“Simply reactive”

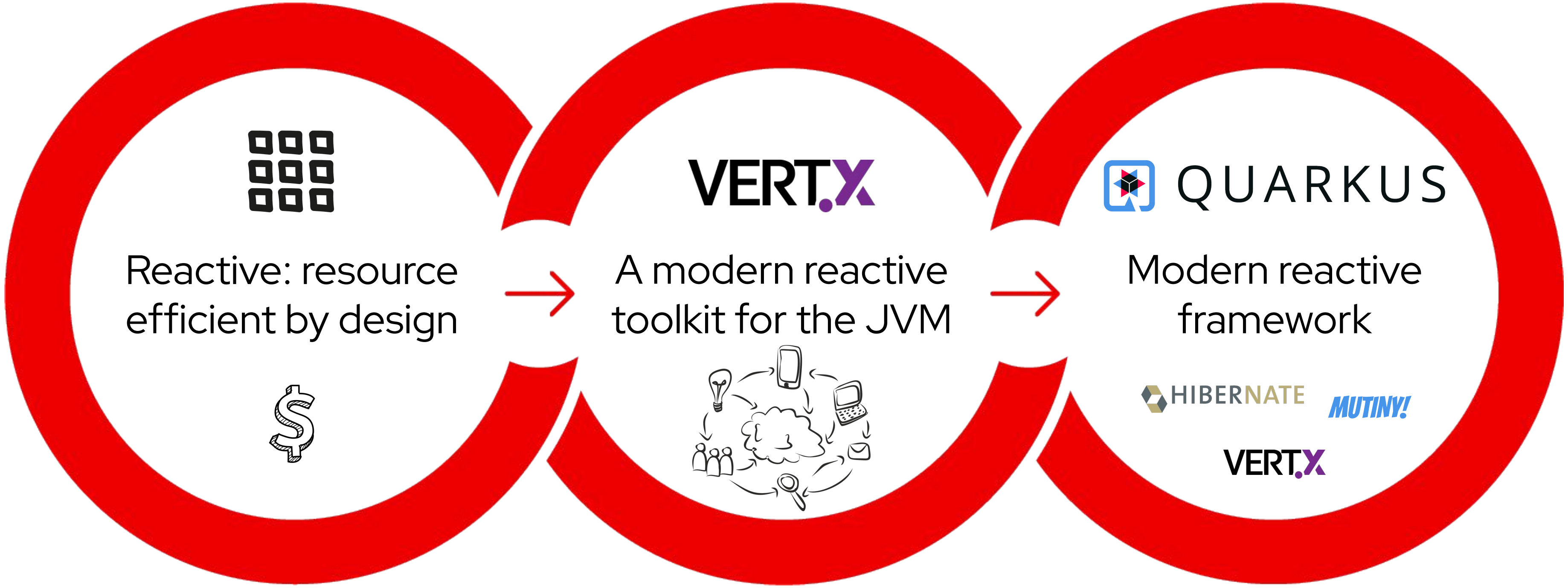
Vert.x, Mutiny, Hibernate Reactive and Quarkus

Julien Ponge

Principal Software Engineer

Agenda

A reactive continuum built by Red Hat

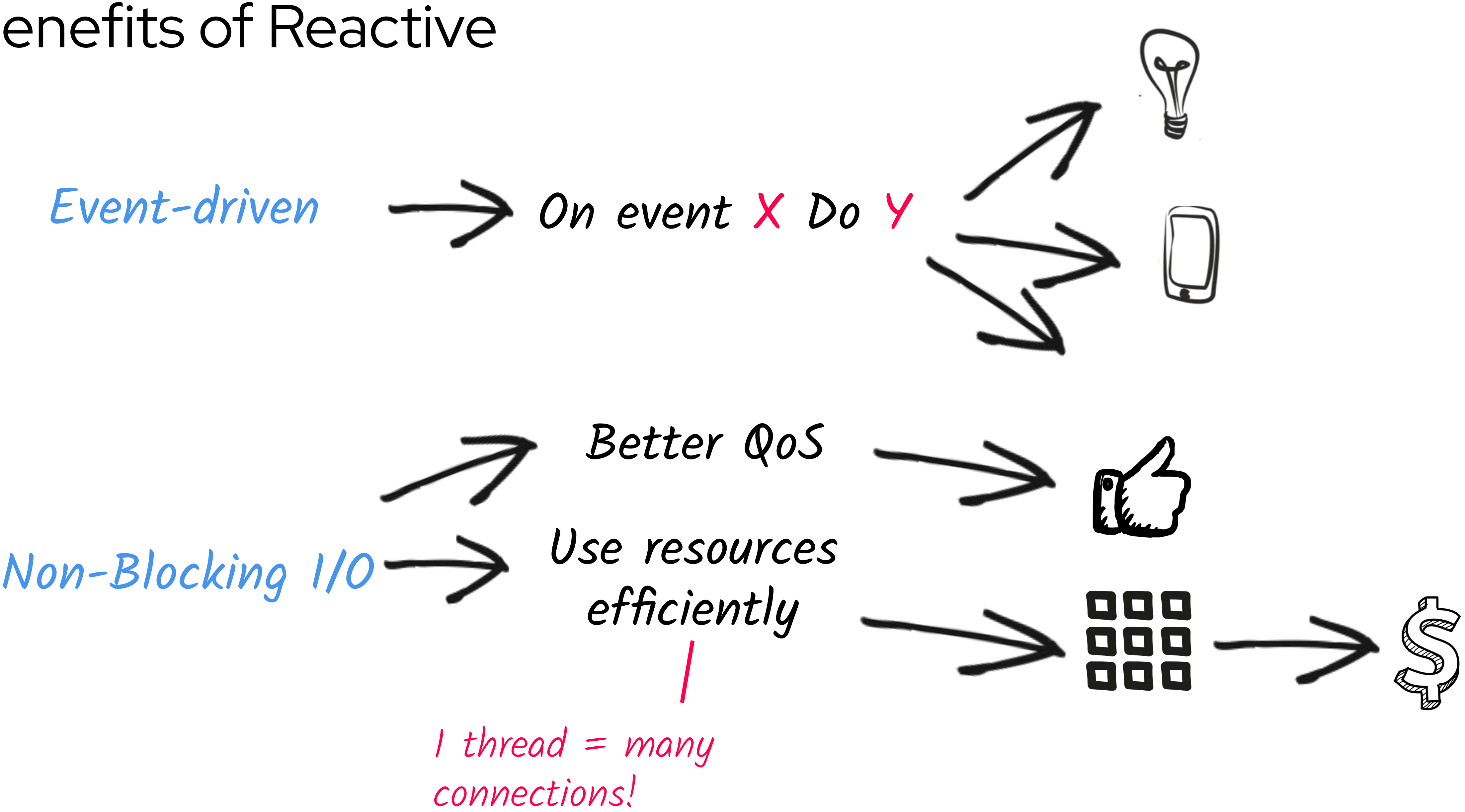


Reactive: resource efficient by design →

A modern reactive toolkit for the JVM →

Modern reactive framework

Benefits of Reactive



Event-driven

On event X Do Y

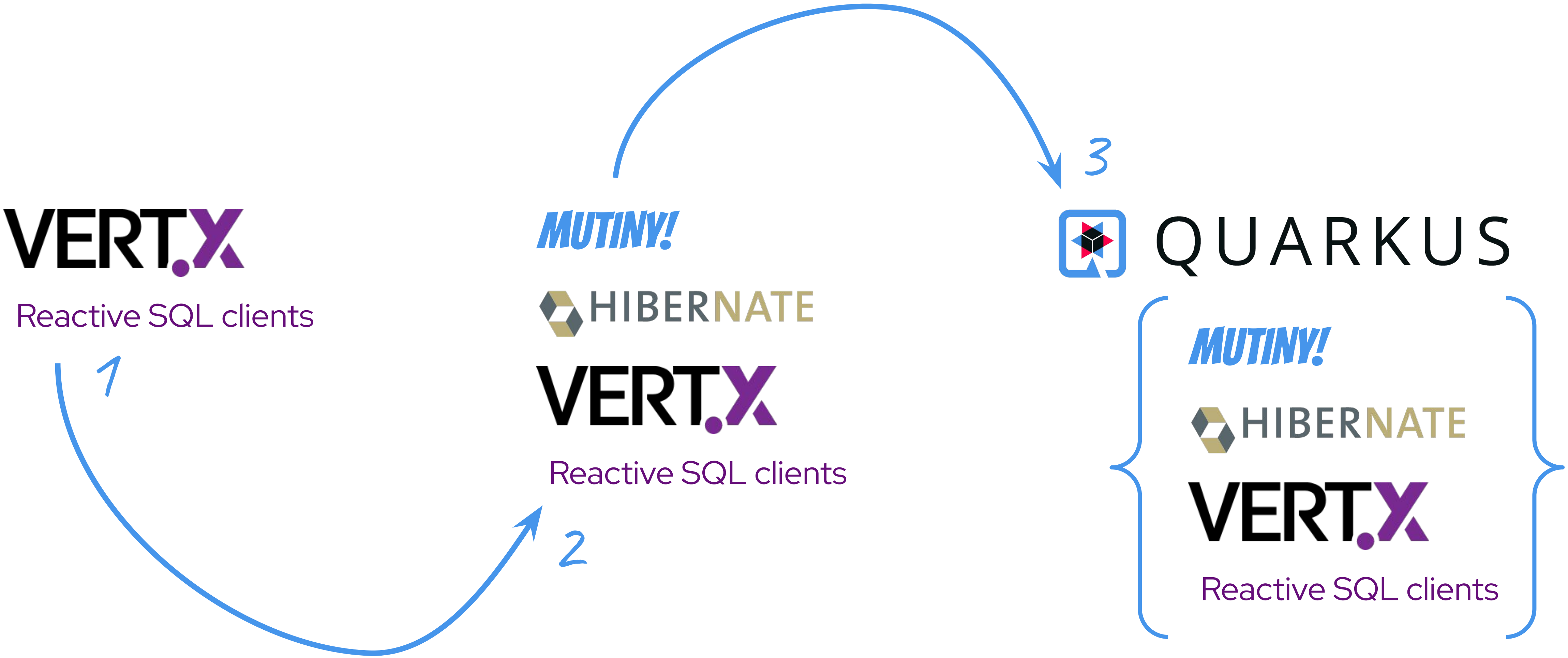
Non-Blocking I/O

Better QoS

Use resources efficiently

1 thread = many connections!

Demos agenda



VERT.X

Reactive SQL clients

MUTINY!

HIBERNATE

VERT.X

Reactive SQL clients

QUARKUS

MUTINY!

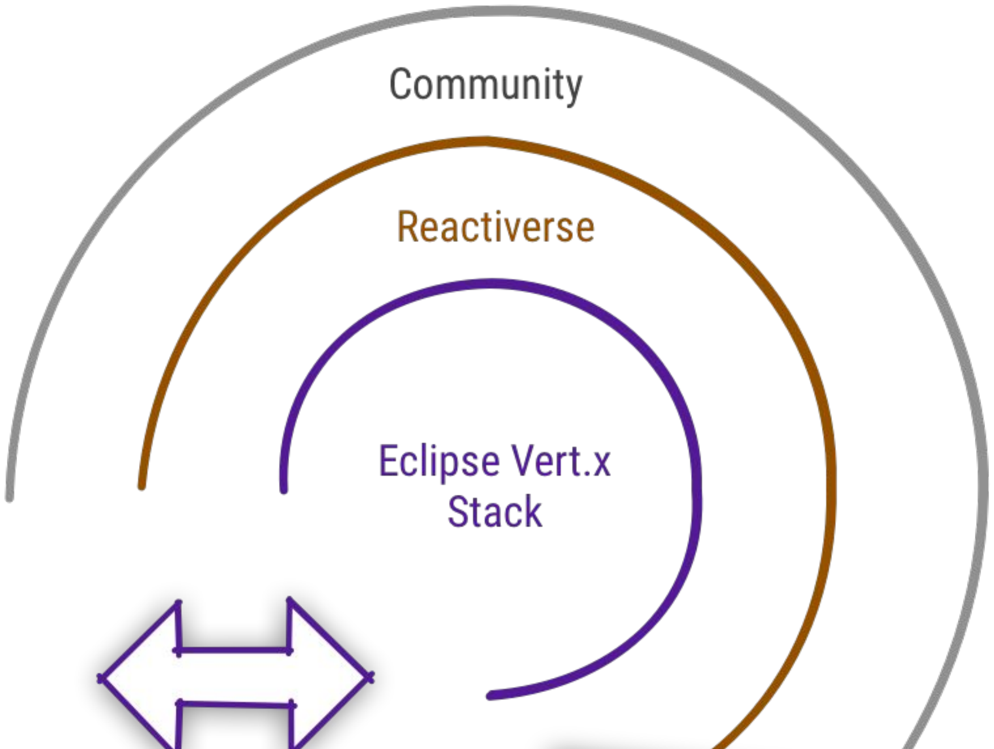
HIBERNATE

VERT.X

Reactive SQL clients

Demo time!

A versatile toolkit!



Reactive database clients

Messaging and event streams

Web APIs and clients

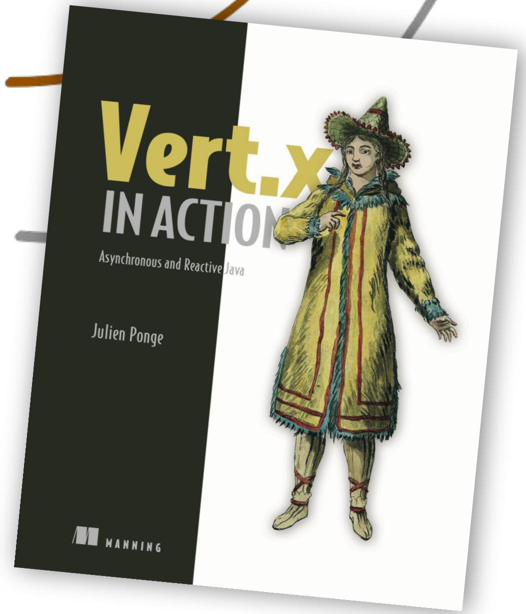
Integration

(...)

Clustering

Micro-services and cloud native

Security and authentication



Taming asynchronous programming?

Future / promises

Reactive extensions

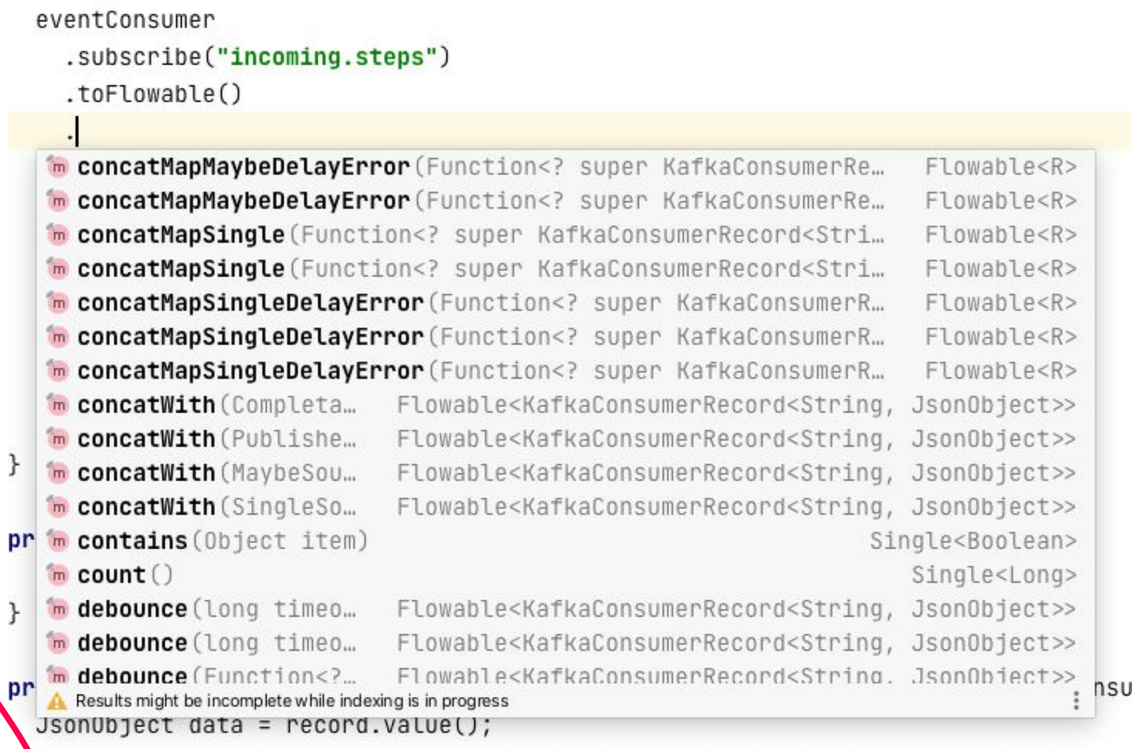
“Coroutines”

...

Why MUTINY!

No more “Monad-hell”!

```java
WebClient webClient = WebClient.create(vertx);
webClient
  .get(3001, "localhost", "/ranking-last-24-hours")
  .as(BodyCodec.jsonArray())
  .rxSend()
  .delay(5, TimeUnit.SECONDS, RxHelper.scheduler(vertx))
  .retry(5)
  .map(HttpResponse::body)
  .flattenAsFlowable(Functions.identity())
  .cast(JsonObject.class)
  .flatMapSingle(json -> whoOwnsDevice(webClient, json))
  .flatMapSingle(json -> fillWithUserProfile(webClient, json))
  .subscribe(
    this::hydrateEntryIfPublic,
    err -> logger.error("Hydratation error", err),
    () -> logger.info("Hydratation completed"));
```



```
eventConsumer
  .subscribe("incoming.steps")
  .toFlowable()
  .|
m concatMapMaybeDelayError(Function<? super KafkaConsumerRe…   Flowable<R>
m concatMapMaybeDelayError(Function<? super KafkaConsumerRe…   Flowable<R>
m concatMapSingle(Function<? super KafkaConsumerRecord<Stri…   Flowable<R>
m concatMapSingle(Function<? super KafkaConsumerRecord<Stri…   Flowable<R>
m concatMapSingleDelayError(Function<? super KafkaConsumerR…   Flowable<R>
m concatMapSingleDelayError(Function<? super KafkaConsumerR…   Flowable<R>
m concatMapSingleDelayError(Function<? super KafkaConsumerR…   Flowable<R>
m concatWith(Completa…   Flowable<KafkaConsumerRecord<String, JsonObject>>
m concatWith(Publishe…   Flowable<KafkaConsumerRecord<String, JsonObject>>
m concatWith(MaybeSou…   Flowable<KafkaConsumerRecord<String, JsonObject>>
m concatWith(SingleSo…   Flowable<KafkaConsumerRecord<String, JsonObject>>
m contains(Object item)                                  Single<Boolean>
m count()                                                   Single<Long>
m debounce(long timeo…   Flowable<KafkaConsumerRecord<String, JsonObject>>
m debounce(long timeo…   Flowable<KafkaConsumerRecord<String, JsonObject>>
m debounce(Function<?…   Flowable<KafkaConsumerRecord<String, JsonObject>>
⚠ Results might be incomplete while indexing is in progress
JsonObject data = record.value();
```

We need navigable APIs!

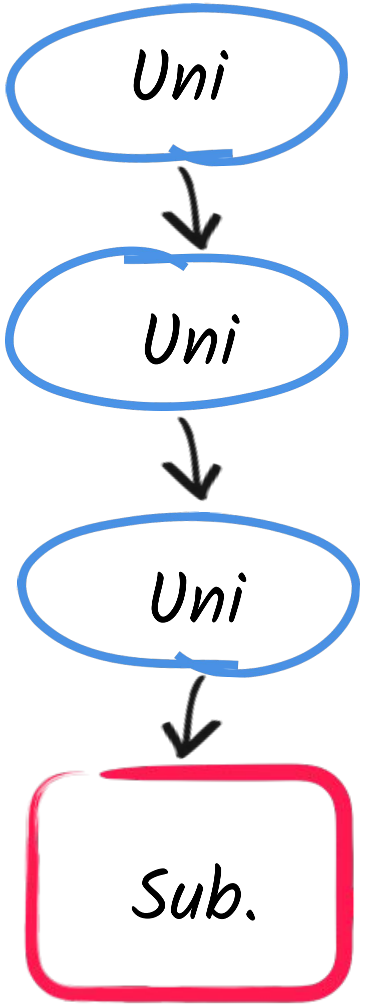
MUTINY!

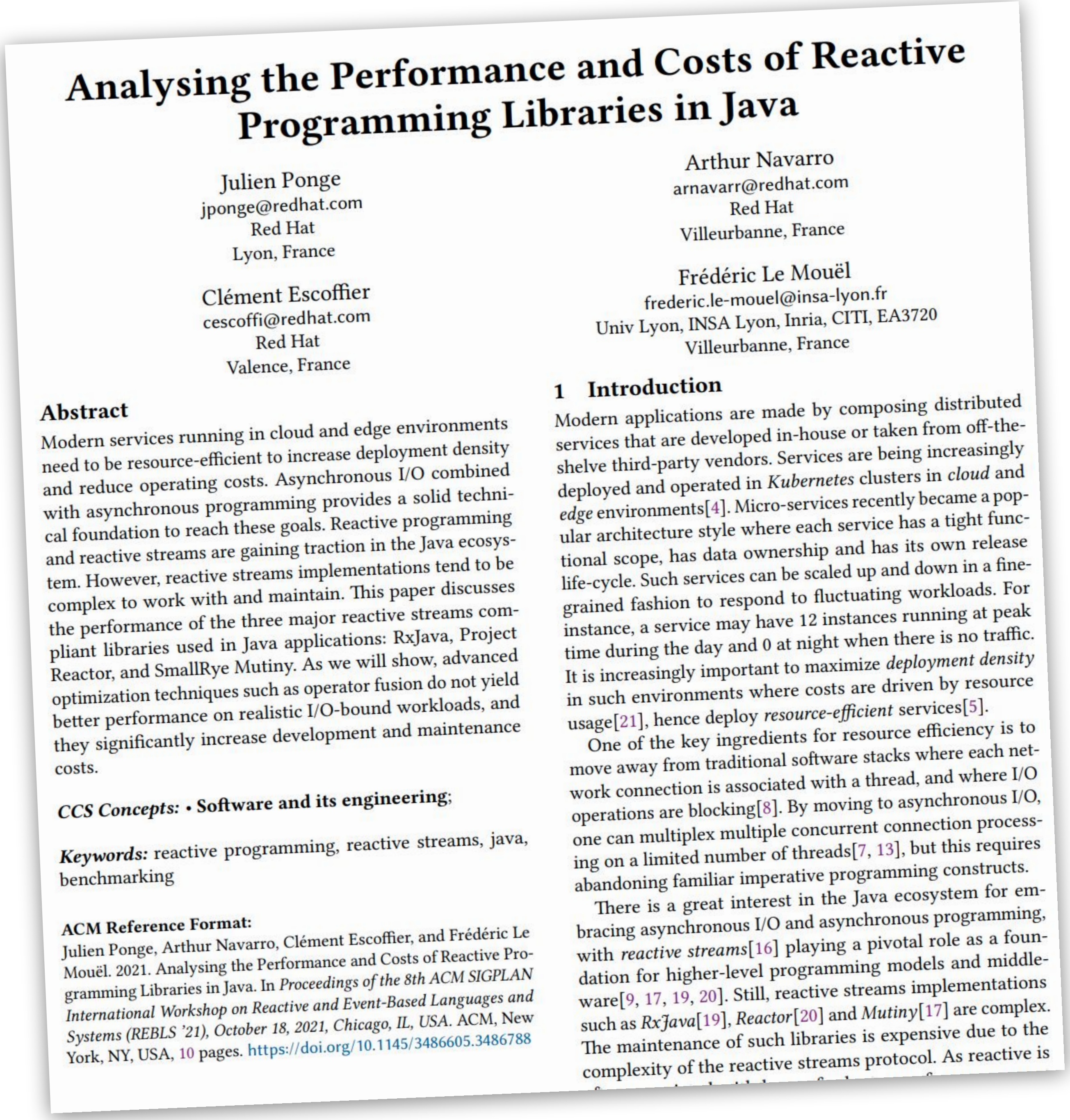
```
service.order(order)

   .onItem().transform(i -> process(i))

   .onFailure().recoverWithItem(fallback)

   .subscribe().with(
       item -> ...
);
```





Analysing the Performance and Costs of Reactive Programming Libraries in Java

Julien Ponge
jponge@redhat.com
Red Hat
Lyon, France

Arthur Navarro
arnavarr@redhat.com
Red Hat
Villeurbanne, France

Clément Escoffier
cescoffi@redhat.com
Red Hat
Valence, France

Frédéric Le Mouël
frederic.le-mouel@insa-lyon.fr
Univ Lyon, INSA Lyon, Inria, CITI, EA3720
Villeurbanne, France

Abstract

Modern services running in cloud and edge environments need to be resource-efficient to increase deployment density and reduce operating costs. Asynchronous I/O combined with asynchronous programming provides a solid technical foundation to reach these goals. Reactive programming and reactive streams are gaining traction in the Java ecosystem. However, reactive streams implementations tend to be complex to work with and maintain. This paper discusses the performance of the three major reactive streams compliant libraries used in Java applications: RxJava, Project Reactor, and SmallRye Mutiny. As we will show, advanced optimization techniques such as operator fusion do not yield better performance on realistic I/O-bound workloads, and they significantly increase development and maintenance costs.

CCS Concepts: • **Software and its engineering**;

Keywords: reactive programming, reactive streams, java, benchmarking

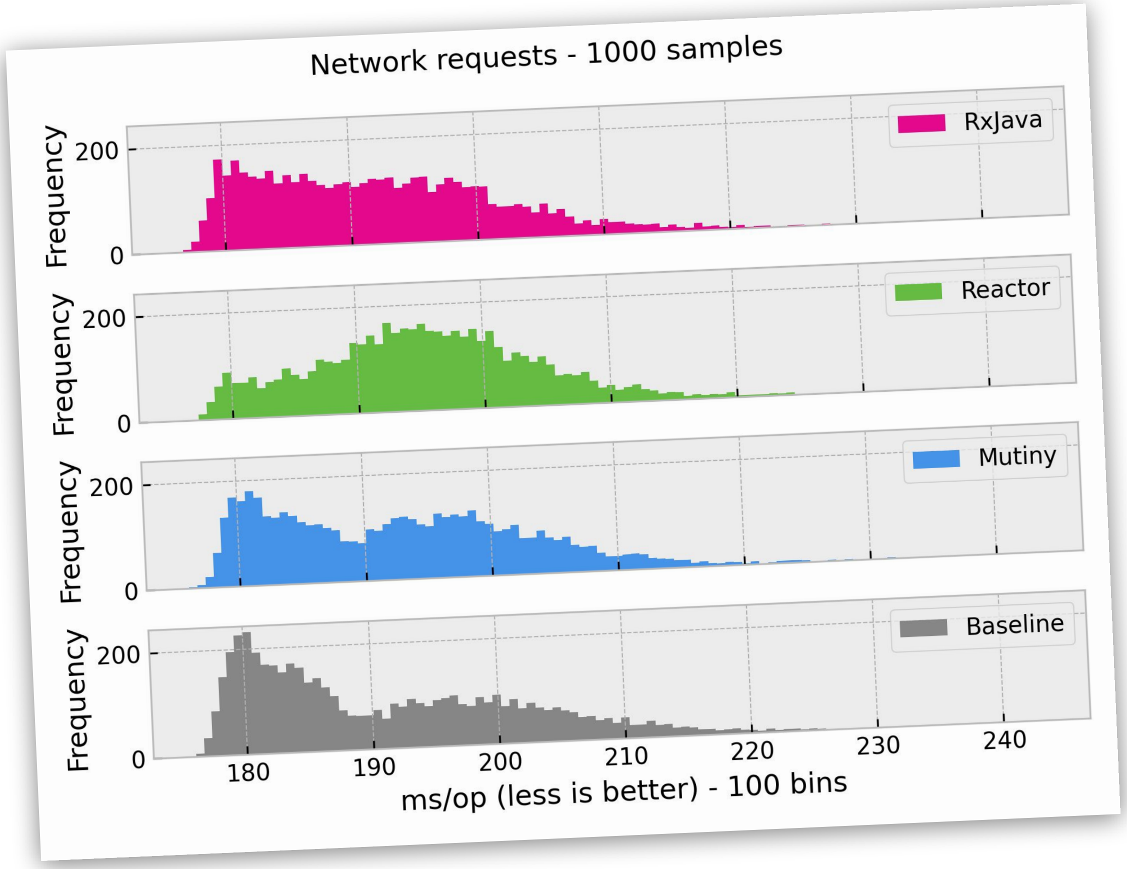
ACM Reference Format:
Julien Ponge, Arthur Navarro, Clément Escoffier, and Frédéric Le Mouël. 2021. Analysing the Performance and Costs of Reactive Programming Libraries in Java. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS '21), October 18, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3486605.3486788

1 Introduction

Modern applications are made by composing distributed services that are developed in-house or taken from off-the-shelve third-party vendors. Services are being increasingly deployed and operated in *Kubernetes* clusters in *cloud* and *edge* environments[4]. Micro-services recently became a popular architecture style where each service has a tight functional scope, has data ownership and has its own release life-cycle. Such services can be scaled up and down in a fine-grained fashion to respond to fluctuating workloads. For instance, a service may have 12 instances running at peak time during the day and 0 at night when there is no traffic. It is increasingly important to maximize *deployment density* in such environments where costs are driven by resource usage[21], hence deploy *resource-efficient* services[5].

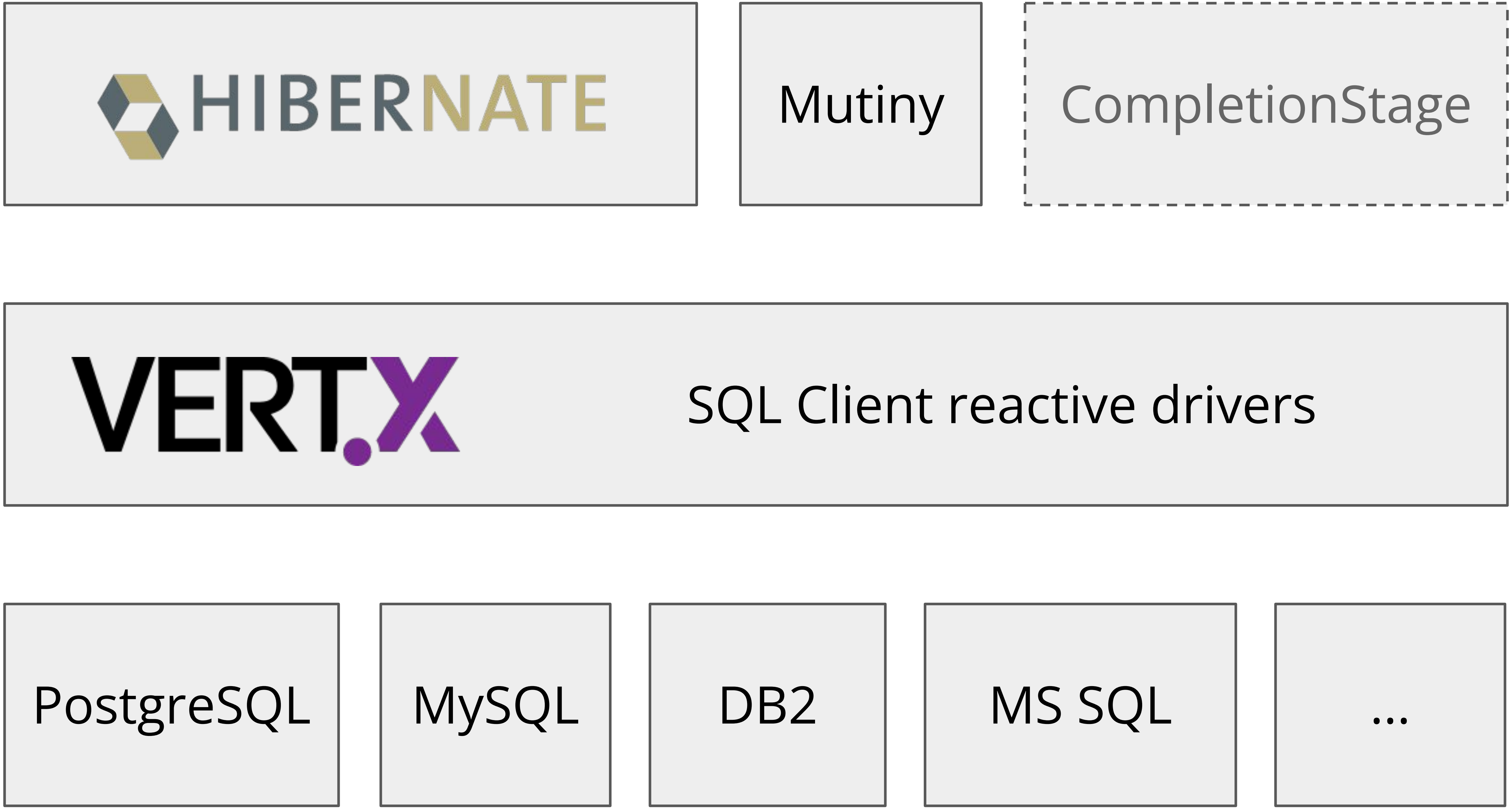
One of the key ingredients for resource efficiency is to move away from traditional software stacks where each network connection is associated with a thread, and where I/O operations are blocking[8]. By moving to asynchronous I/O, one can multiplex multiple concurrent connection processing on a limited number of threads[7, 13], but this requires abandoning familiar imperative programming constructs.

There is a great interest in the Java ecosystem for embracing asynchronous I/O and asynchronous programming, with *reactive streams*[16] playing a pivotal role as a foundation for higher-level programming models and middleware[9, 17, 19, 20]. Still, reactive streams implementations such as *RxJava*[19], *Reactor*[20] and *Mutiny*[17] are complex. The maintenance of such libraries is expensive due to the complexity of the reactive streams protocol. As reactive is



Demo time!

Hibernate goes reactive!



HIBERNATE

Mutiny

CompletionStage

VERT.X

SQL Client reactive drivers

PostgreSQL

MySQL

DB2

MS SQL

...

Demo time!

QUARKUS

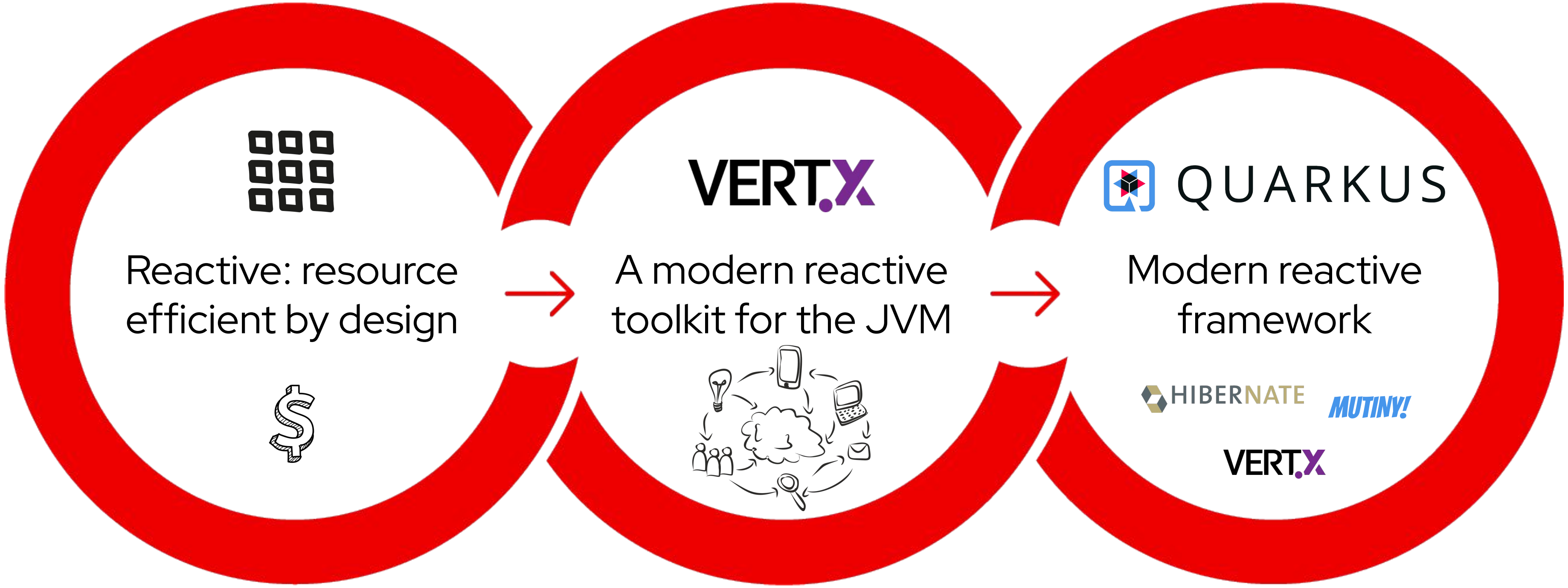
A stack to write ~~Java~~ Reactive apps

Cloud Native, Microservices, Serverless

Demo time!

A cohesive and comprehensive ecosystem

A reactive continuum built by Red Hat



Reactive: resource efficient by design

A modern reactive toolkit for the JVM

Modern reactive framework

Q&A

“Simply reactive”
Vert.x, Mutiny, Hibernate Reactive and Quarkus

Julien Ponge

in linkedin.com/company/red-hat

▶ youtube.com/user/RedHatVideos

f facebook.com/redhatinc

twitter.com/RedHat