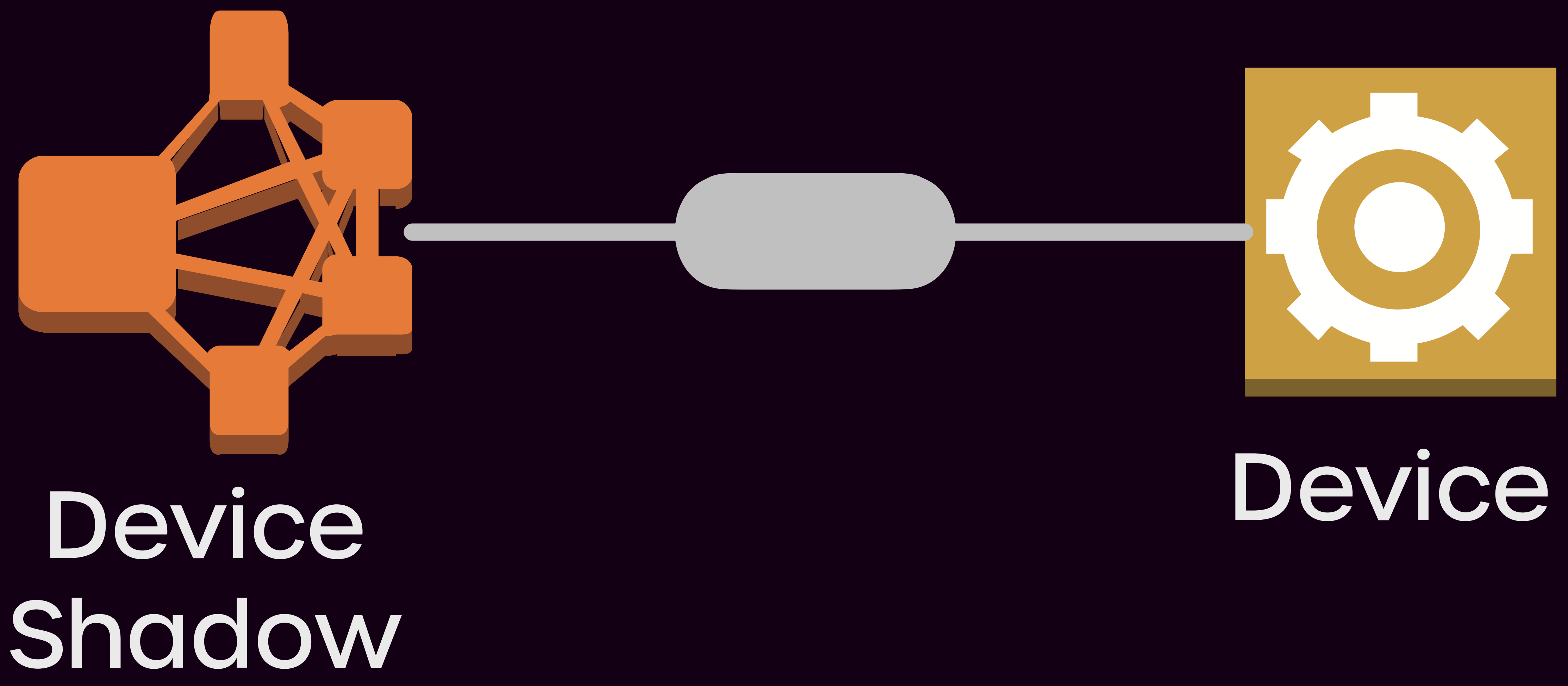
streaming services

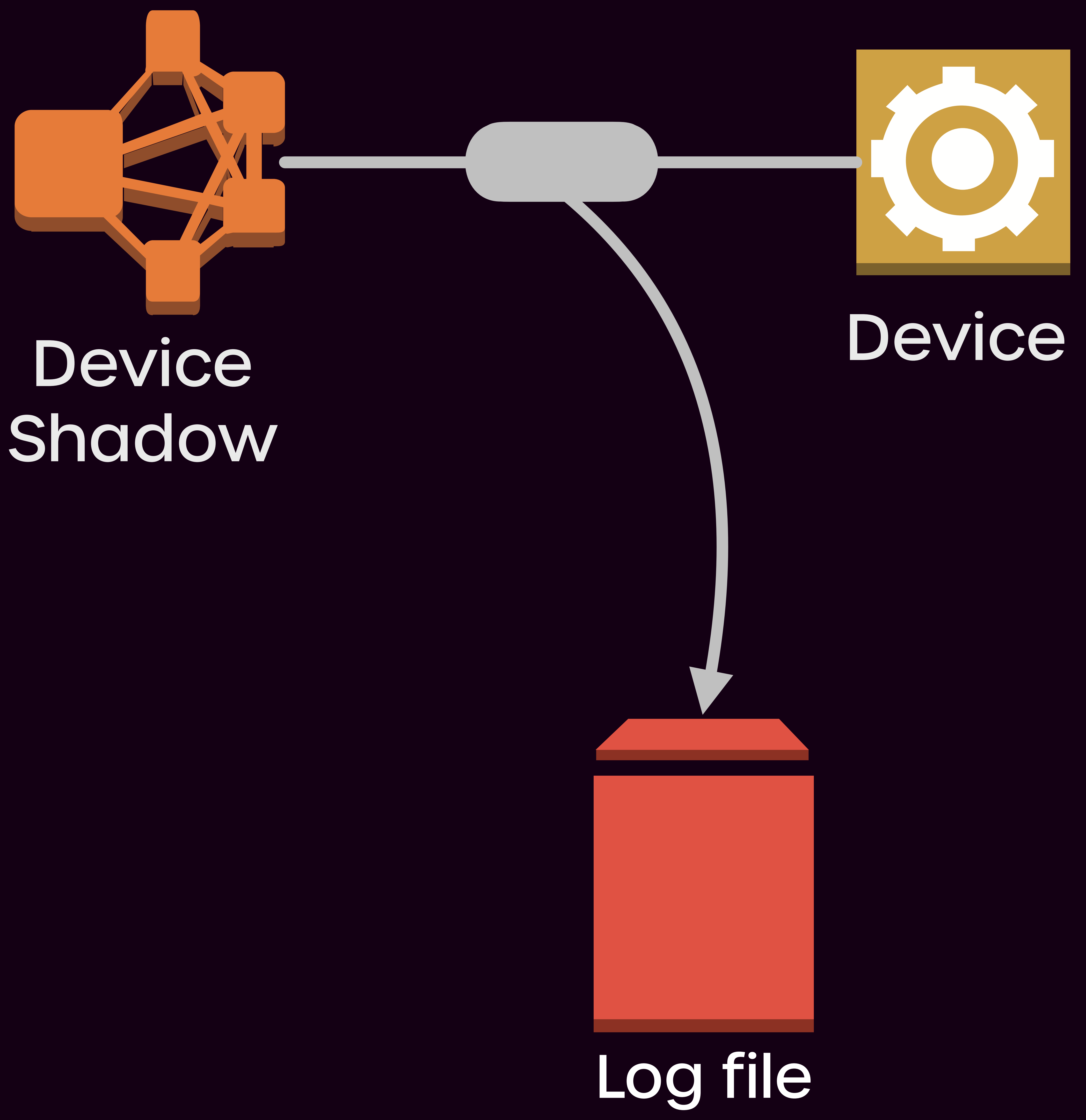
Reactive systems architecture

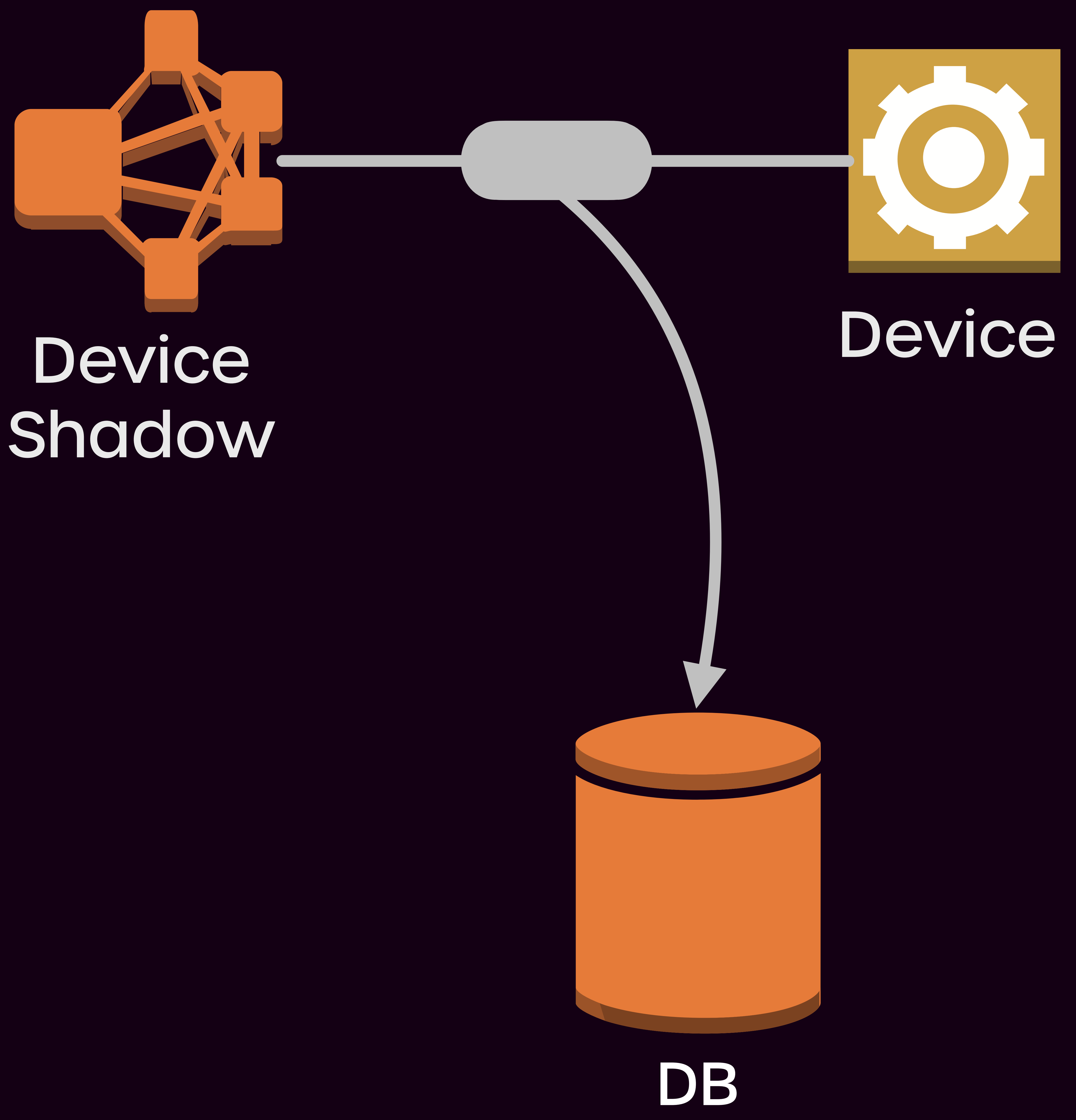
IoT-like media processing

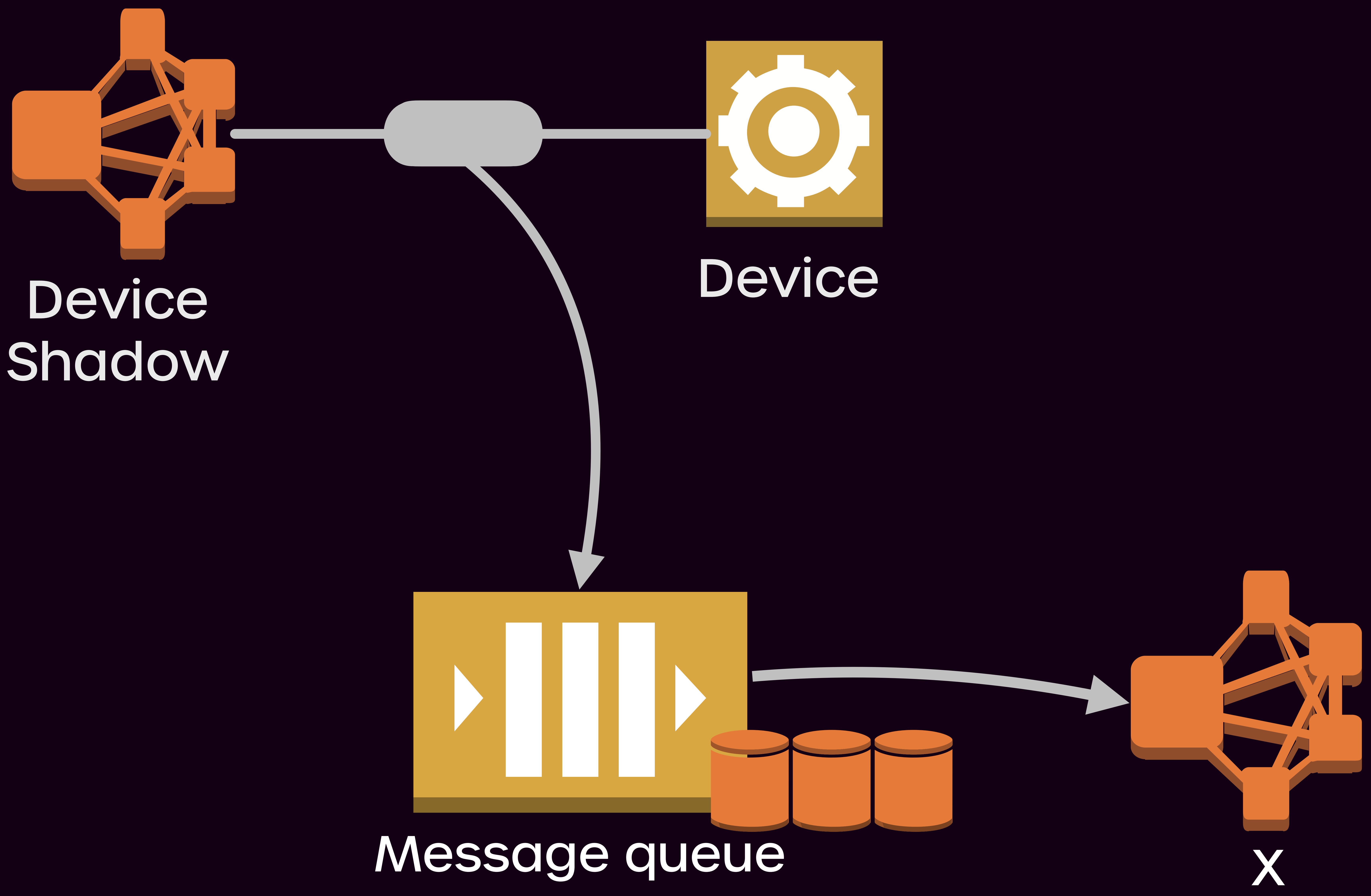
- Operates stream processing devices
- Exposes health-checks and pipeline topology
- Provides a <u>global</u> view of the pipeline

A distributed system without durable messaging easily grows into a monolith


Device Shadow
Device


Device Shadow
Device
Log file


Device
Shadow
Device
DB


Device Shadow
Device
Message queue
X

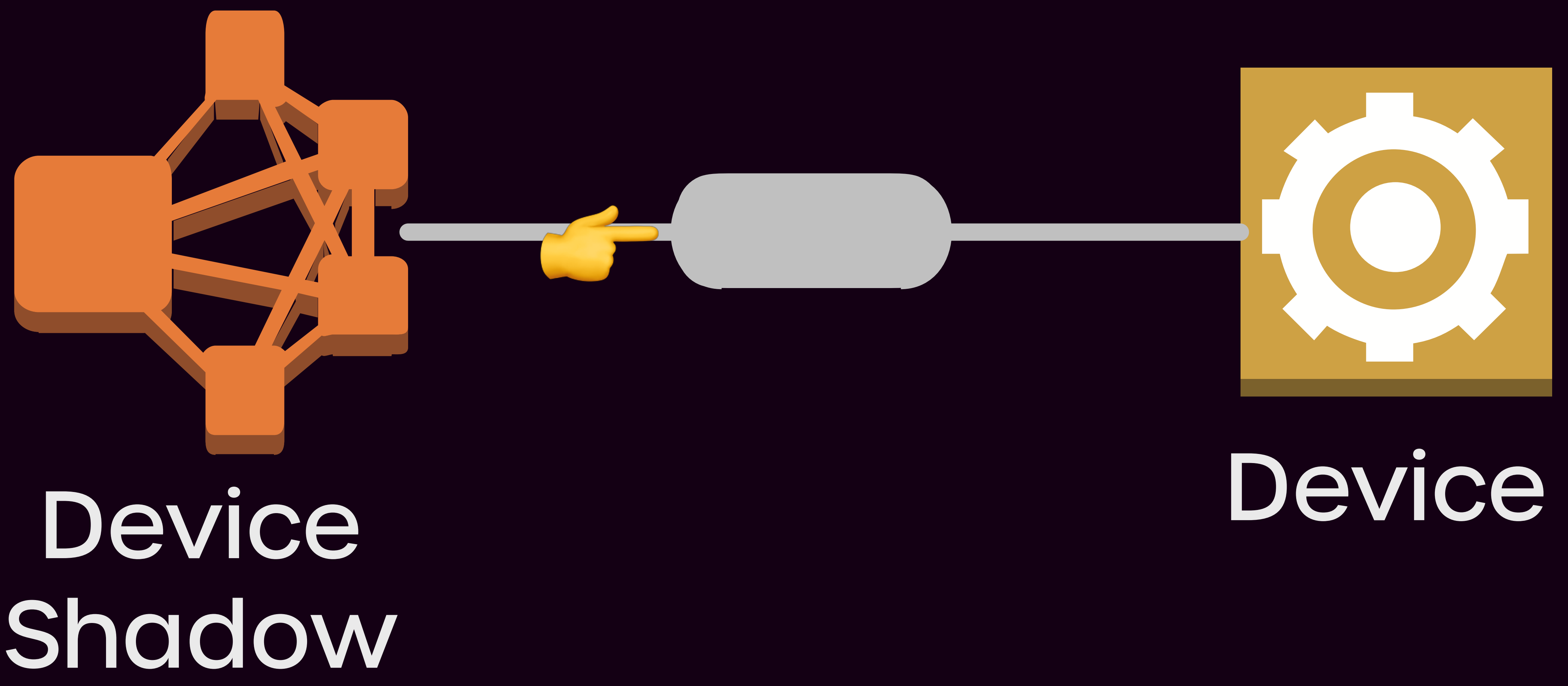
A distributed system without supervision is binary: working or failed

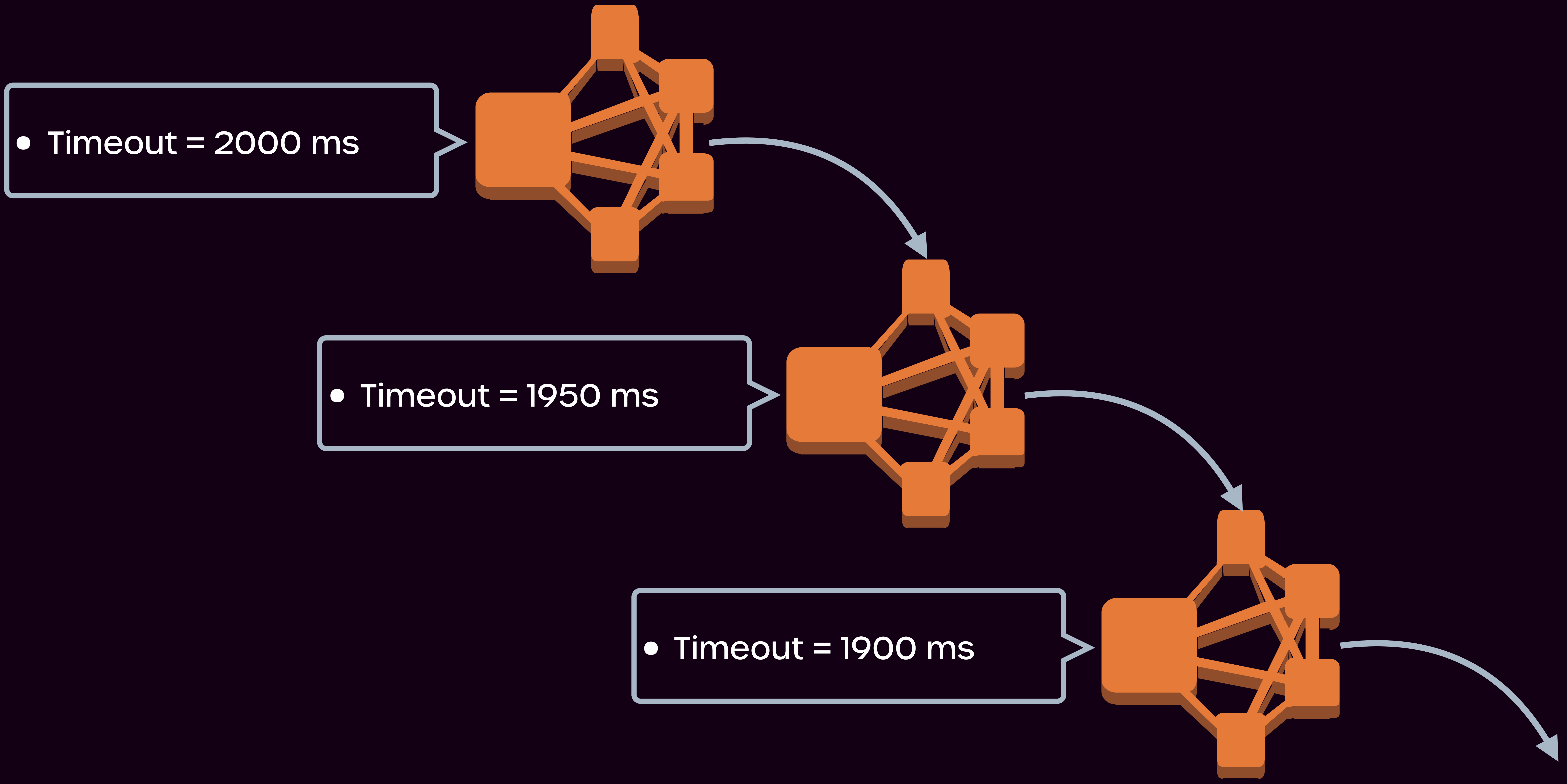
```scala
def makeDeviceRequest(request: DeviceRequest): Future[DeviceResponse] = ???

makeDeviceRequest(request).onComplete {
  case Success(dr) =>
    // working!
  case Failure(ex) =>
    // failed!
    log.error(ex)
    scheduleOnce(1000L, makeDeviceRequest(request))
}
```

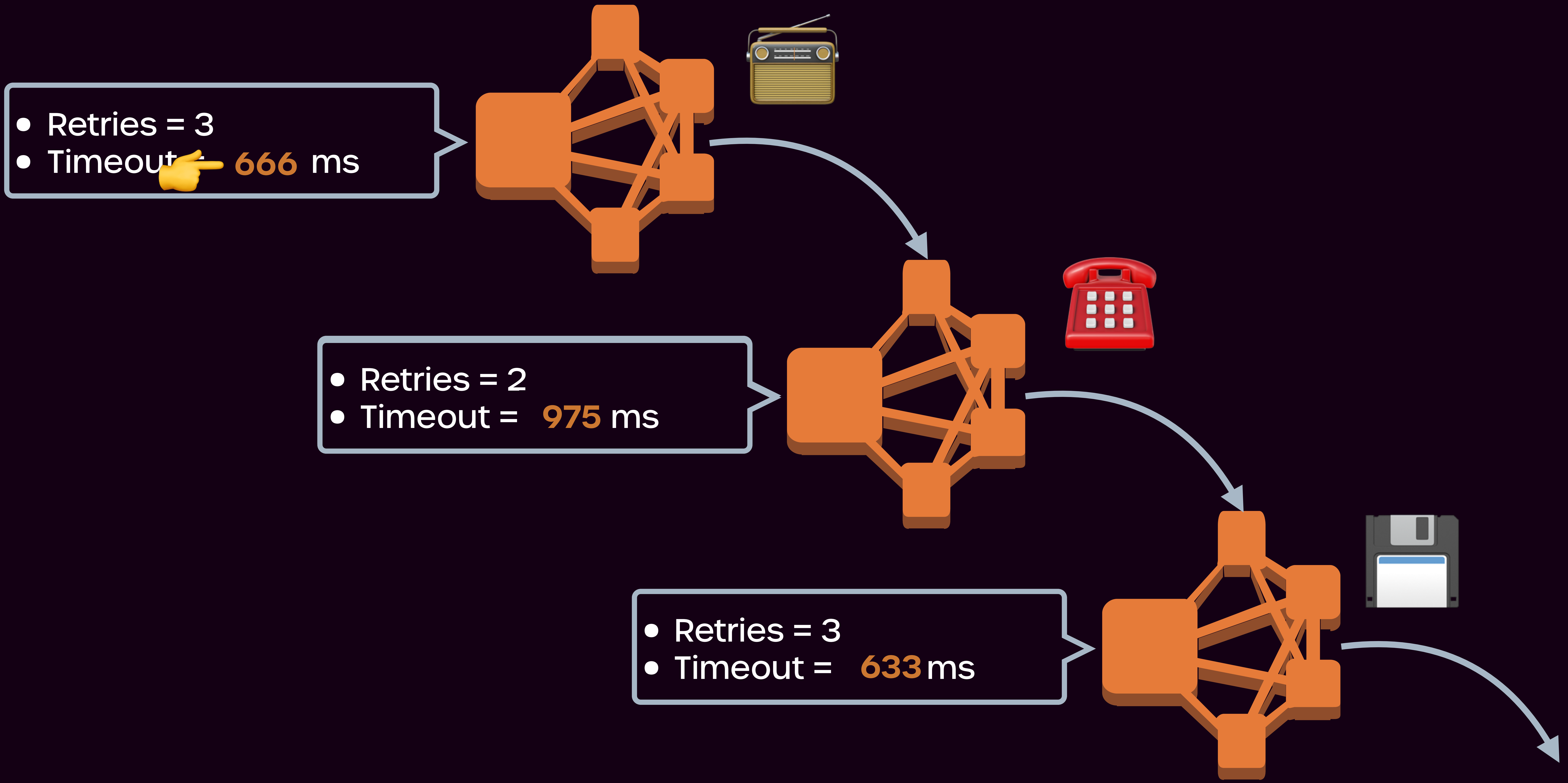
Naïve timeouts cause

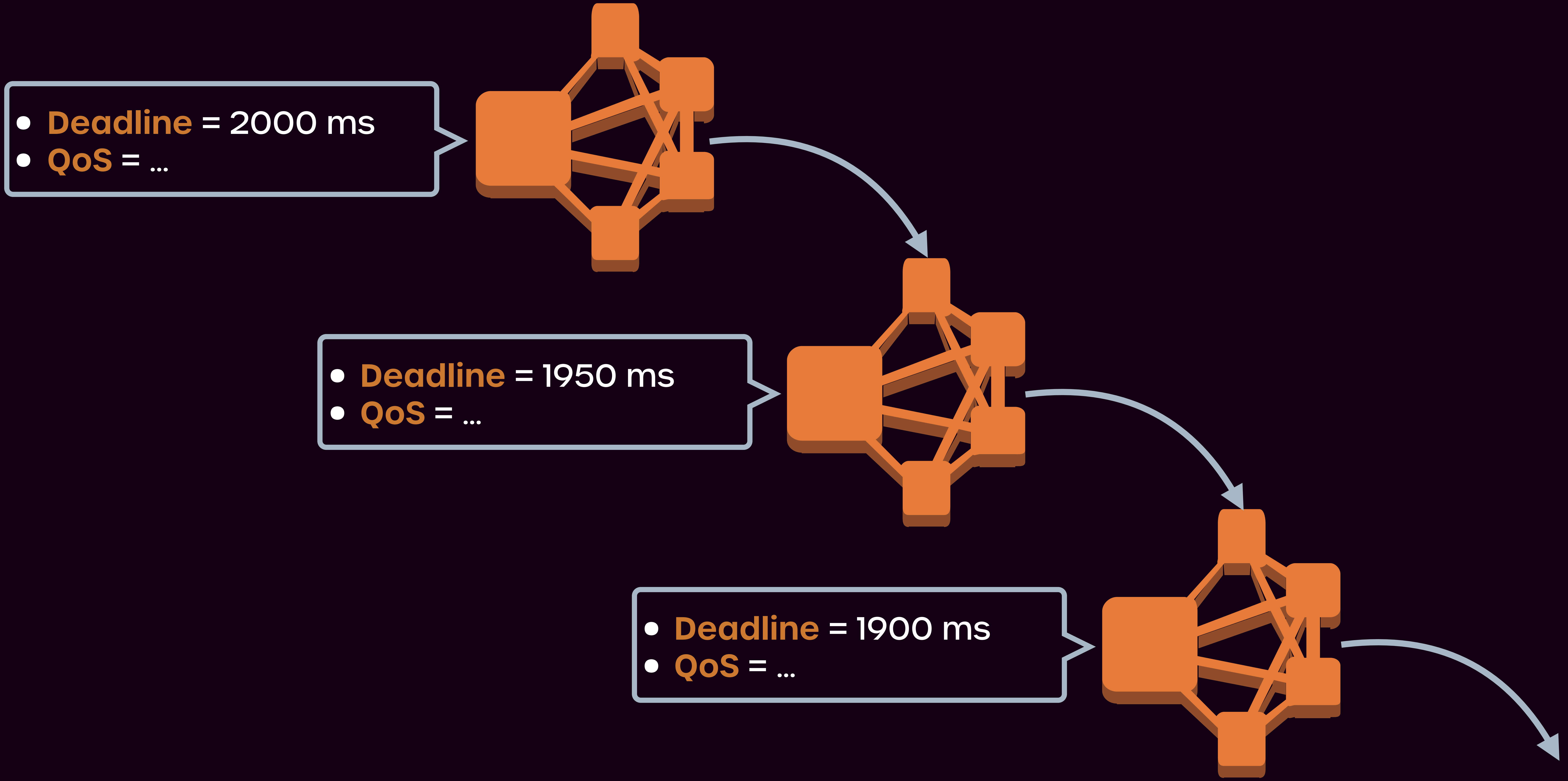
…other timeouts

…excessive downstream load when combined with re-tries


Device Shadow
Device


• Timeout = 2000 ms
• Timeout = 1950 ms
• Timeout = 1900 ms

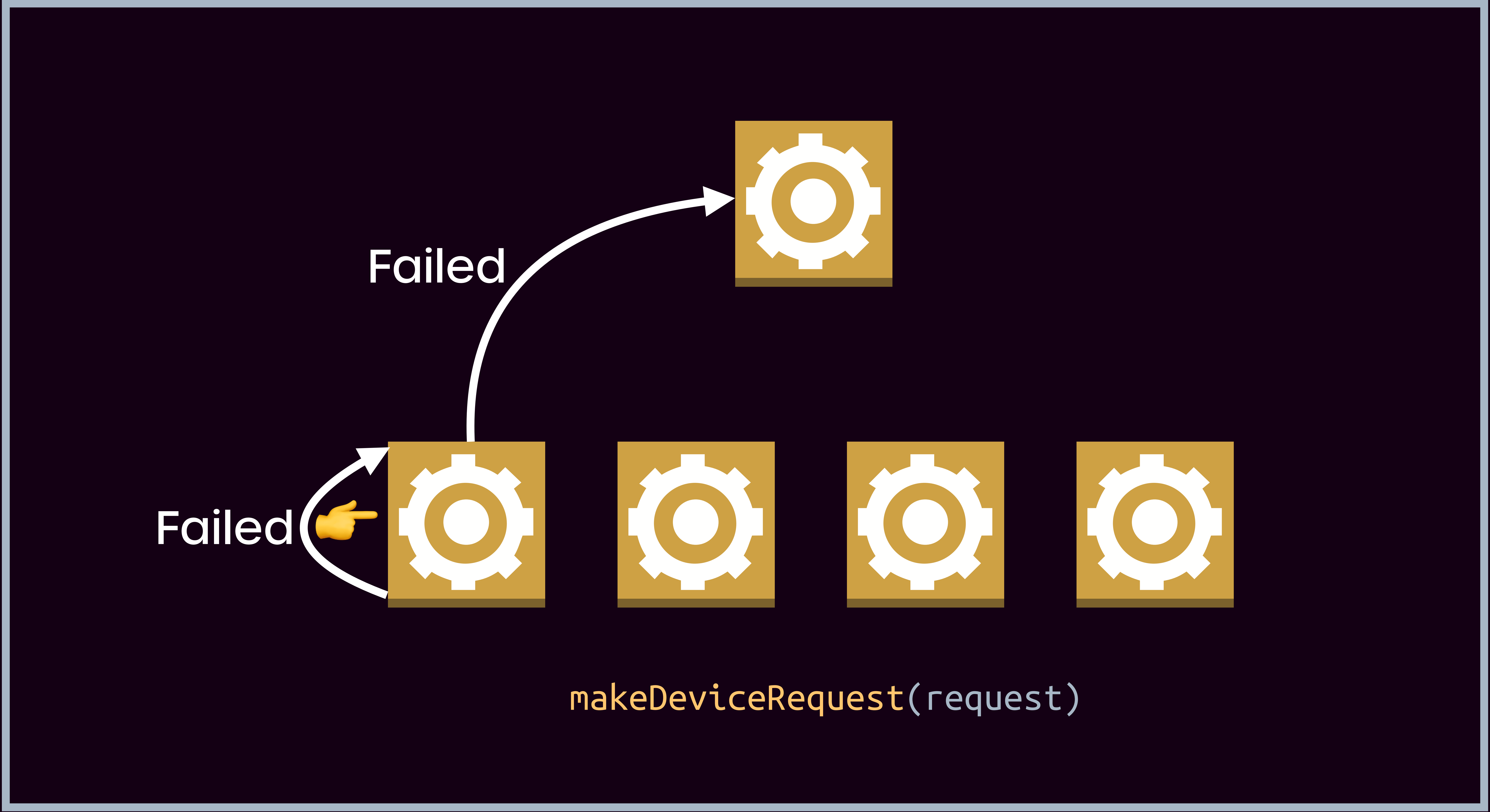

• Retries = 3
• Timeout = 666 ms
• Retries = 2
• Timeout = 975 ms
• Retries = 3
• Timeout = 633 ms



- Deadline = 2000 ms
- QoS = ...
- Deadline = 1950 ms
- QoS = ...
- Deadline = 1900 ms
- QoS = ...

```scala
def makeDeviceRequest(request: DeviceRequest): Future[DeviceResponse] = ???

makeDeviceRequest(request).onComplete {
  case Success(dr) =>
    // working!
  case Failure(ex) =>
    // failed!
    log.error(ex)
    scheduleOnce(1000L, makeDeviceRequest(request))
}
```

```scala
implicit val sys: ActorSystem = …
implicit val mat: ActorMaterializer = …
val base = Uri(…).authority
val pool = Http(sys).cachedHostConnectionPool[String](base.host.address(), base.port)

val correlationId = UUID.randomUUID().toString
val rq = HttpRequest(…)

val _ = Source.single(rq → correlationId)
              .via(pool)
           👉 .recoverWithRetries(5, …)     🎉
              .runForeach(…)
```



```
makeDeviceRequest(request)
```

A distributed system without back-pressure will fail or will make everything around it fail

“Let’s just go with the defaults for the thermal exhaust ports.” —Galen Erso

```scala
implicit val sys: ActorSystem = …
implicit val mat: ActorMaterializer = …
val base = Uri(…).authority
val pool = Http(sys).cachedHostConnectionPool[String](base.host.address(), base.port)

val correlationId = UUID.randomUUID
val rq = HttpRequest(…)

val _ = Source.single(rq → correlationId)
              .via(pool)
              .recoverWithRetries(5, …)
              .runForeach(…)
```

- Pool size
- TCP connect timeout
- TCP receive timeout

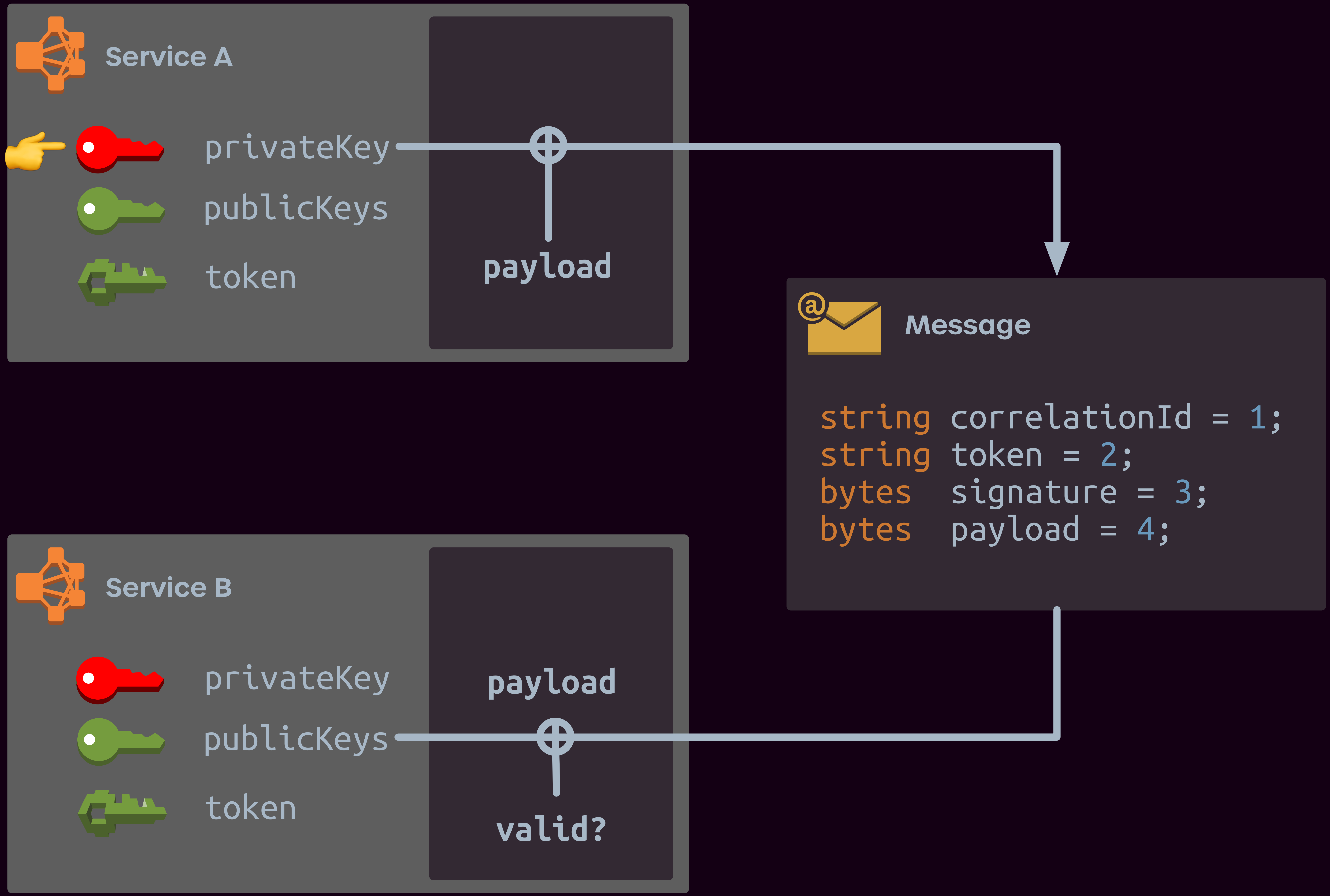
- How many retries?
- Within what time-frame?
- Idempotent endpoints?
- nonces, etc.

- Body receive timeout
- Flow timeout

A distributed system without observability and monitoring is a stack of black boxes

OS
Container
JVM
Actor system
JVM instrumentation
OS monitoring
Container monitoring
Log aggregation
JVM observability
Monitoring
Observability
Incident mangement

A distributed system without robust access control is a ticking time-bomb



```
string correlationId = 1;
string token = 2;
bytes  signature = 3;
bytes  payload = 4;
```

A distributed system without chaos testing is going to fail in the most creative ways

```
val mb = Array[Byte](8, 1, 12, 3, 65, 66, 67,
                    👉99, ..., 99)
X.validate(mb)
```

```
Exception in thread "main" java.lang.StackOverflowError
  at … $StreamDecoder.readTag(…:2051)
  at … $StreamDecoder.skipMessage(…:2158)
  at … $StreamDecoder.skipField(…:2090)
  …
```

```
val mj = """{"x":""" * 2000
JsonFormat.fromJsonString[X](mj)
```

```
Exception in thread "main" java.lang.StackOverflowError
  at … JsonStreamContext.<init>(…:43)
  at … JsonReadContext.<init>(…:58)
  at … JsonReadContext.createChildObjectContext(…:128)
  at … ReaderBasedJsonParser._nextAfterName(…:773)
  at … ReaderBasedJsonParser.nextToken(…:636)
  at … JValueDeserializer.deserialize(…:45)
```

```
message DeviceRequest {
👉string method = 1;
  string uri = 2;
  map<string, string> headers = 3;
  string entity_content_type = 4;
  bytes entity = 5;

  string schedule_time = 10;
  MisfireStrategy misfire_strategy = 11;

  enum MisfireStrategy {
    BEST_EFFORT = 0;
    FORGET = 1;
  }
}

class DeviceActor extends Actor {
  …
  override def receive: Receive = {
    case TopicPartitionOffsetMessage(tpo, dr: DeviceRequest, _) =>
      val d = Duration.between(ZonedDateTime.now, ZonedDateTime.parse(dr.scheduleTime))
      context.system.scheduler.scheduleOnce(FiniteDuration(d.toMillis, TimeUnit.MS), self, dr)
    case d: DeviceRequest =>
      Source.single(d -> …).via(…).run(…)
  }
}
```

```
message DeviceRequest {
  string method = 1;.......................... "Invoking the fèeling of cháos"
  string uri = 2;............................. "a/%%30%30"
  map<string, string> headers = 3;............ Map.empty
  string entity_content_type = 4;............. "#cmds=({'/bin/echo', #eps})"
  bytes entity = 5;........................... "4oGmdGVzdOKBpw=="

  string schedule_time = 10;................👉"2017-10-14T11:42:06+00:00"
  MisfireStrategy misfire_strategy = 11;...... "BEST_EFFORT"

  enum MisfireStrategy {
    BEST_EFFORT = 0;
    FORGET = 1;
  }
}

class DeviceActor extends Actor {
  …
  override def receive: Receive = {
    case TopicPartitionOffsetMessage(tpo, dr: DeviceRequest, _) =>
      val d = Duration.between(ZonedDateTime.now, ZonedDateTime.parse(dr.scheduleTime))
      context.system.scheduler.scheduleOnce(FiniteDuration(d.toMillis, TimeUnit.MS), self, dr)
    case d: DeviceRequest =>
      Source.single(d -> …).via(…).run(…)
  }
}
```

```
class DeviceActor extends Actor {
  …
  override def receive: Receive = {
👉case TopicPartitionOffsetMessage(tpo, dr: DeviceRequest, _) =>
      val d = Duration.between(ZonedDateTime.now, ZonedDateTime.parse(dr.scheduleTime))
      context.system.scheduler.scheduleOnce(FiniteDuration(d.toMillis, TimeUnit.MS), self, dr)
    case d: DeviceRequest =>
      Source.single(d -> …).via(…).run(…)
  }
}
```

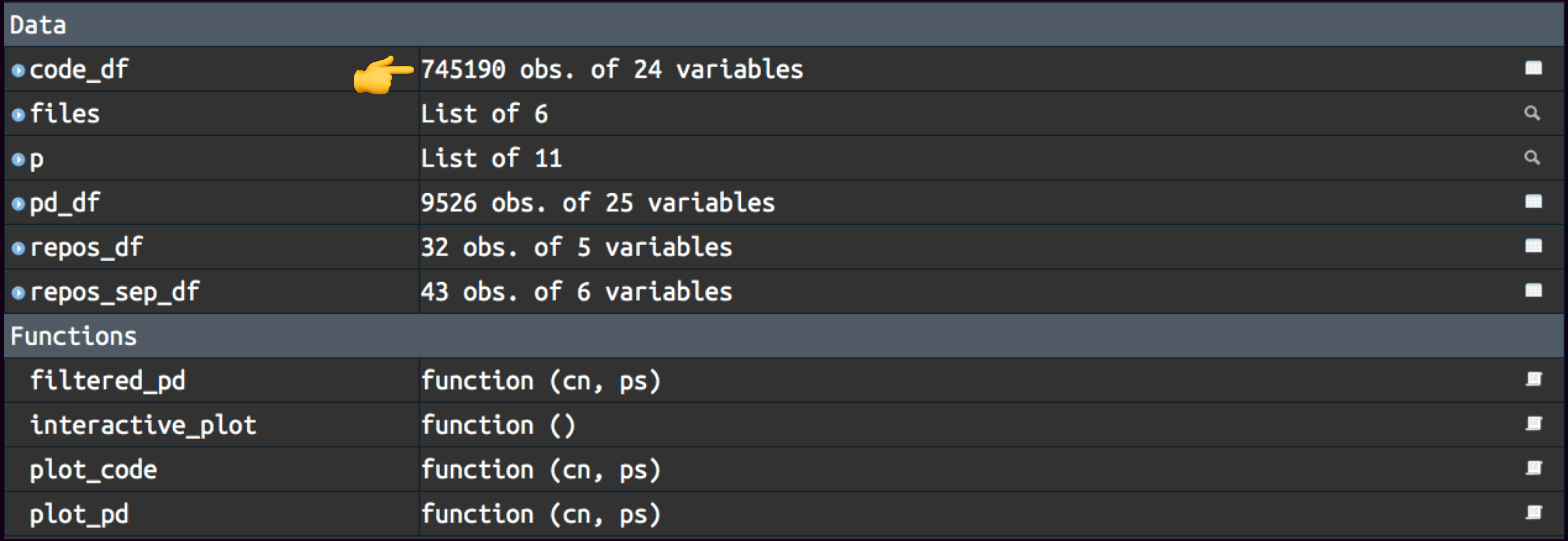
```
💣  Exception in thread "…" java.time.format.DateTimeParseException:
        Text '2014-12-10T05:44:06.635Z[😜] could not be parsed at index 21

💣  Exception in thread "…" java.time.format.DateTimeParseException:
        Text '2014-12-10T05:44:06.635Z[GMT] could not be parsed at index 11

💣  SIGSEGV (0xb) at pc=0x000000010fda8262, pid=21419, tid=18435
    # V  [libjvm.dylib+0x3a8262]  PhaseIdealLoop::idom_no_update(Node*) const+0x12

💣  GET http://host/foo.action Content-Type: #cmds=({'/bin/echo', #eps})
```

https://github.com/minimaxir/big-list-of-naughty-strings

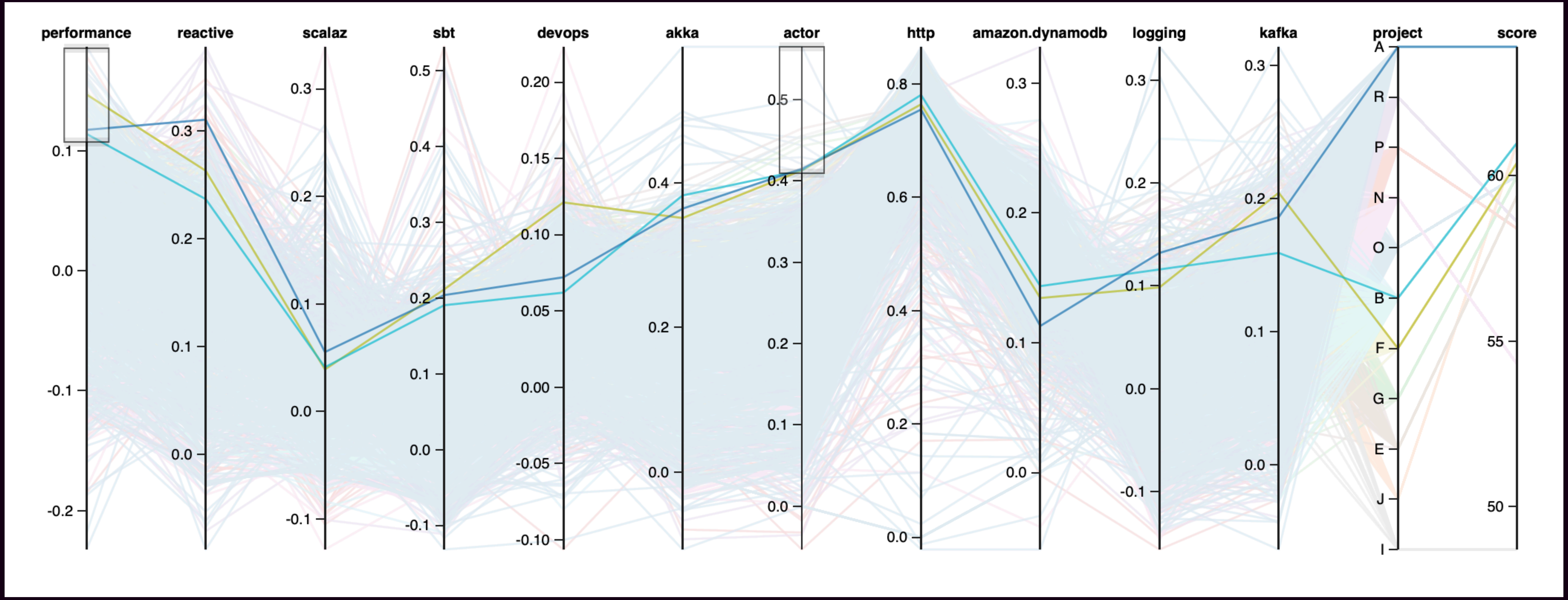
Do tell another anecdote…

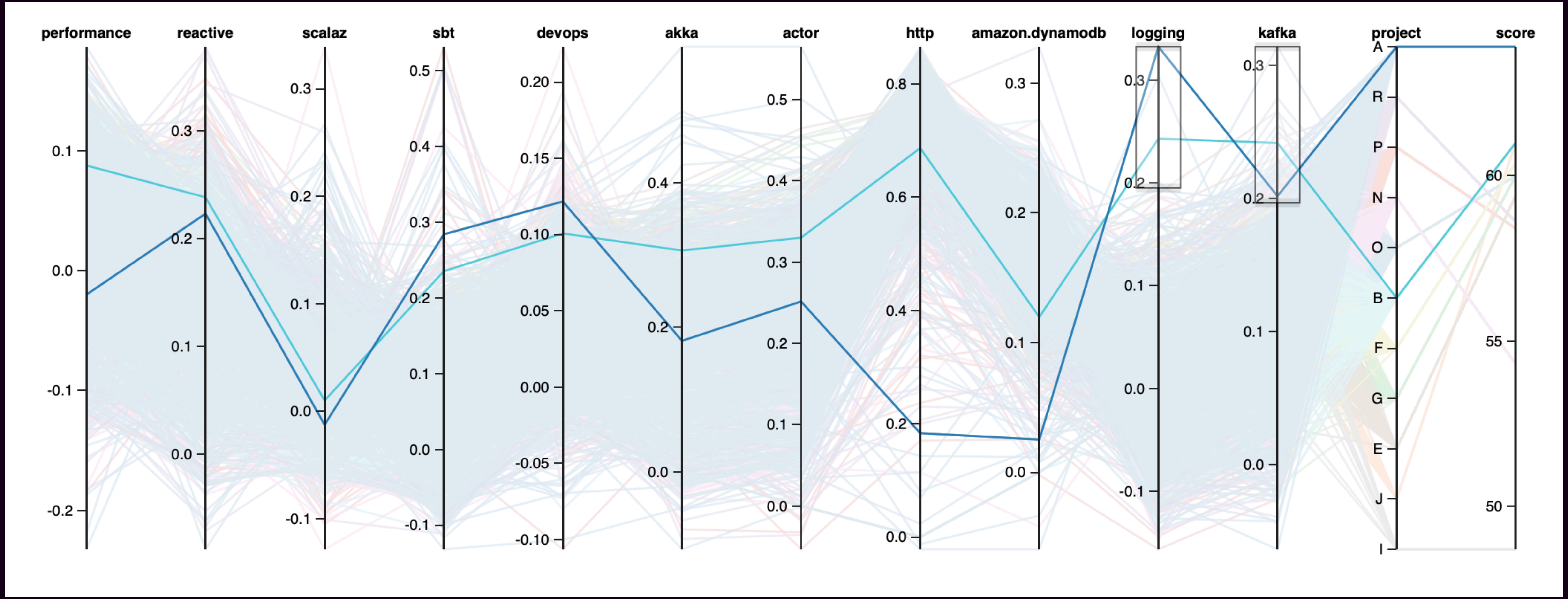
We measured

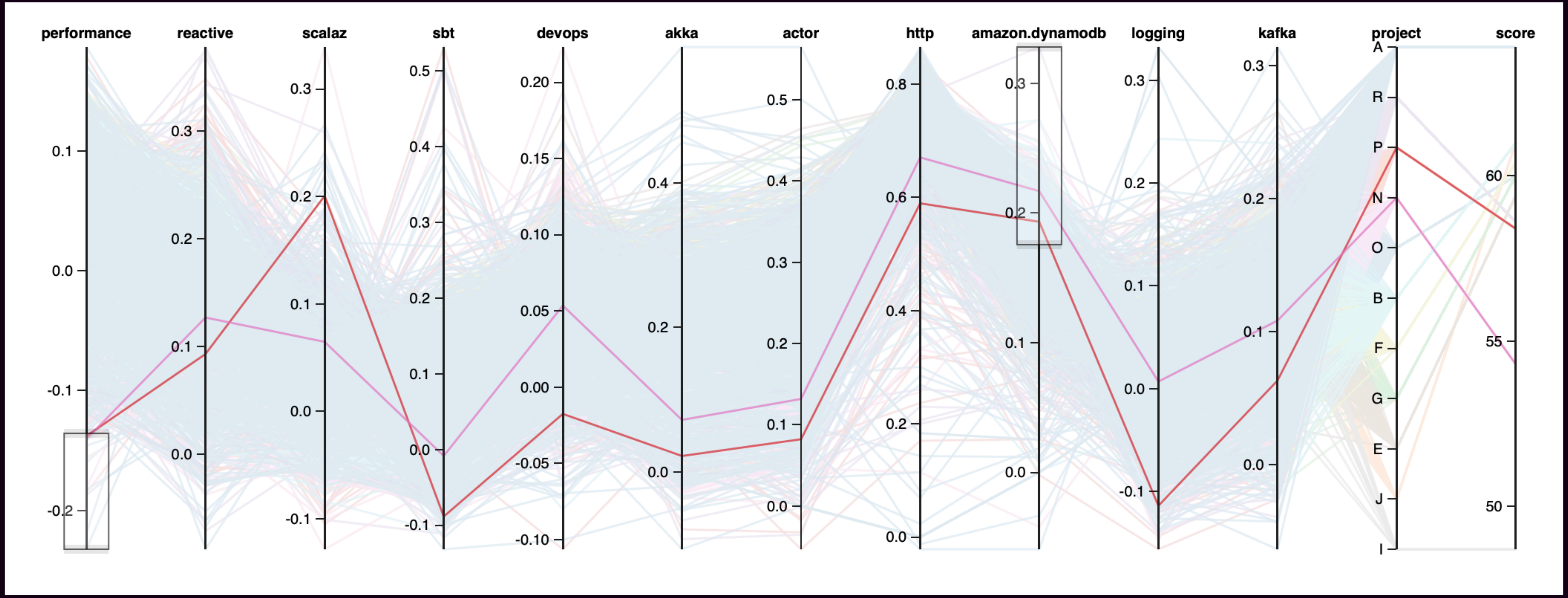
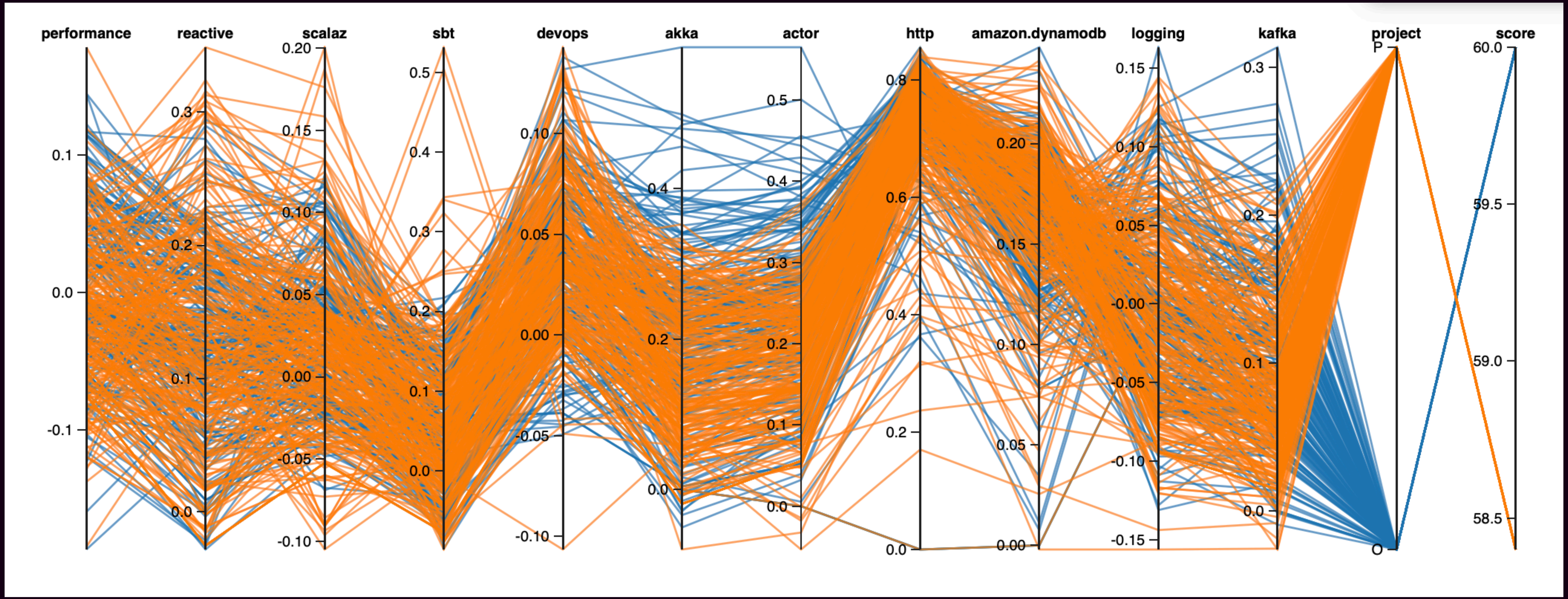
- For every file in every commit in every project…
- Classification of the kind and quality of code
- Matching production performance data from PagerDuty



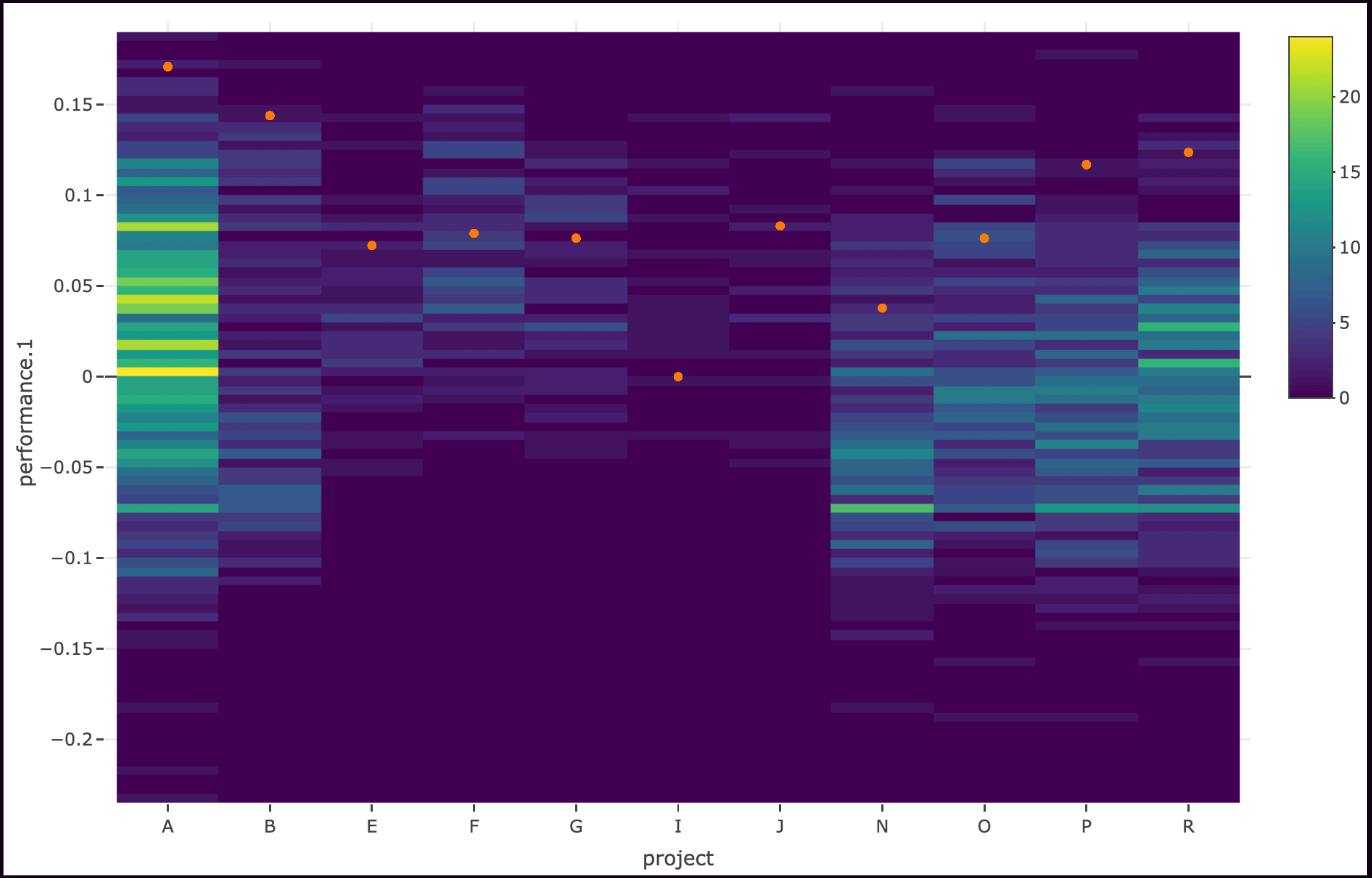
Data	
code_df	👉 745190 obs. of 24 variables
files	List of 6
p	List of 11
pd_df	9526 obs. of 25 variables
repos_df	32 obs. of 5 variables
repos_sep_df	43 obs. of 6 variables
Functions	
filtered_pd	function (cn, ps)
interactive_plot	function ()
plot_code	function (cn, ps)
plot_pd	function (cn, ps)


performance
reactive
scalaz
sbt
devops
akka
actor
http
amazon.dynamodb
logging
kafka
project
score

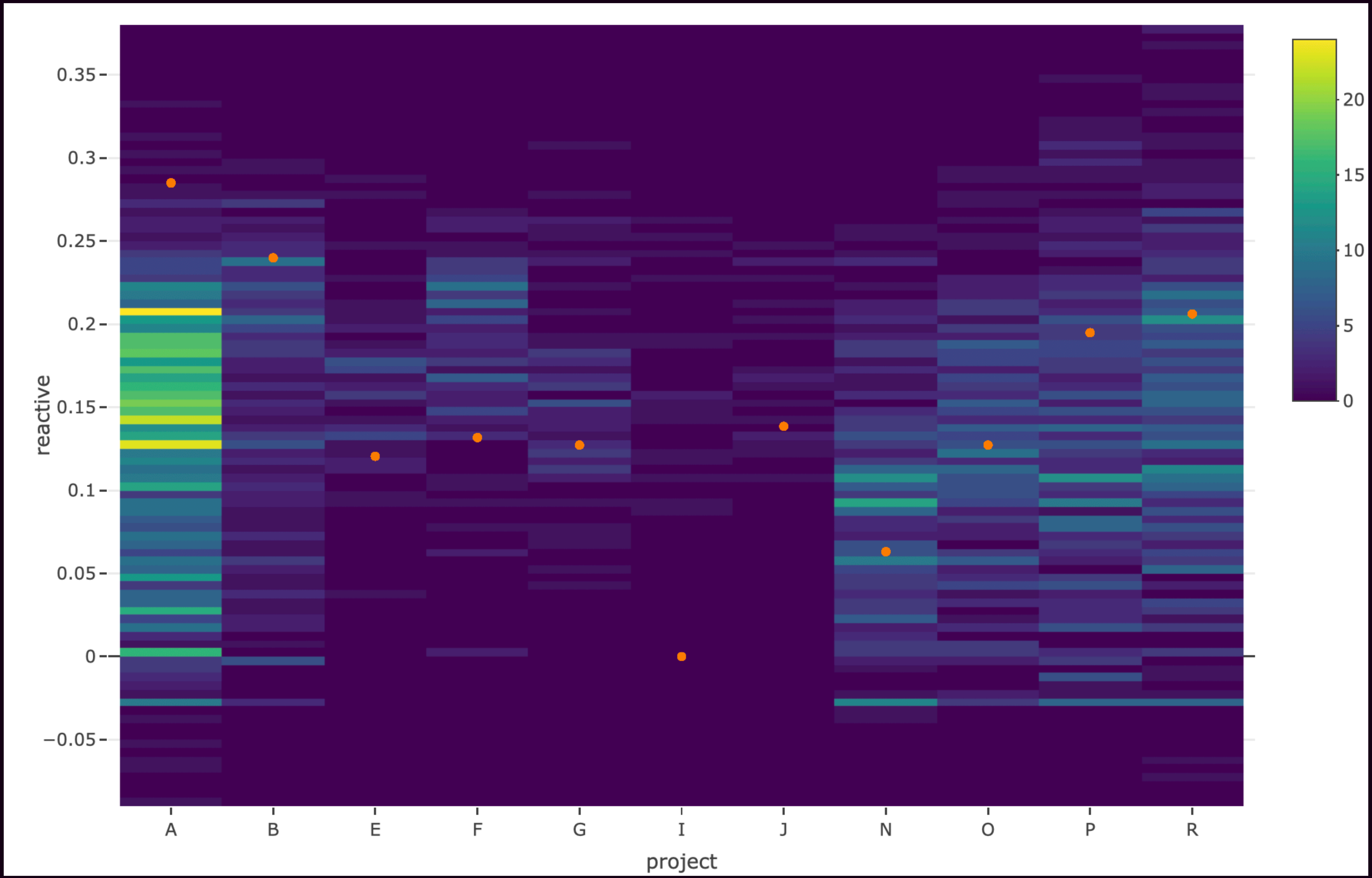

performance
reactive
scalaz
sbt
devops
akka
actor
http
amazon.dynamodb
logging
kafka
project
score


performance
reactive
scalaz
sbt
devops
akka
actor
http
amazon.dynamodb
logging
kafka
project
score

performance
reactive
scalaz
sbt
devops
akka
actor
http
amazon.dynamodb
logging
kafka
project
score

The biggest impact on production performance comes from…

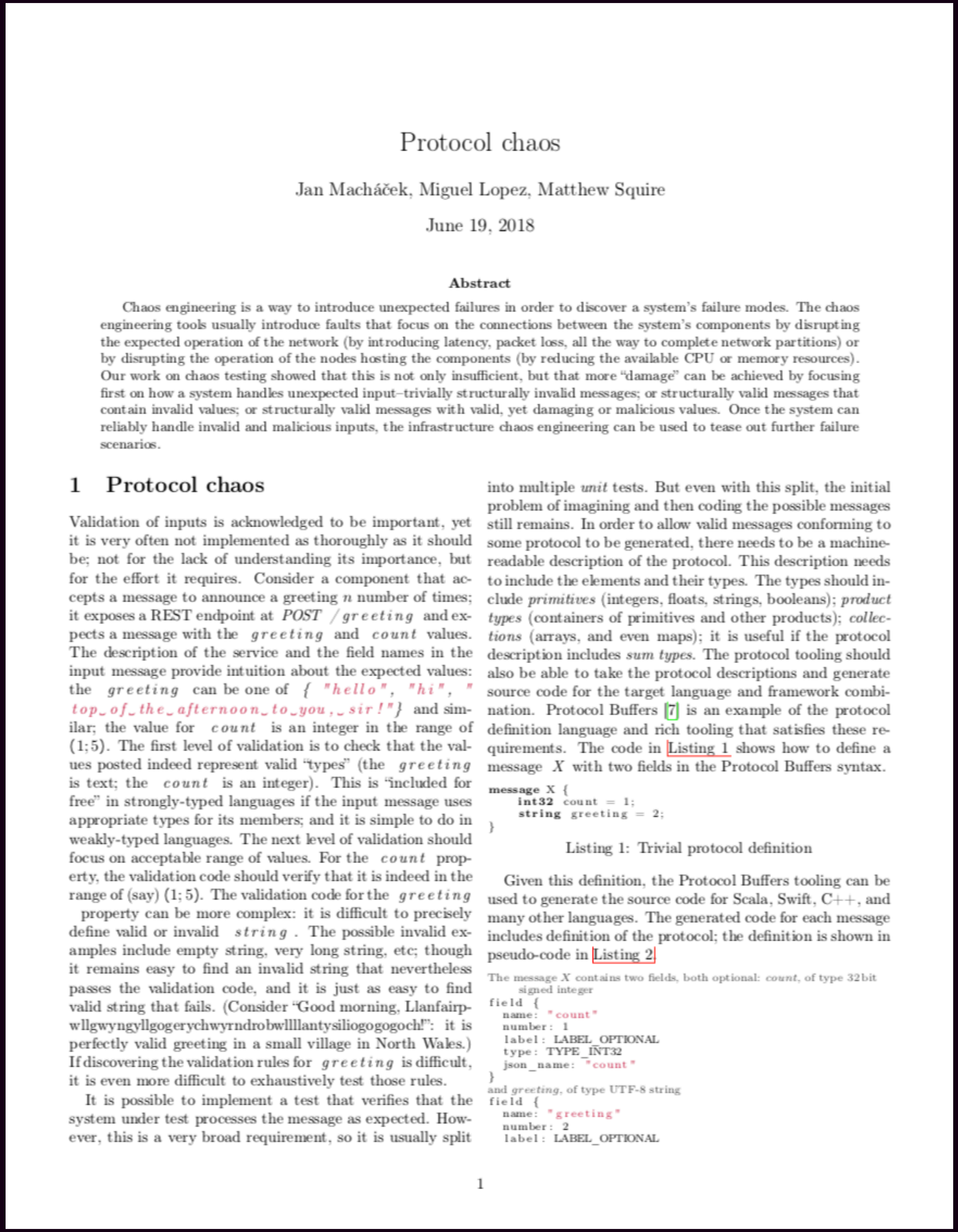


0.15
0.1
0.05
0
−0.05
−0.1
−0.15
−0.2
performance.1
A
B
E
F
G
I
J
N
O
P
R
project
20
15
10
5
0


0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
−0.05
reactive
A
B
E
F
G
I
J
N
O
P
R
project
20
15
10
5
0

Four things successful projects do throughout their commit history

- Structured and performance-tested logging
- Monitoring & [distributed] tracing
- Performance testing
- Reactive architecture & code



Protocol chaos

Jan Macháček, Miguel Lopez, Matthew Squire

June 19, 2018

Abstract

Chaos engineering is a way to introduce unexpected failures in order to discover a system's failure modes. The chaos engineering tools usually introduce faults that focus on the connections between the system's components by disrupting the expected operation of the network (by introducing latency, packet loss, all the way to complete network partitions) or by disrupting the operation of the nodes hosting the components (by reducing the available CPU or memory resources). Our work on chaos testing showed that this is not only insufficient, but that more "damage" can be achieved by focusing first on how a system handles unexpected input–trivially structurally invalid messages; or structurally valid messages that contain invalid values; or structurally valid messages with valid, yet damaging or malicious values. Once the system can reliably handle invalid and malicious inputs, the infrastructure chaos engineering can be used to tease out further failure scenarios.

1 Protocol chaos

Validation of inputs is acknowledged to be important, yet it is very often not implemented as thoroughly as it should be; not for the lack of understanding its importance, but for the effort it requires. Consider a component that accepts a message to announce a greeting n number of times; it exposes a REST endpoint at *POST* */greeting* and expects a message with the *greeting* and *count* values. The description of the service and the field names in the input message provide intuition about the expected values: the *greeting* can be one of { *"hello"*, *"hi"*, *"top_of_the_afternoon_to_you,_sir!"*} and similar; the value for *count* is an integer in the range of (1; 5). The first level of validation is to check that the values posted indeed represent valid "types" (the *greeting* is text; the *count* is an integer). This is "included for free" in strongly-typed languages if the input message uses appropriate types for its members; and it is simple to do in weakly-typed languages. The next level of validation should focus on acceptable range of values. For the *count* property, the validation code should verify that it is indeed in the range of (say) (1; 5). The validation code for the *greeting* property can be more complex: it is difficult to precisely define valid or invalid *string*. The possible invalid examples include empty string, very long string, etc; though it remains easy to find an invalid string that nevertheless passes the validation code, and it is just as easy to find valid string that fails. (Consider "Good morning, Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoch!": it is perfectly valid greeting in a small village in North Wales.) If discovering the validation rules for *greeting* is difficult, it is even more difficult to exhaustively test those rules.

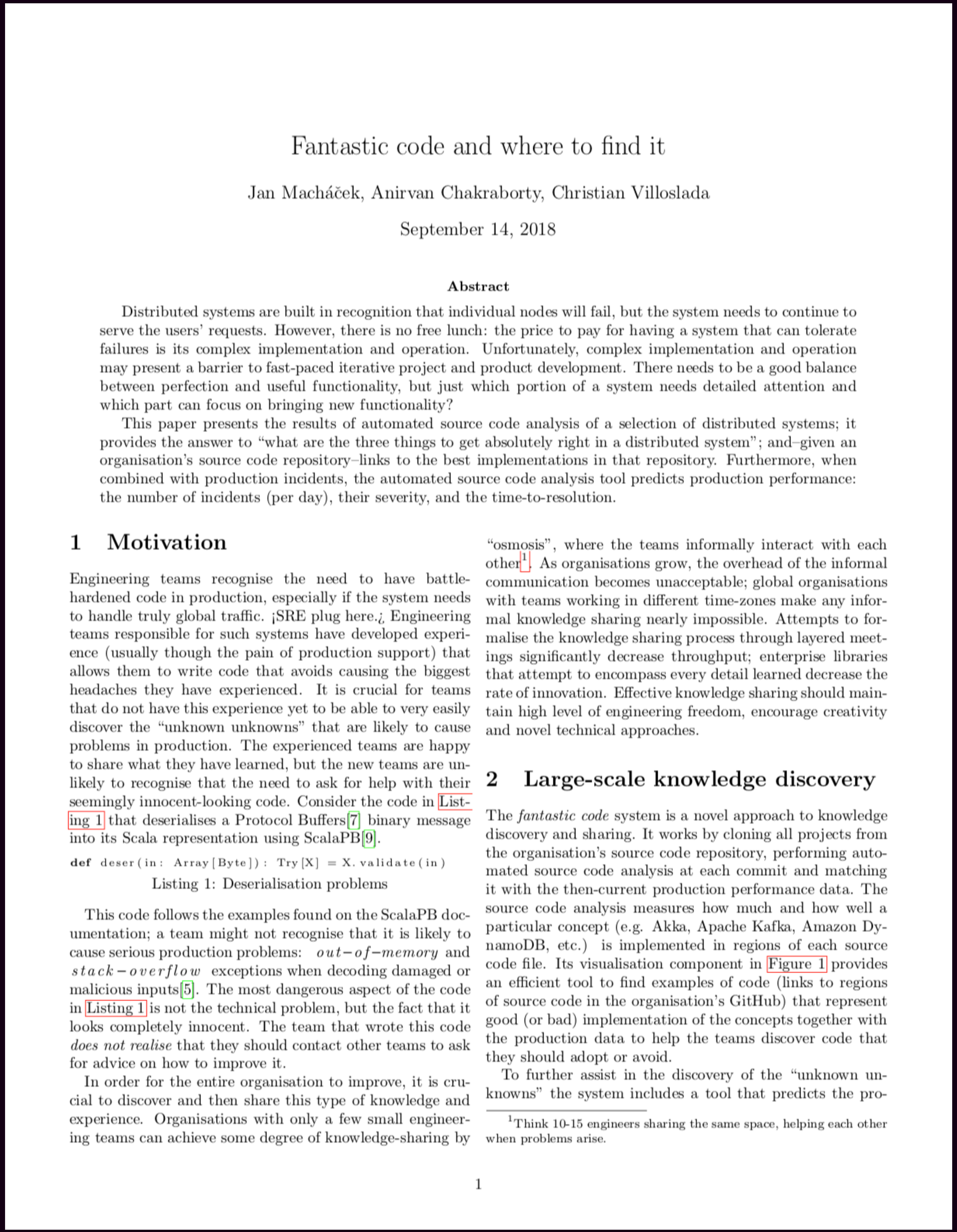
It is possible to implement a test that verifies that the system under test processes the message as expected. However, this is a very broad requirement, so it is usually split into multiple *unit* tests. But even with this split, the initial problem of imagining and then coding the possible messages still remains. In order to allow valid messages conforming to some protocol to be generated, there needs to be a machine-readable description of the protocol. This description needs to include the elements and their types. The types should include *primitives* (integers, floats, strings, booleans); *product types* (containers of primitives and other products); *collections* (arrays, and even maps); it is useful if the protocol description includes *sum types*. The protocol tooling should also be able to take the protocol descriptions and generate source code for the target language and framework combination. Protocol Buffers [7] is an example of the protocol definition language and rich tooling that satisfies these requirements. The code in Listing 1 shows how to define a message X with two fields in the Protocol Buffers syntax.

```
message X {
    int32 count = 1;
    string greeting = 2;
}
```

Listing 1: Trivial protocol definition

Given this definition, the Protocol Buffers tooling can be used to generate the source code for Scala, Swift, C++, and many other languages. The generated code for each message includes definition of the protocol; the definition is shown in pseudo-code in Listing 2.

```
The message X contains two fields, both optional: count, of type 32bit
    signed integer
field {
  name: "count"
  number: 1
  label: LABEL_OPTIONAL
  type: TYPE_INT32
  json_name: "count"
}
and greeting, of type UTF-8 string
field {
  name: "greeting"
  number: 2
  label: LABEL_OPTIONAL
```



Fantastic code and where to find it

Jan Macháček, Anirvan Chakraborty, Christian Villoslada

September 14, 2018

Abstract

Distributed systems are built in recognition that individual nodes will fail, but the system needs to continue to serve the users' requests. However, there is no free lunch: the price to pay for having a system that can tolerate failures is its complex implementation and operation. Unfortunately, complex implementation and operation may present a barrier to fast-paced iterative project and product development. There needs to be a good balance between perfection and useful functionality, but just which portion of a system needs detailed attention and which part can focus on bringing new functionality?

This paper presents the results of automated source code analysis of a selection of distributed systems; it provides the answer to "what are the three things to get absolutely right in a distributed system"; and–given an organisation's source code repository–links to the best implementations in that repository. Furthermore, when combined with production incidents, the automated source code analysis tool predicts production performance: the number of incidents (per day), their severity, and the time-to-resolution.

1 Motivation

Engineering teams recognise the need to have battle-hardened code in production, especially if the system needs to handle truly global traffic. ¡SRE plug here.¿ Engineering teams responsible for such systems have developed experience (usually though the pain of production support) that allows them to write code that avoids causing the biggest headaches they have experienced. It is crucial for teams that do not have this experience yet to be able to very easily discover the "unknown unknowns" that are likely to cause problems in production. The experienced teams are happy to share what they have learned, but the new teams are unlikely to recognise that the need to ask for help with their seemingly innocent-looking code. Consider the code in Listing 1 that deserialises a Protocol Buffers[7] binary message into its Scala representation using ScalaPB[9].

```
def deser(in: Array[Byte]): Try[X] = X.validate(in)
```

Listing 1: Deserialisation problems

This code follows the examples found on the ScalaPB documentation; a team might not recognise that it is likely to cause serious production problems: *out−of−memory* and *stack−overflow* exceptions when decoding damaged or malicious inputs[5]. The most dangerous aspect of the code in Listing 1 is not the technical problem, but the fact that it looks completely innocent. The team that wrote this code *does not realise* that they should contact other teams to ask for advice on how to improve it.

In order for the entire organisation to improve, it is crucial to discover and then share this type of knowledge and experience. Organisations with only a few small engineering teams can achieve some degree of knowledge-sharing by "osmosis", where the teams informally interact with each other[1]. As organisations grow, the overhead of the informal communication becomes unacceptable; global organisations with teams working in different time-zones make any informal knowledge sharing nearly impossible. Attempts to formalise the knowledge sharing process through layered meetings significantly decrease throughput; enterprise libraries that attempt to encompass every detail learned decrease the rate of innovation. Effective knowledge sharing should maintain high level of engineering freedom, encourage creativity and novel technical approaches.

2 Large-scale knowledge discovery

The *fantastic code* system is a novel approach to knowledge discovery and sharing. It works by cloning all projects from the organisation's source code repository, performing automated source code analysis at each commit and matching it with the then-current production performance data. The source code analysis measures how much and how well a particular concept (e.g. Akka, Apache Kafka, Amazon DynamoDB, etc.) is implemented in regions of each source code file. Its visualisation component in Figure 1 provides an efficient tool to find examples of code (links to regions of source code in the organisation's GitHub) that represent good (or bad) implementation of the concepts together with the production data to help the teams discover code that they should adopt or avoid.

To further assist in the discovery of the "unknown unknowns" the system includes a tool that predicts the pro-

[1]Think 10-15 engineers sharing the same space, helping each other when problems arise.

Thank you

jan.machacek@disneystreaming.com
matthew.squire@disneystreaming.com