Event-sourced architectures with Akka

@Sander_Mak
Luminis Technologies

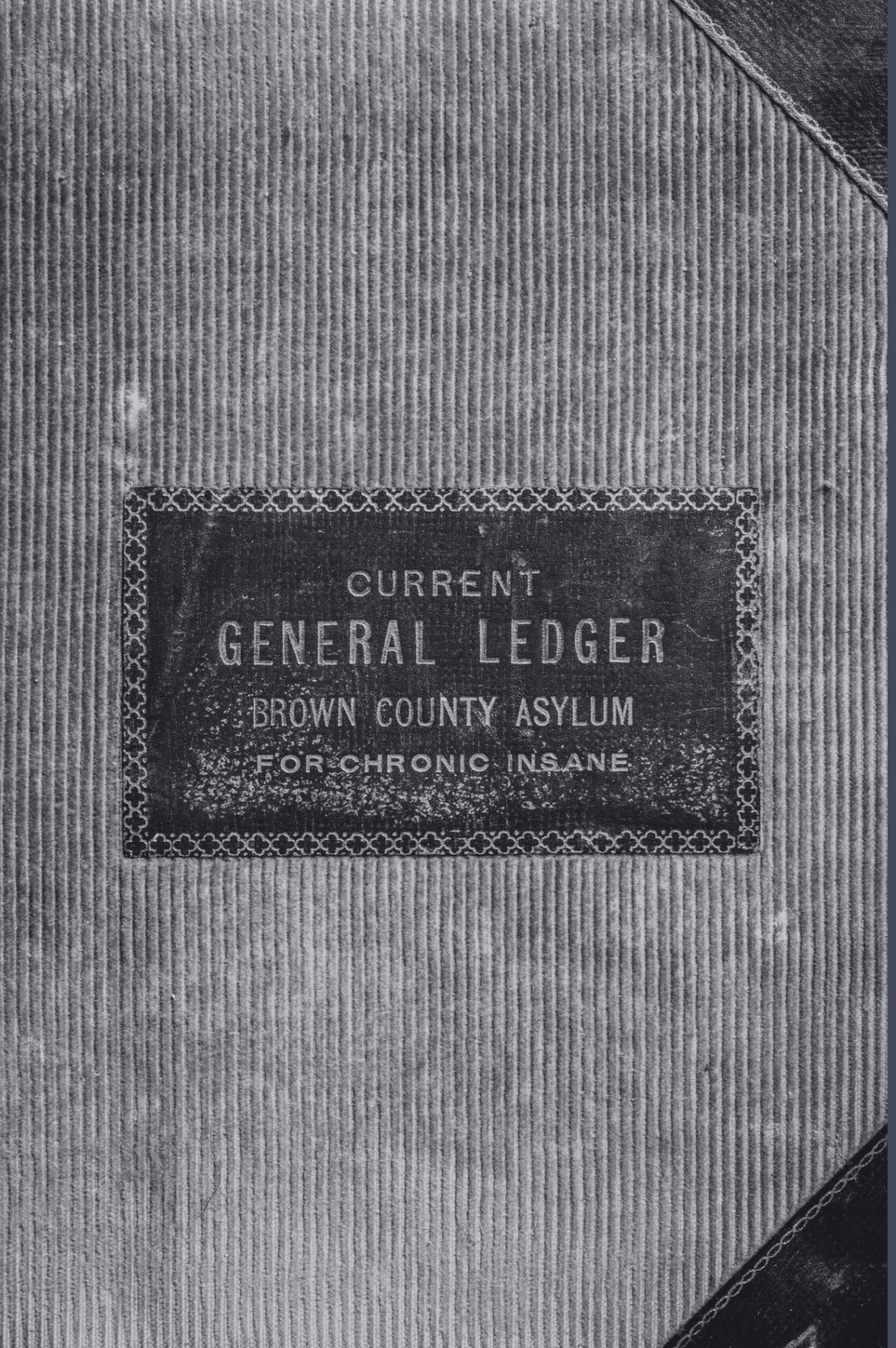
Today's journey

Event-sourcing

Actors

Akka Persistence

Design for ES



Event-sourcing

Is all about getting the **facts** straight

Typical 3 layer architecture

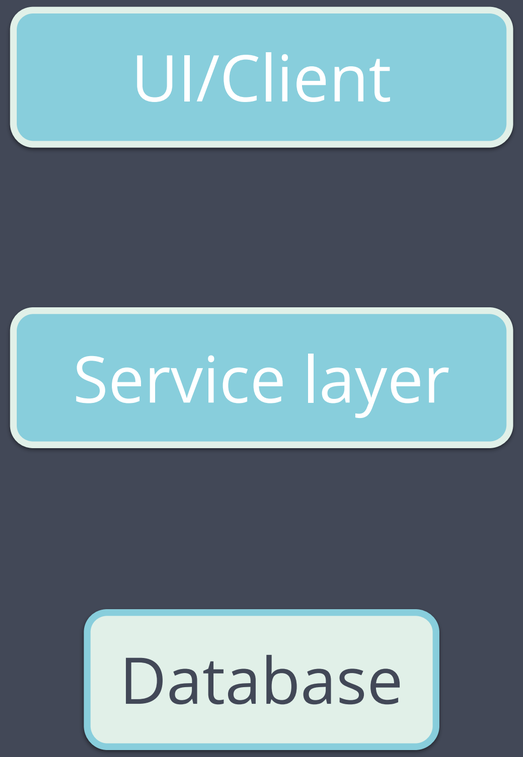
UI/Client

Service layer

Database

fetch ⇝ modify ⇝ store

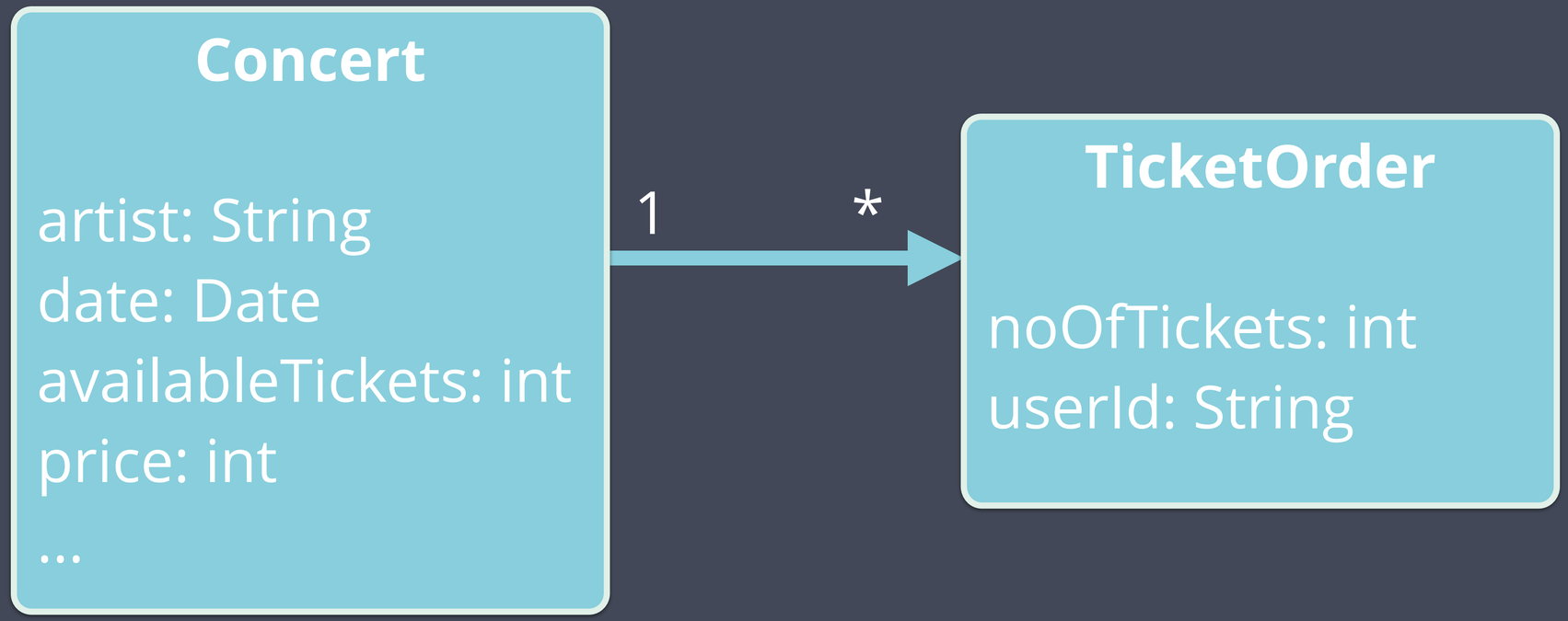
Typical 3 layer architecture



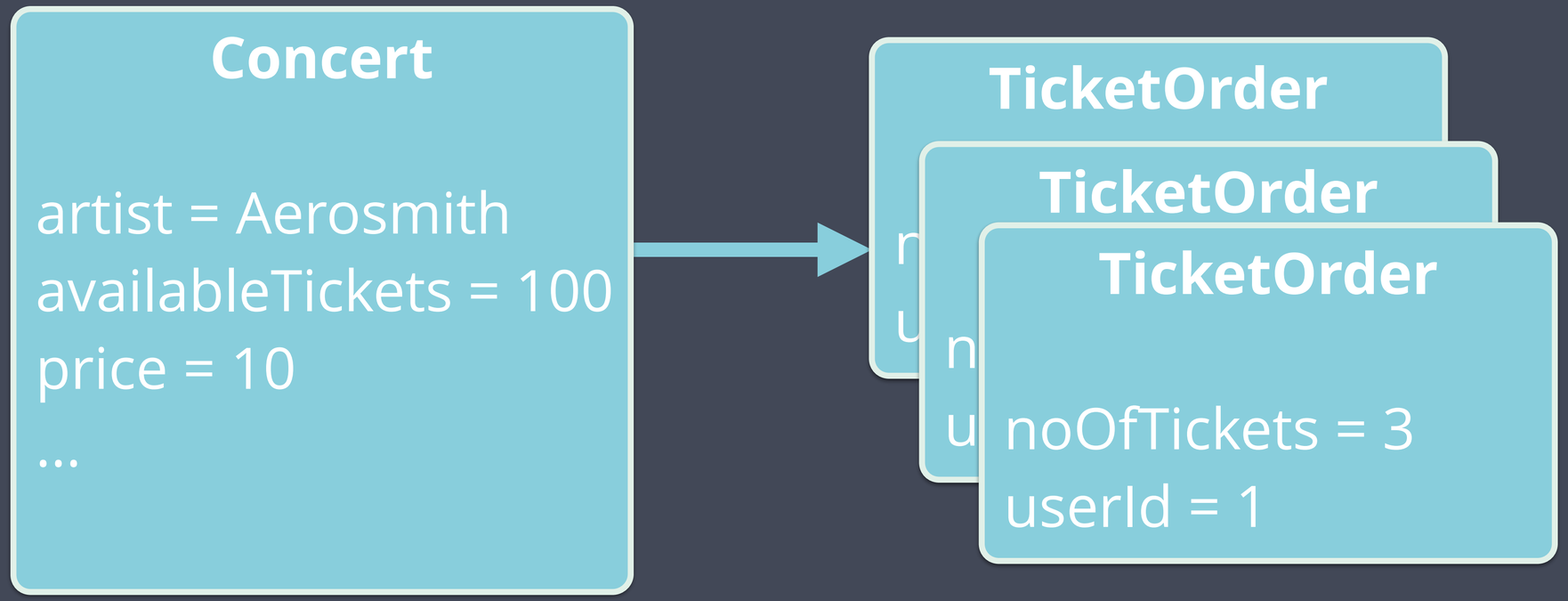
fetch → modify → store

Databases are
shared mutable state

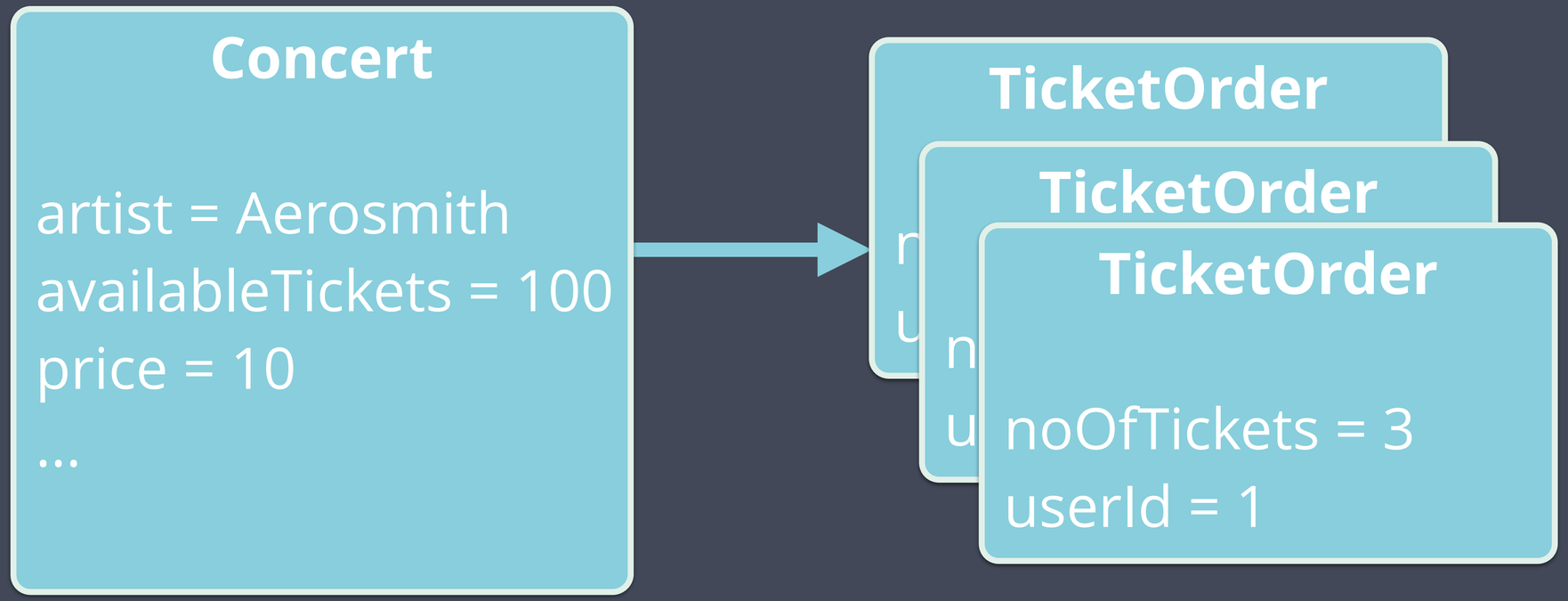
Typical entity modelling



Typical entity modelling

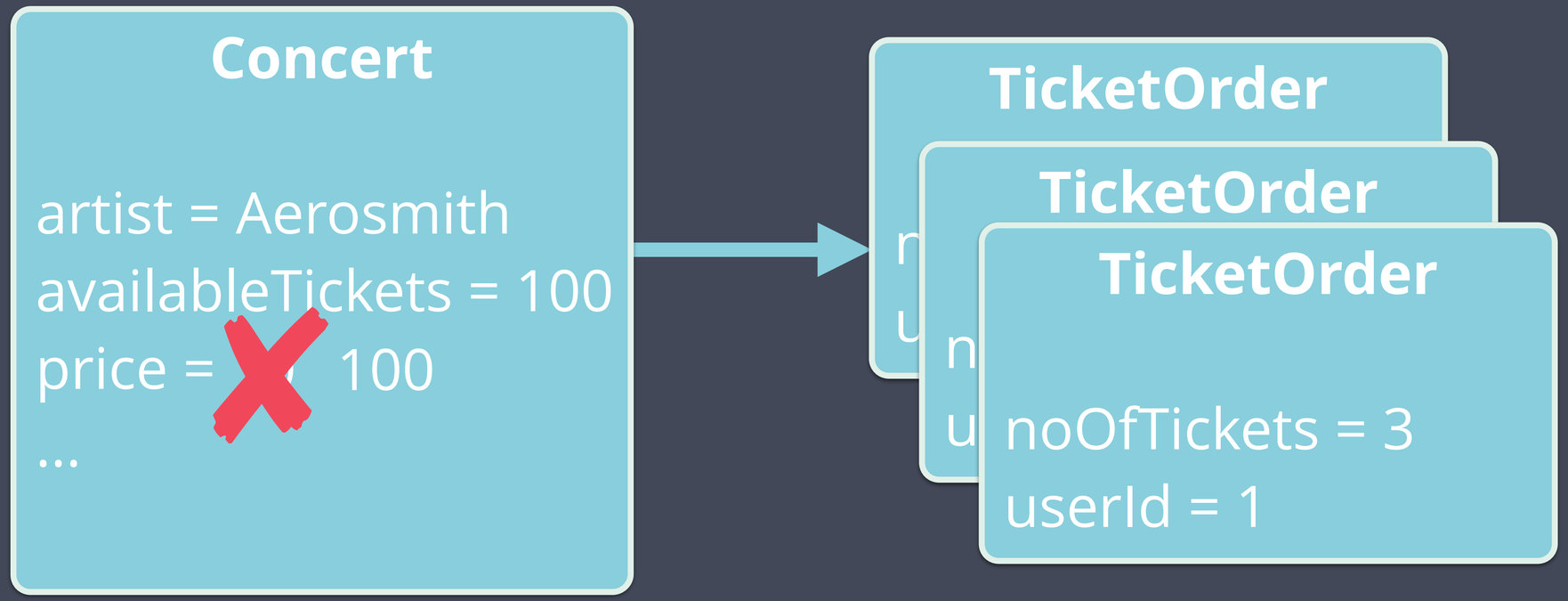


Typical entity modelling



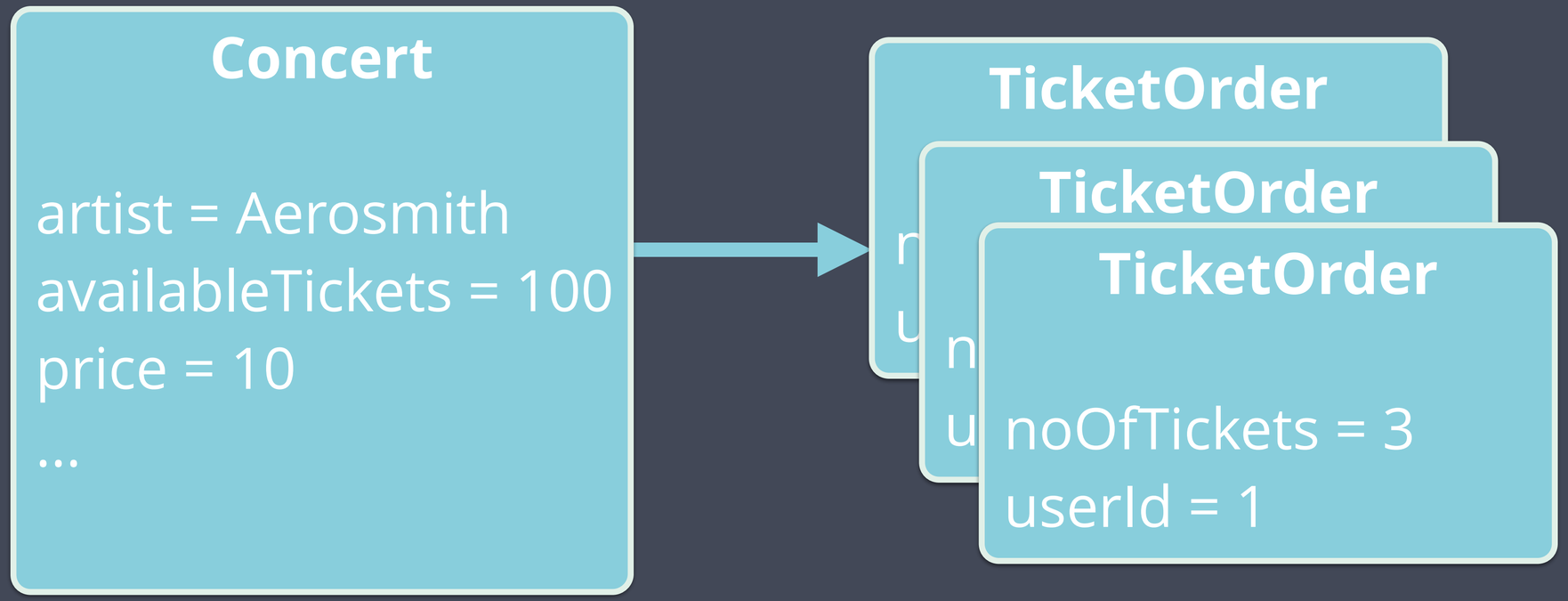
Changing the price

Typical entity modelling



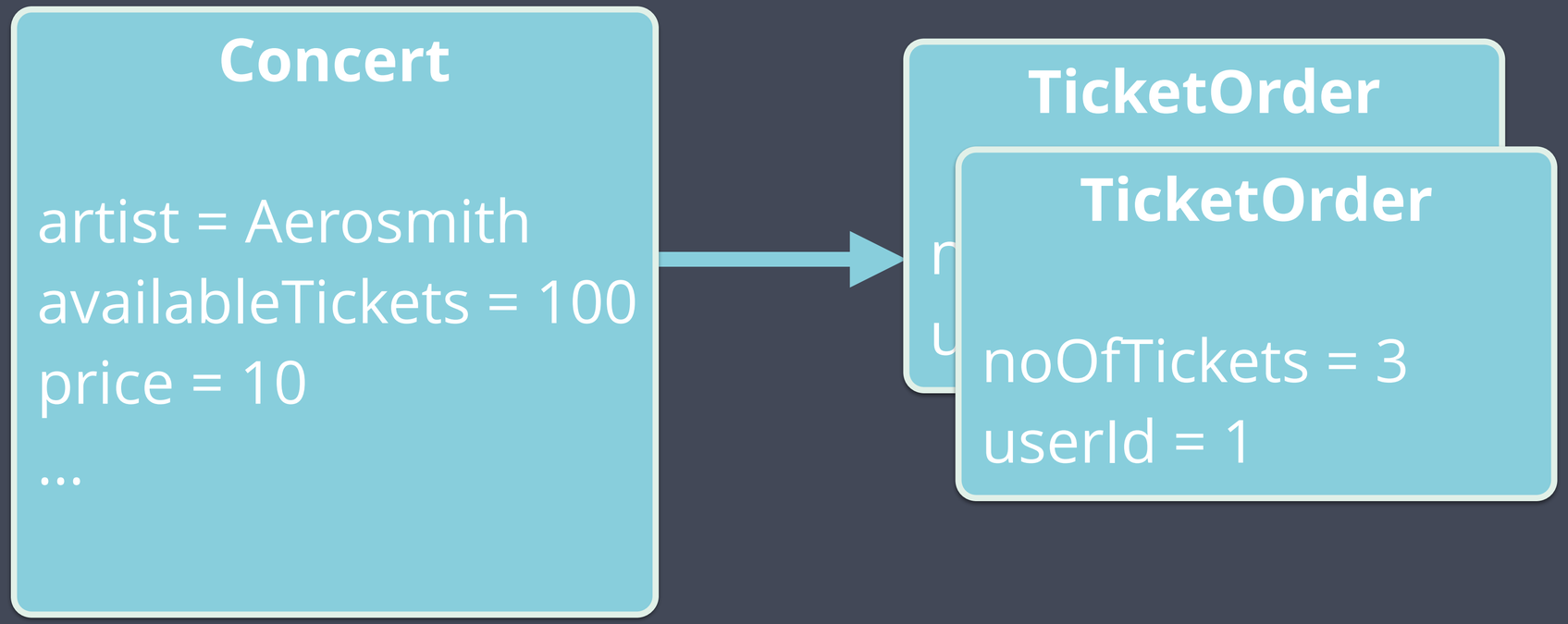
Changing the price

Typical entity modelling



Canceling an order

Typical entity modelling



Canceling an order

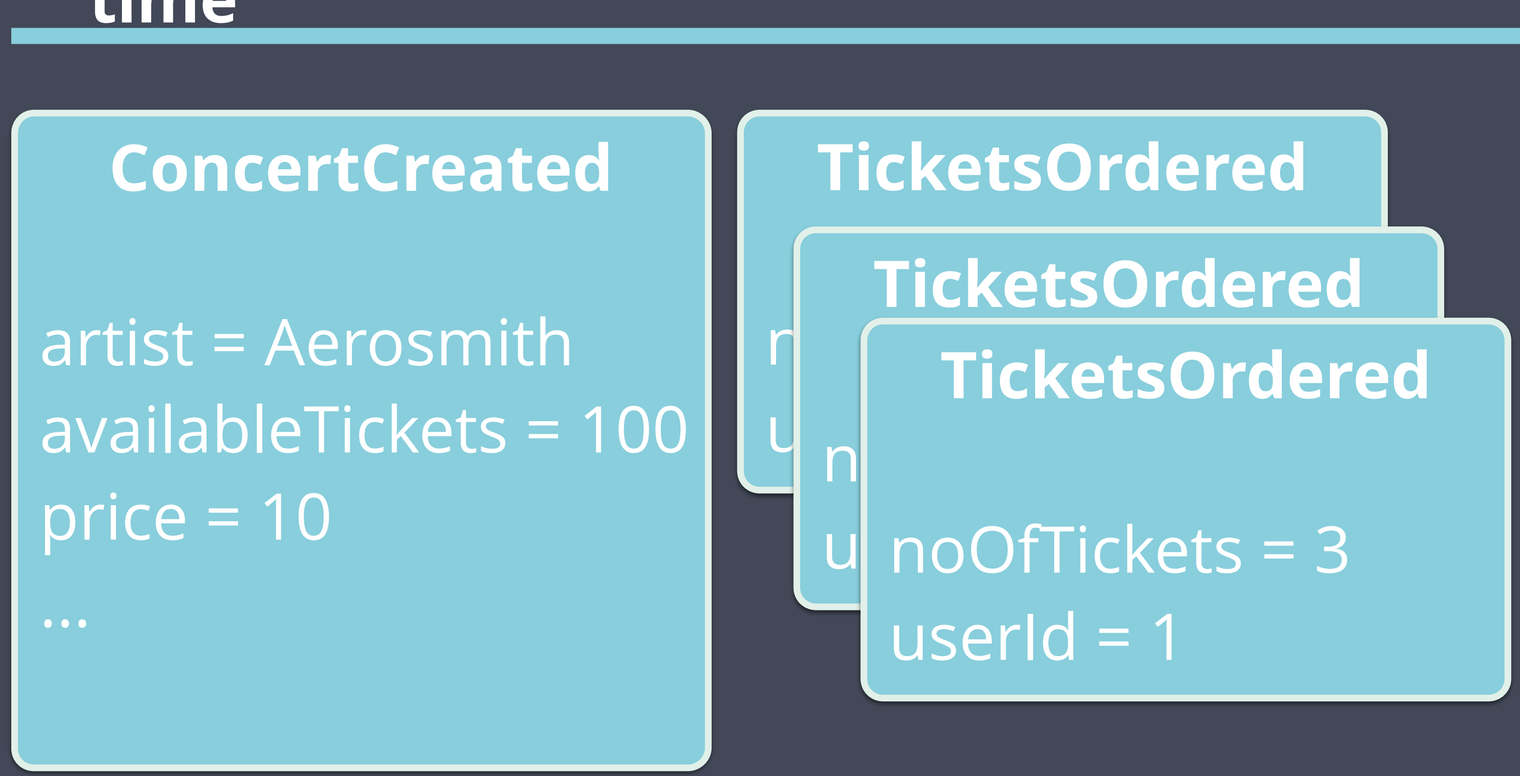
Update or **delete** statements in your app?

Congratulations, you are

LOSING DATA EVERY DAY

Event-sourced modelling

time



Event-sourced modelling

time

ConcertCreated

artist = Aerosmith
availableTickets = 100
price = 10
…

TicketsOrdered

TicketsOrdered

TicketsOrdered

noOfTickets = 3
userId = 1

Changing the price

Event-sourced modelling

time

ConcertCreated

artist = Aerosmith
availableTickets = 100
price = 10
…

TicketsOrdered

TicketsOrdered

TicketsOrdered

noOfTickets = 3
userId = 1

PriceChanged

price = 100

Changing the price

Event-sourced modelling

time

ConcertCreated

artist = Aerosmith
availableTickets = 100
price = 10
…

TicketsOrdered

TicketsOrdered

TicketsOrdered

noOfTickets = 3
userId = 1

PriceChanged

price = 100

Canceling an order

Event-sourced modelling

time

ConcertCreated

artist = Aerosmith
availableTickets = 100
price = 10
...

TicketsOrdered

TicketsOrdered

TicketsOrdered

noOfTickets = 3
userId = 1

PriceChanged

price = 100

OrderCancelled

userId = 1

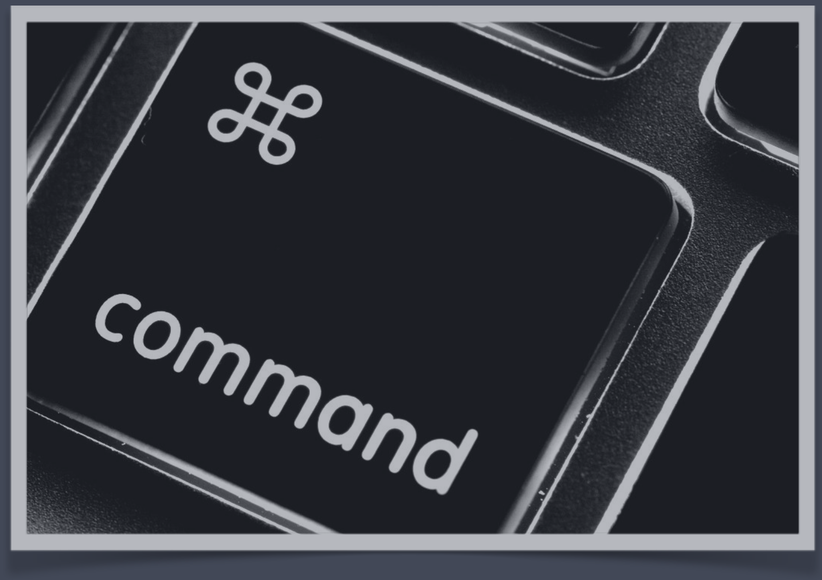
Canceling an order

Event-sourced modelling

- Immutable events
- Append-only storage (scalable)
- Replay events: reconstruct historic state
- Events as integration mechanism
- Events as audit mechanism

Event-sourcing: capture *all changes* to application state as a sequence of events

Events: where from?



Commands & Events

Do something (active)

Can be rejected (validation)

Can be responded to

It happened.
Deal with it.
(facts)

Querying & event-sourcing

How do you query a log?

Querying & event-sourcing

How do you query a log?

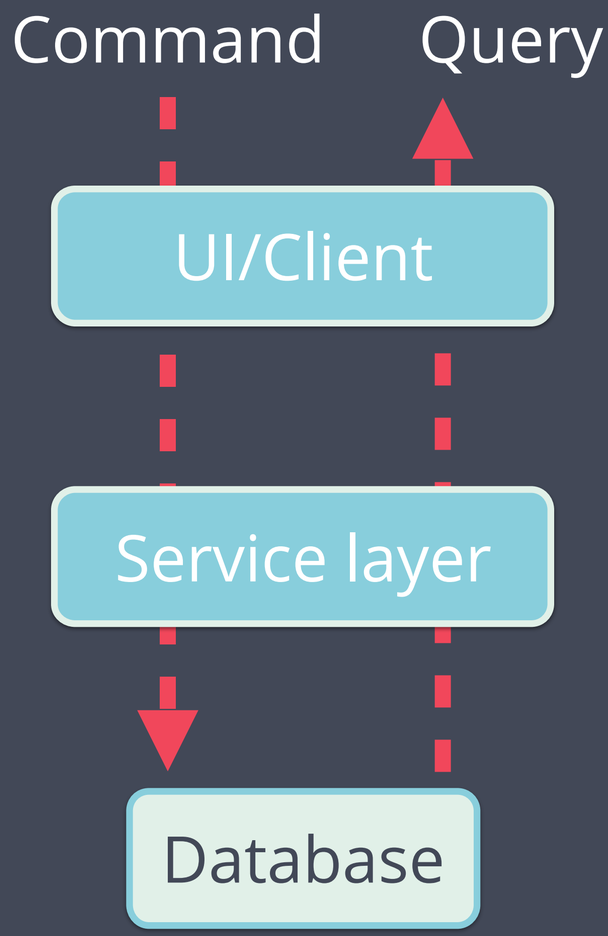
Command

Query

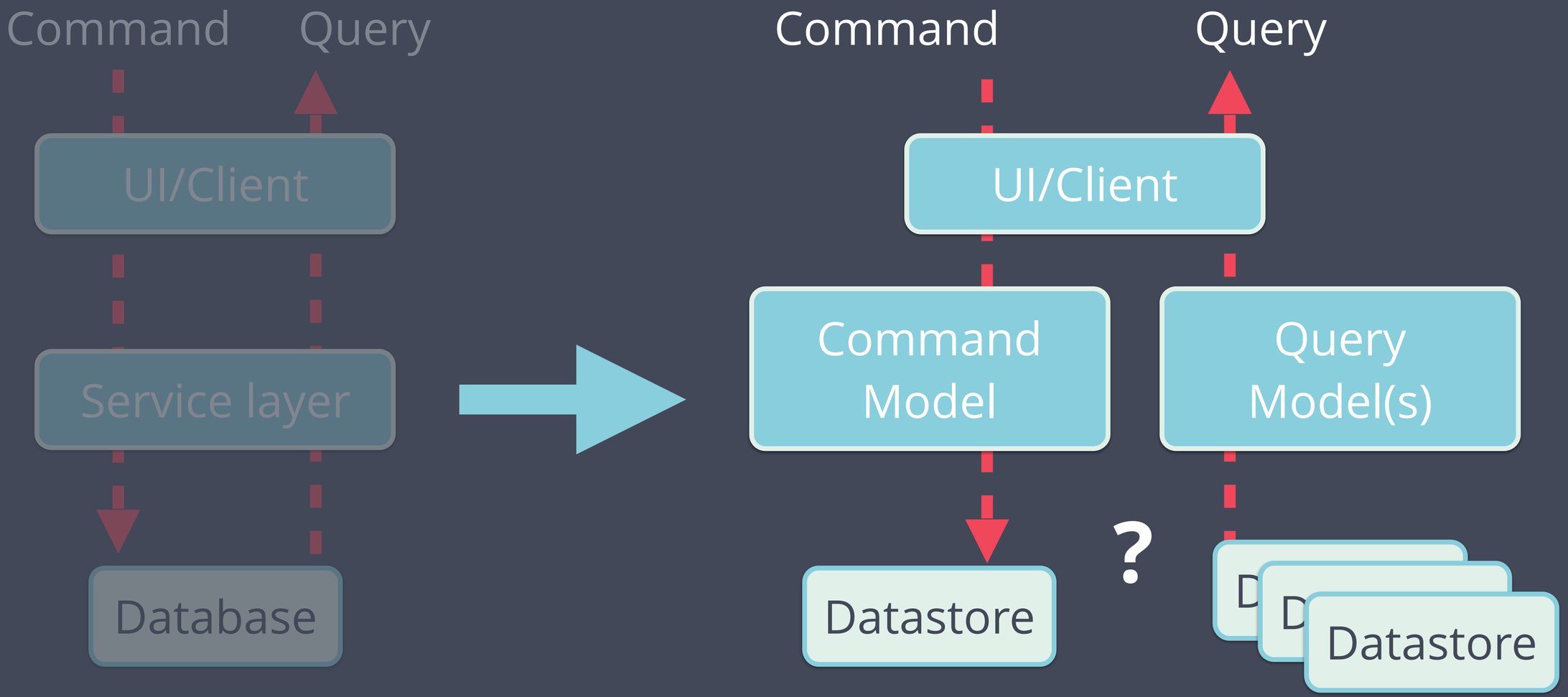
Responsibility

Segregation

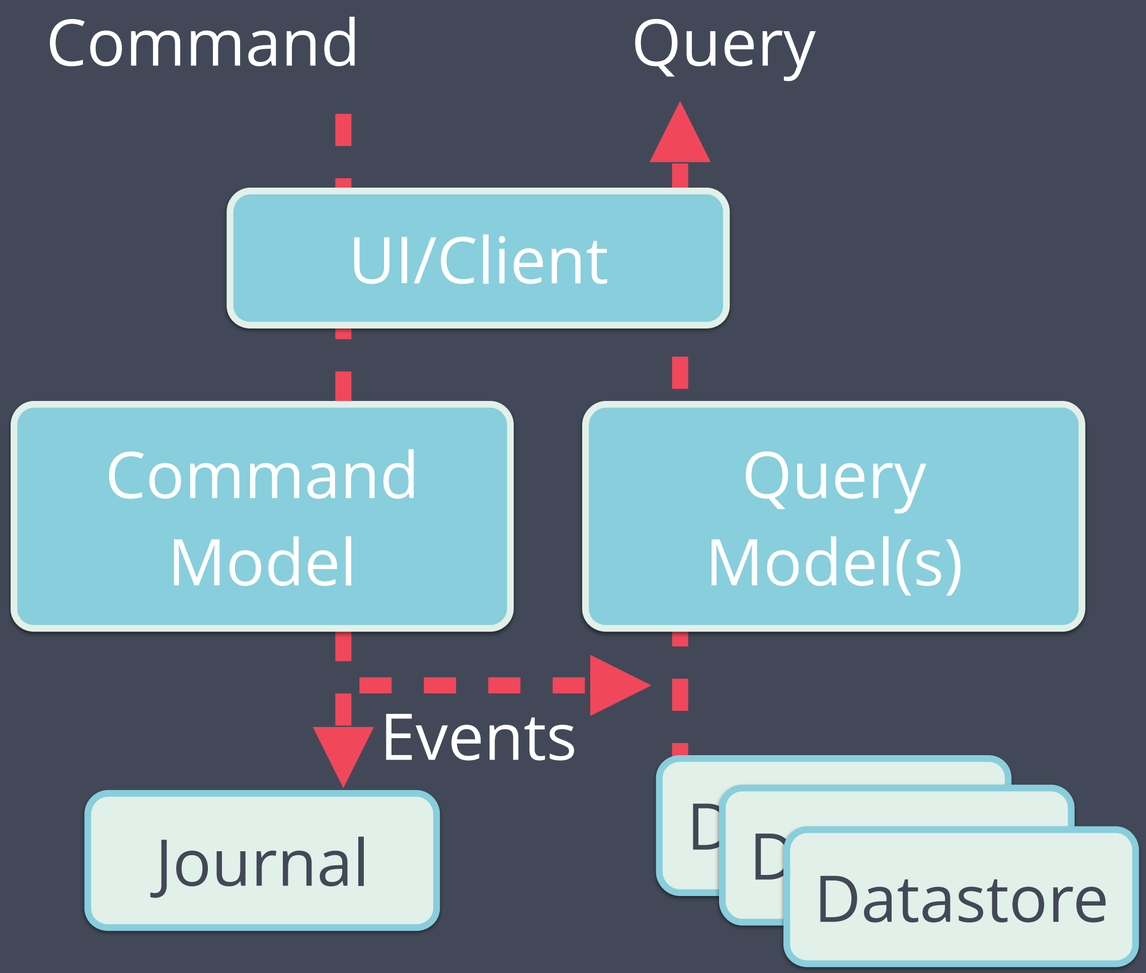
CQRS without ES



CQRS without ES



Event-sourced CQRS

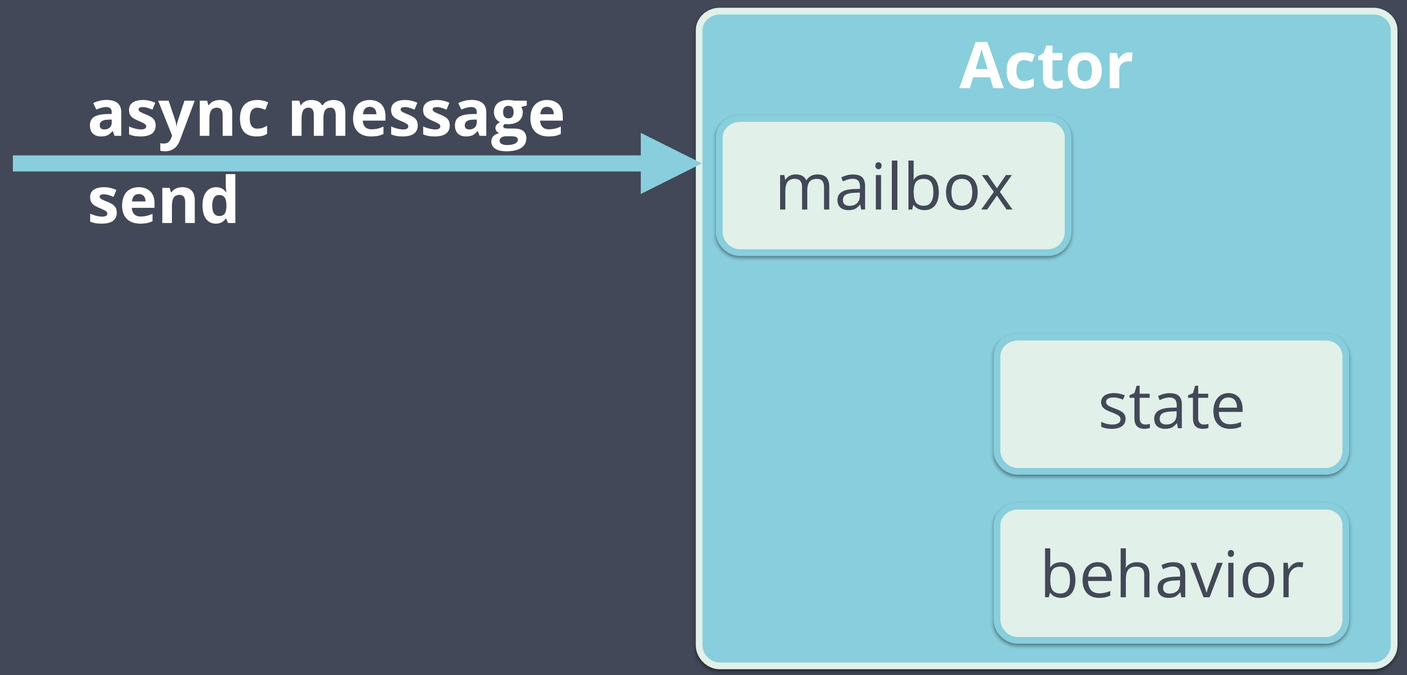


Actors

- Mature and open source
- Scala & Java API
- Akka Cluster

Actors

"an island of consistency in a sea of concurrency"



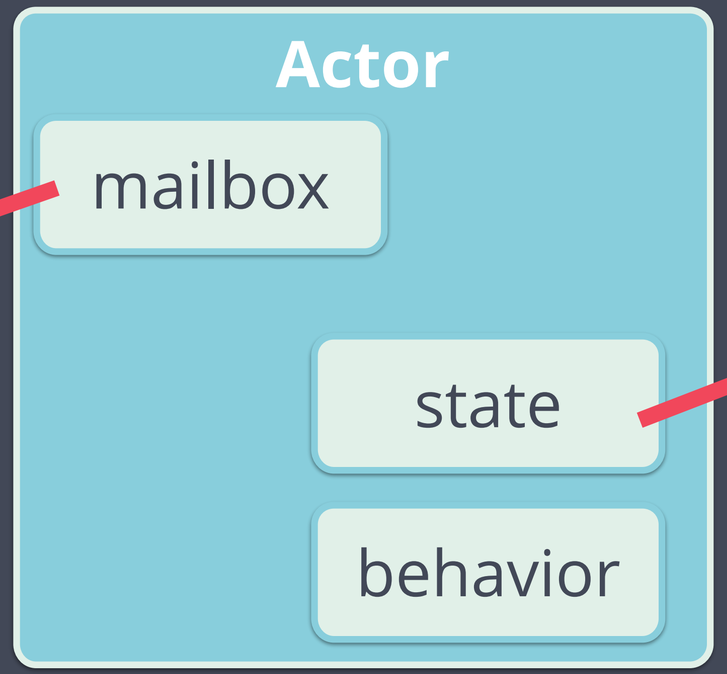
Process message:

- update state
- send messages
- change behavior

Don't worry about concurrency

Actors

A good fit for event-sourcing?

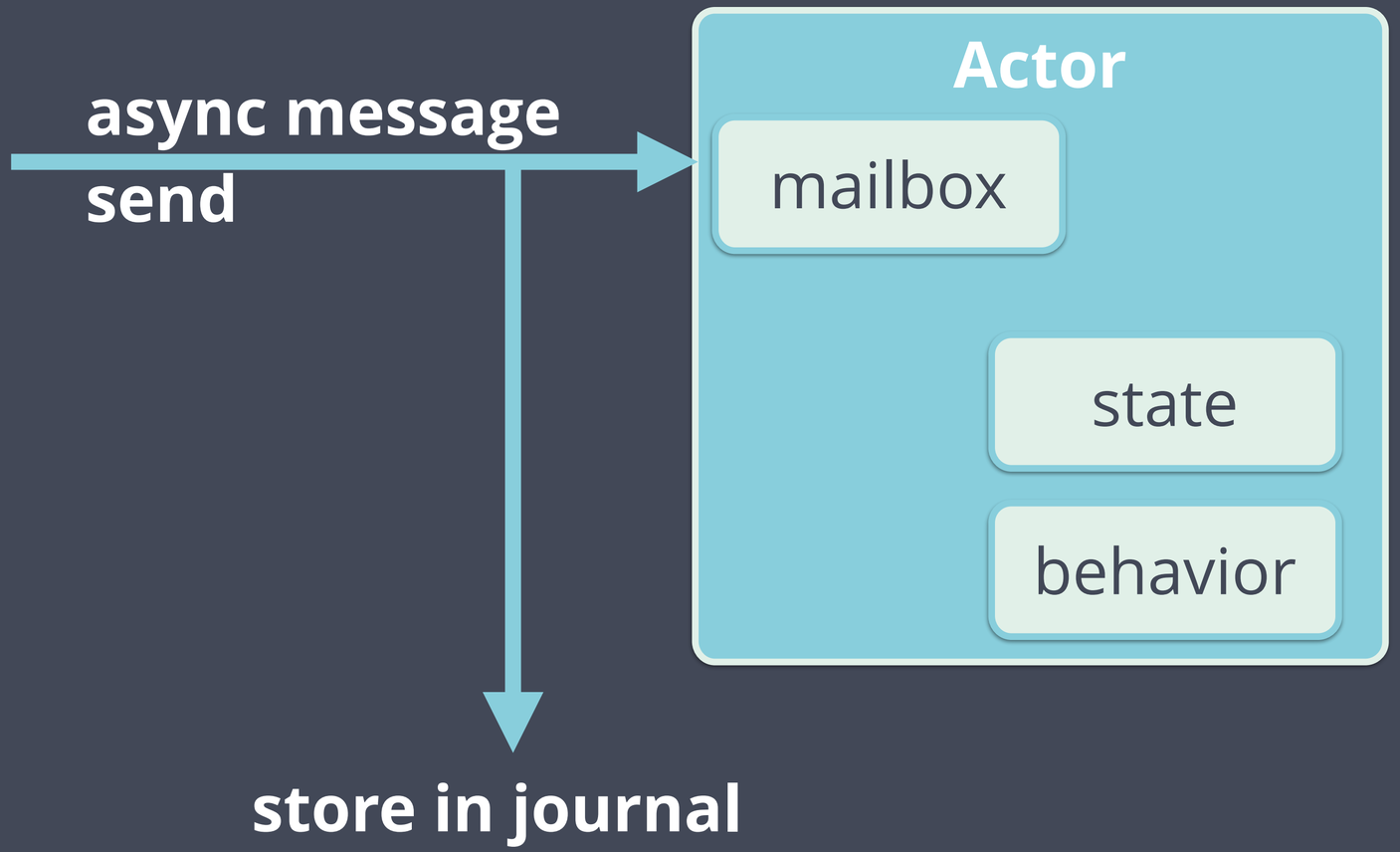


mailbox is non-durable (lost messages)

state is transient

Actors

Just store all incoming messages?



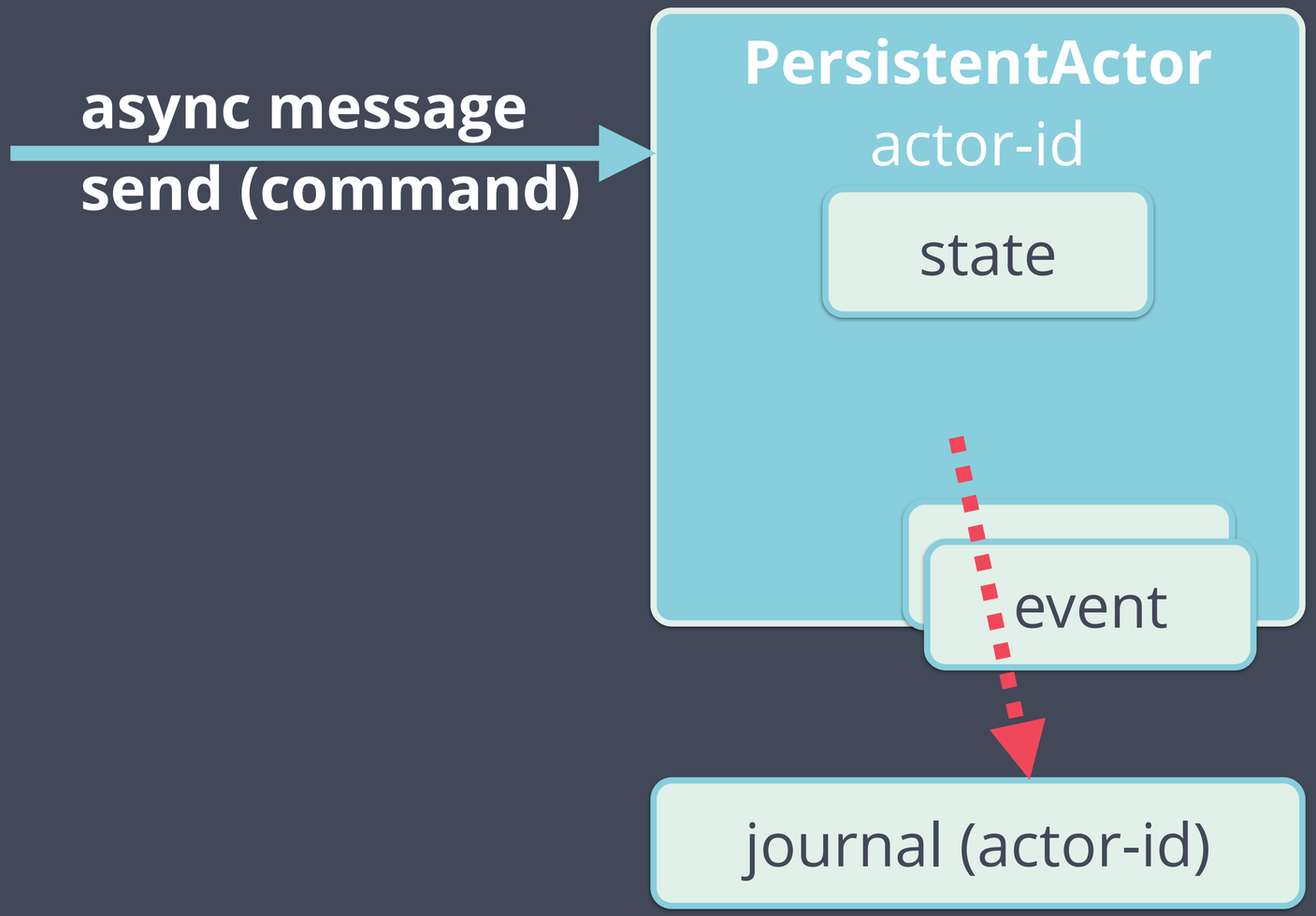
Problems with *command-sourcing:*

- side-effects
- poisonous (failing) messages

Persistence

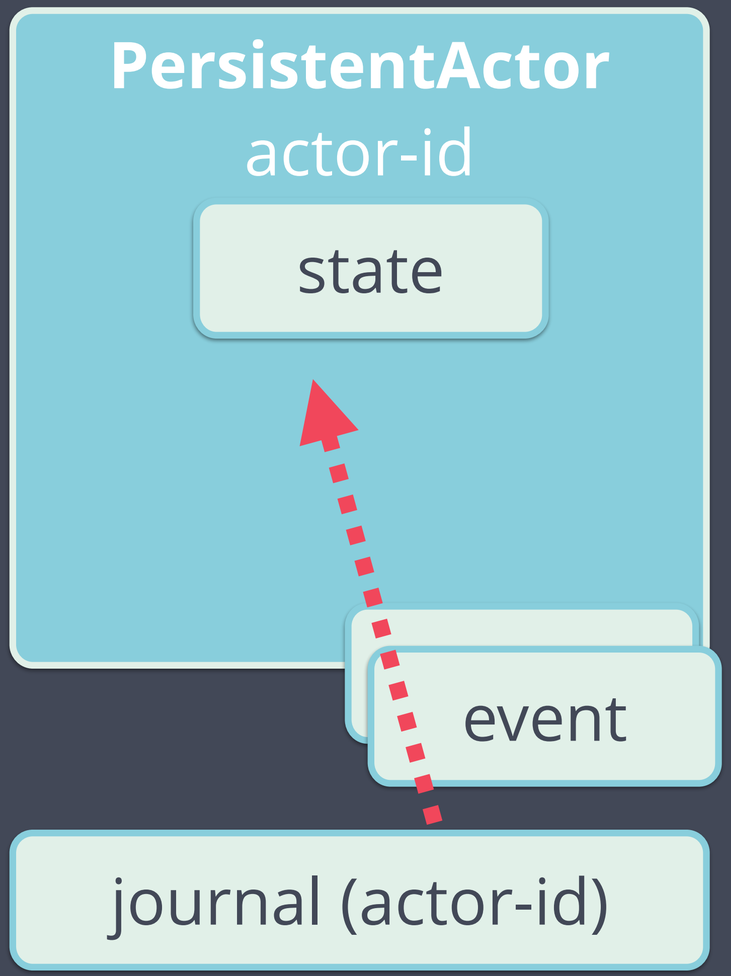
- Experimental Akka module
- Scala & Java API
- Actor state persistence based on event-sourcing

Persistent Actor



- Derive events from commands
- Store events
- Update state
- Perform side-effects

Persistent Actor



Recover by replaying events, that update the state (no side-effects)

Persistent Actor

```
case object Increment      // command
case object Incremented    // event

class CounterActor extends PersistentActor {
  def persistenceId = "counter"

  var state = 0

  val receiveCommand: Receive = {
    case Increment => persist(Incremented) { evt =>
      state += 1
      println("incremented")
    }
  }

  val receiveRecover: Receive = {
    case Incremented => state += 1
  }
}
```

Persistent Actor

```
case object Increment      // command
case object Incremented    // event

class CounterActor extends PersistentActor {
  def persistenceId = "counter"

  var state = 0

  val receiveCommand: Receive = {
    case Increment => persist(Incremented) { evt =>
      state += 1
      println("incremented")
    }
  }

  val receiveRecover: Receive = {
    case Incremented => state += 1
  }
}
```

Persistent Actor

```
case object Increment      // command
case object Incremented    // event

class CounterActor extends PersistentActor {
  def persistenceId = "counter"

  var state = 0

  val receiveCommand: Receive = {
    case Increment => persist(Incremented) { evt =>
      state += 1
      println("incremented")
    }
  }

  val receiveRecover: Receive = {
    case Incremented => state += 1
  }
}
```

async callback
(but safe to close
over state)

Persistent Actor

```
case object Increment      // command
case object Incremented    // event

class CounterActor extends PersistentActor {
  def persistenceId = "counter"

  var state = 0

  val receiveCommand: Receive = {
    case Increment => persist(Incremented) { evt =>
      state += 1
      println("incremented")
    }
  }

  val receiveRecover: Receive = {
    case Incremented => state += 1
  }
}
```

Persistent Actor

```
case object Increment      // command
case object Incremented    // event

class CounterActor extends PersistentActor {
  def persistenceId = "counter"

  var state = 0

  val receiveCommand: Receive = {
    case Increment => persist(Incremented) { evt =>
      state += 1
      println("incremented")
    }
  }

  val receiveRecover: Receive = {
    case Incremented => state += 1
  }
}
```

Isn't recovery with lots of events slow?

Snapshots

```
class SnapshottingCounterActor extends PersistentActor {
  def persistenceId = "snapshotting-counter"

  var state = 0

  val receiveCommand: Receive = {
    case Increment => persist(Incremented) { evt =>
      state += 1
      println("incremented")
    }
    case "takesnapshot" => saveSnapshot(state)
  }

  val receiveRecover: Receive = {
    case Incremented => state += 1
    case SnapshotOffer(_, snapshotState: Int) => state = snapshotState
  }
}
```

Snapshots

```
class SnapshottingCounterActor extends PersistentActor {
  def persistenceId = "snapshotting-counter"

  var state = 0

  val receiveCommand: Receive = {
    case Increment => persist(Incremented) { evt =>
      state += 1
      println("incremented")
    }
    case "takesnapshot" => saveSnapshot(state)
  }

  val receiveRecover: Receive = {
    case Incremented => state += 1
    case SnapshotOffer(_, snapshotState: Int) => state = snapshotState
  }
}
```

Plugins: Journal & Snapshot

Journal	Snapshot
Cassandra	Cassandra
Kafka	Kafka
DynamoDB	
MongoDB	MongoDB
HBase	HBase
MapDB	
JDBC	JDBC

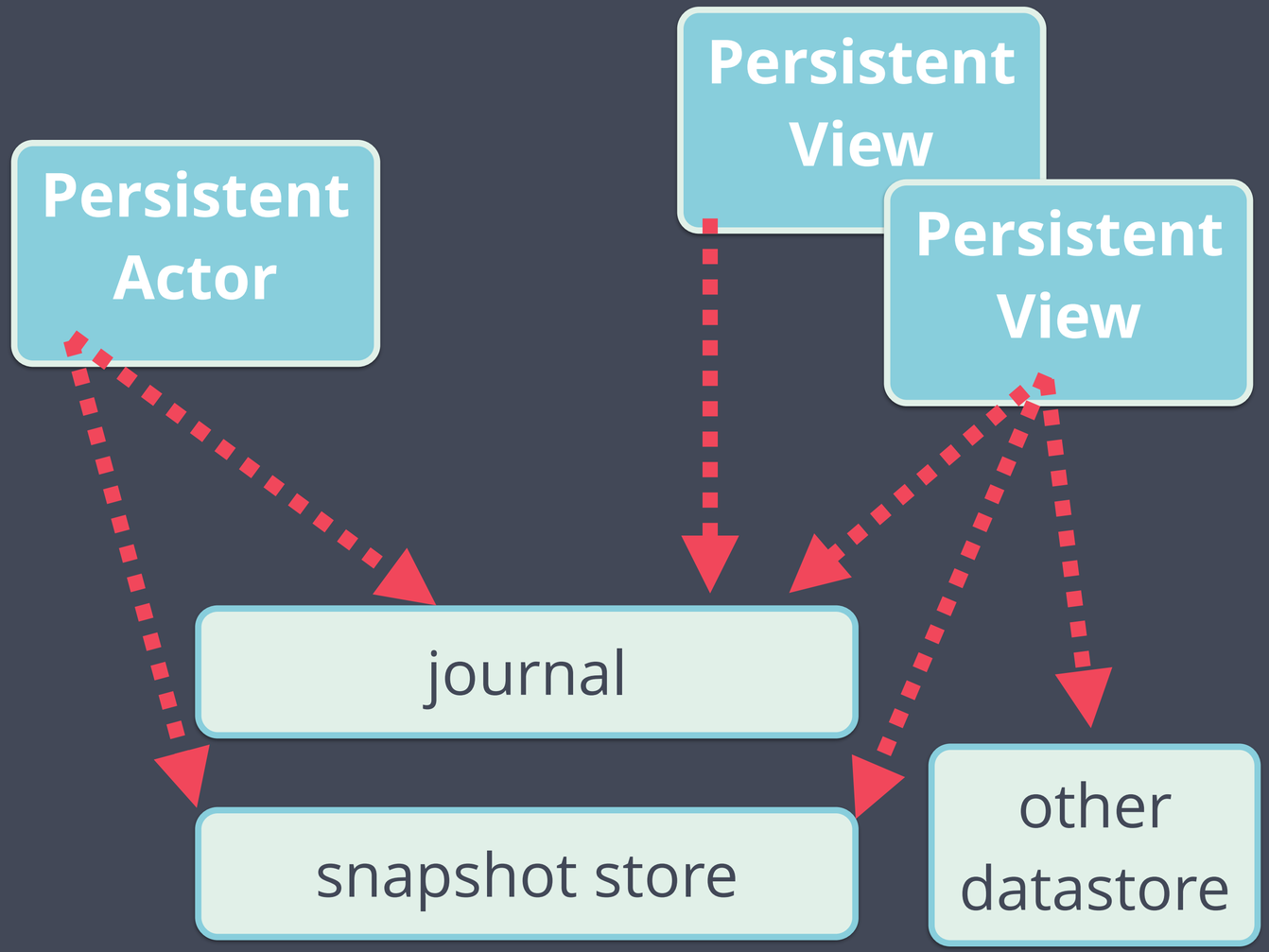
Plugins: Serialization

Default: Java serialization

Pluggable through Akka:

- Protobuf
- Kryo
- Avro
- Your own

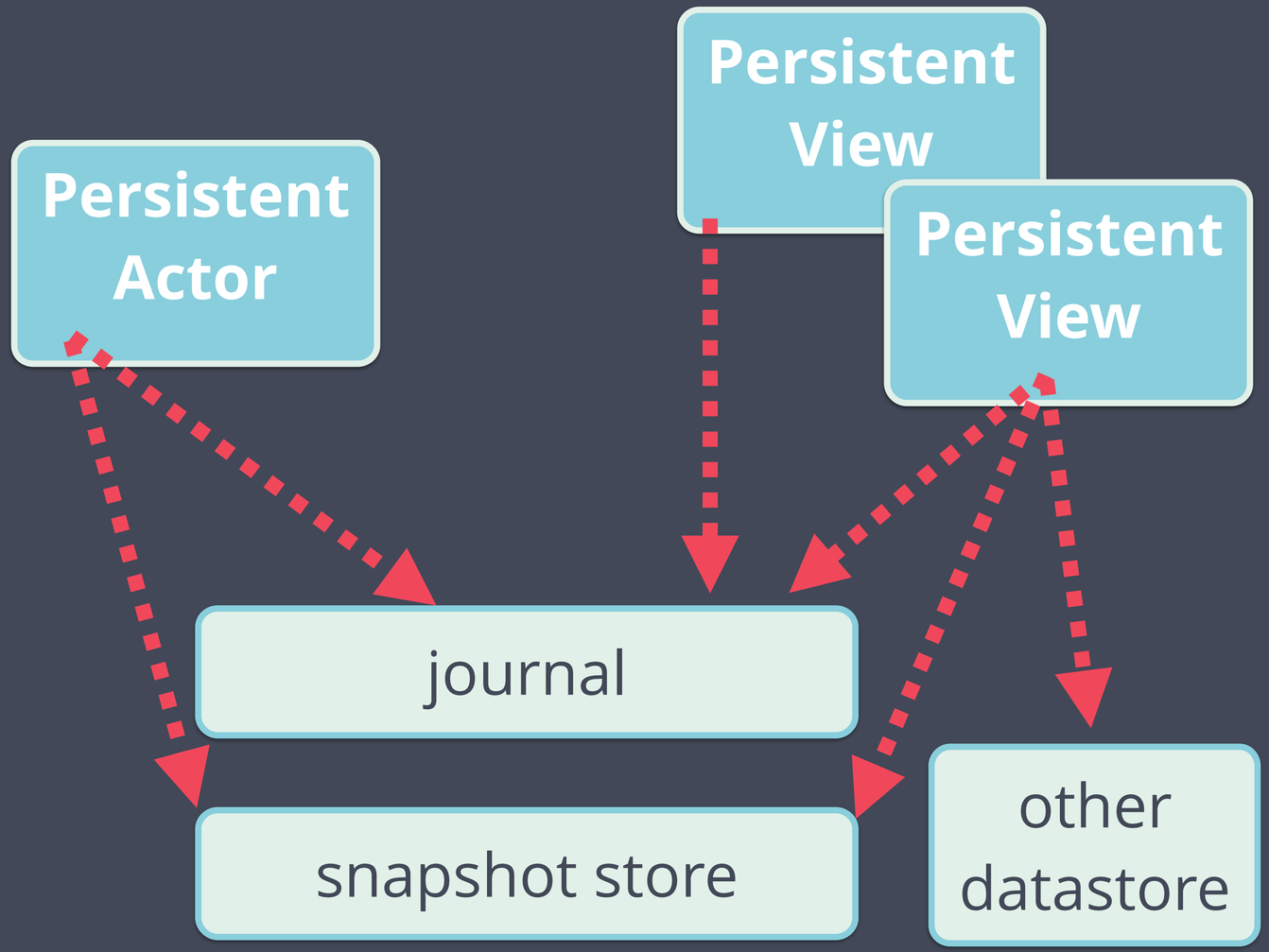
Persistent View



Views poll the journal

- Eventually consistent
- Polling configurable
- Actor may be inactive
- Views track single persistence-id
- Views can have own snapshots

Persistent View



"The Database is a cache of a subset of the log"

- Pat Helland

Persistent View

```
case object ComplexQuery
class CounterView extends PersistentView {
  override def persistenceId: String = "counter"
  override def viewId: String = "counter-view"

  var queryState = 0

  def receive: Receive = {
    case Incremented if isPersistent => {
      queryState = someVeryComplicatedCalculation(queryState)
      // Or update a document/graph/relational database
    }
    case ComplexQuery                => {
      sender() ! queryState;
      // Or perform specialized query on datastore
    }
  }
 }
```

Persistent View

```
case object ComplexQuery
class CounterView extends PersistentView {
  override def persistenceId: String = "counter"
  override def viewId: String = "counter-view"

  var queryState = 0

  def receive: Receive = {
    case Incremented if isPersistent => {
      queryState = someVeryComplicatedCalculation(queryState)
      // Or update a document/graph/relational database
    }
    case ComplexQuery                => {
      sender() ! queryState;
      // Or perform specialized query on datastore
    }
  }
 }
```

Persistent View

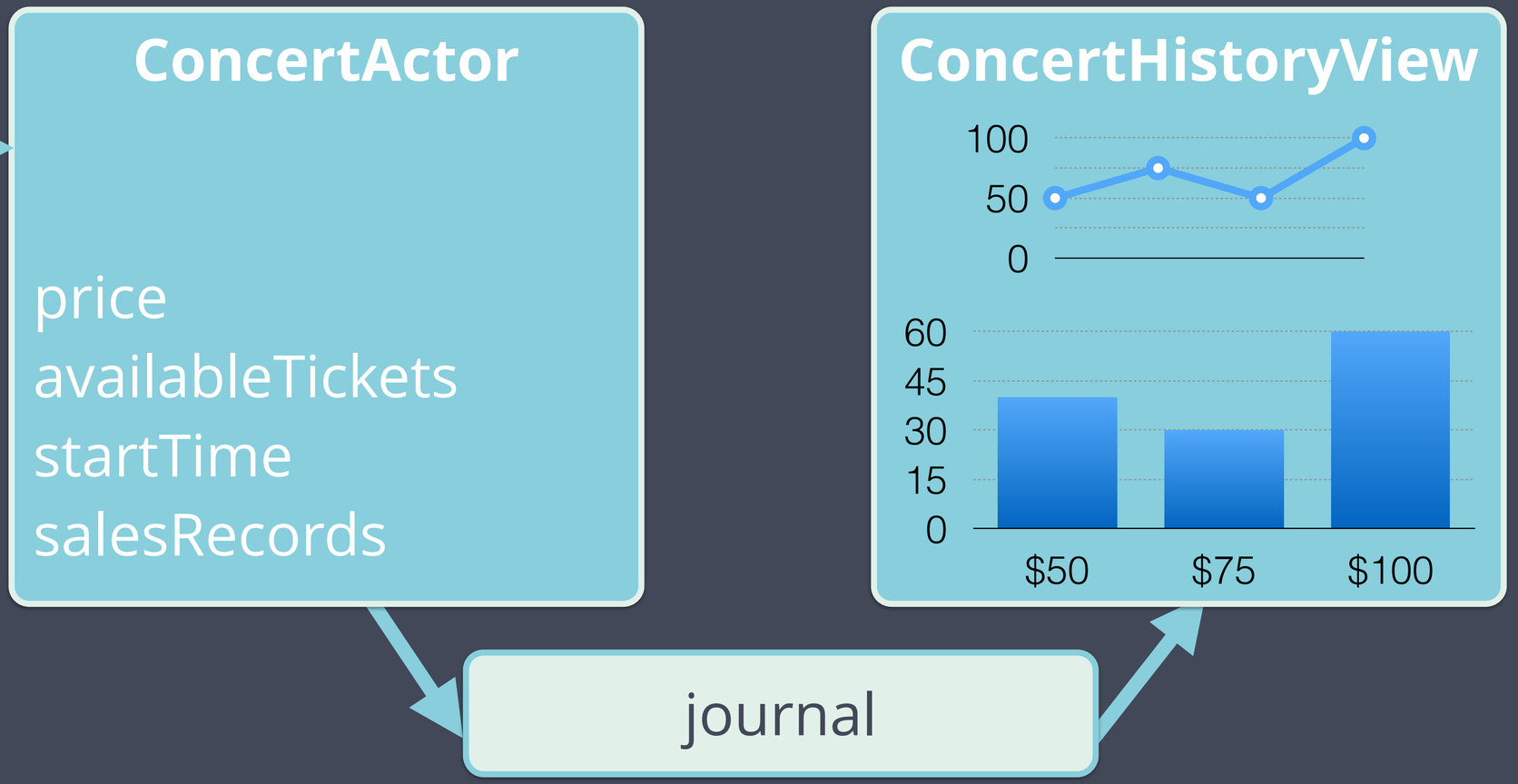
```
case object ComplexQuery
class CounterView extends PersistentView {
  override def persistenceId: String = "counter"
  override def viewId: String = "counter-view"

  var queryState = 0

  def receive: Receive = {
    case Incremented if isPersistent => {
      queryState = someVeryComplicatedCalculation(queryState)
      // Or update a document/graph/relational database
    }
    case ComplexQuery                => {
      sender() ! queryState;
      // Or perform specialized query on datastore
    }
  }
 }
```

Sell concert tickets

code @ bit.ly/akka-es

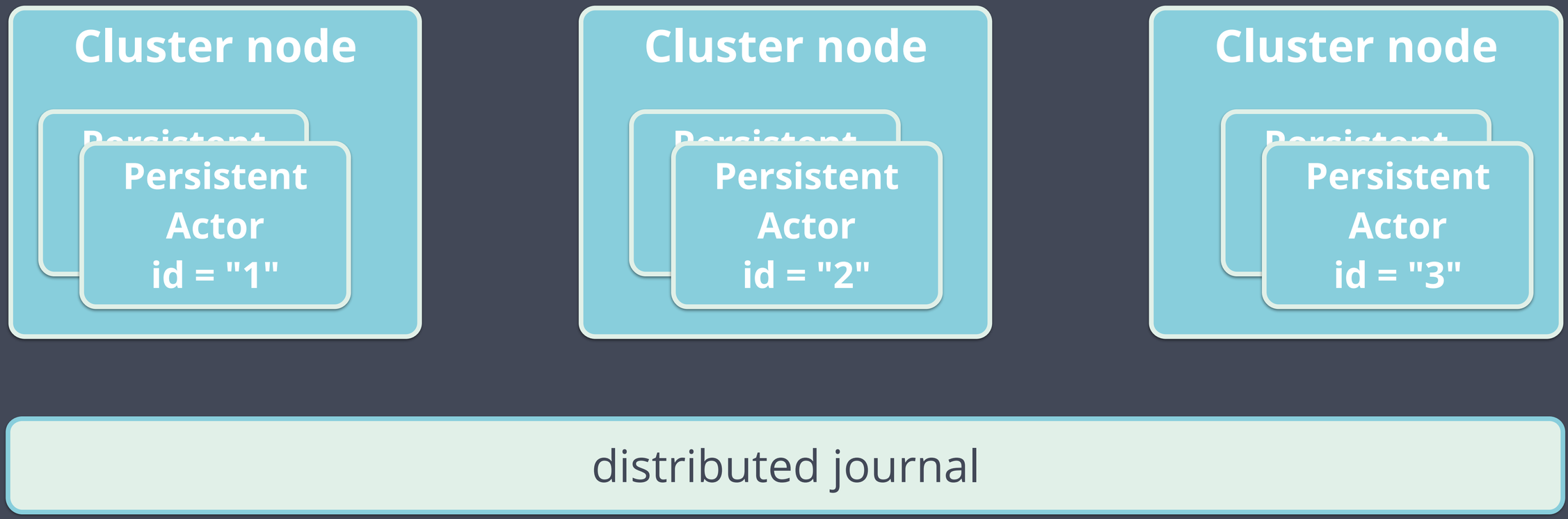
Commands:

CreateConcert
BuyTickets
ChangePrice
AddCapacity



Scaling out: Akka Cluster

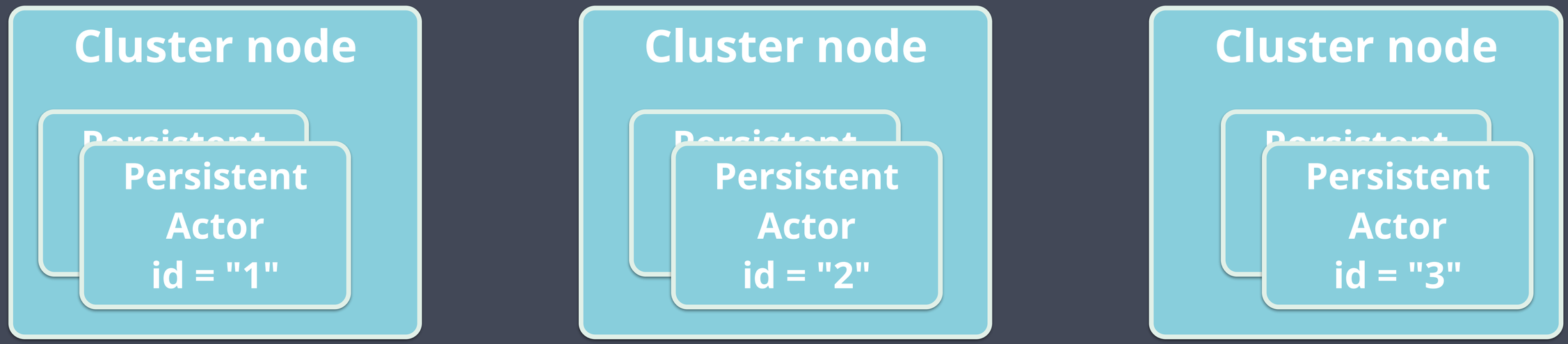
Single writer: persistent actor must be singleton, views may be anywhere



Scaling out: Akka Cluster

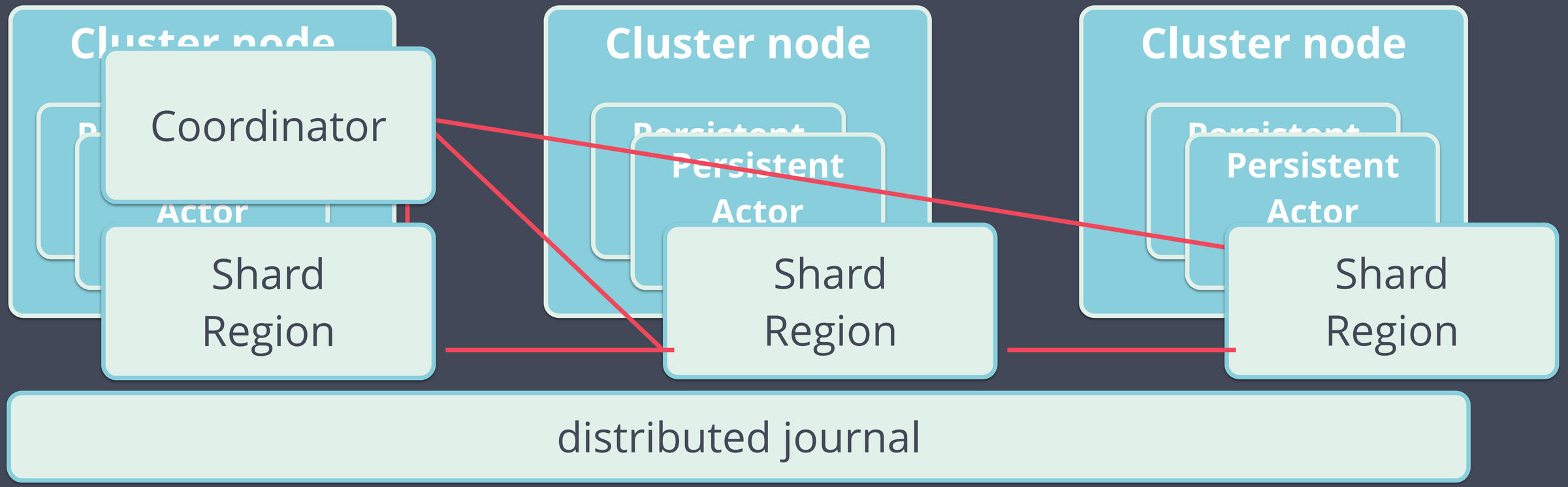
Single writer: persistent actor must be singleton, views may be anywhere

How are persistent actors distributed over cluster?



Scaling out: Akka Cluster

Sharding: Coordinator assigns ShardRegions to nodes (consistent hashing)
Actors in shard can be activated/passivated, rebalanced



Scaling out: Sharding

```
val idExtractor: ShardRegion.IdExtractor = {
    case cmd: Command => (cmd.concertId, cmd)
}
```

IdExtractor allows ShardRegion to route commands to actors

Scaling out: Sharding

```
val idExtractor: ShardRegion.IdExtractor = {
    case cmd: Command => (cmd.concertId, cmd)
}
```

IdExtractor allows ShardRegion to route commands to actors

```
val shardResolver: ShardRegion.ShardResolver =
    msg => msg match {
        case cmd: Command => hash(cmd.concert)
    }
```

ShardResolver assigns new actors to shards

Scaling out: Sharding

```
val idExtractor: ShardRegion.IdExtractor = {
   case cmd: Command => (cmd.concertId, cmd)
}
```

IdExtractor allows ShardRegion to route commands to actors

```
val shardResolver: ShardRegion.ShardResolver =
   msg => msg match {
      case cmd: Command => hash(cmd.concert)
   }
```

ShardResolver assigns new actors to shards

```
ClusterSharding(system).start(
 typeName = "Concert",
 entryProps = Some(ConcertActor.props()),
 idExtractor = ConcertActor.idExtractor,
 shardResolver = ConcertActor.shardResolver)
```

Initialize ClusterSharding extension

Scaling out: Sharding

```
val idExtractor: ShardRegion.IdExtractor = {
   case cmd: Command => (cmd.concertId, cmd)
}
```

IdExtractor allows ShardRegion to route commands to actors

```
val shardResolver: ShardRegion.ShardResolver =
   msg => msg match {
      case cmd: Command => hash(cmd.concert)
   }
```

ShardResolver assigns new actors to shards

```
ClusterSharding(system).start(
 typeName = "Concert",
 entryProps = Some(ConcertActor.props()),
 idExtractor = ConcertActor.idExtractor,
 shardResolver = ConcertActor.shardResolver)
```

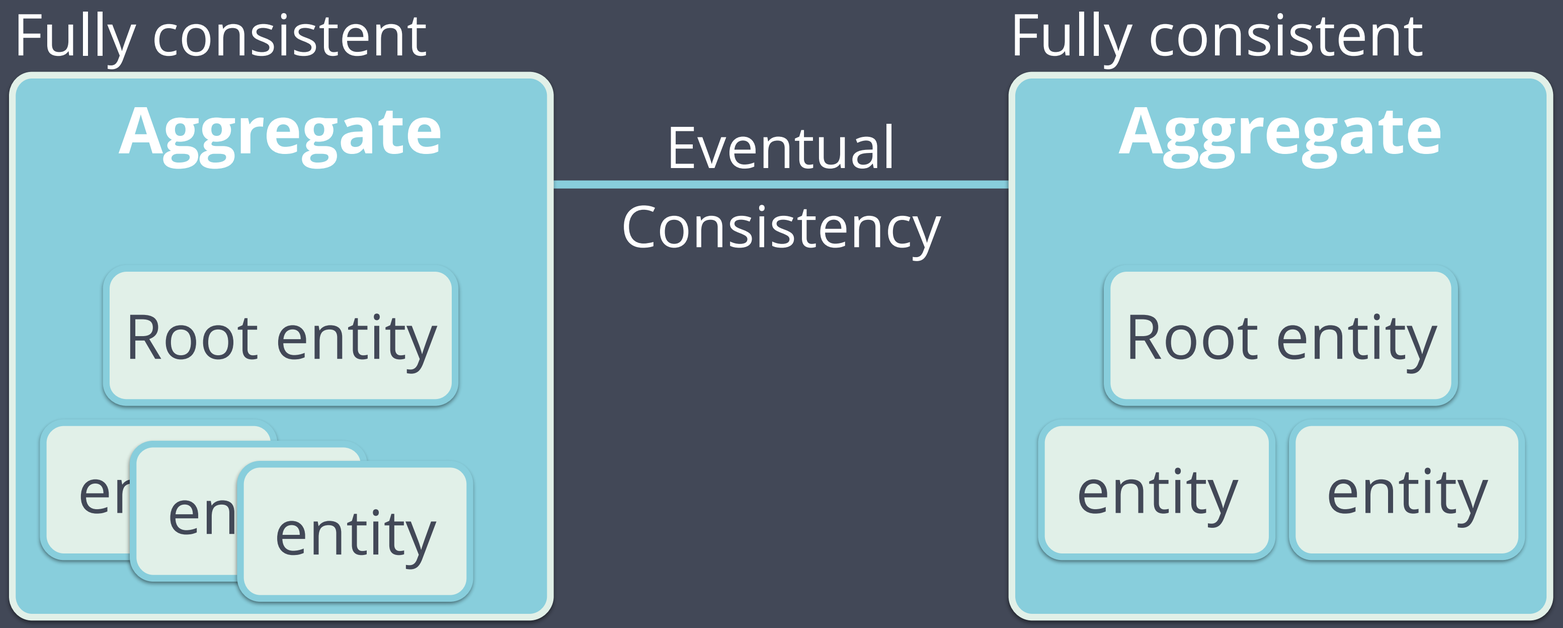
Initialize ClusterSharding extension

```
val concertRegion: ActorRef = ClusterSharding(system).shardRegion("Concert")

concertRegion ! BuyTickets(concertId = 123, user = "Sander", quantity = 1)
```

Design for event-sourcing

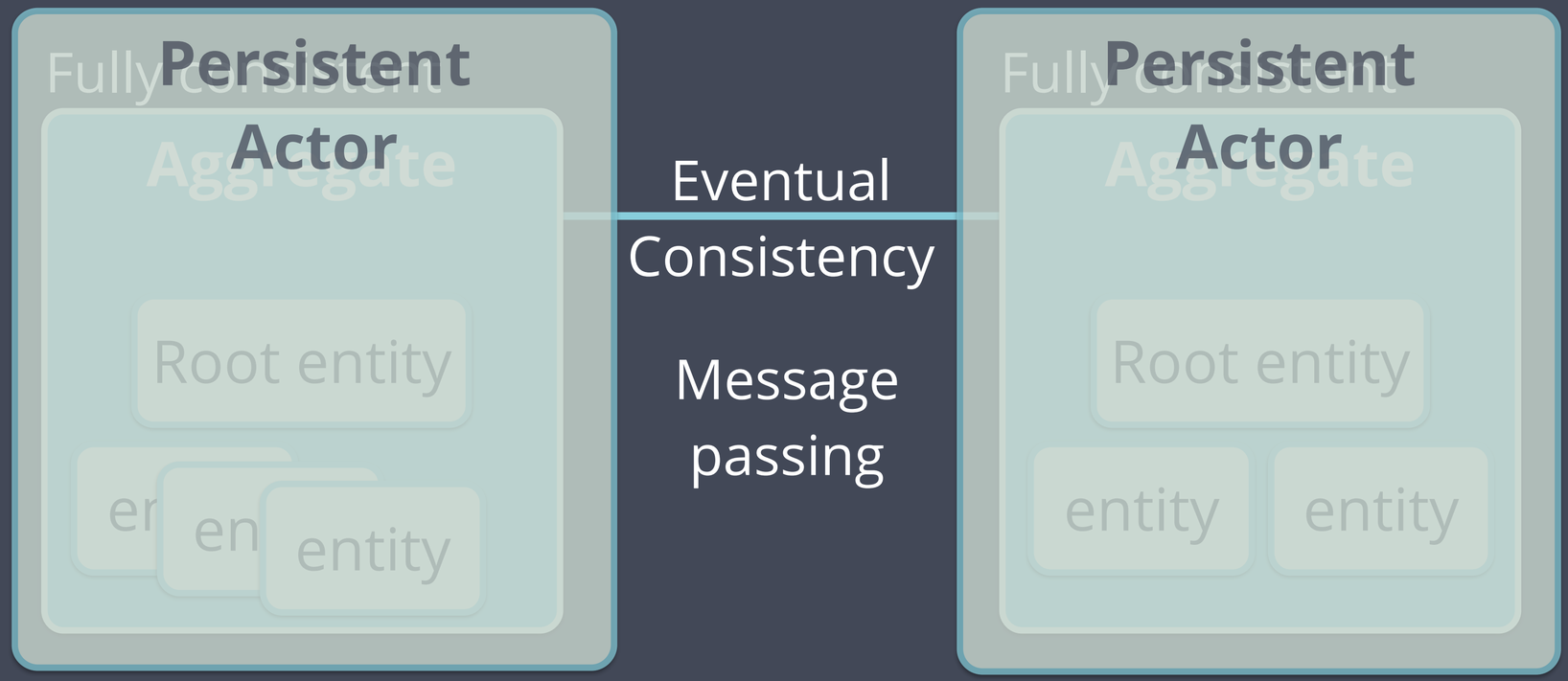
DDD+CQRS+ES

DDD: Domain Driven Design



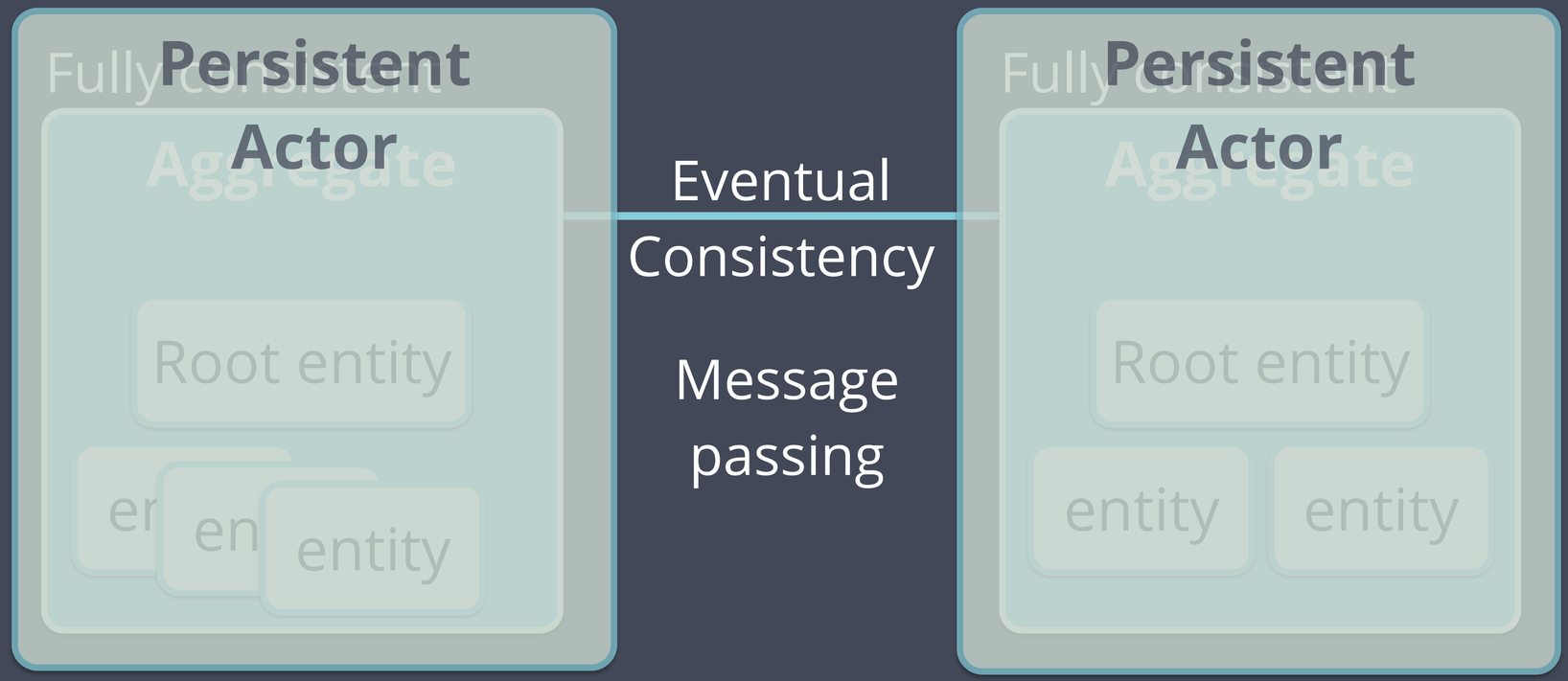
DDD+CQRS+ES

DDD: Domain Driven Design



DDD+CQRS+ES

DDD: Domain Driven Design



Akka Persistence is **not** a DDD/CQRS framework

But it comes awfully close

Designing aggregates

Focus on **events**

Structural representation(s) follow

Designing aggregates

Focus on **events**
Structural representation(s) follow

Size matters. Faster replay, less write contention
Don't store derived info (use views)

Designing aggregates

Focus on **events**
Structural representation(s) follow

Size matters. Faster replay, less write contention
Don't store derived info (use views)

With CQRS **read-your-writes** is not the default
When you need this, model it in a single Aggregate

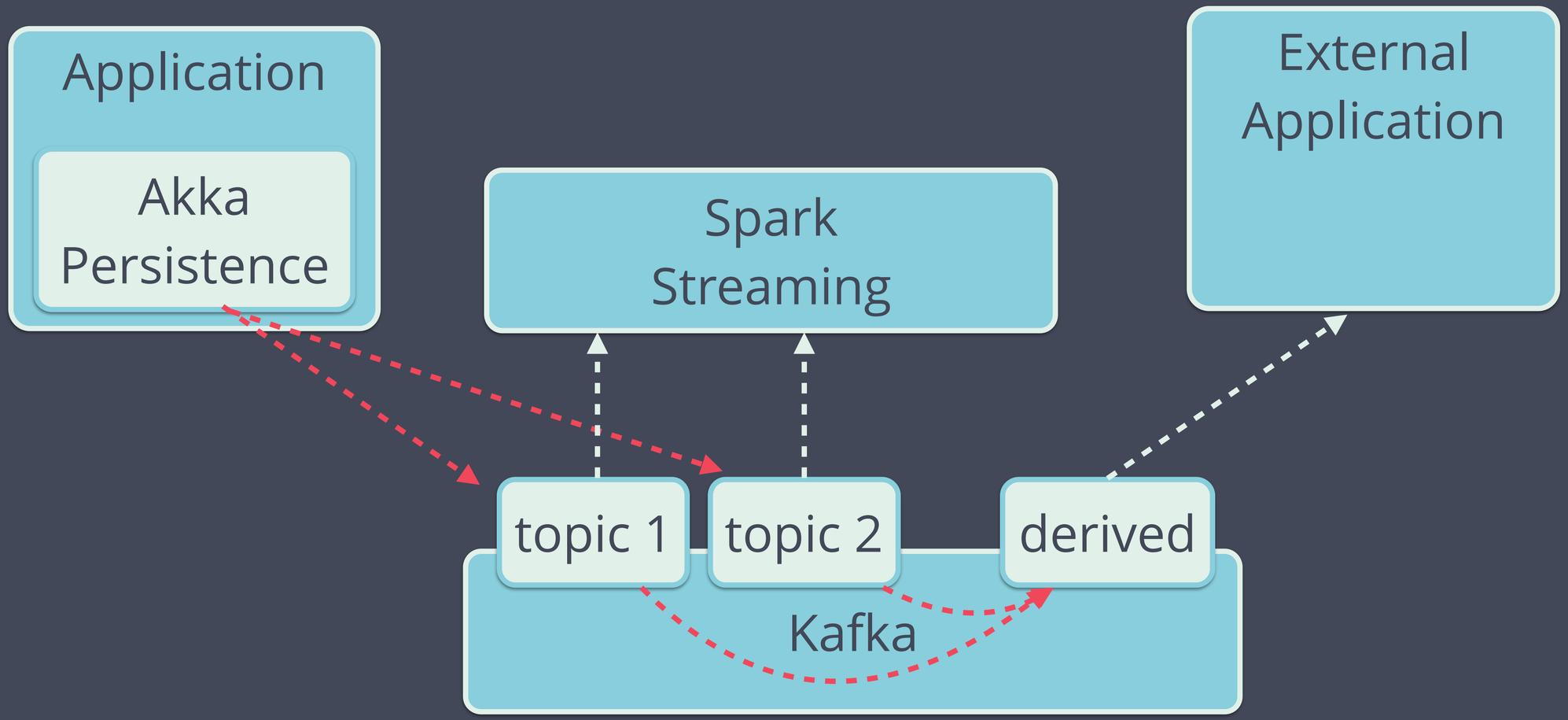
Between aggregates

- Send commands to other aggregates by id

- No distributed transactions
- Use business level ack/nacks
  - SendInvoice -> InvoiceSent/InvoiceCouldNotBeSent
  - Compensating actions for failure
- No guaranteed delivery
- Alternative: *AtLeastOnceDelivery*

Between systems

System integration through event-sourcing



Designing commands

- Self-contained
- Unit of atomic change
- Granularity and intent

Designing commands

- Self-contained
- Unit of atomic change
- Granularity and intent

UpdateAddress
street = …
city = …

vs

ChangeStreet
street = …

ChangeCity
street = …

Designing commands

- Self-contained
- Unit of atomic change
- Granularity and intent

UpdateAddress
street = …
city = …

vs

ChangeStreet
street = …

ChangeCity
street = …

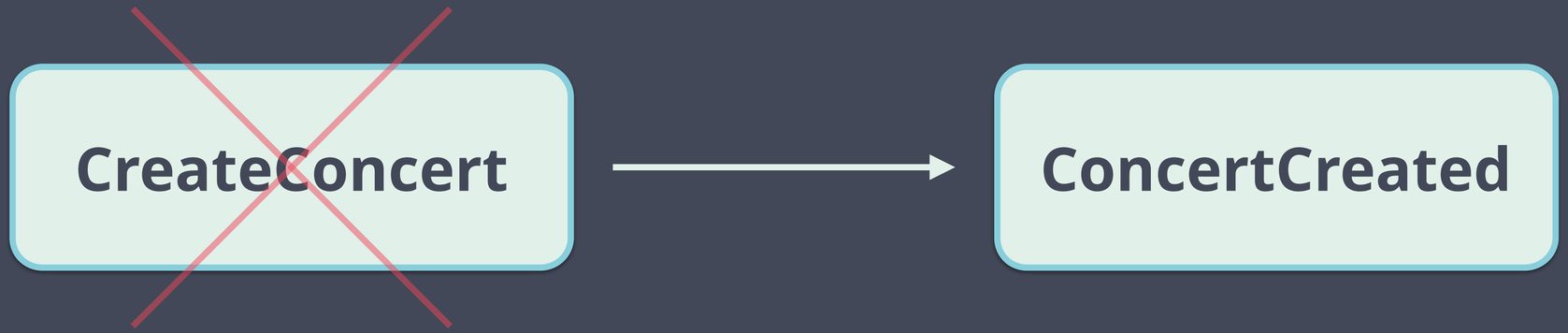
vs

Move
street = …
city = …

Designing events

CreateConcert

Designing events



Past tense
Irrefutable

Designing events

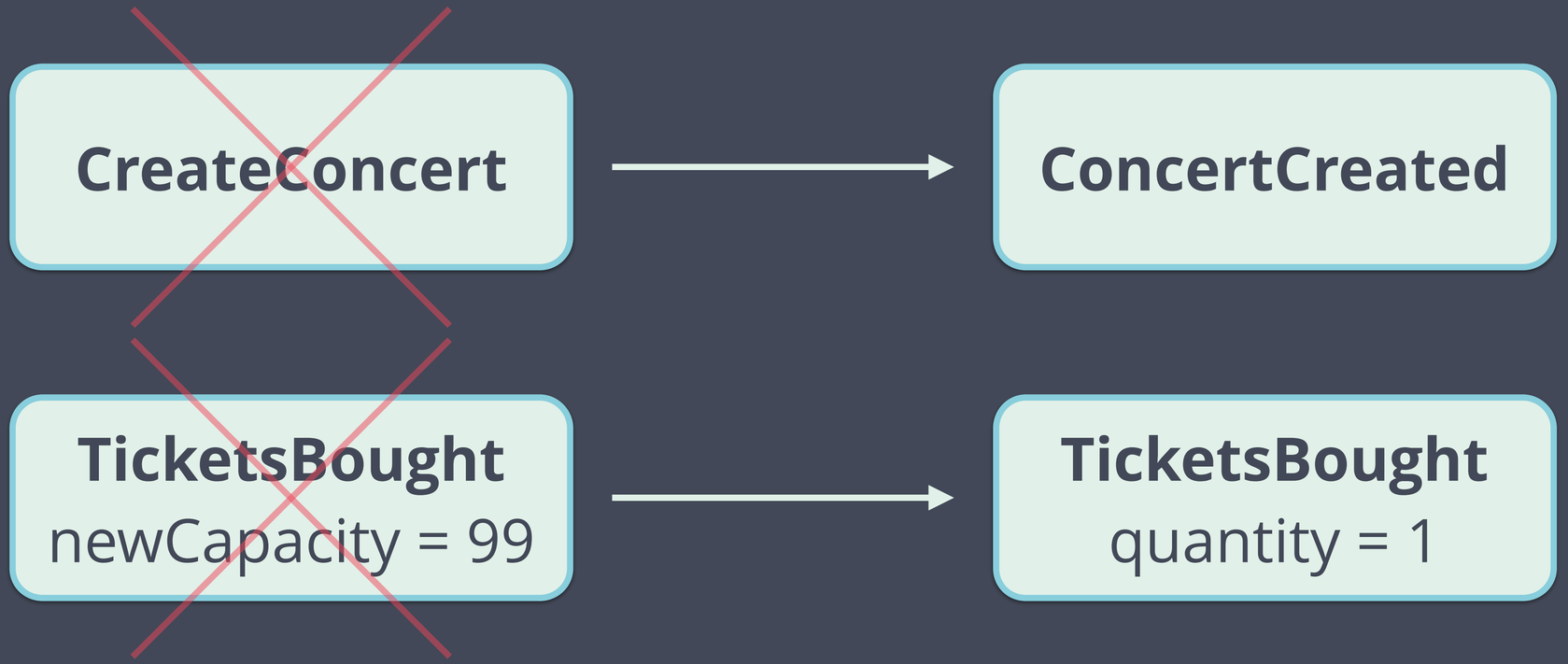
~~**CreateConcert**~~

ConcertCreated

Past tense
Irrefutable

TicketsBought
newCapacity = 99

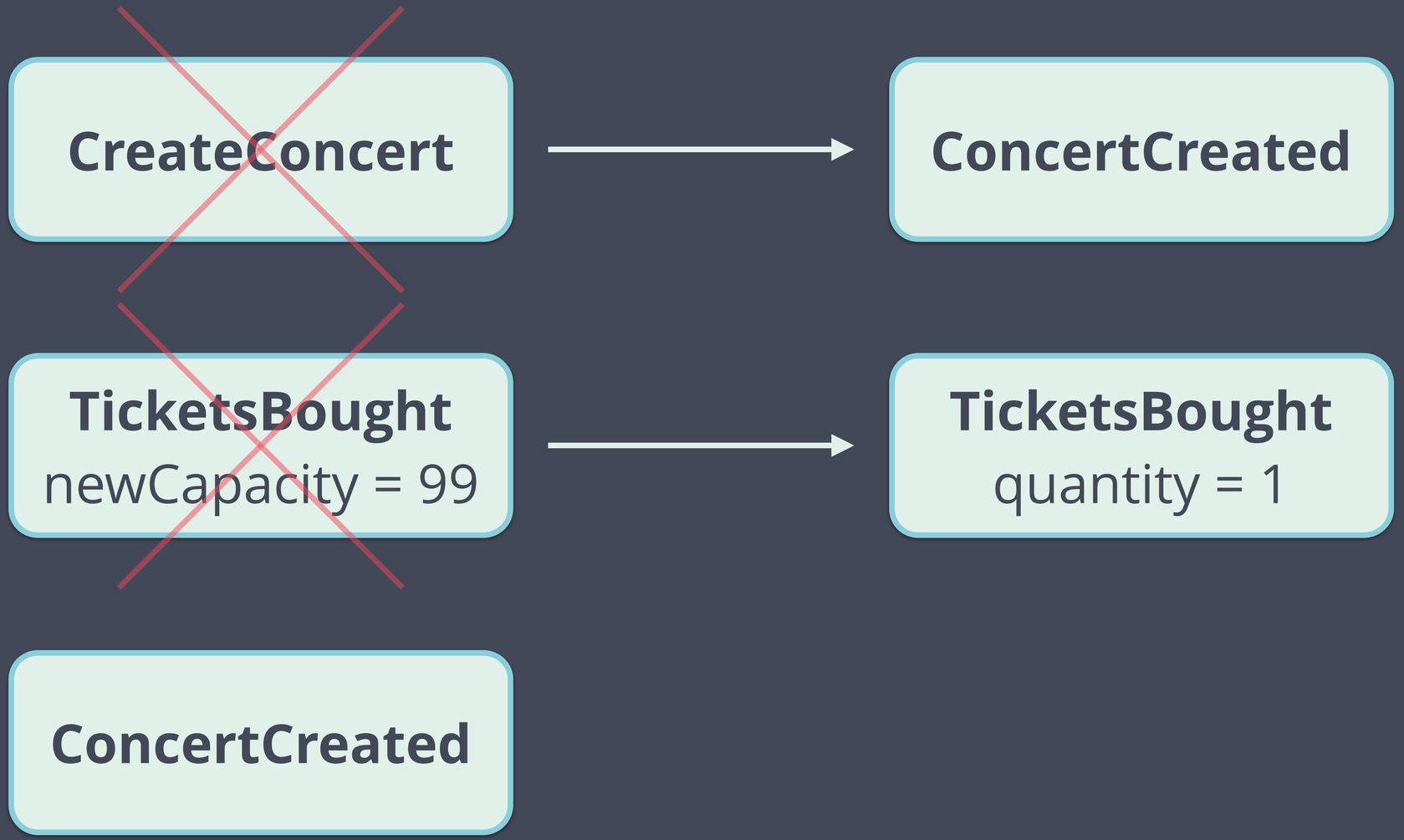
Designing events



Past tense
Irrefutable

Delta-based
No derived info

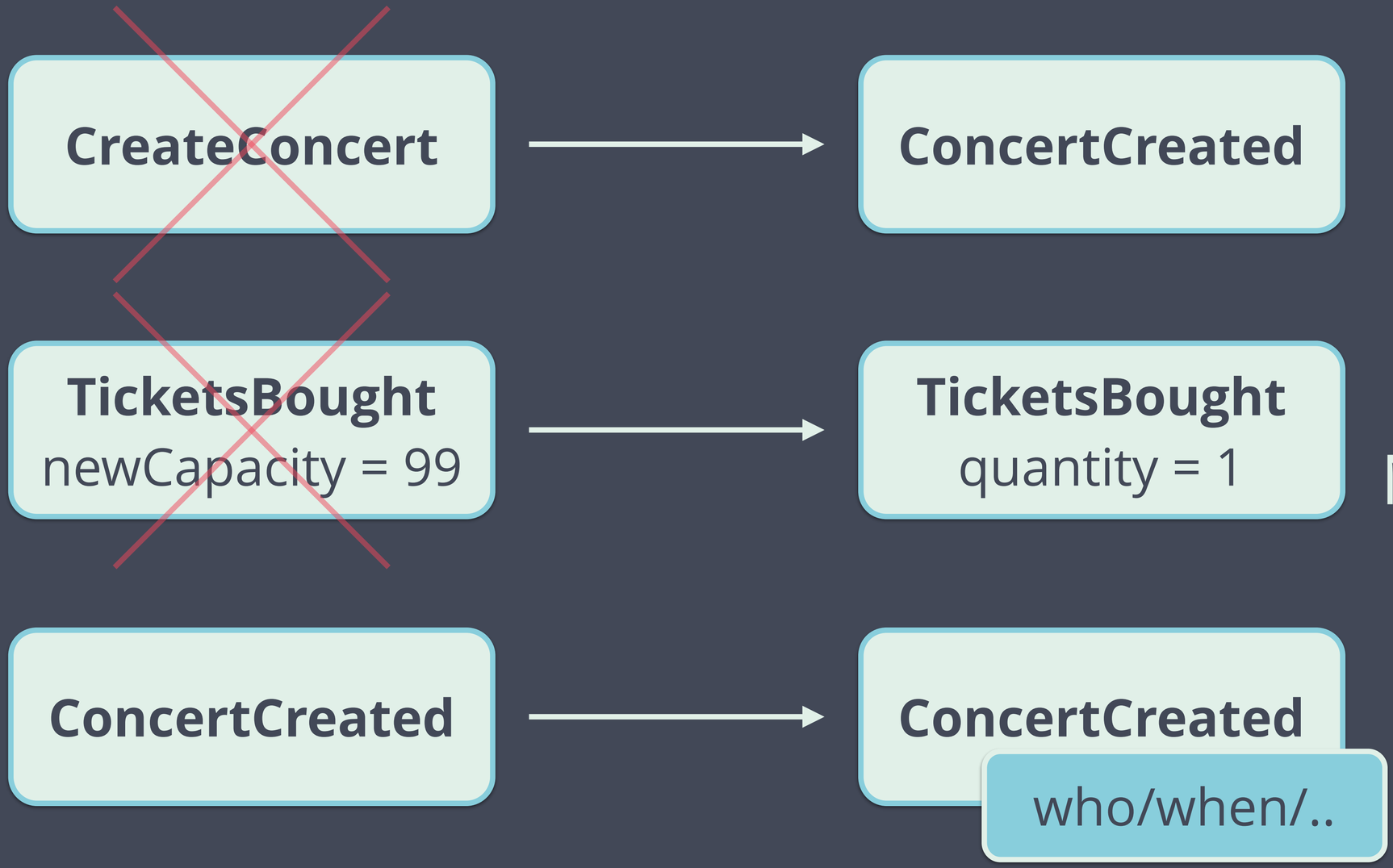
Designing events



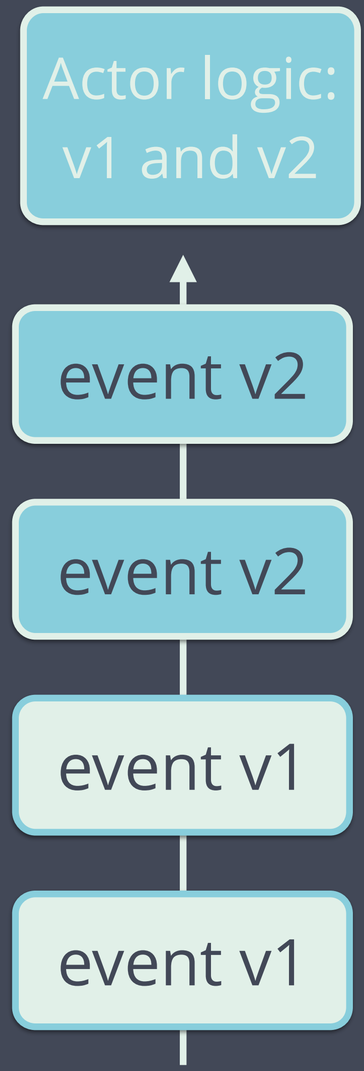
Past tense
Irrefutable

Delta-based
No derived info

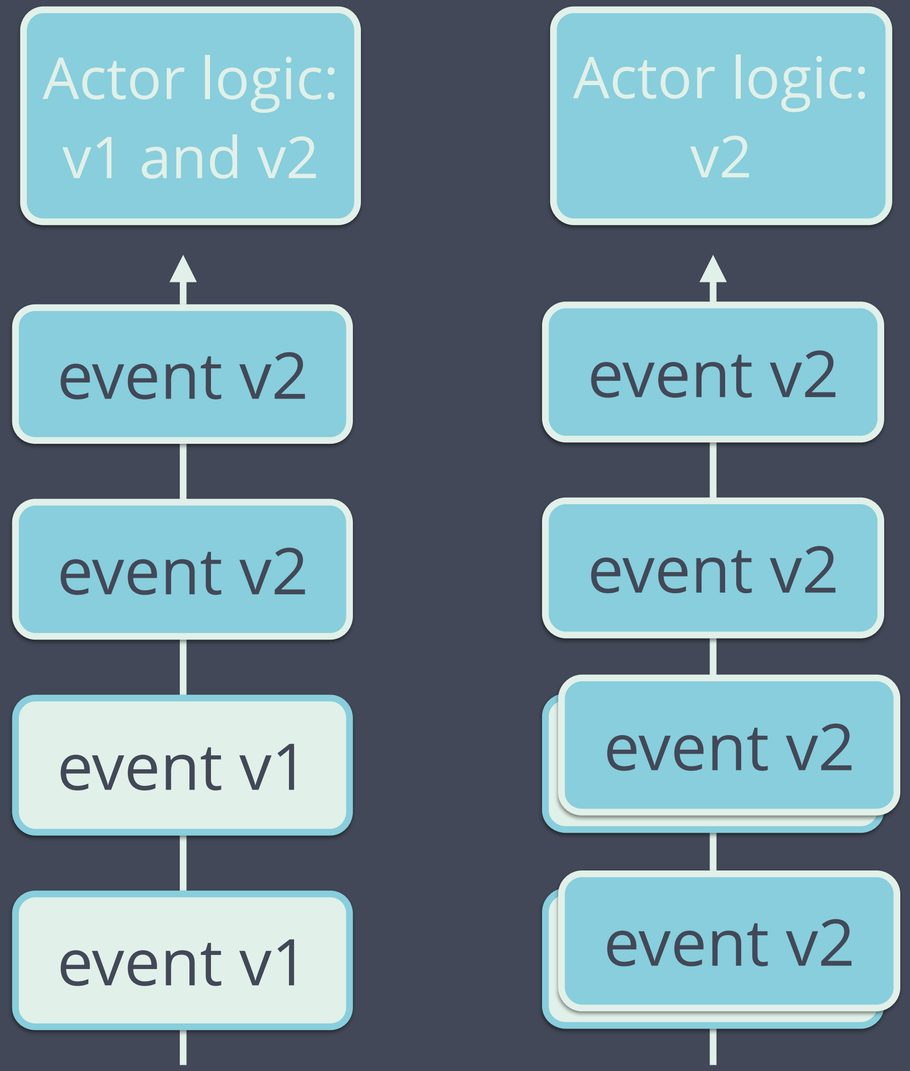
Designing events



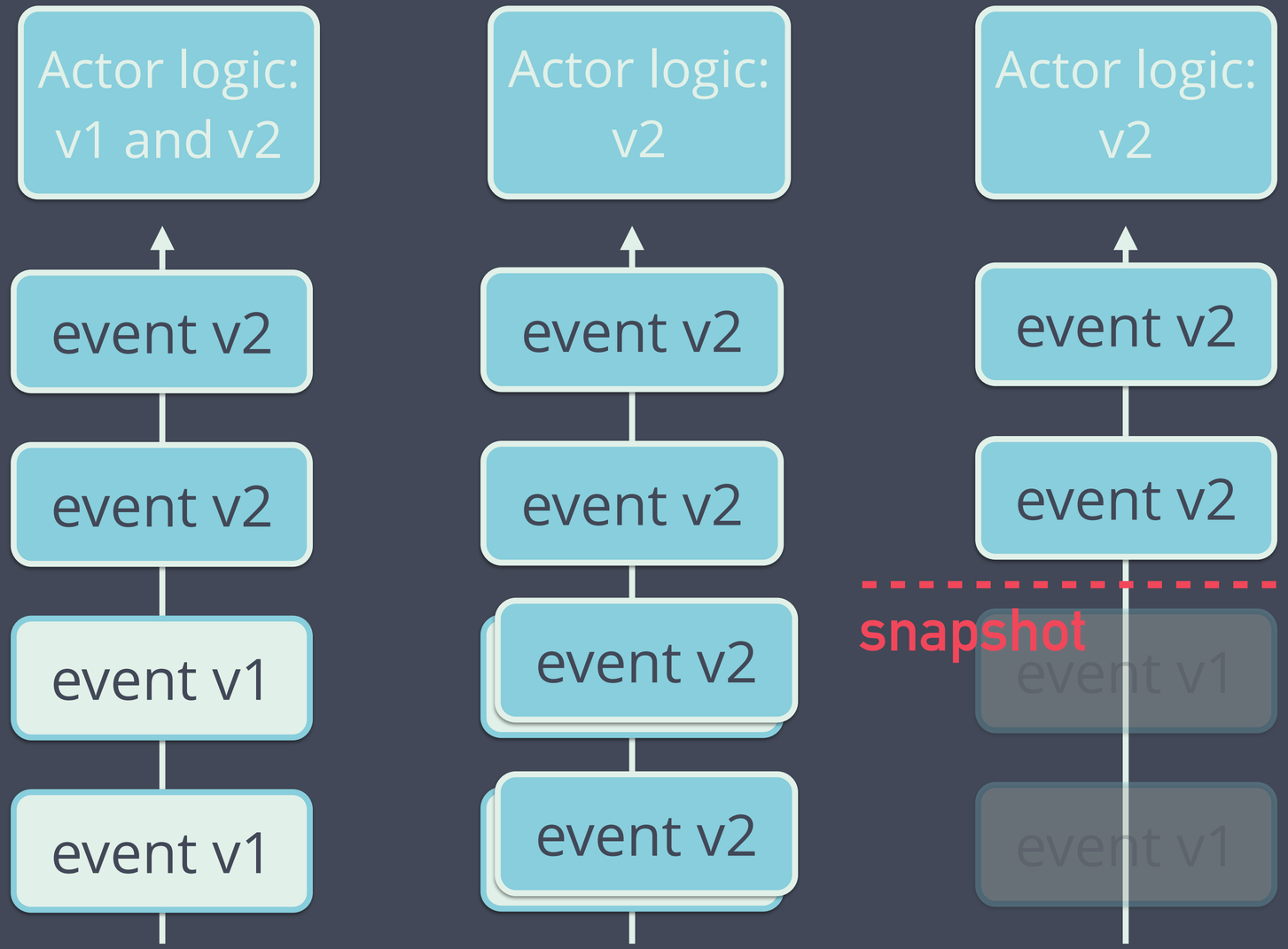
Versioning



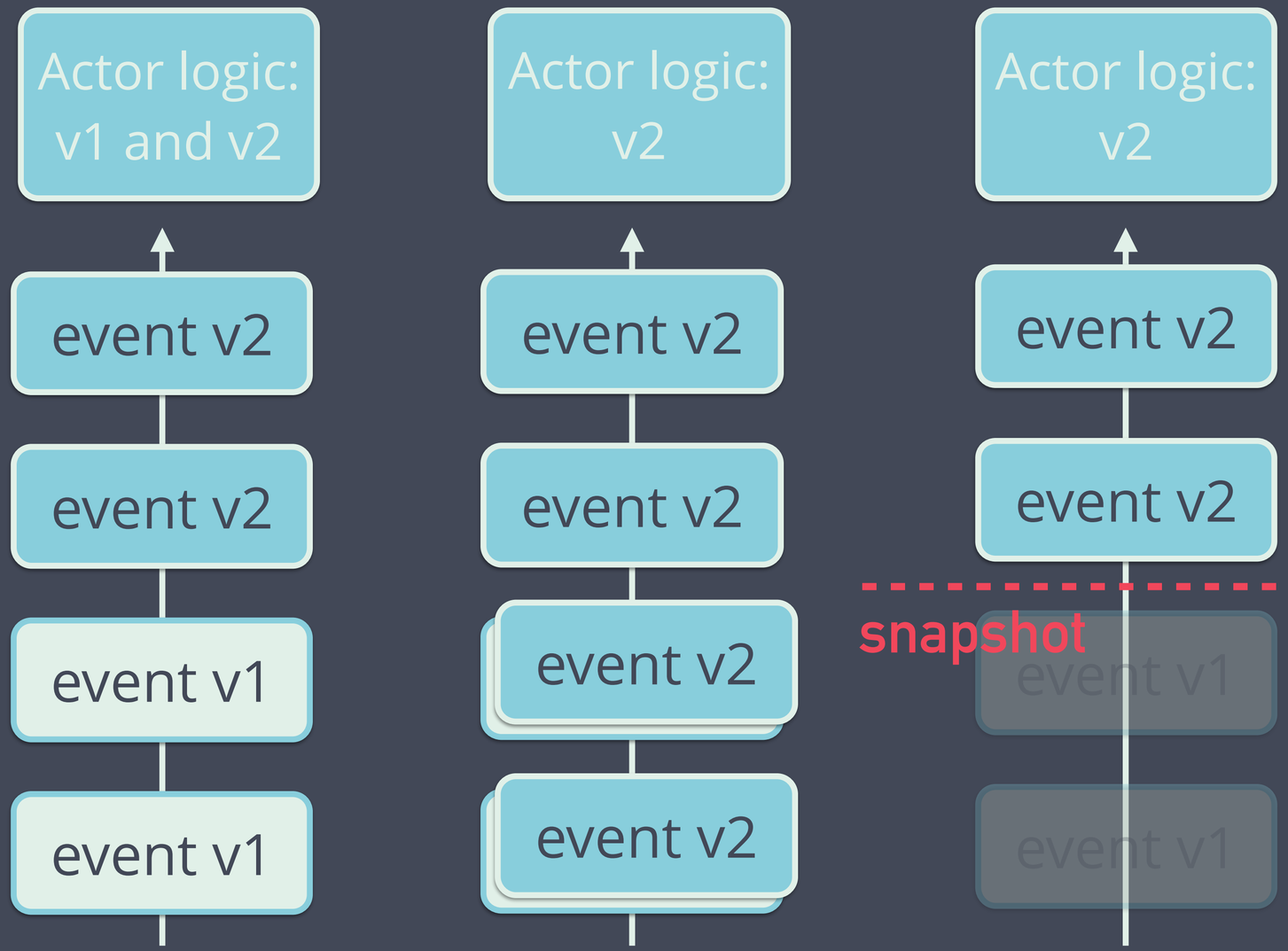
Versioning


Actor logic:
v1 and v2
event v2
event v2
event v1
event v1
Actor logic:
v2
event v2
event v2
event v2
event v2

Versioning


Actor logic: v1 and v2
Actor logic: v2
Actor logic: v2
event v2
event v2
event v1
event v1
event v2
event v2
event v2
event v2
event v2
event v2
snapshot
event v1
event v1

Versioning



- Be backwards compatible
- Avro/Protobuf
- Serializer can do translation

- Snapshot versioning: harder

In conclusion

Event-sourcing is...

Powerful

but unfamiliar

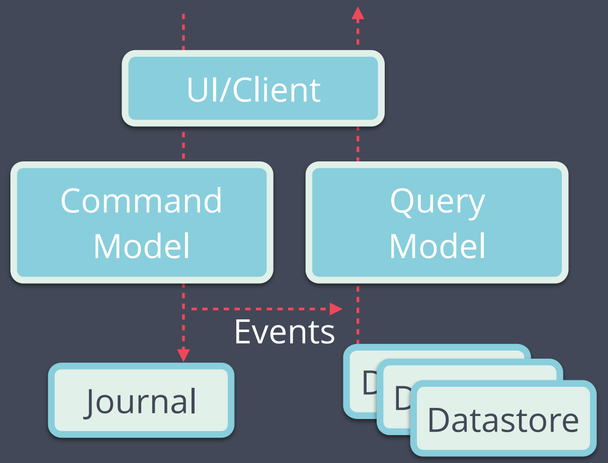
In conclusion

Event-sourcing is...

Powerful

but unfamiliar

Combines well with



DDD/CQRS

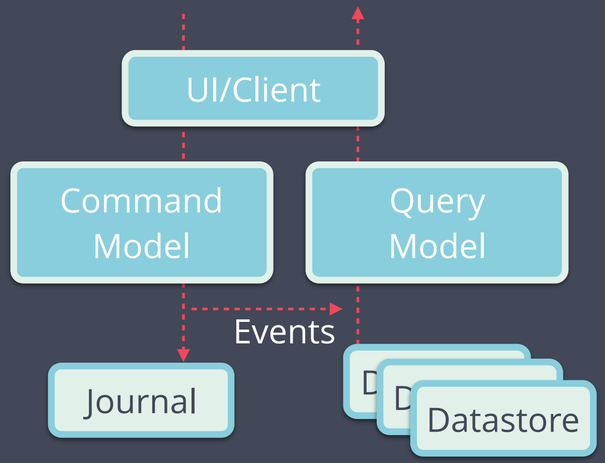
In conclusion

Event-sourcing is...

Powerful

but unfamiliar

Combines well with



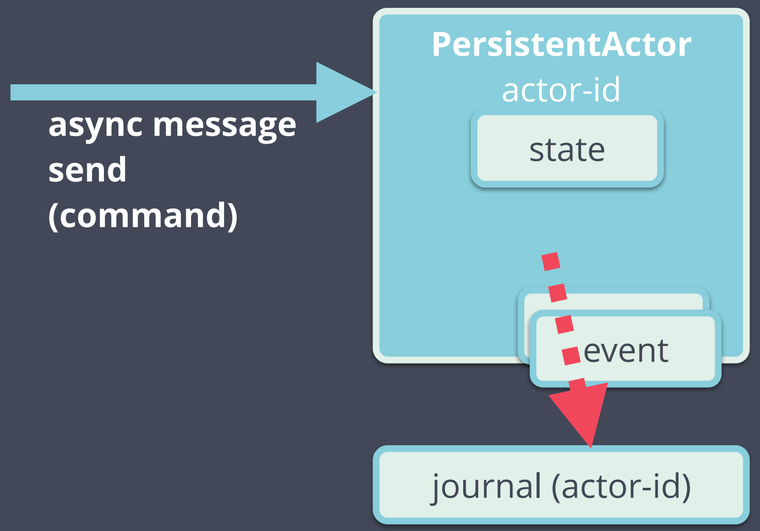
DDD/CQRS

Not a

In conclusion

Akka Actors & Akka Persistence

A good fit for event-sourcing



Experimental, view improvements needed

Thank you.

code @ bit.ly/akka-es

@Sander_Mak
Luminis Technologies