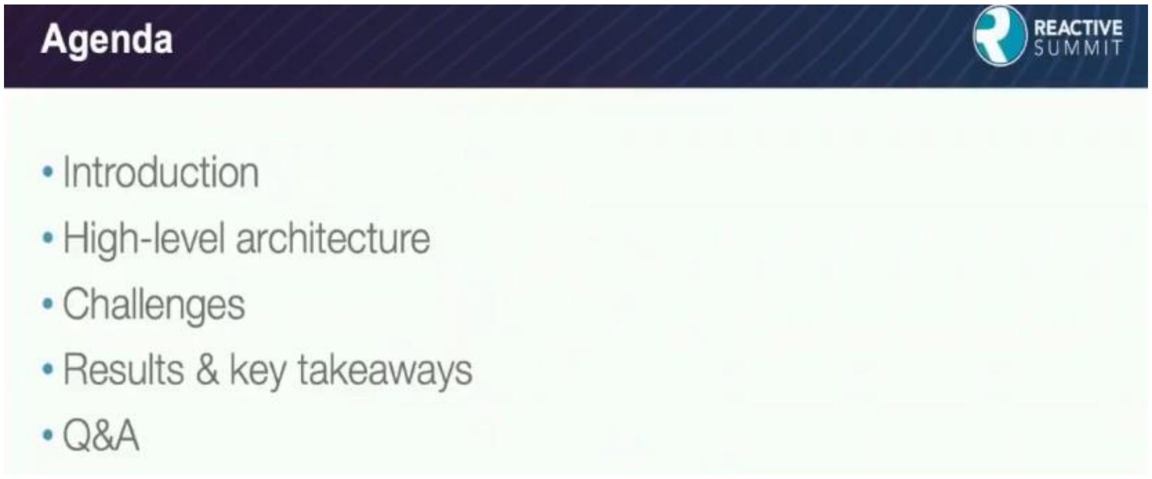
A Reactive Architecture Approach for a Tax Information System Modernization

Sergio Maurenzi, Peperina Software

#reactivesummit

Leading internet players like Amazon, Google, Netflix and others are raising expectations for the user experience of their applications. On the other hand, paying taxes is not inherently a pleasant experience for taxpayers. Therefore, designing a tax information system that is responsive, resilient and scalable is a duty rather than an option. Sergio Maurenzi will present his experience in modernizing a Tax information system through statefull services using technologies and architecture patterns such as Event-driven, Akka Actor Model, Akka Streams, Event Sourcing, CQRS, and others to improve the taxpayer experience.



Agenda

REACTIVE SUMMIT

- Introduction
- High-level architecture
- Challenges
- Results & key takeaways
- Q&A

Introduction, Context

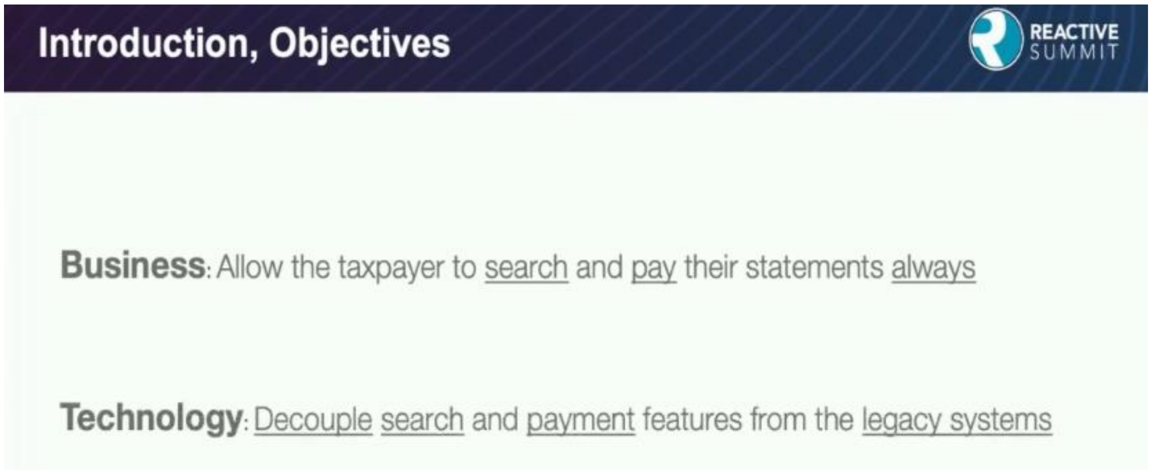
REACTIVE SUMMIT

Córdoba, Argentina

DGR, State Tax Agency

DIRECCIÓN GENERAL DE RENTAS KOLEKTOR

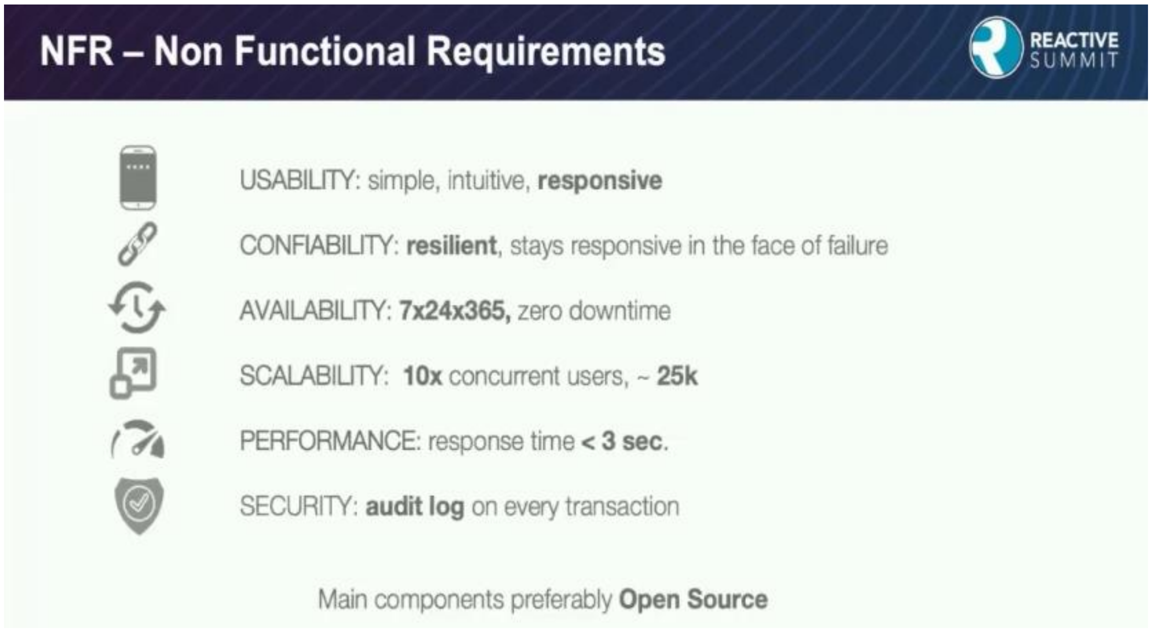
3.5+ Million taxpayers
80+ Million liabilities / year



Introduction, Objectives

Business: Allow the taxpayer to search and pay their statements always

Technology: Decouple search and payment features from the legacy systems



NFR – Non Functional Requirements

USABILITY: simple, intuitive, **responsive**

CONFIABILITY: **resilient**, stays responsive in the face of failure

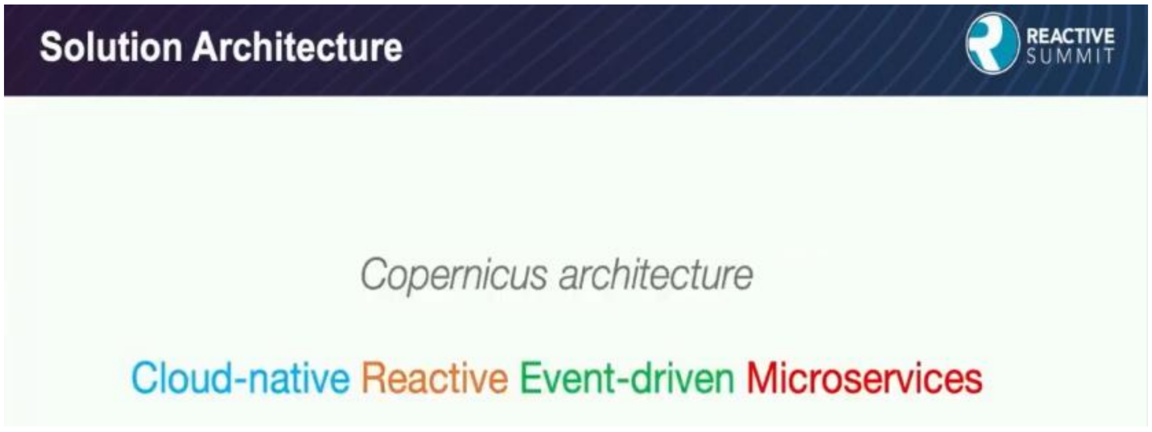
AVAILABILITY: **7x24x365,** zero downtime

SCALABILITY: **10x** concurrent users, ~ **25k**

PERFORMANCE: response time **< 3 sec**.

SECURITY: **audit log** on every transaction

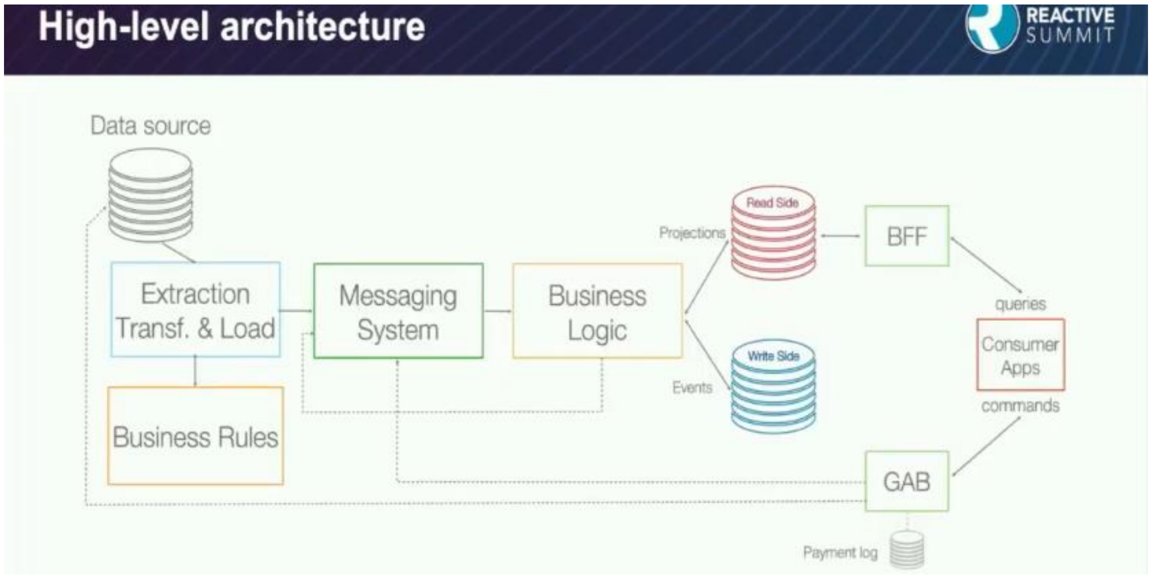
Main components preferably **Open Source**



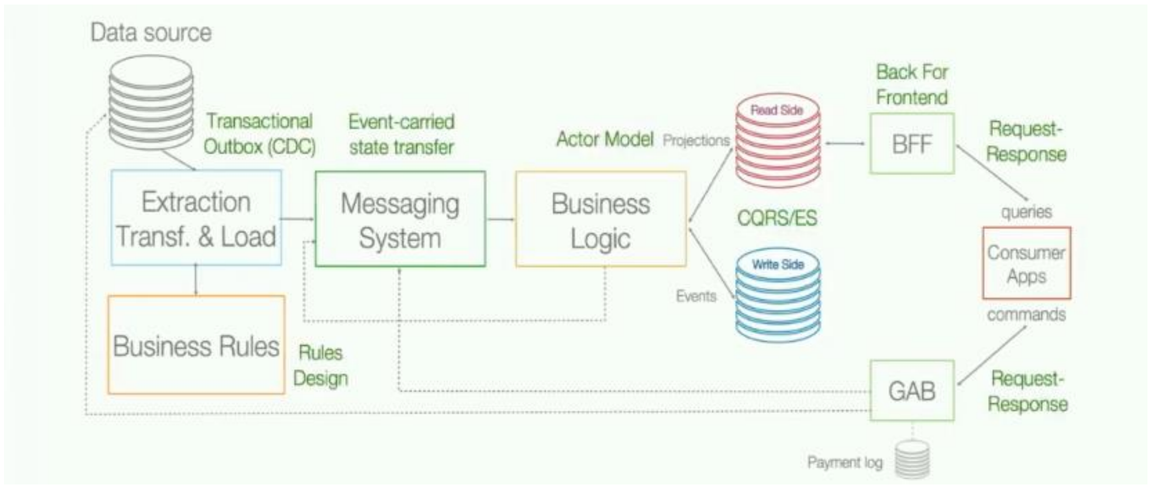
Solution Architecture

Copernicus architecture

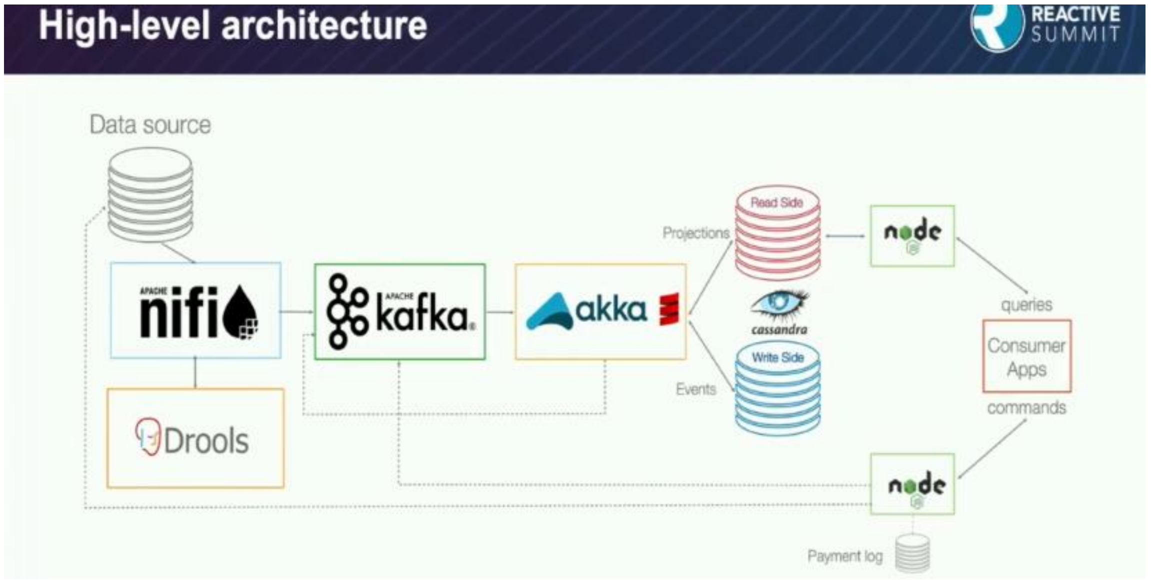
Cloud-native Reactive Event-driven Microservices



This architecture consumes events from the legacy system in a pipeline via CDC that has a trigger that copies the change record into a messaging system.

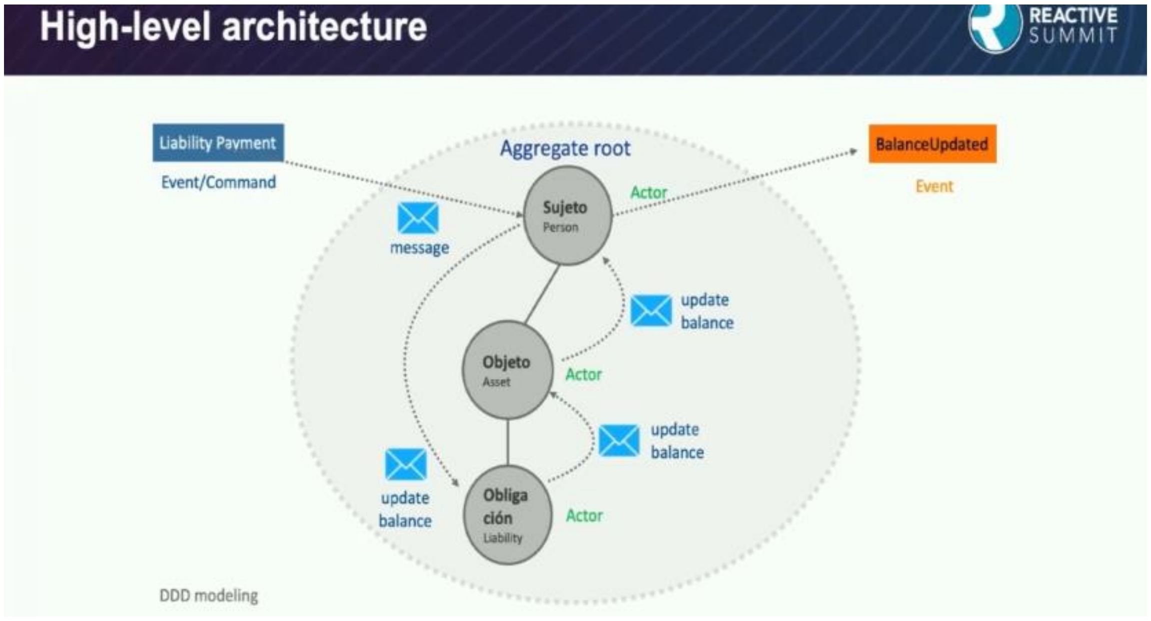


Biz rules can be changed in a static DML file in an eventual consistency system that provides a read model for the BFF.



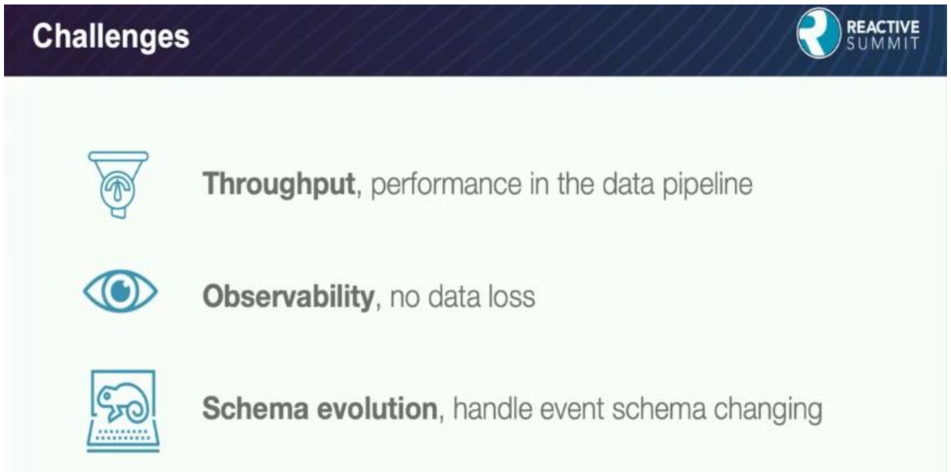
High-level architecture

REACTIVE SUMMIT



Challenges

REACTIVE SUMMIT



Throughput, performance in the data pipeline

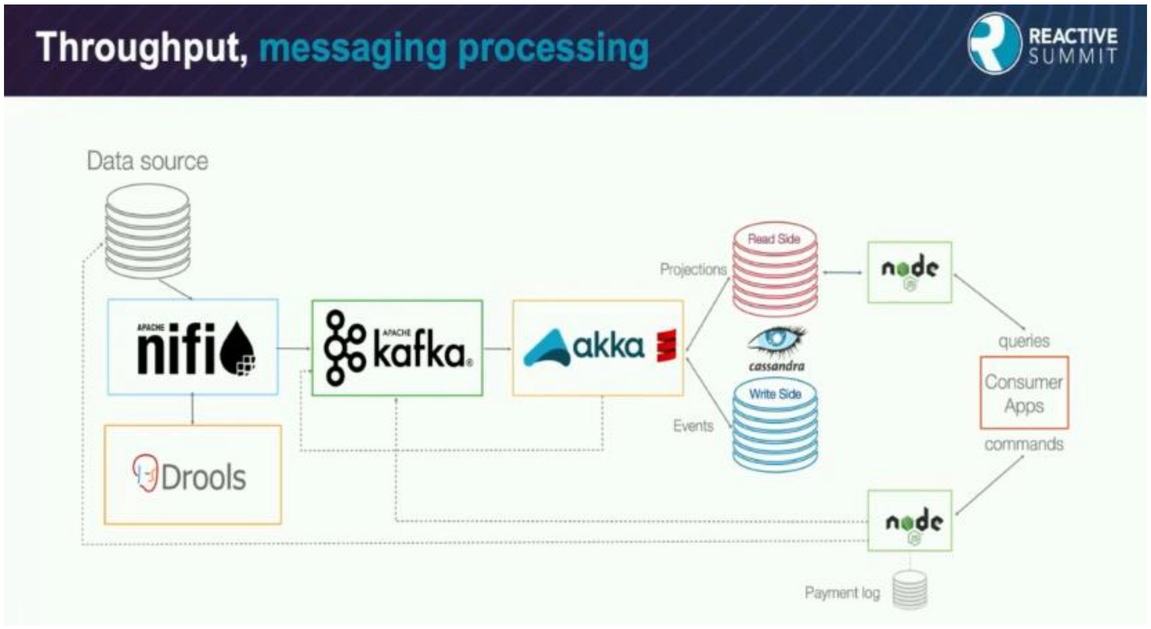
Observability, no data loss

Schema evolution, handle event schema changing

Challenge 1, Throughput

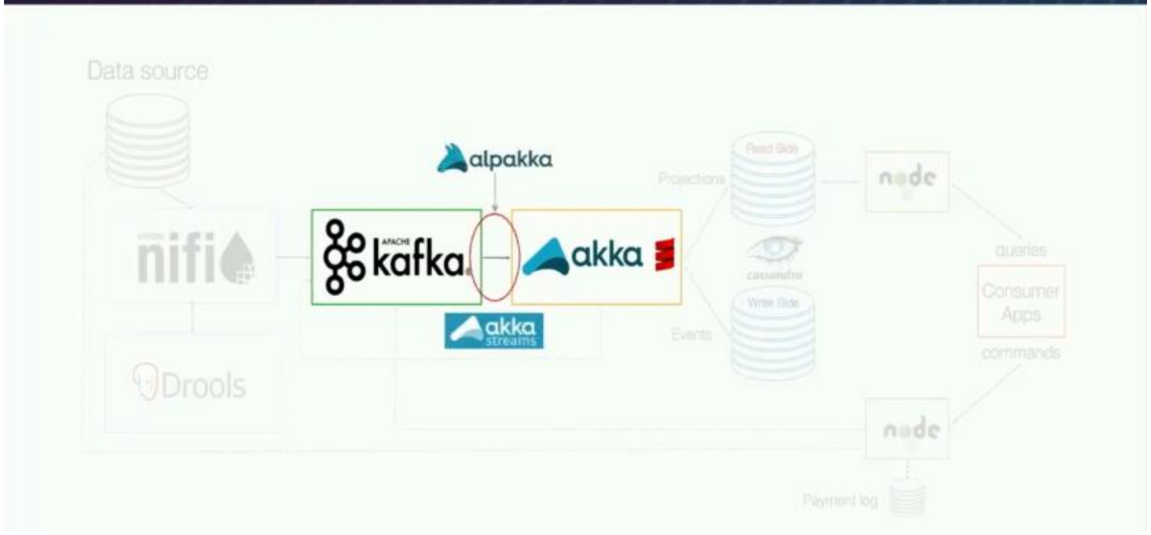
Throughput, messaging processing

REACTIVE SUMMIT



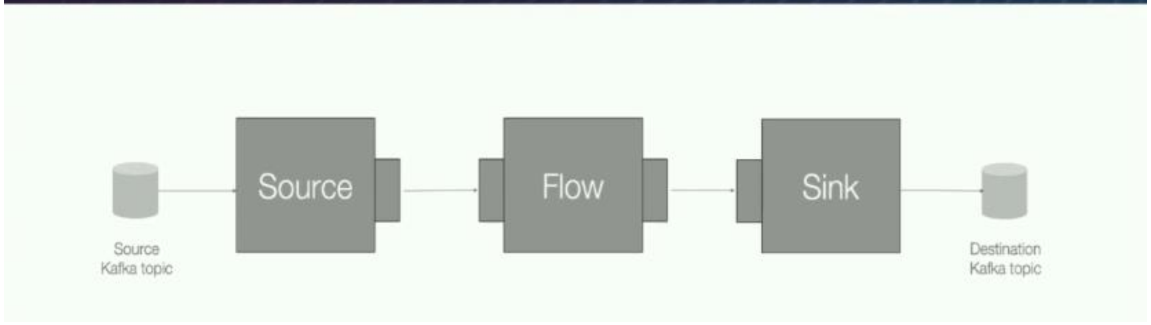
Throughput, messaging processing

REACTIVE SUMMIT



Throughput, messaging processing

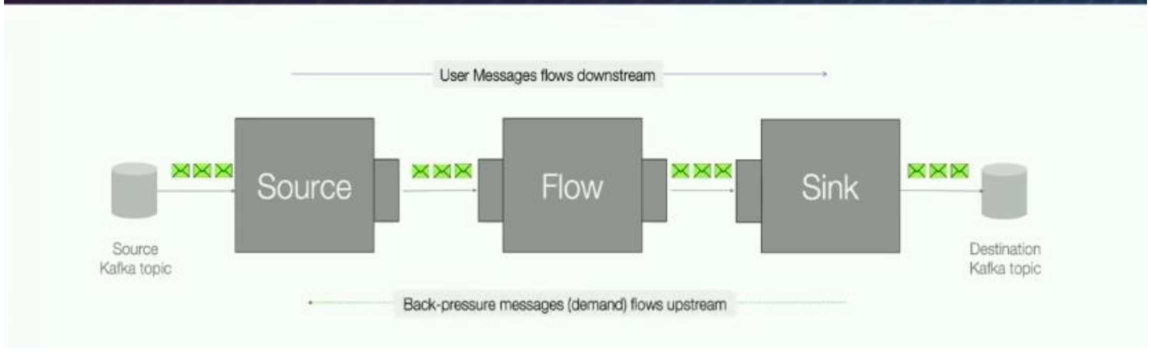
REACTIVE SUMMIT



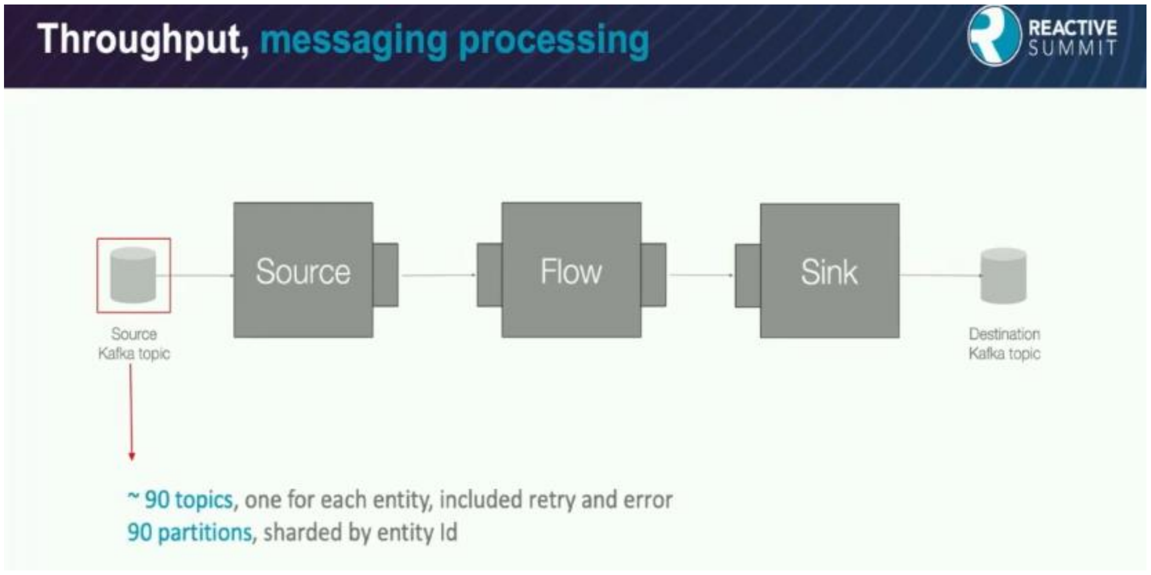
Above is a basic Akka stream pipeline with connectors

Throughput, messaging processing

REACTIVE SUMMIT

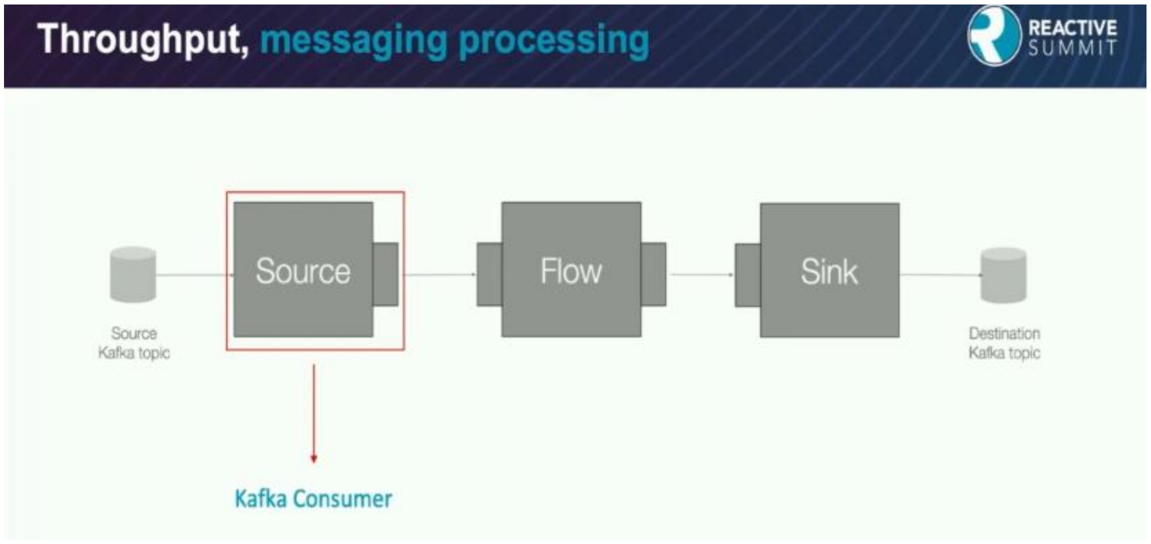

User Messages flows downstream
Source
Flow
Sink
Source Kafka topic
Destination Kafka topic
Back-pressure messages (demand) flows upstream

Throughput, messaging processing



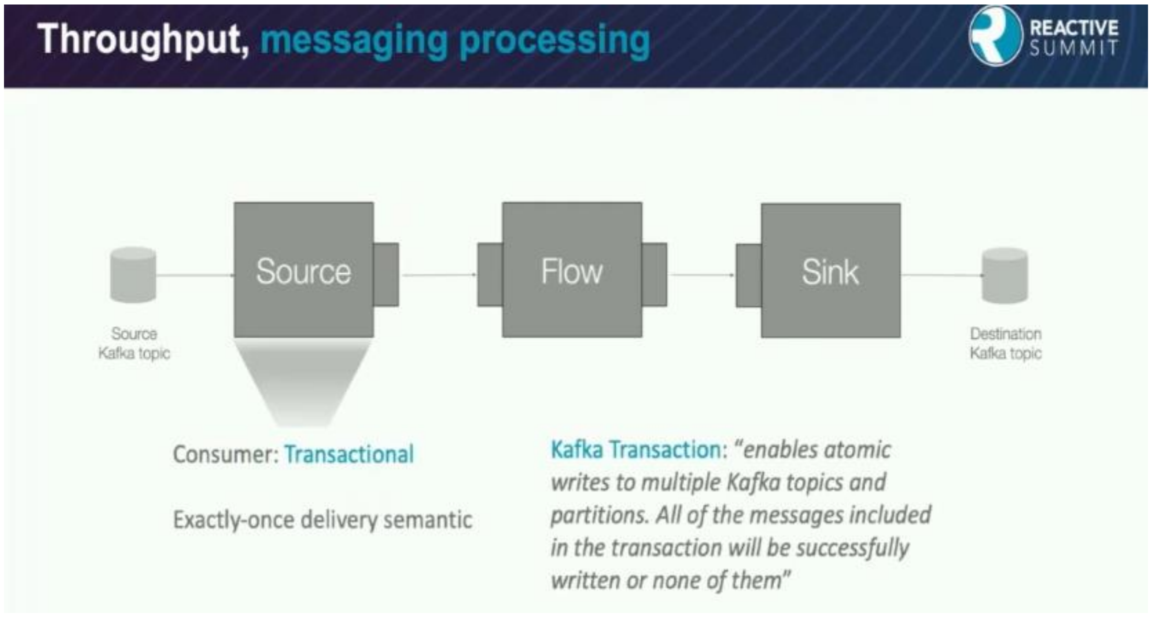
~ 90 topics, one for each entity, included retry and error
90 partitions, sharded by entity Id

Throughput, messaging processing



Kafka Consumer

Throughput, messaging processing



Consumer: Transactional

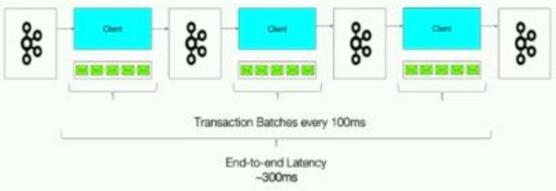
Exactly-once delivery semantic

Kafka Transaction: *"enables atomic writes to multiple Kafka topics and partitions. All of the messages included in the transaction will be successfully written or none of them"*

Throughput, messaging processing

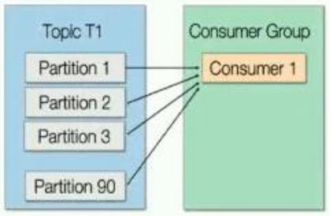
REACTIVE SUMMIT

Issue #1: Transaction pipeline Latency[1]



Issue #2: Partition unaware[2]

Offsets handling	Partition aware	Shared Consumer	Factory method
Transactional	No	No	Transactional.source
Transactional	No	No	Transactional.sourceWithOffsetContext

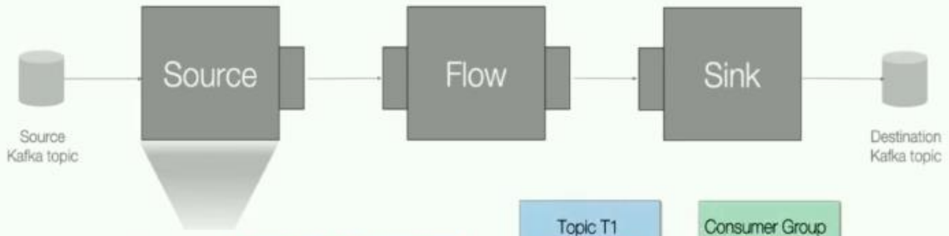


Source:
1. Sean Glover, Fast Data pipelines with Akka Streams and Alpakka, https://youtu.be/ib1oYAS2dh0
2. Akka Documentation: https://doc.akka.io/docs/alpakka-kafka/current/consumer.html

This consumer is not partition-aware and can't manage the partitions effectively

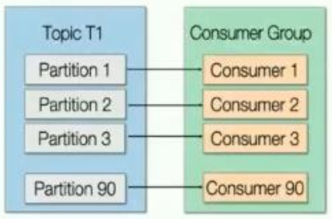
Throughput, messaging processing

REACTIVE SUMMIT

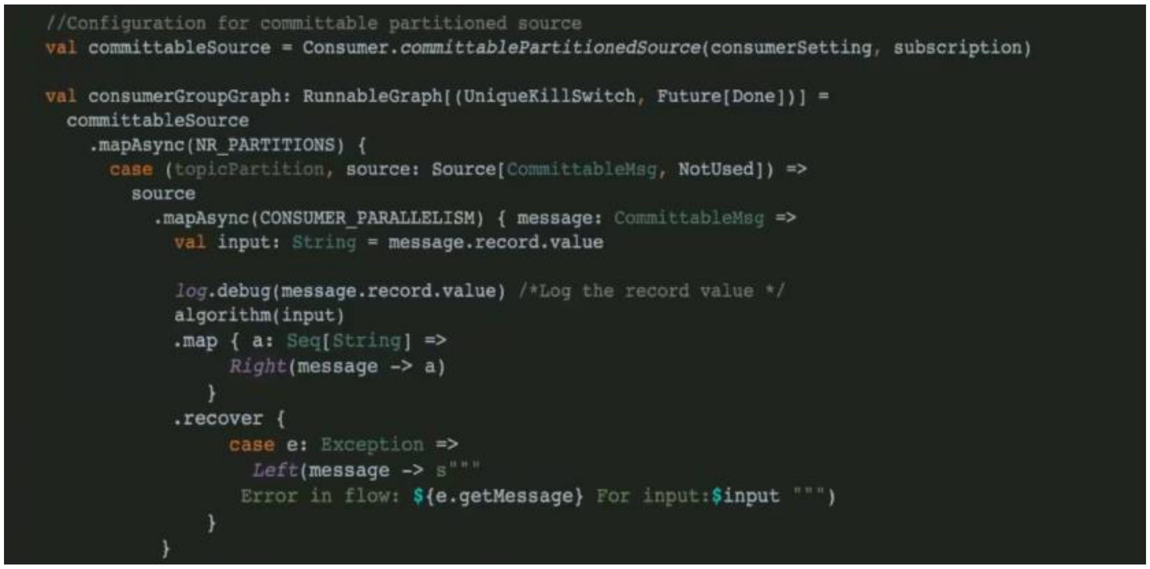


Consumer: Committable Partitioned

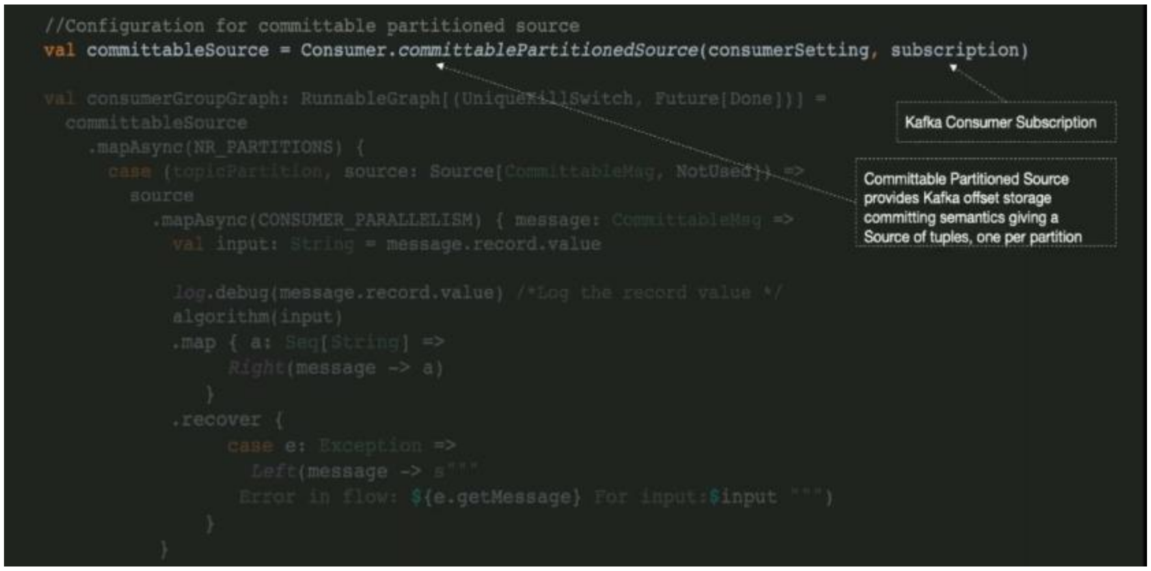
At-Least once delivery semantic



We now have 1 consumer per partition

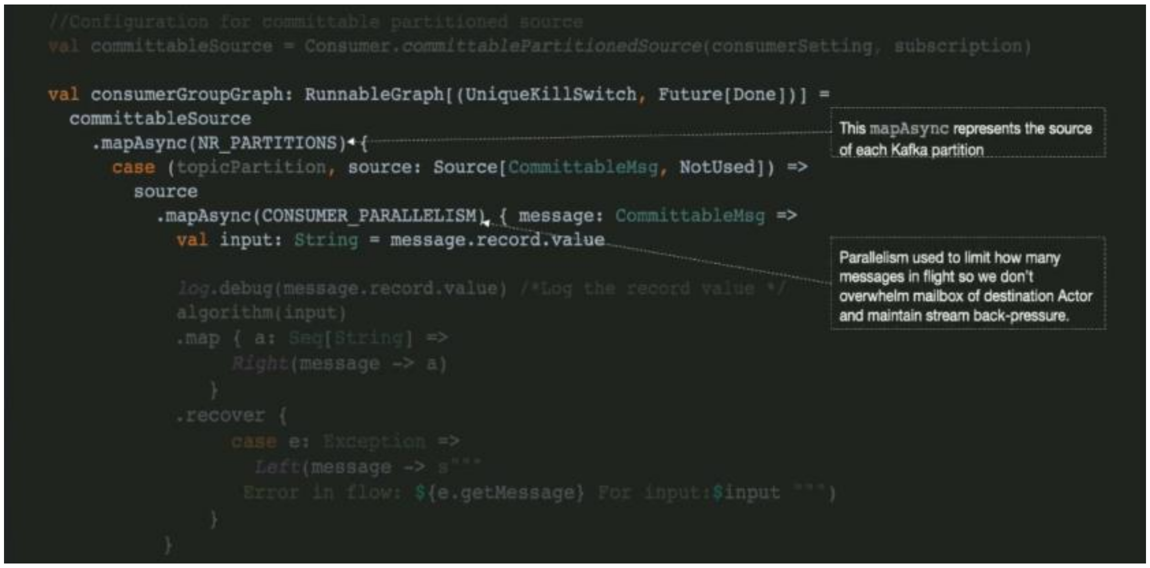


```
//Configuration for committable partitioned source
val committableSource = Consumer.committablePartitionedSource(consumerSetting, subscription)

val consumerGroupGraph: RunnableGraph[(UniqueKillSwitch, Future[Done])] =
  committableSource
    .mapAsync(NR_PARTITIONS) {
      case (topicPartition, source: Source[CommittableMsg, NotUsed]) =>
        source
          .mapAsync(CONSUMER_PARALLELISM) { message: CommittableMsg =>
            val input: String = message.record.value

            log.debug(message.record.value) /*Log the record value */
            algorithm(input)
            .map { a: Seq[String] =>
                 Right(message -> a)
              }
            .recover {
                 case e: Exception =>
                   Left(message -> s"""
                  Error in flow: ${e.getMessage} For input:$input """)
              }
          }
```



```
//Configuration for committable partitioned source
val committableSource = Consumer.committablePartitionedSource(consumerSetting, subscription)

val consumerGroupGraph: RunnableGraph[(UniqueKillSwitch, Future[Done])] =
  committableSource
    .mapAsync(NR_PARTITIONS) {
      case (topicPartition, source: Source[CommittableMsg, NotUsed]) =>
        source
          .mapAsync(CONSUMER_PARALLELISM) { message: CommittableMsg =>
            val input: String = message.record.value

            log.debug(message.record.value) /*Log the record value */
            algorithm(input)
            .map { a: Seq[String] =>
                 Right(message -> a)
              }
            .recover {
                 case e: Exception =>
                   Left(message -> s"""
                  Error in flow: ${e.getMessage} For input:$input """)
              }
          }
```

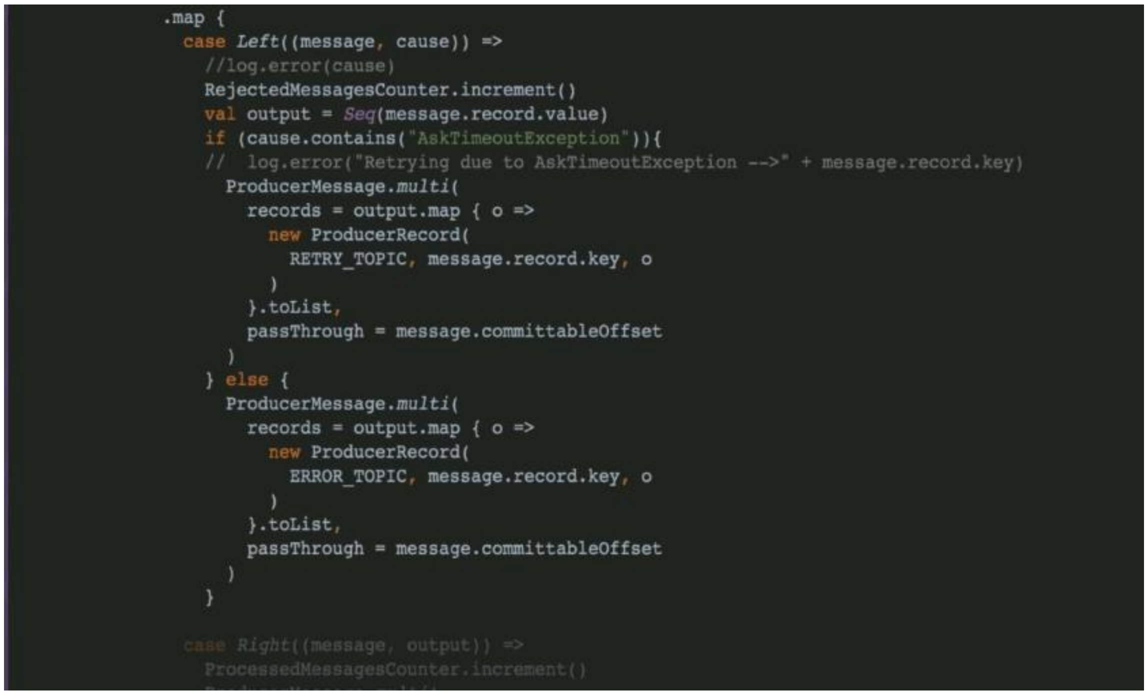
Kafka Consumer Subscription
Committable Partitioned Source provides Kafka offset storage committing semantics giving a Source of tuples, one per partition



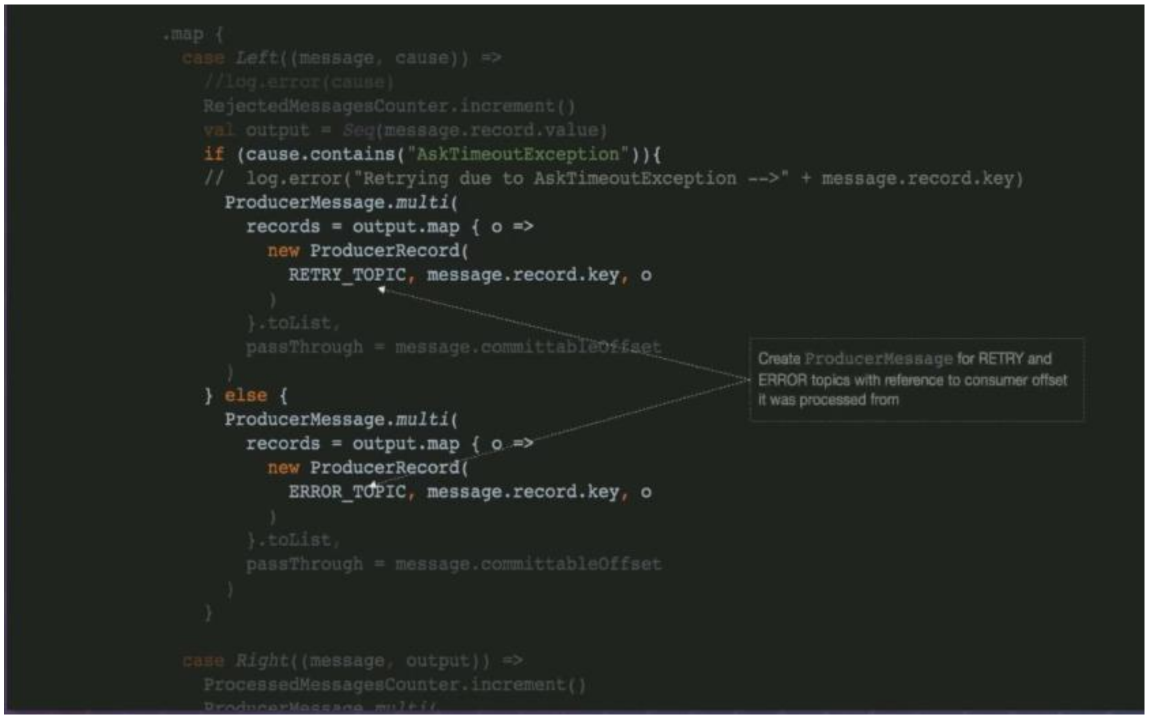
```
//Configuration for committable partitioned source
val committableSource = Consumer.committablePartitionedSource(consumerSetting, subscription)

val consumerGroupGraph: RunnableGraph[(UniqueKillSwitch, Future[Done])] =
  committableSource
    .mapAsync(NR_PARTITIONS) {
      case (topicPartition, source: Source[CommittableMsg, NotUsed]) =>
        source
          .mapAsync(CONSUMER_PARALLELISM) { message: CommittableMsg =>
            val input: String = message.record.value

            log.debug(message.record.value) /*Log the record value */
            algorithm(input)
            .map { a: Seq[String] =>
                 Right(message -> a)
              }
            .recover {
                 case e: Exception =>
                   Left(message -> s"""
                  Error in flow: ${e.getMessage} For input:$input """)
              }
          }
```

This mapAsync represents the source of each Kafka partition
Parallelism used to limit how many messages in flight so we don't overwhelm mailbox of destination Actor and maintain stream back-pressure.

This makes sure things are done in order

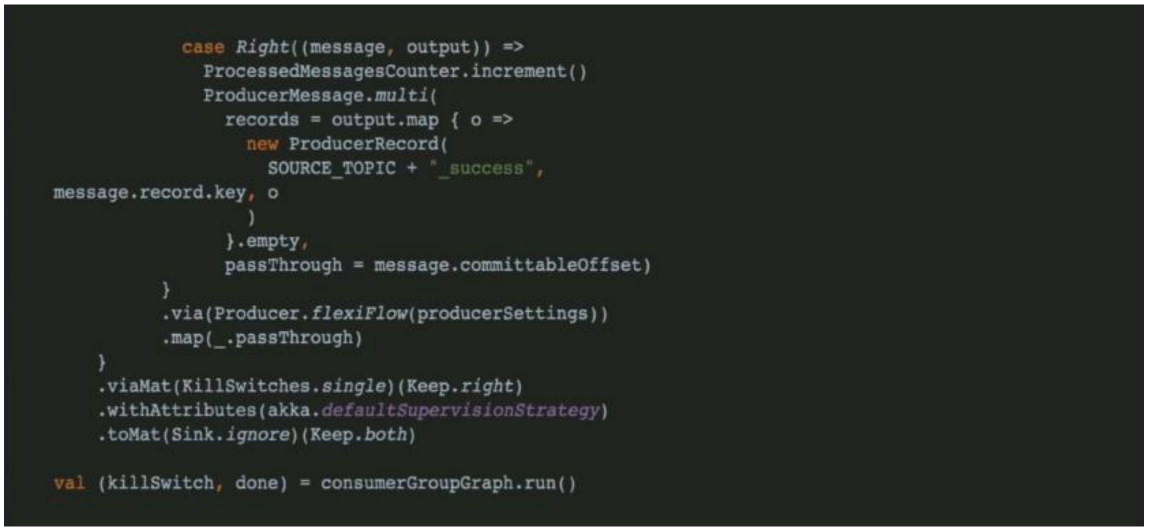


```
.map {
  case Left((message, cause)) =>
    //log.error(cause)
    RejectedMessagesCounter.increment()
    val output = Seq(message.record.value)
    if (cause.contains("AskTimeoutException")){
    //  log.error("Retrying due to AskTimeoutException -->" + message.record.key)
      ProducerMessage.multi(
        records = output.map { o =>
          new ProducerRecord(
            RETRY_TOPIC, message.record.key, o
          )
        }.toList,
        passThrough = message.committableOffset
      )
    } else {
      ProducerMessage.multi(
        records = output.map { o =>
          new ProducerRecord(
            ERROR_TOPIC, message.record.key, o
          )
        }.toList,
        passThrough = message.committableOffset
      )
    }

  case Right((message, output)) =>
    ProcessedMessagesCounter.increment()
```

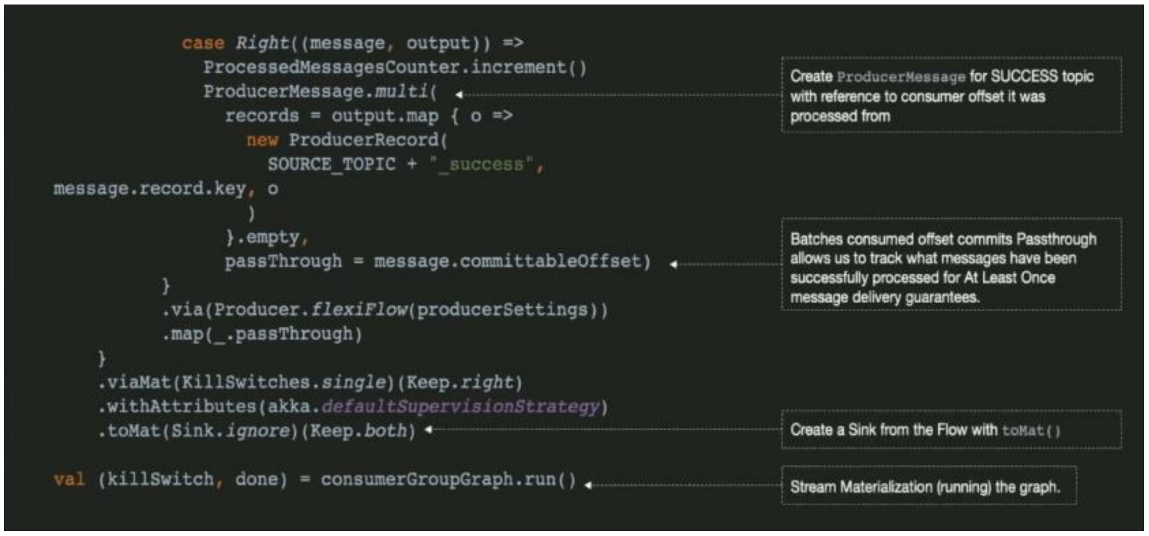


```
.map {
  case Left((message, cause)) =>
    //log.error(cause)
    RejectedMessagesCounter.increment()
    val output = Seq(message.record.value)
    if (cause.contains("AskTimeoutException")){
    //  log.error("Retrying due to AskTimeoutException -->" + message.record.key)
      ProducerMessage.multi(
        records = output.map { o =>
          new ProducerRecord(
            RETRY_TOPIC, message.record.key, o
          )
        }.toList,
        passThrough = message.committableOffset
      )
    } else {
      ProducerMessage.multi(
        records = output.map { o =>
          new ProducerRecord(
            ERROR_TOPIC, message.record.key, o
          )
        }.toList,
        passThrough = message.committableOffset
      )
    }

  case Right((message, output)) =>
    ProcessedMessagesCounter.increment()
    ProducerMessage.multi(
```

Create ProducerMessage for RETRY and ERROR topics with reference to consumer offset it was processed from



```
            case Right((message, output)) =>
              ProcessedMessagesCounter.increment()
              ProducerMessage.multi(
                records = output.map { o =>
                  new ProducerRecord(
                    SOURCE_TOPIC + "_success",
message.record.key, o
                  )
                }.empty,
                passThrough = message.committableOffset)
          }
          .via(Producer.flexiFlow(producerSettings))
          .map(_.passThrough)
    }
    .viaMat(KillSwitches.single)(Keep.right)
    .withAttributes(akka.defaultSupervisionStrategy)
    .toMat(Sink.ignore)(Keep.both)

val (killSwitch, done) = consumerGroupGraph.run()
```



```
            case Right((message, output)) =>
              ProcessedMessagesCounter.increment()
              ProducerMessage.multi(
                records = output.map { o =>
                  new ProducerRecord(
                    SOURCE_TOPIC + "_success",
message.record.key, o
                  )
                }.empty,
                passThrough = message.committableOffset)
          }
          .via(Producer.flexiFlow(producerSettings))
          .map(_.passThrough)
    }
    .viaMat(KillSwitches.single)(Keep.right)
    .withAttributes(akka.defaultSupervisionStrategy)
    .toMat(Sink.ignore)(Keep.both)

val (killSwitch, done) = consumerGroupGraph.run()
```

Create ProducerMessage for SUCCESS topic with reference to consumer offset it was processed from

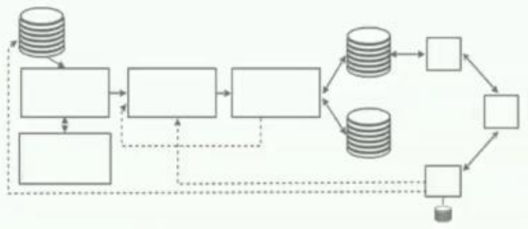
Batches consumed offset commits Passthrough allows us to track what messages have been successfully processed for At Least Once message delivery guarantees.

Create a Sink from the Flow with toMat()

Stream Materialization (running) the graph.

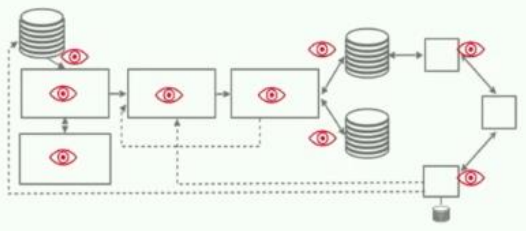
Challenge 2, Observability

Observability, No data loss



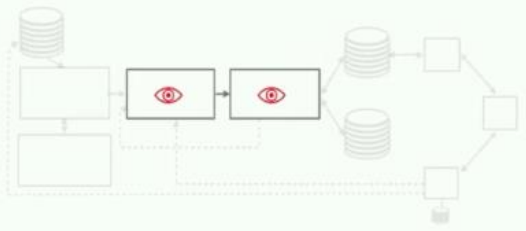
- Infrastructure and application metrics
- Logs centralization
- Tracing
- EV_ID = timestamp + DB sequence

Observability, No data loss

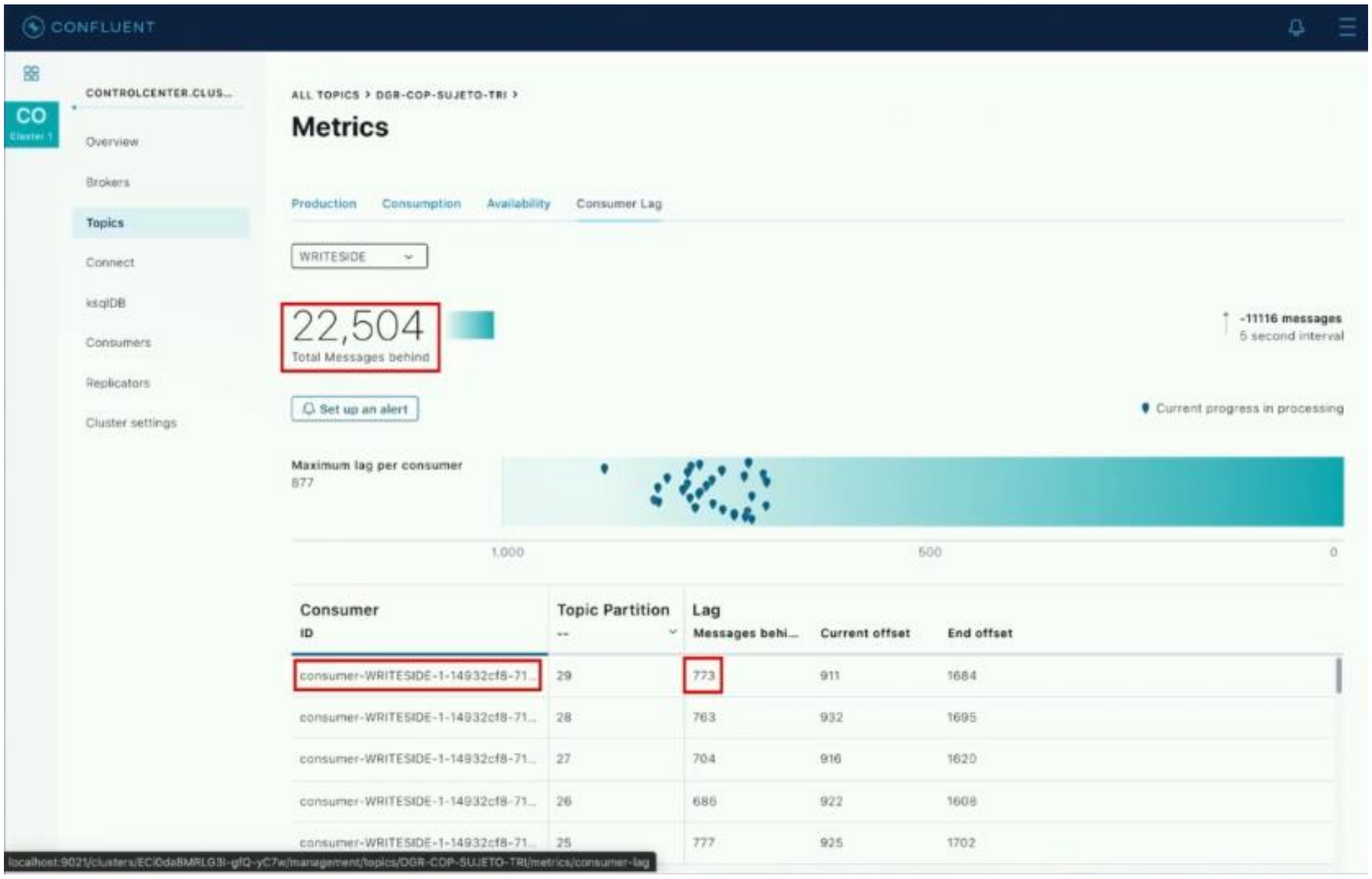


- Infrastructure and application metrics
- Logs centralization
- Tracing
- EV_ID = timestamp + DB sequence

Observability, No data loss

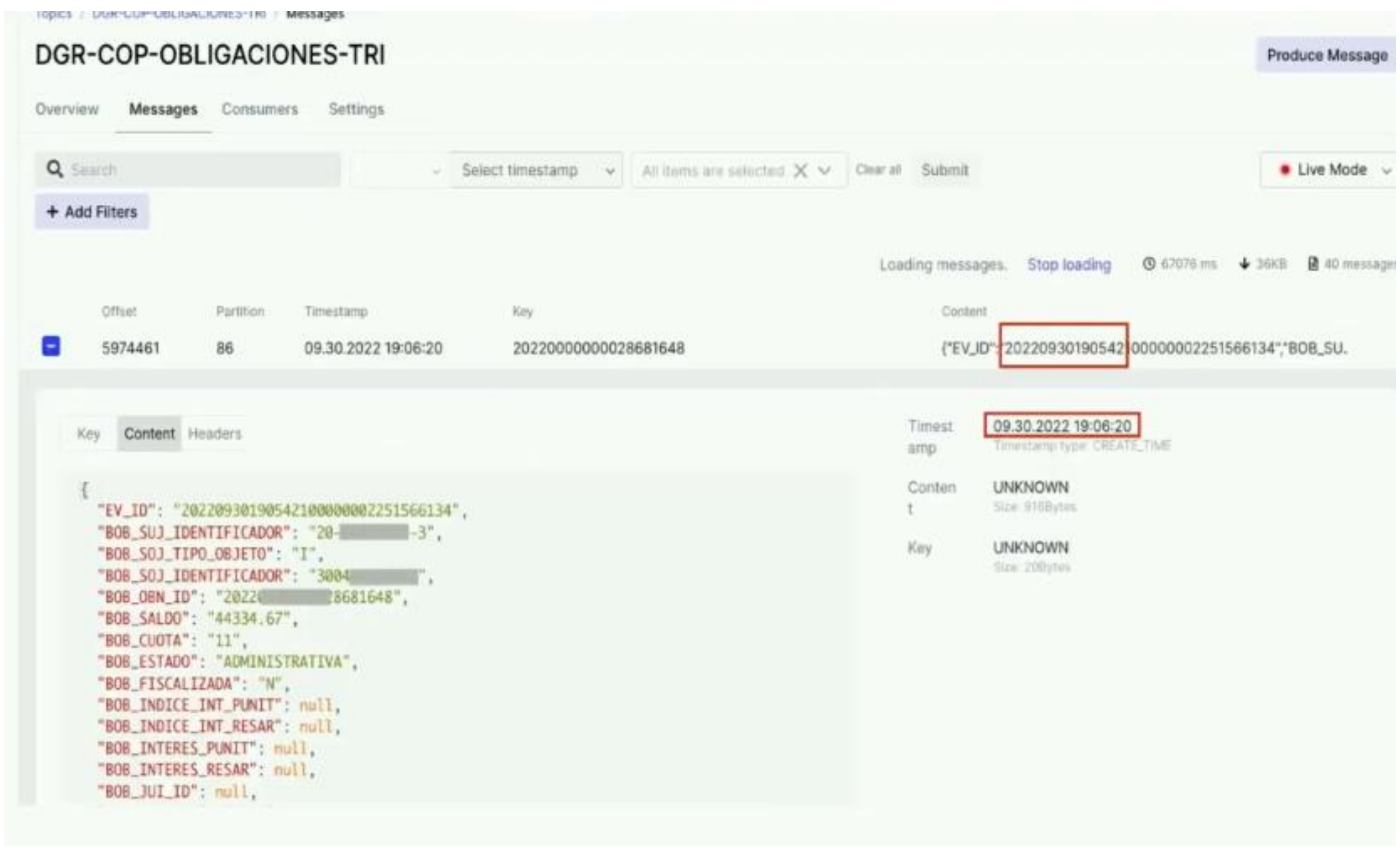


- Infrastructure and application metrics
- Logs centralization
- Tracing
- EV_ID = timestamp + DB sequence



Consumer ID	Topic Partition	Lag Messages behi...	Current offset	End offset
consumer-WRITESIDE-1-14932cf8-71...	29	773	911	1684
consumer-WRITESIDE-1-14932cf8-71...	28	763	932	1695
consumer-WRITESIDE-1-14932cf8-71...	27	704	916	1620
consumer-WRITESIDE-1-14932cf8-71...	26	686	922	1608
consumer-WRITESIDE-1-14932cf8-71...	25	777	925	1702

localhost:9021/clusters/EClOdaBMRLG3I-gfQ-yC7w/management/topics/DGR-COP-SUJETO-TRI/metrics/consumer-lag

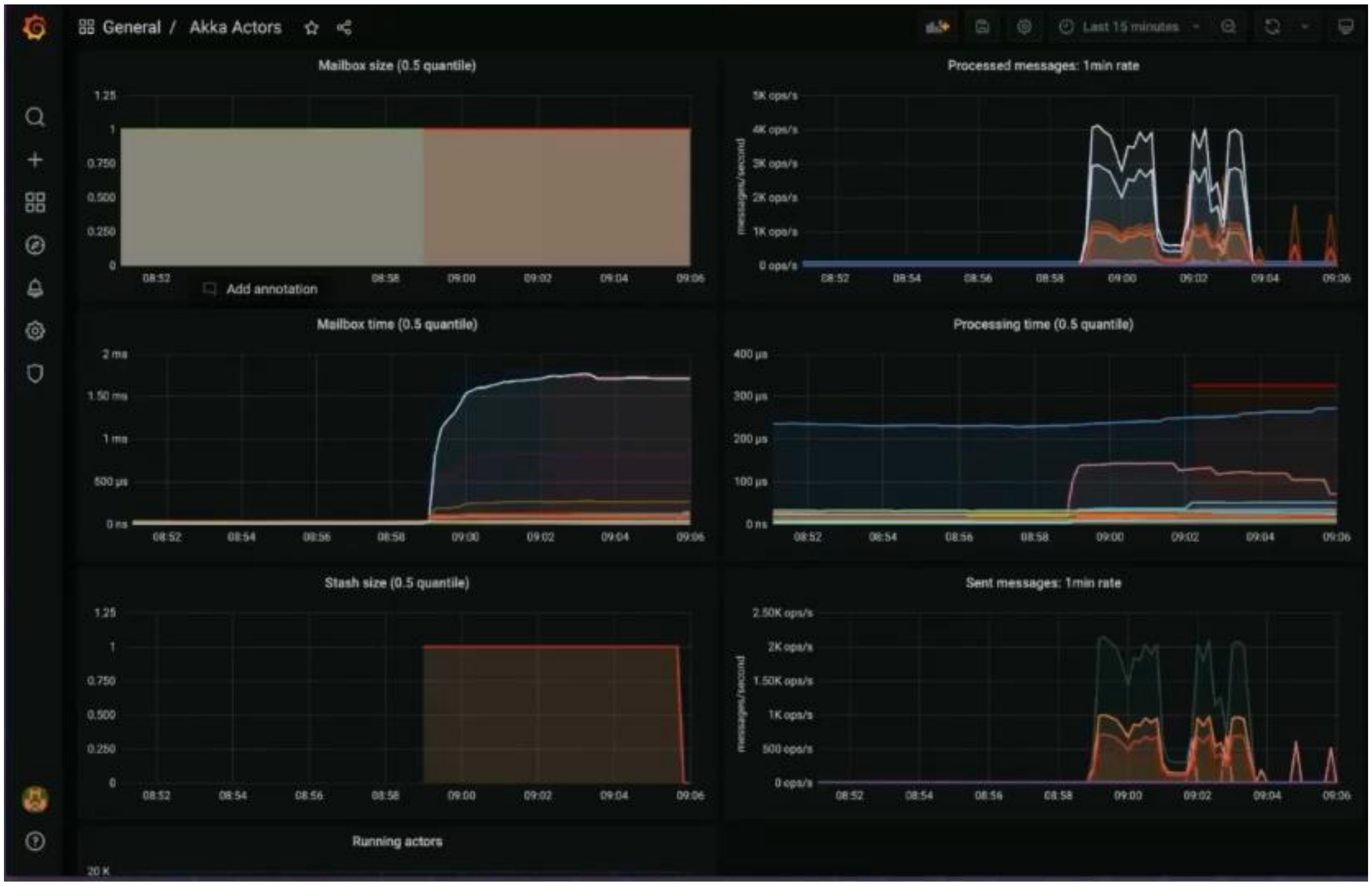


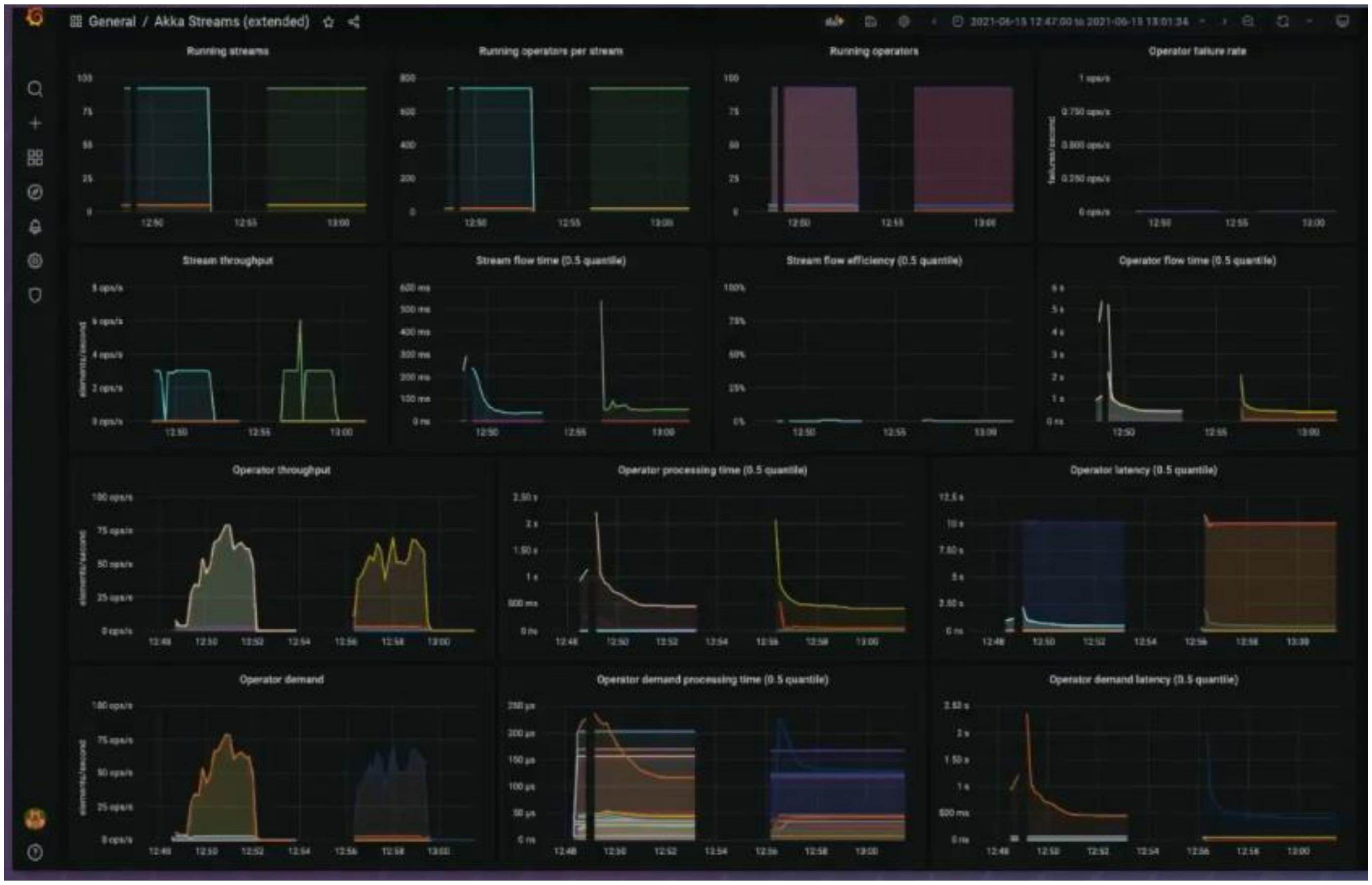
Offset	Partition	Timestamp	Key	Content
5974461	86	09.30.2022 19:06:20	20220000000028681648	{"EV_ID":"20220930190542100000002251566134","BOB_SU.

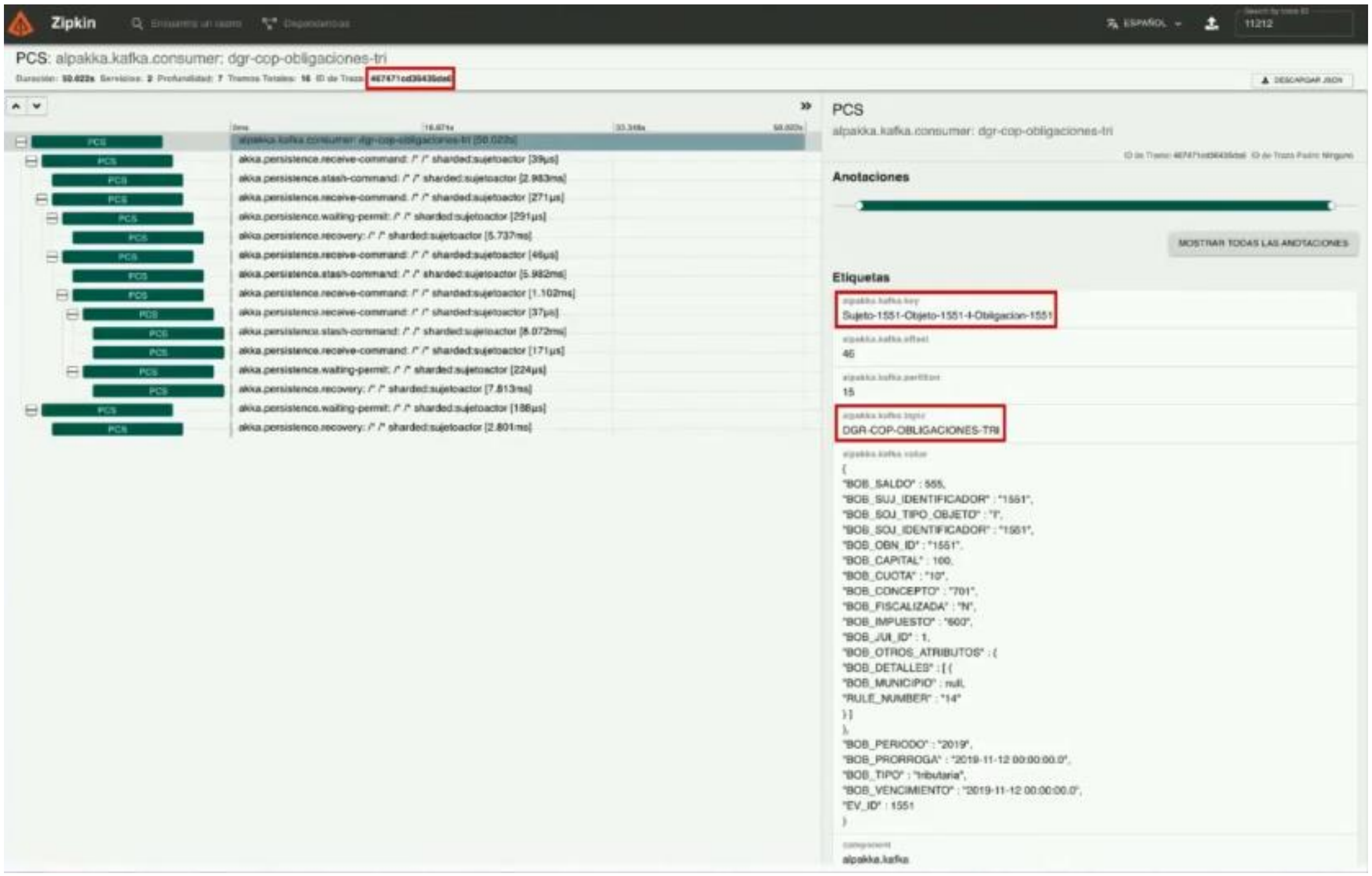
```
{
  "EV_ID": "20220930190542100000002251566134",
  "BOB_SUJ_IDENTIFICADOR": "20-[redacted]-3",
  "BOB_SOJ_TIPO_OBJETO": "I",
  "BOB_SOJ_IDENTIFICADOR": "3004[redacted]",
  "BOB_OBN_ID": "2022[redacted]8681648",
  "BOB_SALDO": "44334.67",
  "BOB_CUOTA": "11",
  "BOB_ESTADO": "ADMINISTRATIVA",
  "BOB_FISCALIZADA": "N",
  "BOB_INDICE_INT_PUNIT": null,
  "BOB_INDICE_INT_RESAR": null,
  "BOB_INTERES_PUNIT": null,
  "BOB_INTERES_RESAR": null,
  "BOB_JUI_ID": null,
```

Timestamp: 09.30.2022 19:06:20
Timestamp type: CREATE_TIME
Content: UNKNOWN
Size: 916Bytes
Key: UNKNOWN
Size: 20Bytes

This is the Kafka UI OSS tool to show the message console and compare timestamps


General / Akka Actors
Last 15 minutes
Mailbox size (0.5 quantile)
Processed messages: 1min rate
Add annotation
Mailbox time (0.5 quantile)
Processing time (0.5 quantile)
Stash size (0.5 quantile)
Sent messages: 1min rate
messages/second
Running actors
20 K


General / Akka Streams (extended)
2021-06-15 12:47:00 to 2021-06-15 13:01:34
Running streams
Running operators per stream
Running operators
Operator failure rate
Stream throughput
Stream flow time (0.5 quantile)
Stream flow efficiency (0.5 quantile)
Operator flow time (0.5 quantile)
Operator throughput
Operator processing time (0.5 quantile)
Operator latency (0.5 quantile)
Operator demand
Operator demand processing time (0.5 quantile)
Operator demand latency (0.5 quantile)
elements/second
failures/second



Zipkin

PCS: alpakka.kafka.consumer: dgr-cop-obligaciones-tri

Duración: 50.022s Servicios: 2 Profundidad: 7 Tramos Totales: 16 ID de Traza: 467471cd36435da6

```
alpakka.kafka.consumer: dgr-cop-obligaciones-tri [50.022s]
akka.persistence.receive-command: /* /* sharded:sujetoactor [39µs]
akka.persistence.stash-command: /* /* sharded:sujetoactor [2.983ms]
akka.persistence.receive-command: /* /* sharded:sujetoactor [271µs]
akka.persistence.waiting-permit: /* /* sharded:sujetoactor [291µs]
akka.persistence.recovery: /* /* sharded:sujetoactor [5.737ms]
akka.persistence.receive-command: /* /* sharded:sujetoactor [46µs]
akka.persistence.stash-command: /* /* sharded:sujetoactor [5.982ms]
akka.persistence.receive-command: /* /* sharded:sujetoactor [1.102ms]
akka.persistence.receive-command: /* /* sharded:sujetoactor [37µs]
akka.persistence.stash-command: /* /* sharded:sujetoactor [8.072ms]
akka.persistence.receive-command: /* /* sharded:sujetoactor [171µs]
akka.persistence.waiting-permit: /* /* sharded:sujetoactor [224µs]
akka.persistence.recovery: /* /* sharded:sujetoactor [7.813ms]
akka.persistence.waiting-permit: /* /* sharded:sujetoactor [188µs]
akka.persistence.recovery: /* /* sharded:sujetoactor [2.801ms]
```

PCS

alpakka.kafka.consumer: dgr-cop-obligaciones-tri

Anotaciones

Etiquetas

alpakka.kafka.key: Sujeto-1551-Objeto-1551-I-Obligacion-1551

alpakka.kafka.offset: 46

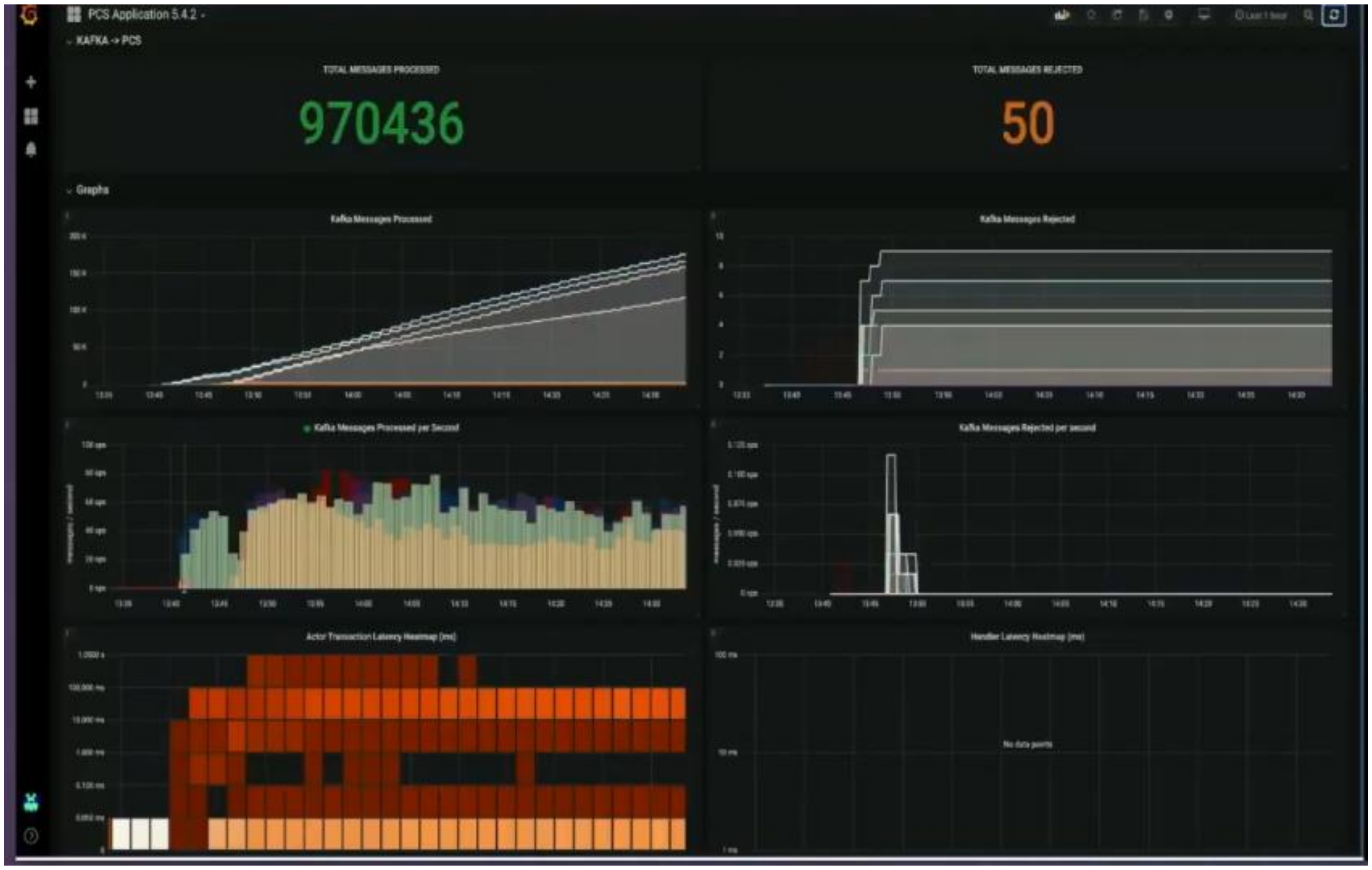
alpakka.kafka.partition: 15

alpakka.kafka.topic: DGR-COP-OBLIGACIONES-TRI

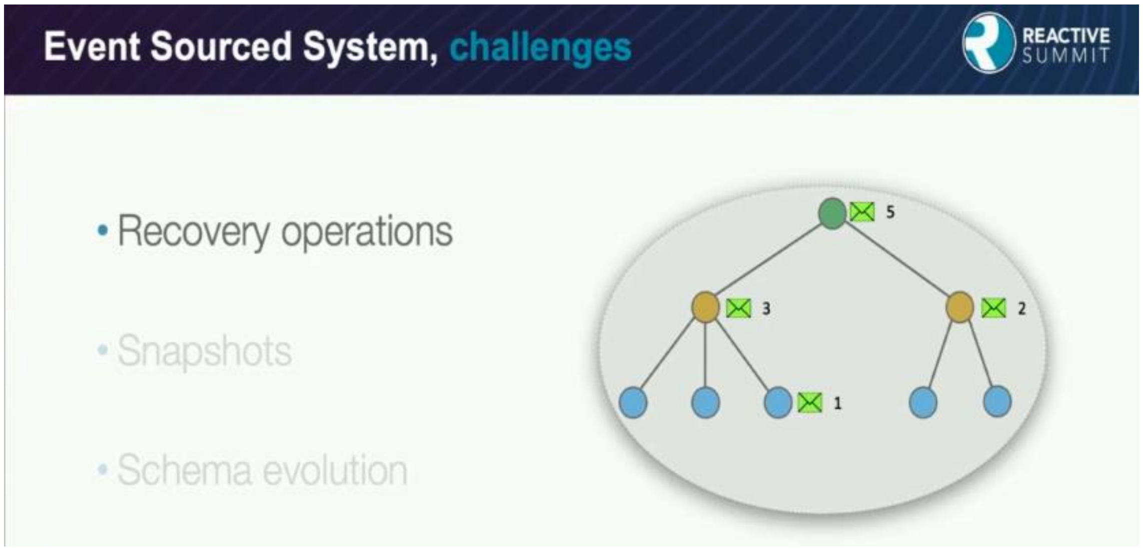
alpakka.kafka.value:

```
{
"BOB_SALDO" : 555,
"BOB_SUJ_IDENTIFICADOR" : "1551",
"BOB_SOJ_TIPO_OBJETO" : "I",
"BOB_SOJ_IDENTIFICADOR" : "1551",
"BOB_OBN_ID" : "1551",
"BOB_CAPITAL" : 100,
"BOB_CUOTA" : "10",
"BOB_CONCEPTO" : "701",
"BOB_FISCALIZADA" : "N",
"BOB_IMPUESTO" : "600",
"BOB_JUI_ID" : 1,
"BOB_OTROS_ATRIBUTOS" : {
"BOB_DETALLES" : [ {
"BOB_MUNICIPIO" : null,
"RULE_NUMBER" : "14"
} ]
},
"BOB_PERIODO" : "2019",
"BOB_PRORROGA" : "2019-11-12 00:00:00.0",
"BOB_TIPO" : "tributaria",
"BOB_VENCIMIENTO" : "2019-11-12 00:00:00.0",
"EV_ID" : 1551
}
```

component: alpakka.kafka

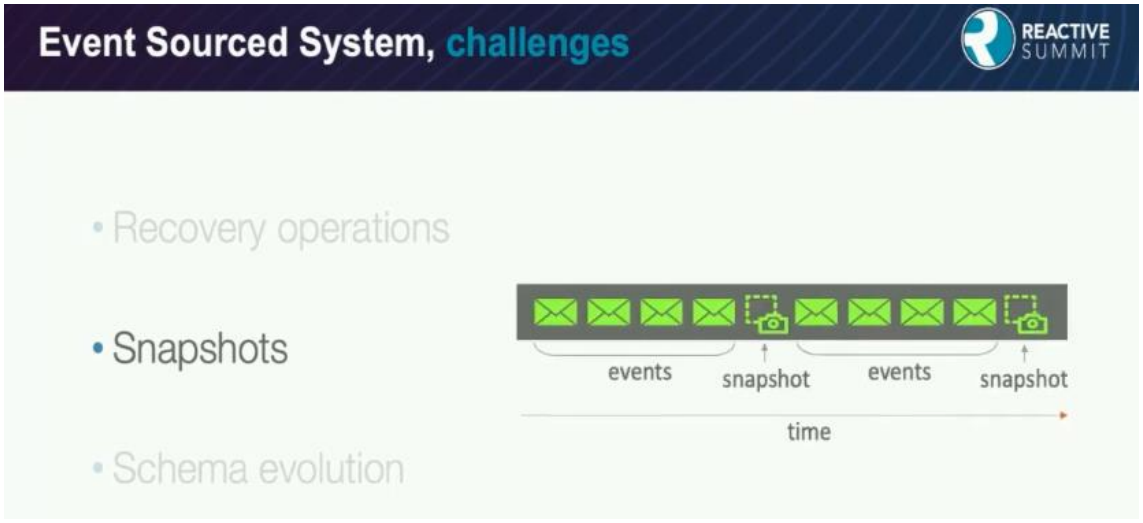


Challenge 3, Schema Evolution



Event Sourced System, challenges

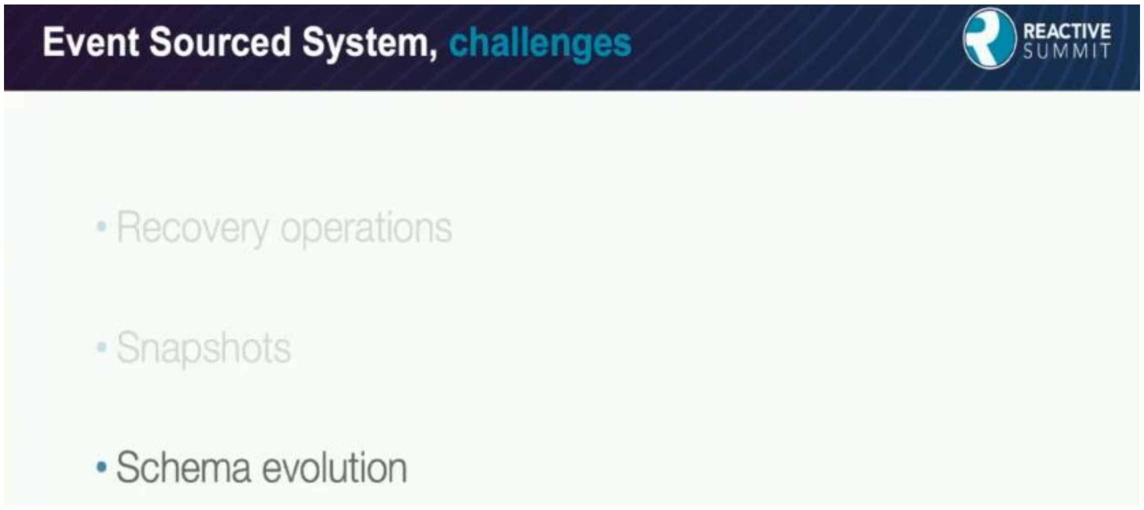
- Recovery operations
- Snapshots
- Schema evolution



Event Sourced System, challenges

- Recovery operations
- Snapshots
- Schema evolution

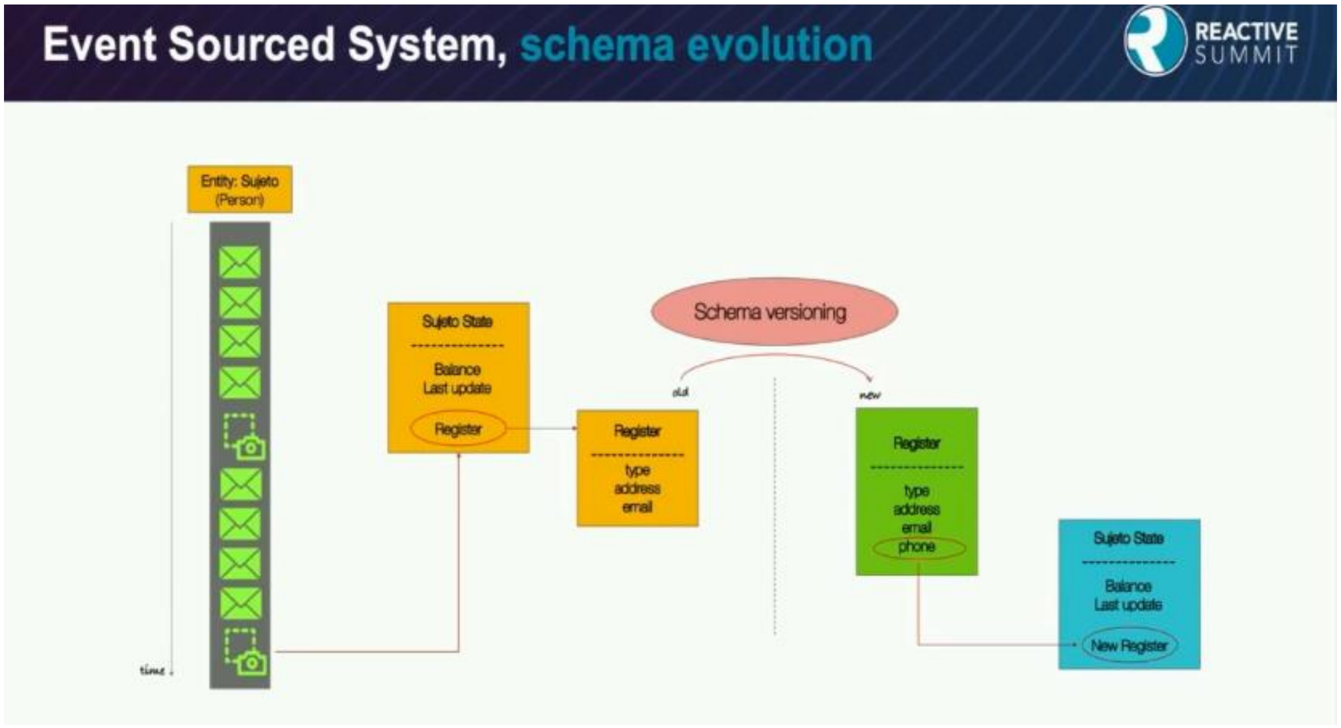
With Akka streams, you can configure the system to take the state snapshot after N events



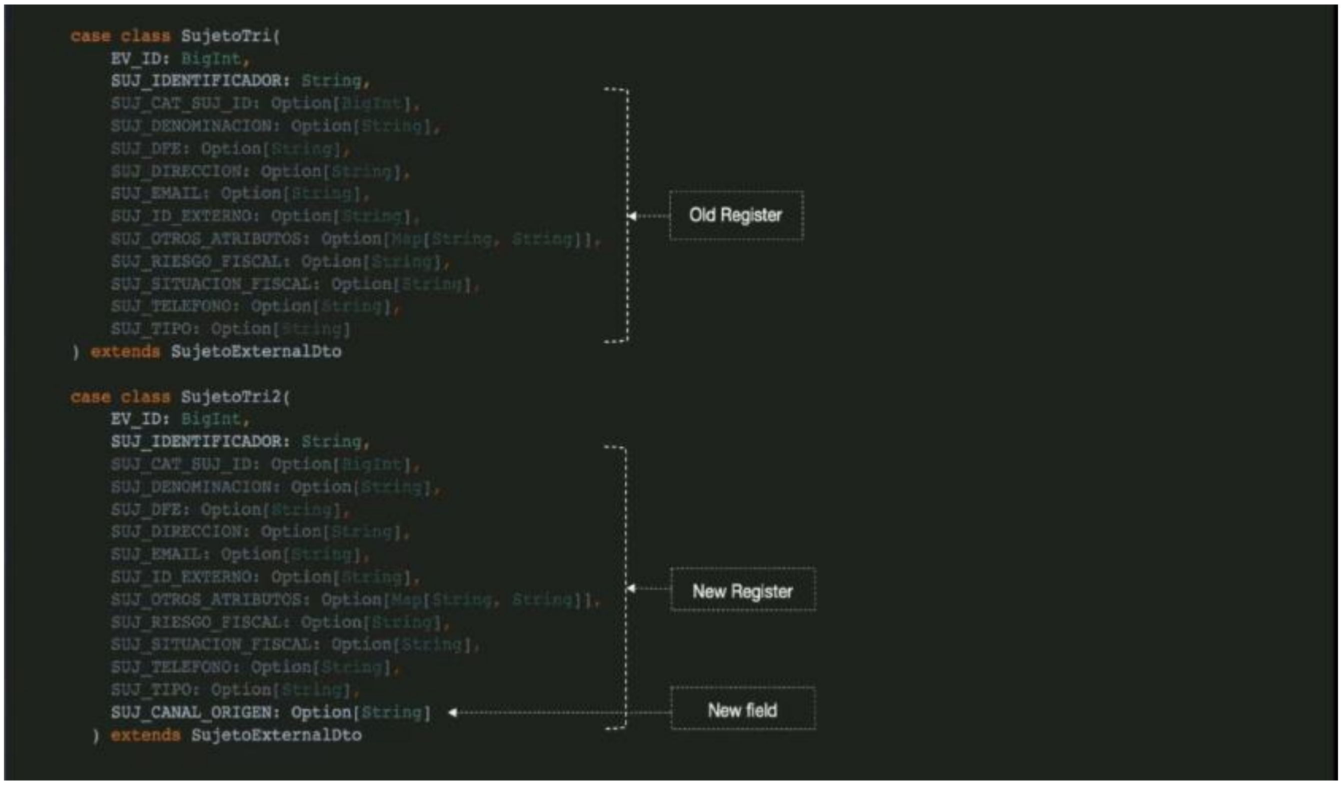
Event Sourced System, challenges

- Recovery operations
- Snapshots
- Schema evolution

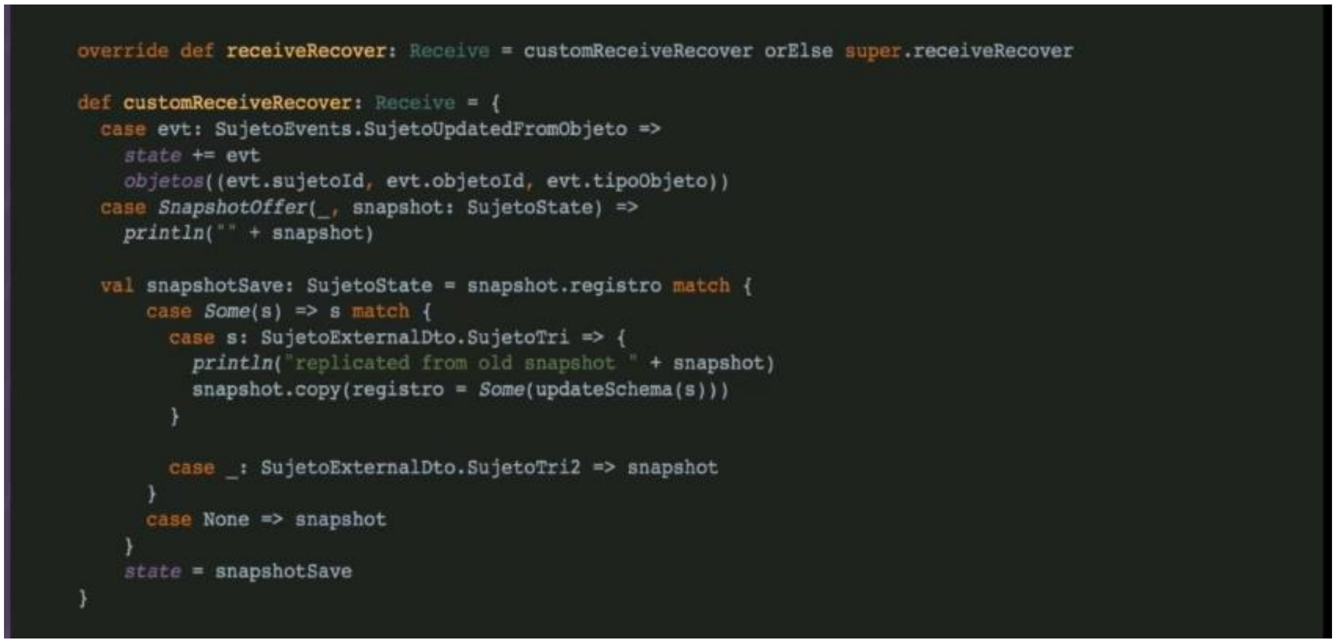
Event Sourced System, schema evolution

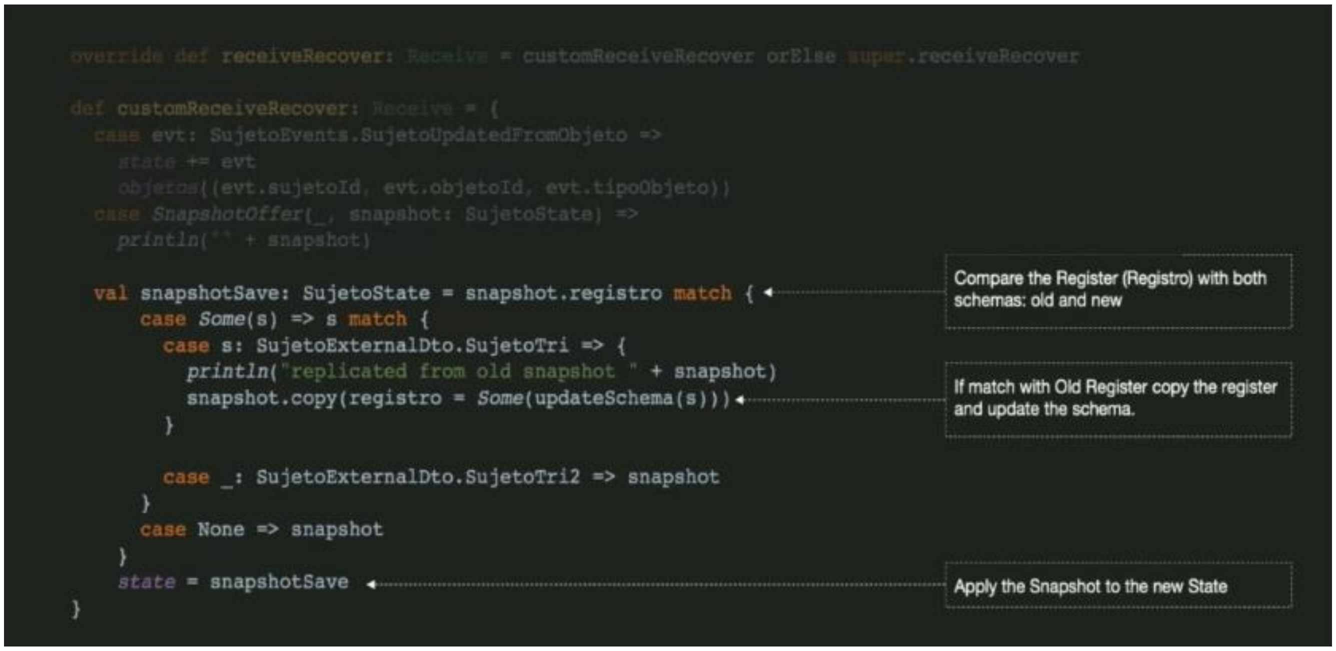


For schema changes like adding a new field, we added a new class for the new schema and let the tool compare versions



```
case class SujetoTri(
    EV_ID: BigInt,
    SUJ_IDENTIFICADOR: String,
    SUJ_CAT_SUJ_ID: Option[BigInt],
    SUJ_DENOMINACION: Option[String],
    SUJ_DFE: Option[String],
    SUJ_DIRECCION: Option[String],
    SUJ_EMAIL: Option[String],
    SUJ_ID_EXTERNO: Option[String],
    SUJ_OTROS_ATRIBUTOS: Option[Map[String, String]],
    SUJ_RIESGO_FISCAL: Option[String],
    SUJ_SITUACION_FISCAL: Option[String],
    SUJ_TELEFONO: Option[String],
    SUJ_TIPO: Option[String]
) extends SujetoExternalDto

case class SujetoTri2(
    EV_ID: BigInt,
    SUJ_IDENTIFICADOR: String,
    SUJ_CAT_SUJ_ID: Option[BigInt],
    SUJ_DENOMINACION: Option[String],
    SUJ_DFE: Option[String],
    SUJ_DIRECCION: Option[String],
    SUJ_EMAIL: Option[String],
    SUJ_ID_EXTERNO: Option[String],
    SUJ_OTROS_ATRIBUTOS: Option[Map[String, String]],
    SUJ_RIESGO_FISCAL: Option[String],
    SUJ_SITUACION_FISCAL: Option[String],
    SUJ_TELEFONO: Option[String],
    SUJ_TIPO: Option[String],
    SUJ_CANAL_ORIGEN: Option[String]
  ) extends SujetoExternalDto
```


```
override def receiveRecover: Receive = customReceiveRecover orElse super.receiveRecover

def customReceiveRecover: Receive = {
  case evt: SujetoEvents.SujetoUpdatedFromObjeto =>
    state += evt
    objetos((evt.sujetoId, evt.objetoId, evt.tipoObjeto))
  case SnapshotOffer(_, snapshot: SujetoState) =>
    println("" + snapshot)

  val snapshotSave: SujetoState = snapshot.registro match {
      case Some(s) => s match {
        case s: SujetoExternalDto.SujetoTri => {
          println("replicated from old snapshot " + snapshot)
          snapshot.copy(registro = Some(updateSchema(s)))
        }

        case _: SujetoExternalDto.SujetoTri2 => snapshot
      }
      case None => snapshot
    }
    state = snapshotSave
}
```


```
override def receiveRecover: Receive = customReceiveRecover orElse super.receiveRecover

def customReceiveRecover: Receive = {
  case evt: SujetoEvents.SujetoUpdatedFromObjeto =>
    state += evt
    objetos((evt.sujetoId, evt.objetoId, evt.tipoObjeto))
  case SnapshotOffer(_, snapshot: SujetoState) =>
    println("" + snapshot)

  val snapshotSave: SujetoState = snapshot.registro match {
      case Some(s) => s match {
        case s: SujetoExternalDto.SujetoTri => {
          println("replicated from old snapshot " + snapshot)
          snapshot.copy(registro = Some(updateSchema(s)))
        }

        case _: SujetoExternalDto.SujetoTri2 => snapshot
      }
      case None => snapshot
    }
    state = snapshotSave
}
```
Compare the Register (Registro) with both schemas: old and new
If match with Old Register copy the register and update the schema.
Apply the Snapshot to the new State

Results & key takeaways

Results & key takeaways

- From Development to Production
- Key takeaways
- Final words