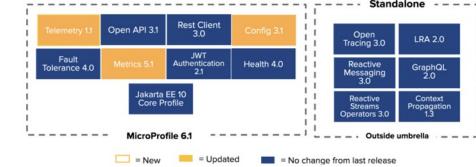
# MicroProfile Metrics



- Add monitoring endpoint /metrics
  - OpenMetrics (text/plain), JSON (application/json)
- Expose metrics from application
- Details about configuration? → [MP Metrics 5.1.0](#)

```
@Counted(name = "processing-counter", absolute = true, tags={"tag1=value1"})
@Timed(name = "processTimer", description = "execution time", unit = MetricUnits.MILLISECONDS)
public void processMessage() { }
```

# MicroProfile Health

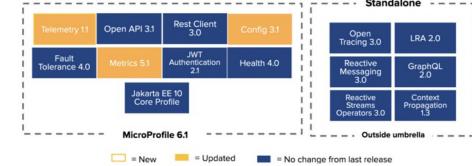


- Validate status and availability of application
- Machine to machine M2M focused
  - Replace unhealthy instances with new health instances
  - Monitored by container platforms
- Provide «Liveness», «Readiness» and «Startup» endpoints `/health{/ready,/live,/started}`
- Details about configuration? → [MP Health 4.0](#)

```
@ApplicationScoped
@Liveness
@Readiness
public class MyCheck implements HealthCheck {

    public HealthCheckResponse call() {
        [...]
    }
}
```

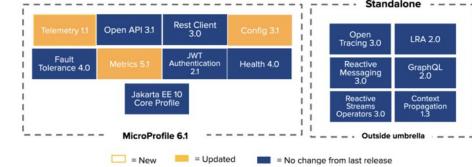
# MicroProfile Fault Tolerance



- Toolkit for resilient applications
  - Timeout, Retry, Fallback, Circuit Breaker, Bulkhead
- Designed to separate execution logic from execution
- Details about configuration? → [MP Fault Tolerance 4.0](#)

```
public interface MyServiceClient {  
    @GET  
    @Path("/greet")  
    // timeout is 400ms  
    @Timeout(400)  
    // 0 - 400ms delay, max 10  
    @Retry(delay = 400, maxDuration= 3200, jitter = 400, maxRetries = 10)  
    // Specify fallback handler class  
    @Fallback(GreetingStringFallbackHandler.class)  
    // Scenario: Success, Failure, Success, Success, Failure, CircuitBreakerOpenException  
    @CircuitBreaker(successThreshold = 10, requestVolumeThreshold = 4, failureRatio=0.5, delay = 1000)  
    // maximum 10 concurrent requests allowed  
    @Bulkhead(10)  
    Response greet();  
}
```

# MicroProfile Telemetry



- Replaces Open Tracing
- Trace request flow across service boundaries
  - Support for JDBC, Kafka or MongoDB available
- Focused on easy instrumentation of services
- RESTful Web Services (server and client) are automatically traced.
  - Annotate methods with `@WithSpan` to generate custom spans
- Details about configuration? → [MP Telemetry 1.1](#)

```
@ApplicationScoped
public class GreetingService {

    @WithSpan("helloSpan")
    public String sayHello() {
        return "hello";
    }
}
```

# SmallRye

«APIs and implementations tailored for Cloud development, including but not limited to, Eclipse MicroProfile», [smallrye.io](https://smallrye.io)

Implementation currently used by:

- Quarkus
- WildFly
- Thorntail
- Open Liberty



# Resources

Eclipse MicroProfile Developer Resources

<https://projects.eclipse.org/projects/technology.microprofile/developer>

MicroProfile Specifications available at Github (e.g. Reactive Messaging):

<https://github.com/eclipse/microprofile-reactive-messaging>

SmallRye MicroProfile Implementation Documentation:

<https://smallrye.io/docs/index/index.html>

SmallRye Reactive Messaging Documentation:

<https://smallrye.io/smallrye-reactive-messaging>

MicroProfile 6.1 Release Presentation:

<https://drive.google.com/drive/folders/1NpSIxyEa4jtA1c-gVqexuY07ij8SHnJb>

# Agenda - Day 1

- Microservices architecture
- Quarkus introduction
- MicroProfile specification
- RESTful microservices with Quarkus

# RESTful microservices with Quarkus

Build two microservices with a RESTful endpoint and a RESTful client

## Data Producer

- Implement RESTful endpoint /data with JSON data
- Produce random SensorMeasurement

## Data Consumer

- Consumes SensorMeasurement from producers REST endpoint /data
- Implement RESTful client for /data endpoint

Hint: use the quarkus dev mode while developing. Enjoy.

# RESTful microservices with Quarkus

Add resiliency with MicroProfile fault-tolerance to our microservices

- Add some random errors
- Slow down the producing of the SensorMeasurement

How can the Data Consumer be more resilient to errors on the producer?

- Try fault-tolerance features retry and timeout

# RESTful microservices with Quarkus

Implement microservices in a reactive way

- Stream the data with ServerSentEvents
- Use Reactive JDBC for database calls

## IMPERATIVE

```
@Inject  
SayService say;  
  
@GET  
@Produces(MediaType.TEXT_PLAIN)  
public String hello() {  
    return say.hello();  
}
```

## REACTIVE

```
@Inject @Channel("kafka")  
Publisher<String> reactiveSay;  
  
@GET  
@Produces(MediaType.SERVER_SENT_EVENTS)  
public Publisher<String> stream() {  
    return reactiveSay;  
}
```

# Day 2

# Agenda - Day 2

- 8 fallacies of distributed computing
- Cloud patterns
- Continuous integration and delivery
- Event driven architecture and messaging
- Observability – Metrics and Tracing
- Writing your own Quarkus extension

# Questions Day 1?

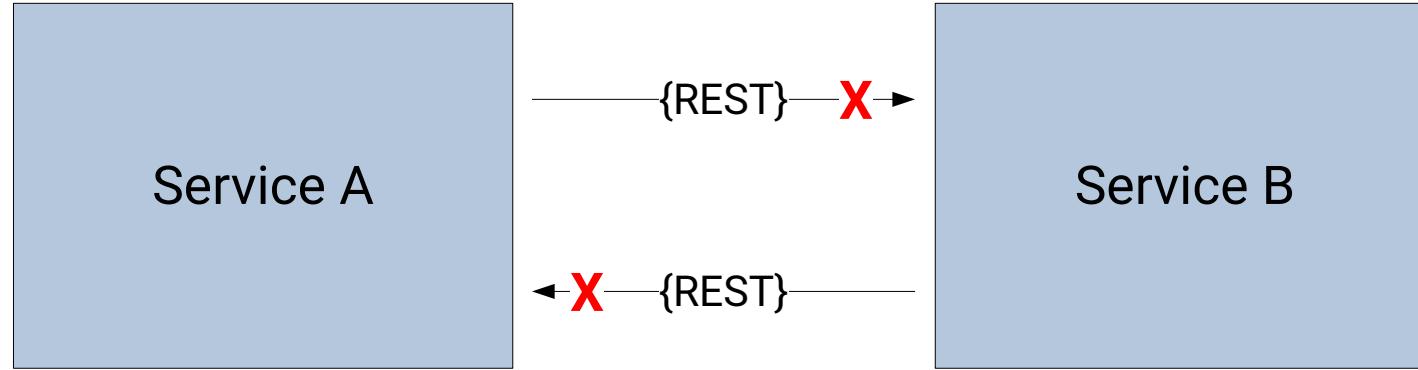
# Agenda - Day 2

- 8 fallacies of distributed computing
- Cloud patterns
- Continuous integration and delivery
- Event driven architecture and messaging
- Observability – Metrics and Tracing
- Writing your own Quarkus extension

# 8 fallacies of distributed computing

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous.

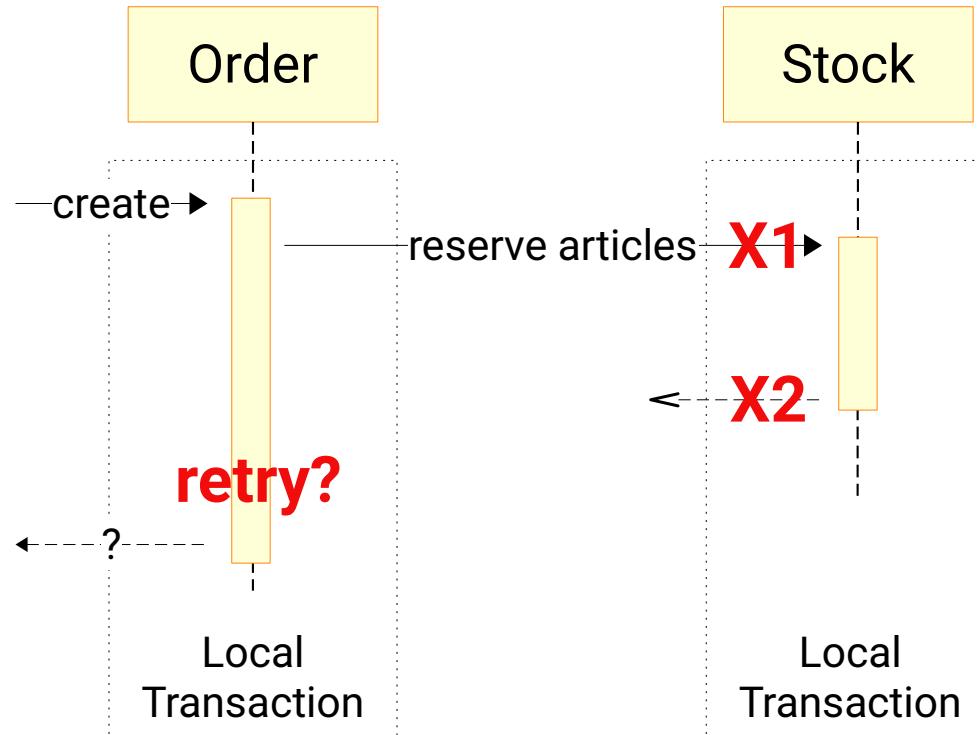
# 1. The network is reliable



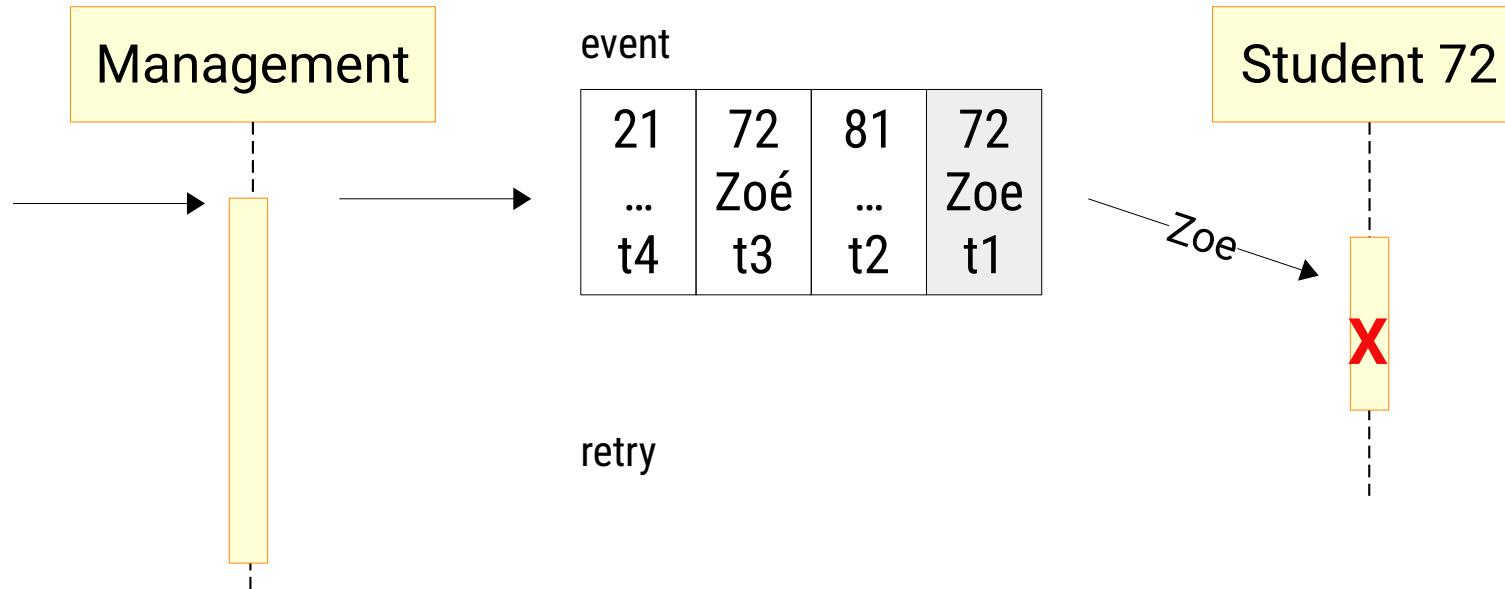
**Problem:** The network is not reliable. Requests will fail.

**Options:** Automatic Retries, Message Queues, Timeouts, Circuit-Breaker

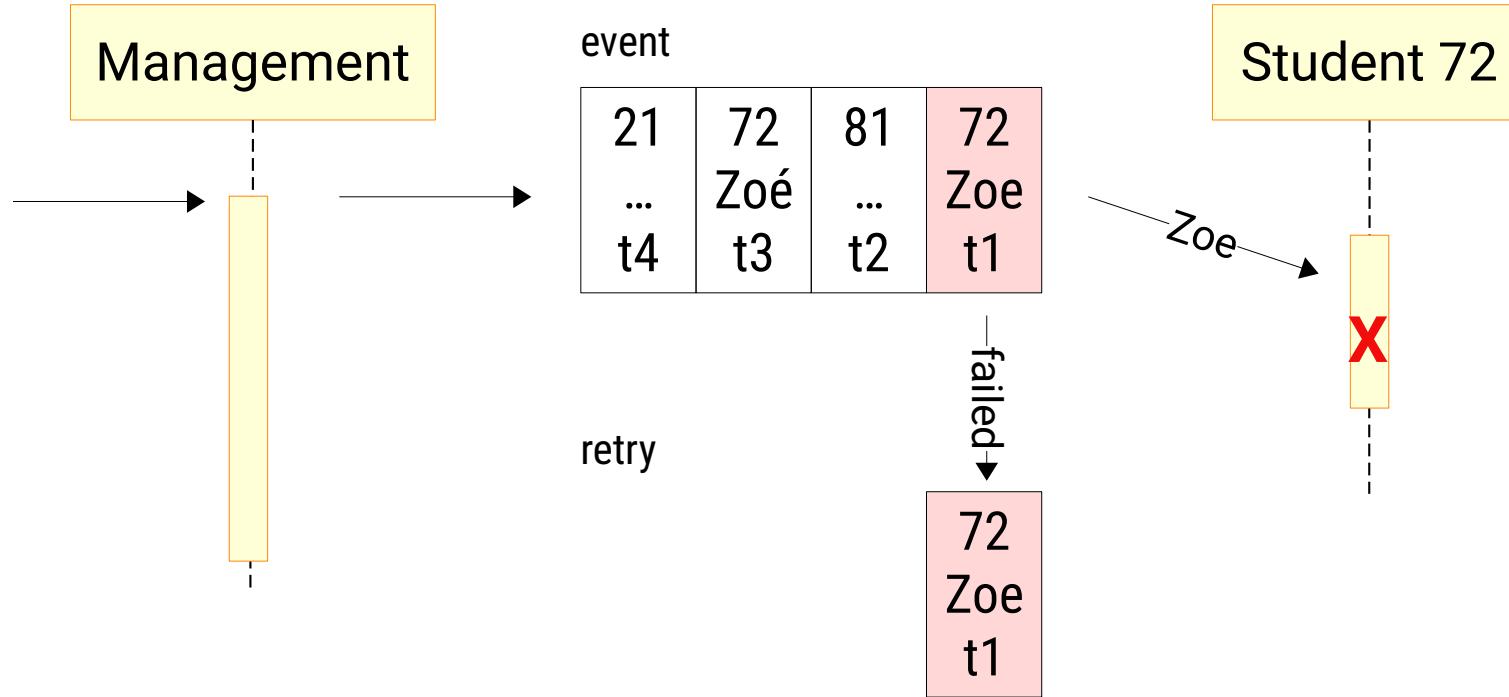
# Error Handling #1 - REST



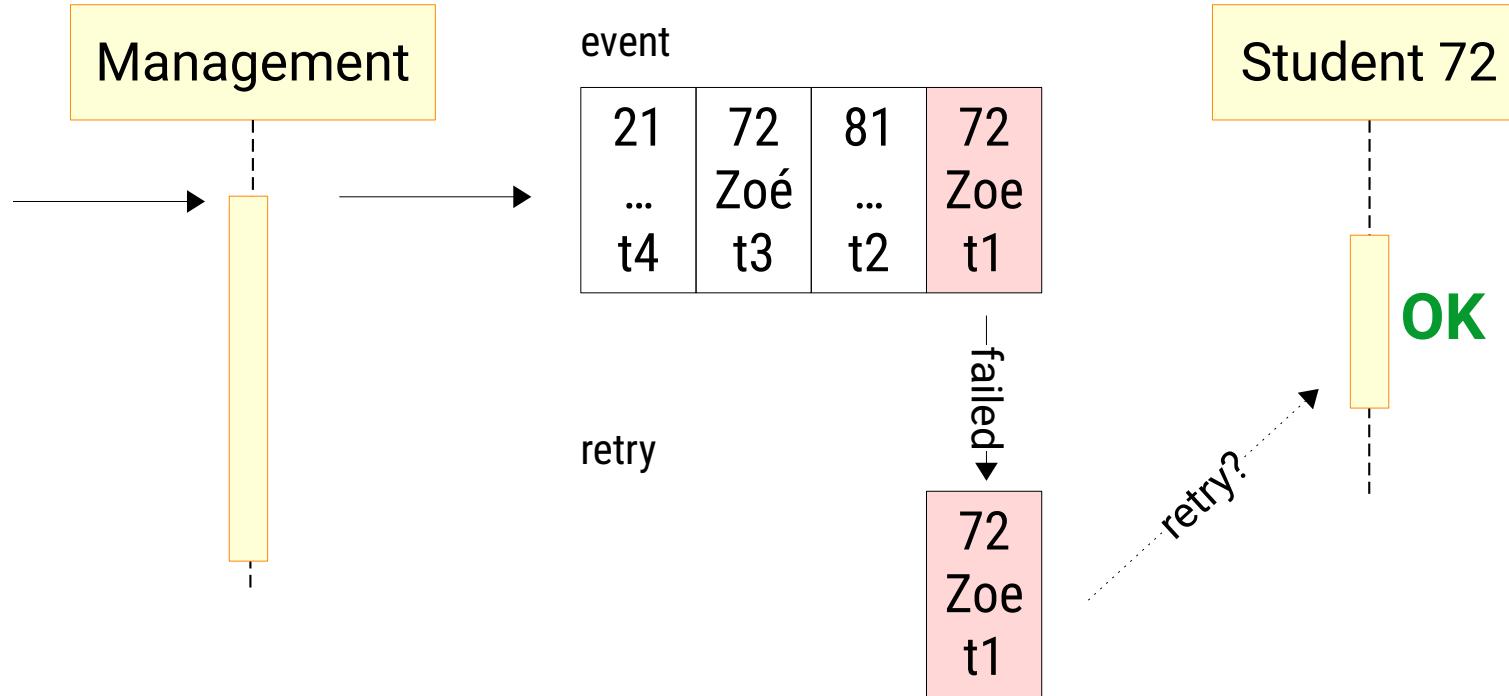
# Error Handling #2 - Events



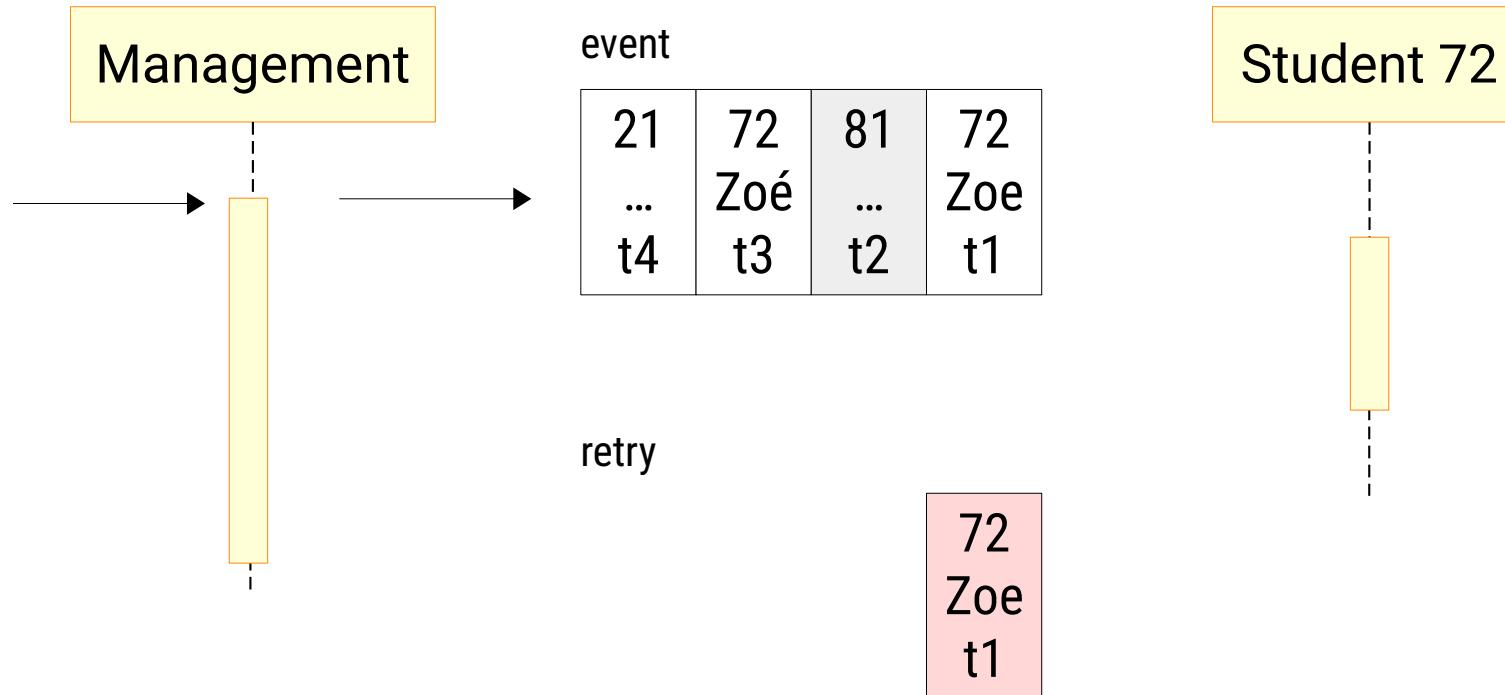
# Error Handling #2 - Events



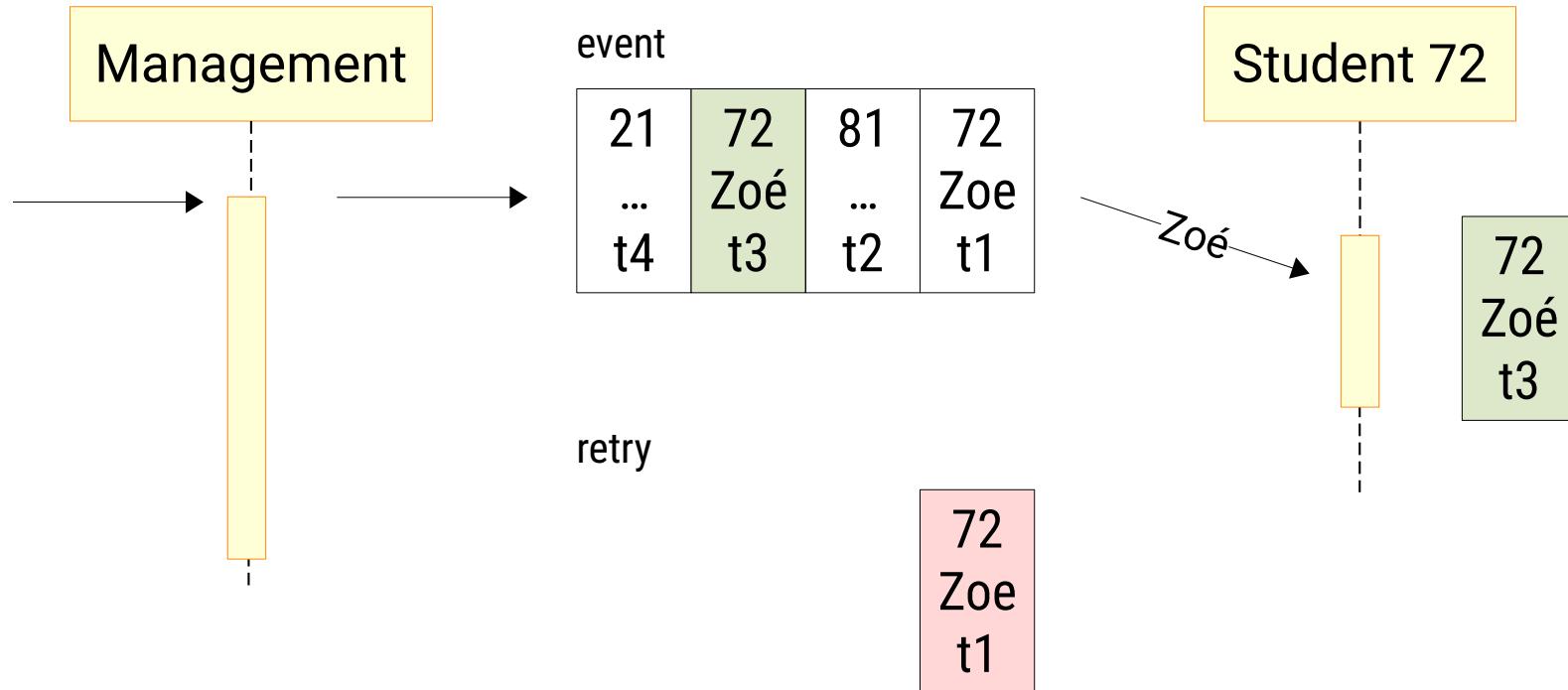
# Error Handling #2 - Events



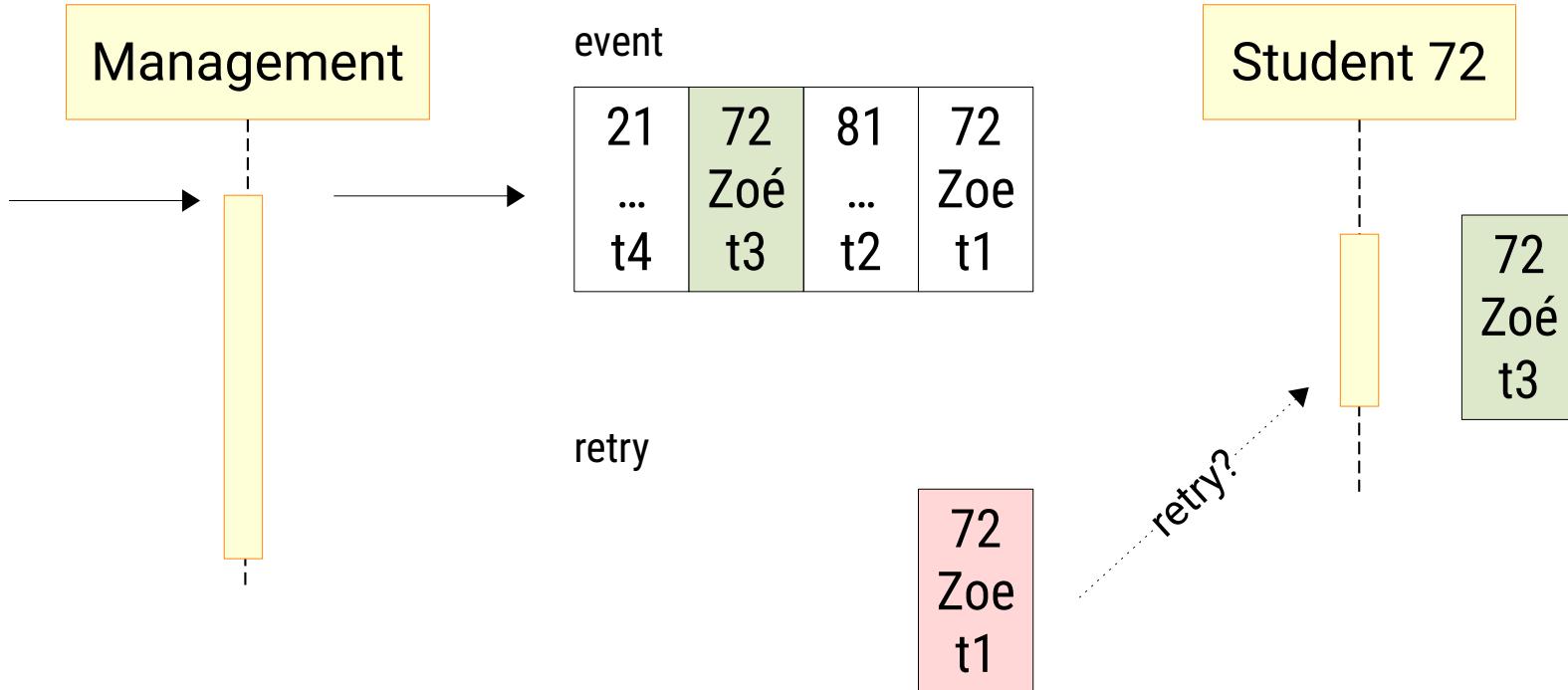
# Error Handling #2 - Events



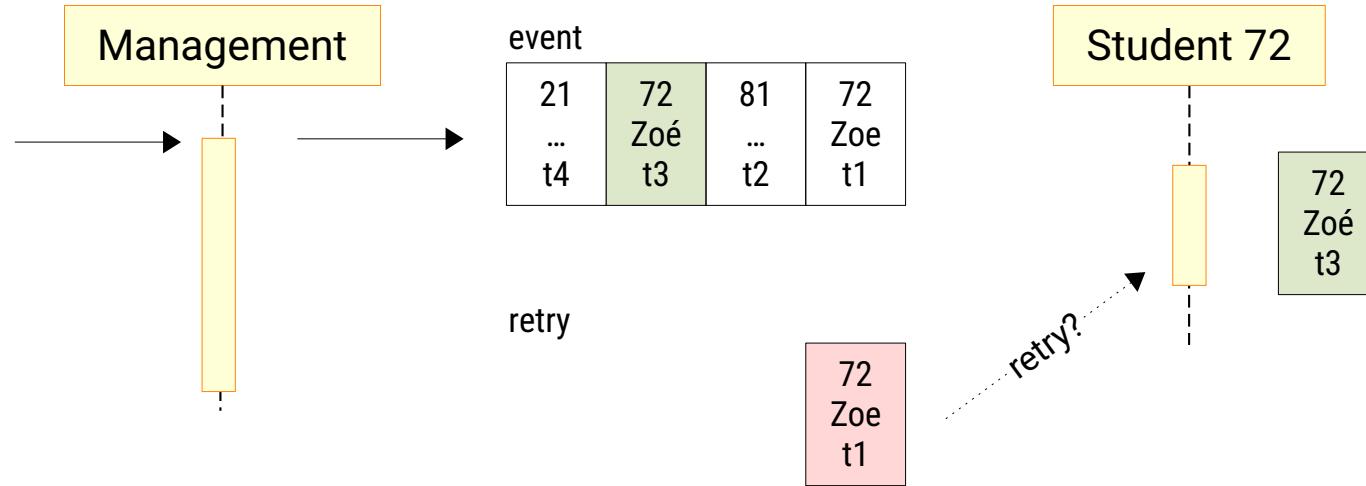
# Error Handling #2 - Events



# Error Handling #2 - Events



# Error Handling #2 - Events



**Can ignore event**  
Name change

$T1 < T3 \rightarrow$  Drop

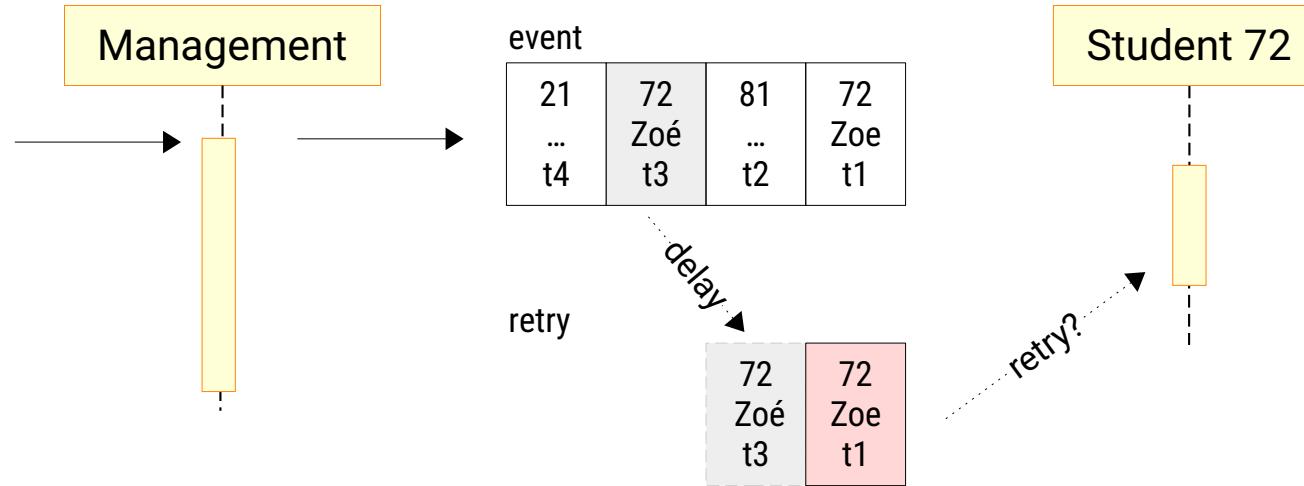
**Order does not matter**  
Coffee order

→ Just retry/apply

**Order matters**  
Bank account (overdraft penalty)

$T3 \rightarrow$  retry behind T1

# Error Handling #2 - Events



**Can ignore event**  
Name change

$T1 < T3 \rightarrow$  Drop

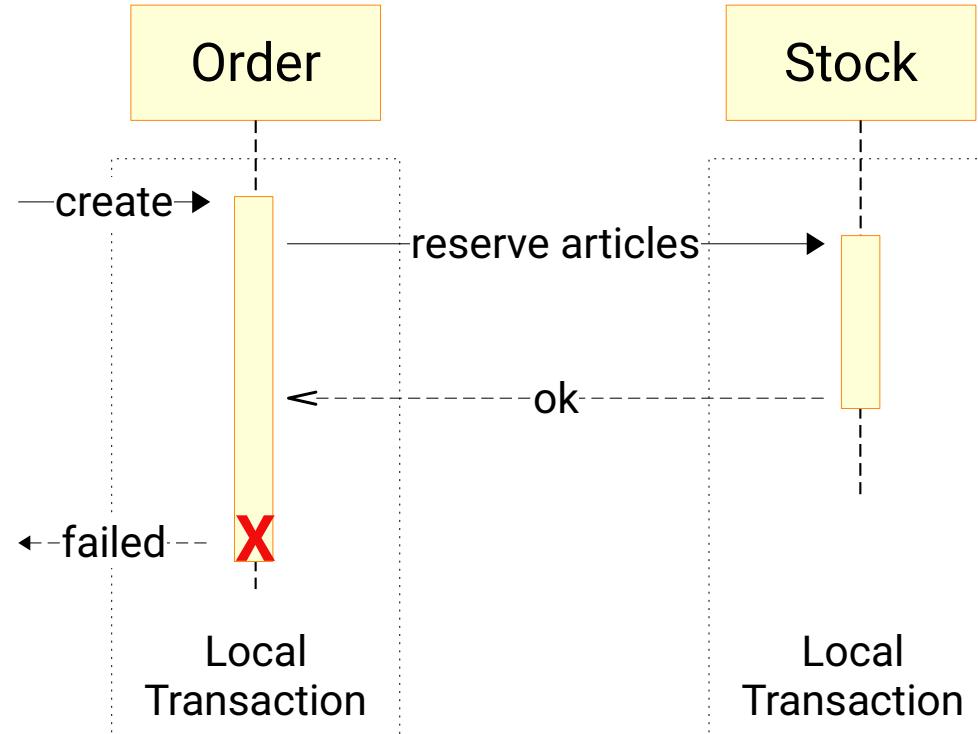
**Order does not matter**  
Coffee order

→ Just retry/apply

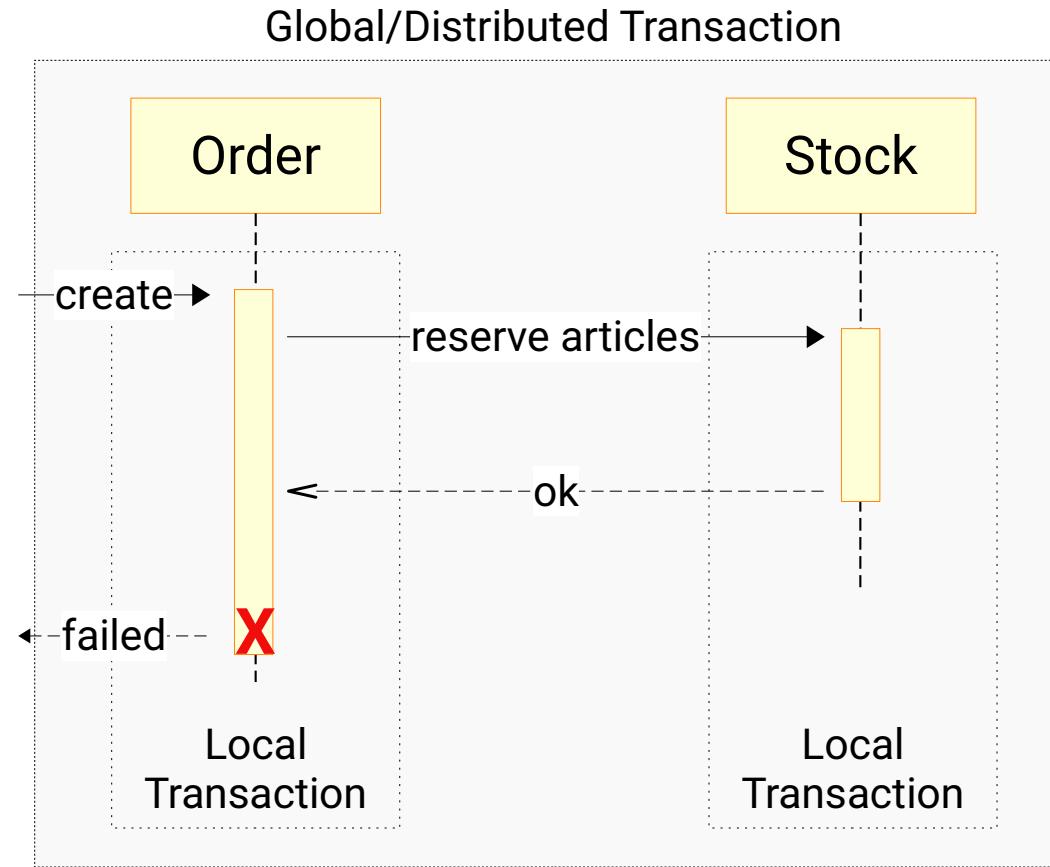
**Order matters**  
Bank account (overdraft penalty)

$T3 \rightarrow$  retry behind T1

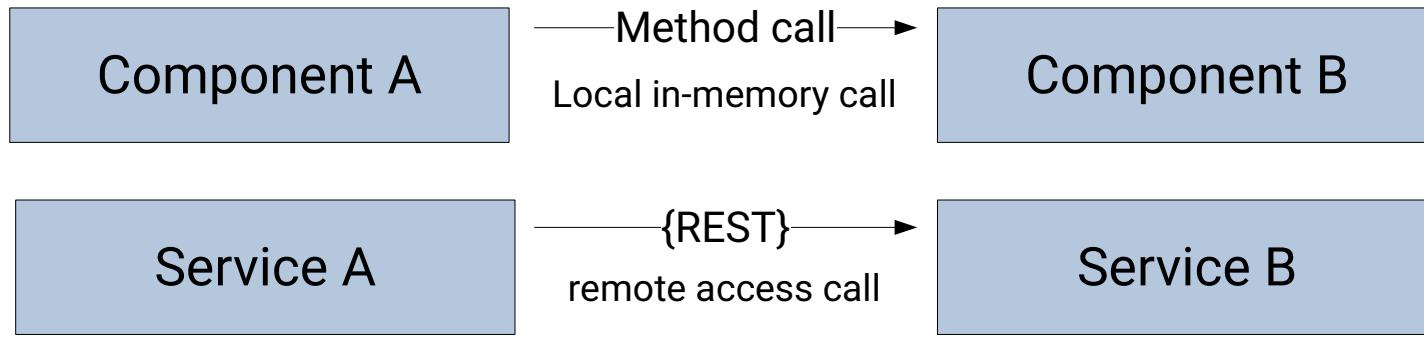
# Data Consistency - Transactions



# Data Consistency - Transactions



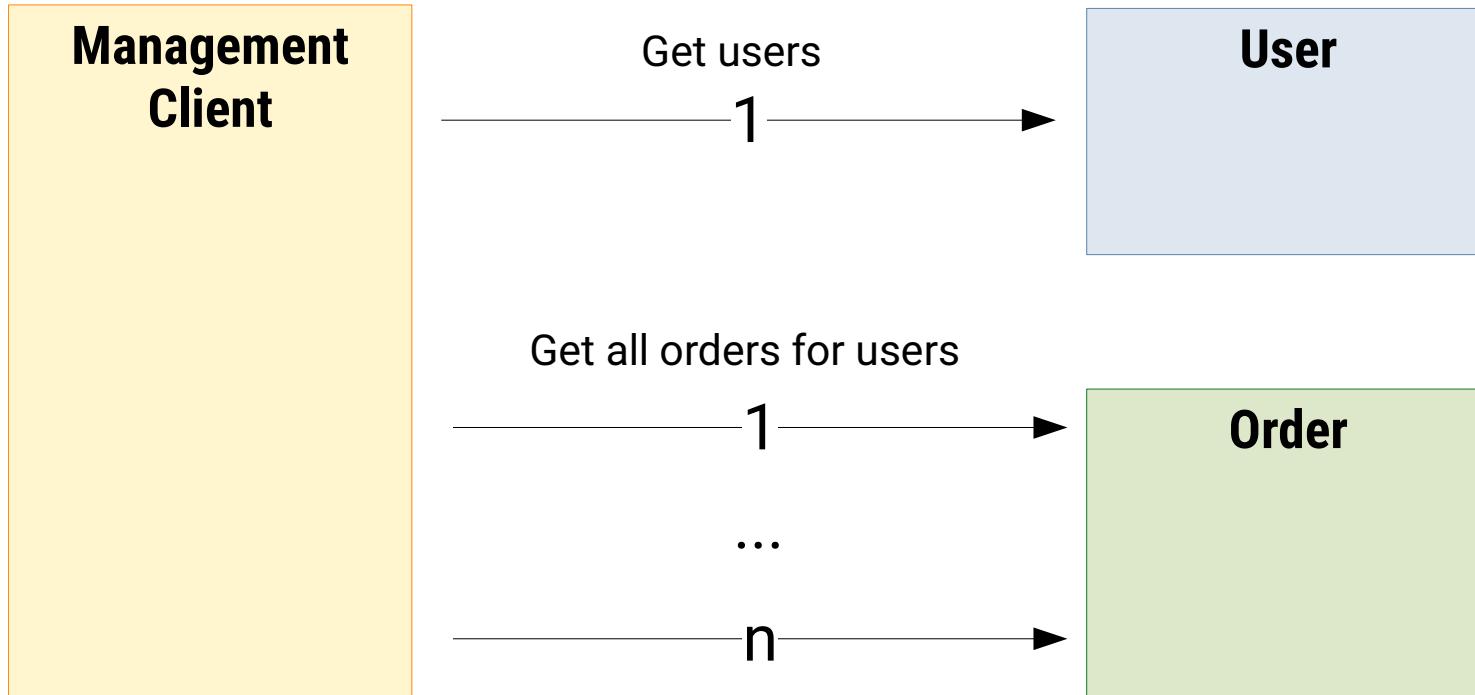
## 2. Latency is zero



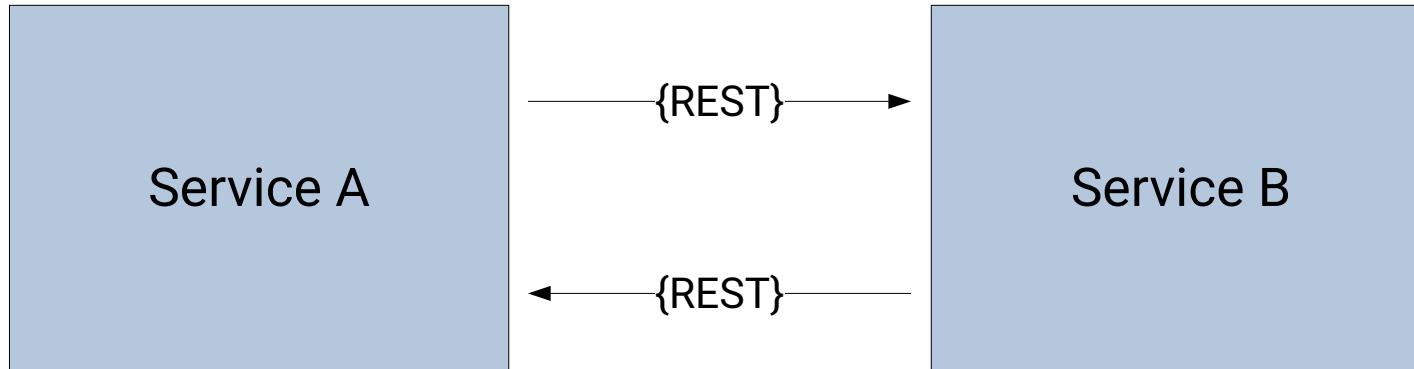
**Problem:** Network calls are orders of magnitude slower than in-memory calls

**Options:** Caching, Bulk Requests, Availability Zones

# Select N+1 Problem



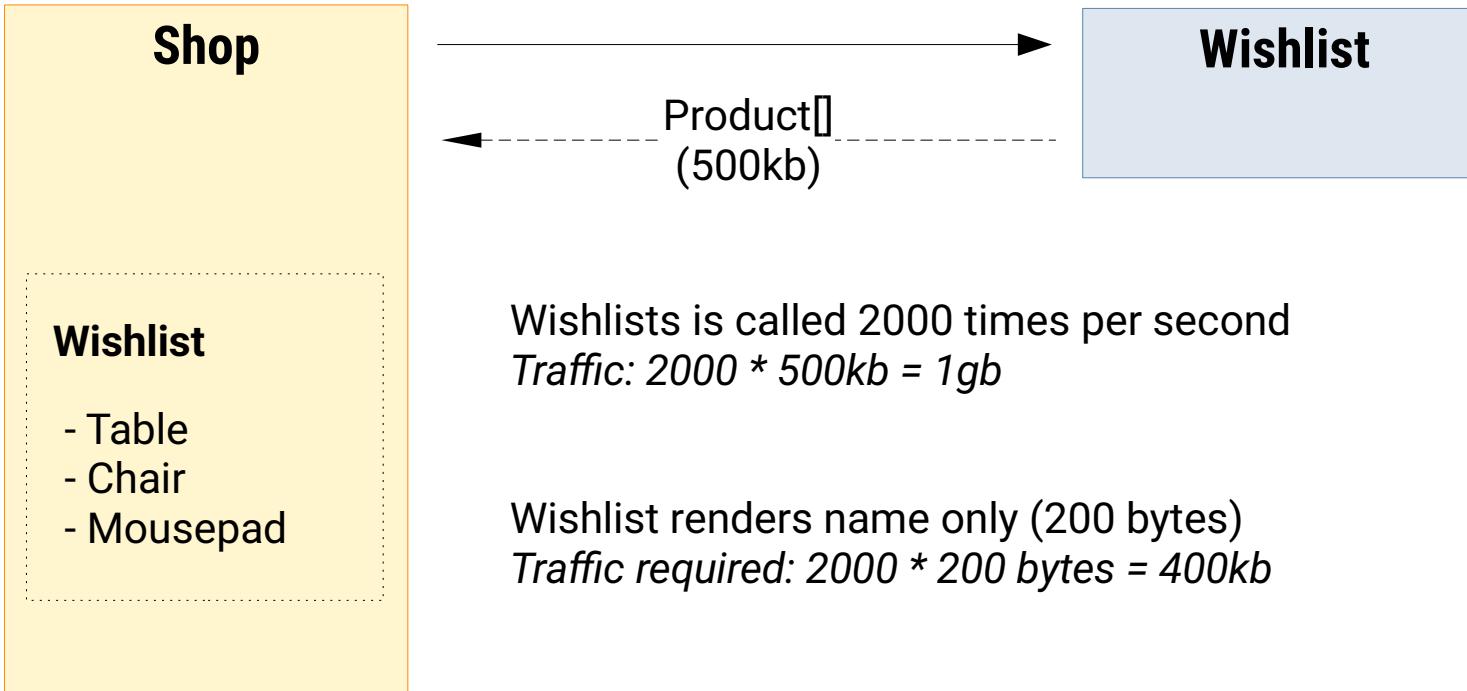
### 3. Bandwidth is infinite



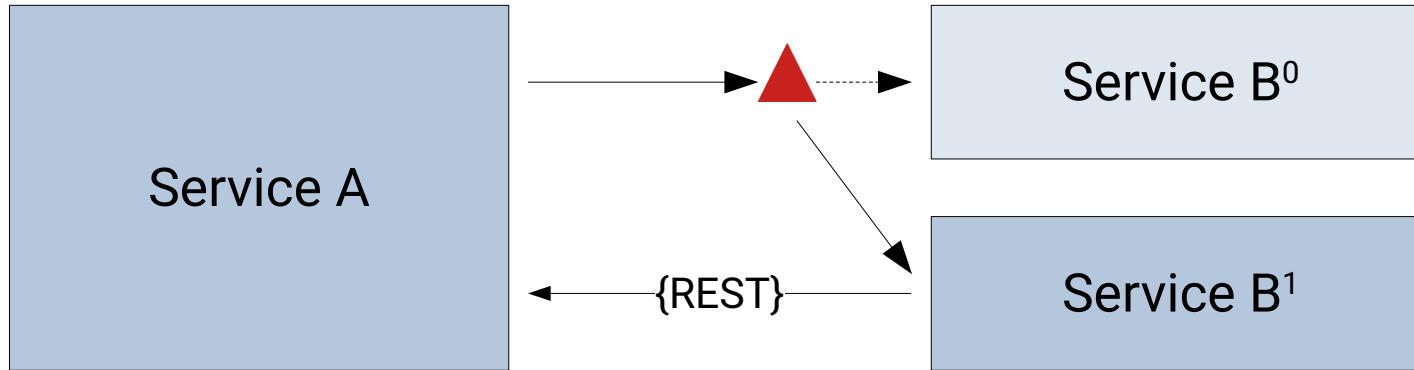
**Problem:** Bandwidth and also amount of connections is not infinite

**Options:** Small payloads, Optimized data formats, DTOs, Throttling

# Bandwidth



## 5. Topology does not change

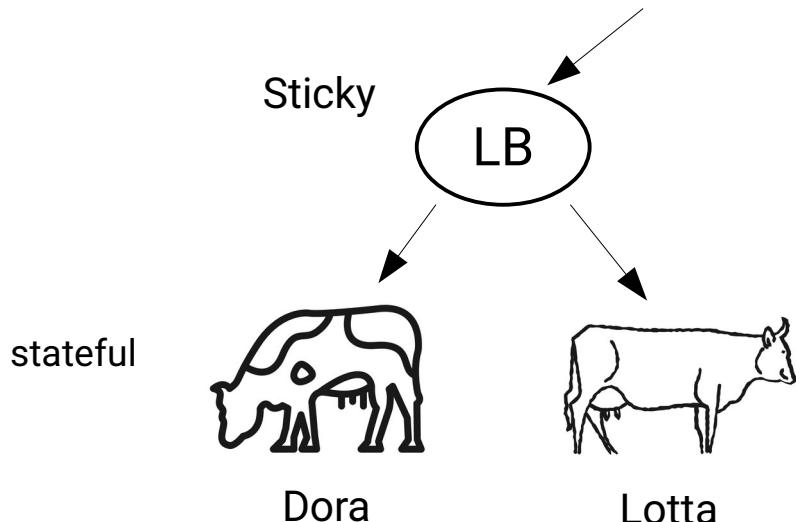


**Problem:** Topology changes constantly. Service die and restart.

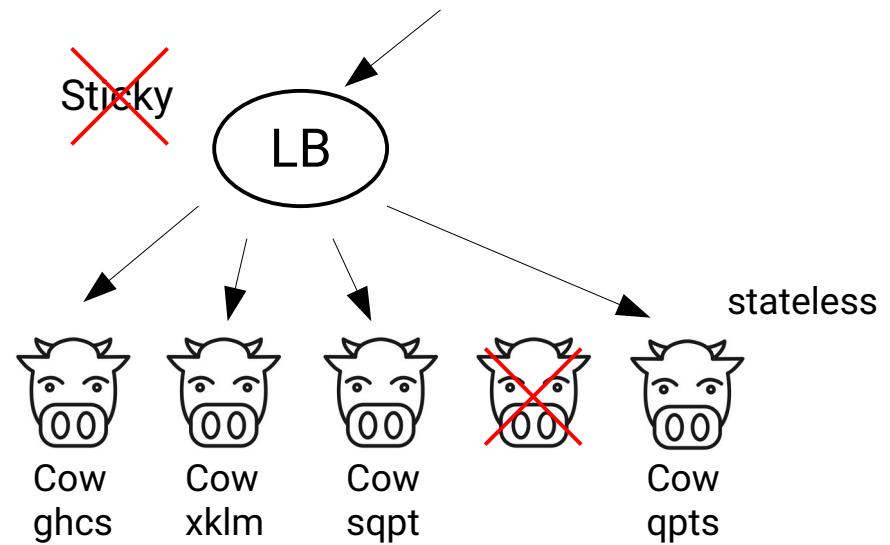
**Options:** Rely on DNS, No IP hardcoding, Service Discovery

# Pets vs. Cattle

Traditional Applications  
(Pets)



Cloud Native Applications  
(Cattle)



# Agenda - Day 2

- 8 fallacies of distributed computing
- Cloud patterns
- Continuous integration and delivery
- Event driven architecture and messaging
- Observability – Metrics and Tracing
- Writing your own Quarkus extension

# Cloud patterns

- The Twelve-Factor App
- Design for Failure (Retry, Timeout, Circuitbreaker)
- Microservice Communication
- Orchestration vs. Choreography
- Event Driven Architecture
- Transaction management

# 12 factor application

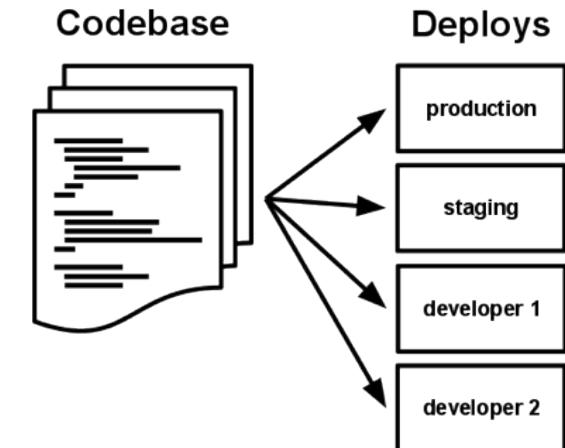
«The Twelve-Factor App methodology is a methodology for building software-as-a-service applications. These best practices are designed to enable applications to be built with portability and resilience when deployed to the web.[1]»

# 12 factor application

- I. Codebase - One codebase tracked in revision control, many deploys
- II. Dependencies - Explicitly declare and isolate dependencies
- III. Config - Store config in the environment
- IV. Backing services - Treat backing services as attached resources
- V. Build, release, run - Strictly separate build and run stages
- VI. Processes - Execute the app as one or more stateless processes
- VII. Port binding - Export services via port binding
- VIII. Concurrency - Scale out via the process model
- IX. Disposability - Maximize robustness with fast startup and graceful shutdown
- X. Dev/prod parity - Keep development, staging, and production as similar as possible
- XI. Logs - Treat logs as event streams
- XII. Admin processes - Run admin/management tasks as one-off processes

# I. Codebase

- One codebase for a single deployed service
- Each microservice might have separated codebase
- One codebase many deployments
- Deployments should differ in config



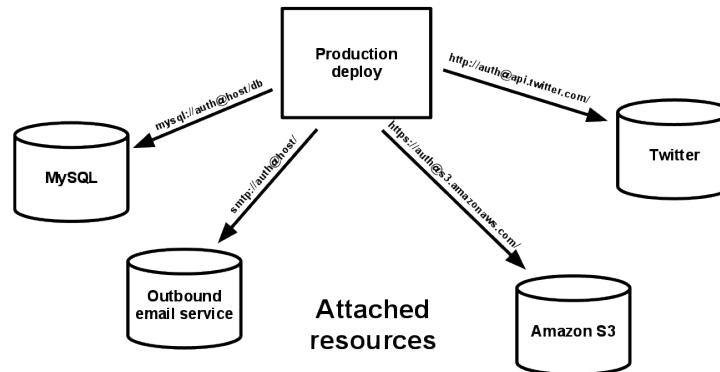
# III. Config

- Separation of config from code
- Environment variables define deployments
- Modern frameworks support this approach

```
order:  
  image: kafka-order:latest  
  ports:  
    - 8080:8080  
  networks:  
    - kafka  
  depends_on:  
    - order-db  
    - kafka  
  environment:  
    - QUARKUS_HTTP_PORT=8080  
    - QUARKUS_DATASOURCE_JDBC_URL=jdbc:tracing:postgresql://order-db:5432/admin  
    - KAFKA_BOOTSTRAP_SERVERS=kafka:9092
```

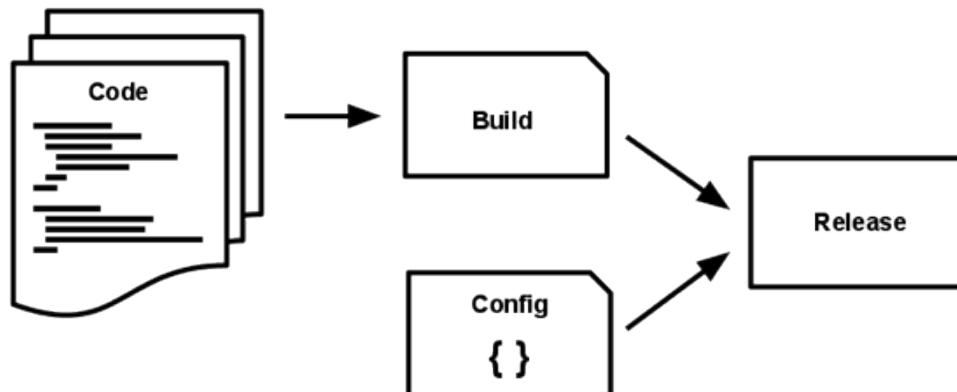
# IV. Backing services

- Make backing services (Databases, message brokers, etc) attachable resource
- Loosely coupled even with third party services
- Attach and detach or substitute with ease



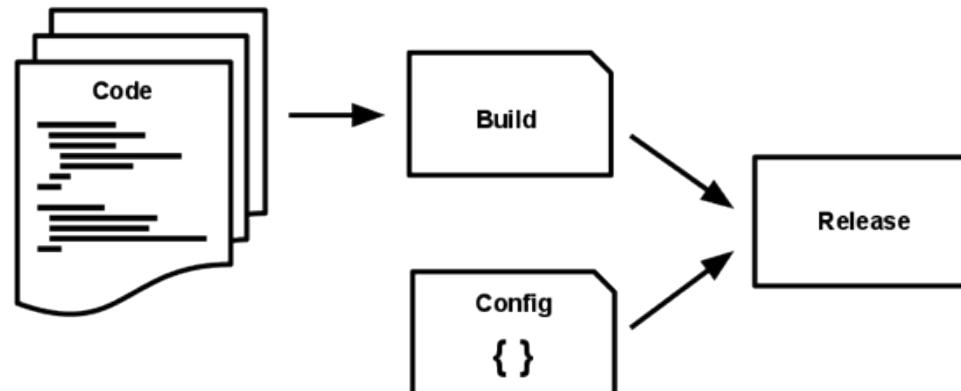
## V. Build, release, run

- Transform codebase into deploy with three stages
- Use the same artifact (Docker image) for all environments
- Use a single pipeline to track all steps



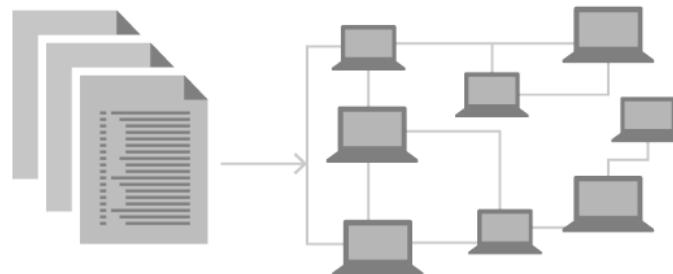
# V. Build

- Use automated pipelines to build and test codebase
- Create single deployable for all deployment environments
- Prepare your deployable to be parameterized for each deployment



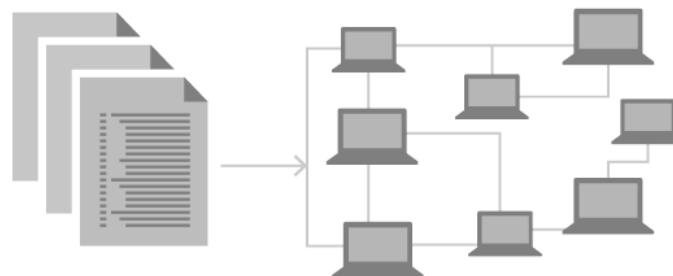
# V. Release

- Deliver the parameterizable deployable to environment
- Use environment variables to fill parameters to your needs
- Release fast and often
- Best practice: Infrastructure as code



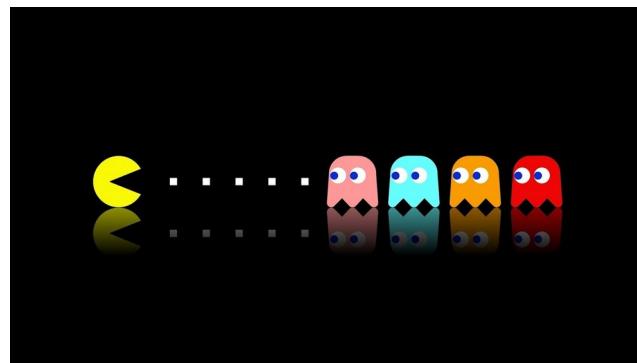
## V. Run

- Delivered deployable should run on each environment
- Tag and track deployed applications via build or release ids



# IX. Disposability

- Scalable applications should have low startup times
- Stopped and started with low cost
- Robustness against sudden death



## X. Dev/Prod parity

- Environments (dev/prod) should use same deployable
- As similar as possible
- Difference in environment variables preferably via code
- Tests will have meaning

# Cloud patterns

- The Twelve-Factor App
- Design for Failure (Retry, Timeout, Circuitbreaker)
- Microservice Communication
- Orchestration vs. Choreography
- Event Driven Architecture
- Transaction management

# Design for failure

- Microservices need a lot of communication
- Communication brings external coupling
- External coupling will vulnerable to failure
- Use patterns to create robustness

# Retry

- Microservice A calls a remote service B
- B is not available (can and will happen)
- Call from Microservice A will be retried after a short delay and will hopefully succeed

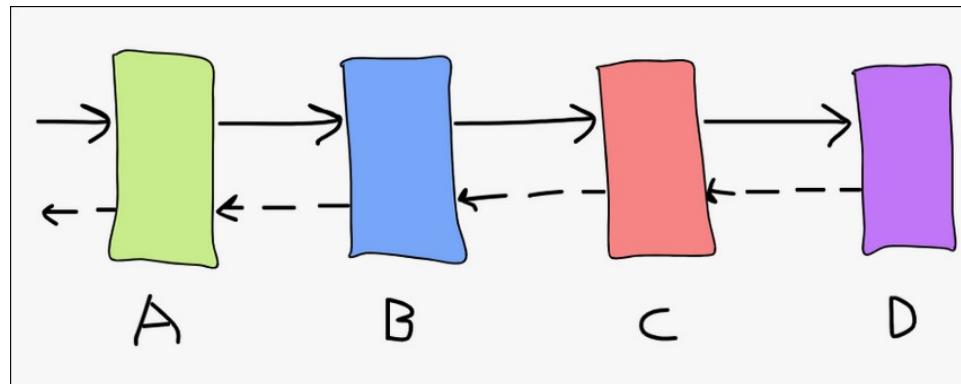


# Timeout

- Microservice A calls a remote service B
- B experiences a lot of load and has longer response time
- A defines a timeout period to wait until the remote call will be dismissed
- Careful with combining retries and timeouts

# Retry storms

- Number of retries = 3
- A calls B, B calls C, C calls D
- D responses with 100% errors
- B will face 300% load, C 900% load, D 2700% load



# Fallback

- Fallbacks can be defined when alternatives are available
- Whenever a remote source fails, repeat on fallback
- Example: Use last cached result instead of new query
- Careful compromise between robustness and vulnerability and loss of consistency

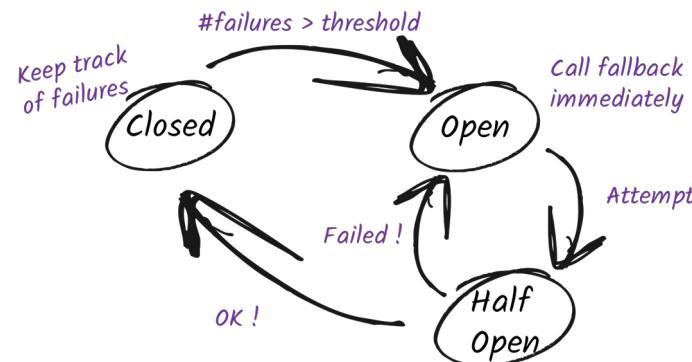


# Circuit Breaker

- Real world analogy to electrical circuit breaker
- Resources that fail often in a defined period of time will open a circuit breaker
- Continue operation without failure cascading through entire system

# Circuit Breaker

- Failures will trigger circuit breaker to open
- If circuit breaker is open, calls will fail immediately, fallback will handle
- After succession, circuit breaker will be half open and retry
- If request succeeds circuit breaker will close and operation will continue normally



# Cloud patterns

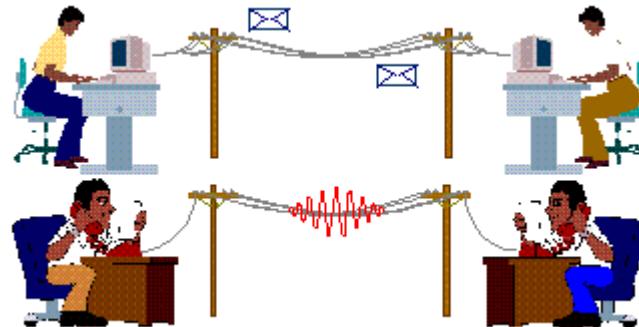
- The Twelve-Factor App
- Design for Failure (Retry, Timeout, Circuitbreaker)
- Microservice Communication
- Orchestration vs. Choreography
- Event Driven Architecture
- Transaction management

# Microservice Communication

- Microservice heavily reliant on communication
- More distribution brings more communication
- Synchronous / Asynchronous protocols

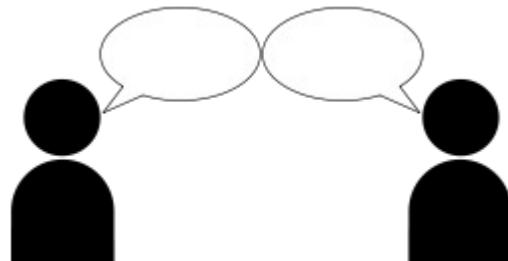
# Synchronous vs Asynchronous

- Synchronous – Telegraph
- Asynchronous - Email



# Synchronous communication

- Well known
- Easy implemented
- Hard to scale



# Synchronous: REST (HTTP)

- REST communication simple and efficient
- Transition from monolithic approach easy
- Fault tolerance patterns widely supported
- Brings tighter coupling
- Blocking operations

# Asynchronous: Messaging

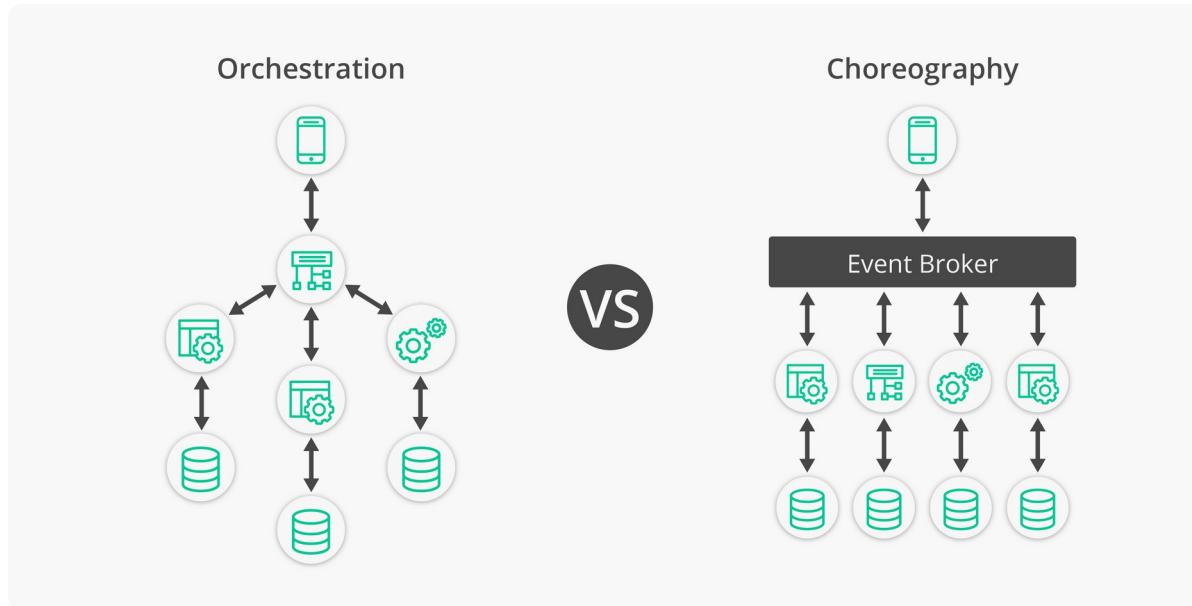
- Loose coupling
- Non-blocking operations
- Simple to scale
- Responsibility shift to broker

# Cloud patterns

- The Twelve-Factor App
- Design for Failure (Retry, Timeout, Circuitbreaker)
- Microservice Communication
- Orchestration vs. Choreography
- Event Driven Architecture
- Transaction management

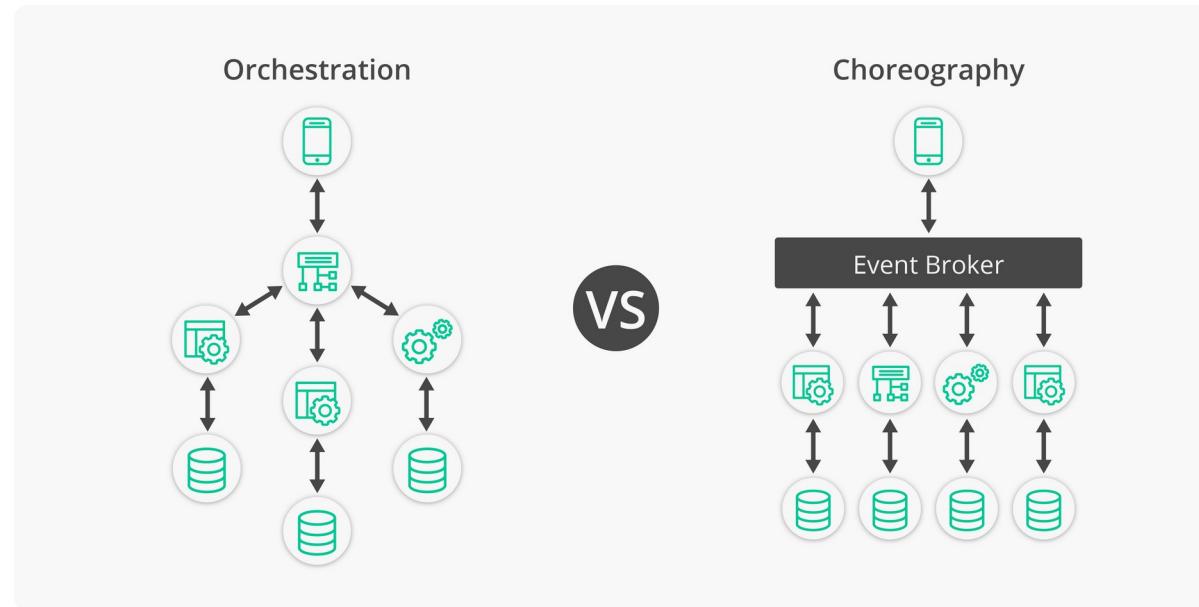
# Orchestration vs. Choreography

- Different approach for communication
- Responsibility question



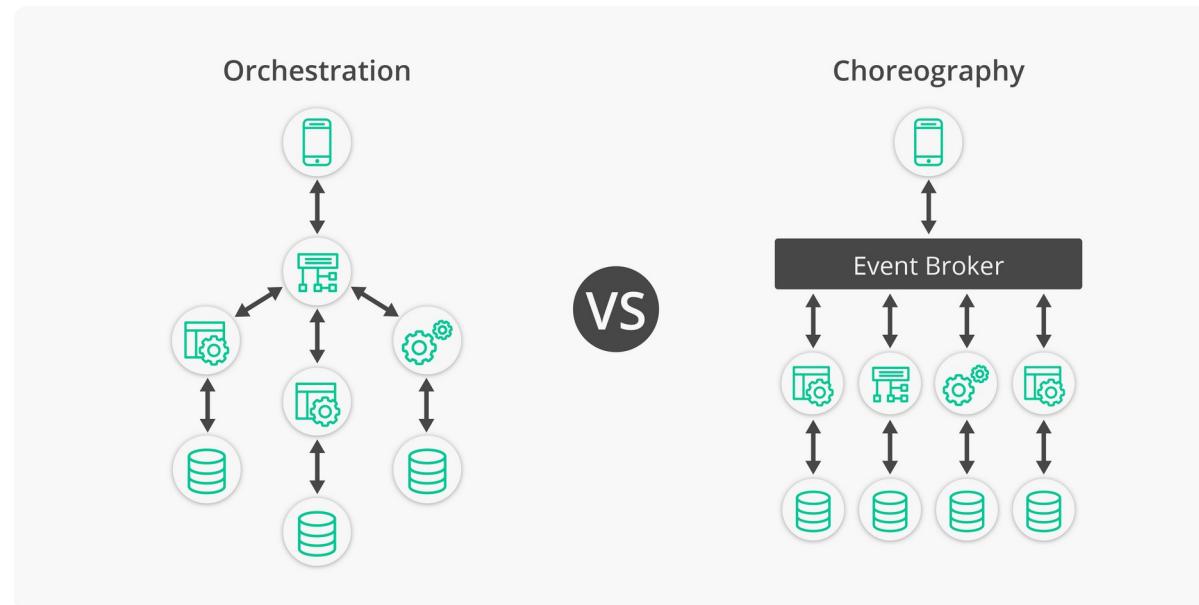
# Choreography

- Each part knows its responsibility
- Microservice knows the workflow



# Orchestration

- Central orchestrator has responsibility
- Controls general workflow within the system



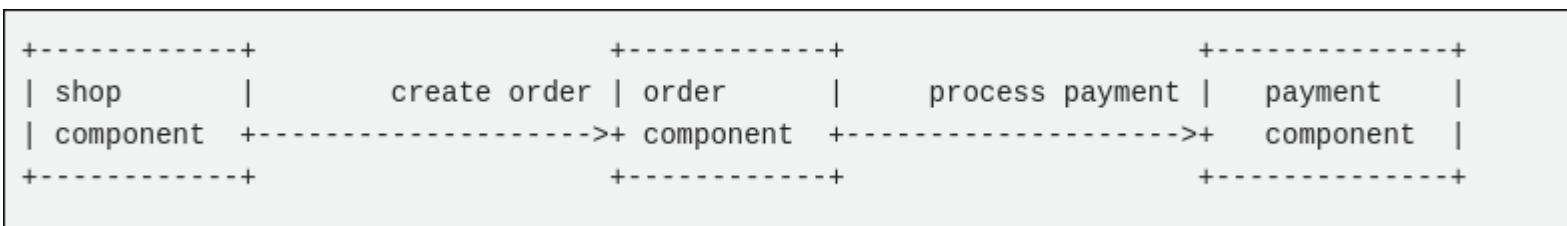
# Cloud patterns

- The Twelve-Factor App
- Design for Failure (Retry, Timeout, Circuitbreaker)
- Microservice Communication
- Orchestration vs. Choreography
- Event Driven Architecture
- Transaction management

# Event Driven Architecture

- Design tends to be imperative
- Creates coupling direct between services
- Services need to know their communication partner

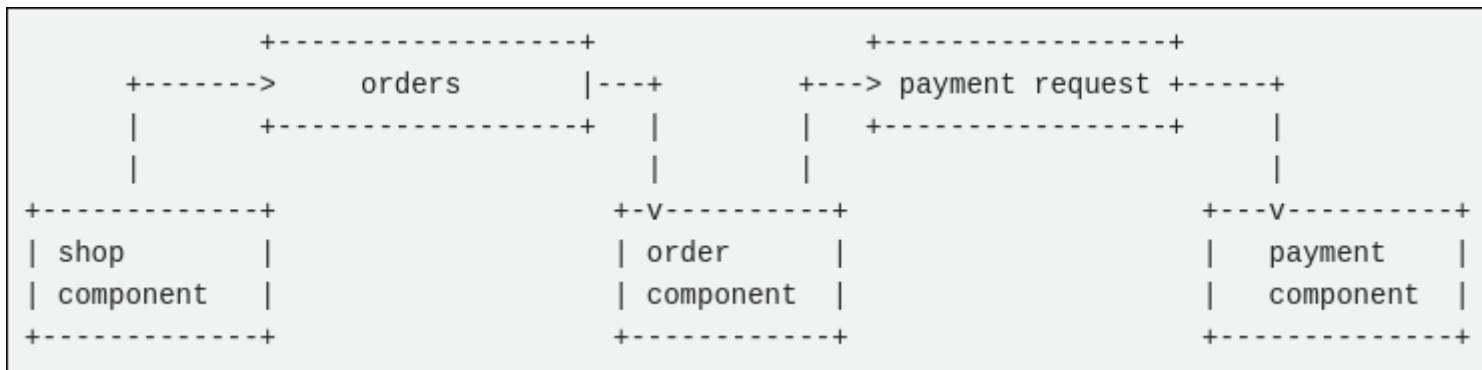
# Event Driven Architecture



# Event Driven Architecture

- Take a look from another perspective
- Call from Shop to Order can be viewed as an event
- Workflow becomes reactive instead of imperative
- Central broker takes responsibility for delivering events
- Single source of truth in event bus

# Event Driven Architecture



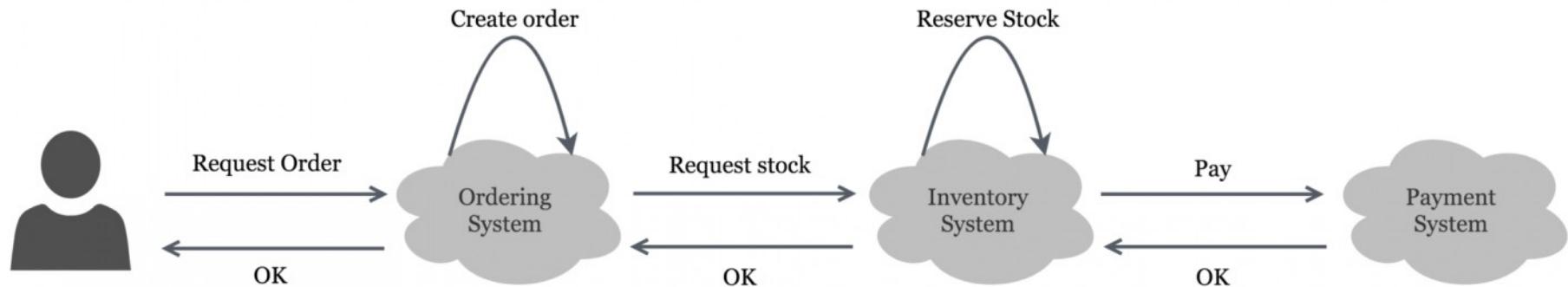
# Transaction management

- What are distributed transaction?
- Patterns for distributed transactions
- Saga pattern

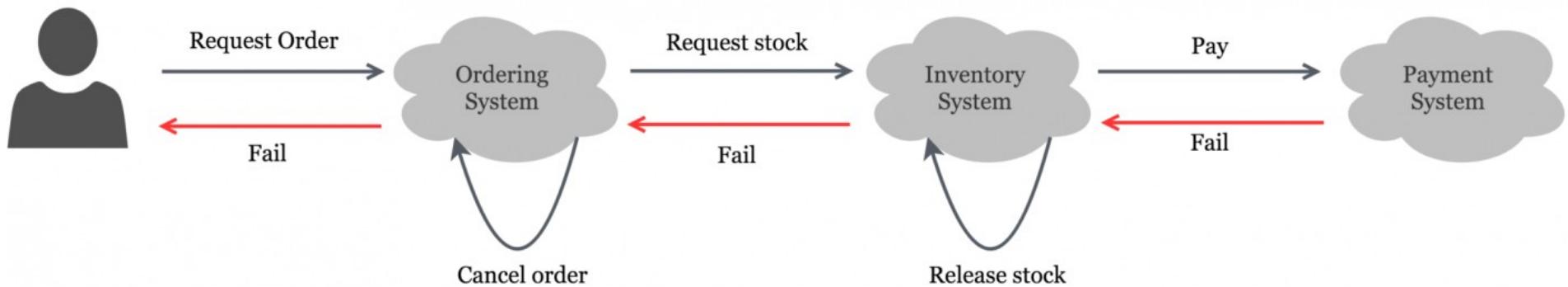
# Transaction management

- Decentralization from workflow generates problem
- ACID (atomicity, consistency, isolation, durability)
- Several patterns how to manage transactions
- Best practice: Database per service

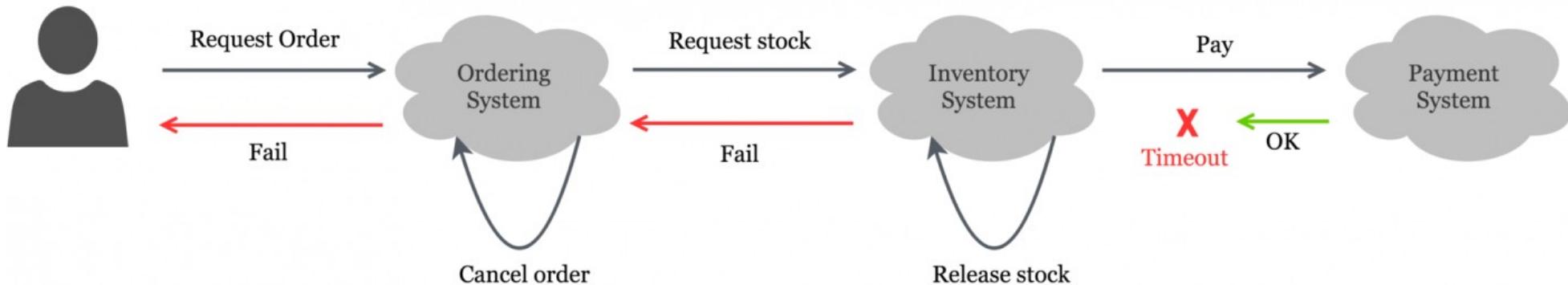
# Transaction management



# Transaction management



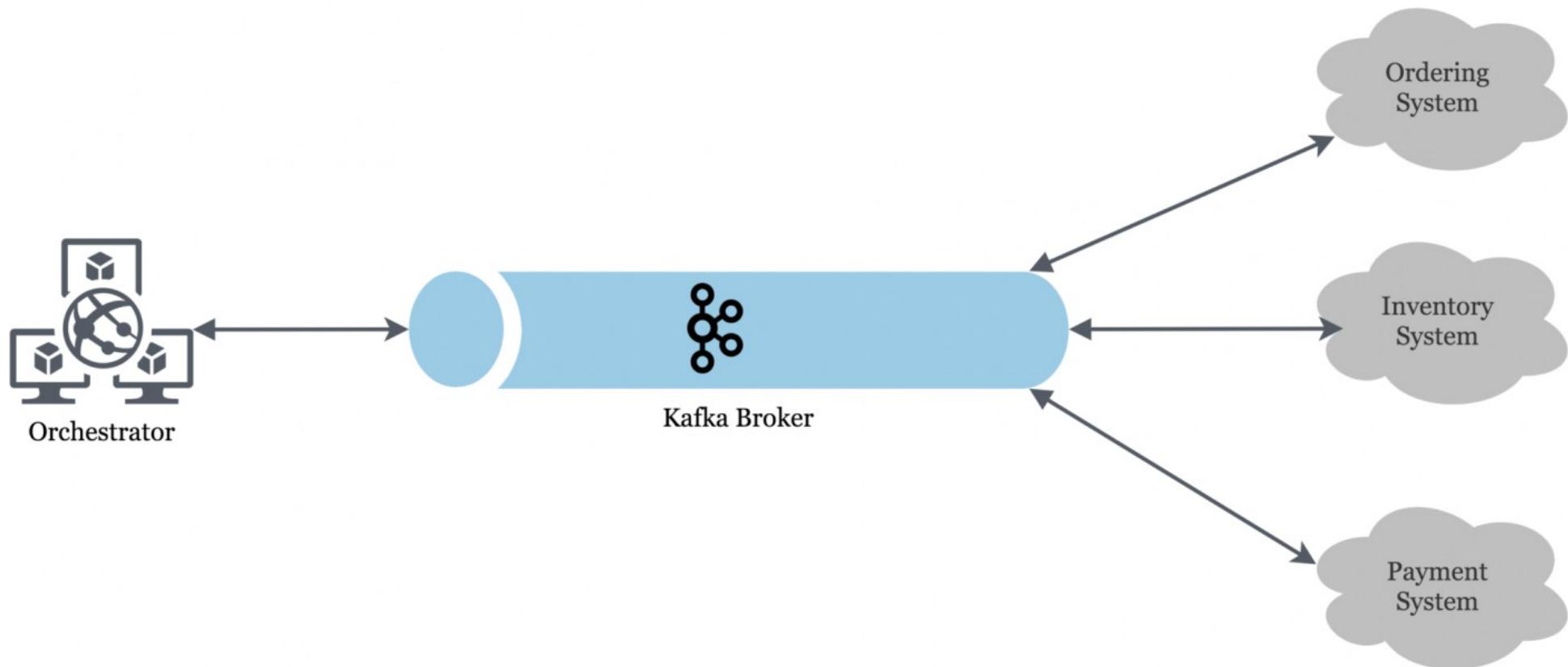
# Transaction management



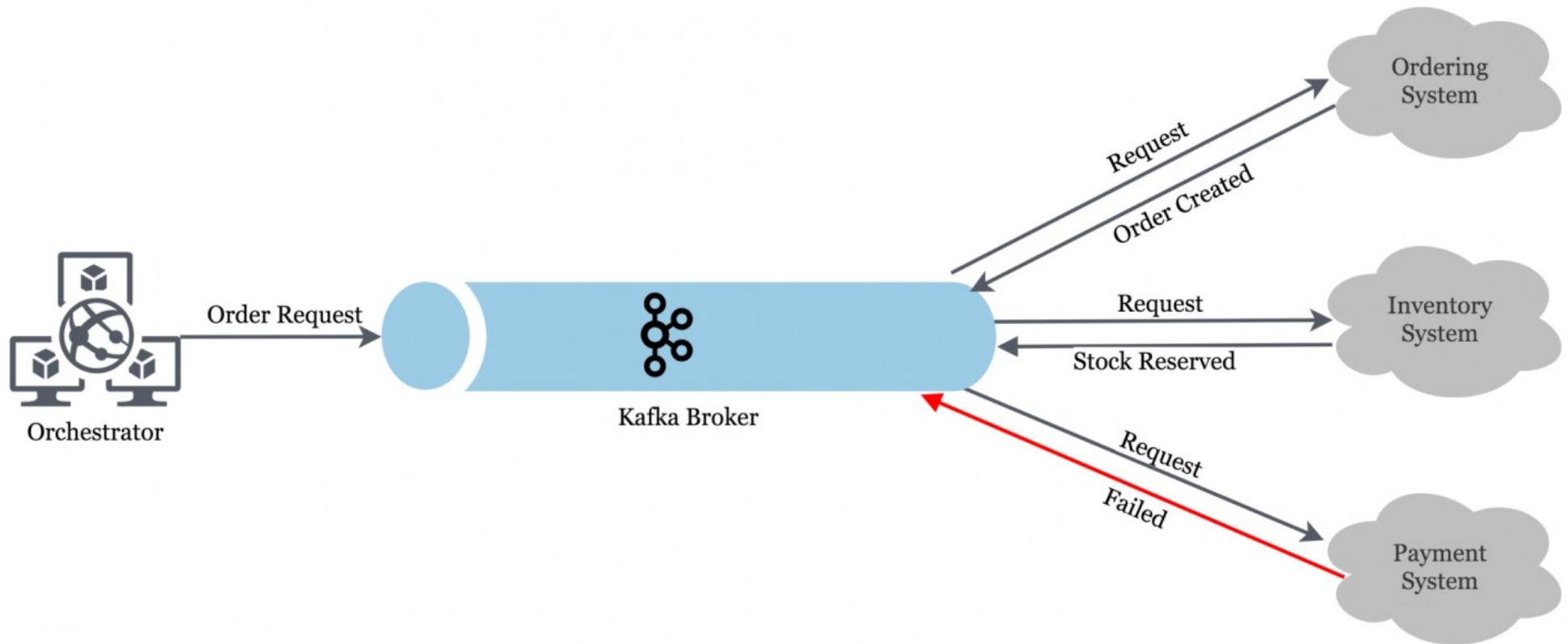
# Saga Pattern

- Global distributed transactions as series of local ACID transactions
- Compensations as rollback mechanism
- Global state depends of local transaction executions
- Orchestration / Choreography

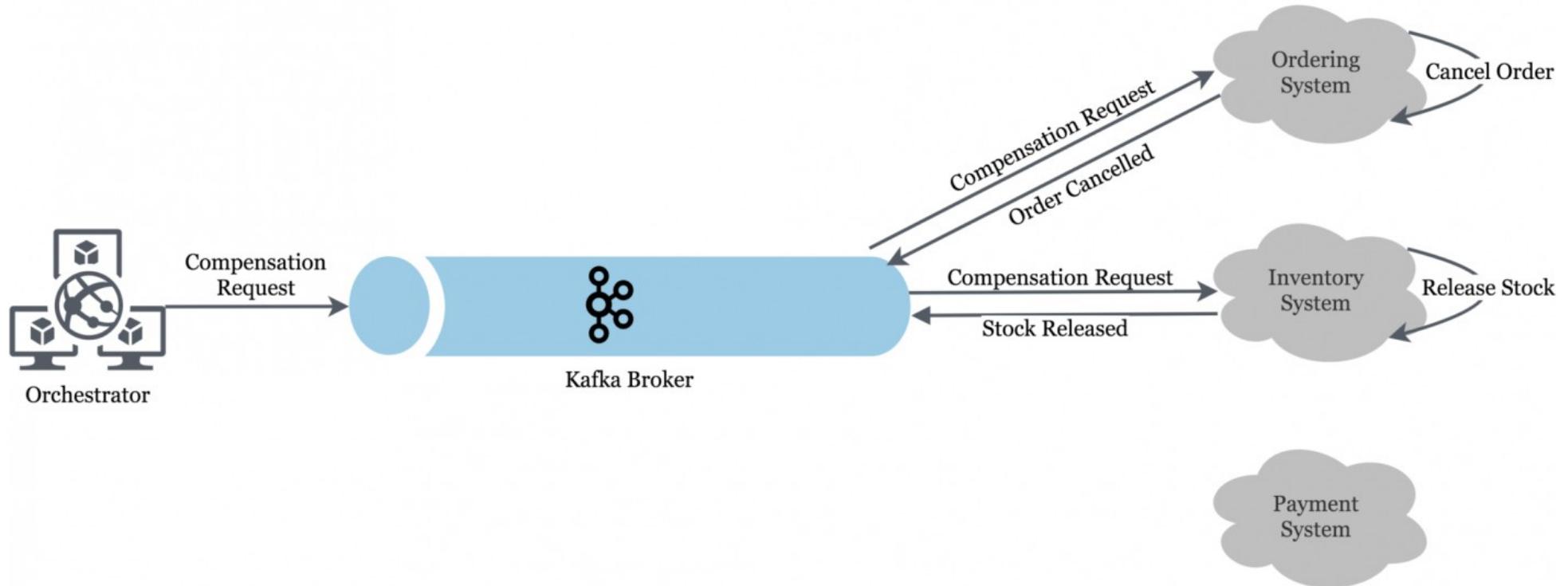
# Saga Pattern



# Saga Pattern

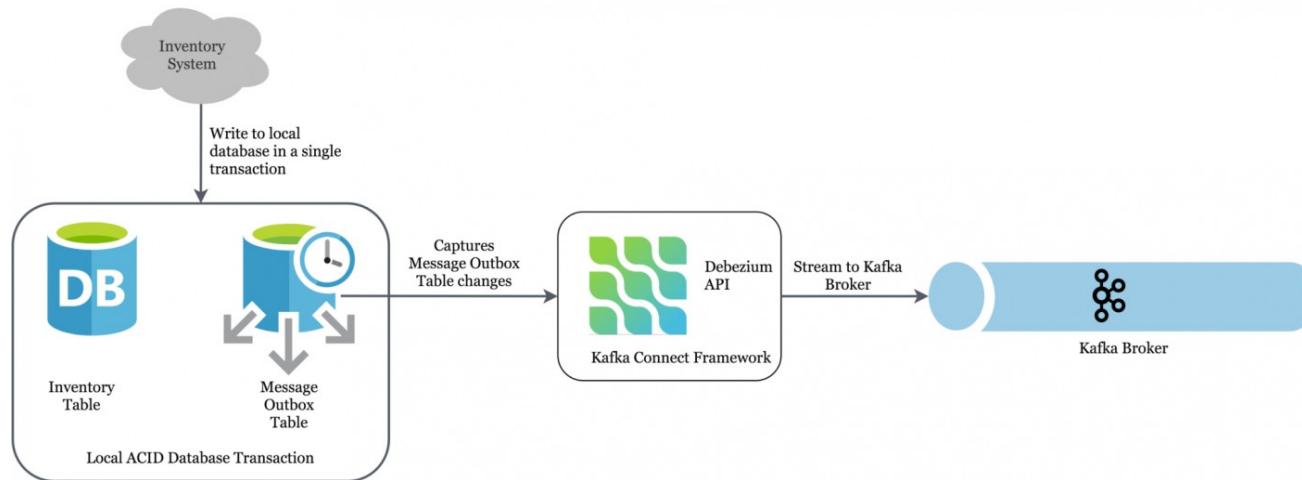


# Saga Pattern



# Transactional outbox pattern

- Database operations will atomically inserted to outbox table in the same transaction
- Outbox table will then produce events to publish transactions



# Agenda - Day 2

- 8 fallacies of distributed computing
- Cloud patterns
- Continuous integration and delivery
- Event driven architecture and messaging
- Observability – Metrics and Tracing
- Writing your own Quarkus extension

# Deployment automation

- DevOps culture
- Applications can be rolled out independently
- Continuous Integration and Delivery is crucial
- Build and Deploy often
- Automate as much as possible

# DevOps



# DevOps

«DevOps is the combination of cultural philosophies, practices, and tools that increases an organization's ability to deliver applications and services at high velocity: evolving and improving products at a faster pace than organizations using traditional software development and infrastructure management processes.»

<https://aws.amazon.com/devops/what-is-devops/>

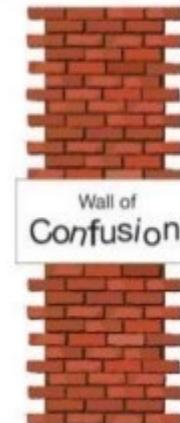
# DevOps

«DevOps is the combination of **cultural philosophies**, **practices**, and **tools** that increases an organization's ability to deliver applications and services at high velocity: evolving and improving products at a **faster pace** than organizations using traditional software development and infrastructure management processes.»

<https://aws.amazon.com/devops/what-is-devops/>

# DevOps

## Benefits of DevOps Overcoming DevOps Silos



# DevOps - Before

- Classical split Devs and Ops
- Dev develops application
- Application thrown over fence
- Ops should operate unknown application
- Big knowledge gap



# DevOps culture

- Tear down wall of confusion
- Enable Devs to be Ops
- Integrate Operations to daily development
- Mindset should develop

YOU FOOL!  
It's not about the tools, it's about the **CULTURE!**

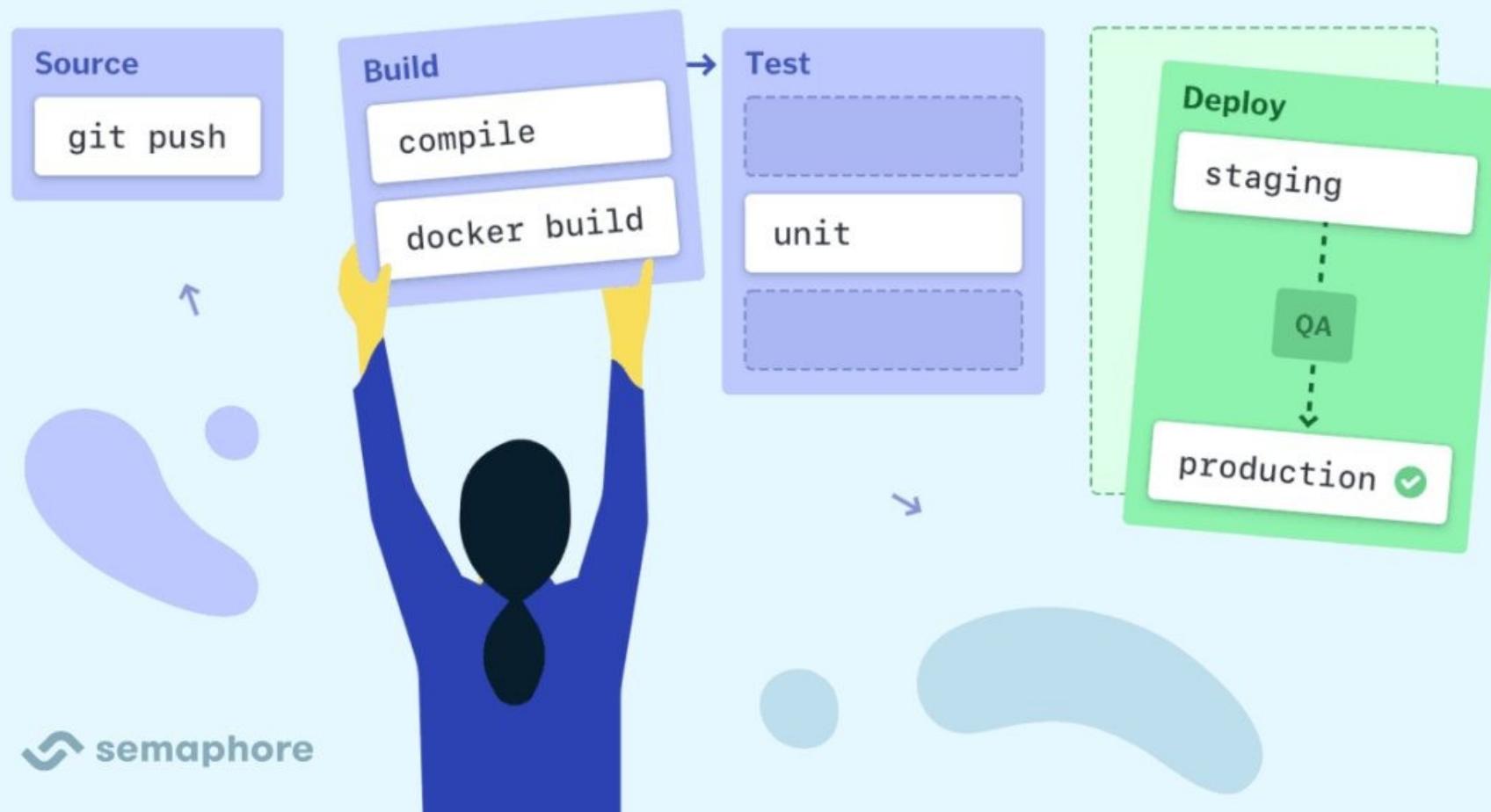


<http://markrainslife.files.wordpress.com/2011/07/wepmentcartoon.jpg>

# Infrastructure as Code (IaC)

- Infrastructure objects defined in code
- Development process involves infrastructure
- Infrastructure gets detachable
- Fast creation of new environments

# Deployment automation



# Deployment automation

- Automated pipelines for build, test, release
- Deploy fast and often
- «Handle every commit on master as deployable»
- Feedback-Loop gets faster

# Deployment automation

- API changes need to be categorized
- Non-breaking changes deployed fast
- Breaking changes need to deploy all dependent services
- Communication is key

# Agenda - Day 2

- 8 fallacies of distributed computing
- Cloud patterns
- Continuous integration and delivery
- Event driven architecture and messaging
- Observability – Metrics and Tracing
- Writing your own Quarkus extension

# Reactive Messaging

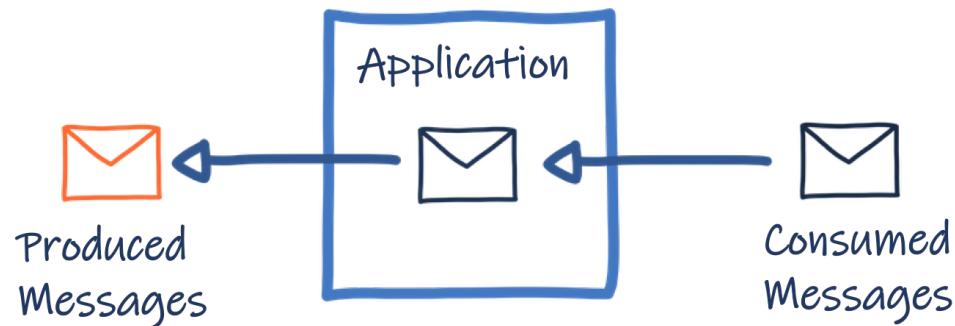
«Framework for building event-driven, data streaming and event sourcing applications using CDI», smallrye.io



# Reactive Messaging Concepts

## Message, Payload, Metadata

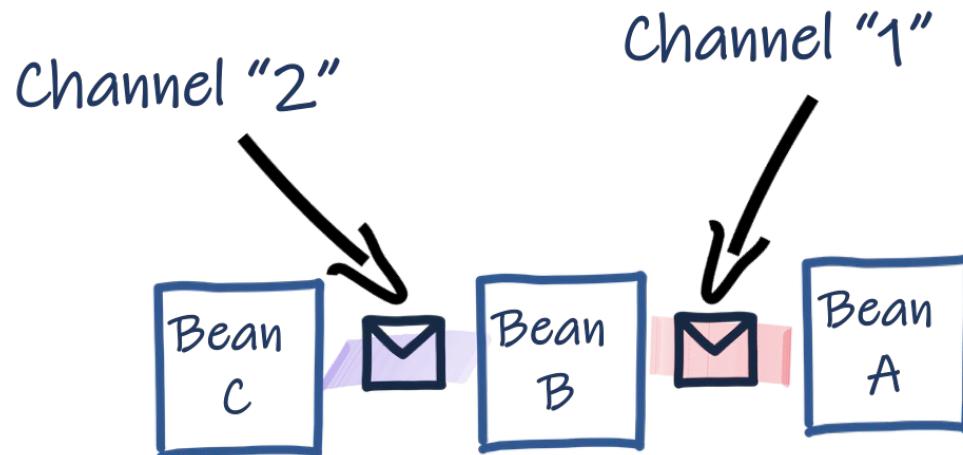
- Application receives, processes and send messages
- Message is an envelop around payload
- Messages can contain metadata (e.g tracing context)



# Reactive Messaging Concepts

## Channels and Streams

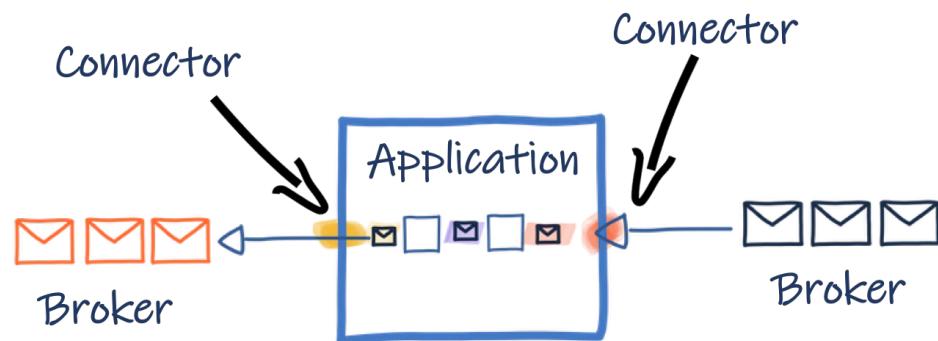
- Messages transit on channels
- Virtual destination (identified by name)



# Reactive Messaging Concepts

## Connectors

- Connect to broker
- Poll or write messages
- Map messages to channels
- Dedicated to a technology (e.g. Kafka)



# Reactive Messaging

## Connector Support

- Kafka Connector
- AMQP Connector
- MQTT Connector
- HTTP Connector
- Camel Connector (SmallRye Camel Quarkus)
- In Memory (for testing)

# Reactive Messaging Example

Application Class

```
/**  
 * A bean consuming data from the "prices" Kafka topic and applying some conversion.  
 * The result is pushed to the "my-data-stream" stream.  
 */  
@ApplicationScoped  
public class PriceConverter {  
  
    private static final double CONVERSION_RATE = 0.88;  
  
    @Incoming("prices")  
    @Outgoing("my-data-stream")  
    public double process(int priceInUsd) {  
        return priceInUsd * CONVERSION_RATE;  
    }  
}
```

# Reactive Messaging Example

Application Configuration (Serializer/Deserializer properties omitted)

```
# Configure the SmallRye Kafka connector
kafka.bootstrap.servers=localhost:9092      { kafka address

# Configure the Kafka source (we read from it)
mp.messaging.incoming.prices.connector=smallrye-kafka
mp.messaging.incoming.prices.topic=prices-raw

# Configure the Kafka sink (we write to it)          connector type
mp.messaging.outgoing.my-data-stream.connector=smallrye-kafka
mp.messaging.outgoing.my-data-stream.topic=prices-converted

{ channel }                                     { kafka topic }
```

# Reactive Messaging

## Documentation

- [SmallRye Reactive Messaging](#)

## View supported [Method Signatures](#)

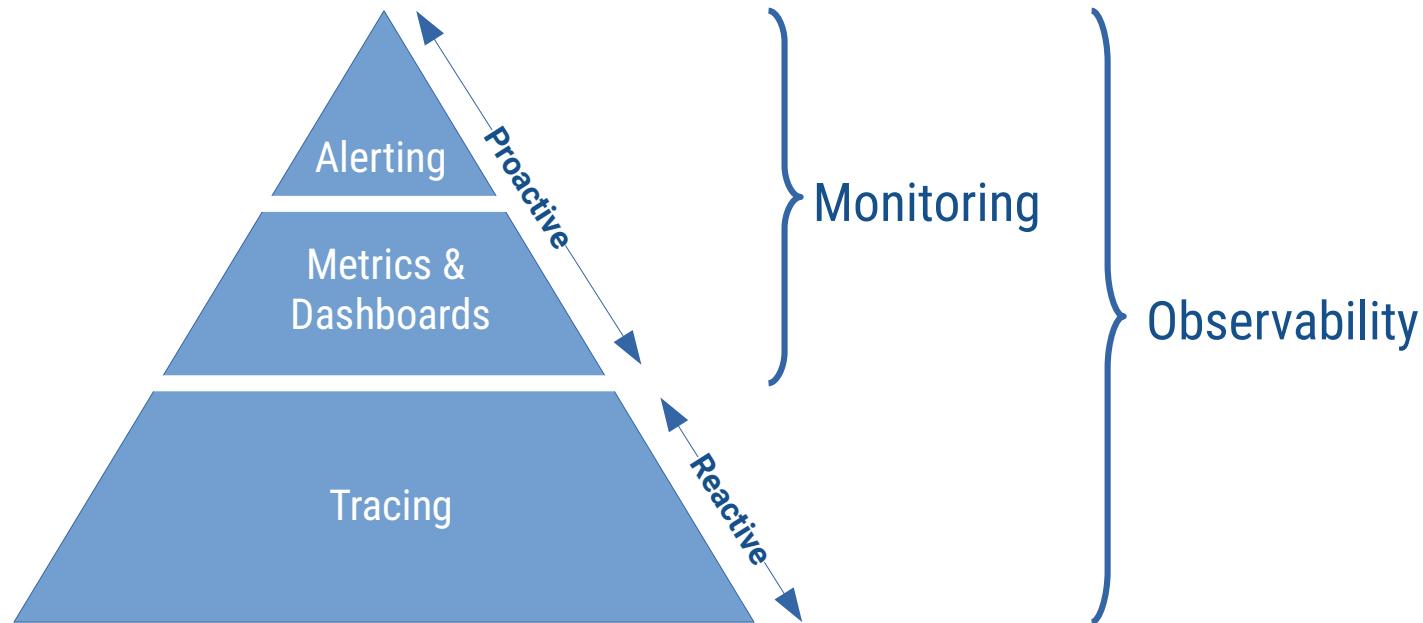
- They affect the acknowledgement strategies

# Agenda - Day 2

- 8 fallacies of distributed computing
- Cloud patterns
- Continuous integration and delivery
- Event driven architecture and messaging
- **Observability – Metrics and Tracing**
- Writing your own Quarkus extension

# Observability extends Monitoring

«Monitoring tells you whether a system is working, observability lets you ask why it is not working.», Baron Schwartz 2017



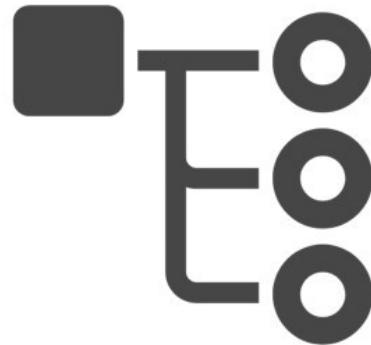
# Why do we need it?

- More complexity
- More diverse problems
- Runtime issues
- Request path may vary
- 1:n communication
- Monitoring: targets known problems
- Tracing: Instrumentation for problems we dont know yet

# Three pillars of observability



Metrics



Traces



Logs

# Metrics

```
number of orders created:23
number of failed invocations of /order:3
response time of /order:823ms
```

- Capture system state at given time
- Visualize and analyze metrics with dashboards
- History might be useful
- Send alerts based on metrics

# Logs

```
2020-07-12 11:45:34 Transaction id 12398 failed on update  
2020-09-23 19:23:11 POST /order - status:201 - response_ms:21
```

- Collect logs centrally
- Analyze logs in case of problem
- What happens at this time
- Find application misbehaviour and exceptions

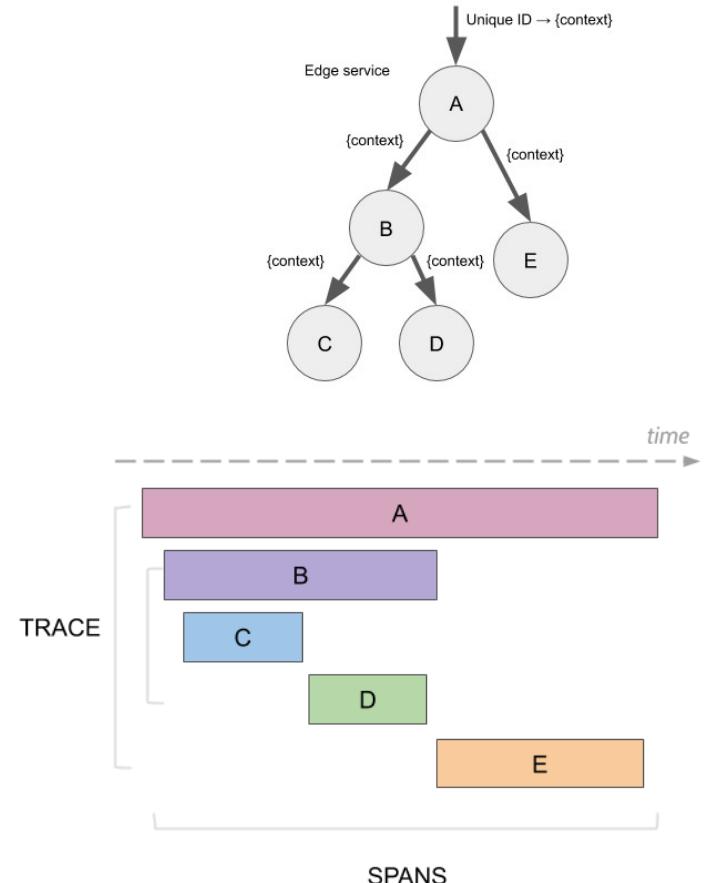
# Tracing - Terminology

**Trace:** Execution path through the system

- Acyclic graph of spans

**Span:** Logical unit of work

- Name, Start-Time and Duration
- Could be nested, ordered



# Tracing

```
request with id 98183 took 62ms
  - service a took 22ms
  - service b took 40ms
```

- Sample real requests
- Find erroneous components
- Find bottlenecks

# Observability Example

# Observability – Example

## Logs

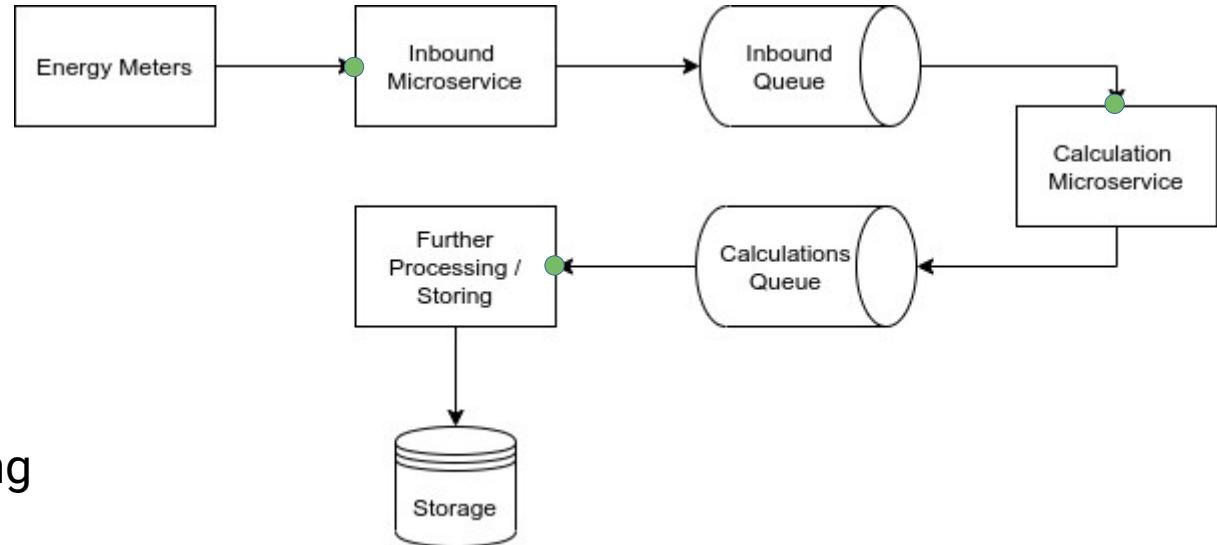
- Centralized logs

## Metrics

- JVM Metrics Microservices
- System metrics
- Network metrics
- Message Brokers
- Storage
- Duration of message processing

## Alerts

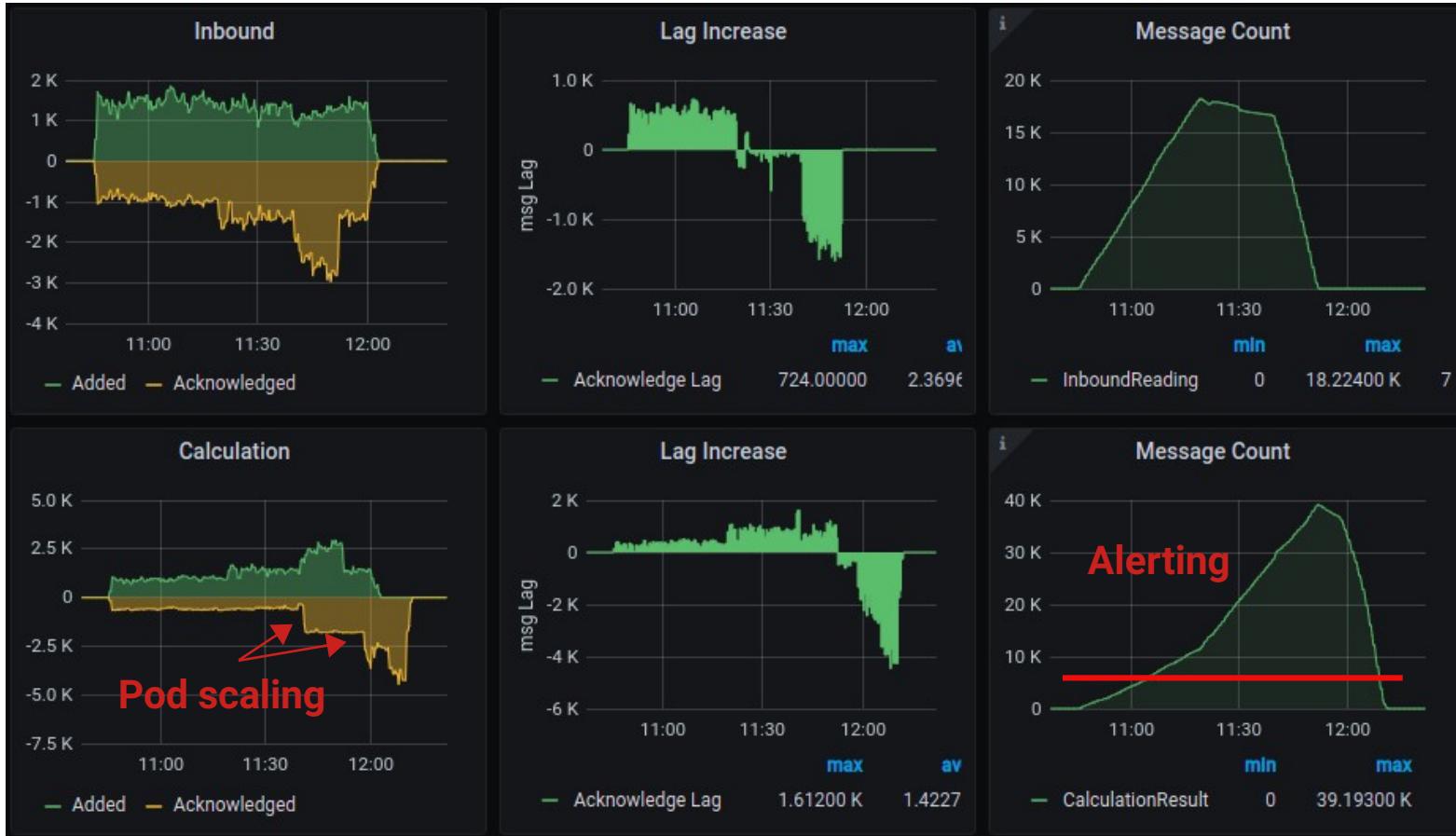
- Queue size
- ...



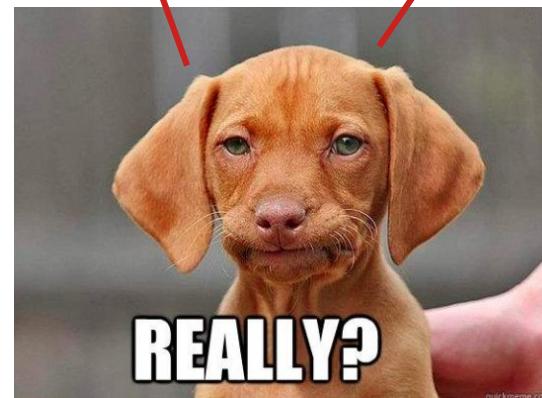
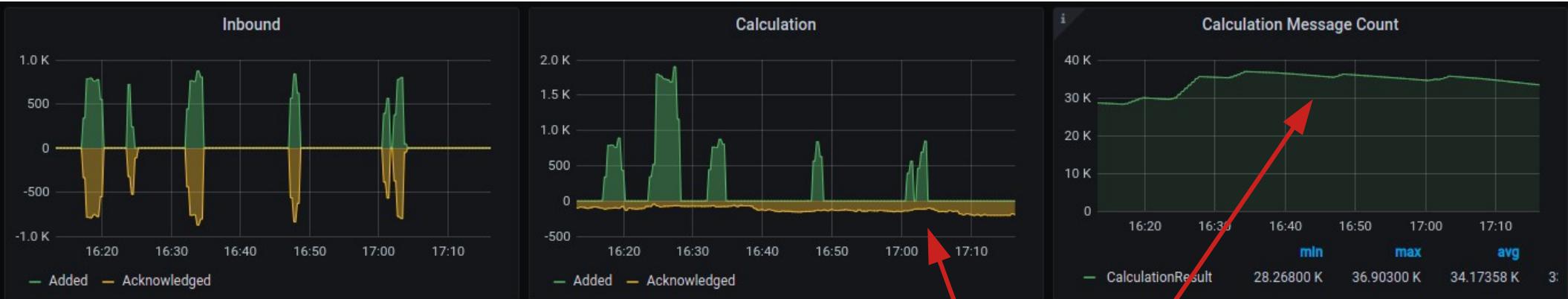
# Observability – Example



# Observability – Example



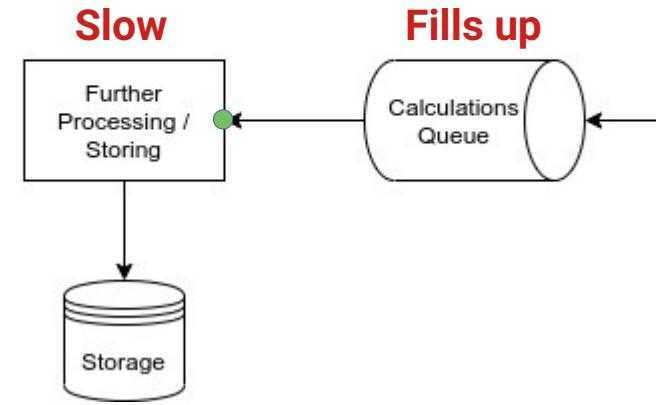
# Observability – Example



# Observability – Example

**Time to investigate ...**

1. Log: what happened at that time?
2. Metrics: processing is slow
3. Metrics: network, system does not show any problems



# Observability – Example

## Results from Tracing

- Huge amount of repeated spans
- Three-fourths of time in foreach
- «complex» calls db & services
- Select n + 1 problem
- Increasing size of data[] increases request count

```
processCalculations(Message message) {  
    Data[] data = message.data;  
  
    // handle data  
    data.foreach({  
        // complex  
    });  
  
    store(data);  
}
```

# Observability – Example

## Optimization

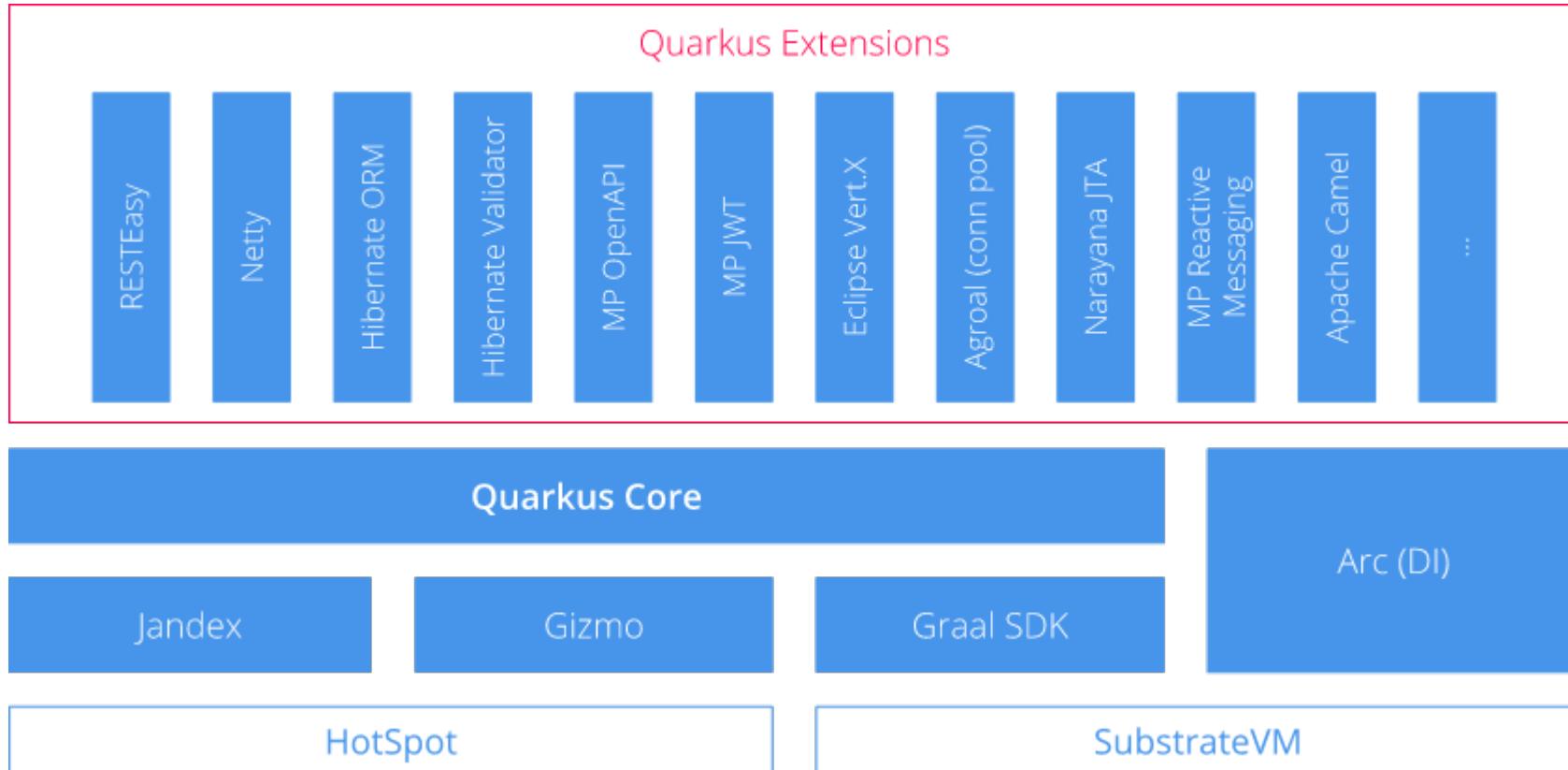
- Prefetch data once (as batch)
- Less interaction in foreach
- Code runs 10x faster
- Increasing `data[]` does not increase request count

```
processCalculations(Message message) {  
    Data[] data = message.data;  
  
    // prefetch required data once  
    fetchData();  
  
    // handle data  
    data.foreach({  
        // less complex  
    });  
  
    store(data);  
}
```

# Agenda - Day 2

- 8 fallacies of distributed computing
- Cloud patterns
- Continuous integration and delivery
- Event driven architecture and messaging
- Observability – Metrics and Tracing
- Writing your own Quarkus extension

# Quarkus structure



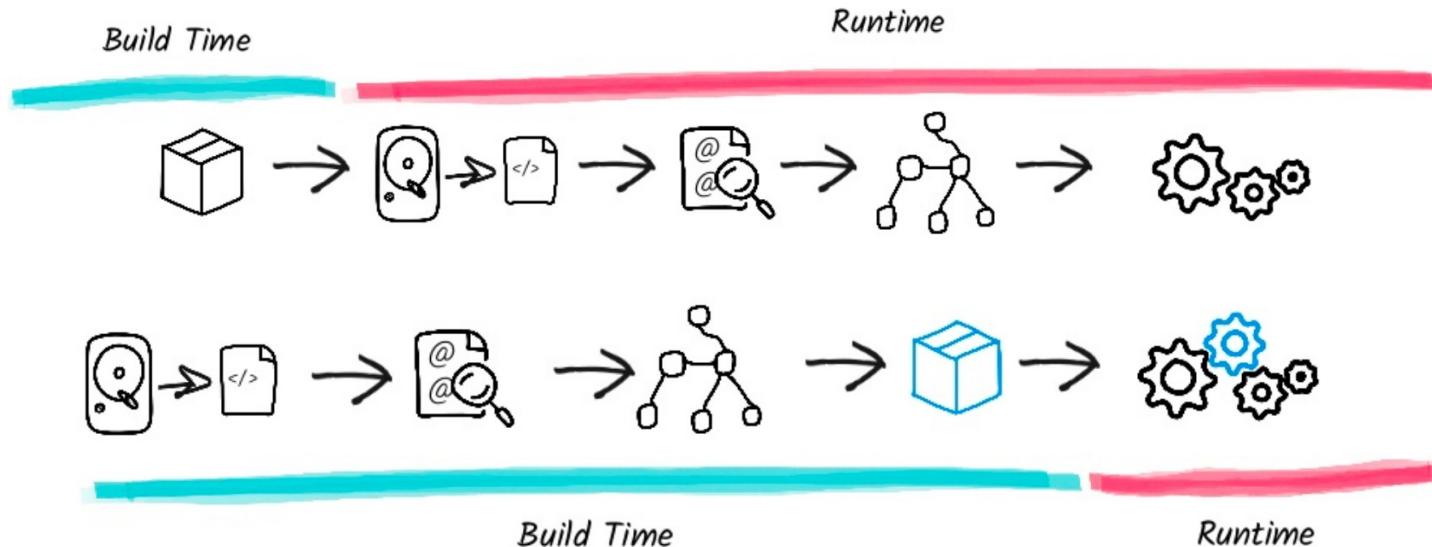
# Extensions

- Extend Quarkus framework
  - With custom functionality
  - Adopting libraries or frameworks to the Quarkus world
- Maven multi-module project
  - Deployment
  - Runtime

# Extensions

Adopting libraries or frameworks to the Quarkus world

- Third party libraries might or might not work out of the box



# Deployment Module

- How to deploy the extension code
- Processing done with @BuildSteps which run at build time
- Has dependency to extension runtime module
- Dev UI integration

# Deployment Module - @BuildStep

- Produce and consume BuildItems
- Are ordered by dependencies
- A BuildStep is run when all dependencies are available
- Can record invocations by using recorders from runtime module

# Deployment Module - @BuildStep

Example: Find classes ending with MessageConverter and register for CDI

- Jandex Access → CombinedIndexBuildItem
- Register additional beans for CDI → AdditionalBeanBuildItem

```
@BuildStep
void messageConvertersAsBean(CombinedIndexBuildItem index,
                             BuildProducer<AdditionalBeanBuildItem> additionalBeans) {

    List<String> converters = index.getIndex().getKnownClasses().stream()
        .filter(ci -> !Modifier.isAbstract(ci.flags()))
        .map(ci -> ci.name().toString())
        .filter(c -> c.startsWith("my.dependency.package."))
        .filter(c -> c.endsWith("MessageConverter"))
        .collect(Collectors.toList());

    additionalBeans.produce(new AdditionalBeanBuildItem.Builder()
        .addBeanClasses(converters)
        .setUnremovable()
        .setDefaultScope(DotNames.APPLICATION_SCOPED)
        .build());
}
```

# Deployment Module - @BuildStep

Example: Add an undertow servlet

- Access to current LaunchMode of Quarkus → LaunchModeBuildItem
- Access build configuration → Custom defined MyBuildConfig
- Add Servlet → ServletBuildItem

```
@BuildStep
void createServlet(LaunchModeBuildItem mode,
                   MyExtConfig config,
                   BuildProducer<ServletBuildItem> servlets) {

    if(mode.getLaunchMode().isDevOrTest() && config.enabled) {
        servlets.produce(ServletBuildItem.builder(config.basePath, MyExtServlet.class.getName())
                           .addMapping(config.basePath)
                           .build());
    }
}
```

# Runtime Module

- Runtime features (extension code, configuration)
- Application depends on this extension module
- Contains recorders

```
@Recorder
class HelloRecorder {

    public void sayHello(String name) {
        System.out.println("Hello" + name);
    }
}
```

## @BuildStep (deployment module)

```
@Record(RUNTIME_INIT)
@BuildStep
public void helloBuildStep(HelloRecorder recorder) {
    recorder.sayHello("World");
}
```

# Exposing Configuration

- Extensions can expose configuration
  - Remember different configuration phases

```
@ConfigRoot(name = "my-ext", phase = ConfigPhase.BUILD_TIME)
public final class MyExtConfig {

    @ConfigItem(defaultValue = "true")
    public boolean enabled;

    @ConfigItem(defaultValue = "/myext")
    public String basePath;
}
```

application.properties

```
quarkus.my-ext.enabled=false
```

# Extension Resources

Writing your own extension

<https://quarkus.io/guides/writing-extensions>

Building my first extension

<https://quarkus.io/guides/building-my-first-extension>

How to write Quarkus extensions

<https://peter.palaga.org/presentations/210407-quarkus-insights-how-to-write-quarkus-extensions>

Quarkus core extensions

<https://github.com/quarkusio/quarkus/tree/main/extensions>

Quarkus community extensions (quarkiverse)

<https://github.com/quarkiverse>



# Goodbye



Raffael Hertle  
Cloud Architect  
[rafael@rokt.cloud](mailto:rafael@rokt.cloud)  
[@g1raffi](https://twitter.com/g1raffi)



Christof Lüthi  
Kafka Messaging Engineer  
[luethi@puzzle.ch](mailto:luethi@puzzle.ch)  
[@christofluethi](https://twitter.com/christofluethi)

