# REACTIVE « JAMMED » ARCHITECTURE

## ou comment survivre à l'A86 en heure de pointe



Nicolas PHUNG @nsphung

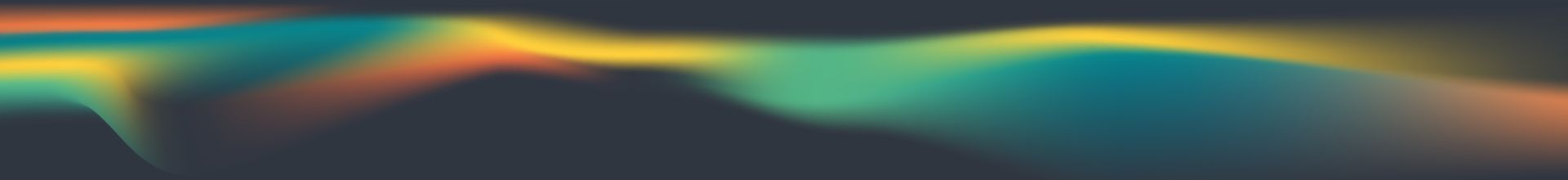

Jennifer AGUIAR @yaourtcorp

# The COSMO project

# Next generation e-checking system

- Nationwide train passenger e-checking
- Near real time
- Offline device capabilities
- Work with complex existing services

# Crowded Services/Partners



# Crowded services/partners

- Galaxies of services
- Each services got it speed limit and various QoS
- We are not the only customer !
- Some services are already jammed with lots of layers

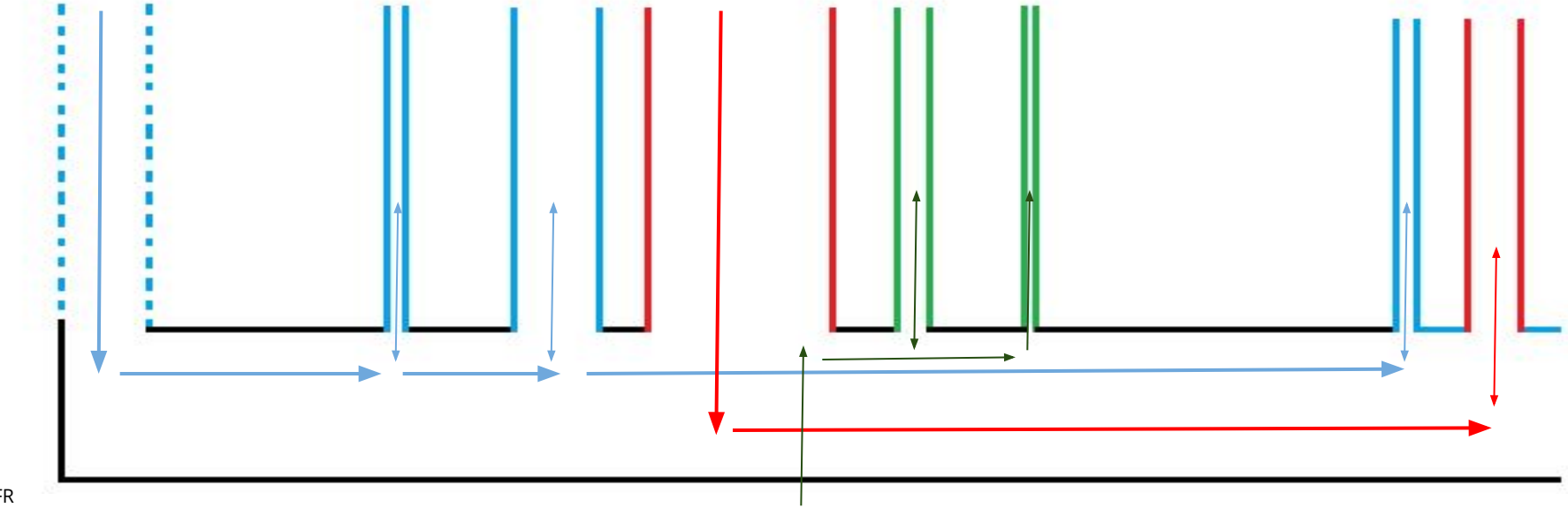
# Different Constraints

- “Real Time”
- Scalable (sometimes)
- Resilient
- Batch !?

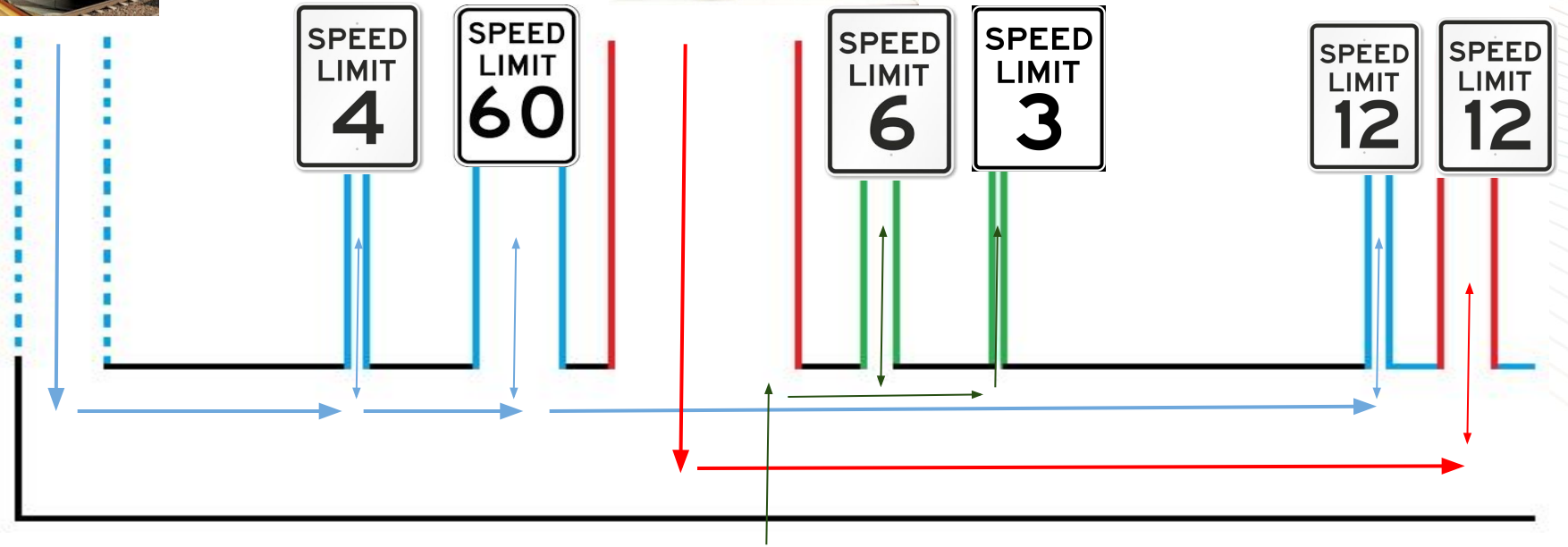

The A-team | mertcaliskan | CC3

# ROADS...MAP


BILLET à composter avant l'accès au train
BILLET à composter avant l'accès au train
BILLET à composter avant l'accès au train

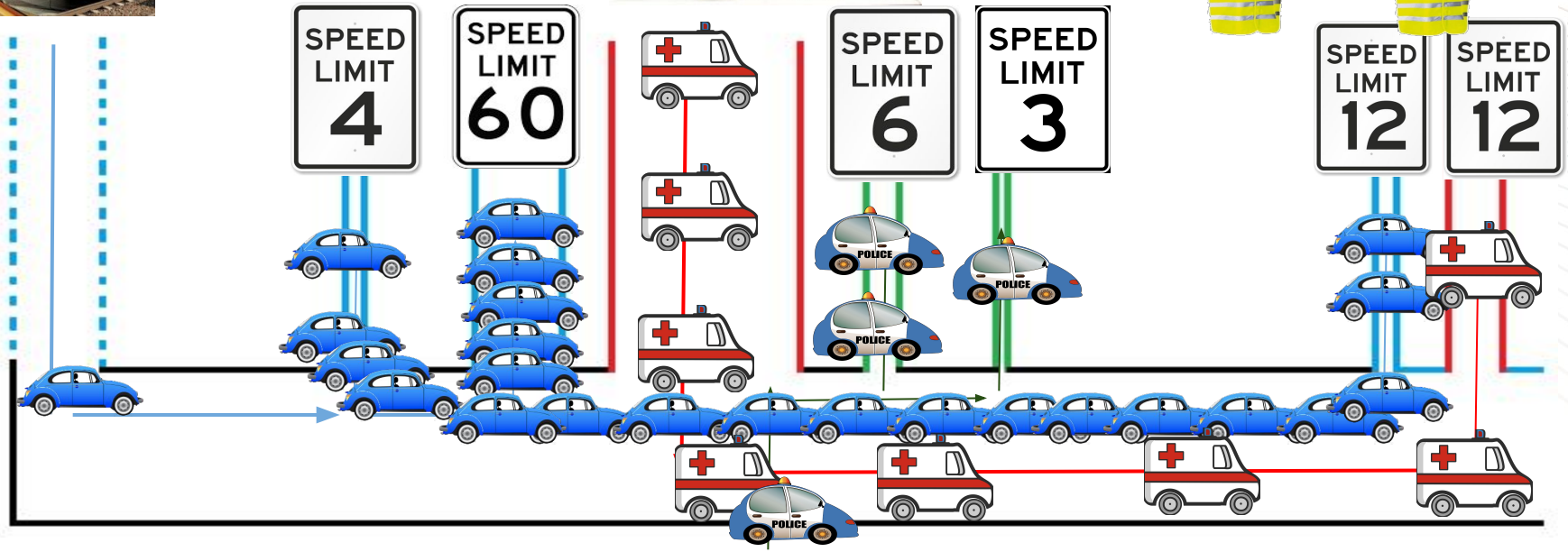


# + SPEED...LIMITS

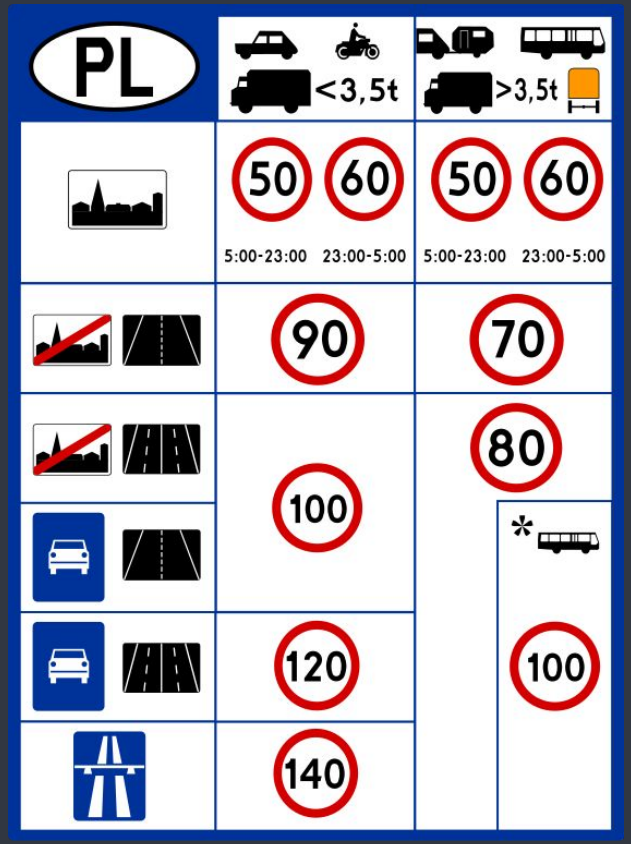



# = TRAFFIC...JAM


BILLET à composter avant l'accès au train
BILLET à composter avant l'accès au train
BILLET à composter avant l'accès au train


SPEED LIMIT 4
SPEED LIMIT 60
SPEED LIMIT 6
SPEED LIMIT 3
SPEED LIMIT 12
SPEED LIMIT 12
POLICE
POLICE
POLICE
POLICE

# REQUIREMENTS



# KAFKA and AKKA to the rescue

# CONCEPTS THAT WILL HELP US

- Speed limit => Throttle 
- Keep up with jammed services => Backpressure
- Address near real time concern => Event based Architecture

# Back to the pressure

Dès que le trafic devient plus dense, la circulation ralentit.

**Dès que le trafic devient plus dense, la circulation ralentit. Jusqu'à l'arrêt ou presque. Chacun s'attend à apercevoir l'accident ou les travaux à l'origine du ralentissement quand, sans raison apparente, la situation se débloque**. Chacun reprend alors son rythme de croisière. Mais quelques kilomètres plus loin, ça recommence... "Les 'accordéons' se forment quand la circulation est déjà congestionnée, explique Christine Buisson, du Laboratoire d'ingénierie Circulation Transports (Licit), à Vaulx-en-Velin. Dans ces conditions, la vitesse d'un véhicule est imposée par le véhicule qui le précède. Si elle ralentit, les autres aussi."

L'explication vient de la dynamique des fluides, car les voitures en circulation sont comme un fluide de n'importe quelle matière qui s'écoule dans un contenant, un tuyau par exemple, à la capacité limitée. Ainsi, les théoriciens du trafic routier utilisent les lois de l'hydrodynamique pour simuler la circulation des véhicules.

**une vague de compression qui se propage dans la circulation vers l'arrière** !!!

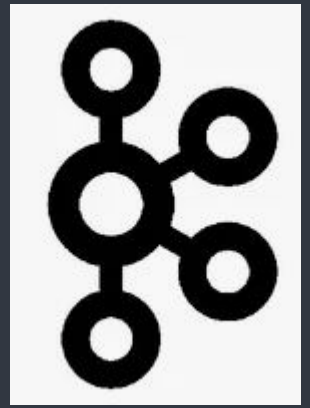
"D'un point de vue macroscopique, on voit bien dans nos simulations que si le trafic passe rapidement de 2 000 à 4 000 véhicules par heure, une compression se forme qui se propage comme une vague." Une vague qui présente la particularité de déferler dans le sens inverse de la circulation !

Source : https://www.science-et-vie.com/questions-reponses/pourquoi-la-circulation-fait-elle-laccordeon-5939

# Back to the pressure (Part II)

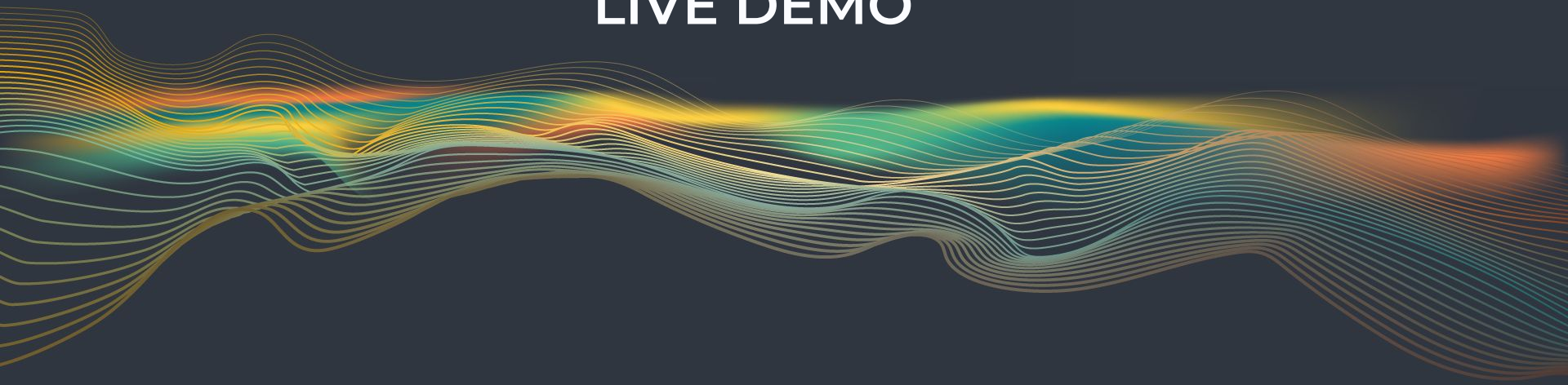
We may need a big buffer...


Insane Traffic jam | Sploid



# Back to the pressure (Part III)

Is this a holdup?
No, it's a science experiment!

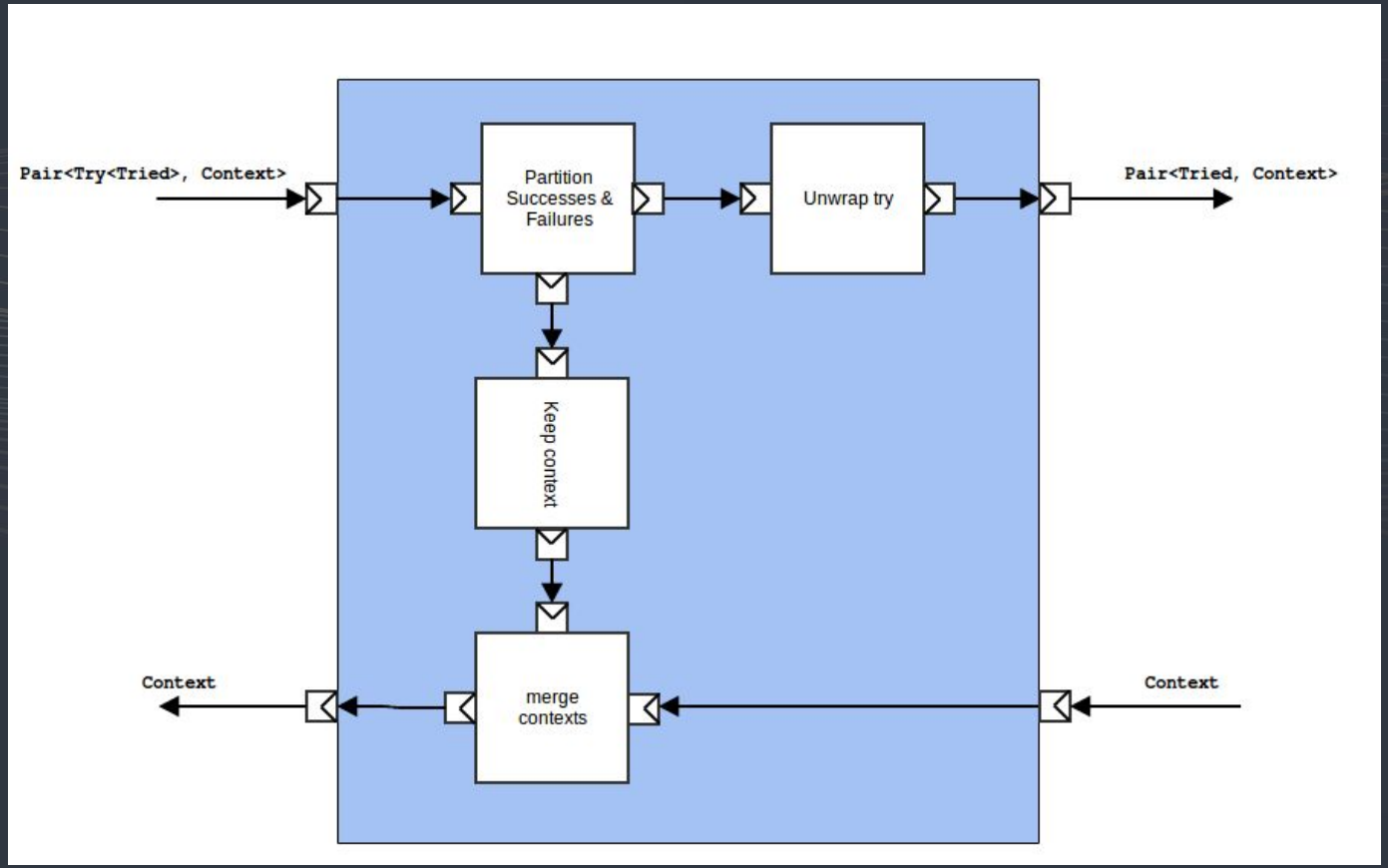


# CONCEPTS THAT WILL HELP US

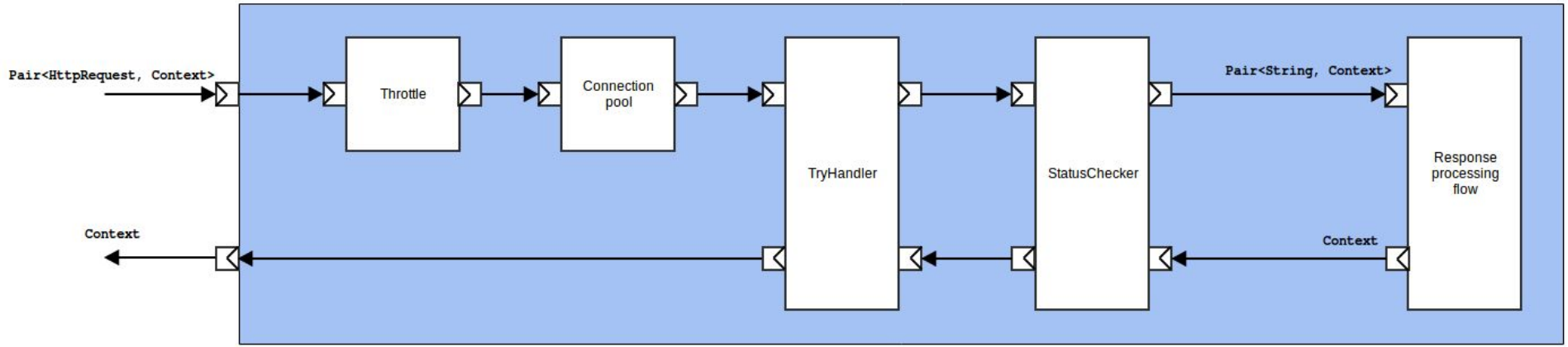
- Speed limit => Throttle ✔
- Keep up with jammed services => Backpressure ✔
- Address near real time concern => Event-based Architecture

# KISS with AKKA-Stream

# FIRST VERSION



# Not so V

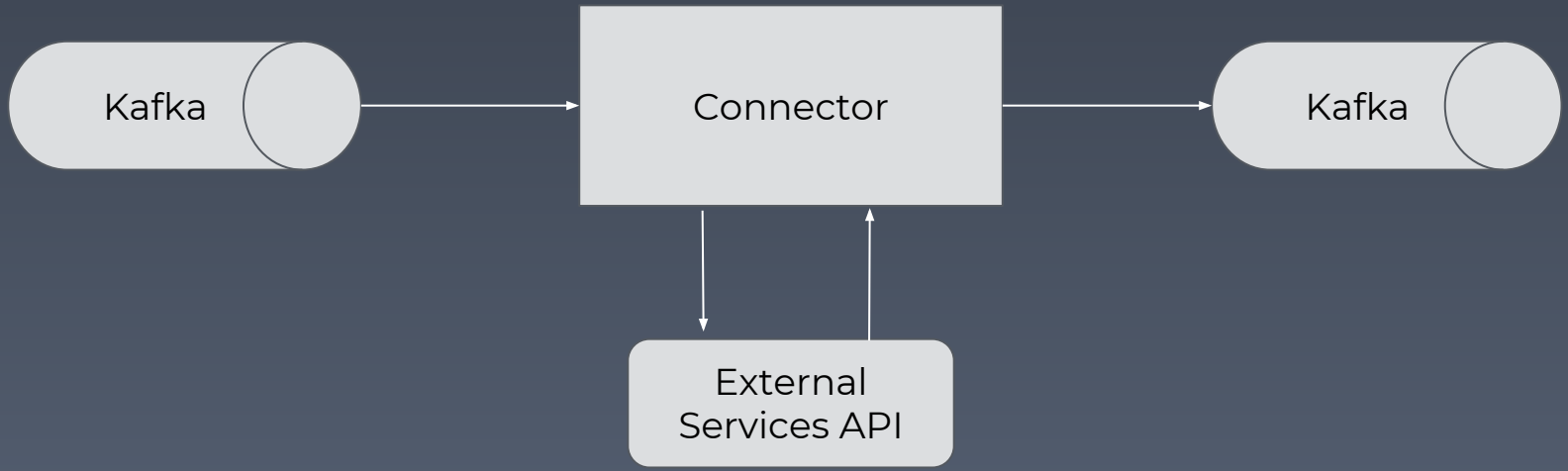



Lots of code / clutter components

Really difficult to test/evolve

THAT'S BEEN ONE OF MY MANTRAS
*FOCUS & SIMPLICITY*
SIMPLE CAN BE HARDER THAN COMPLEX; YOU HAVE TO WORK HARD TO GET YOUR THINKING CLEAN AND MAKE IT SIMPLE
- STEVE JOBS

# Main use case



# Simpler version

```
val graph = source
  .collect(CollectSpecificTypeInEnvelope[OrderRequestEvent])
  .filter(keepEventAfterYesterdaySessionId(_, MARKER, skipPastDays))
  .via(buildStreamContextFlow)
  .throttle(throttling.maxRequestsPerSecond, throttling.period, throttling.maximumBurst, throttling.throttleMode)
  .mapAsync(mapAsyncParallelism)(CallOrderDetailsPaoWebService(orderDetailsUtils, redisService, paoSettings))
  .mapConcat[(OrderDetailsContext, Option[TicketScope])](pairEventsToContextWithOptionalOffsetTupled)
  .via(buildKafkaProducerMessageFlow)
  .via(producerFlow)
  .map(elem => elem.message.passThrough)
  // -- Extract only OrderByTrainContext with Some committable offset
  .collect { case OrderDetailsContext(_, _, _, _, Some(offset), _) => offset }
  .via(commitOffsetBackToKafkaTopicSourceFlow(commitParallelism))
  .withAttributes(ActorAttributes.withSupervisionStrategy(streamDecider))
  .to(sink)
```

# Simpler version

Read from kafka

```
val graph = source
  .collect(CollectSpecificTypeInEnvelope[OrderRequestEvent])
  .filter(keepEventAfterYesterdaySessionId(_, MARKER, skipPastDays))
  .via(buildStreamContextFlow)
  .throttle(throttling.maxRequestsPerSecond, throttling.period, throttling.maximumBurst, throttling.throttleMode)
  .mapAsync(mapAsyncParallelism)(CallOrderDetailsPaoWebService(orderDetailsUtils, redisService, paoSettings))
  .mapConcat[(OrderDetailsContext, Option[TicketScope])](pairEventsToContextWithOptionalOffsetTupled)
  .via(buildKafkaProducerMessageFlow)
  .via(producerFlow)
  .map(elem => elem.message.passThrough)
  // -- Extract only OrderByTrainContext with Some committable offset
  .collect { case OrderDetailsContext(_, _, _, _, Some(offset), _) => offset }
  .via(commitOffsetBackToKafkaTopicSourceFlow(commitParallelism))
  .withAttributes(ActorAttributes.withSupervisionStrategy(streamDecider))
  .to(sink)
```

# Simpler version

```
val graph = source
  .collect(CollectSpecificTyp                                    t])
  .filter(keepEventAfterYeste                                    oPastDays))
  .via(buildStreamContextFlow)
  .throttle(throttling.maxRequestsPerSecond, throttling.period, throttling.maximumBurst, throttling.throttleMode)
  .mapAsync(mapAsyncParallelism)(CallOrderDetailsPaoWebService(orderDetailsUtils, redisService, paoSettings))
  .mapConcat[(OrderDetailsContext, Option[TicketScope])](pairEventsToContextWithOptionalOffsetTupled)
  .via(buildKafkaProducerMessageFlow)
  .via(producerFlow)
  .map(elem => elem.message.passThrough)
  // -- Extract only OrderByTrainContext with Some committable offset
  .collect { case OrderDetailsContext(_, _, _, _, Some(offset), _) => offset }
  .via(commitOffsetBackToKafkaTopicSourceFlow(commitParallelism))
  .withAttributes(ActorAttributes.withSupervisionStrategy(streamDecider))
  .to(sink)
```

**Filter stuff**

# Simpler version

```
val graph = source
  .collect(CollectSpecificTypeInEnvelope[OrderRequestEvent])
  .filter(keepEventAfterYester
  .via(buildStreamContextFlow)
  .throttle(throttling.maxRequestsPerSecond, throttling.period, throttling.maximumBurst, throttling.throttleMode)
  .mapAsync(mapAsyncParallelism)(CallOrderDetailsPaoWebService(orderDetailsUtils, redisService, paoSettings))
  .mapConcat[(OrderDetailsContext, Option[TicketScope])](pairEventsToContextWithOptionalOffsetTupled)
  .via(buildKafkaProducerMessageFlow)
  .via(producerFlow)
  .map(elem => elem.message.passThrough)
  // -- Extract only OrderByTrainContext with Some committable offset
  .collect { case OrderDetailsContext(_, _, _, _, Some(offset), _) => offset }
  .via(commitOffsetBackToKafkaTopicSourceFlow(commitParallelism))
  .withAttributes(ActorAttributes.withSupervisionStrategy(streamDecider))
  .to(sink)
```

**Add some context ( correlationId, offset…)**

# Simpler version

```
val graph = source
  .collect(CollectSpecificTypeInEnvelope[OrderRequestEvent])
  .filter(keepEventAfterYesterdaySessionId(_, MARKER, skipPastDays))
  .via(buildStreamContextFlow)
  .throttle(throttling.maxRequest                    d, throttling.maximumBurst, throttling.throttleMode)
  .mapAsync(mapAsyncParallelism)(CallOrderDetailsPaoWebService(orderDetailsUtils, redisService, paoSettings))
  .mapConcat[(OrderDetailsContext, Option[TicketScope])](pairEventsToContextWithOptionalOffsetTupled)
  .via(buildKafkaProducerMessageFlow)
  .via(producerFlow)
  .map(elem => elem.message.passThrough)
  // -- Extract only OrderByTrainContext with Some committable offset
  .collect { case OrderDetailsContext(_, _, _, _, Some(offset), _) => offset }
  .via(commitOffsetBackToKafkaTopicSourceFlow(commitParallelism))
  .withAttributes(ActorAttributes.withSupervisionStrategy(streamDecider))
  .to(sink)
```

**Speeeeeed limit!**

# Simpler version

```
val graph = source
  .collect(CollectSpecificTypeInEnvelope[OrderRequestEvent])
  .filter(keepEventAfterYesterdaySessionId(_, MARKER, skipPastDays))
  .via(buildStreamContextFlow)
  .throttle(throttling.maxRequest
  .mapAsync(mapAsyncParallelism)(
  .mapConcat[(OrderDetailsContext, Option[TicketScope])](pairEventsToContextWithOptionalOffsetTupled)
  .via(buildKafkaProducerMessageFlow)
  .via(producerFlow)
  .map(elem => elem.message.passThrough)
  // -- Extract only OrderByTrainContext with Some committable offset
  .collect { case OrderDetailsContext(_, _, _, _, Some(offset), _) => offset }
  .via(commitOffsetBackToKafkaTopicSourceFlow(commitParallelism))
  .withAttributes(ActorAttributes.withSupervisionStrategy(streamDecider))
  .to(sink)
```

**Call webservice in // but don't forget the order please!**

# Simpler version

```
val graph = source
  .collect(CollectSpecificTypeInEnvelope[OrderRequestEvent])
  .filter(keepEventAfterYesterdaySessionId(_, MARKER, skipPastDays))
  .via(buildStreamContextFlow)
  .throttle(throttling.maxRequestsPerSecond, throttling.period, throttling.maximumBurst, throttling.throttleMode)
  .mapAsync(mapAsyncParallelism)(CallOrderDetailsPaoWebService(orderDetailsUtils, redisService, paoSettings))
  .mapConcat[(OrderDetailsContext
  .via(buildKafkaProducerMessageFlow)
  .via(producerFlow)
  .map(elem => elem.message.passThrough)
  // -- Extract only OrderByTrainContext with Some committable offset
  .collect { case OrderDetailsContext(_, _, _, _, Some(offset), _) => offset }
  .via(commitOffsetBackToKafkaTopicSourceFlow(commitParallelism))
  .withAttributes(ActorAttributes.withSupervisionStrategy(streamDecider))
  .to(sink)
```

**Ok, this one's tricky! We'll talk about sashimis just after**

# Simpler version

```
val graph = source
  .collect(CollectSpecificTypeInEnvelope[OrderRequestEvent])
  .filter(keepEventAfterYesterdaySessionId(_, MARKER, skipPastDays))
  .via(buildStreamContextFlow)
  .throttle(throttling.maxRequestsPerSecond, throttling.period, throttling.maximumBurst, throttling.throttleMode)
  .mapAsync(mapAsyncParallelism)(CallOrderDetailsPaoWebService(orderDetailsUtils, redisService, paoSettings))
  .mapConcat[(OrderDetailsContext, Option[TicketScope])](pairEventsToContextWithOptionalOffsetTupled)
  .via(buildKafkaProducerMessageFlow)
  .via(producerFlow)
  .map(elem => elem.message.passThrough)
  // -- Extract only OrderByTrainContext with Some committable offset
  .collect { case OrderDetailsContext(_, _, _, _, Some(offset), _) => offset }
  .via(commitOffsetBackToKafkaTopicSourceFlow(commitParallelism))
  .withAttributes(ActorAttributes.withSupervisionStrategy(streamDecider))
  .to(sink)
```

**Write to kafka**

# Simpler version

```
val graph = source
  .collect(CollectSpecificTypeInEnvelope[OrderRequestEvent])
  .filter(keepEventAfterYesterdaySessionId(_, MARKER, skipPastDays))
  .via(buildStreamContextFlow)
  .throttle(throttling.maxRequestsPerSecond, throttling.period, throttling.maximumBurst, throttling.throttleMode)
  .mapAsync(mapAsyncParallelism)(CallOrderDetailsPaoWebService(orderDetailsUtils, redisService, paoSettings))
  .mapConcat[(OrderDetailsContext, Option[TicketScope])](pairEventsToContextWithOptionalOffsetTupled)
  .via(buildKafkaProducerMessageFlow)
  .via(producerFlow)
  .map(elem => elem.message.passThroug
  // -- Extract only OrderByTrainConte
  .collect { case OrderDetailsContext(
  .via(commitOffsetBackToKafkaTopicSou
  .withAttributes(ActorAttributes.withSupervisionStrategy(streamDecider))
  .to(sink)
```

**Just commit the kafka offset if you can but in batching mode!**

# Simpler version

```
val graph = source
  .collect(CollectSpecificTypeInEnvelope[OrderRequestEvent])
  .filter(keepEventAfterYesterdaySessionId(_, MARKER, skipPastDays))
  .via(buildStreamContextFlow)
  .throttle(throttling.maxRequestsPerSecond, throttling.period, throttling.maximumBurst, throttling.throttleMode)
  .mapAsync(mapAsyncParallelism)(CallOrderDetailsPaoWebService(orderDetailsUtils, redisService, paoSettings))
  .mapConcat[(OrderDetailsContext, Option[TicketScope])](pairEventsToContextWithOptionalOffsetTupled)
  .via(buildKafkaProducerMessageFlow)
  .via(producerFlow)
  .map(elem => elem.message.passThrough)
  // -- Extract only OrderByTrainContext with Some committable offset
  .collect { case OrderDetailsContext(_, _, _, _, Some(offset), _) => offset }
  .via(commitOffsetBackToKafkaTopicSourceFlow(commitParallelism))
  .withAttributes(ActorAttributes.withSupervisio
  .to(sink)
```

**Don’t forget the “in case of emergency” part…**

# Simpler version

```
val graph = source
  .collect(CollectSpecificTypeInEnvelope[OrderRequestEvent])
  .filter(keepEventAfterYesterdaySessionId(_, MARKER, skipPastDays))
  .via(buildStreamContextFlow)
  .throttle(throttling.maxRequestsPerSecond, throttling.period, throttling.maximumBurst, throttling.throttleMode)
  .mapAsync(mapAsyncParallelism)(CallOrderDetailsPaoWebService(orderDetailsUtils, redisService, paoSettings))
  .mapConcat[(OrderDetailsContext, Option[TicketScope])](pairEventsToContextWithOptionalOffsetTupled)
  .via(buildKafkaProducerMessageFlow)
  .via(producerFlow)
  .map(elem => elem.message.passThrough)
  // -- Extract only OrderByTrainContext with Some committable offset
  .collect { case OrderDetailsContext(_, _, _, _, Some(offset), _) => offset }
  .via(commitOffsetBackToKafkaTopicSourceFlow(commitParallelism))
  .withAttributes(ActorAttributes.withSupervisionStrategy(streamDecider))
  .to(sink)
```

**… and go nowhere!**

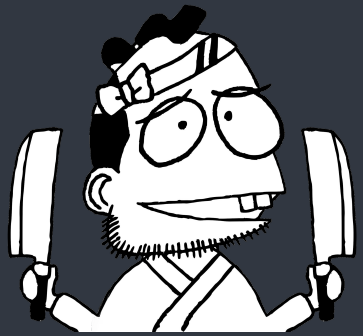
# Just linear graph logic is enough !

# Get salmons from the river



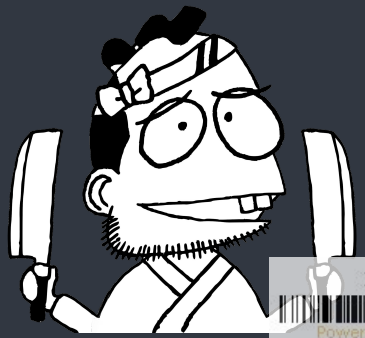

# My Chef / Webservices



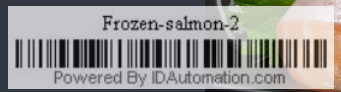
# Tracing SASHIMIS...



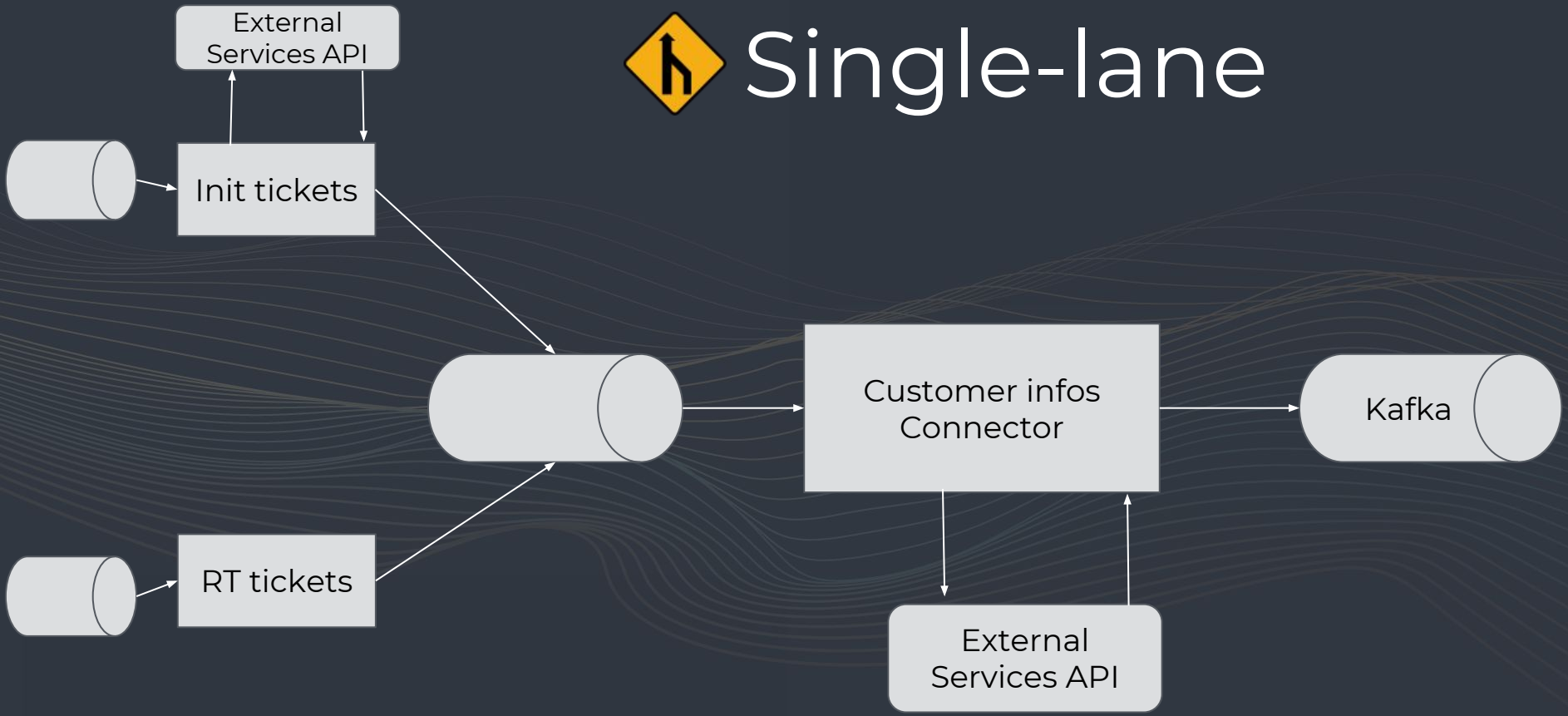







# Speed up “real time”

# Single-lane



# Rush hour

Initialization processing sends many many many more events than the so-called “Real Time” one

“Real time” events are lost in the transportation flow

Image by Golda Falk from Pixabay

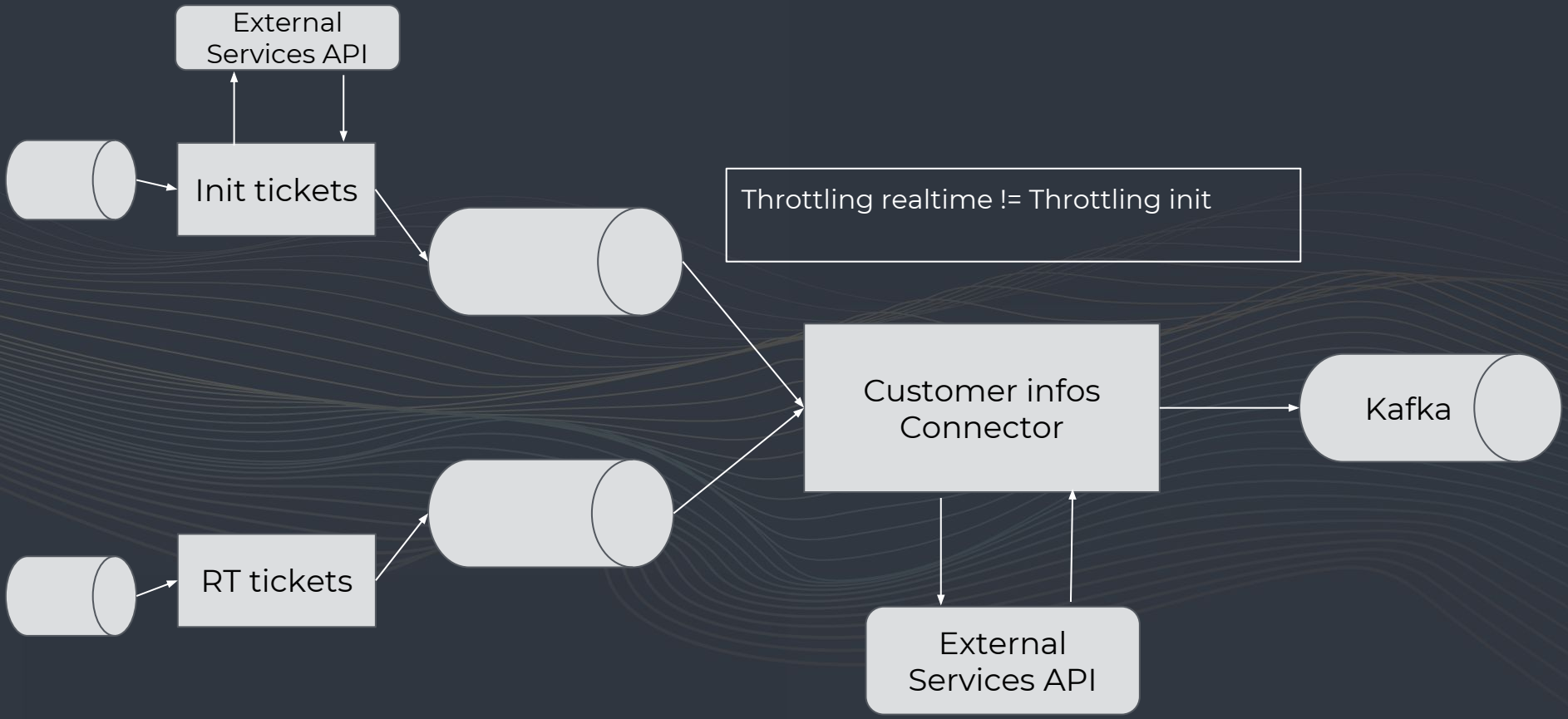
# Rush hour

Initialization processing sends many many many more events than the so-called “Real Time” one

“Real time” events are lost in the transportation flow

And these critical events are late… really late…

Still waiting for my train | Henry Burrows | CC2



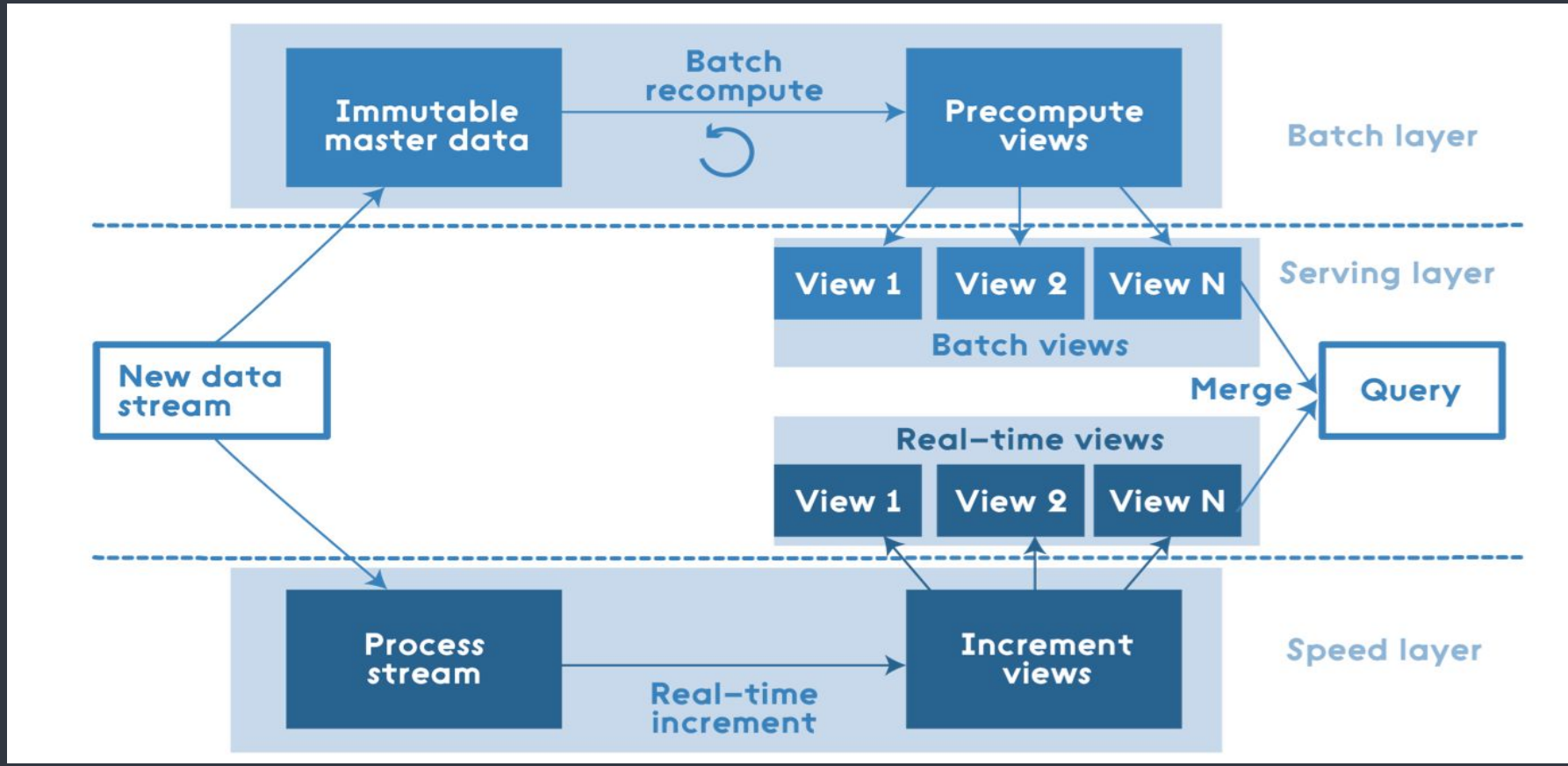
# So basically, we've just "created" a bus lane...

# Into the wild code

```
trait EmeraudeDiscovery extends ServiceDiscovery {
  def configuration: Configuration

  def redisService: RedisService

  override def zookeeperConnect: String = configuration.getZookeeperConnectString

  override def partnerName: String = Partner.EMR.name()

  override def serverPort: Int = configuration.getServerPort

  override def createStreams(): Seq[StreamDiscovery with StreamControl] = {
    // -- Register your Stream here
    val passengersInfoDispatchStreamGraph = new PassengersInfoDispatchStreamGraph()
    val passengerDataInitStreamGraph = new PassengerDataStreamGraph("INIT", redisService, "streams.passenger-data-init")
    val passengerDataRealTimeStreamGraph = new PassengerDataStreamGraph("REAL_TIME", redisService, "streams.passenger-data-real-time")

    Seq(
      passengersInfoDispatchStreamGraph,
      passengerDataInitStreamGraph,
      passengerDataRealTimeStreamGraph
    )
  }
```

**It’s the same code but with a different configuration! ( throttling, topics…)**

# Look like Lambda Architecture



# Operating streams

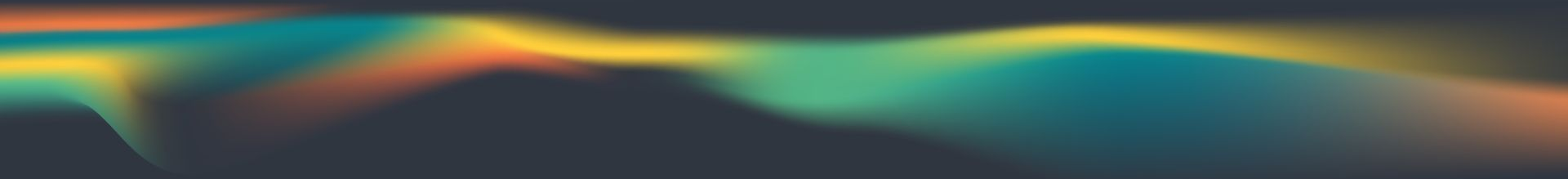
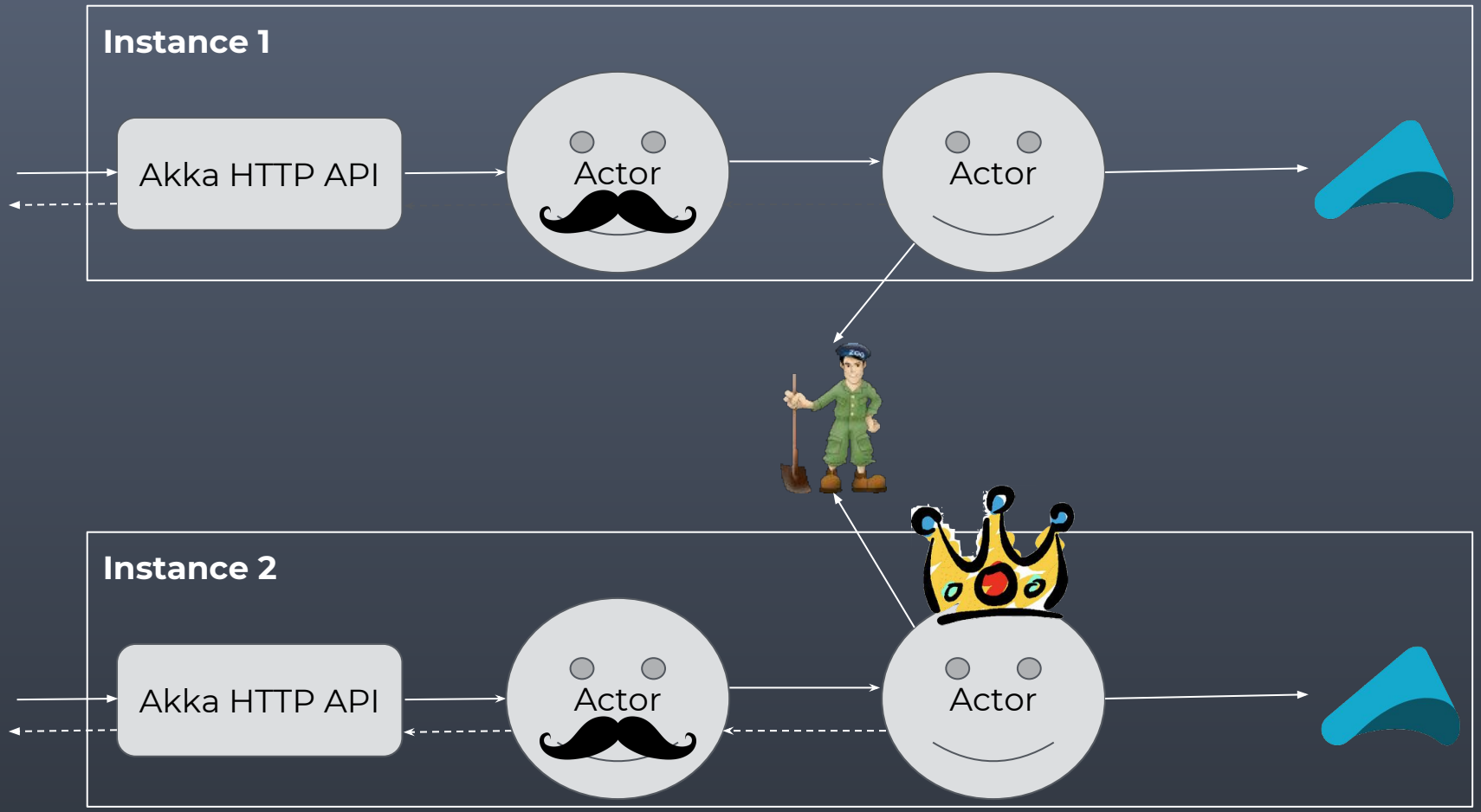


Photo by Jonathan Wheeler on Unsplash

# Operating streams

- Throttling is static in akka-stream : when your graph is built and running, you cannot change it dynamically!
- Sometimes it may start, stop, restart…
- Or the stream is a batch



Rex across the fields | Reiterlied CC BY-NC-SA 2.0

# Feedback

# LEARNING

- Try to keep simple graph (as easy to read and maintain)
- Use Akka DSL only when necessary
- Avoid clutter components when simple programming language is enough

# LEARNING

- Use metadata on your message to measure, identify, correlate…
- Measure, measure, measure!
- Beware of lag

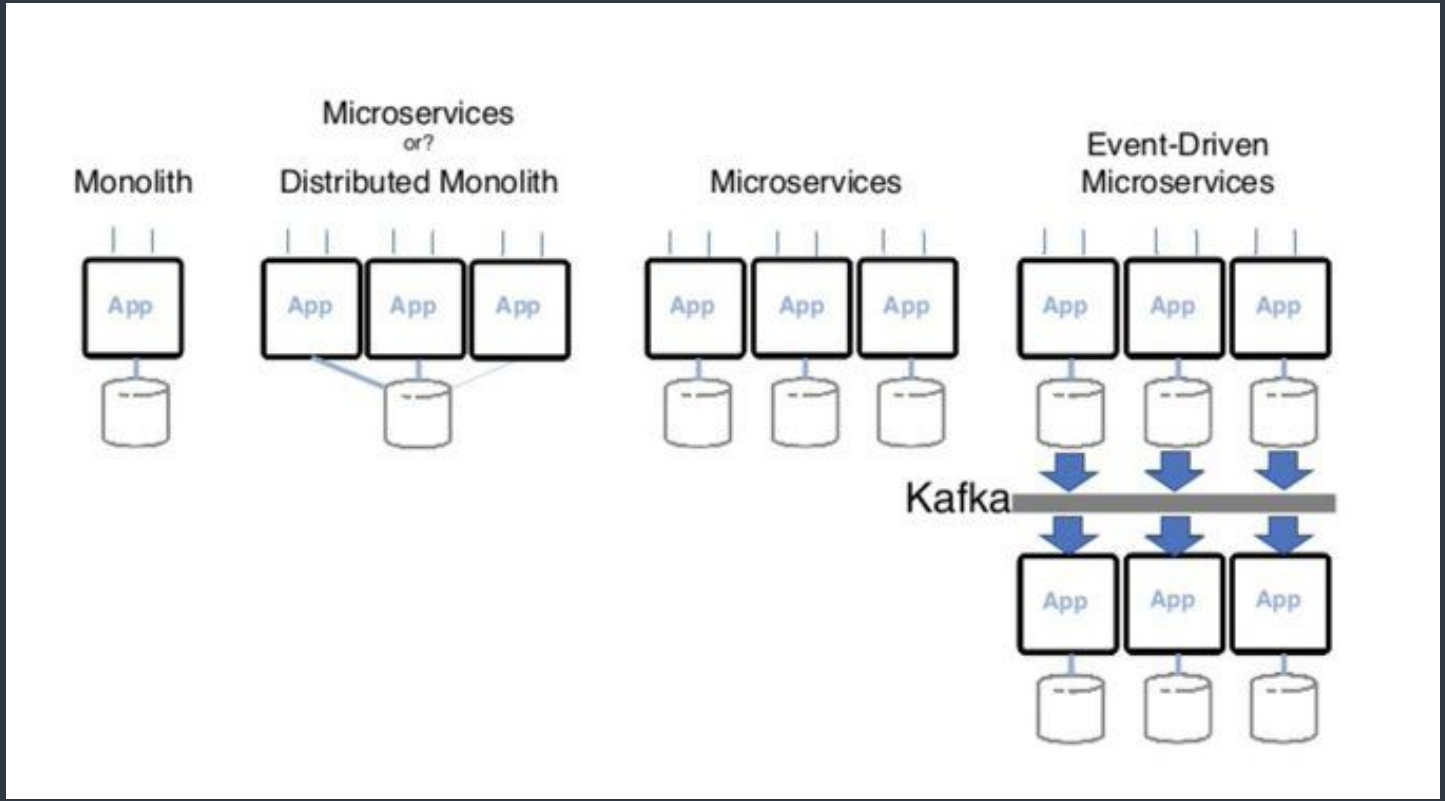
# LEARNING

Sometimes backpressure is your friend and sometimes it’s not...

# LEARNING

Sometimes backpressure is your friend and sometimes it’s not...

# Architecture Evolution




Source: https://twitter.com/cdavisafc/status/1105572564712210432

# CONCEPTS THAT WILL HELP US

- Speed limit => Throttle 
- Keep up with jammed services => Backpressure 
- Address near real time concern => Event-based Architecture 

# Thank you!

# ANNEXES

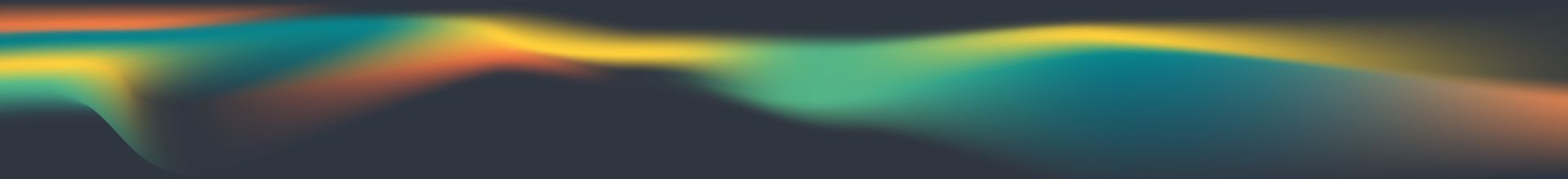

https://nsphung.netlify.com/csm-backpressure/

# “AKKA FTW” -Jonas Boner

# Akka* scales?

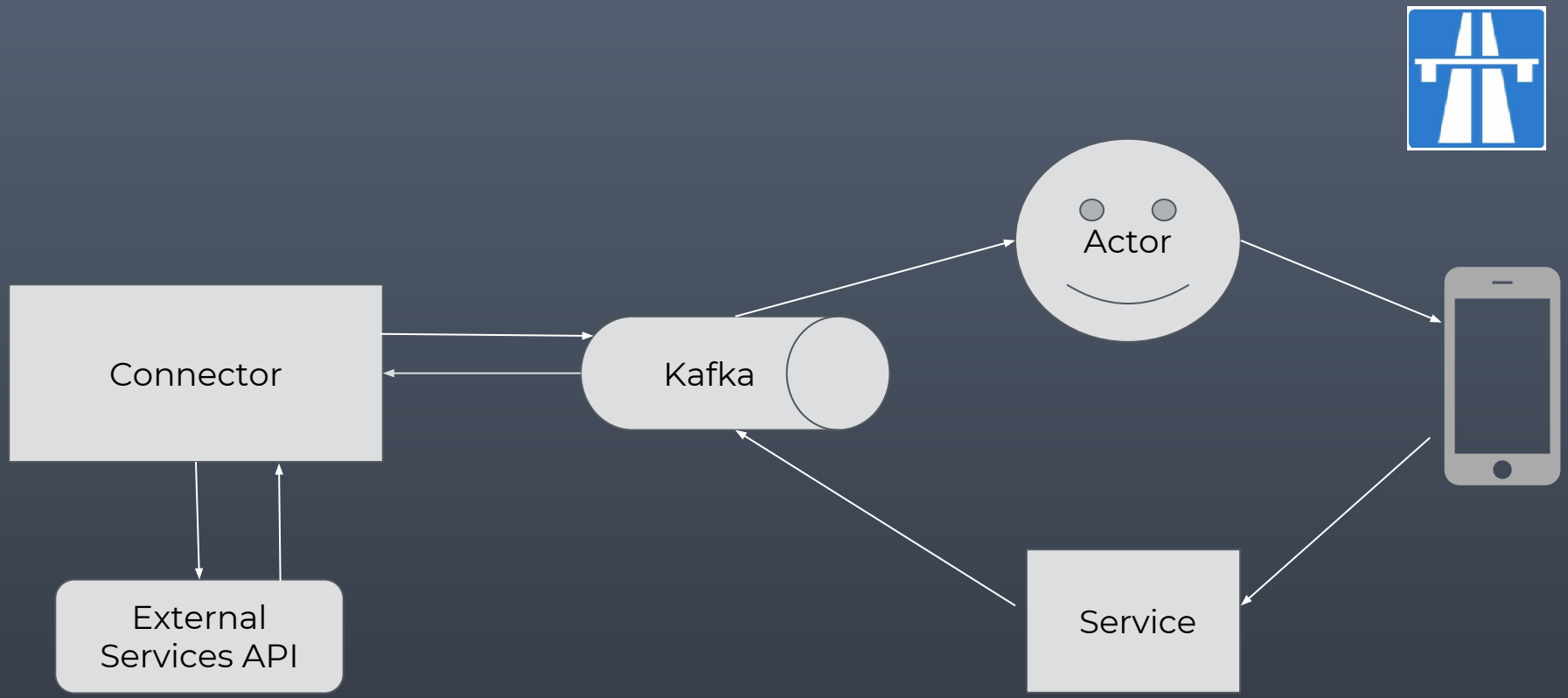

Lightbend
@lightbend
Following
How to scale? Here you go:
With over 125 million players, and supporting over 8 million *concurrent gaming sessions*, we are really happy to learn that #Fortnite is running #Akka under the hood!
Dragos Manolescu @hysteresis
Epic's #fortnighte running on AWS and "powered by Java, #Akka and even Go" #reInvent
9:24 AM - 27 Nov 2018
70 Retweets 151 Likes
1
70
151

# Checkmate with Actors



# Tickets Check

- Handle events from partners and expose them to a mobile front-end
- Receive events from a mobile front-end and forward them to a partner
- All these events may affect the same data
- Everything has an end!



# LEARNING

- Separate your reads and your writes
- Sometimes RAM is enough (and sometimes not!)
- Your partners may fail but you should keep the “closest to reality” internal state
- Actors are fine for keeping “no-lifetime” data ( kafka is here for that ) but think about boot time and possible inconsistencies between instances