Quarkus Basics

Training

PUZZLE ITC
changing IT for the better

Nice to meet you

Raffael Hertle
Senior Software Engineer
hertle@puzzle.ch

Christof Lüthi
Kafka Messaging Engineer
luethi@puzzle.ch

One Team

Agenda - Day 1

- Microservices architecture
- Quarkus introduction
- MicroProfile specification
- RESTful microservices with Quarkus
- Building docker containers

Agenda - Day 2

- Cloud patterns
- Continuous integration and delivery (CICD) with Tekton
- Event driven architecture and messaging with Apache Kafka
- Observability – Metrics and Tracing
- Writing your own Quarkus extension

Vorstellungsrunde

Agenda - Day 1

- Microservices architecture
- Quarkus introduction
- MicroProfile specification
- RESTful microservices with Quarkus
- Building docker containers

Microservices

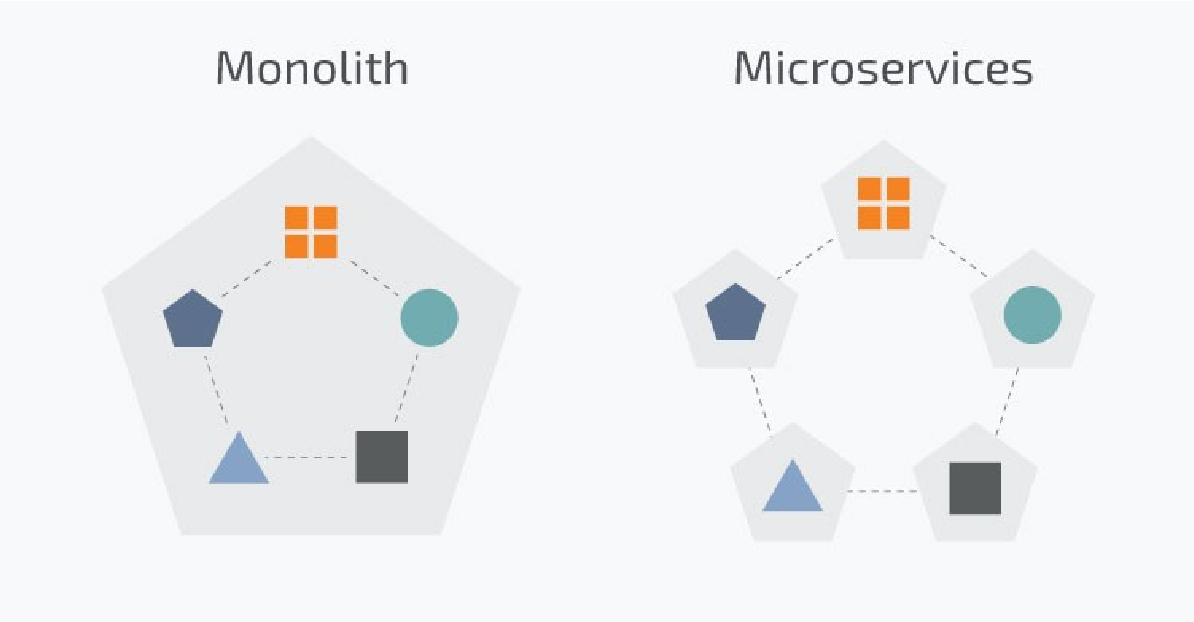
- What are microservices
- Why and when to use them
- Advantages / Disadvantages
- Approach to migrate

Monolithic Architecture

Monolithic Architecture

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

Microservices Architecture



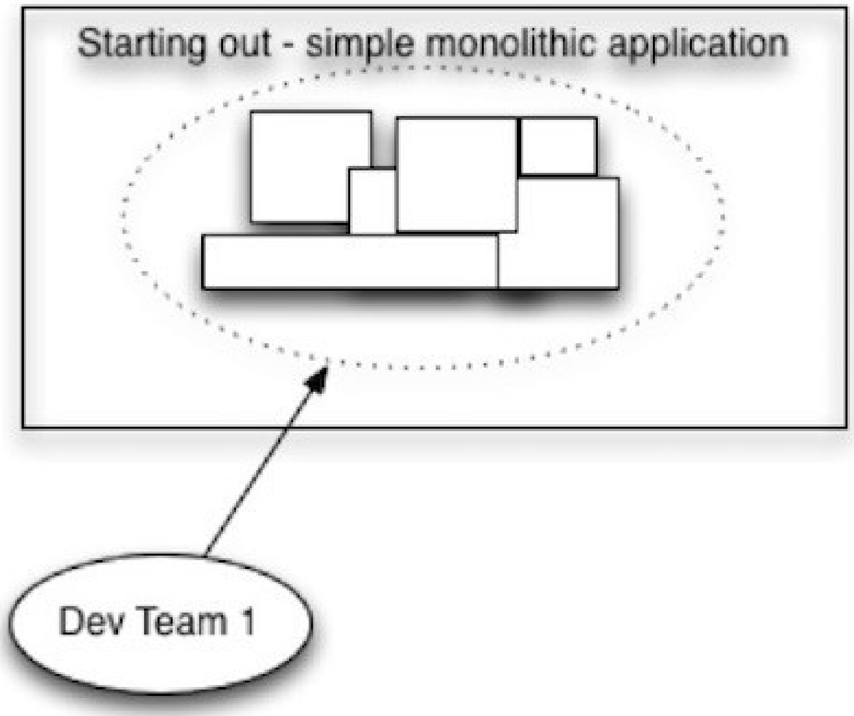
Microservices Architecture

- 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?

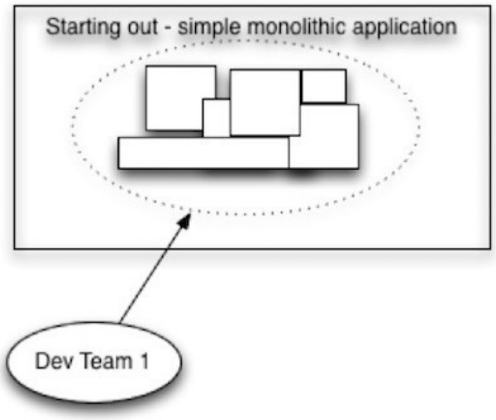
New Application



Advantages of monoliths

Simple architecture

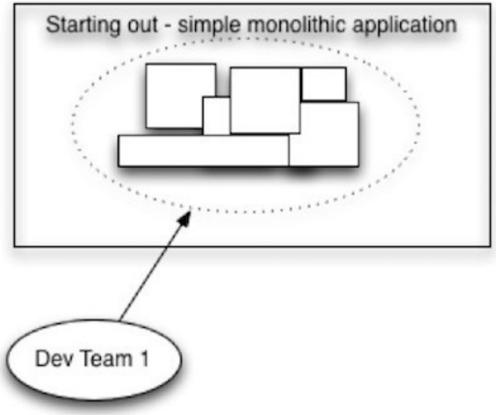
- Everything is local
- High productivity
- Limited attack vectors
- Easy testing
- Performance matches requirements



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

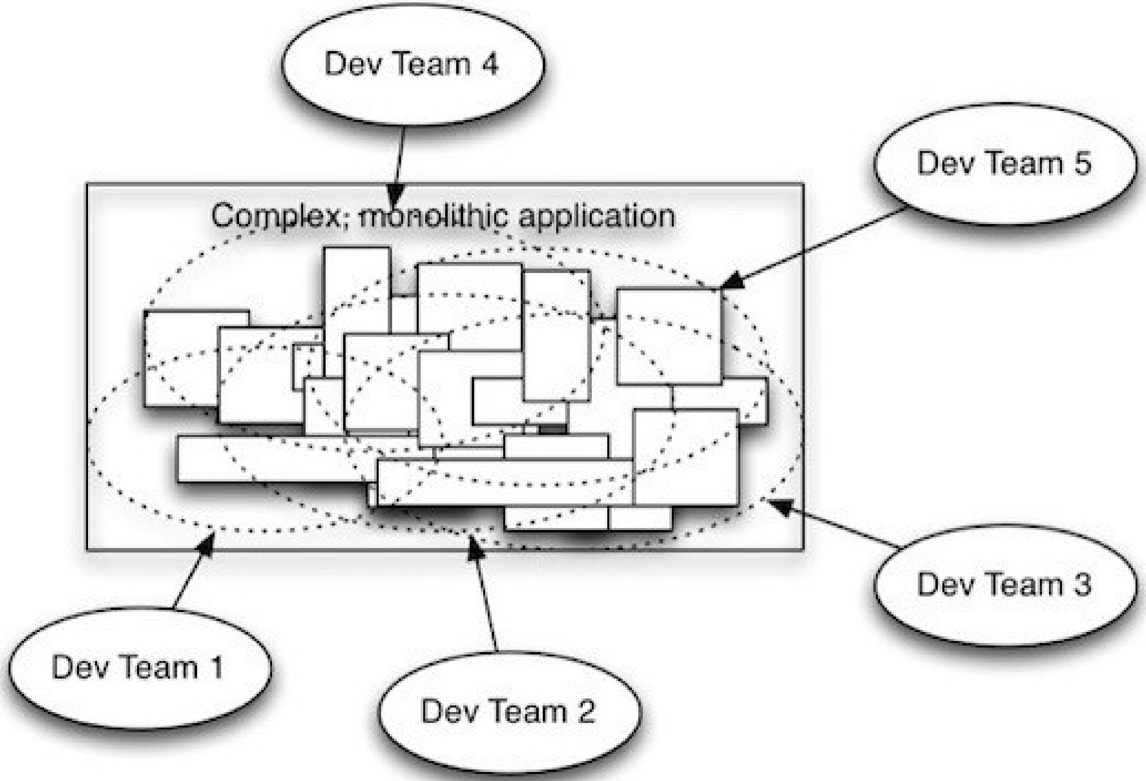
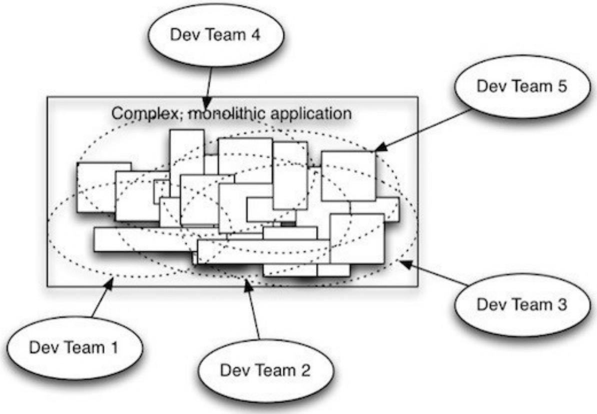


Image source: linkedin.com

Disadvantages of monoliths

Complex architecture

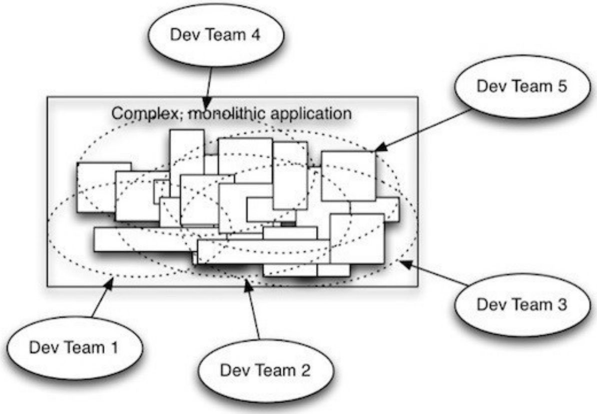
- Architectural changes are difficult
- Impact of code change are hard to estimate
- Keeping up code quality needs extra effort
- Newer technologies are hard to pickup



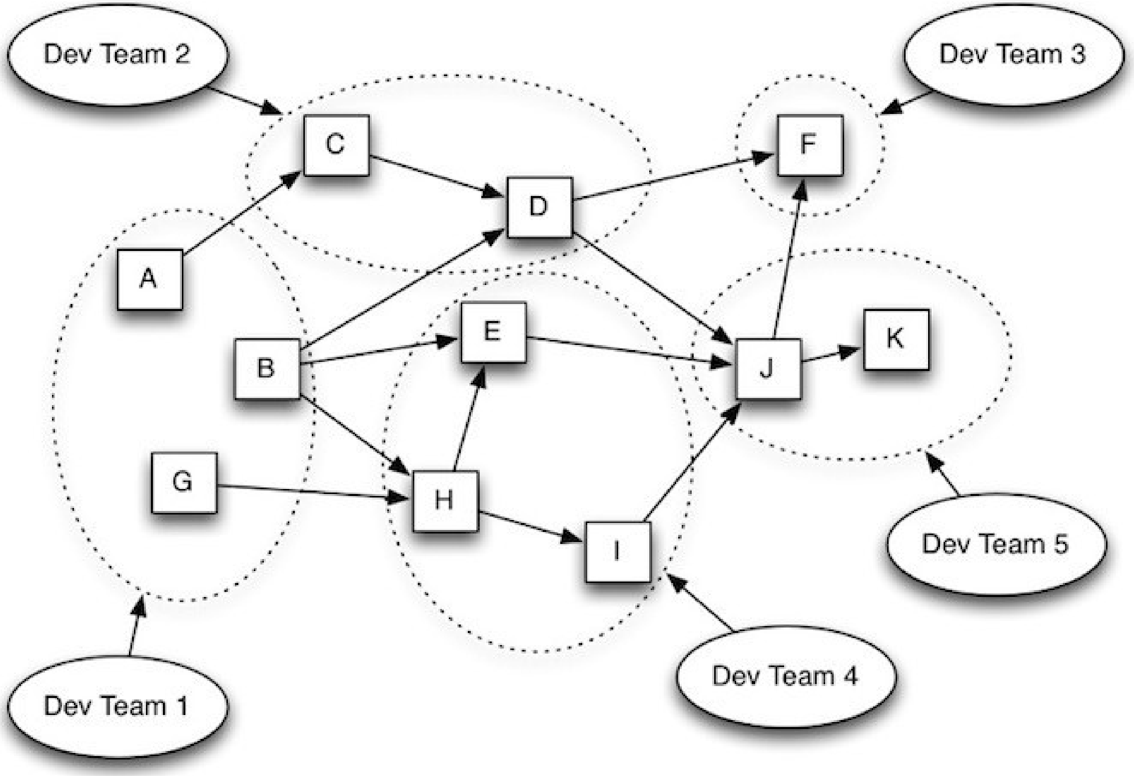
Disadvantages of monoliths (cont.)

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?

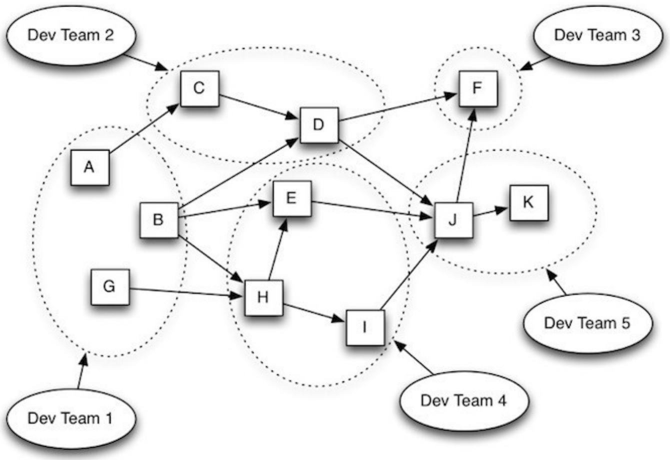


Dev Team 2
Dev Team 3
C
D
F
A
E
K
B
J
G
H
I
Dev Team 5
Dev Team 1
Dev Team 4

Advantages of microservices

Architecture

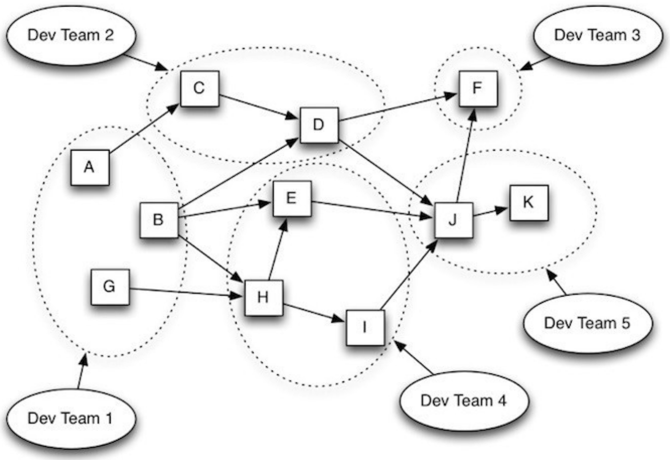
- Independent modules
- Defined boundaries (APIs, Events)
- Loosely coupled (if done right)
- Polyglot (what best fits the task)



Advantages of microservices

Team

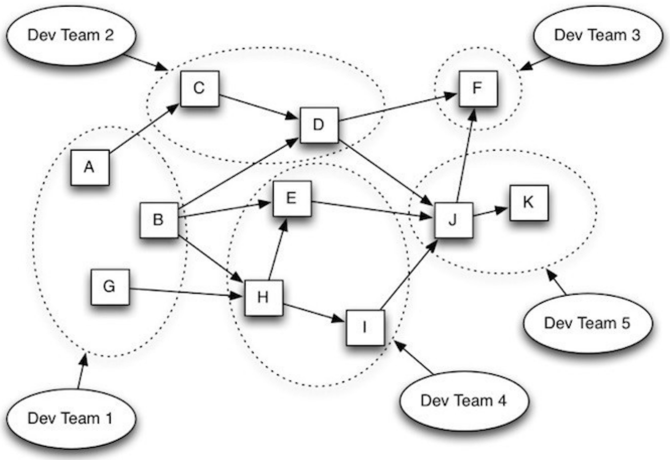
- Teams work independently
- Need to agree to defined boundaries
- Independent within their microservice
- Easier onboarding due limited scope



Advantages of microservices

Deployment and Runtime

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



But ...

Image source: turnoff.us

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

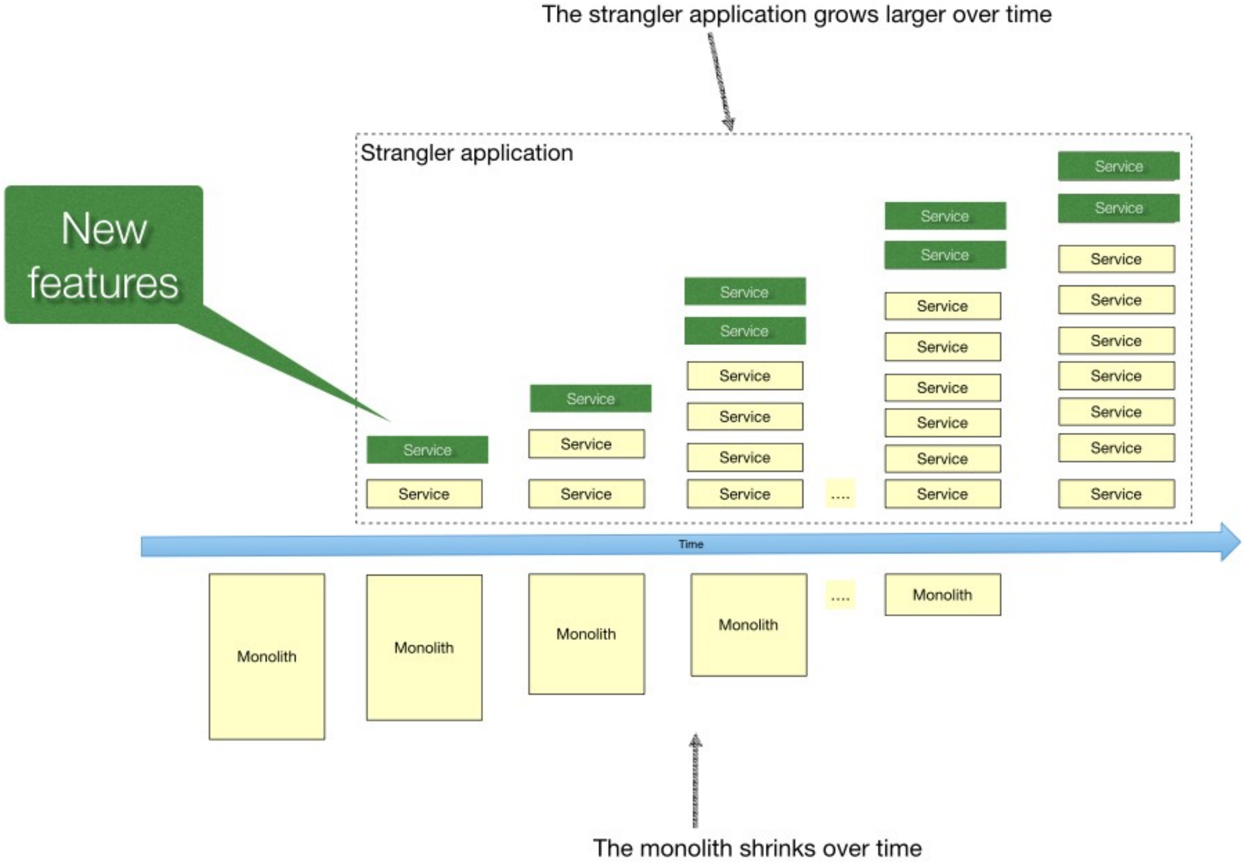
But they introduce technical complexity

Microservices Architecture

- How to migrate from monolithic appilcation?

Strangler pattern

Strangler pattern



Agenda - Day 1

- Microservices architecture
- Quarkus introduction
- MicroProfile specification
- RESTful microservices with Quarkus
- Building docker containers

Quarkus introduction

- What is Quarkus
- Some numbers
- Traditional vs Quarkus
- Configuration phases
- Quarkus structure
- Quarkus modes & downsides of GraalVM
- Extensions and standards
- Developer Joy

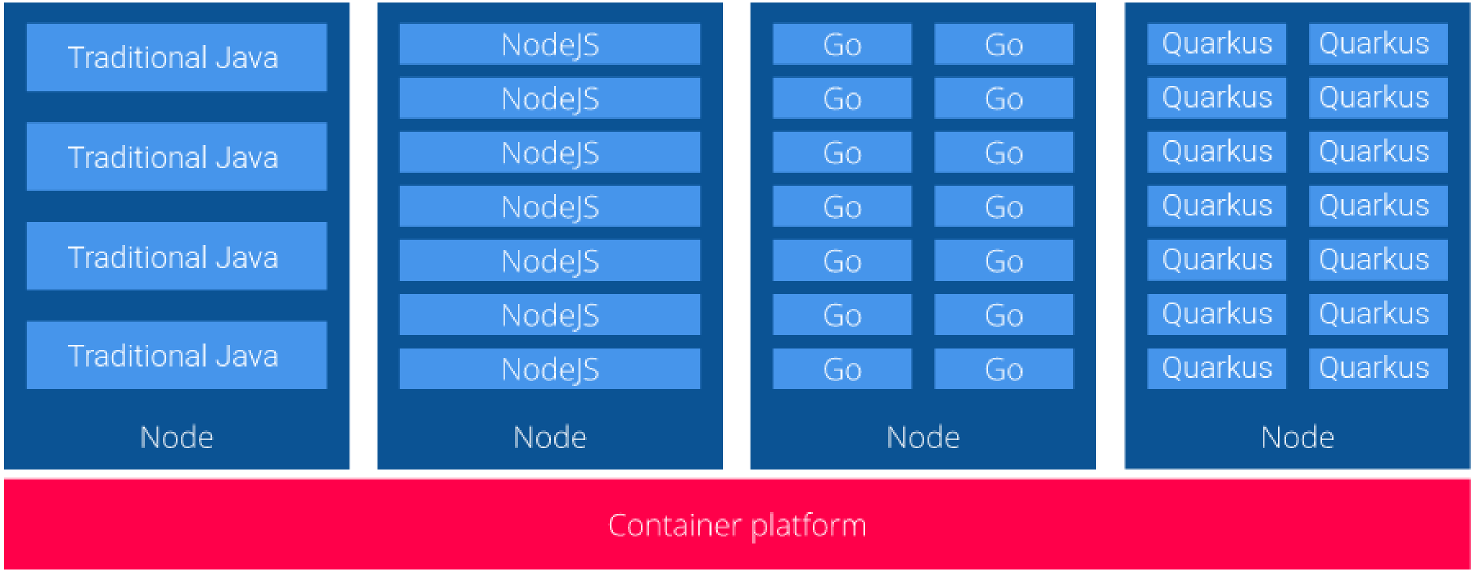
What is Quarkus?

«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

What is Quarkus?



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

Image source: Red Hat

Show me numbers!

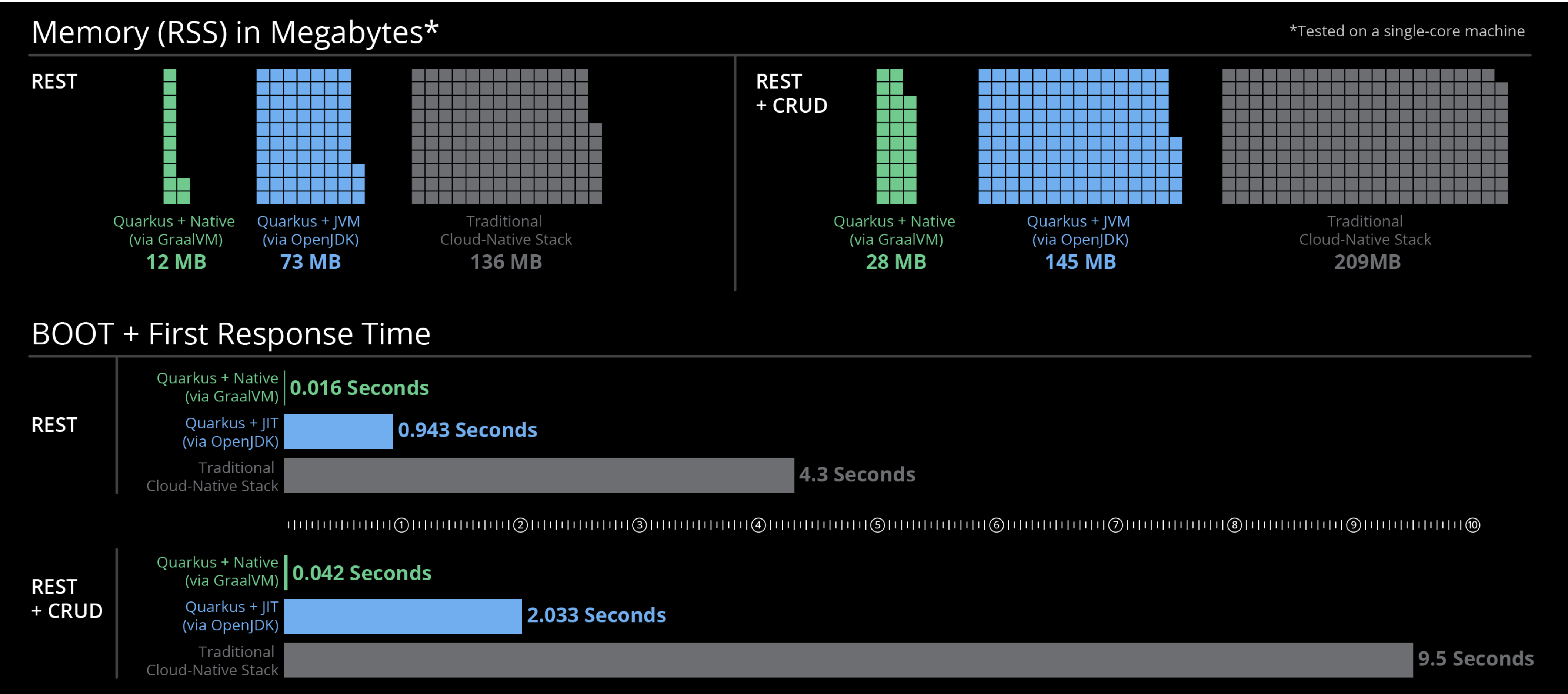
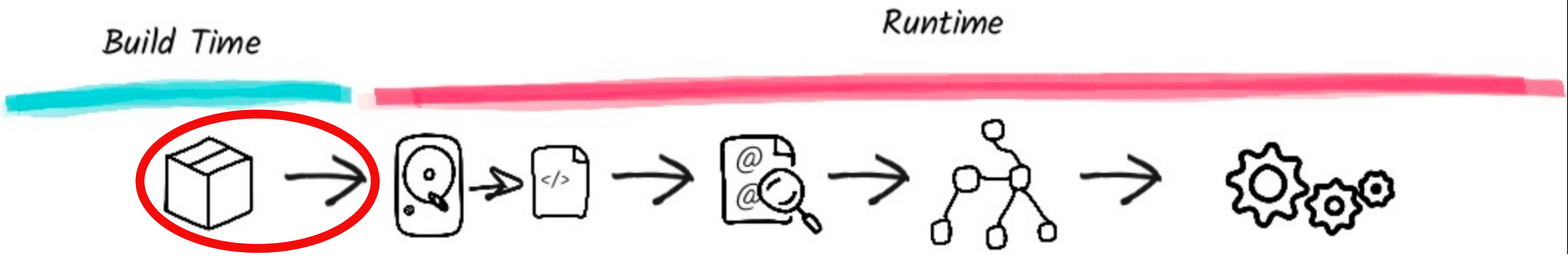


Image source: quarkus.io

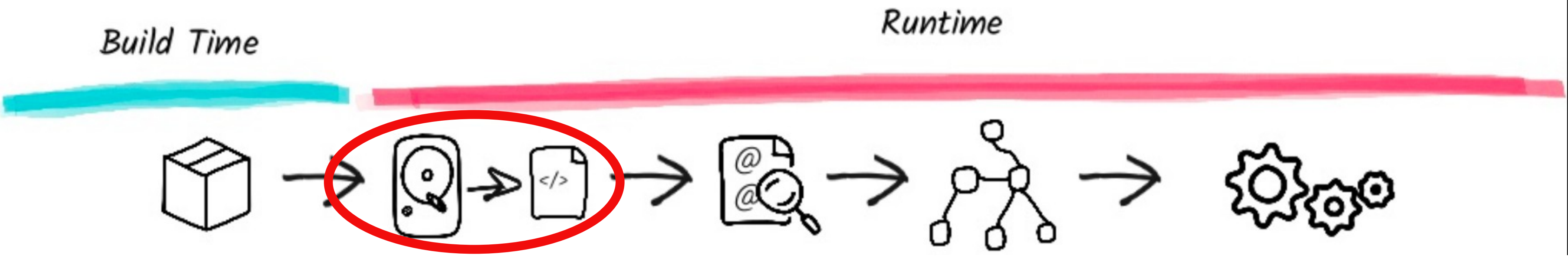
How a framework starts



Package Application
(maven, gradle, ...)

Image source: Red Hat

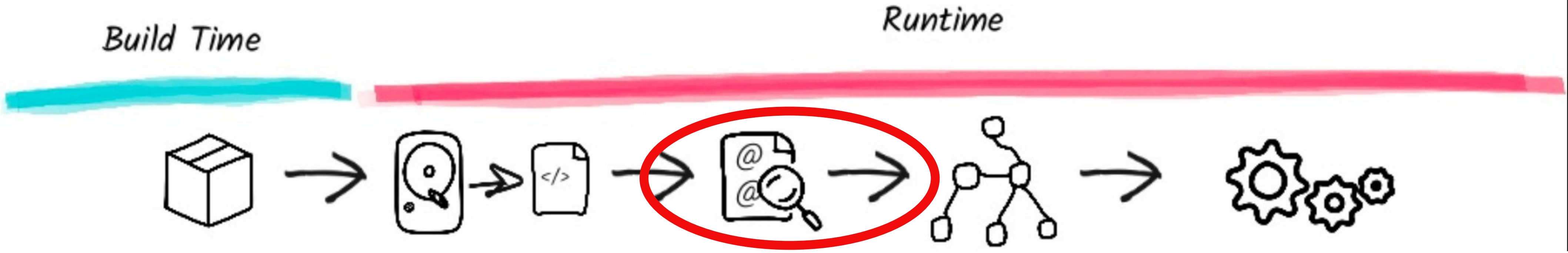
How a framework starts



Load config files
(parse it, ...)

Image source: Red Hat

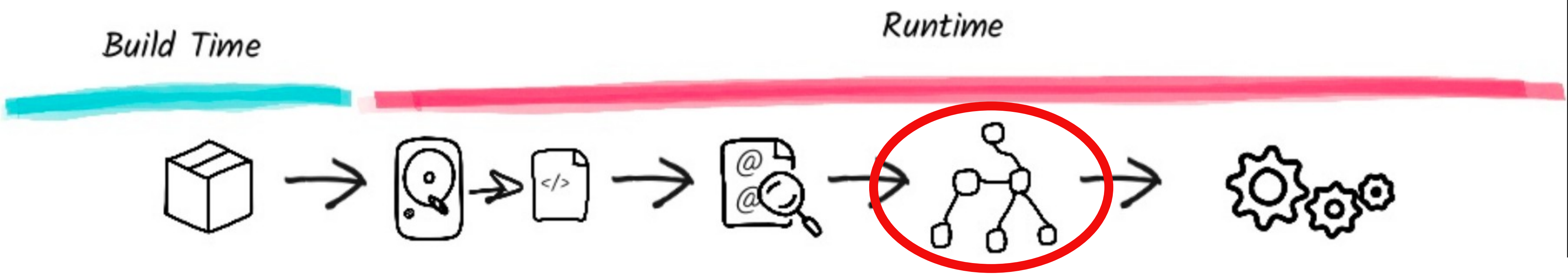
How a framework starts



Scan classpath for annotations
(load classes, enable/disable features, ...)

Image source: Red Hat

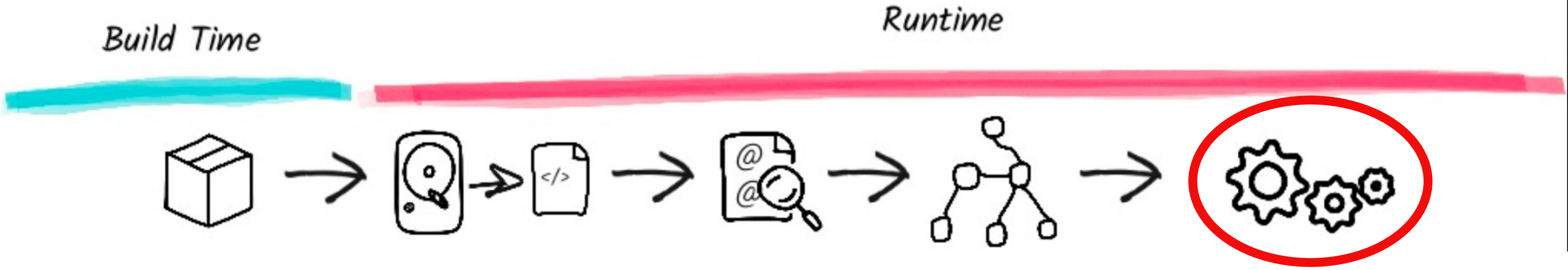
How a framework starts



Build its model of the world

Image source: Red Hat

How a framework starts



Start Application
(threads, pools, ...)

Image source: Red Hat

How Quarkus does it

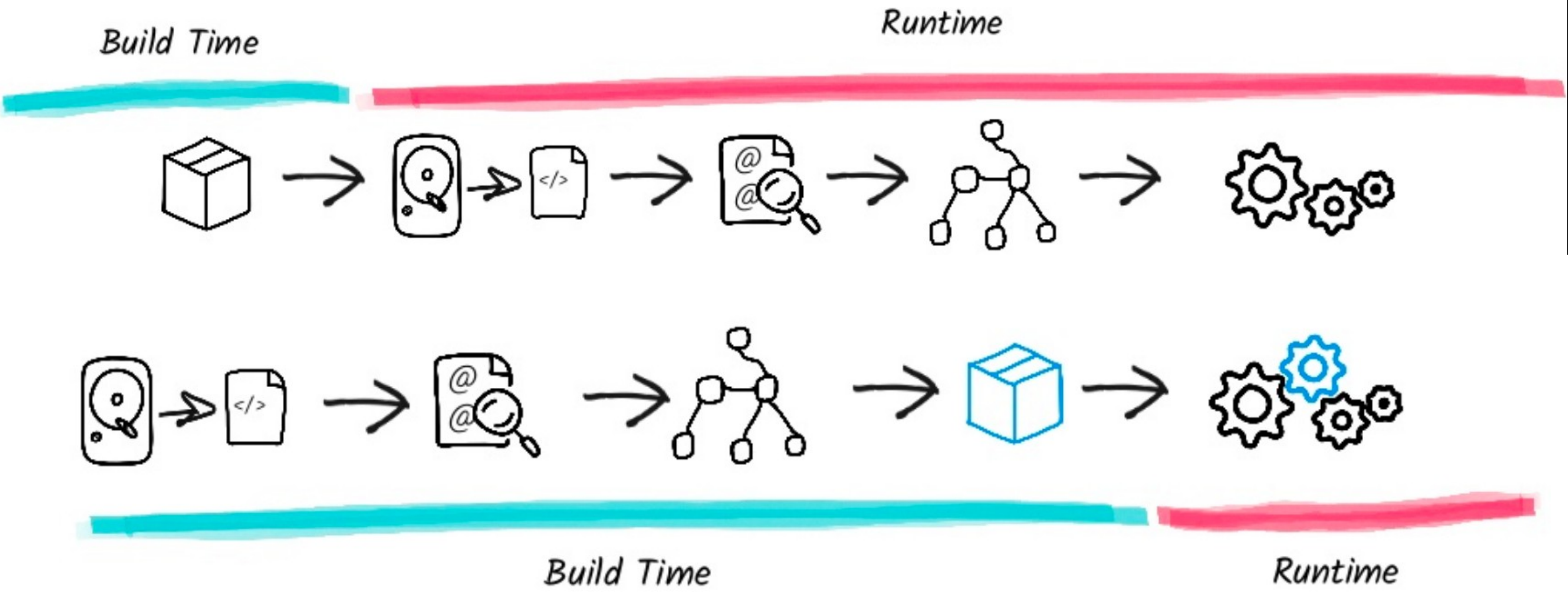


Image source: Red Hat

Traditional vs Quarkus

Traditional

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

Quarkus

- Built time approach (Dead code elimination)
- As much work as possible during build
- Built time augmentation
  - Extensions
- Output: recorded wiring bytecode

Configuration phases

- Build process of Quarkus needs build time configurations
- 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
🔒 `quarkus.datasource.db-kind` The kind of database we will connect to (e.g. h2, postgresql...).	string	
🔒 `quarkus.datasource.health.enabled` 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`
🔒 `quarkus.datasource.metrics.enabled` Whether or not datasource metrics are published in case a metrics extension is present. This is a Show more	boolean	`false`
`quarkus.datasource.username` The datasource username	string	
`quarkus.datasource.password` The datasource password	string	

Quarkus structure

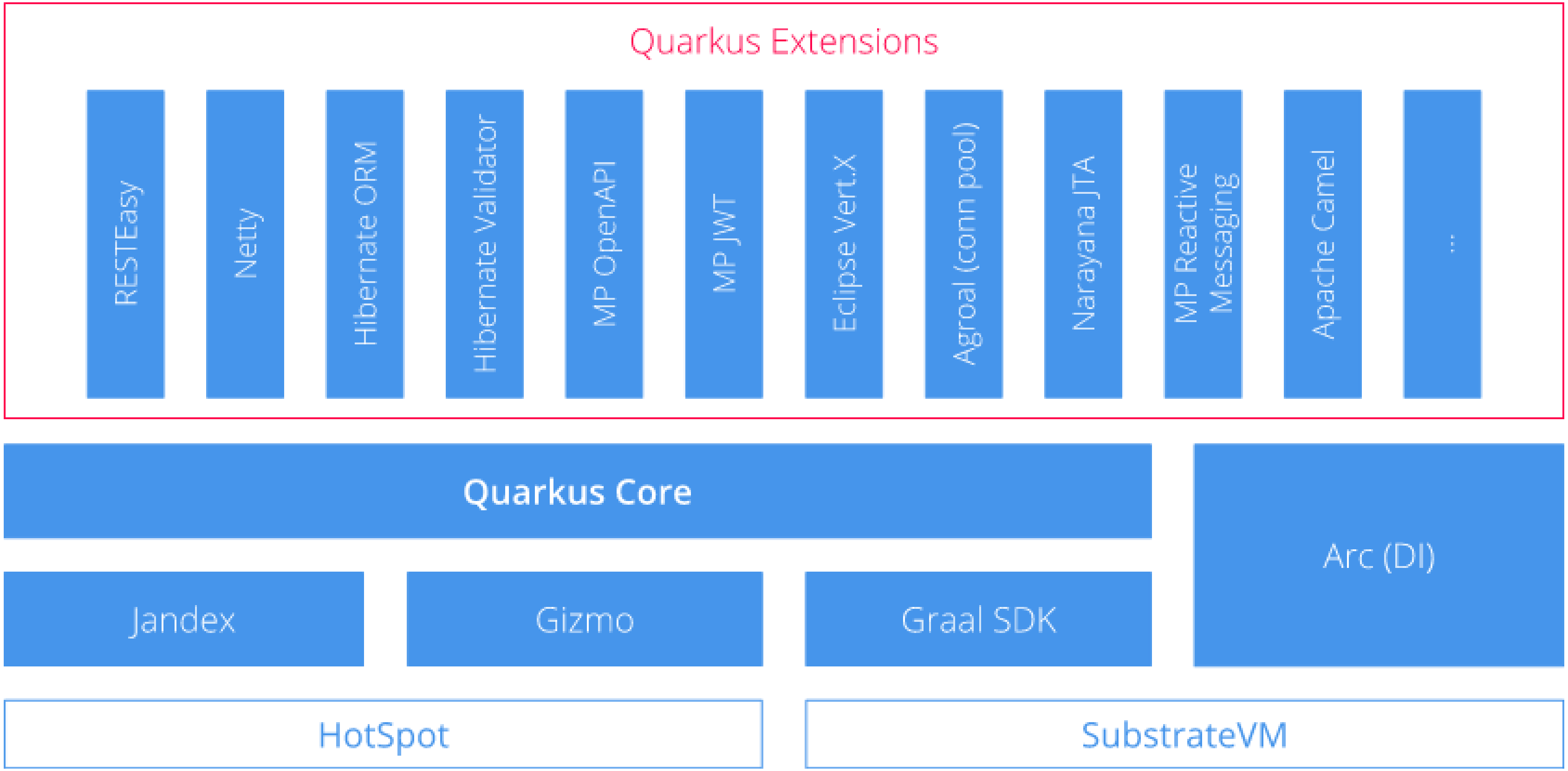
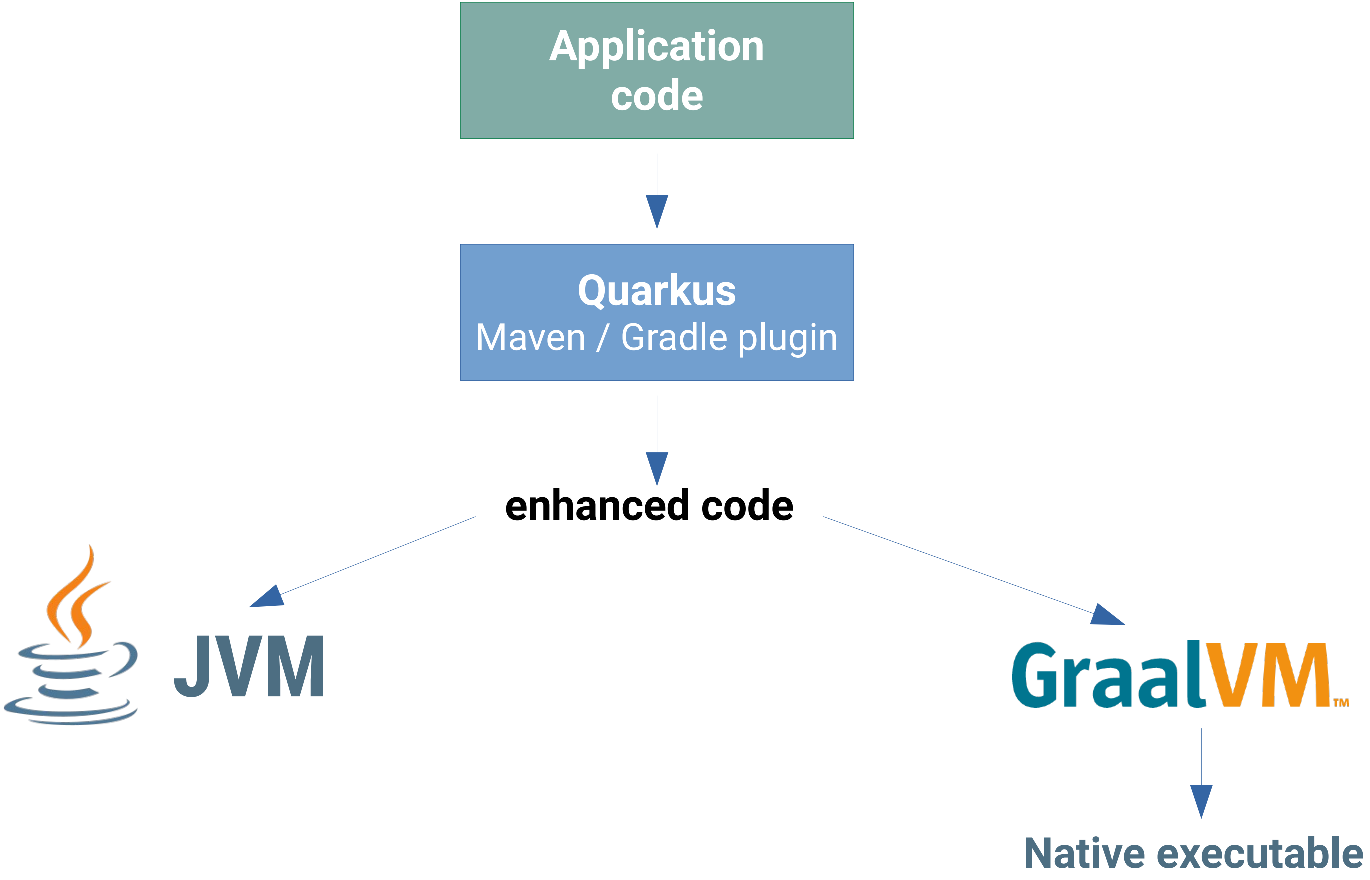


Image source: Red Hat

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, Traceing, …)
- No Java debugger
  - Native debugger GDB

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

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 (~150 extensions)

Independent sources

github.com/apache/camel-quarkus

github.com/datastax/cassandra-quarkus

github.com/debezium/debezium-quarkus-outbox

...

Quarkiverse: Incubator & Community extensions

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

code.quarkus.io provides a big list of extensions

Developer Joy

Easy to start & fast bootstrap

```
mvn io.quarkus:quarkus-maven-plugin:1.13.2.Final:create \
    -DprojectGroupId=ch.puzzle.quarkustechlab \
    -DprojectArtifactId=dev-demo \
    -DclassName="ch.puzzle.quarkustechlab.GreetingResource" \
    -Dpath="/hello"

cd dev-demo

mvn compile quarkus:dev

curl localhost:8080/hello
Hello RESTEasy
```

Developer Joy

Live Reload

```
mvn compile quarkus:dev

curl localhost:8080/hello
Hello RESTEasy

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


-> Hot replace total time: 0.256s
```

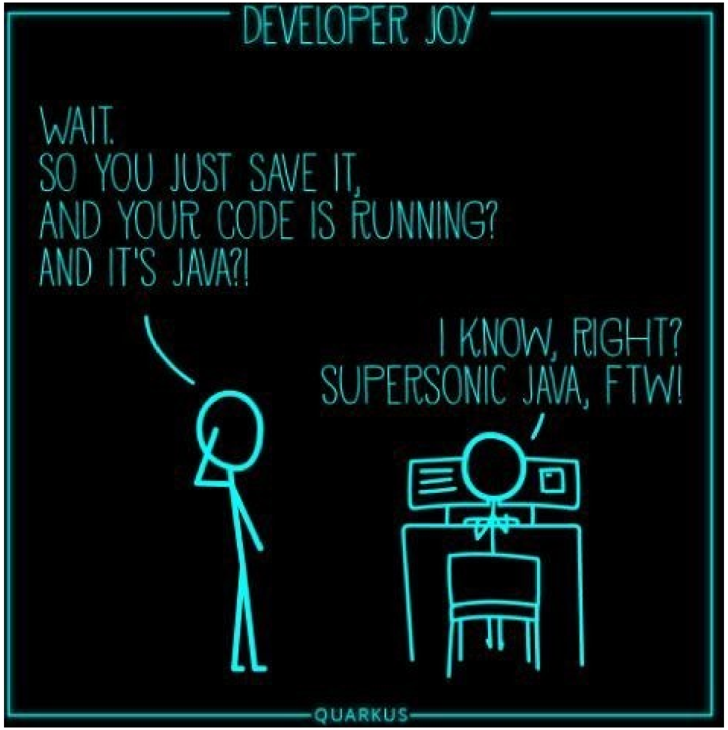
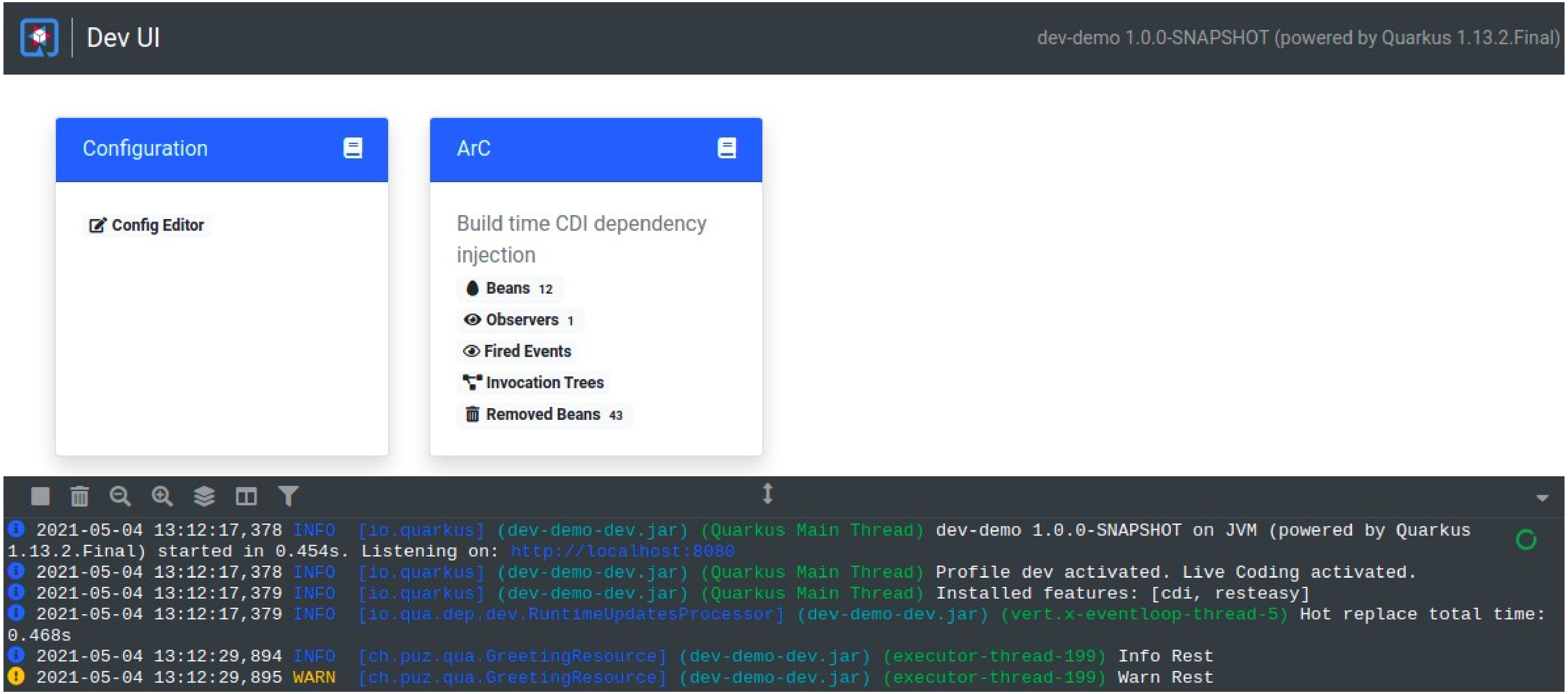


Image source: quarkus.io

Developer Joy

Development UI - http://localhost:8080/q/dev



Developer Joy

- Fast start (easy bootstrap)
- Live reload
- Easy Configuration
- Documentation & Guides
- Development UI
  - Change configurations, Log console, Extension Integration

Agenda - Day 1

- Microservices architecture
- Quarkus introduction
- MicroProfile specification
- RESTful microservices with Quarkus
- Building docker containers

What is MicroProfile?

«MicroProfile is an open-source community specification for Enterprise Java microservices», 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

Warum MicroProfile?

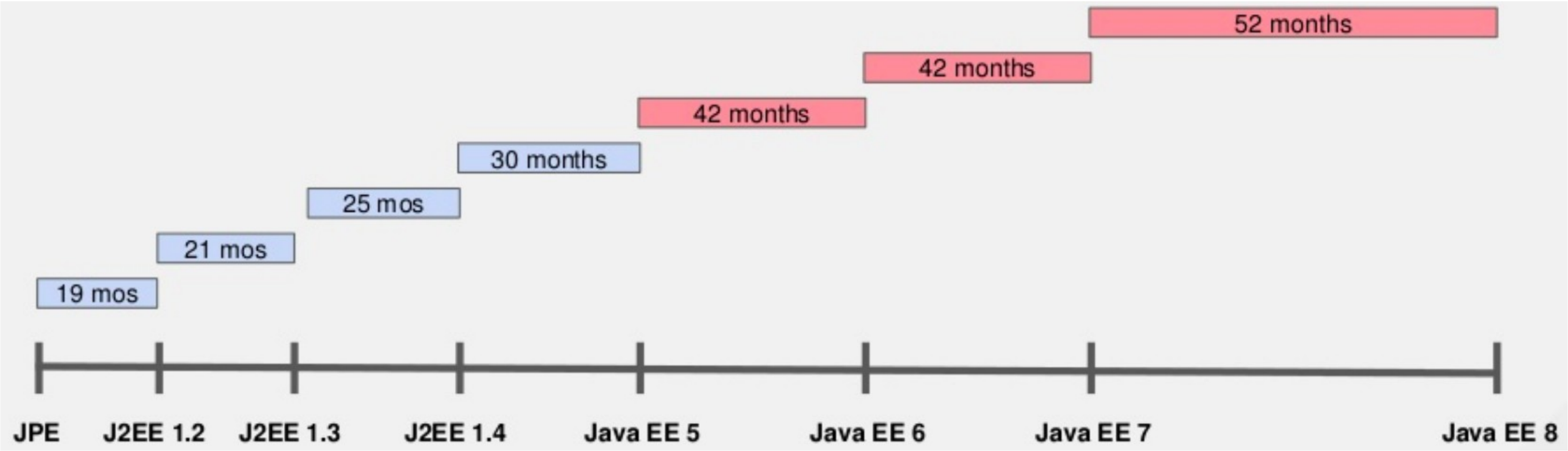
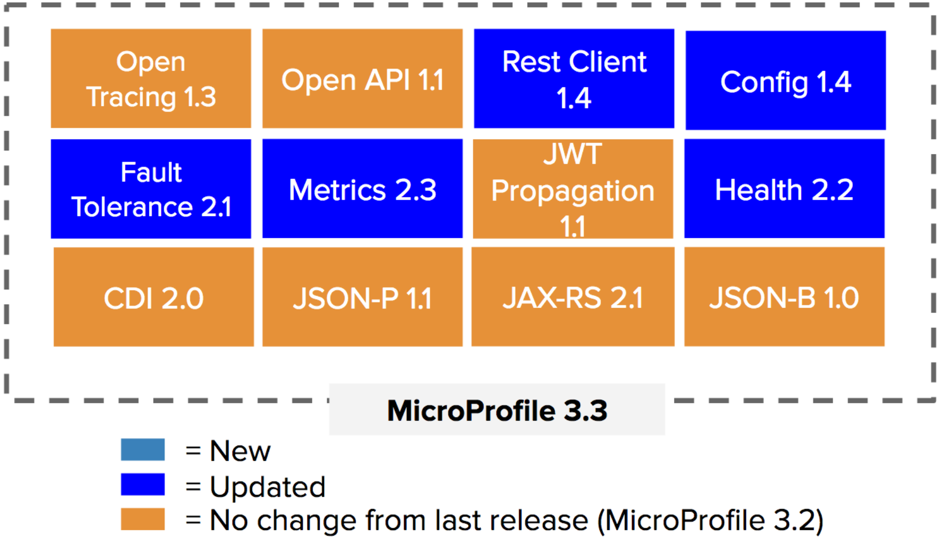


Image source: Red Hat

MicroProfile 1.0 ...

September 2016

February 2020



MicroProfile 4.0 (Current)

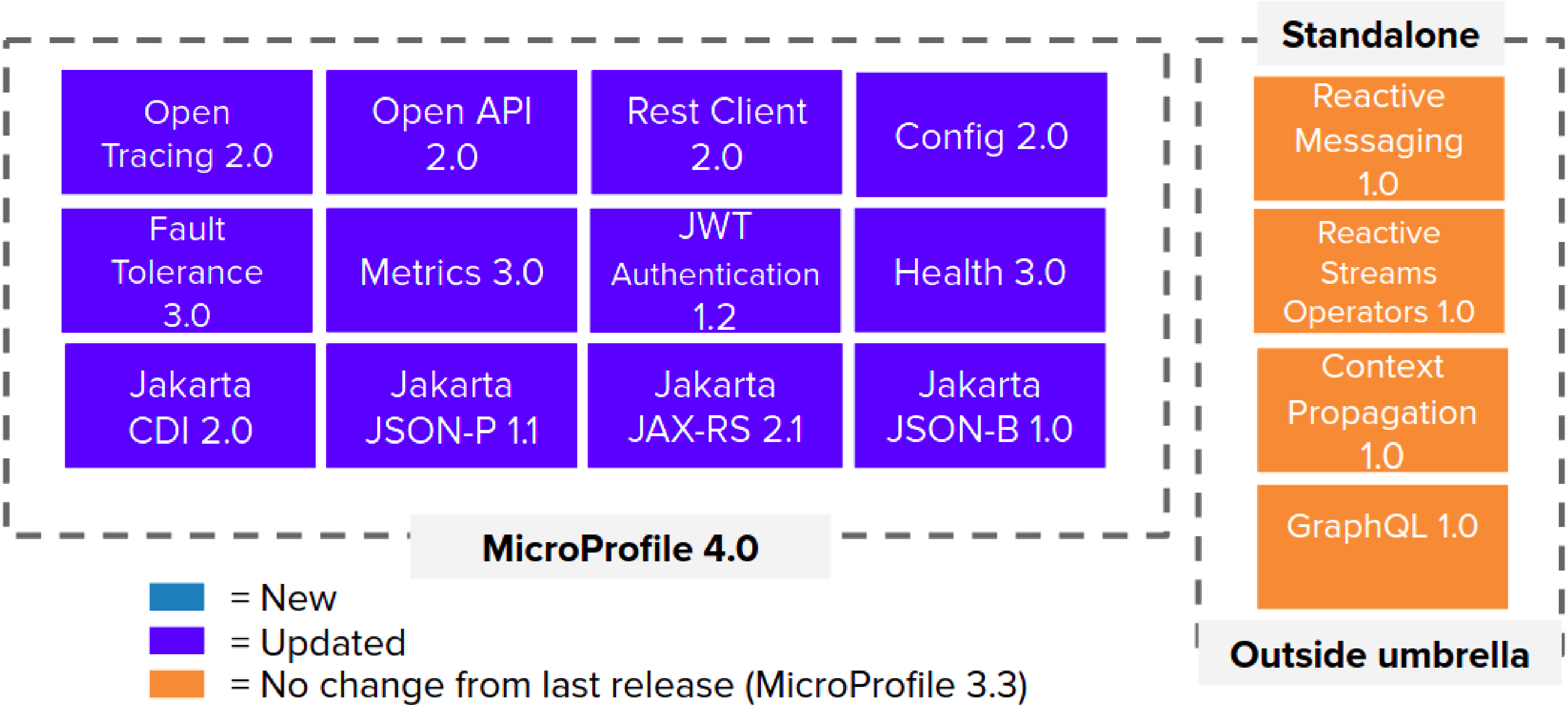
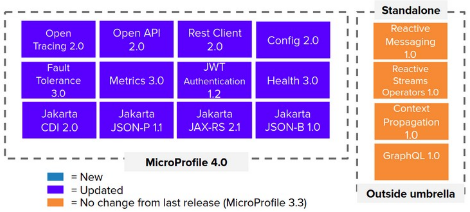


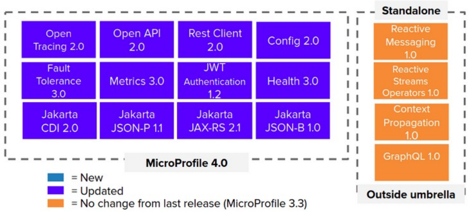
Image source: microprofile.io

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 2.0

MicroProfile Configuration



Injection @ConfigProperty

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

Programatically

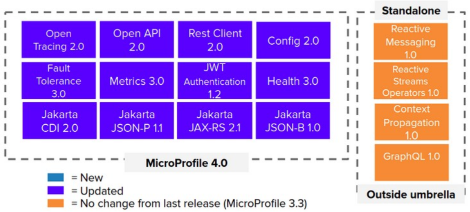
```
ConfigProvider.getConfig().getValue("greeting.suffix", String.class);
```

With @ConfigProperty

```
@ConfigProperties(prefix = "greeting")
public interface GreetingConfiguration {

    @ConfigProperty(defaultValue = "!")
    String getSuffix();
}
```

MicroProfile Rest Client



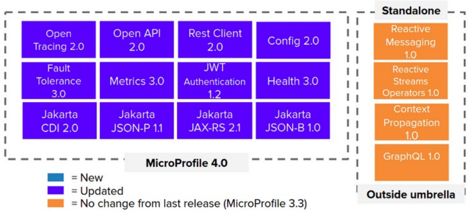
- Make typesafe calls to Rest APIs
- Defined as Java Interface
- Details about configuration? → Rest Client 2.0

```
@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)
```

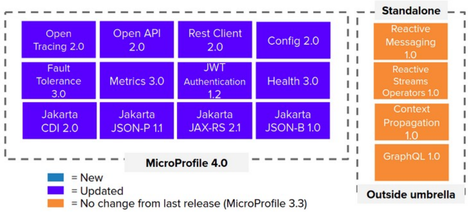
MicroProfile Metrics



- Add monitoring endpoint /metrics
  - OpenMetrics (text/plain), JSON (application/json)
- Expose metrics from application
- There is a shift of MP Metrics to Micrometer!
- Details about configuration? → MP Metrics 3.0

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

MicroProfile Health

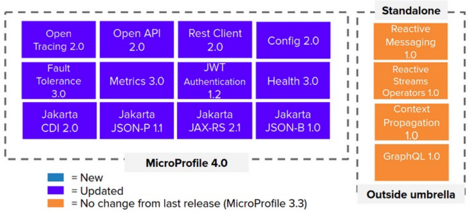


- Validate status and availability of application
- Machine to machine M2M focused
  - Replace unhealthy instances with new health instances
  - Monitored by container platforms
- Provides «Liveness» and «Readiness» endpoint /health
- Details about configuration? → MP Health 3.0

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

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

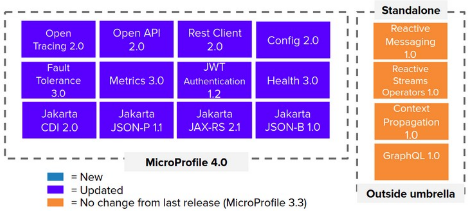
MicroProfile Fault Tolerance



- Toolkit for resilient applications
  - Timeout, Retry, Fallback, Circuit Breaker, Bulkhead
- Designed to separate execution logic from execution
- Details about configuration? → MP Fault Tolerance 3.0

```
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, CurcuitBreakerOpenException
    @CircuitBreaker(successThreshold = 10, requestVolumeThreshold = 4, failureRatio=0.5, delay = 1000)
    // maximum 10 concurrent requests allowed
    @Bulkhead(10)
    Response greet();
}
```

MicroProfile Tracing



- Trace request flow across service boundaries
  - Support for JDBC, Kafka or MongoDB available
- Focused on easy instrumentation of services
- Rest endpoints are automatically traced
  - Tracing further classes or methods with @Traced
- Details about configuration? → MP OpenTracing 2.0

```
@Traced
@ApplicationScoped
public class GreetingService {

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

SmallRye

«APIs and implementations tailored for Cloud development, including but not limited to, Eclipse MicroProfile», 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/index.html

SmallRye Reactive Messaging Documentation:

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

MicroProfile 4.0 Release Presentation:

https://drive.google.com/drive/folders/1zqIzegskeYUCualDek3Wq4Sle9qW-_eY

Agenda - Day 1

- Microservices architecture
- Quarkus introduction
- MicroProfile specification
- RESTful microservices with Quarkus
- Building docker containers

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	REACTIVE

IMPERATIVE

```java
@Inject
SayService say;

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

REACTIVE

```java
@Inject @Channel("kafka")
Publisher<String> reactiveSay;

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

Image source: quarkus.io

Agenda - Day 1

- Microservices architecture
- Quarkus introduction
- MicroProfile specification
- RESTful microservices with Quarkus
- Building docker containers

Quarkus structure

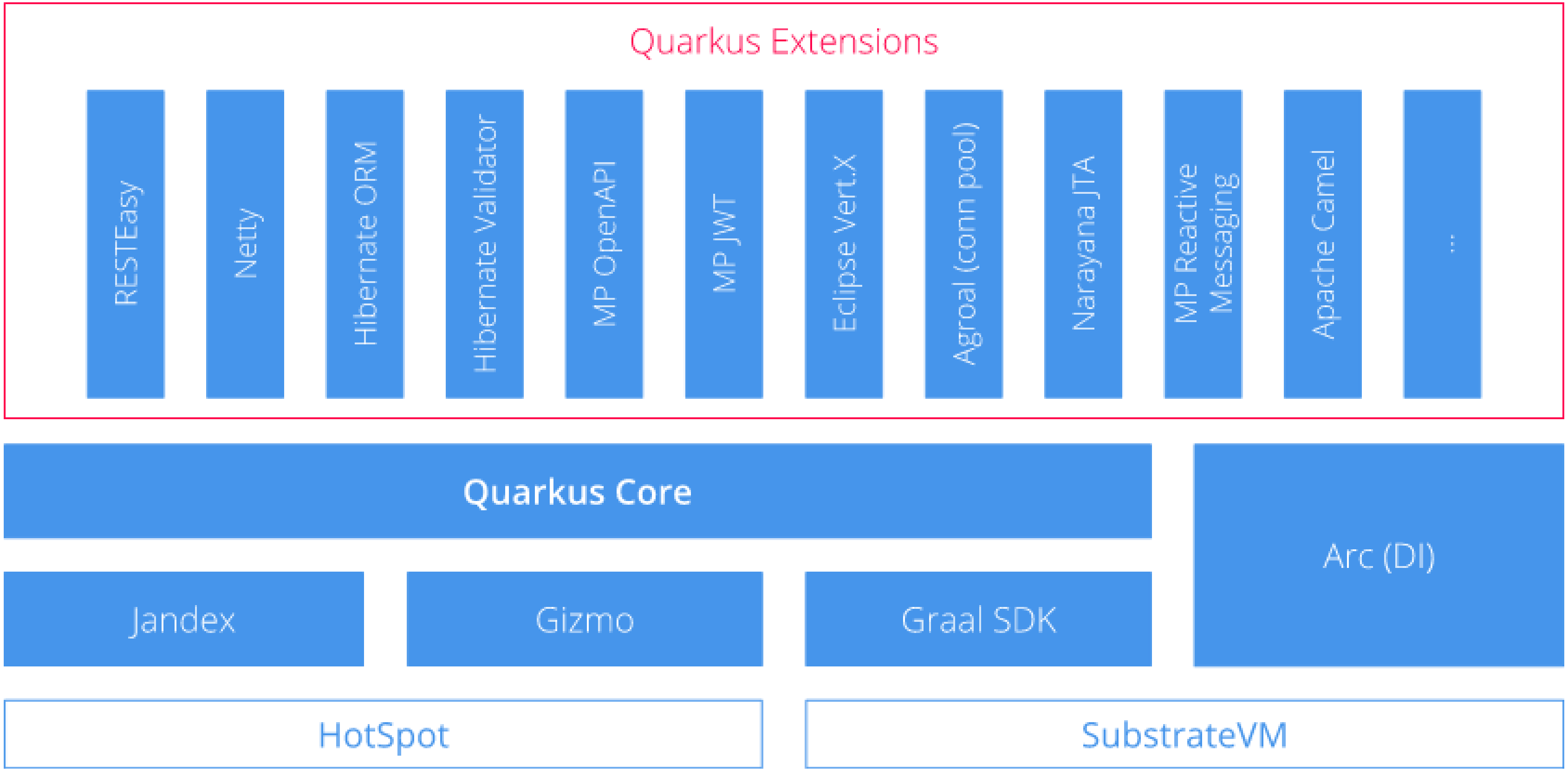
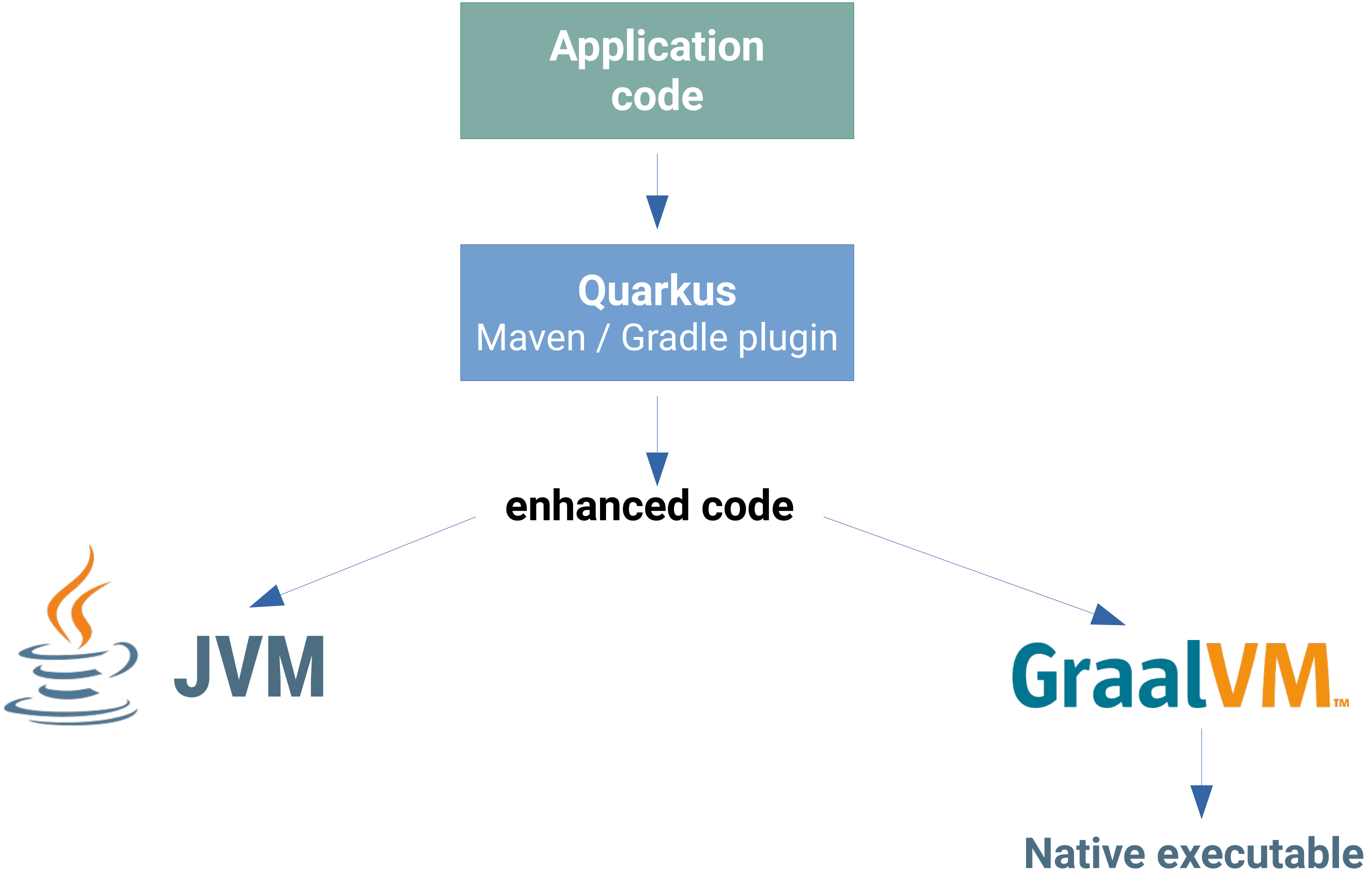


Image source: Red Hat

Building Docker containers



Agenda - Day 2

- Cloud patterns
- Continuous integration and delivery (CICD) with Tekton
- Event driven architecture and messaging with Apache Kafka
- Observability – Metrics and Tracing
- Writing your own Quarkus extension

Questions Day 1?

Agenda - Day 2

- Cloud patterns
- Continuous integration and delivery
- Event driven architecture and messaging with Apache Kafka
- Observability – Metrics and Tracing
- Writing your own Quarkus extension

05.05.2021

Cloud patterns

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

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]»

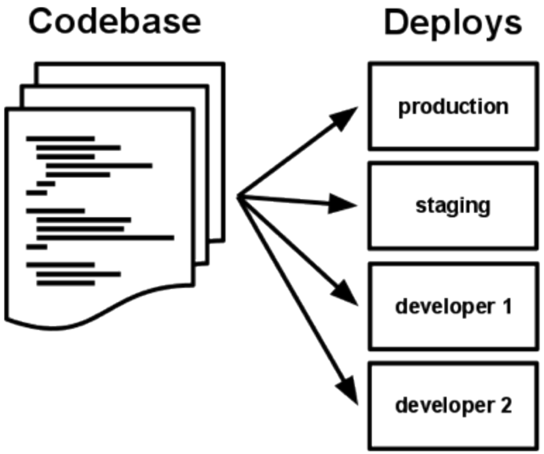
Hofmann, Michael; Schnabel, Erin; Stanley, Katherine (13 March 2017). Microservices Best Practices for Java. IBM Redbooks. pp. 2–3. ISBN 9780738442273.

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 seperated codebase
- One codebase many deployments
- Deployments should differ in config



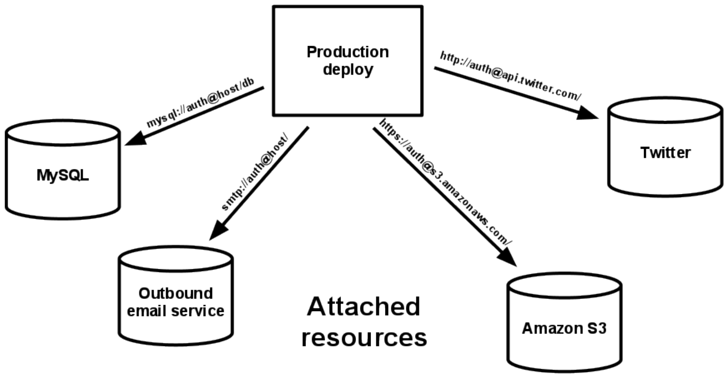
III. Config

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

```yaml
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
```

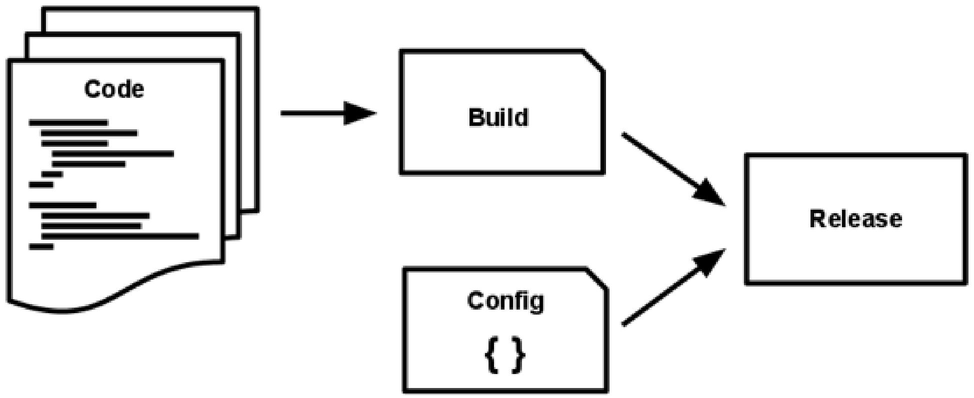
IV. Backing services

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



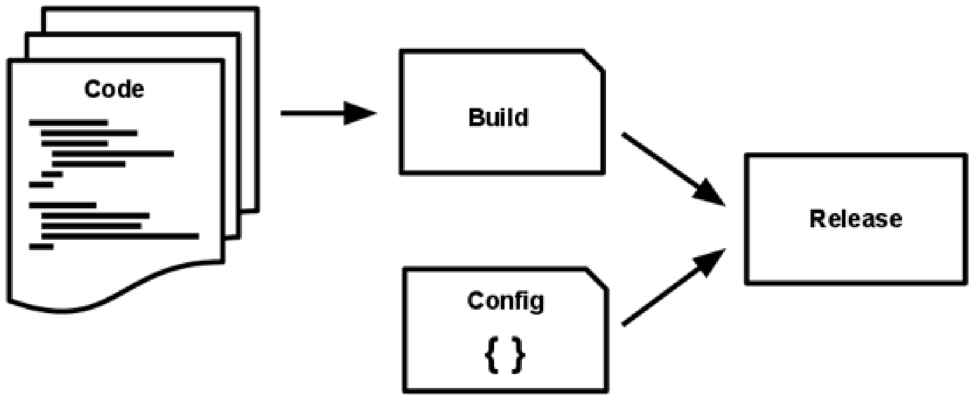
V. Build, release, run

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



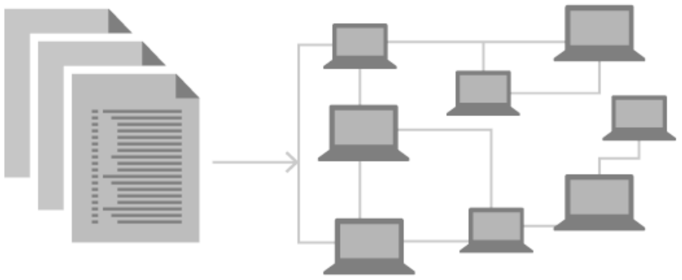
V. Build

- 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

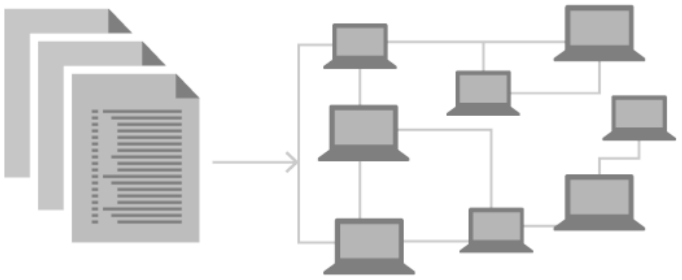
- Deliver the parameterizable deployable to environment
- Use environment variables to fill parameters to your needs
- Release fast and often
- Best practice: Infrastructure as code



V. Run

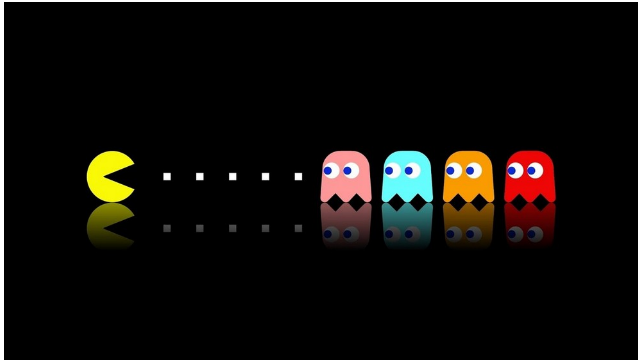
- Delivered deployable should run on each environment

- 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

05.05.2021

Cloud patterns

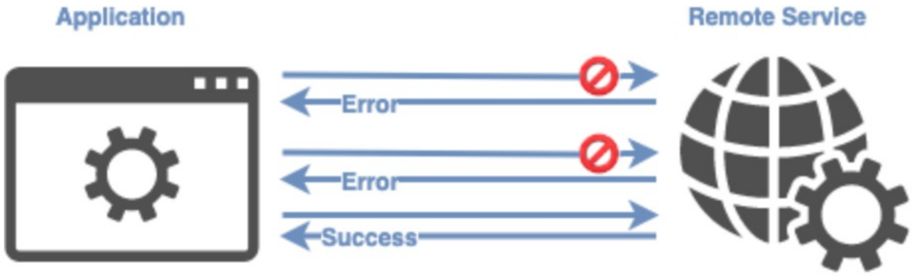
- The Twelve-Factor App
- Design for Failure (Retry, Timeout, Circuitbreaker)
- Microservice Communication
- Orchestration vs. Choreography
- Event Driven Architecture

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

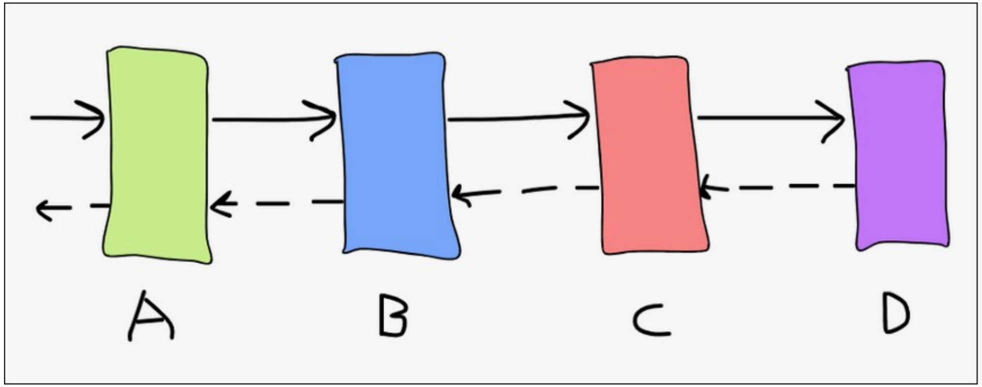


Timeout

- 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

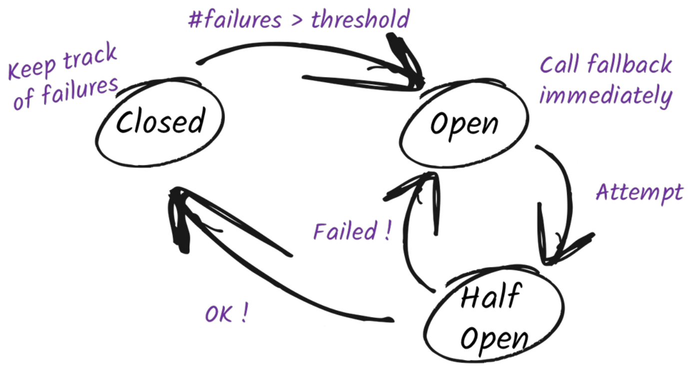
- Fallbacks can be defined when alternatives are available
- Whenever a remote source fails, repeat on fallback
- Example: Use last cached result instead of new query
- Careful compromise between robustness and vulnerability and loss of consistency

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



05.05.2021

Cloud patterns

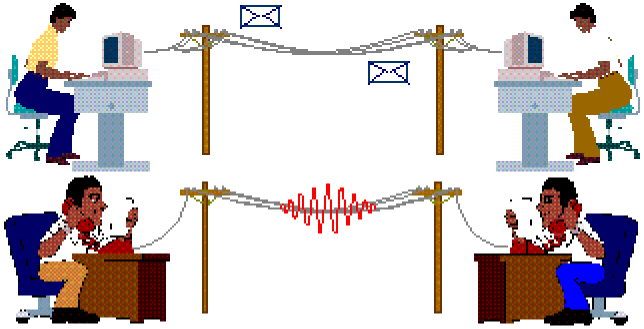
- The Twelve-Factor App
- Design for Failure (Retry, Timeout, Circuitbreaker)
- Microservice Communication
- Orchestration vs. Choreography
- Event Driven Architecture

Microservice Communication

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

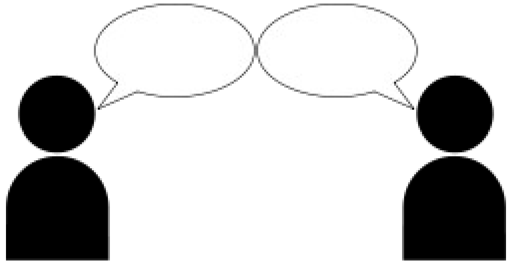
Synchronous vs Asynchronous

- 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

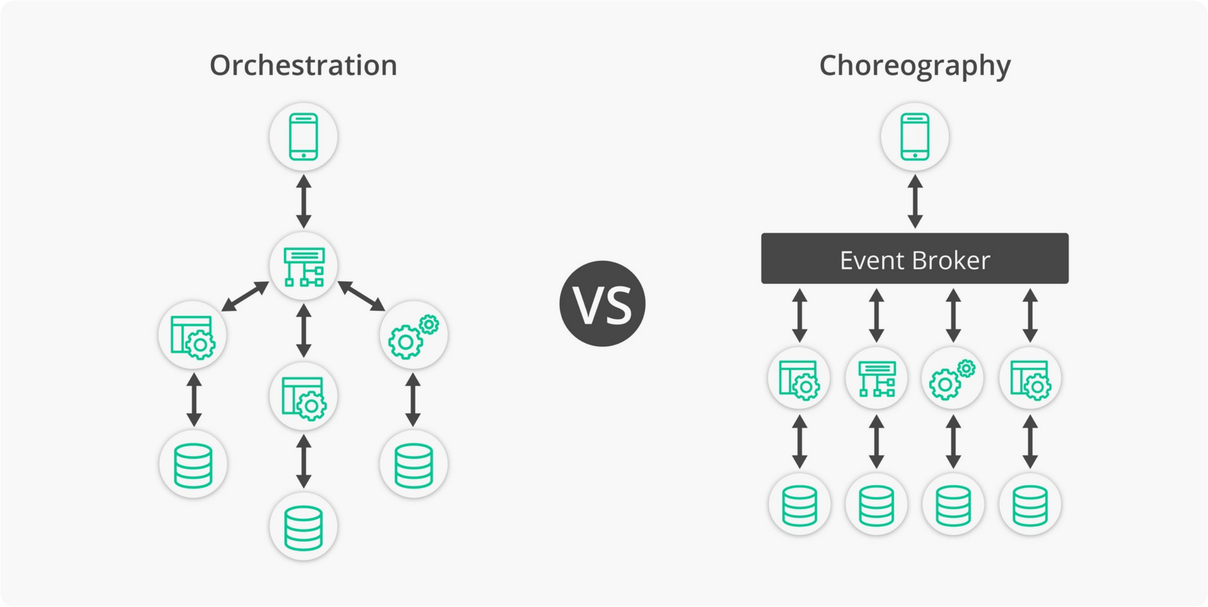
05.05.2021

Cloud patterns

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

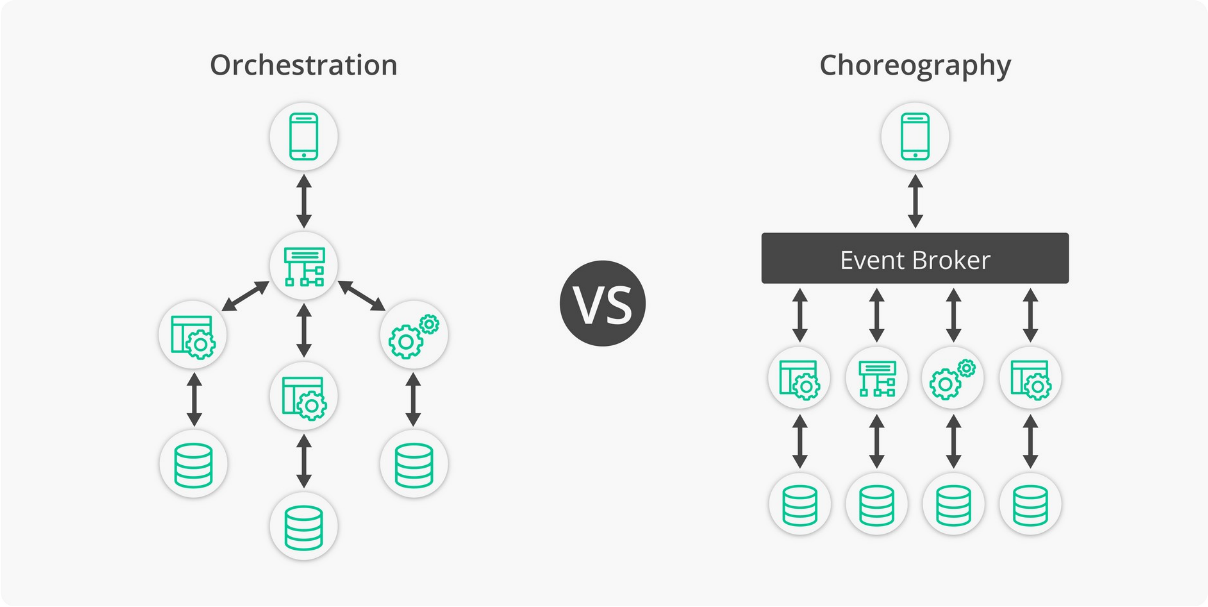
Orchestration vs. Choreography

- Different approach for communication
- Responsibility question



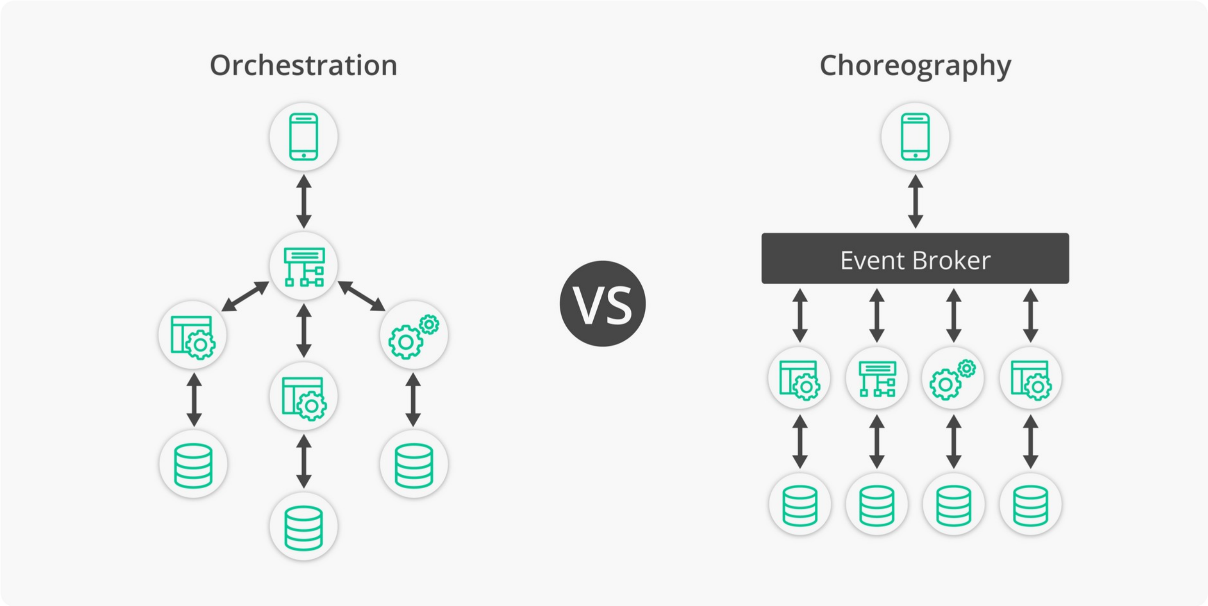
Choreography

- Each part knows it's responsibility
- Microservice knows the workflow



Orchestration

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



05.05.2021

Cloud patterns

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

Event Driven Architecture

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

Event Driven Architecture

```
+------------+                    +------------+                    +--------------+
| shop       |       create order | order      |    process payment |   payment    |
| component  +------------------->+ component  +------------------->+   component  |
+------------+                    +------------+                    +--------------+
```

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

```
              +------------------+                +----------------+
      +------->    orders        |---+       +---> payment request +-----+
      |       +------------------+   |       |    +----------------+     |
      |                              |       |                           |
+-------------+                    +-v----------+                    +---v----------+
| shop        |                    | order      |                    |   payment    |
| component   |                    | component  |                    |   component  |
+-------------+                    +------------+                    +--------------+
```

Agenda - Day 2

- Cloud patterns
- Continuous integration and delivery
- Event driven architecture and messaging with Apache Kafka
- Observability – Metrics and Tracing
- Writing your own Quarkus extension

05.05.2021

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.»

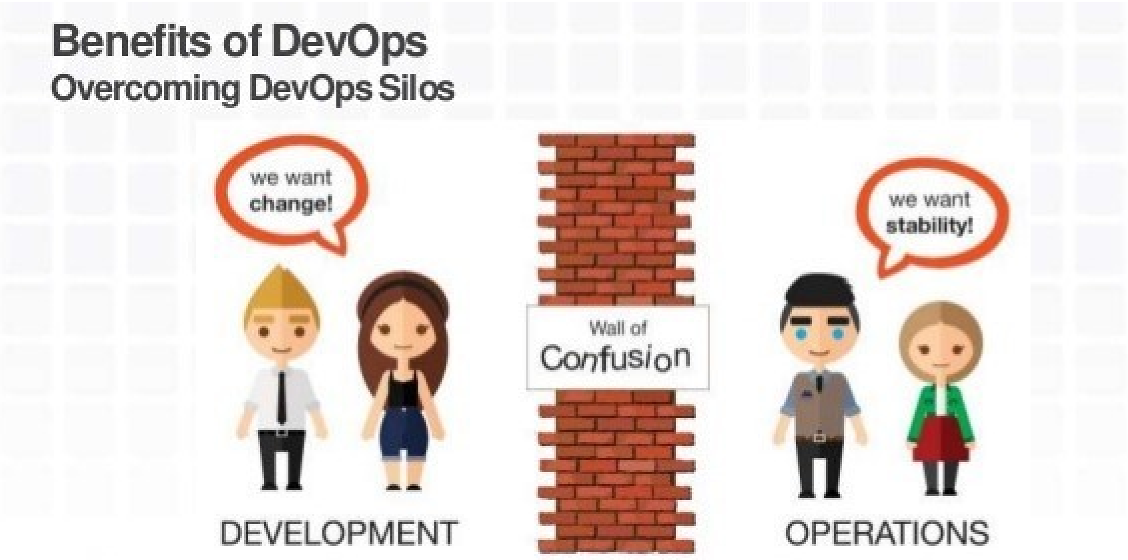
https://aws.amazon.com/devops/what-is-devops/

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/

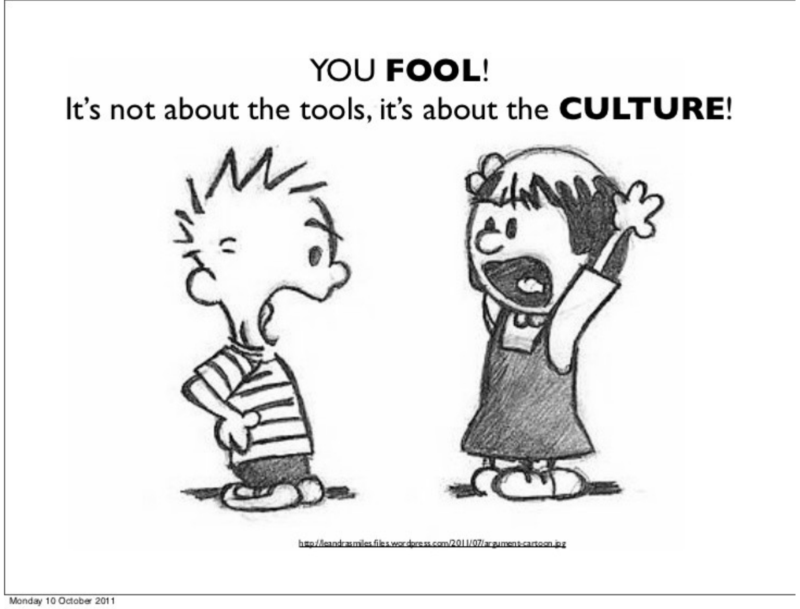
DevOps


Benefits of DevOps
Overcoming DevOps Silos
we want change!
Wall of Confusion
we want stability!
DEVELOPMENT
OPERATIONS

DevOps - Before

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

DevOps culture

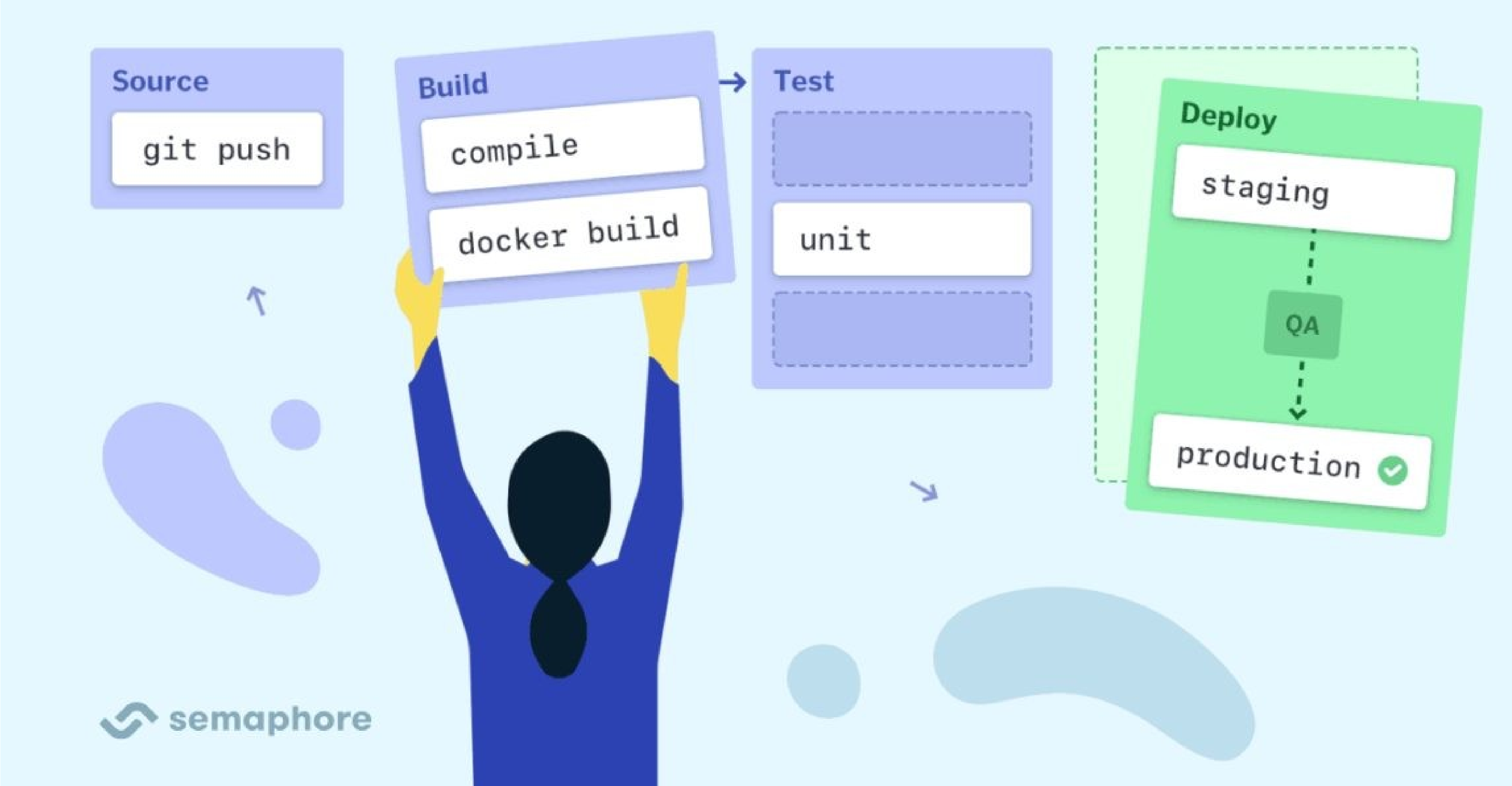


- Tear down wall of confusion
- Enable Devs to be Ops
- Integrate Operations to daily development
- Mindset should develop

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

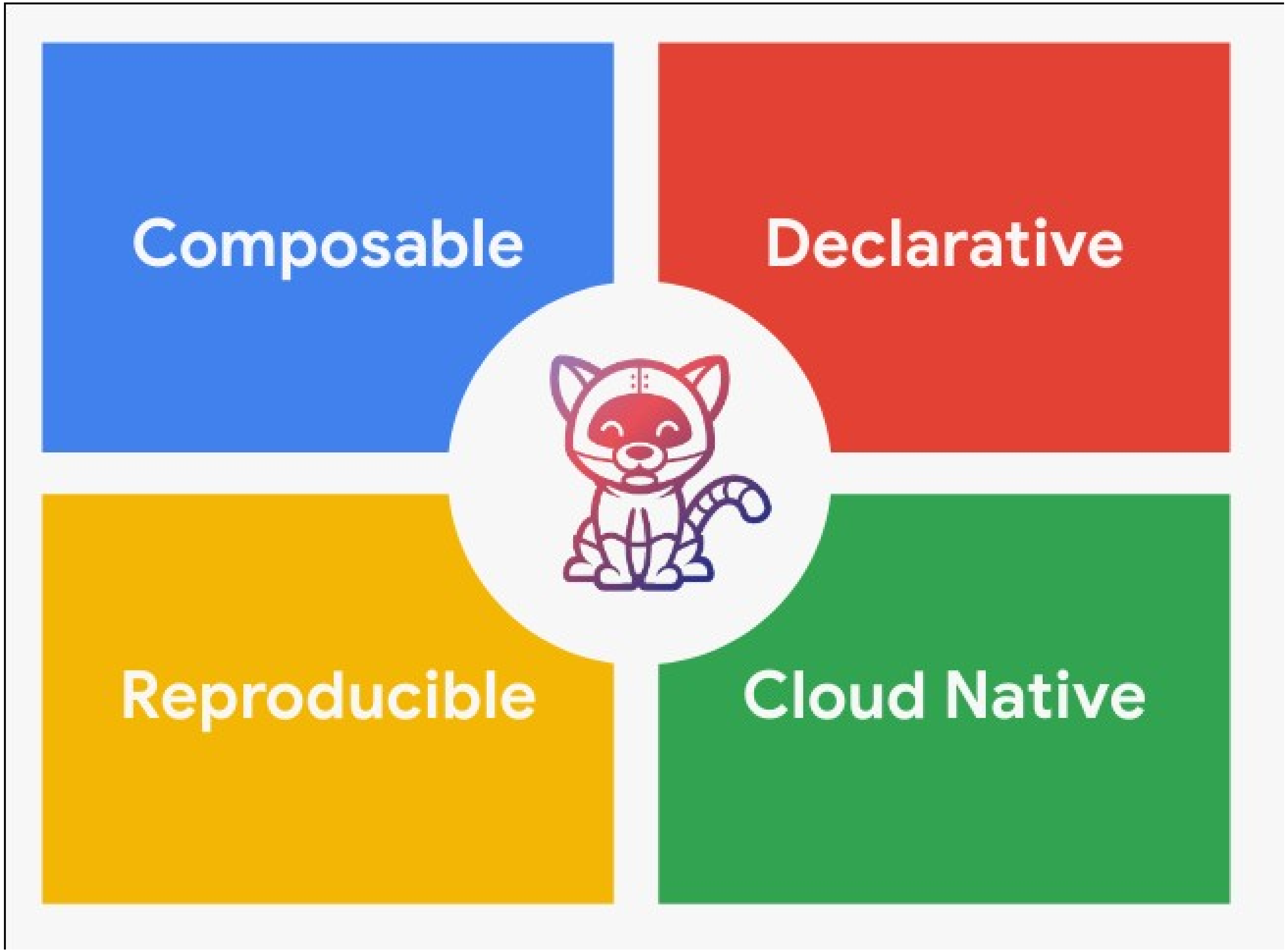
- API changes need to be categorized
- Non-breaking changes deployed fast
- Breaking changes need to deploy all dependent services
- Communication is key

TEKTON

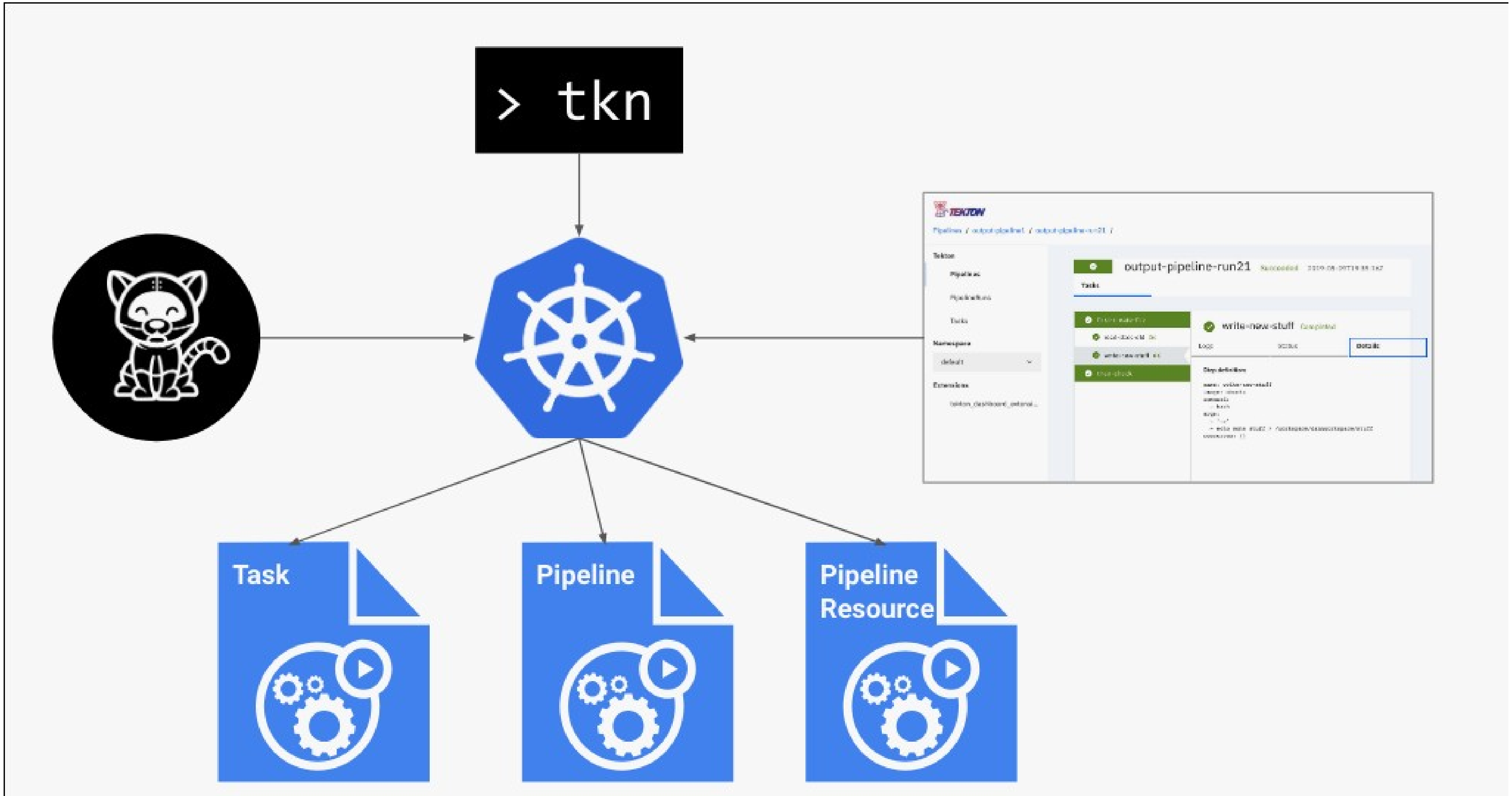
Tekton

- Open API spec describing CI/CD pipelines
- Open-source CI/CD plattform implementation
- Kubernetes native

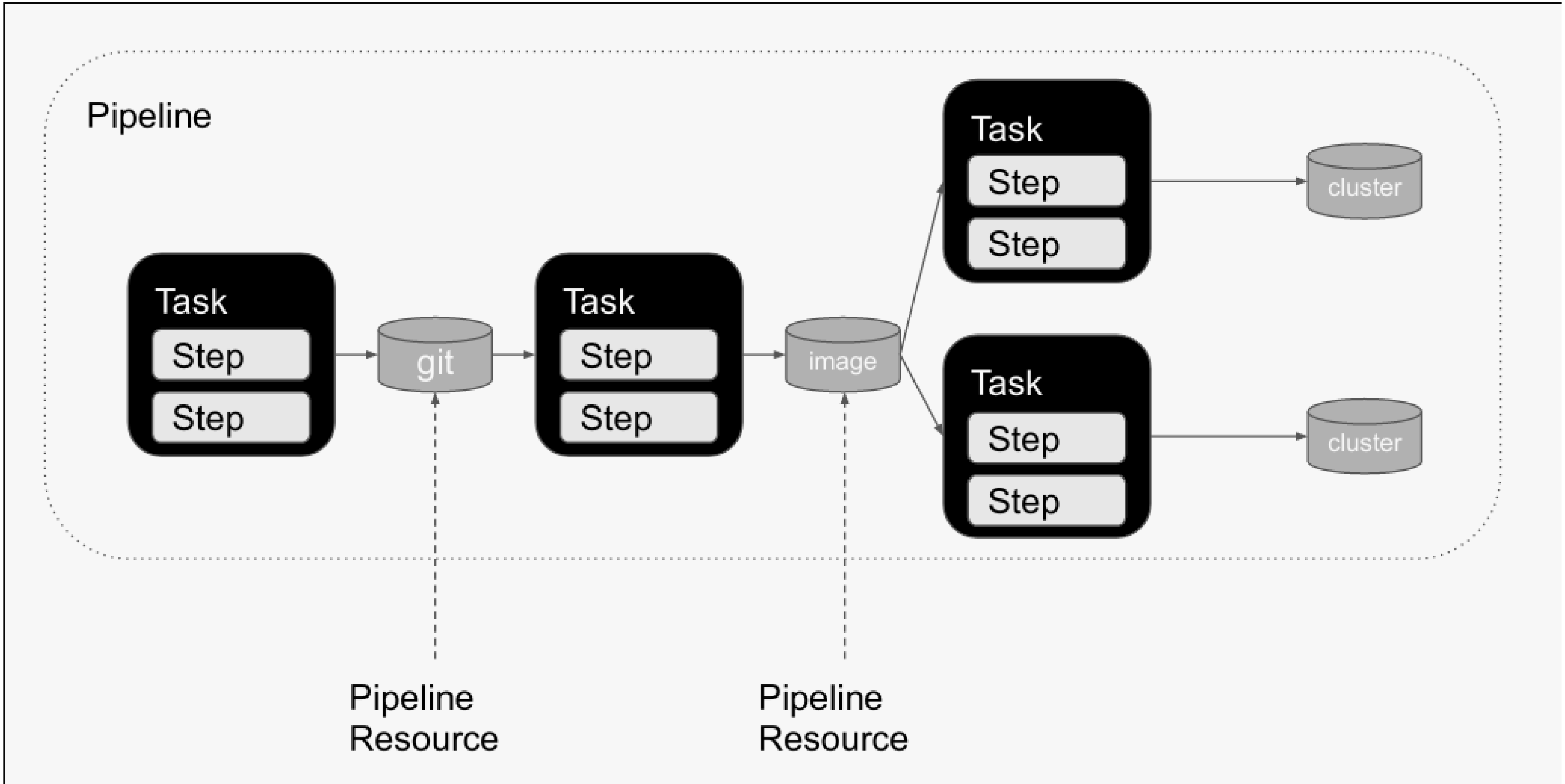
Tekton Goals



Tekton Architecture



Tekton CRD



Pipeline
Task
Step
Step
git
Task
Step
Step
image
Task
Step
Step
cluster
Task
Step
Step
cluster
Pipeline Resource
Pipeline Resource

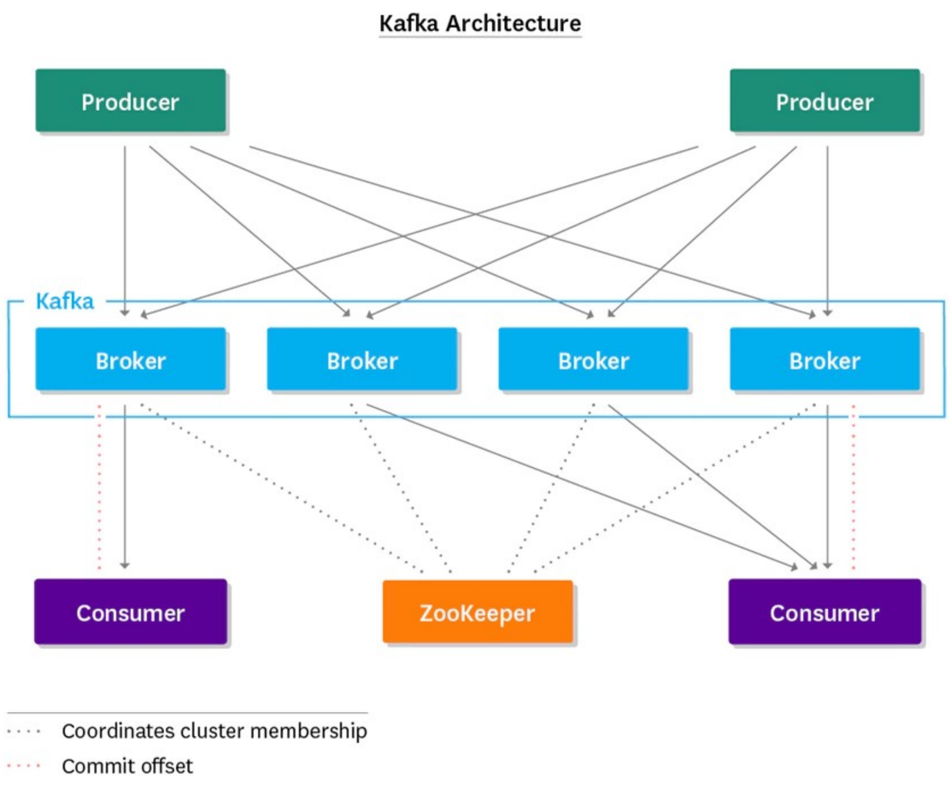
Agenda - Day 2

- Cloud patterns
- Continuous integration and delivery
- Event driven architecture and messaging with Apache Kafka
- Observability – Metrics and Tracing
- Writing your own Quarkus extension

Apache Kafka

- Publish-Subscribe model
- Write once read many
- Highly scalable
- Performant
- Durability

A distributed streaming platform



Terminology

Broker

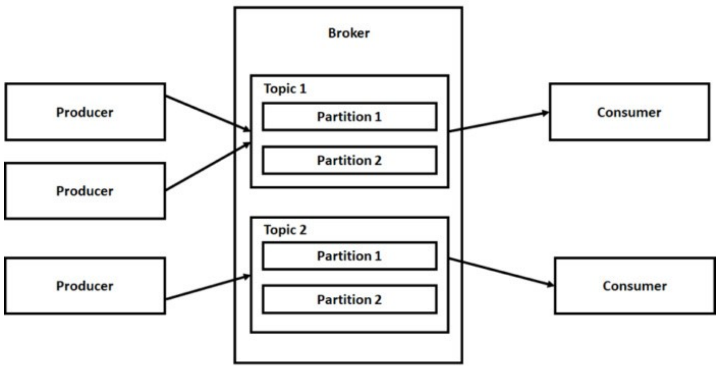
- Designed High-Available
- Data is replicated
- Data retention by time or size
- Data compaction by key

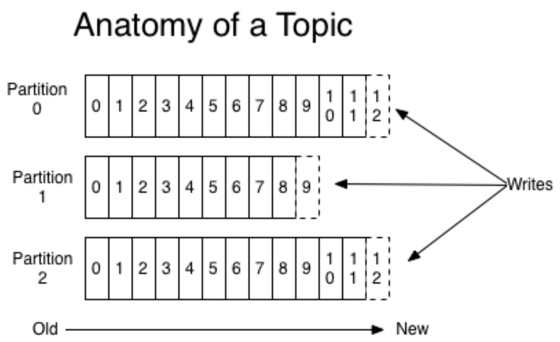
Topic

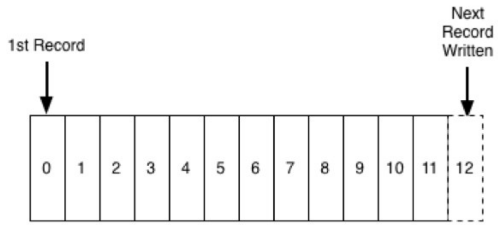
- Where messages are published
- Topics are partitioned

Log / Partition

- Append only
- Totally ordered sequence (by time)
- Contain messages
- Messages are immutable







Why Apache Kafka

- Decoupling of Consumers and Producers
- Asynchronous communication
- Enables Near-Realtime features
- Use Cases
  - Messaging platform
  - Integration platform
  - Stream processing
  - Data store / event store in Event Sourcing
  - Event driven Microservices

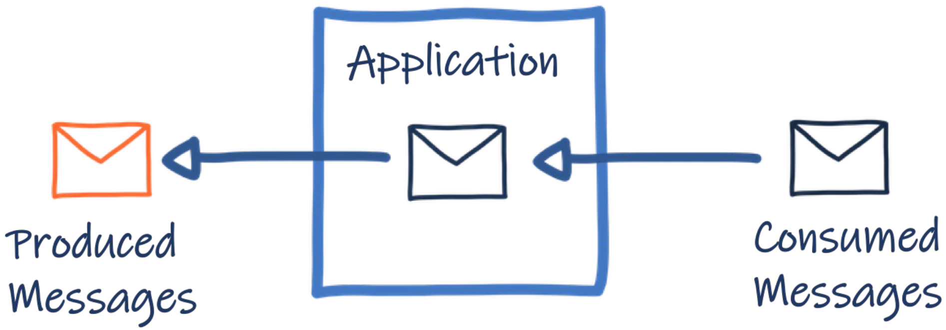
Reactive Messaging

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

Reactive Messaging Concepts

Message, Payload, Metadata

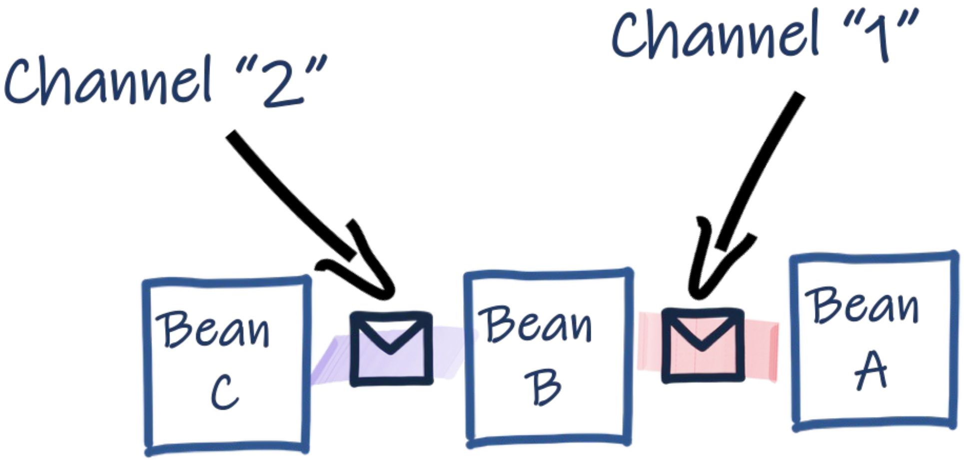
- Application receives, processes and send messages
- Message is an envelop around payload
- Messages can contain metadata (e.g tracing context)



Reactive Messaging Concepts

Channels and Streams

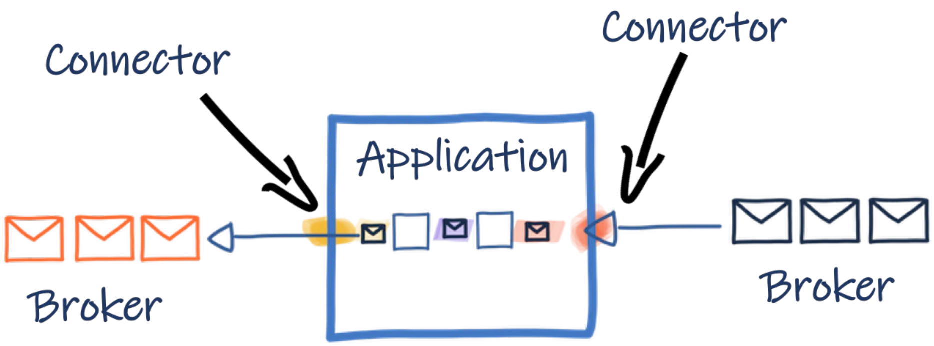
- Messages transit on channels
- Virtual destination (identified by name)



Reactive Messaging Concepts

Connectors

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



Reactive Messaging Example

Application Class

```java
/**
 * 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
```

connector type

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

channel

kafka topic

Agenda - Day 2

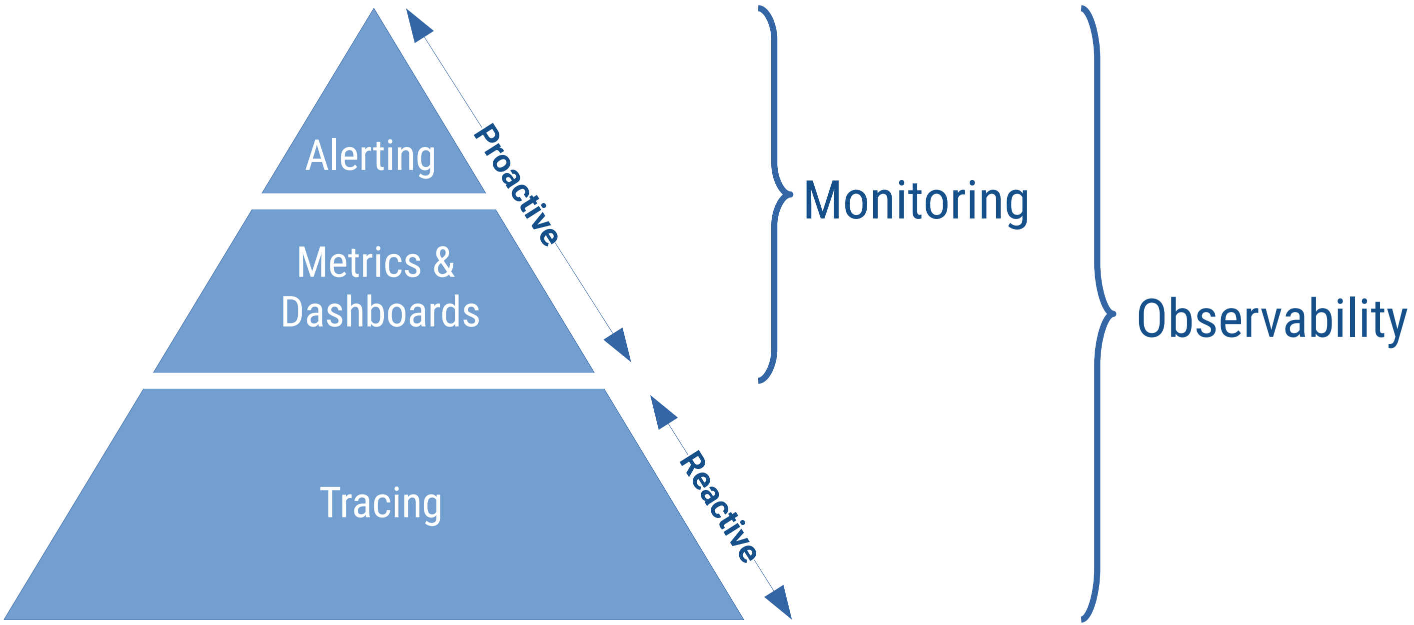
- Cloud patterns
- Continuous integration and delivery
- Event driven architecture and messaging with Apache Kafka
- Observability – Metrics and Tracing
- Writing your own Quarkus extension

Observability

- What is observability
- Why we need it
- Three pillars of observability

Observability extends Monitoring

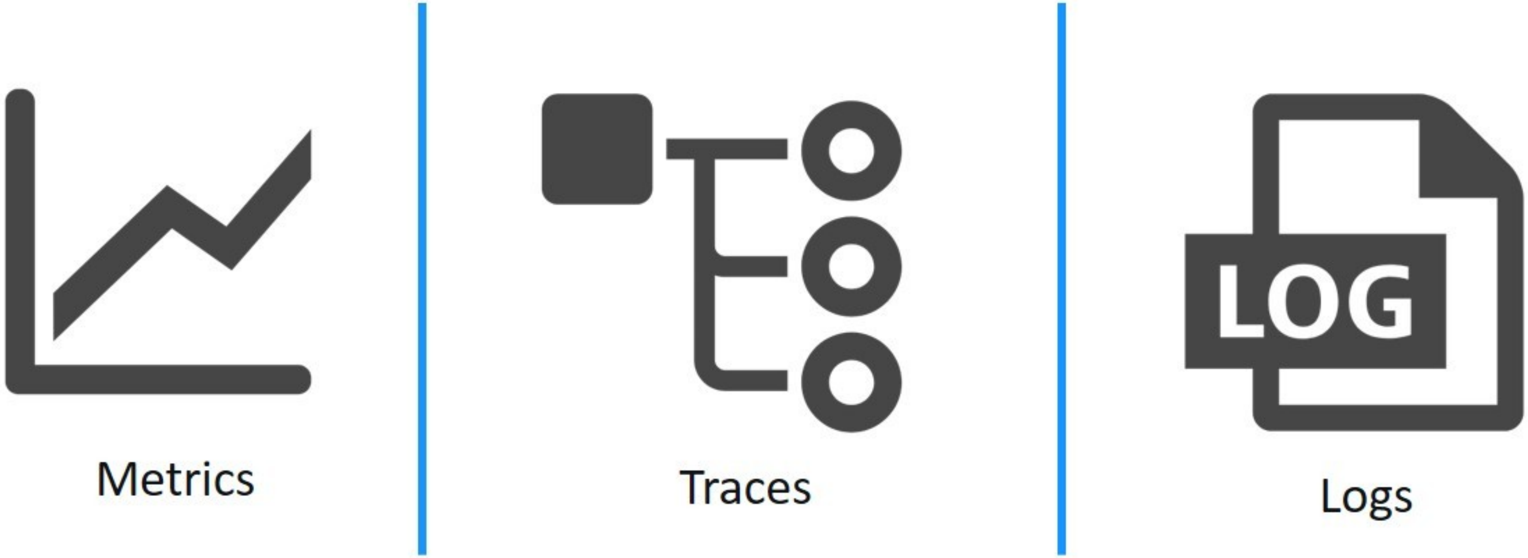
«Monitoring tells you whether a system is working, observability lets you ask why it is not working.», Baron Schwartz 2017



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



source: dynatrace.com

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

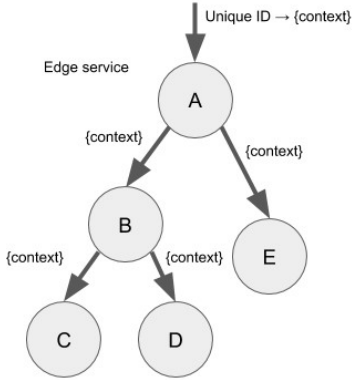
```
2020-07-12 11:45:34 Transaction id 12398 failed on update
2020-09-23 19:23:11 POST /order - status:201 - response_ms:21
```

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

Tracing - Terminology

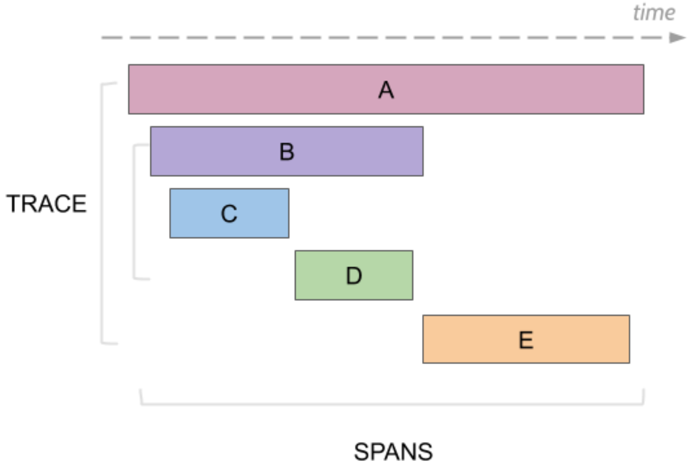
Trace: Execution path through the system

- 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

Agenda - Day 2

- Cloud patterns
- Continuous integration and delivery
- Event driven architecture and messaging with Apache Kafka
- Observability – Metrics and Tracing
- Writing your own Quarkus extension

Quarkus structure

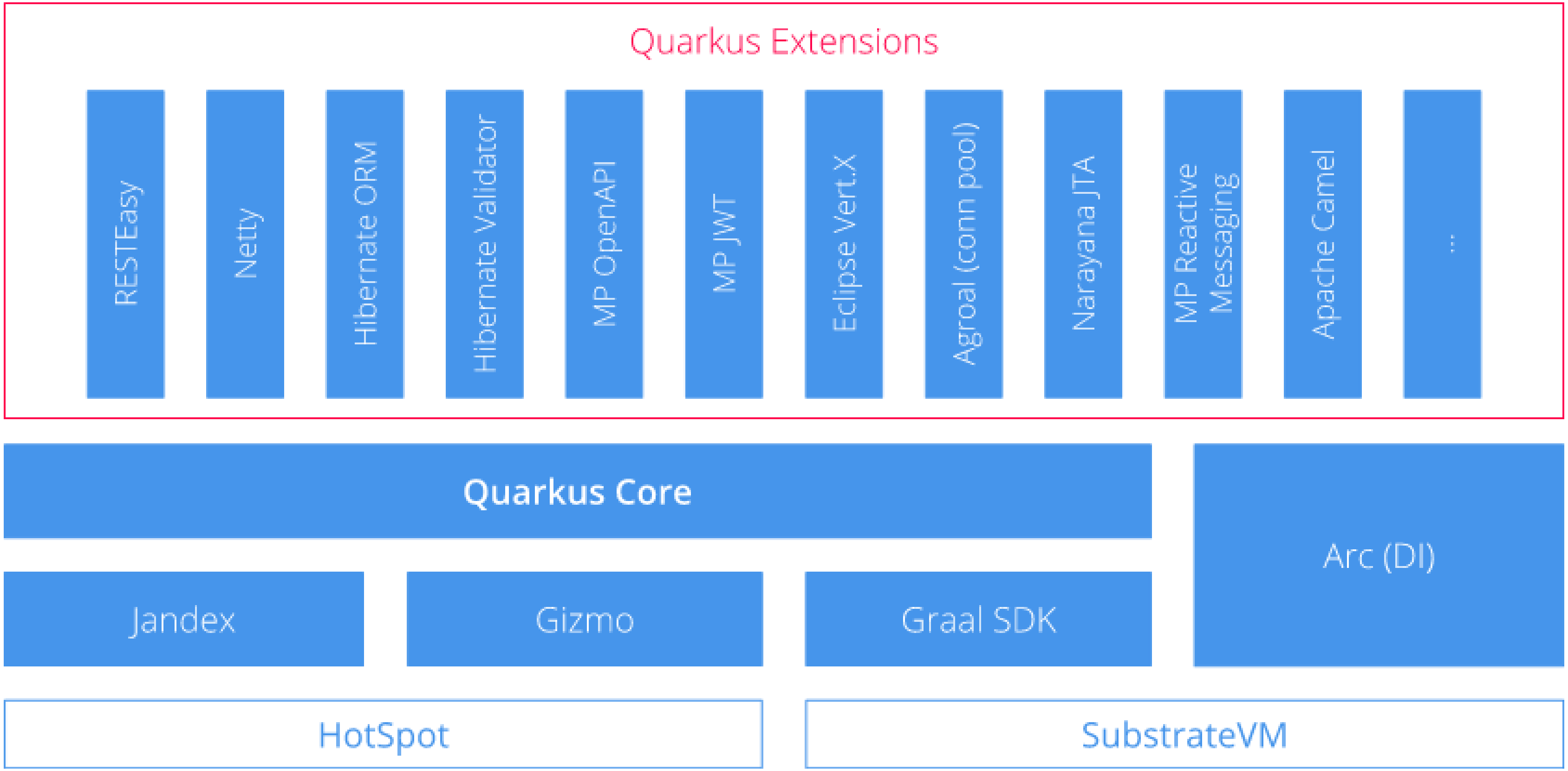


Image source: Red Hat

Extensions

- Extend Quarkus framework with custom functionality
- Contain information how to deploy extension
- Maven multimodule project
  - Deployment
  - Runtime
- Deployment depends on runtime
- Application depends on extension (runtime)

Deployment Module

- How to deploy the extension code
- BuildSteps with instructions run at build time
  - Find annotations or classes
  - Register additional beans for CDI
  - Add a servlet
- Can record invokations by using recorders from runtime module
- Dev UI integration

```
@BuildStep
ServletBuildItem createServlet() {

  return ServletBuildItem.builder("my-extension", MyExtensionServlet.class.getName())
      .addMapping("/greeting")
      .build();
}
```

Runtime Module

- Runtime features (extension code)
- 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-extension", phase = ConfigPhase.BUILD_TIME)
public final class MyExtensionConfig {
    @ConfigItem
    public String name;
}
```

application.properties

```
quarkus.my-extension.name=my build time property
```