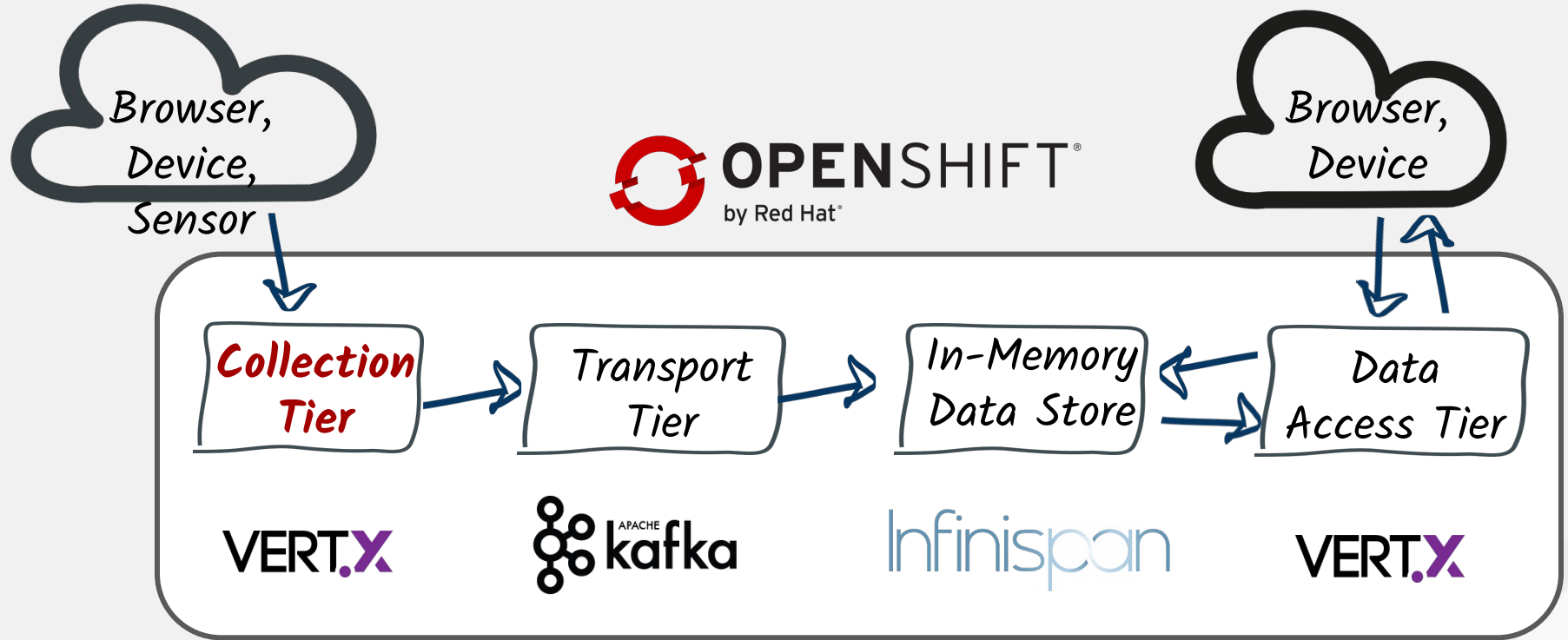




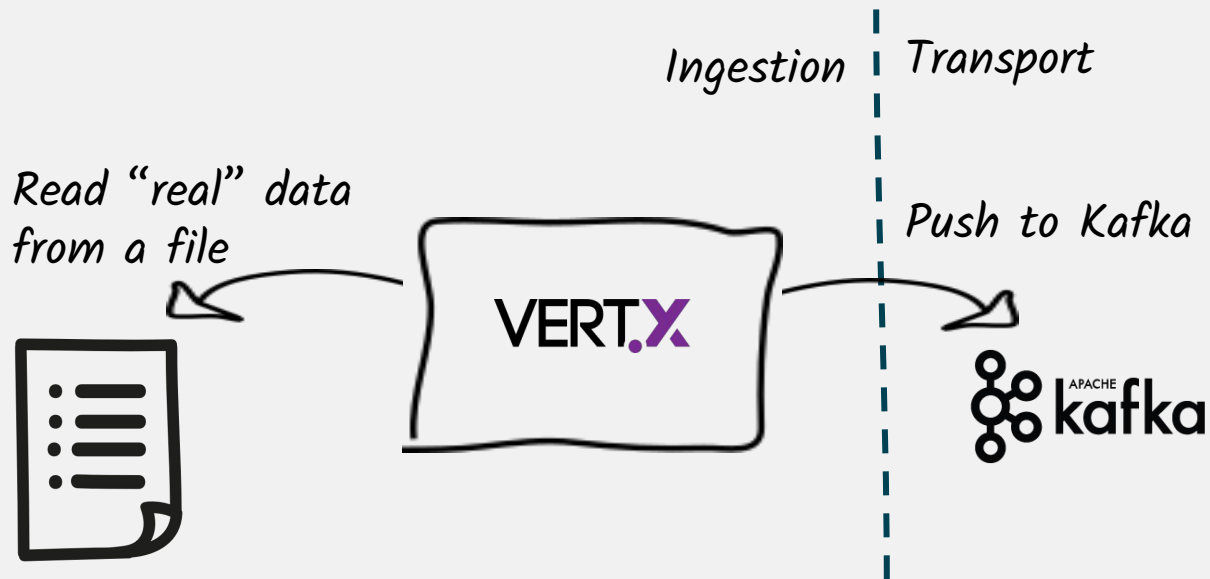
**red**hat.

Ingesting data

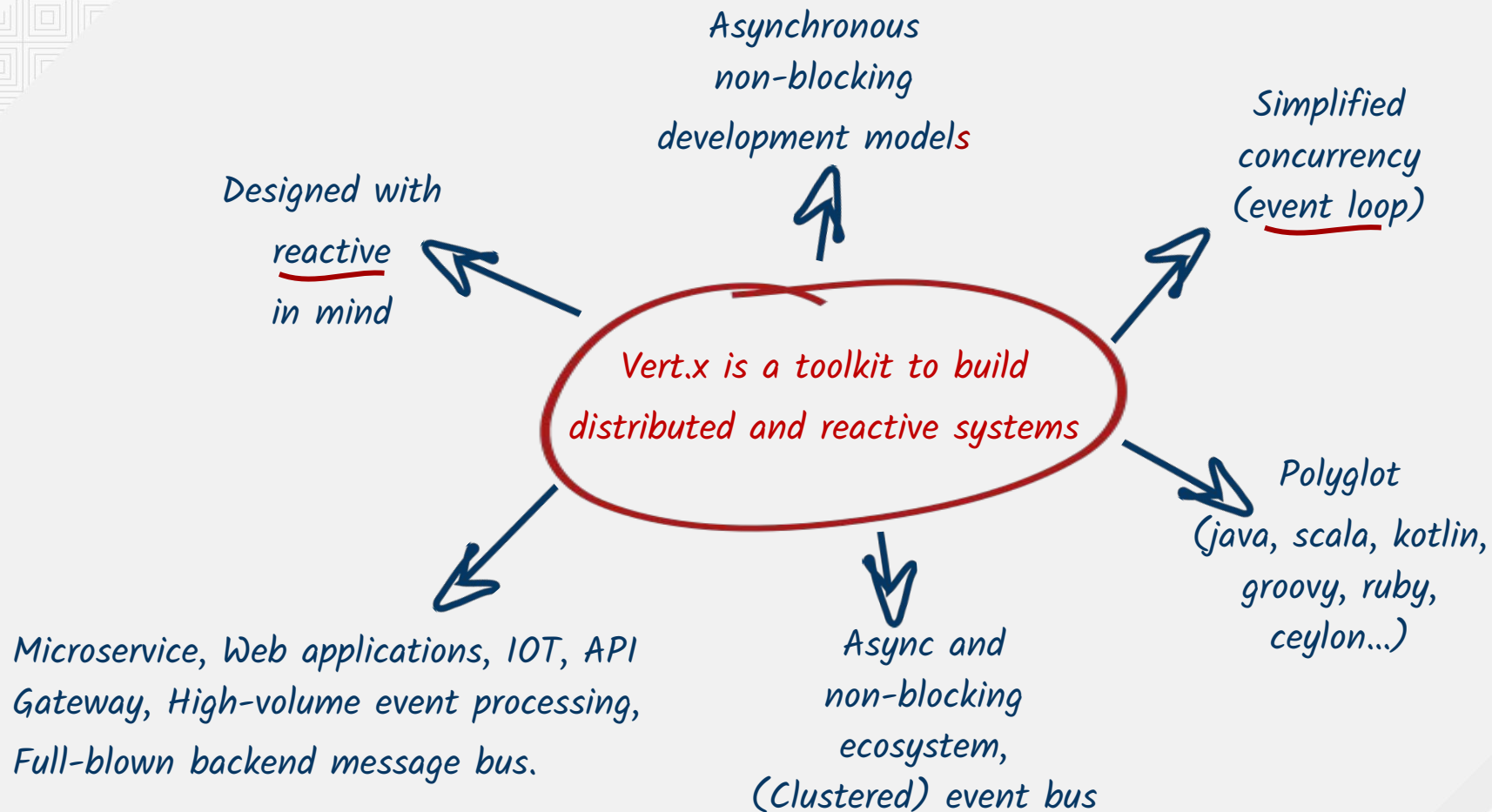
# Streaming Data - Deep Dive



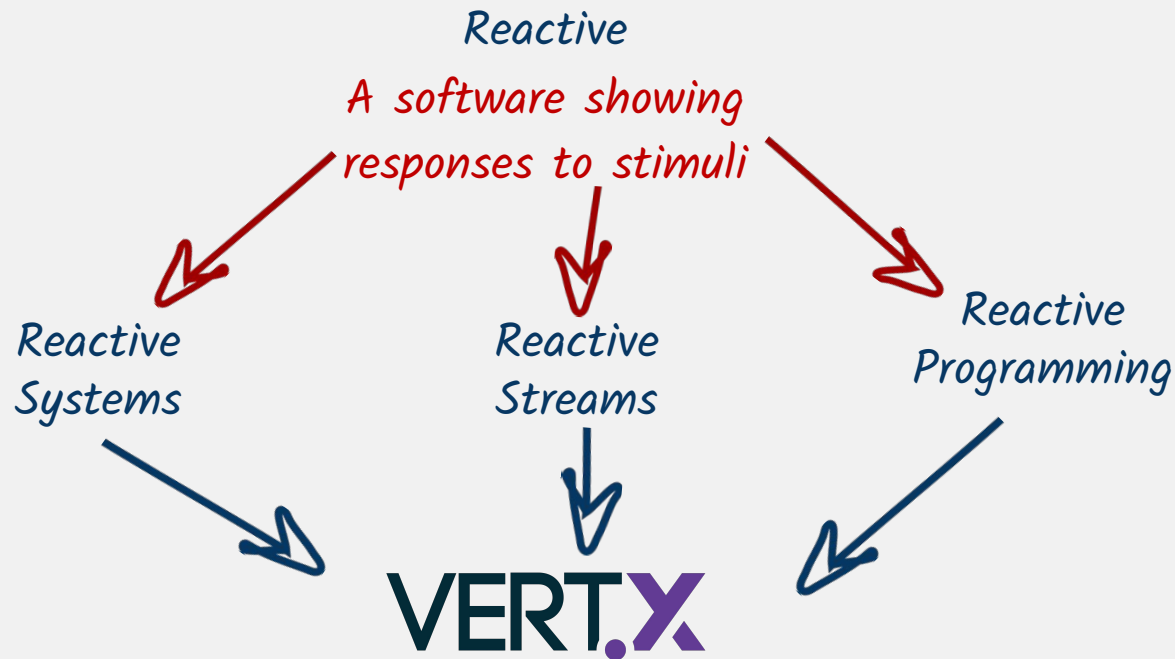
# Ingestion architecture



*Eclipse Vert.x*



# Vert.x - the all-in-one toolkit

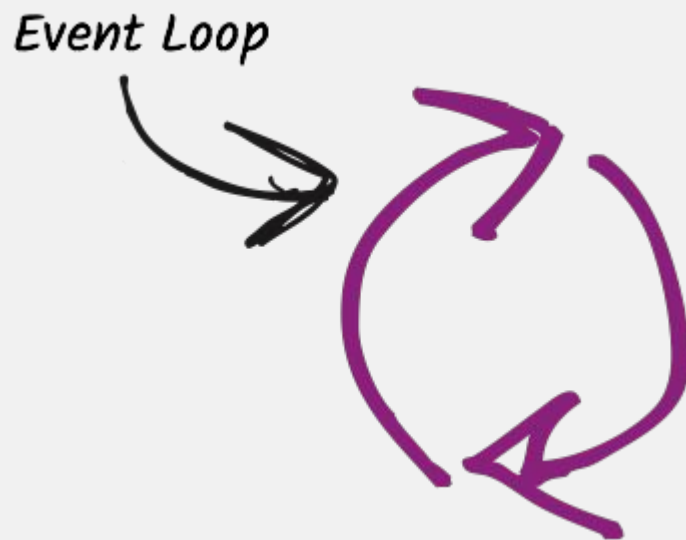


# Shape up your system as you want...



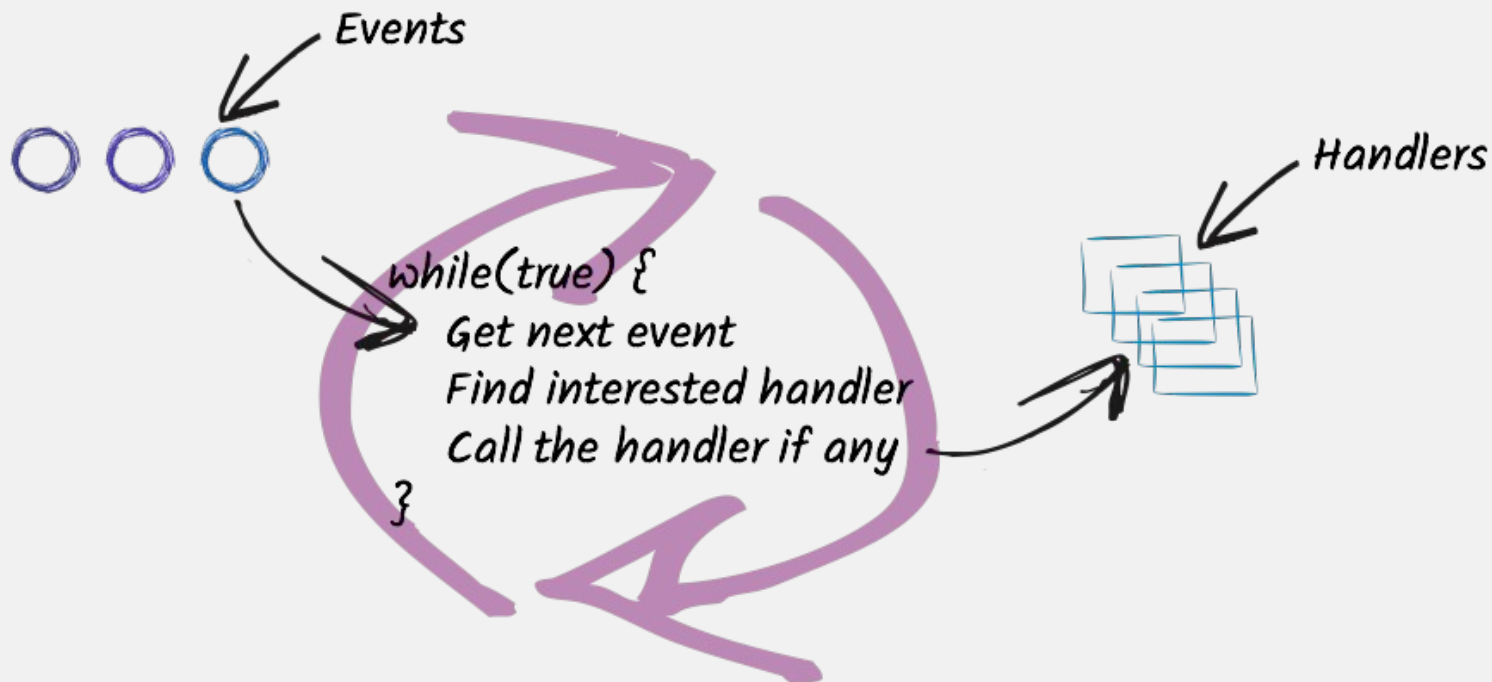
*Don't let frameworks lead, you are back in charge*

# Event loop

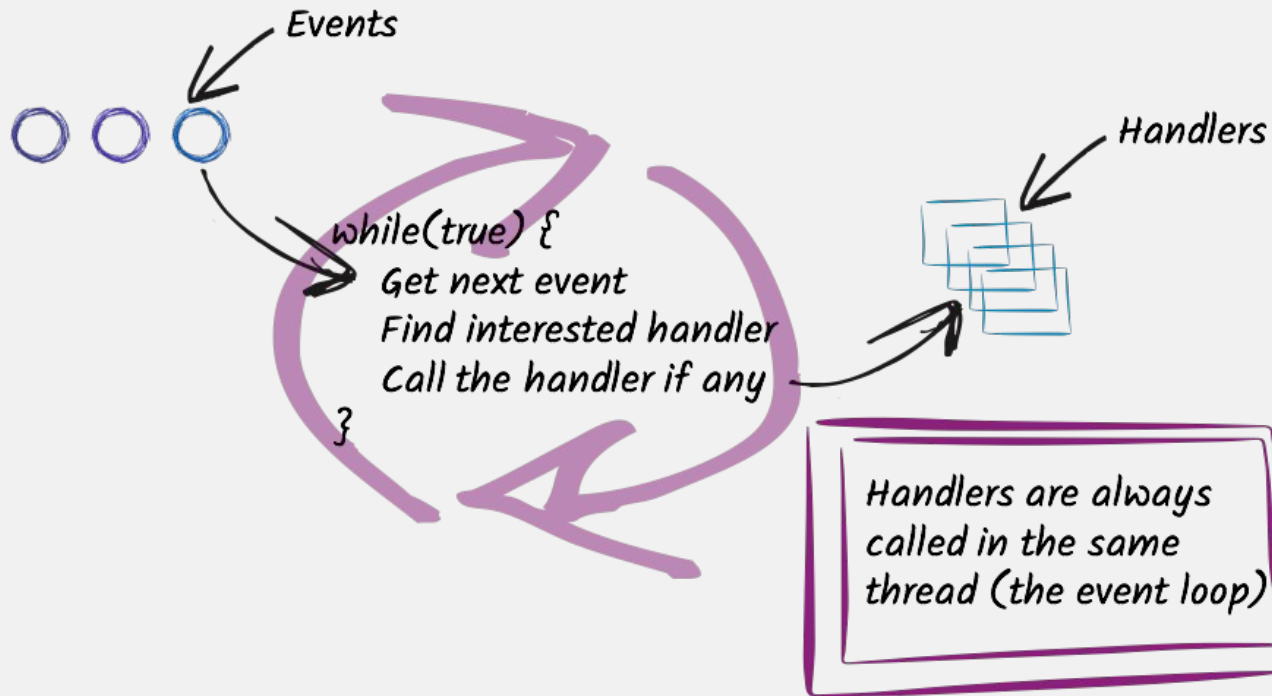




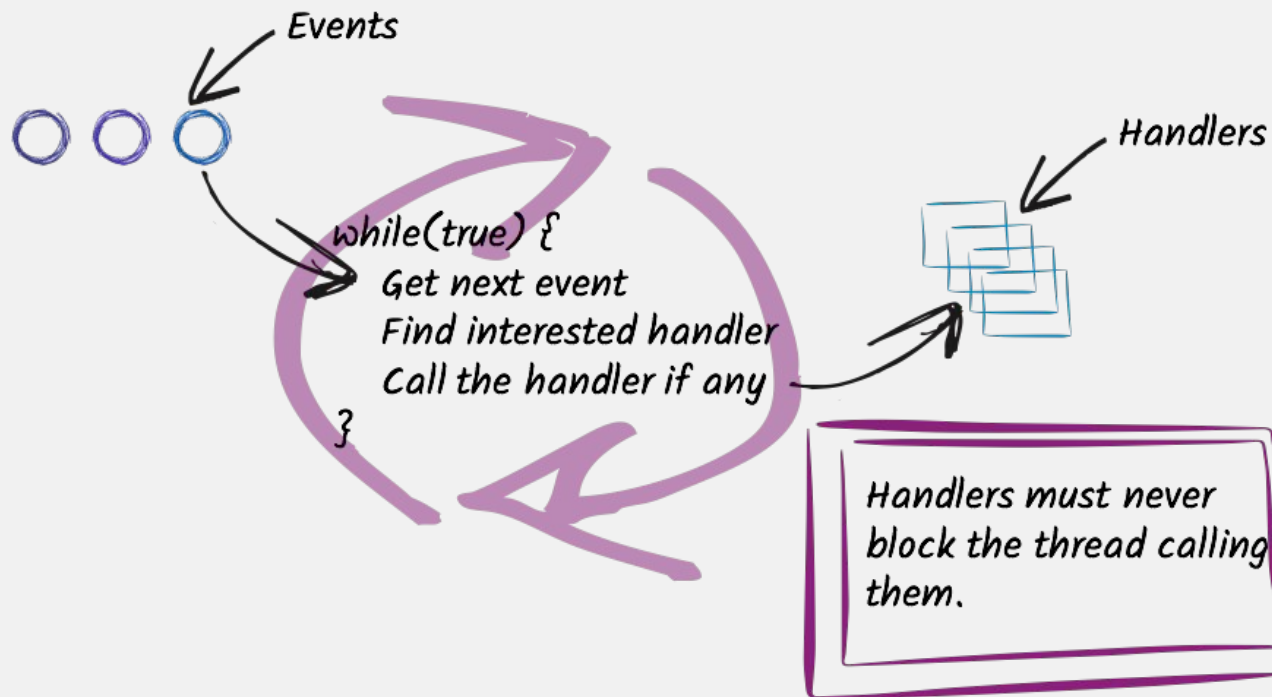
# Event loop



# Event loop



# Event loop



# Callbacks for notifications and async actions

## *Web Server Hello World*

```
vertx.createHttpServer()  
  .requestHandler(req -> // Async reaction  
    req.response().end("Reactive greetings")  
  )  
  .listen(8080, ar -> { // Async operation  
    //...  
  });
```

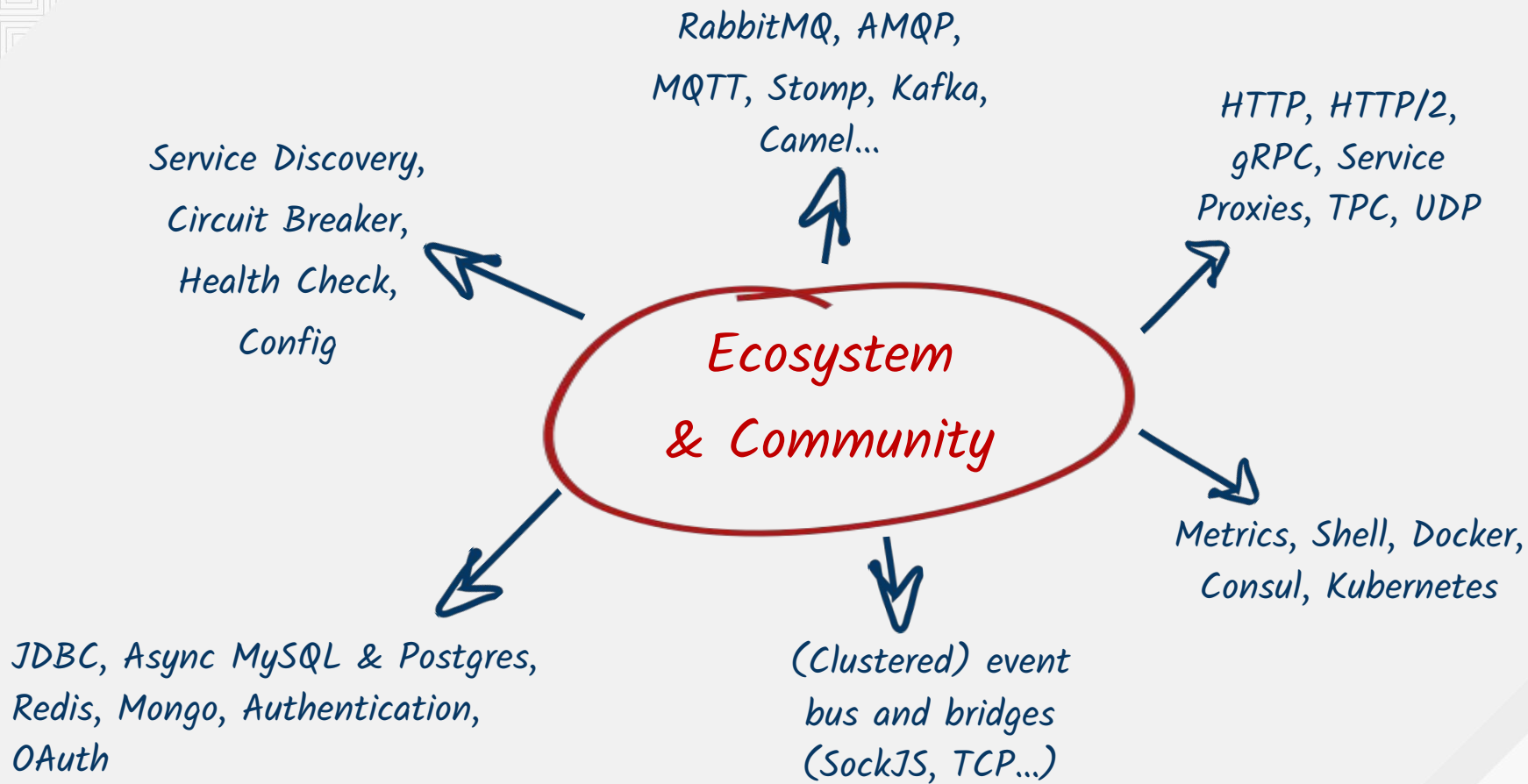
# Programming models

## *Callbacks*

```
void call_service(int a,  
    Handler<AsyncResult<Integer>> h) {  
    // ...  
}  
  
call_service(1, ar -> {  
    int res = ar.result();  
});
```

## *RX*

```
Single<Integer> call_service(int a) {  
    // ...  
}  
  
Single<Integer> single =  
    call_service(1, 2);  
single.subscribe(  
    res -> {  
  
    }  
);
```



# *Vertices*

# Verticles

*Verticles are a simple, scalable, actor-like deployment and concurrency model.*

- chunks of code that get deployed and run by Vert.x.
- applications are typically composed of many verticle instances



# Polyglot

Bare java -> class extending `io.vertx.core.AbstractVerticle`

RX Java 2 -> class extending `io.vertx.reactivex.core.AbstractVerticle`

Groovy -> Script or class extending `io.vertx.core.AbstractVerticle`

...

# Example of Verticle

```
package me.escoffier.example;
import io.vertx.reactivex.core.AbstractVerticle;
public class MyVerticle extends AbstractVerticle {
    @Override
    public void start() throws Exception {
        // the vertx instance is available from the `vertx` field
    }
    @Override
    public void stop() throws Exception {
        // optional
    }
}
```

# Asynchronous start

```
public class MyVerticle extends AbstractVerticle {  
    @Override  
    public void start(Future<Void> startFuture) {  
        vertx.rxDeployVerticle("some.other.verticle")  
            .toCompletable()  
            .subscribe(CompletableHelper.toObserver(startFuture));  
    }  
}
```

# *Vert.x, RX Java and Schedulers*

## Event loop golden rule

**NEVER BLOCK THE EVENT LOOP**

# Need to offload work to *workers*

*Thread pools are naughty beasts*

*Tuning thread pools is dark magic*

*Switching between threads is not that simple*

# Schedulers

*Scheduler => Abstraction for pooling threads and scheduling tasks to be executed by them*

- When is called a subscriber, push the emission
- On which thread is called a subscriber, pushed the emission
- Schedulers do not schedule, they manage workers

# Schedulers

## **Computation** Scheduler

- CPU-bound tasks, bounded pool, Can reduce concurrency

## **IO** Scheduler

- IO tasks, unbounded pool, can OOM

## **New** Scheduler

- new thread every time, Can OOM

## **Vert.x Context** Scheduler

- Use the Vert.x Context (event loop or worker context)



# subscribeOn

*subscribeOn => instruct the source stream on which Scheduler to emit items on*



stream

```
.subscribeOn(Schedulers.computation());
```



# observeOn

*observeOn => switch emissions to a different Scheduler*



*Whatever thread*

`stream`

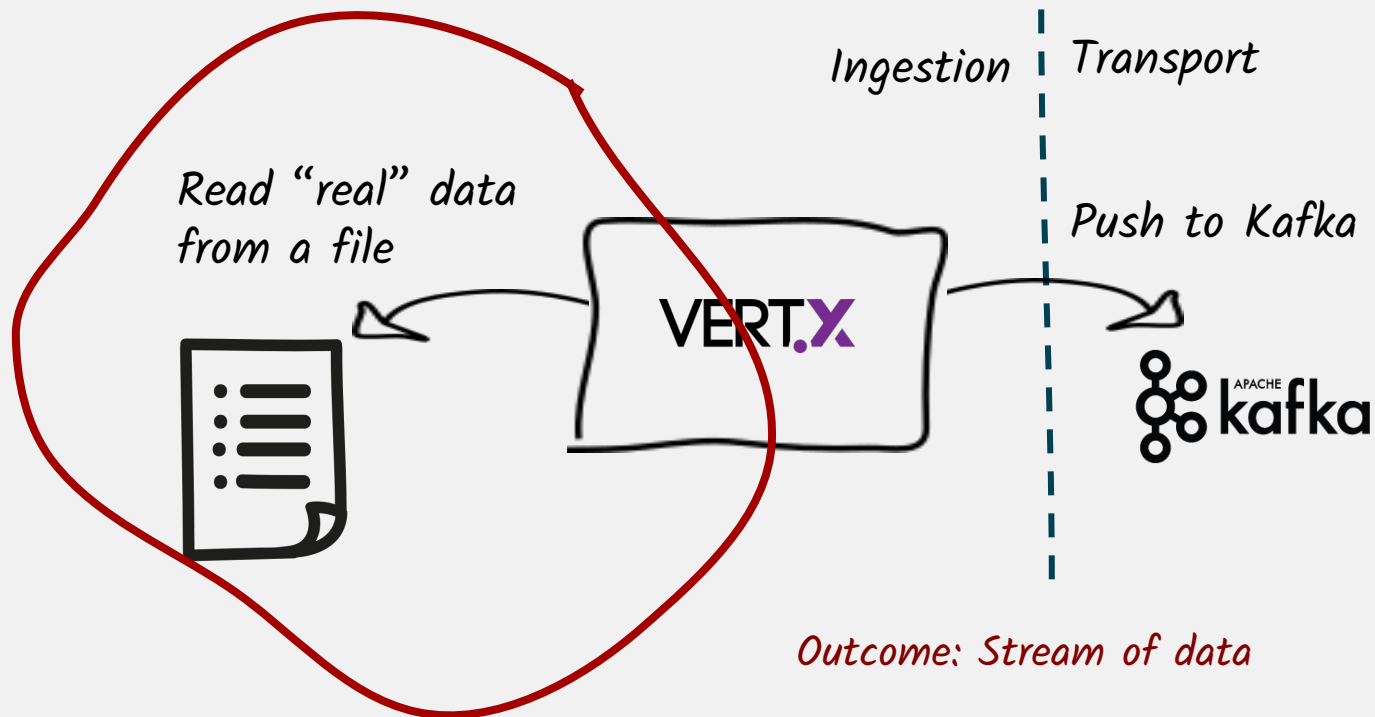
```
.observeOn(Schedulers.computation());
```



*Computation thread*

*Generating a stream: Reading entries*

# Ingestion architecture

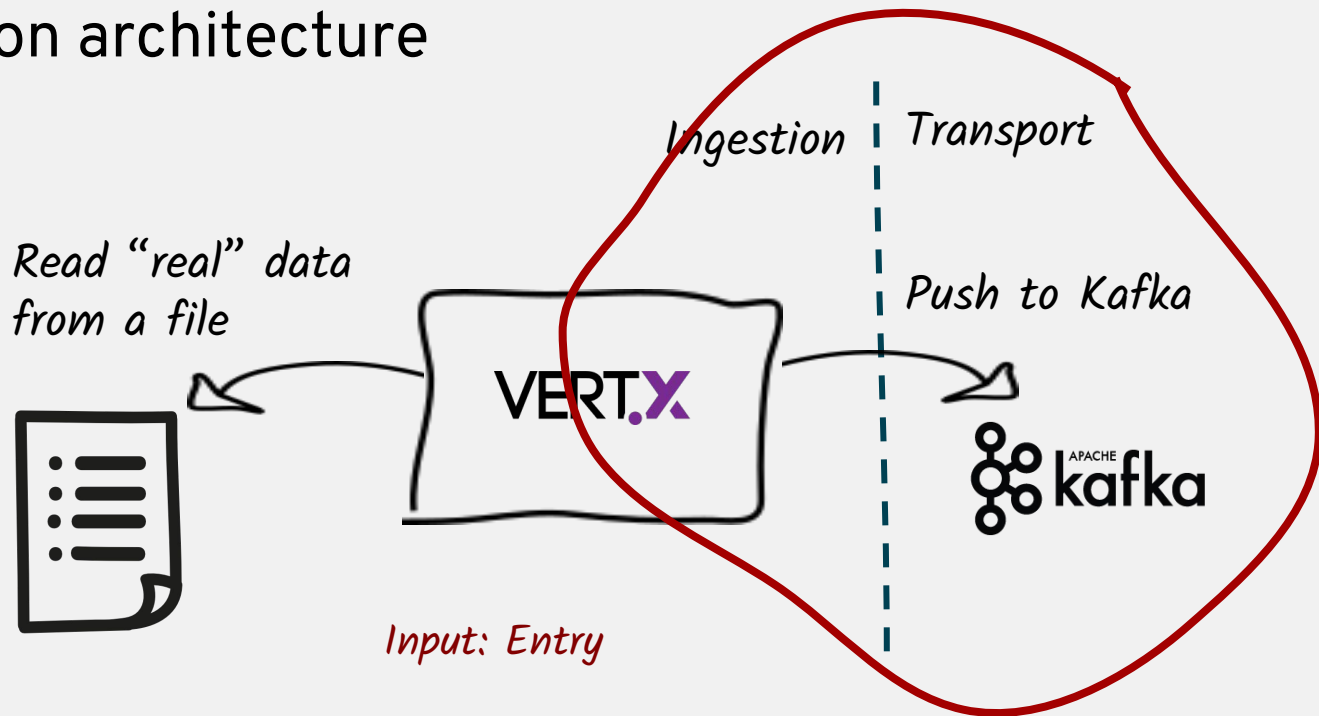


# Generating a stream with back pressure

```
return Flowable.<String, BufferedReader>generate(  
    () -> { // Initialization  
        InputStream gzipStream = // ...  
        Reader decoder = new InputStreamReader(...);  
        return new BufferedReader(decoder);  
    },  
    (bufferedReader, emitter) -> { // Called on consumer requests  
        String line = bufferedReader.readLine();  
        if (line != null) { emitter.onNext(line); } else { emitter.onComplete(); }  
    }, BufferedReader::close // Cleanup  
)  
.subscribeOn(Schedulers.io()); // Execute the generation on the io scheduler
```

*Pushing data to Kafka*

# Ingestion architecture



# Vert.x Kafka Client

*Vert.x Kafka Client provides a reactive way to interact with Apache Kafka*

- Write **record** to a Kafka topic
- Read **records** from Kafka topic

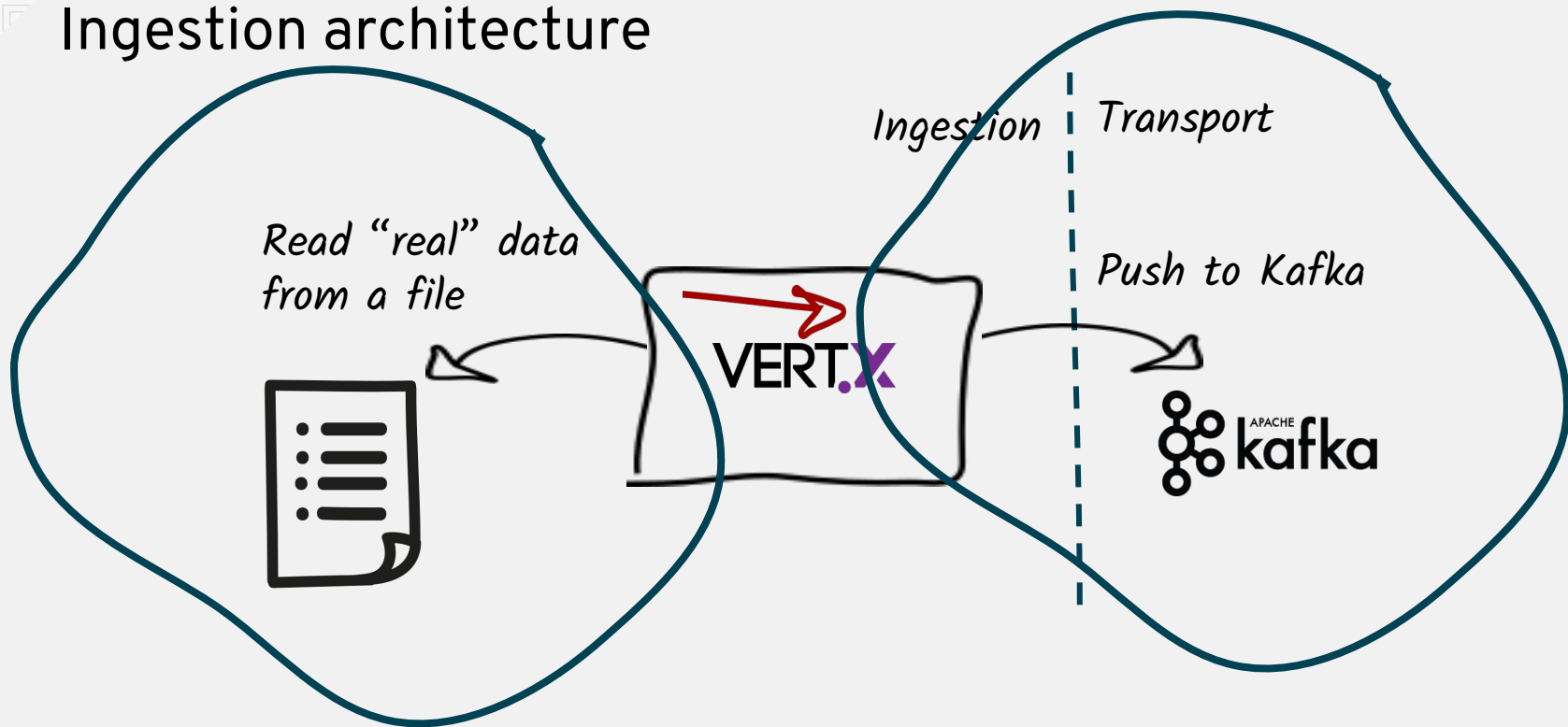


# Kafka Write Stream

```
private Completable dispatch(Map.Entry<String, String> entry) {
    ProducerRecord<String, String> record // ← Kafka Record
    = new ProducerRecord<>(TOPIC, entry.getKey(), entry.getValue());
    return new AsyncResultCompletable( // ← Vert.x async result to RX
        handler ->
            stream.write(record, x -> {
                if (x.succeeded()) {
                    handler.handle(Future.succeededFuture());
                } else {
                    handler.handle(Future.failedFuture(x.cause()));
                }
            }
        ));
}
```

# *From File to Kafka*

# Ingestion architecture



# Inject method

*Read the data from the file*



*Map line to entry*

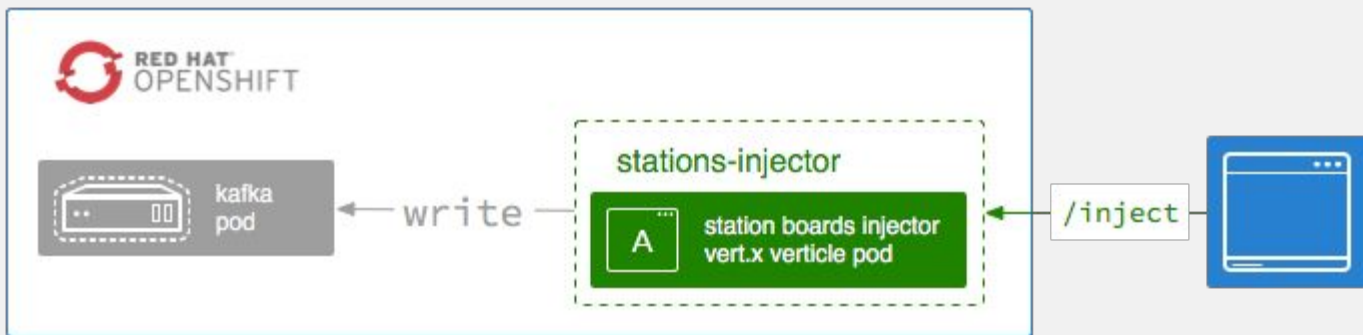


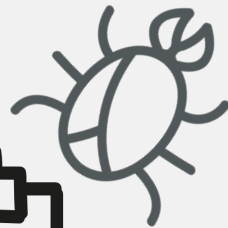
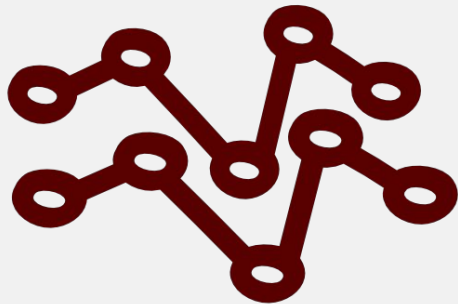
*Send the entry to Kafka*



```
rxReadGunzippedTextResource("...")  
  .map(PositionsInjector::toEntry)  
  .flatMapCompletable(this::dispatch)
```

# Diagram





*DEMO TIME!*

