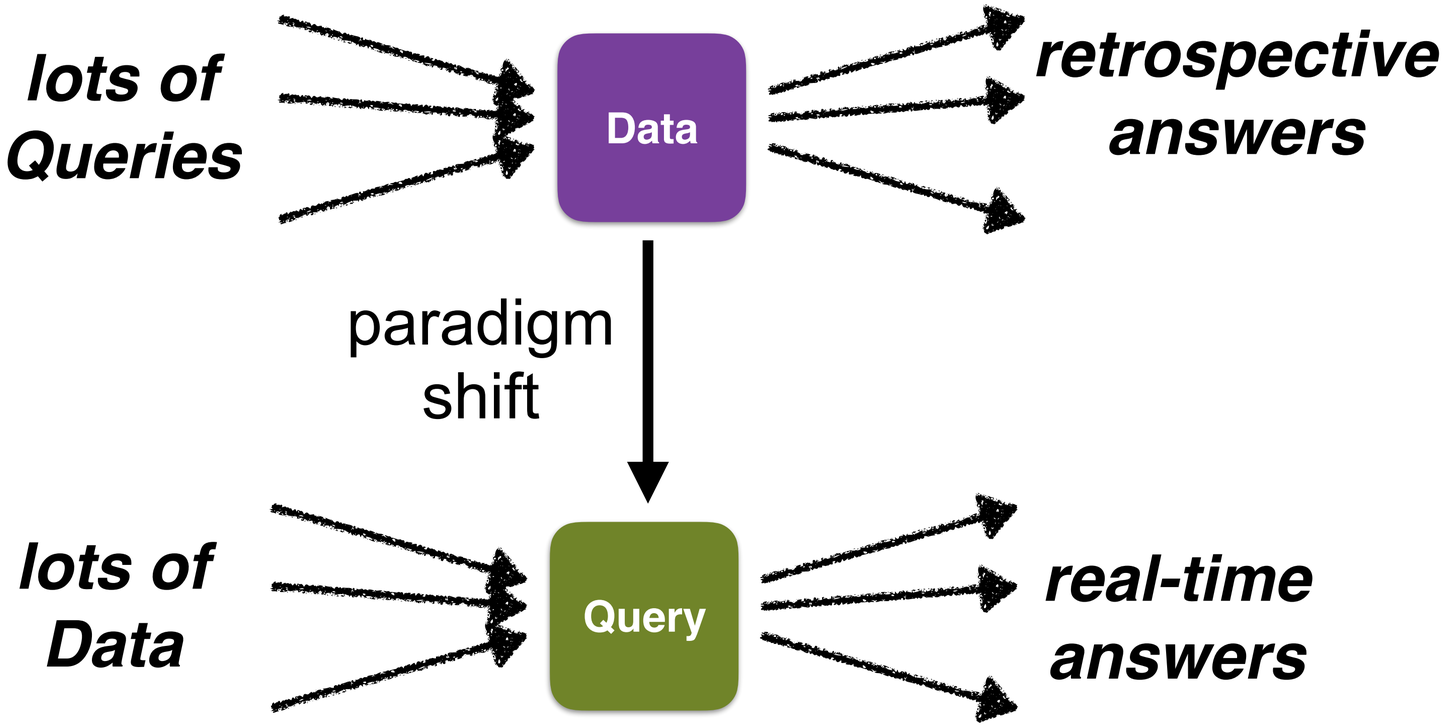
RI.
SE

Reliable Stream Processing at Scale with Apache Flink

Paris Carbone <paris.carbone@ri.se>
Senior Researcher @ **RISE**
Committer @ **Apache Flink**

Paradigm Shift in Data Processing



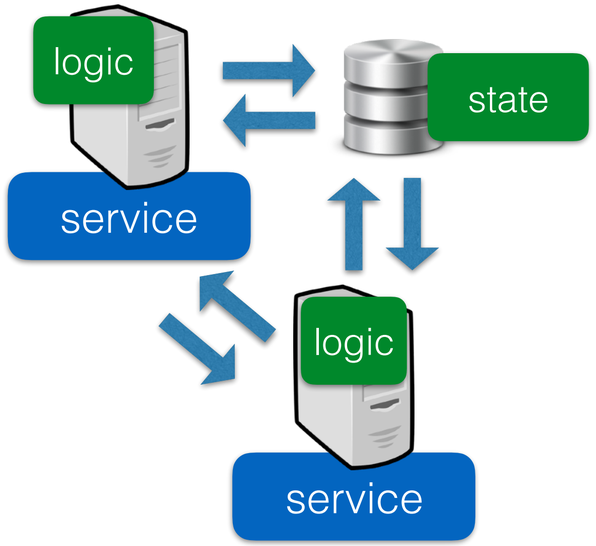
The Real-Time Analytics Stack

High Level Models	Stream SQL, CEP…
Compute	Flink, Beam, Kafka-Streams, Apex, Storm, Spark
Storage	Kafka, Pub/Sub, Kinesis, Pravega…

- **Data Stream Processing** as a 24/7 execution paradigm

Actors vs Streams

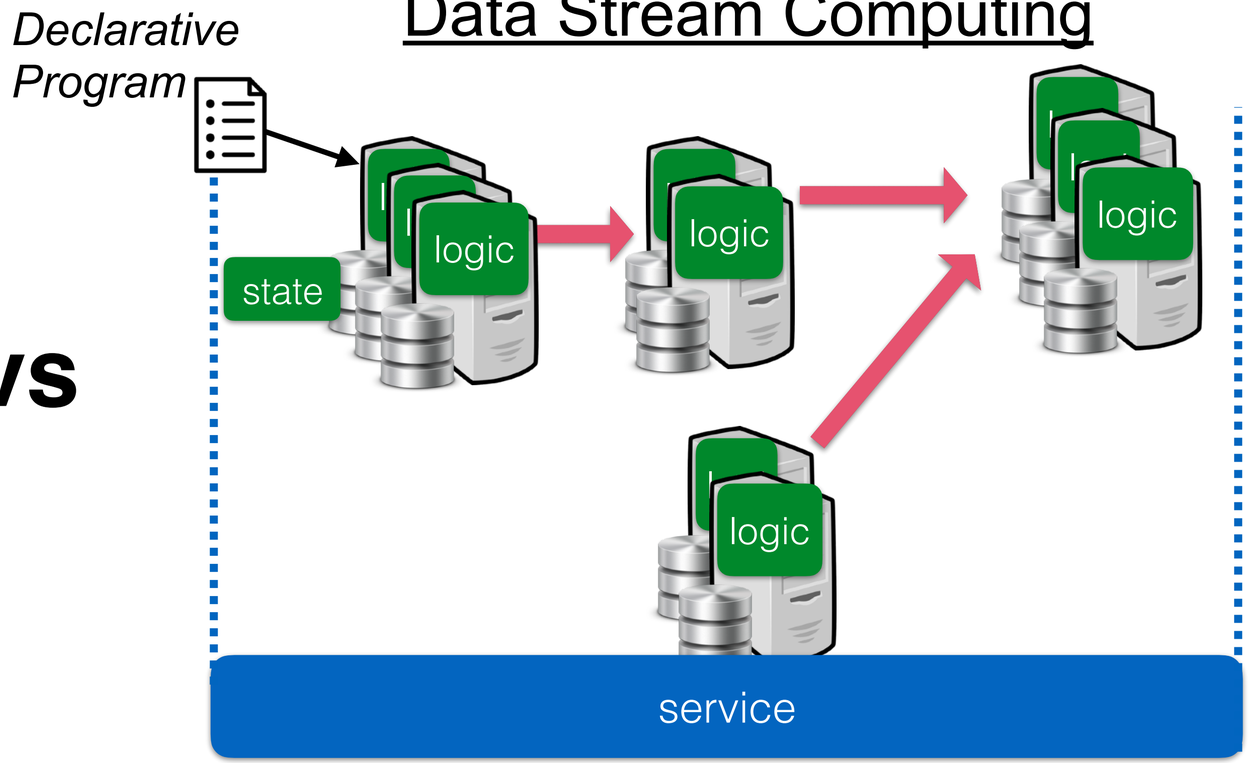
Actor Programming



- Low-Level Event-Based Programming
- Manual/External State
- Partial/Inflexible scaling
- Not Robust: Manual Fault Tolerance

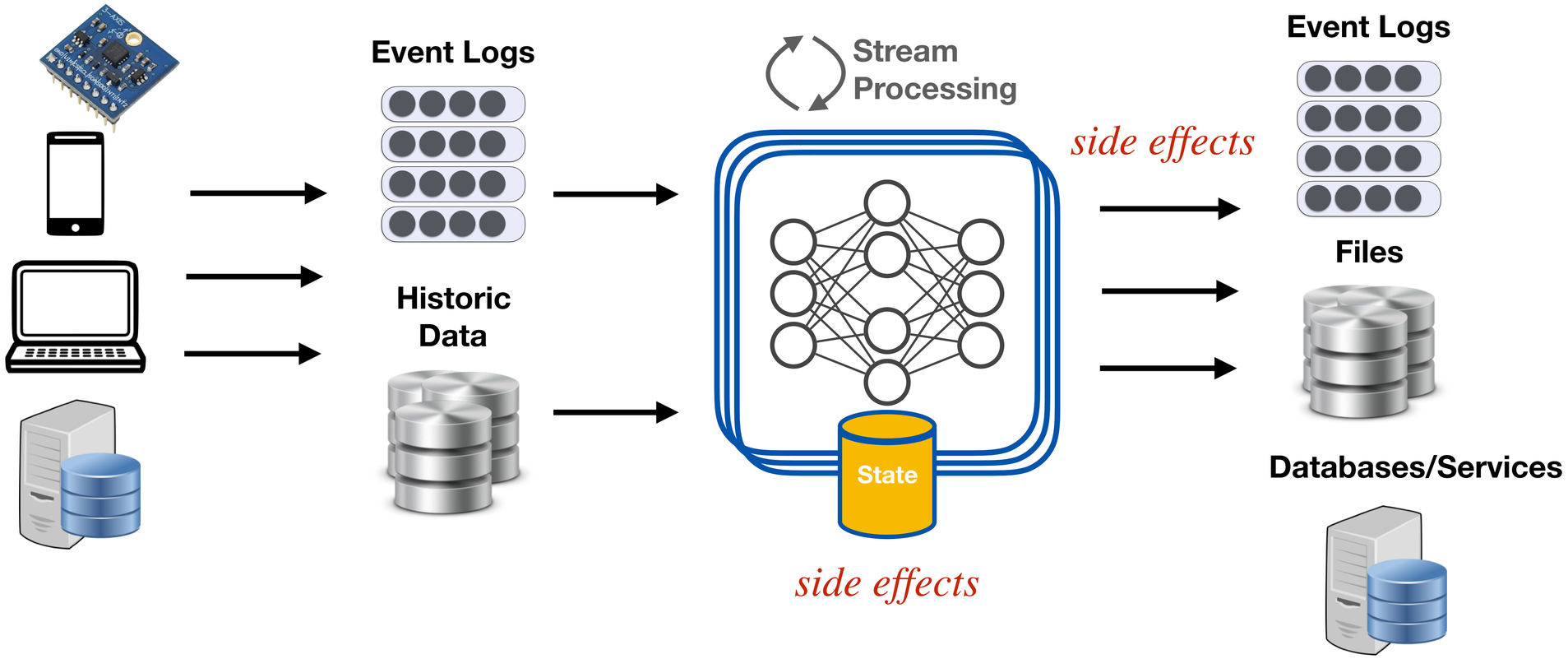
vs

Data Stream Computing



- Declarative Programming
- State Managed by the system
- Fully Scalable Deployments
- **End-to-End Reliable Processing**

What is End-to-End



About Flink

Stream Model, State Management, Window Aggregation, Reliability

- Top-level Apache Project
- #1 stream processor (2019)
- Production-Proof

- > 400 contributors
- 100s of deployments

production deployments

state management, windows, sql

influenced

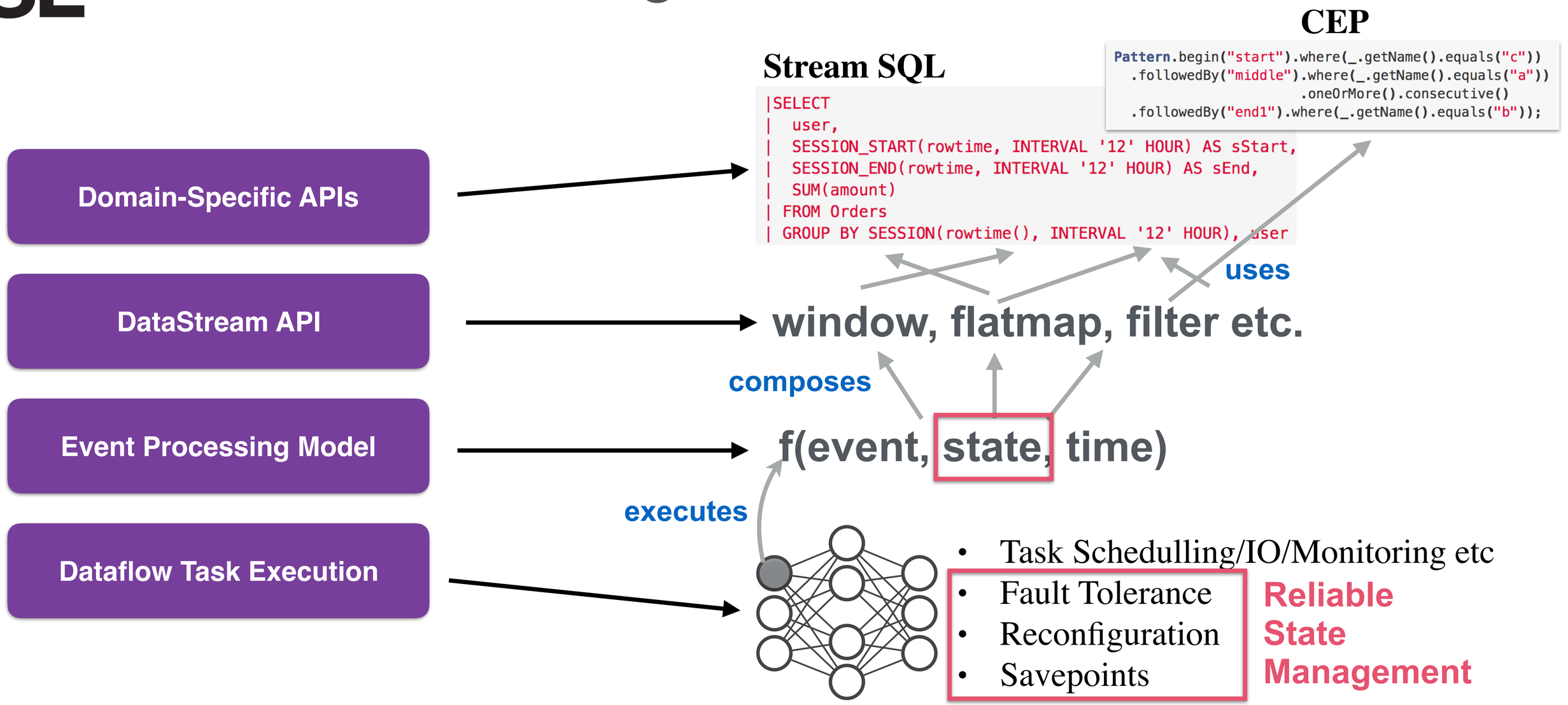
Google beam Spark APEX STORM

MediaMath ING airbnb bouygues King Alibaba Group Microsoft HUAWEI NETFLIX UBER SK telecom yelp lyft Telefonica

Technologies Behind Flink

- Flink runs on the **JVM.**
- **Master/Slave** architecture ~ Hadoop **(JobManager, TaskManagers)**
- **Java** and **Scala** 100% supported.
- **Depends** on: *Zookeeper, Akka, RocksDB (persistent (disk) state).*
- **Connectors**: *Kafka, Cassandra, Kinesis, Elasticsearch, HDFS, RabbitMQ, NiFi, Google Cloud PubSub, Twitter API* etc.

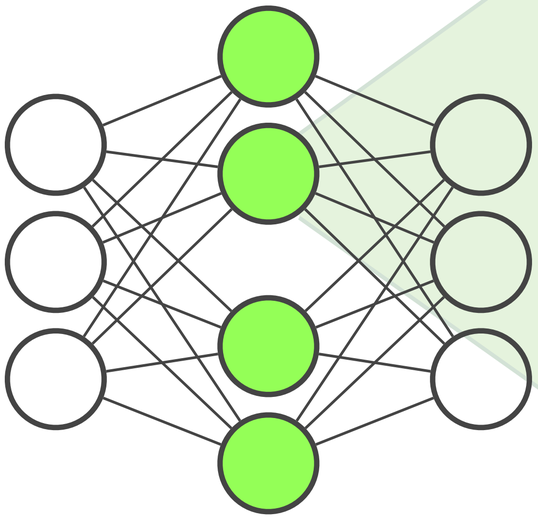
Building Blocks of Flink



Dynamic State

- Encapsulates **any** Event-Processing Logic as: **f(event, state, time)**

```
// the source data stream
val stream: DataStream[…] = ...
val result: DataStream[…] = stream
  .keyBy(0)
  .process(new MyCustomLogic())
  .addSink(…)
```



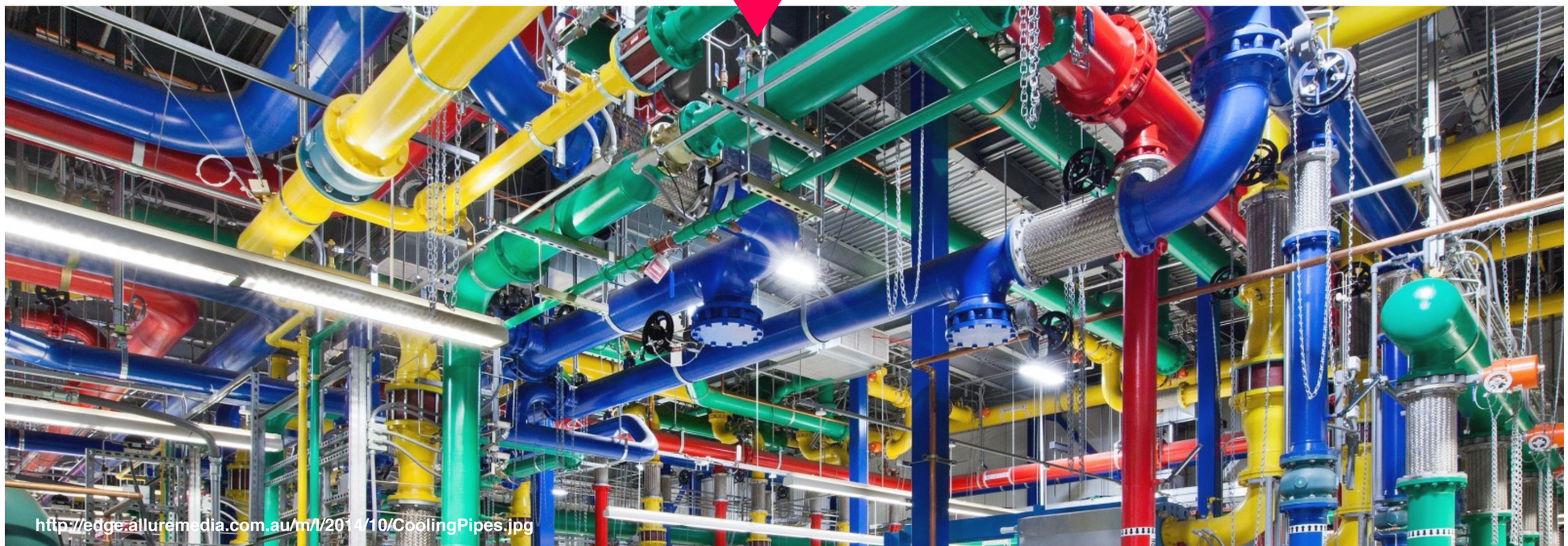
```
class MyCustomLogic extends KeyedProcessFunction[…] {

  /** The state that is maintained by this process function */
  lazy val state: ValueState[…] = getRuntimeContext
    .getState(new ValueStateDescriptor[…]("myState", classOf[…]))


  override def processElement(element: …)
    …
    state.update(…)
    …
    ctx.timerService.registerEventTimeTimer(…)
    …
  }

  override def onTimer( timestamp: Long,  StreamContext, TimerContext,  out: …): Unit = {

    state.value match {
      case foo => out.collect((key, count))
      case _ =>
    }
  }
}
```

Declarative Data Streaming

```
val windowCounts = text.flatMap { w => w.split("\\s") }
  .map { w => WordWithCount(w, 1) }
  .keyBy("word")
  .timeWindow(Time.seconds(5))
  .sum("count")
```

Window Word Count (Apache Flink)

24/7



Reliability Means…

handling failures!

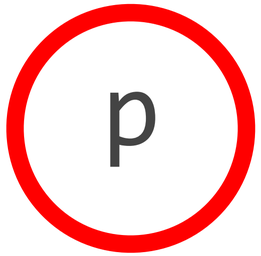
consistent application updates

trusting external output

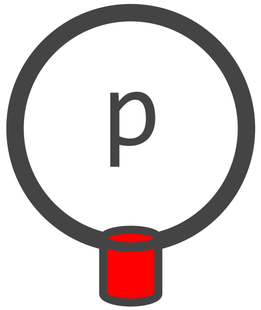
adding more/less workers

reconfiguring/upgrading the system correctly

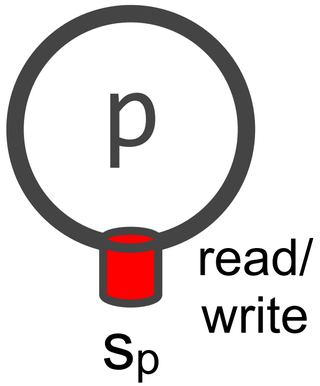
Event Processing Model



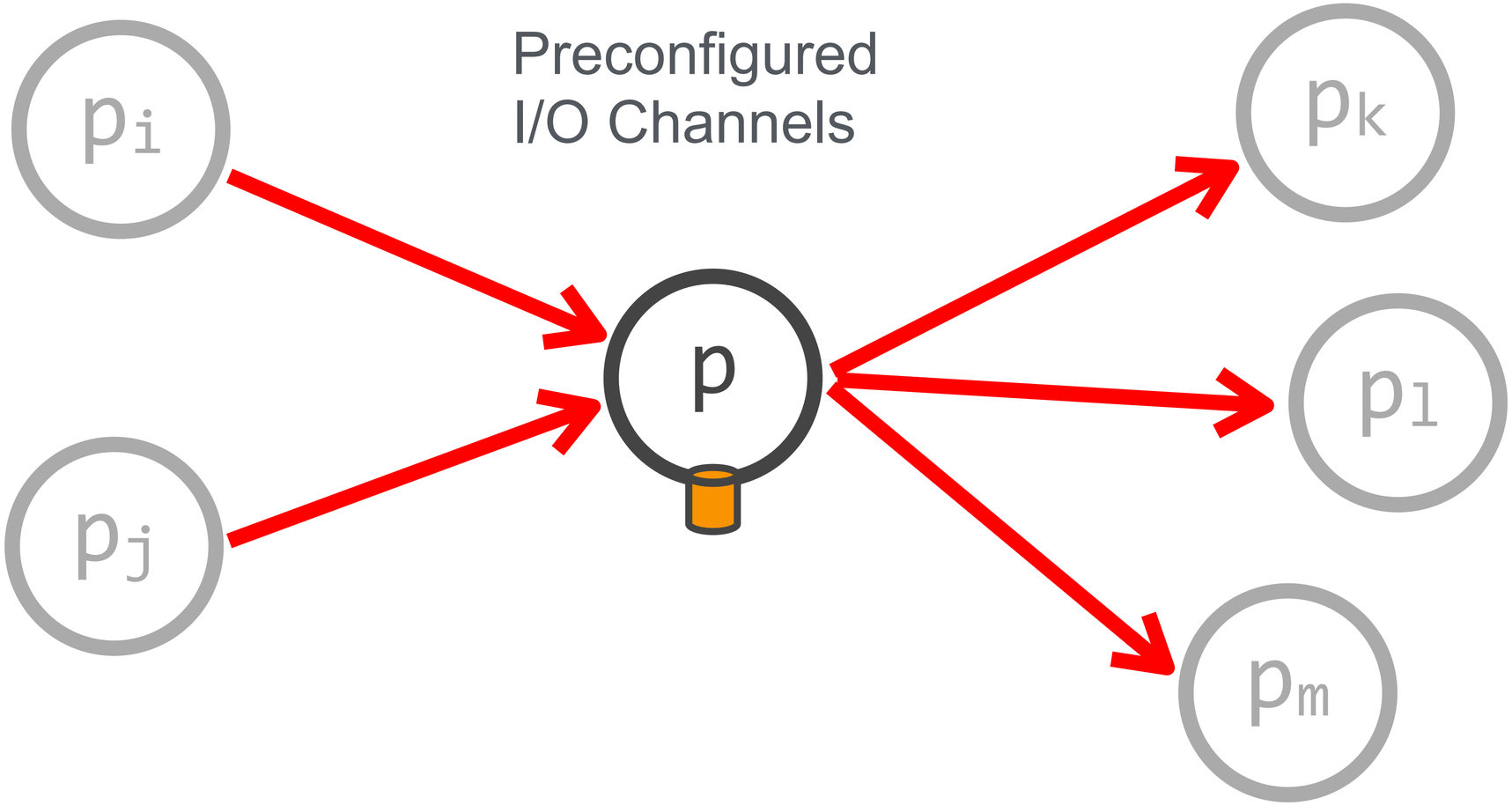
Event Processing Model



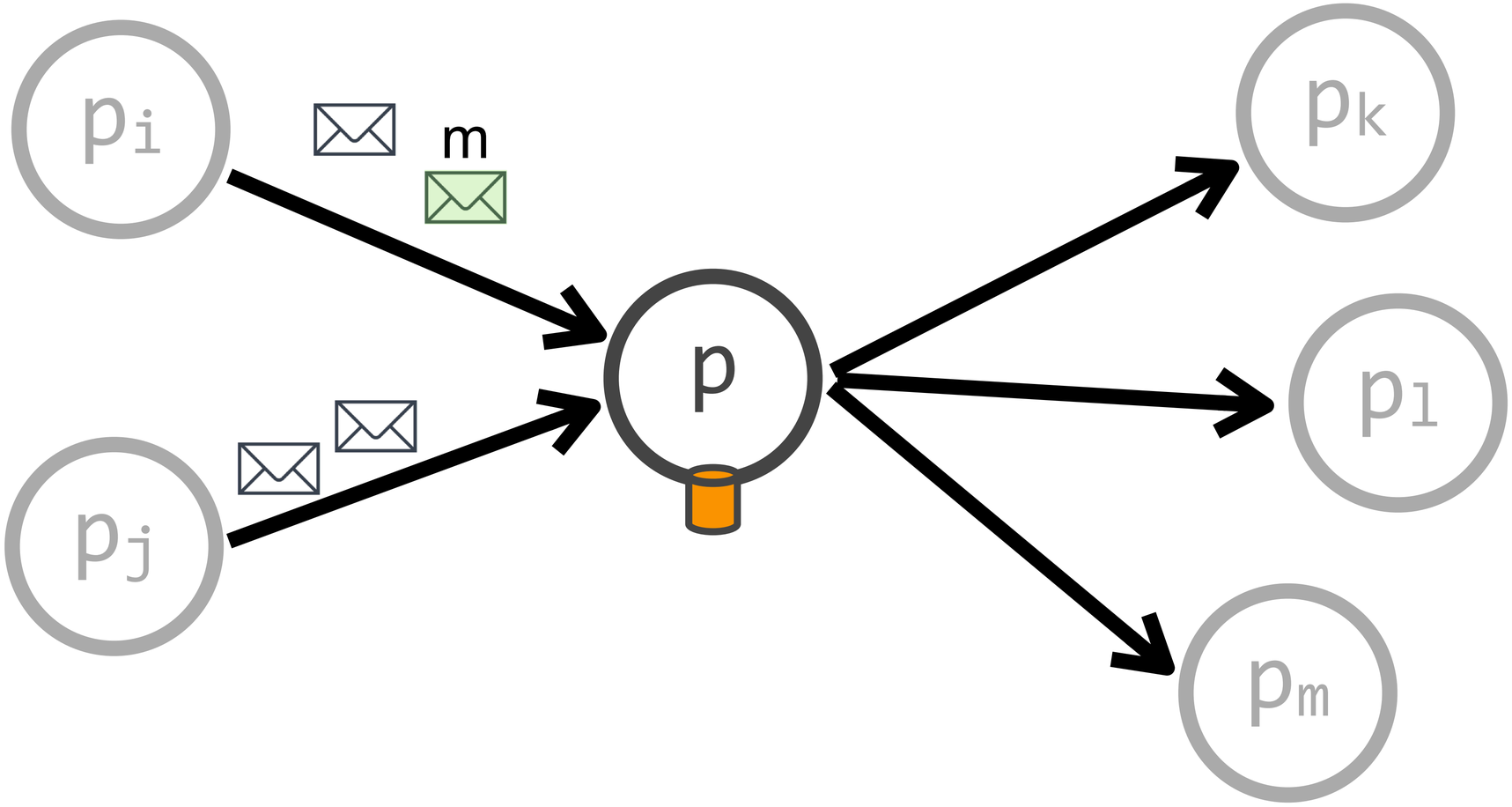
Event Processing Model



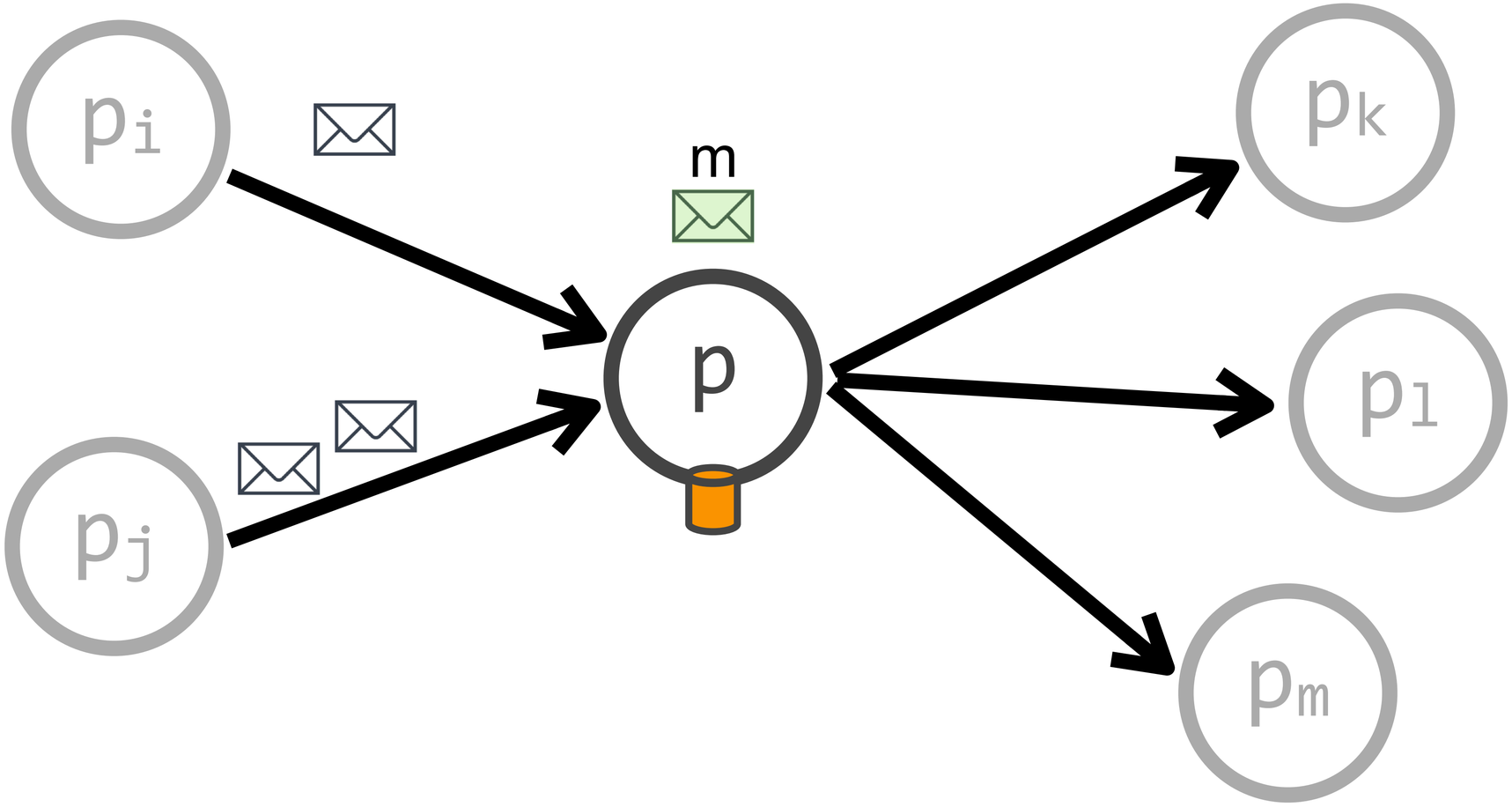
Event Processing Model



Event Processing Model

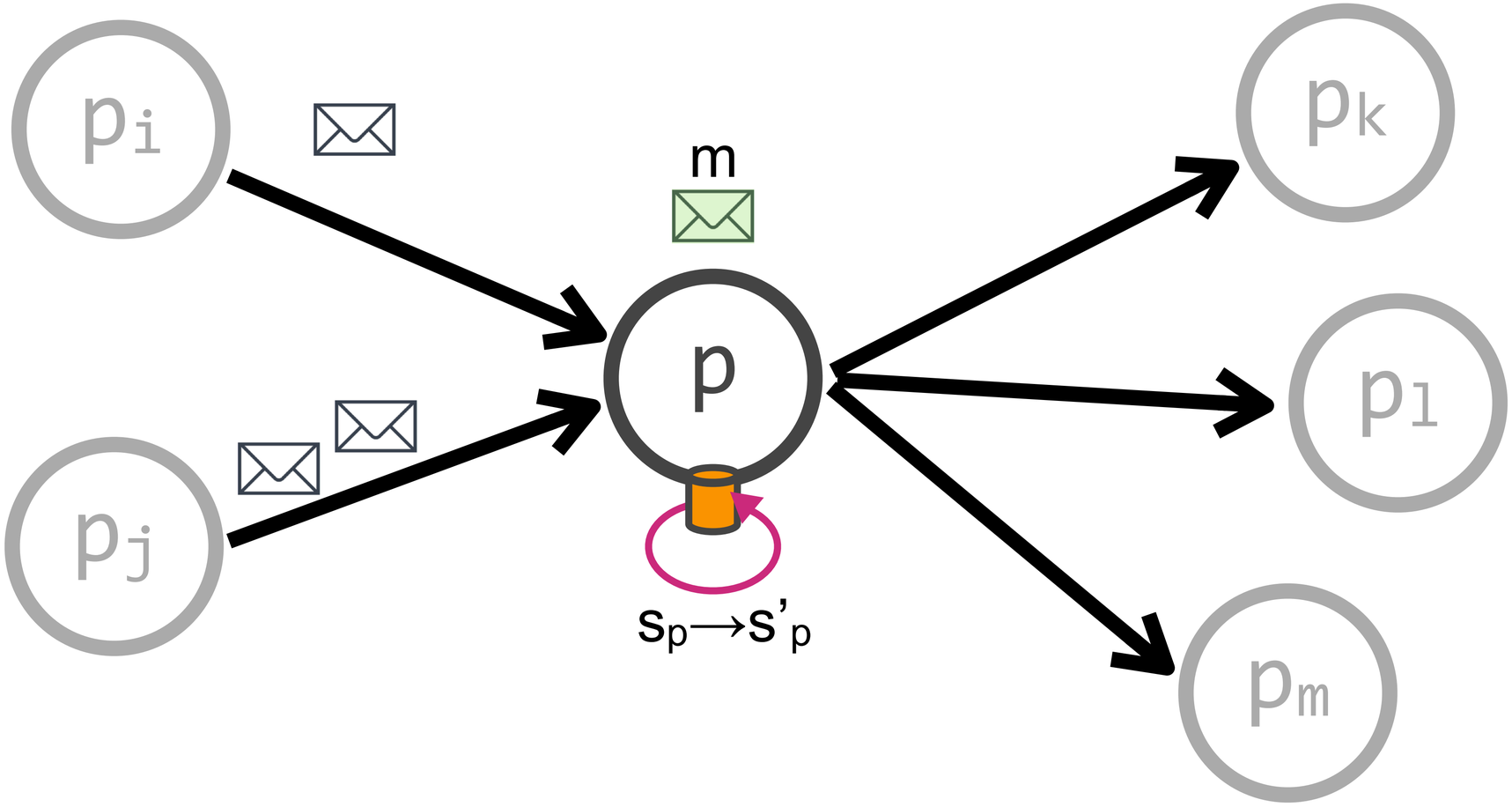


Event Processing Model



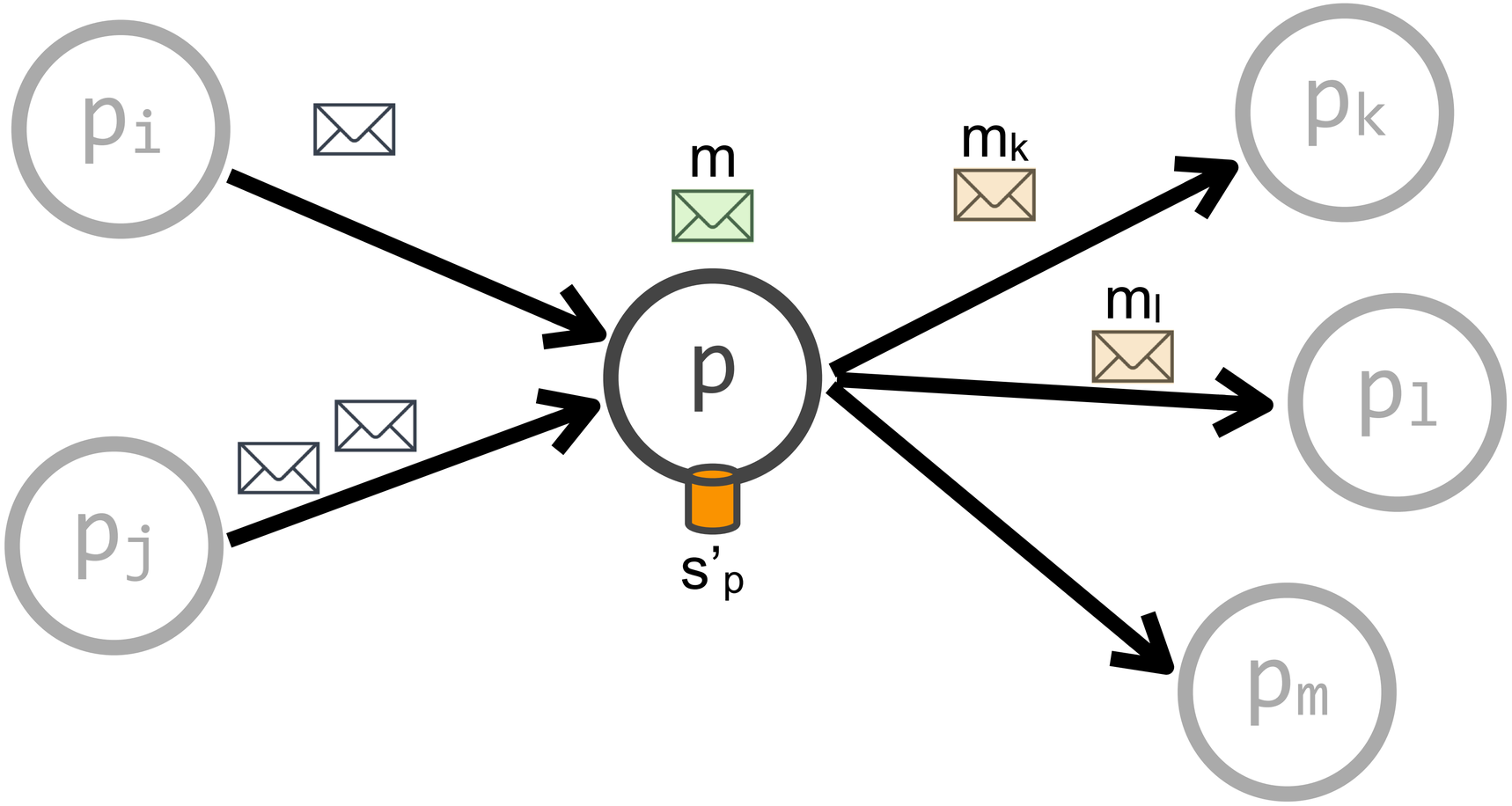
Action:{ ⟨recv,m⟩}

Event Processing Model



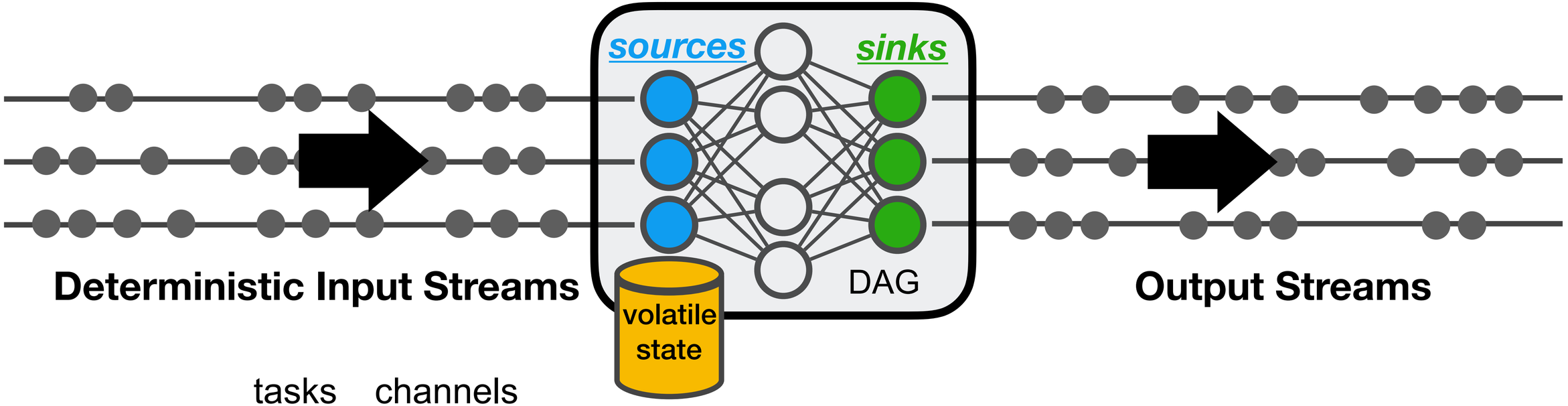
Action:{ ⟨recv,m⟩, ⟨$s_p \rightarrow s'_p$⟩ }

Event Processing Model



Action:{ ⟨recv,m⟩, ⟨$s_p \rightarrow s'_p$⟩, ⟨send,m_k⟩, ⟨send,m_l⟩ }

Stream Process Graphs

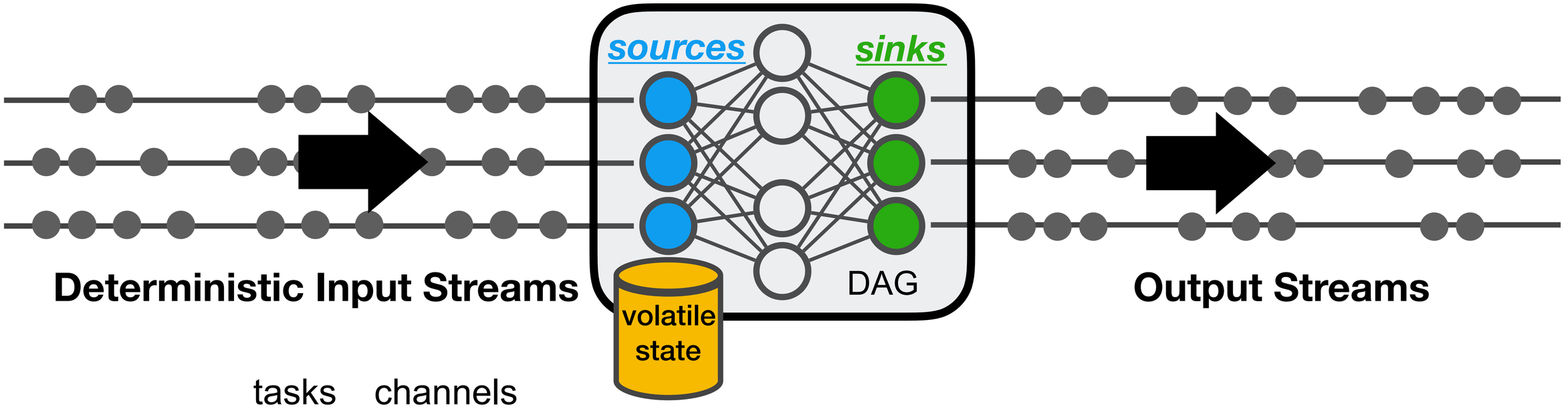


tasks channels

System : $\{\Pi, \mathbb{E}\}$

System Execution : $\ldots \rightarrow \{\Pi_*, M\} \rightarrow \{\Pi'_*, M'\} \rightarrow \ldots$

Stream Process Graphs



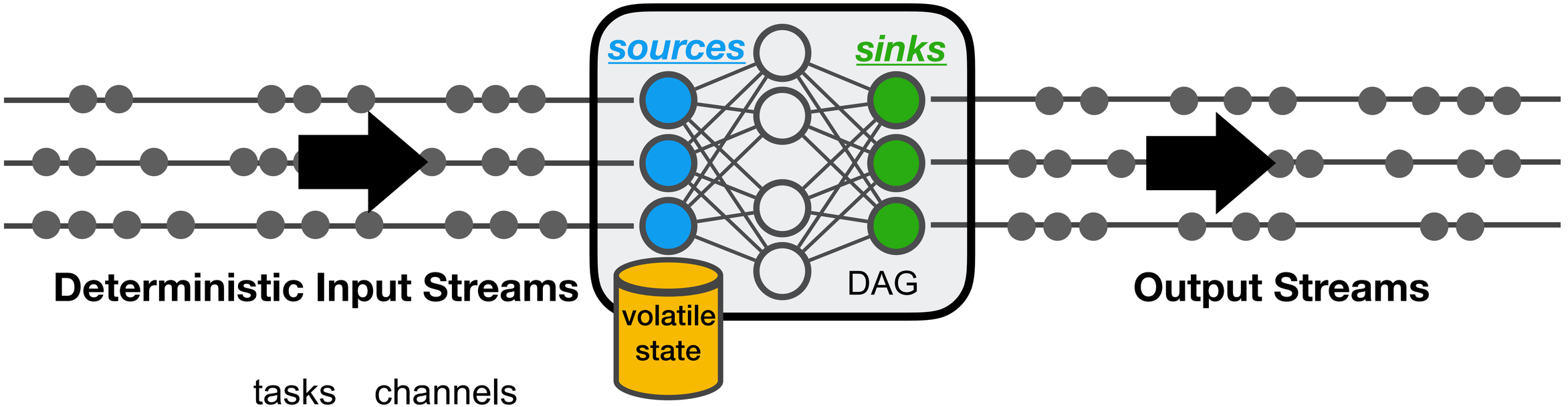
tasks channels

System : $\{\Pi, \mathbb{E}\}$

Task Actions

System Execution : $\ldots \boxed{\rightarrow} \{\Pi_*, M\} \boxed{\rightarrow} \{\Pi'_*, M'\} \boxed{\rightarrow} \ldots$

Stream Process Graphs

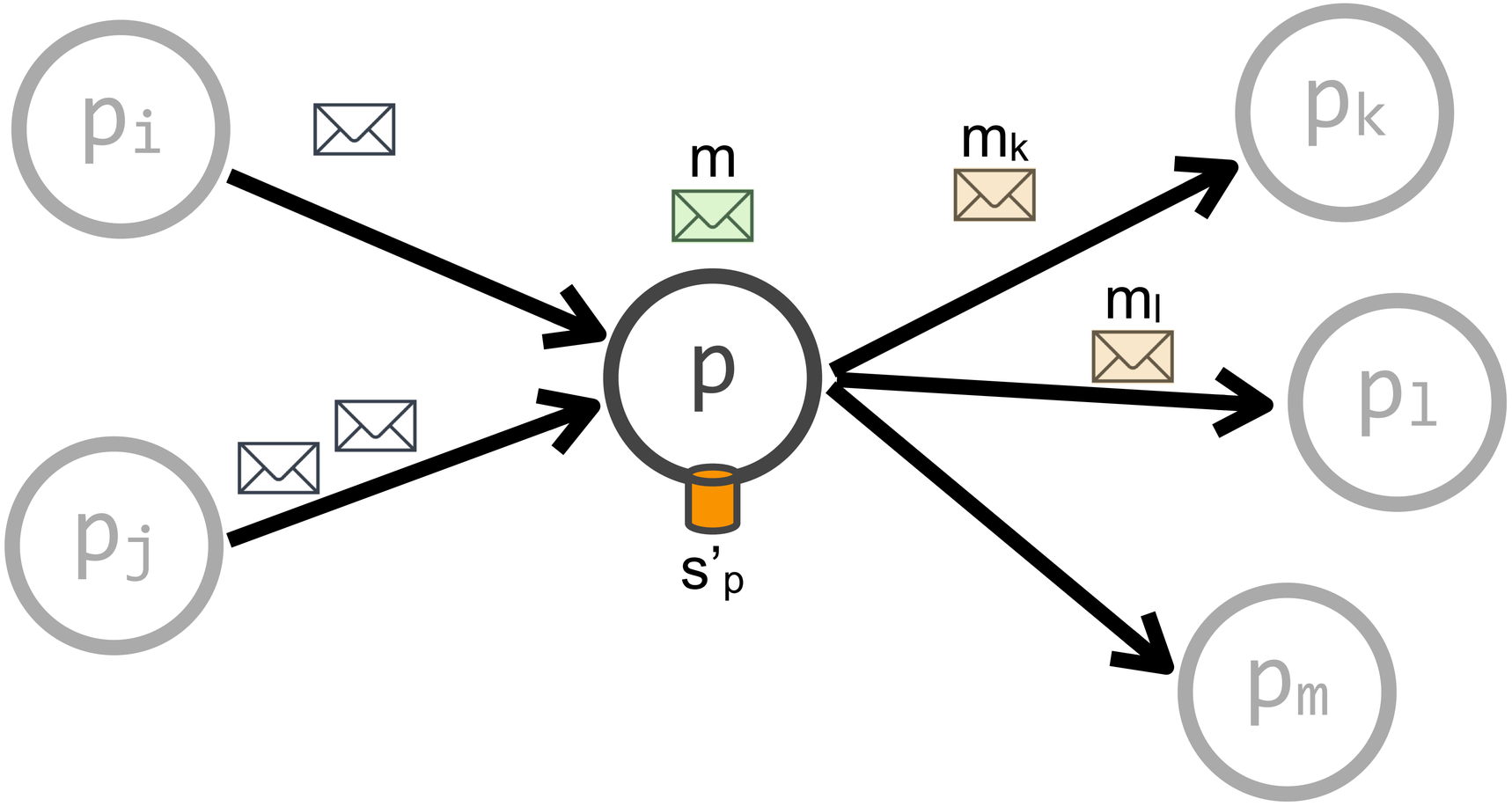


tasks channels

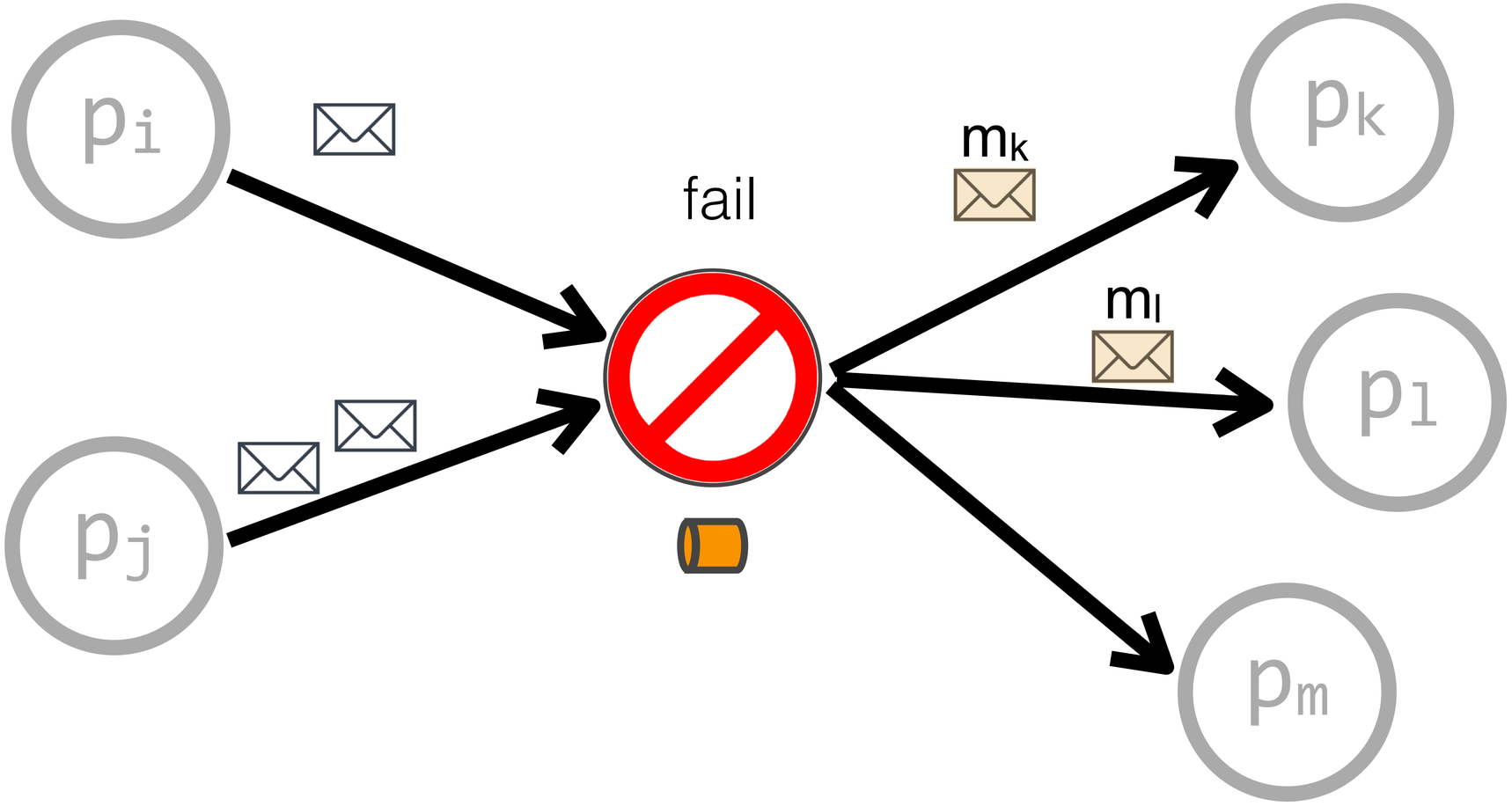
System : $\{\Pi, \mathbb{E}\}$ **System Configurations** (states, messages in-transit)

System Execution : $\ldots \rightarrow \{\Pi_*, M\} \rightarrow \{\Pi'_*, M'\} \rightarrow \ldots$

Fault Tolerance

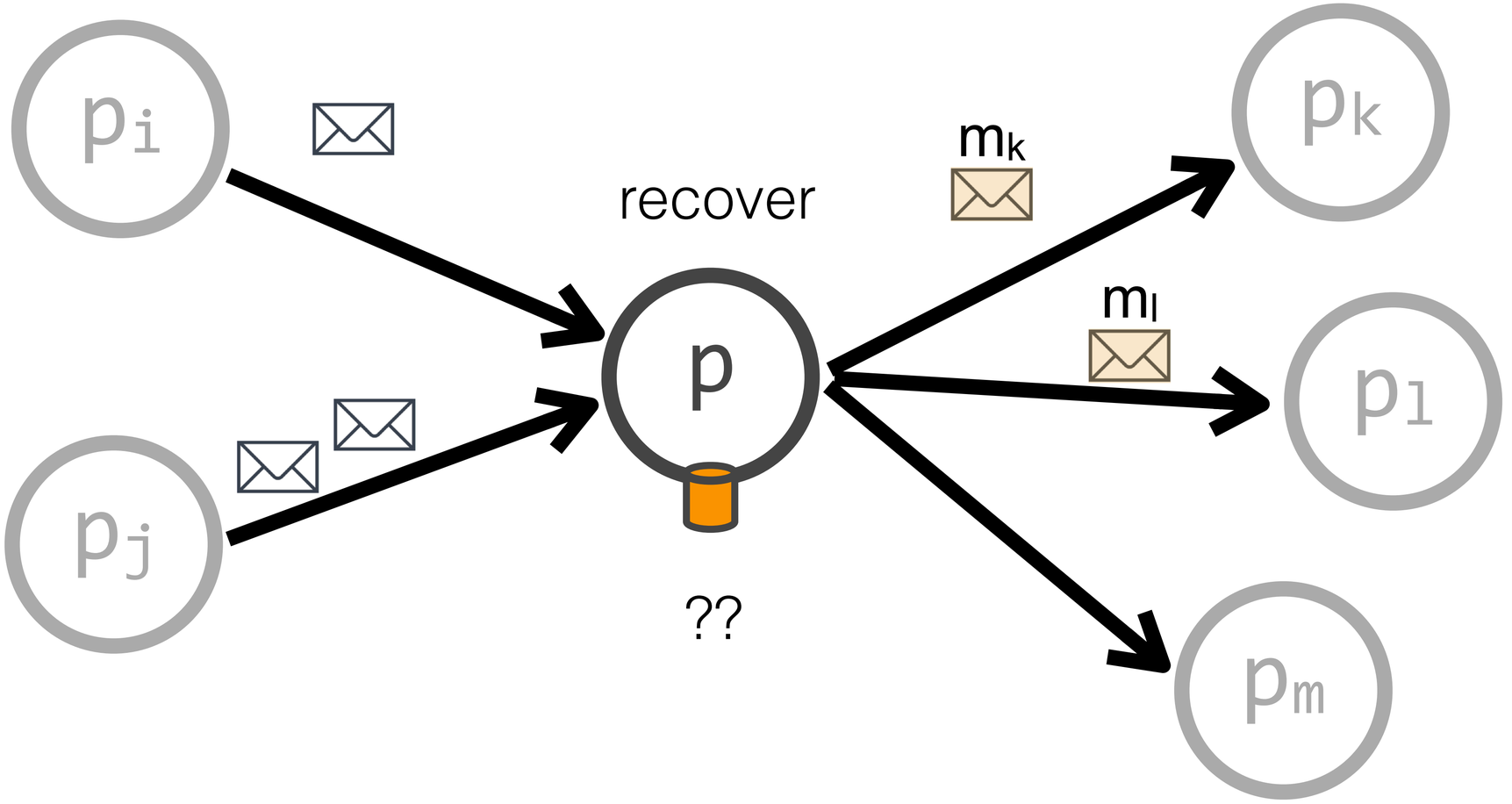


Fault Tolerance



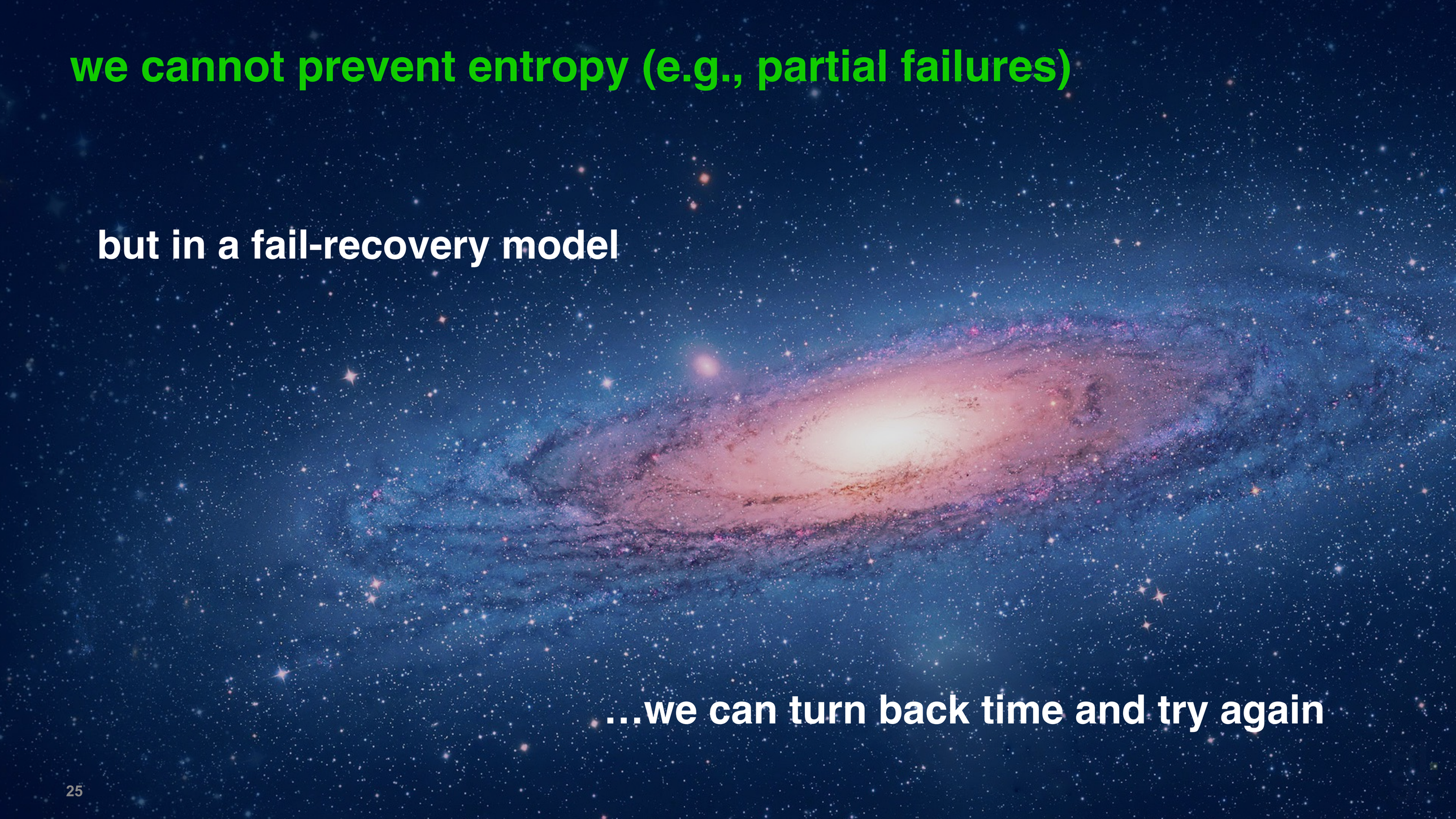
Task Recovery

*Problem: It is **Hard** to reason about individual action completeness*



- Has m been fully processed?
- Have m_k and m_l been delivered? Have they caused other actions?

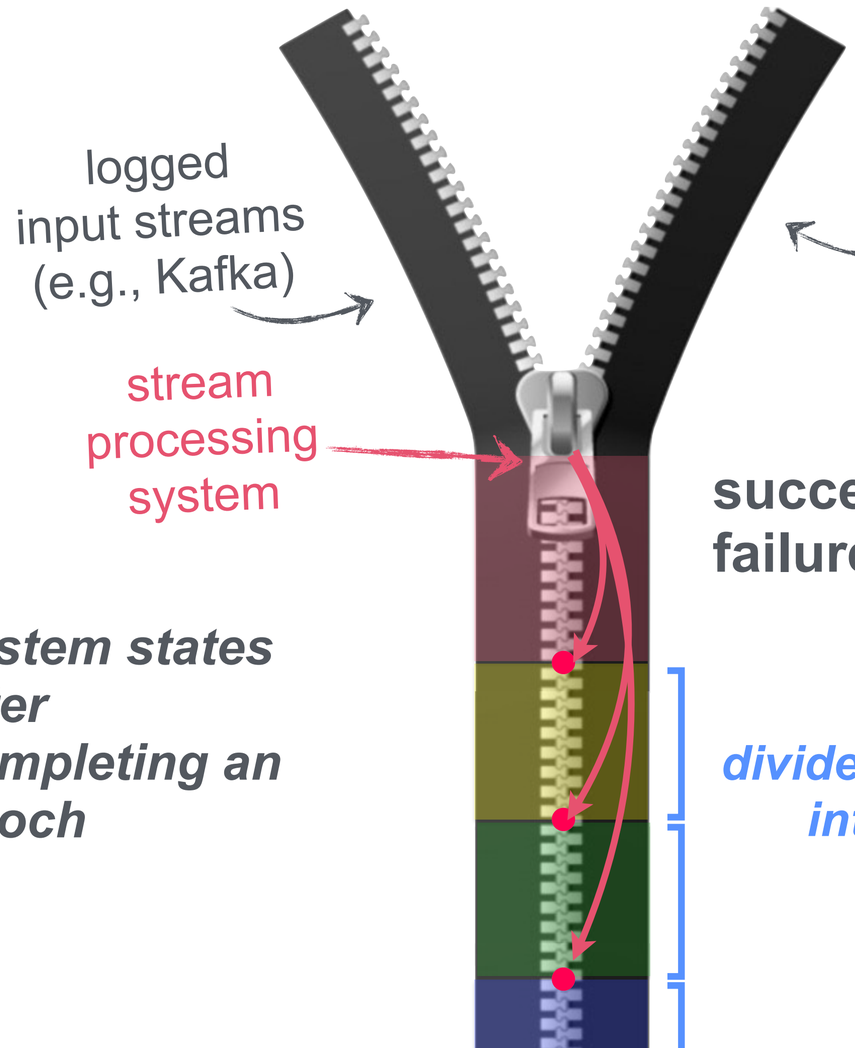
we cannot prevent entropy (e.g., partial failures)



but in a fail-recovery model

…we can turn back time and try again

Transactional Stream Processing
The Intuition

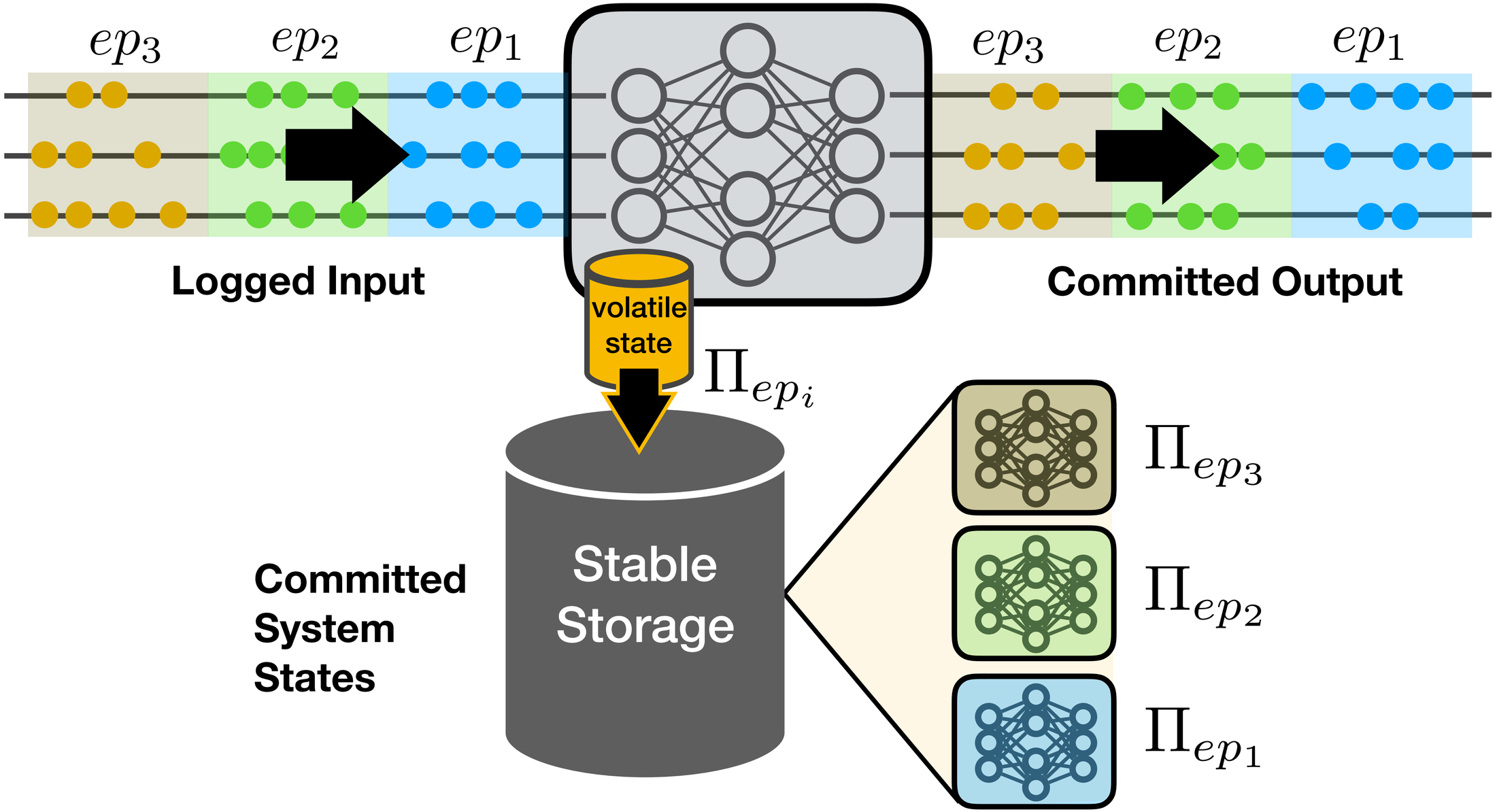


success: commit system configuration
failure: abort and start from previous epoch

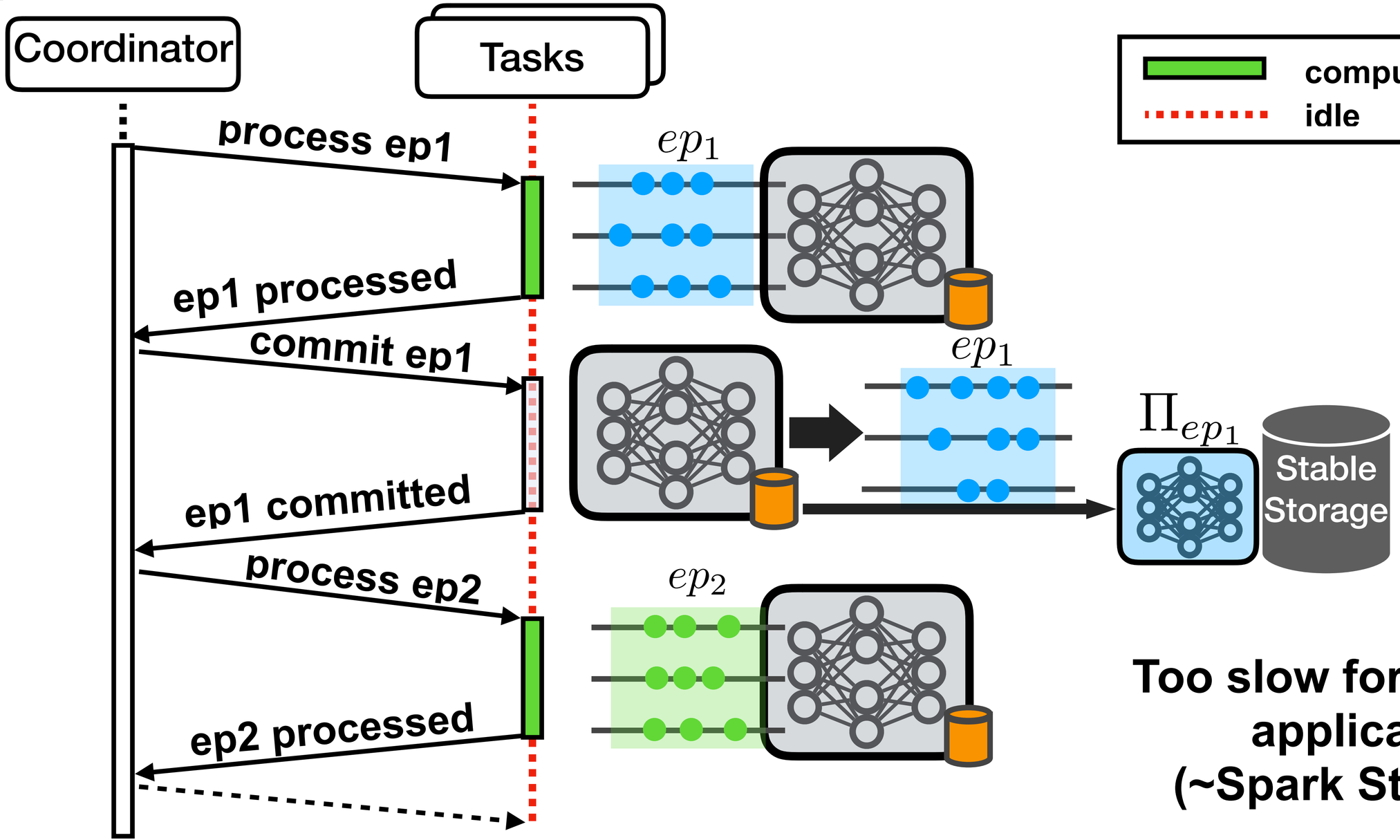
- ***system states after completing an epoch***

divide computation into epochs

Approach Overview

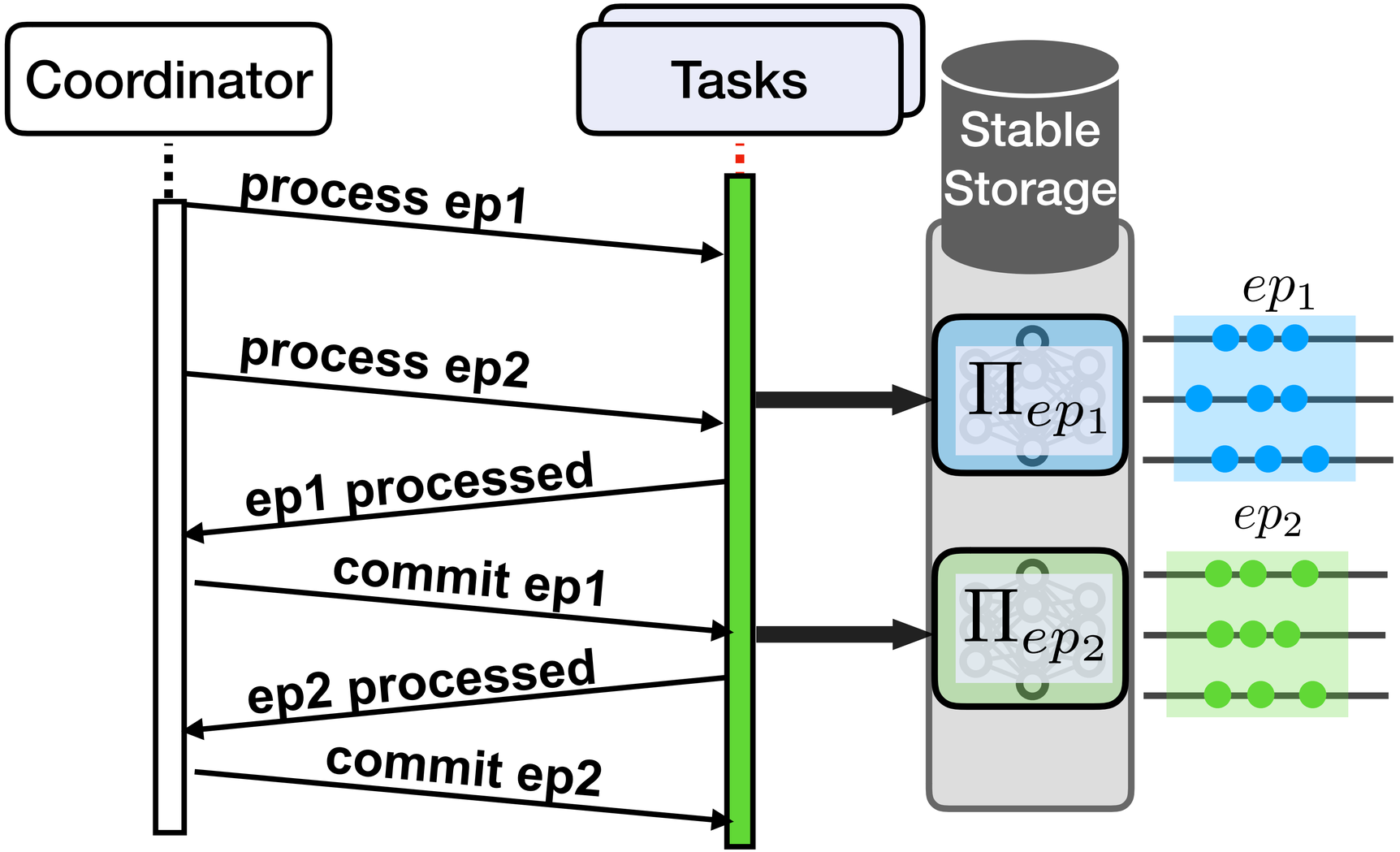


Synchronous Epoch Commits



Too slow for streaming applications (~Spark Streaming)

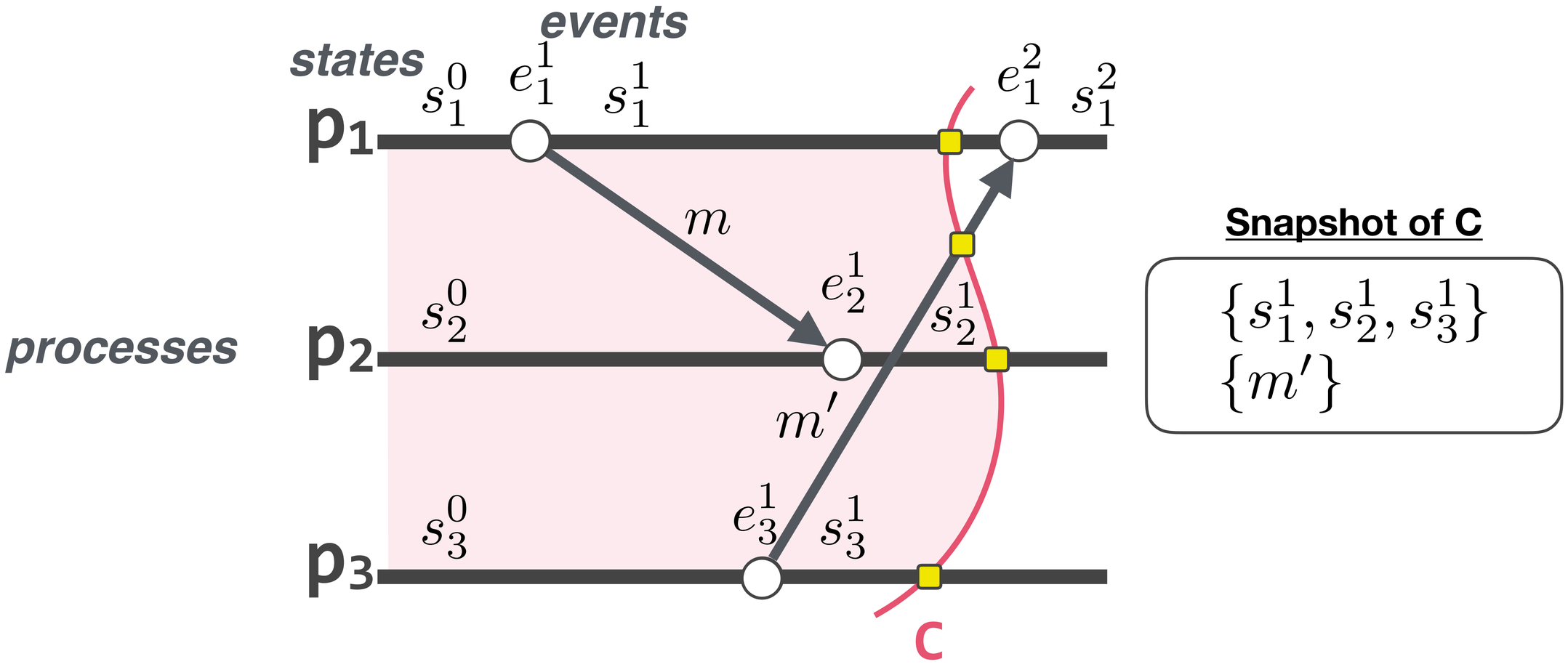
Asynchronous Epoch Commits



How? Using Distributed Snapshotting

Distributed Snapshotting 101

Valid Snapshot: A set of system states that form a **causally-correct distributed cut** in an execution

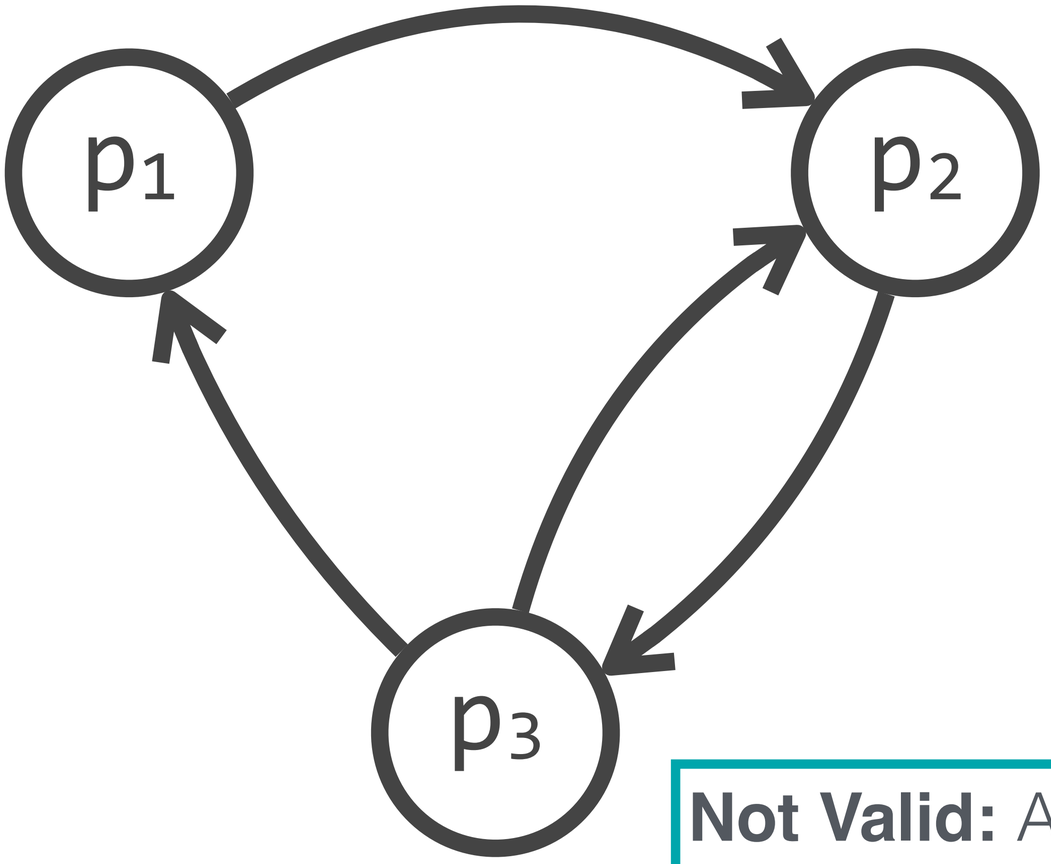


Snapshot of C

$\{s_1^1, s_2^1, s_3^1\}$
$\{m'\}$

Validity Explained

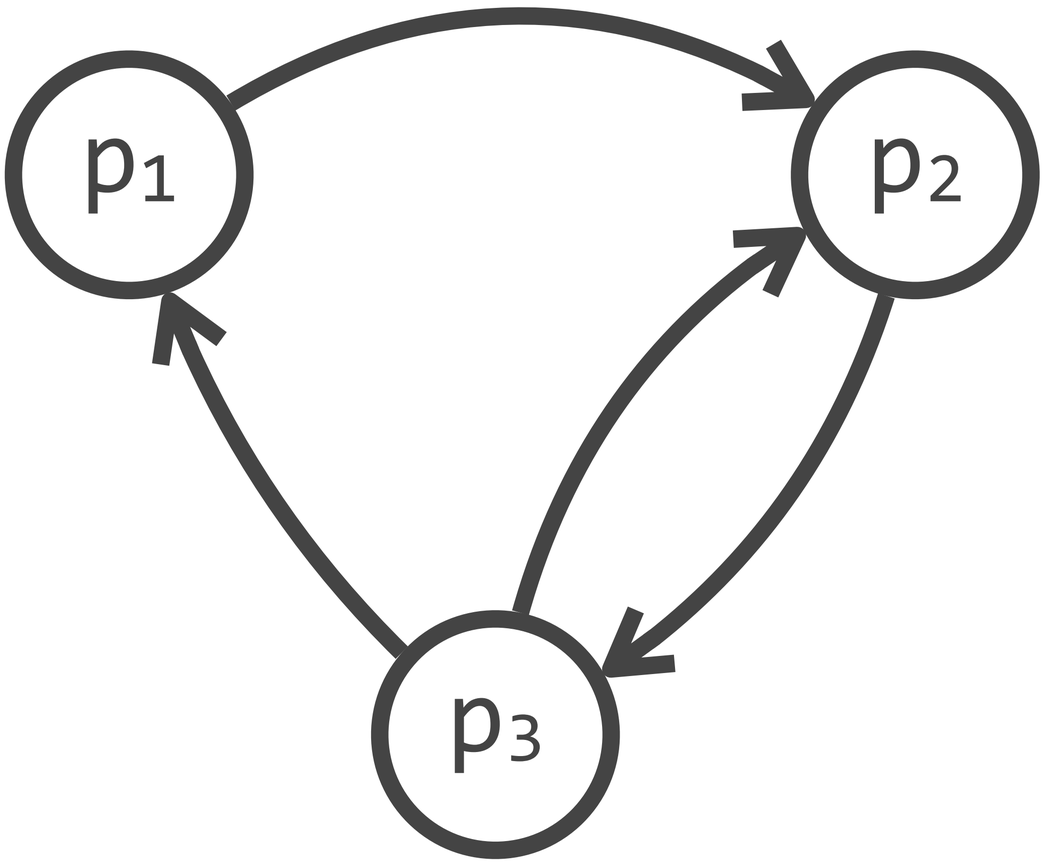
System



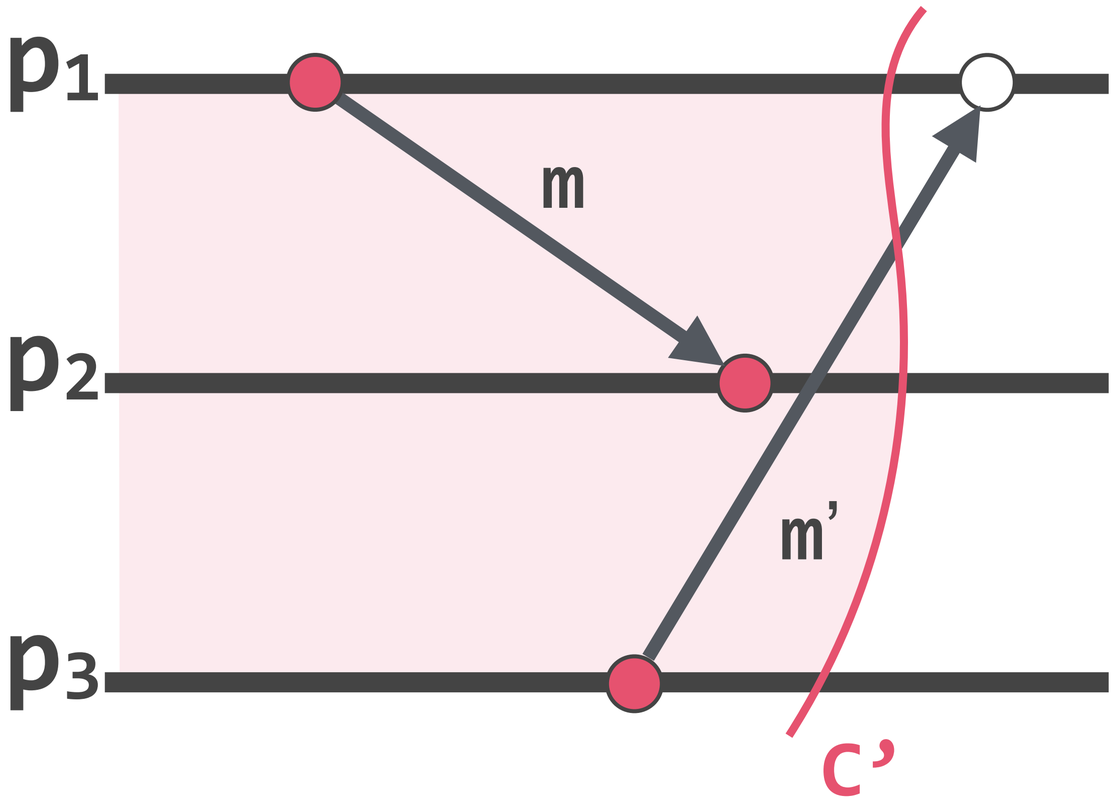
Not Valid: According to C, **m'** was received but never sent **(Violates Causality)**

Validity Explained

System



Possible Execution

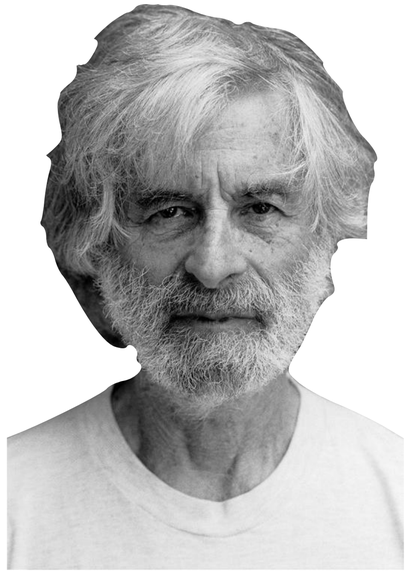


Valid: No causality violations in C'
(**C' is a consistent cut)**

The Chandy-Lamport Algorithm

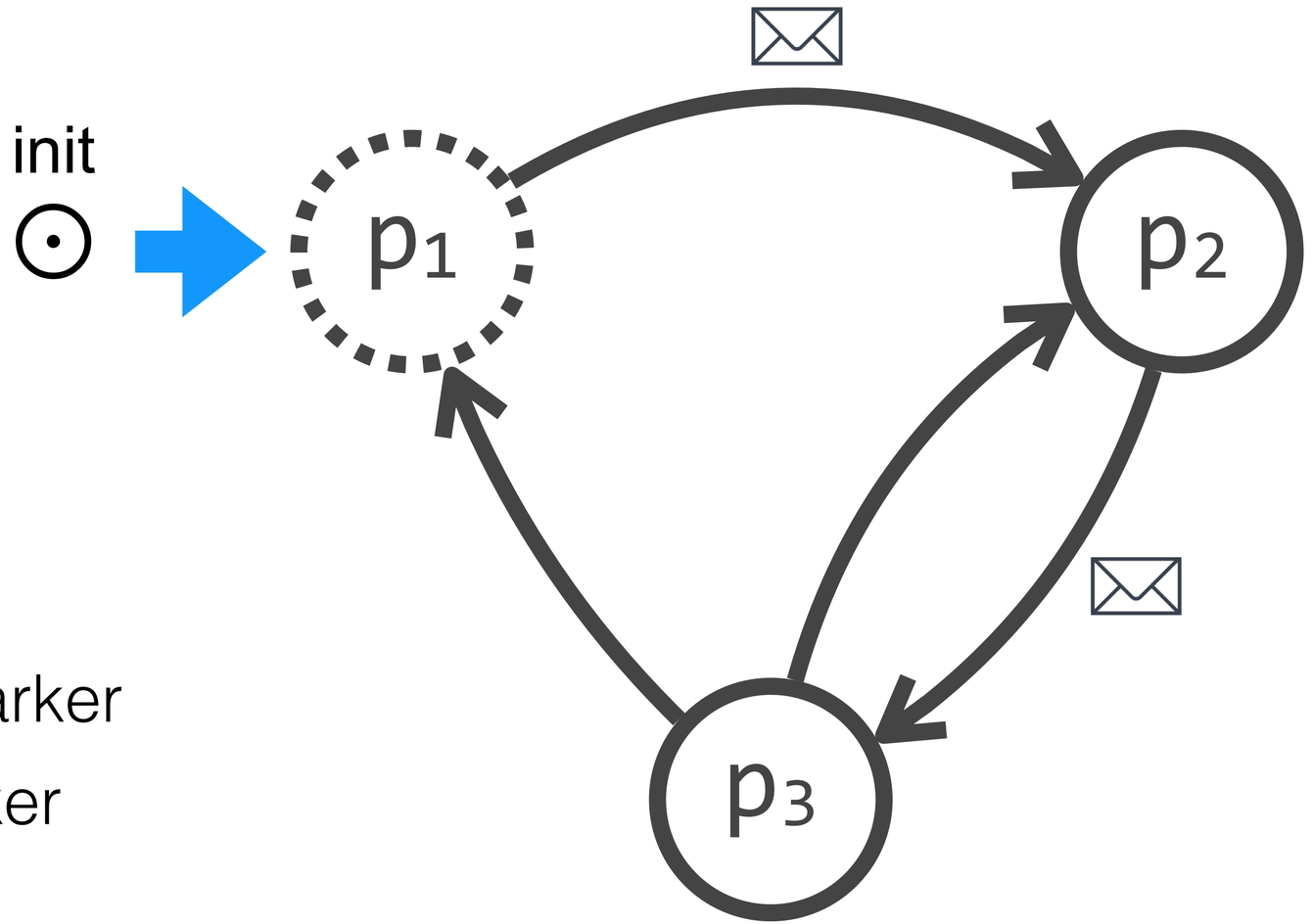
- Assumptions:
  - FIFO Reliable Channels
  - Strongly Connected Graph
  - Single Initiating Process

Obtains Valid Snapshot
i.e., no causality violations



Leslie Lamport

The Chandy-Lamport Algorithm



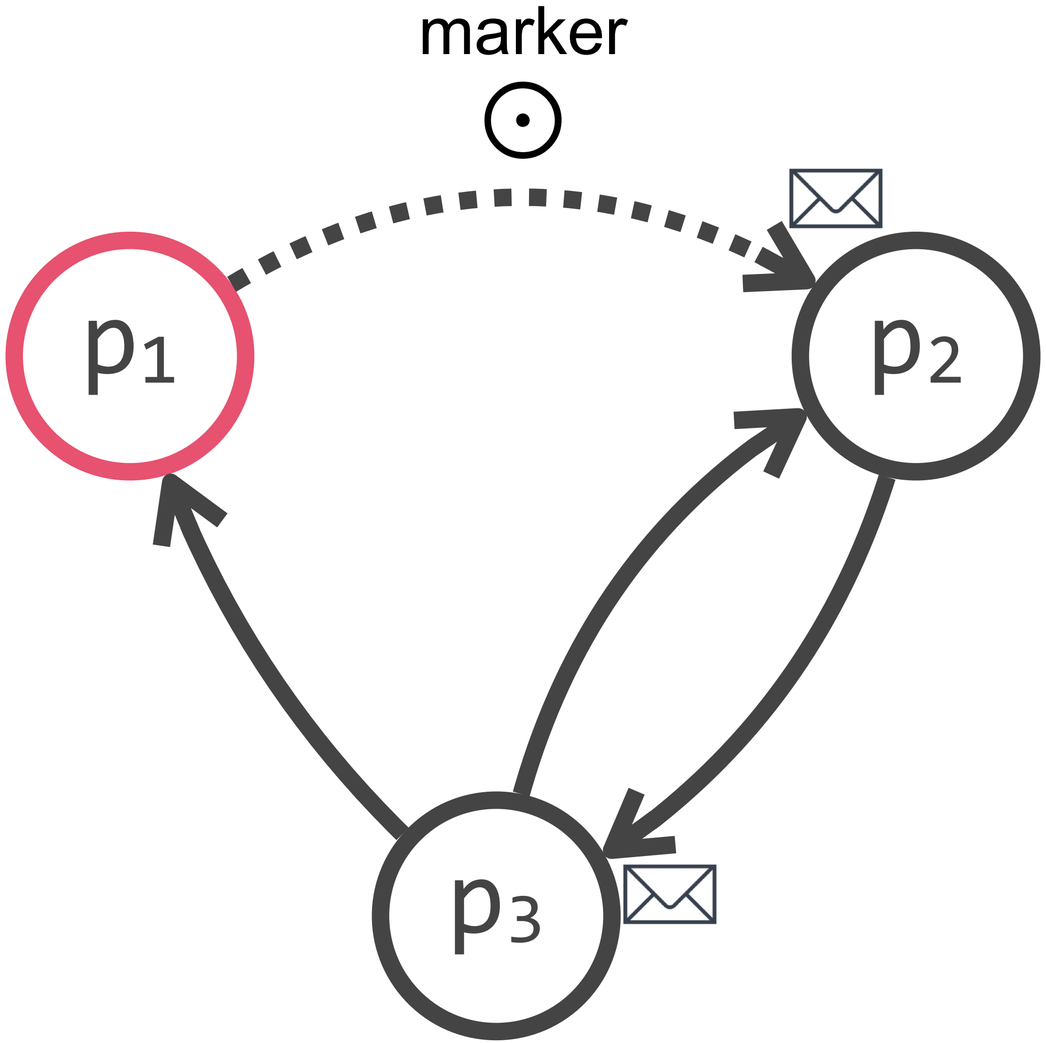
Snapshot

s1

before marker

after marker

The Chandy-Lamport Algorithm



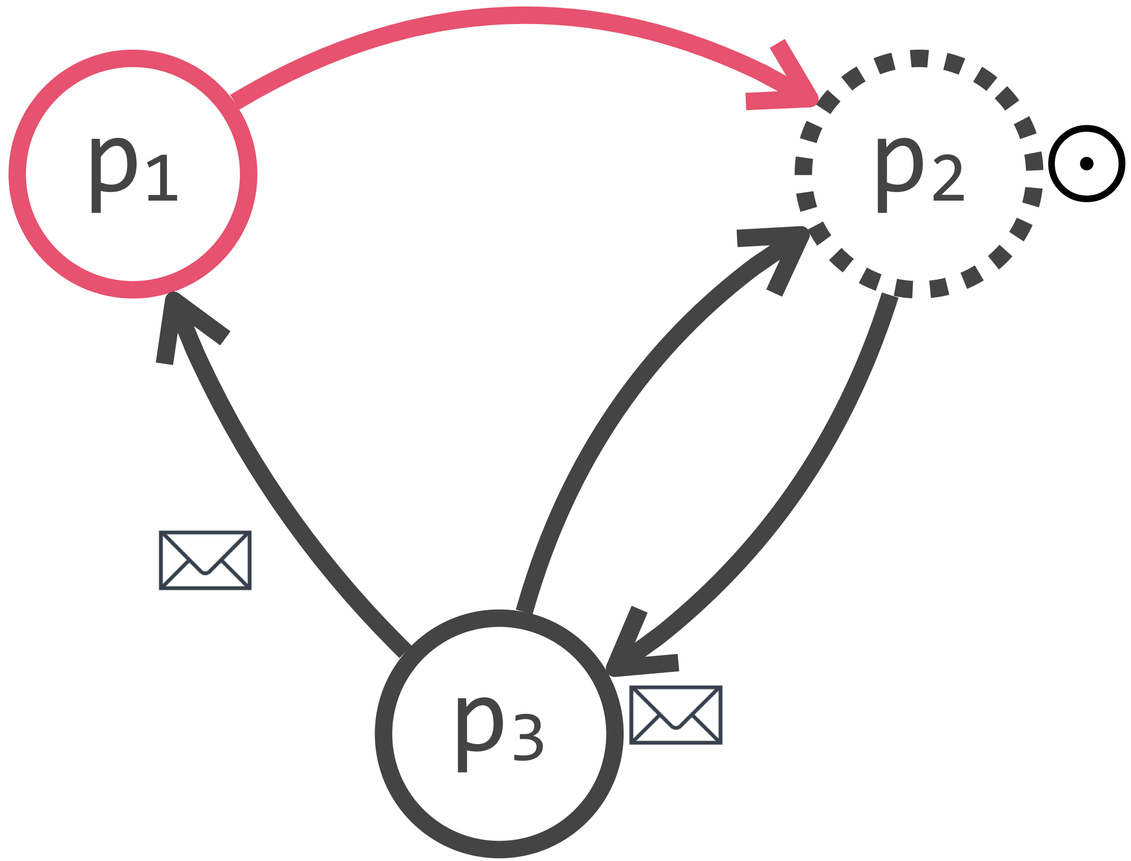
Snapshot

s1

before marker

after marker

The Chandy-Lamport Algorithm



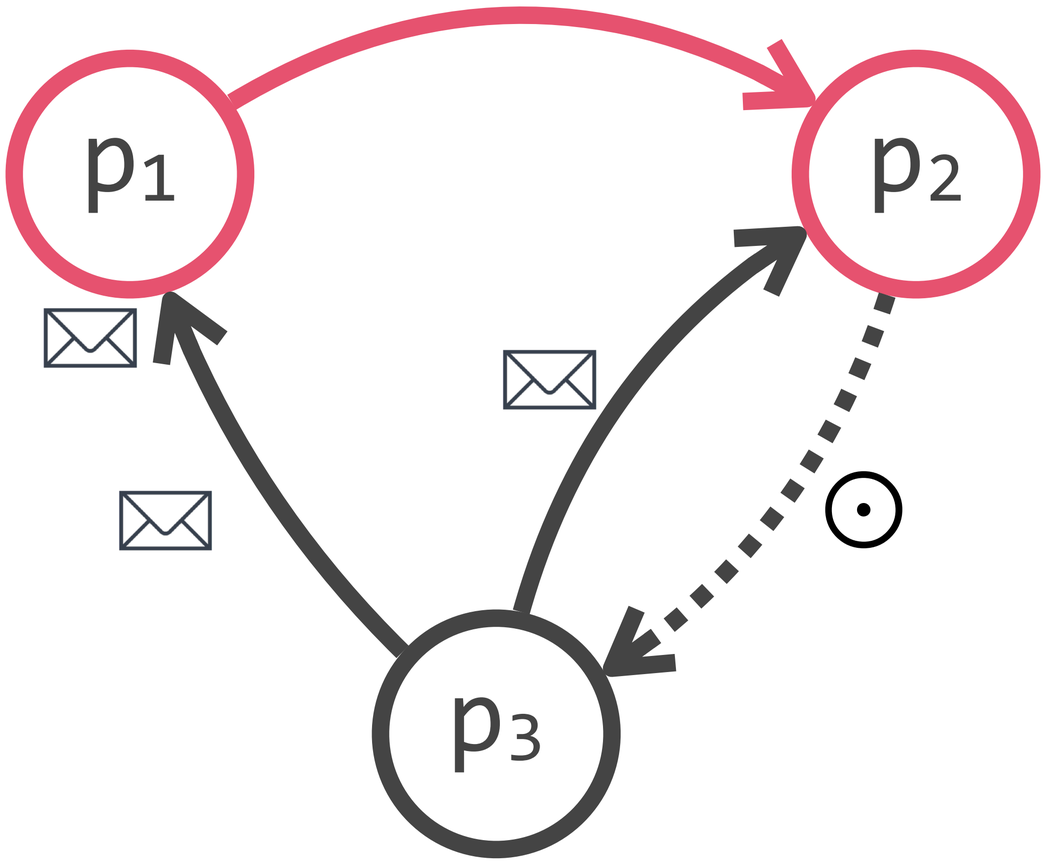
Snapshot

s1, s2

before marker

after marker

The Chandy-Lamport Algorithm



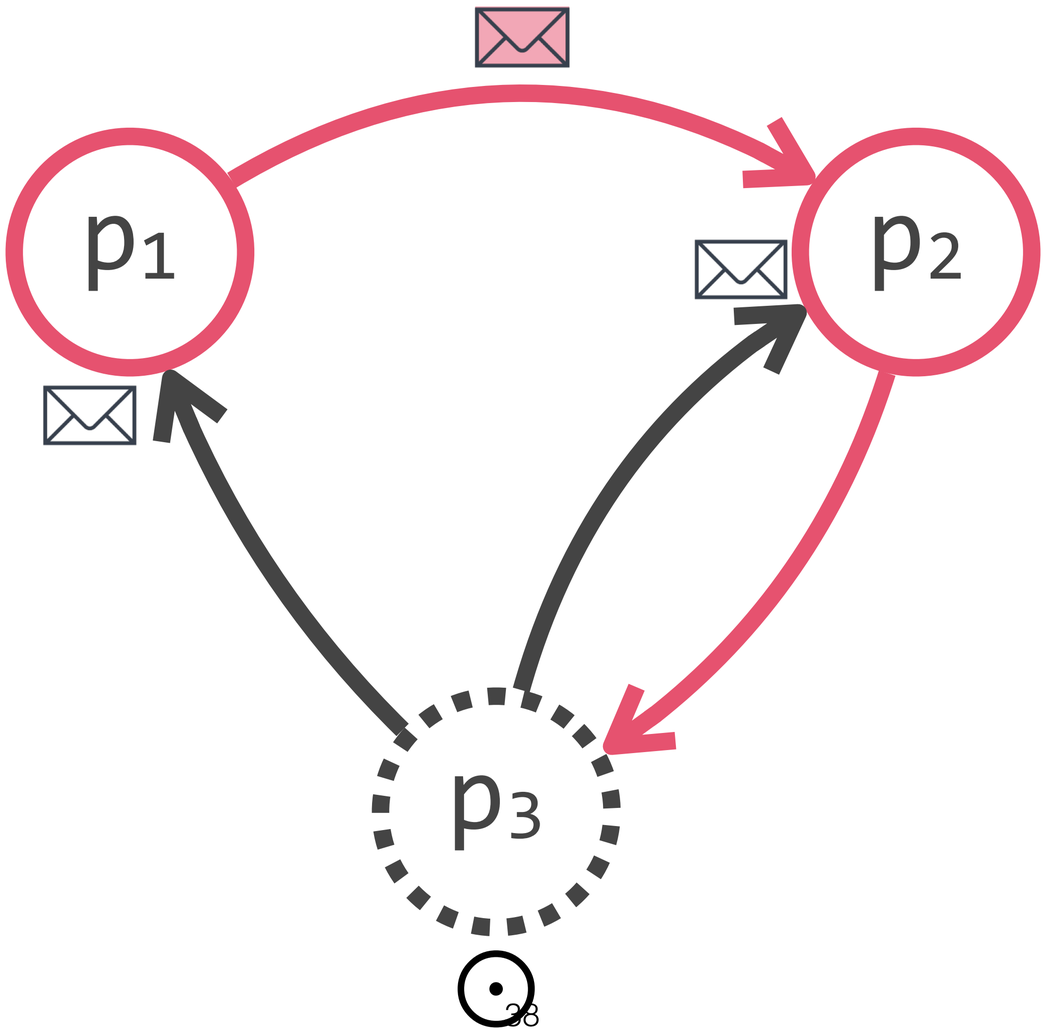
before marker

after marker

Snapshot

s1, s2

The Chandy-Lamport Algorithm



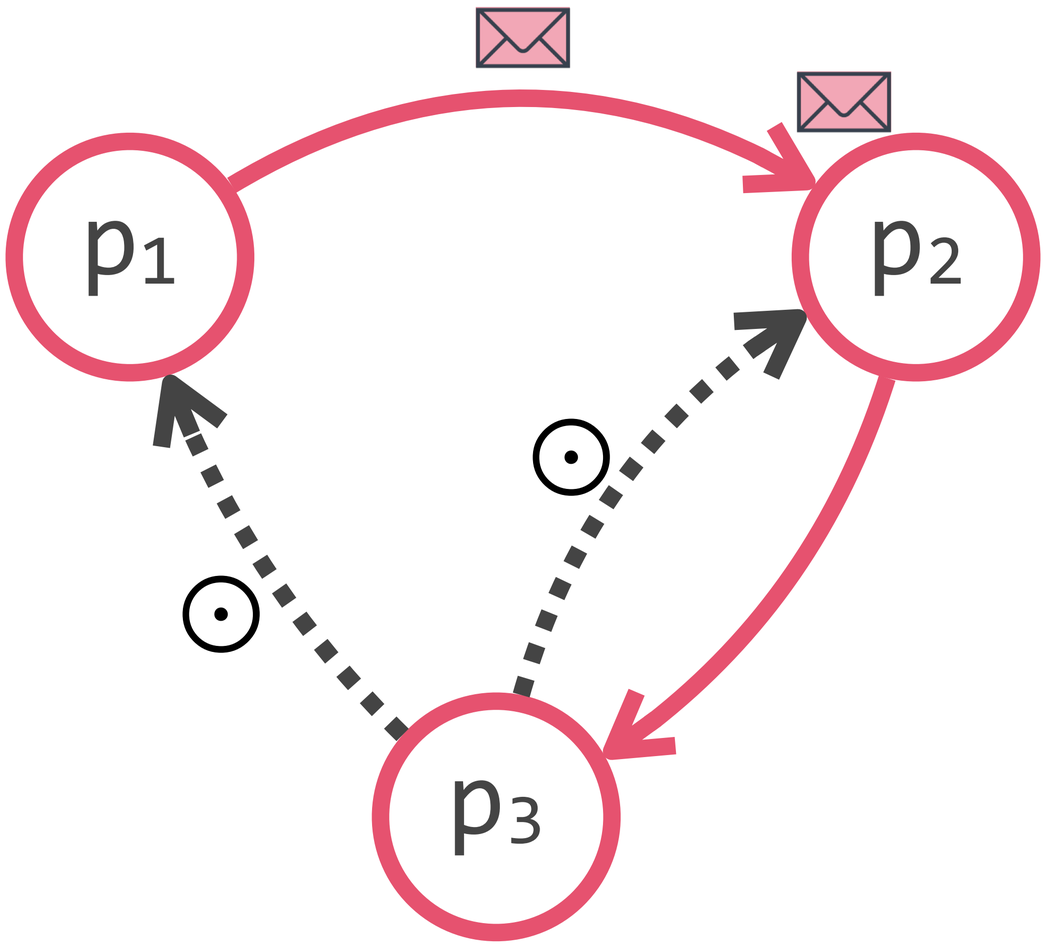
before marker

after marker

Snapshot

s1, s2, s3

The Chandy-Lamport Algorithm



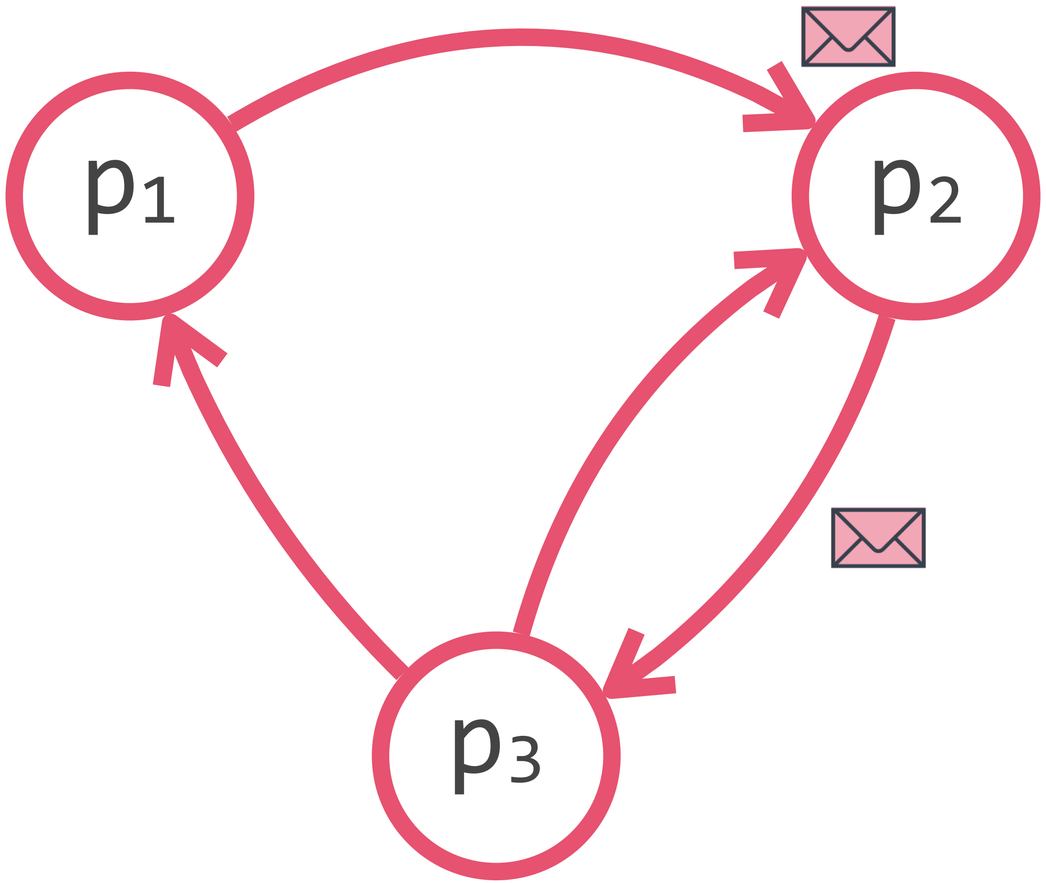
before marker

after marker

Snapshot

s1, s2, s3

The Chandy-Lamport Algorithm



Snapshot

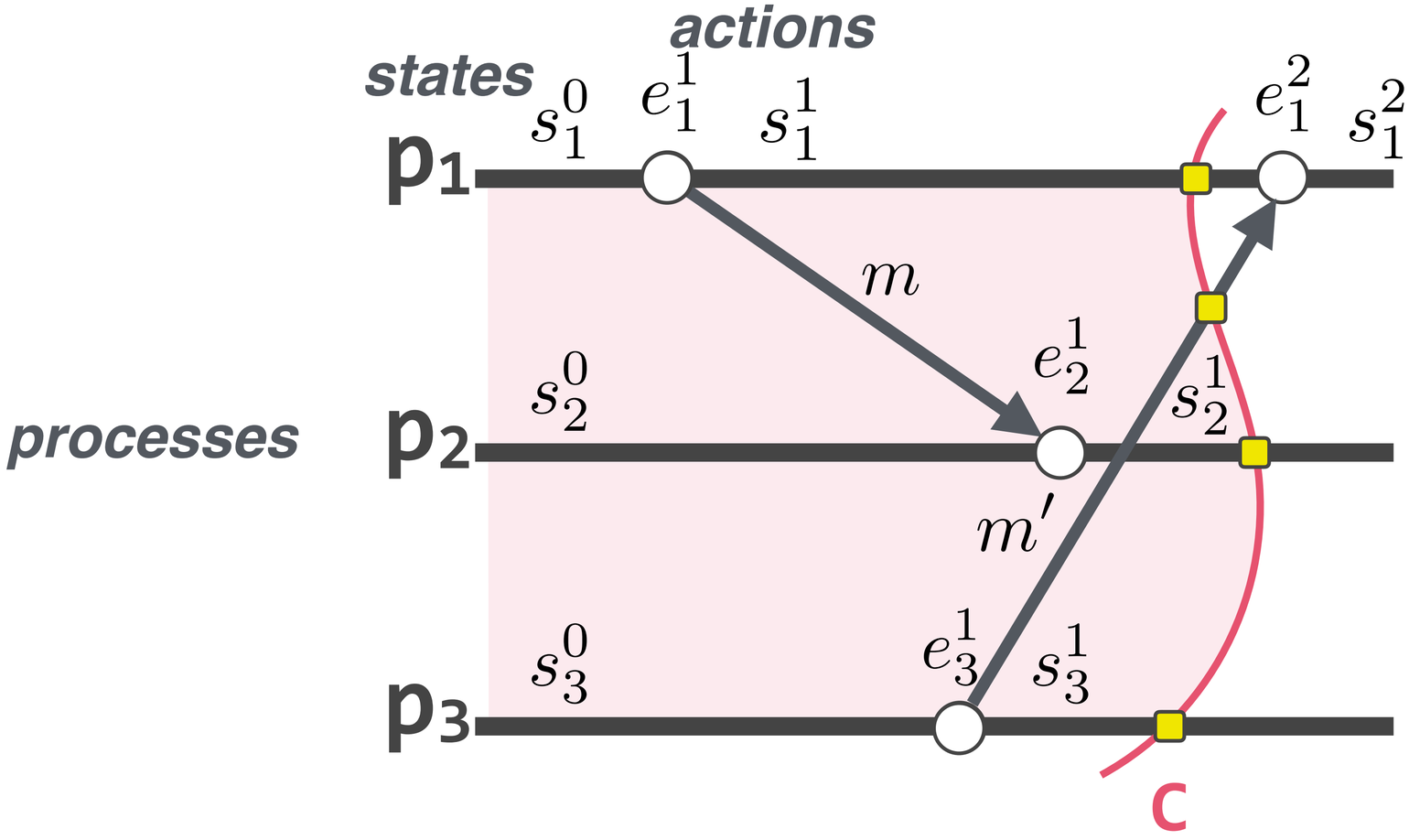
s1, s2, s3

before marker

after marker

Recap: Snapshotting Protocols

Traditional Snapshotting Protocols: Distributed Algorithms that capture system states that form a **distributed cuts** in a system execution



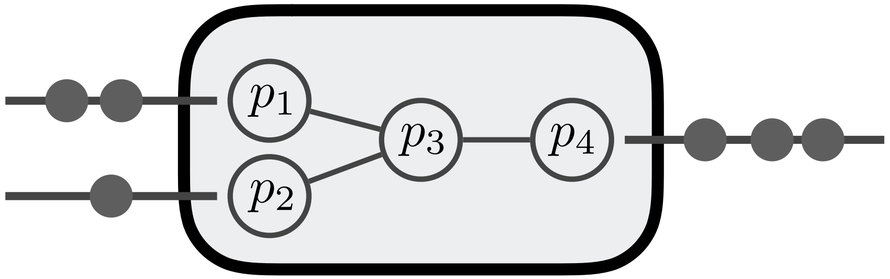
Snapshot of C

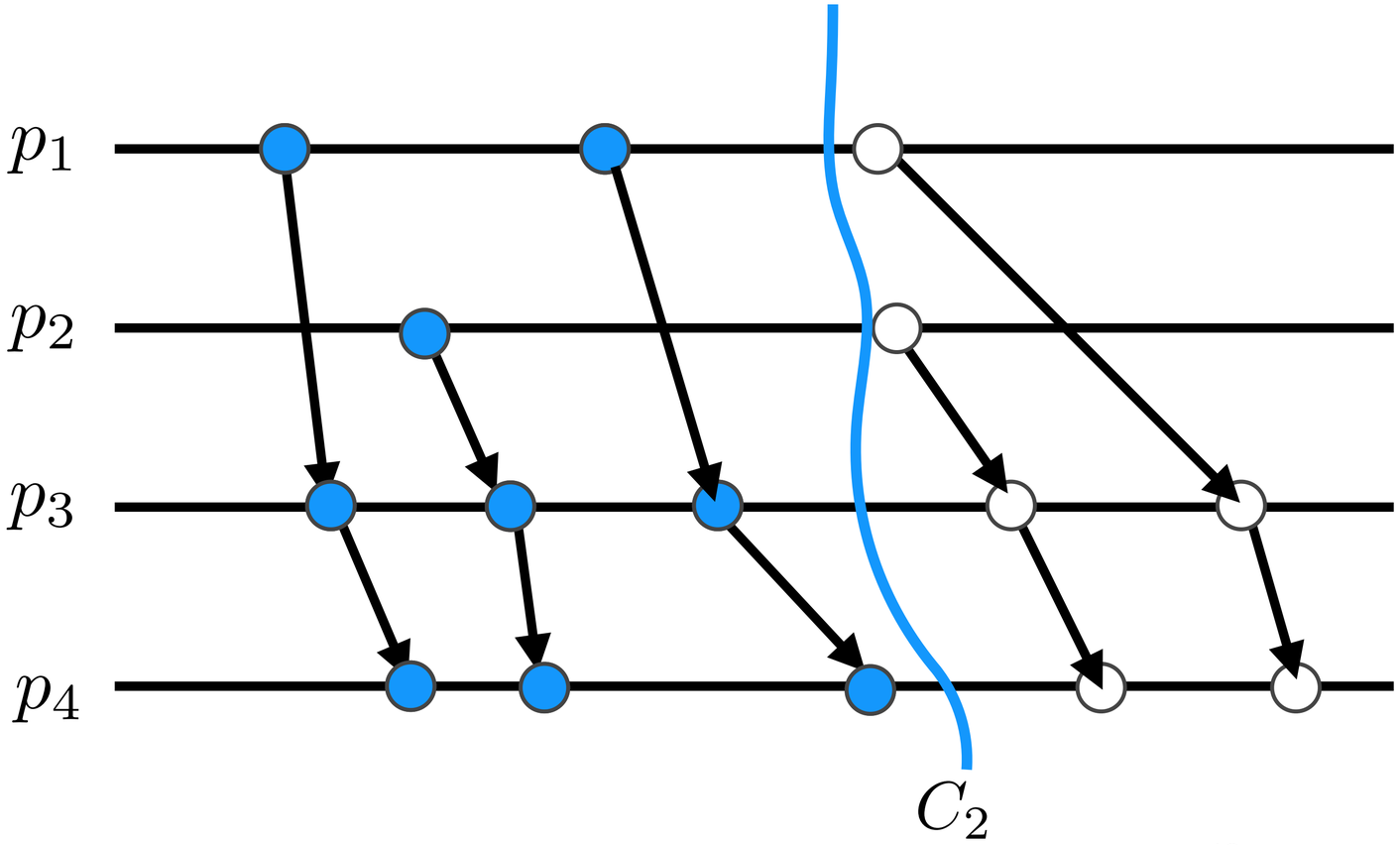
$\{s_1^1, s_2^1, s_3^1\}$

$\{m'\}$

But we need to complete *in-progress* computation in Stream Processing

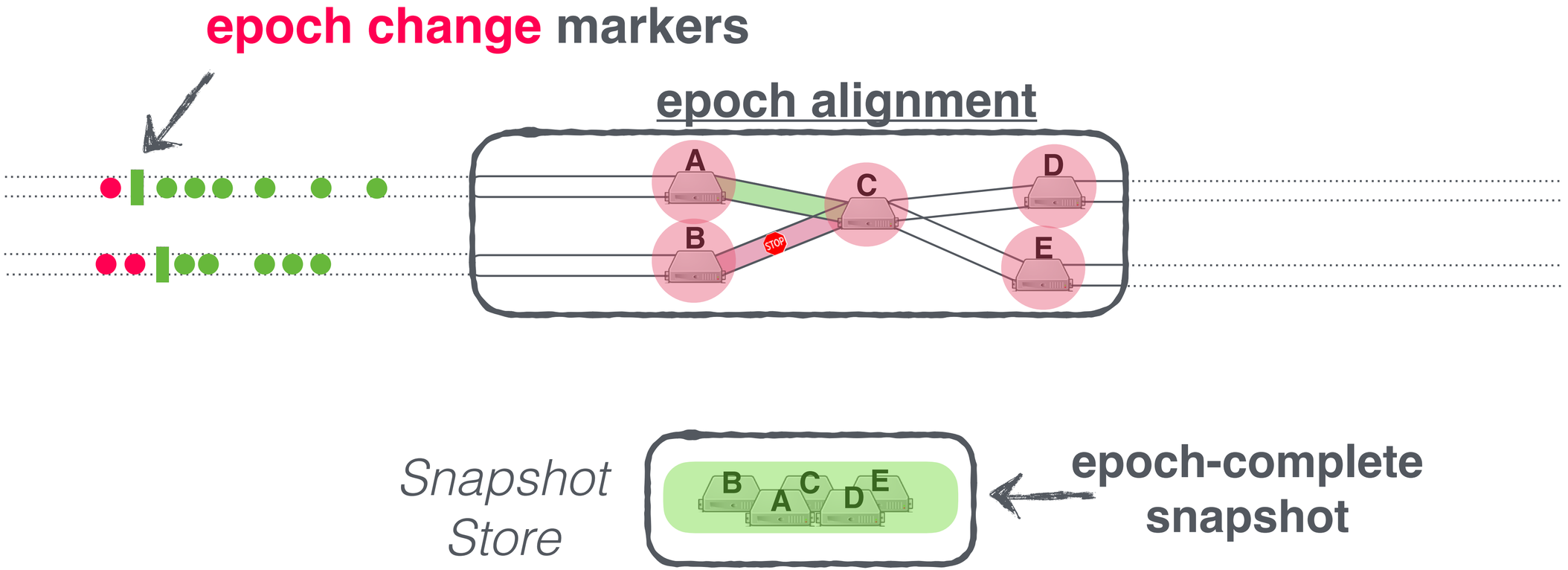
Epoch-Cuts



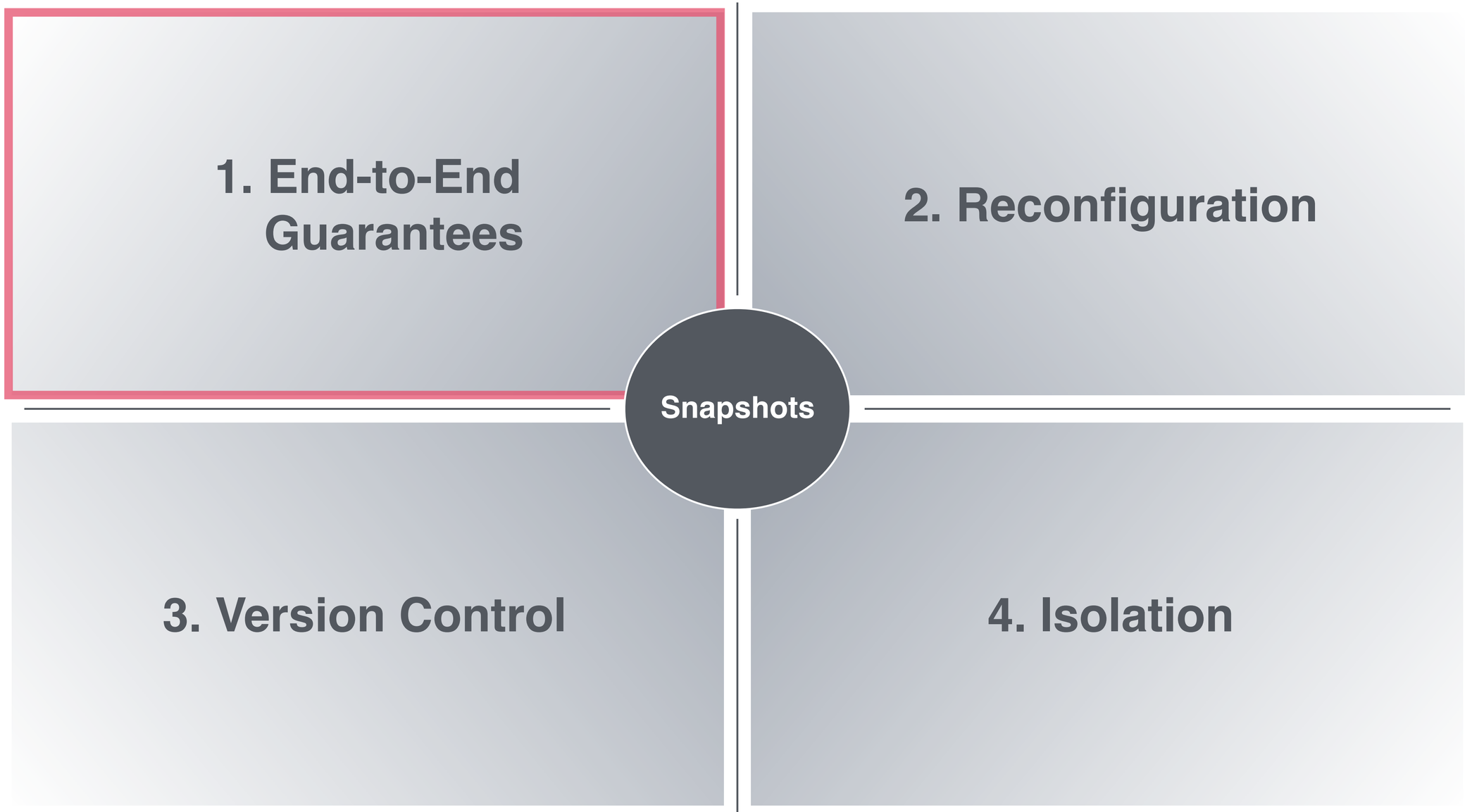


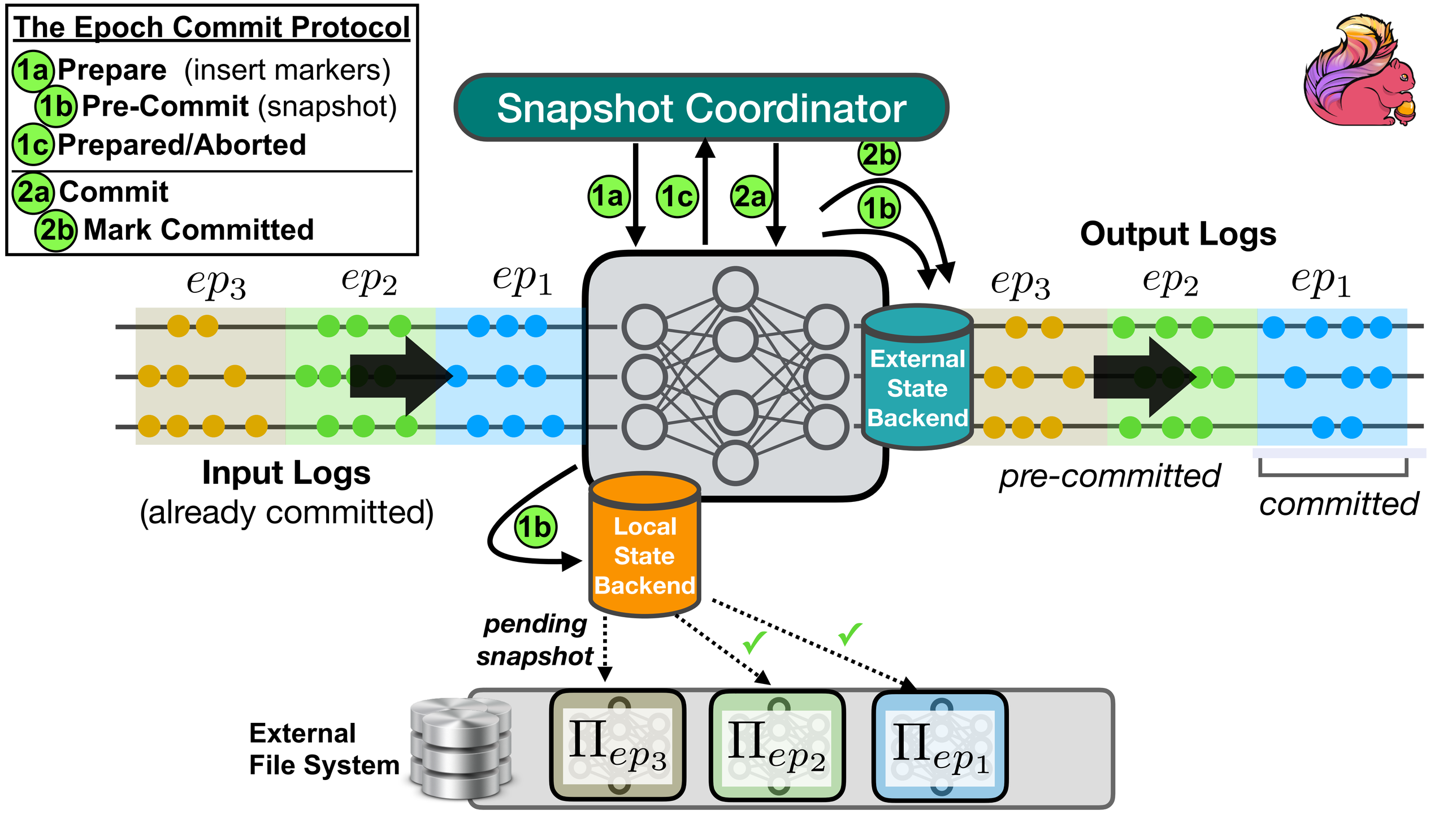
aka Distributed Cuts **without pending side effects**

Epoch Snapshotting Algorithm



Snapshot Usages

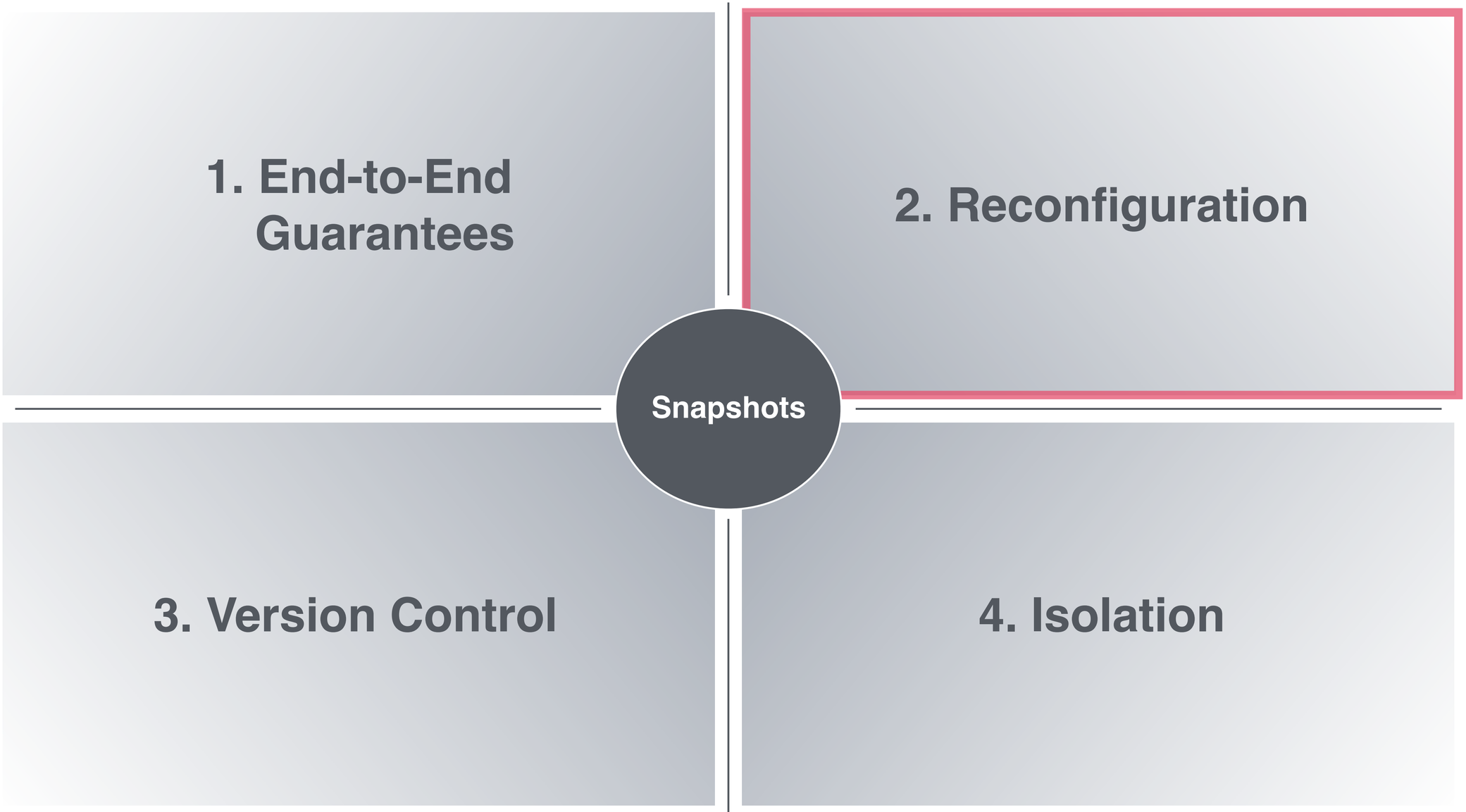



The Epoch Commit Protocol
1a Prepare (insert markers)
1b Pre-Commit (snapshot)
1c Prepared/Aborted
2a Commit
2b Mark Committed
Snapshot Coordinator
1a
1c
2a
2b
1b
Output Logs
ep_3
ep_2
ep_1
External State Backend
Input Logs
(already committed)
pre-committed
committed
Local State Backend
pending snapshot
External File System
Π_{ep_3}
Π_{ep_2}
Π_{ep_1}

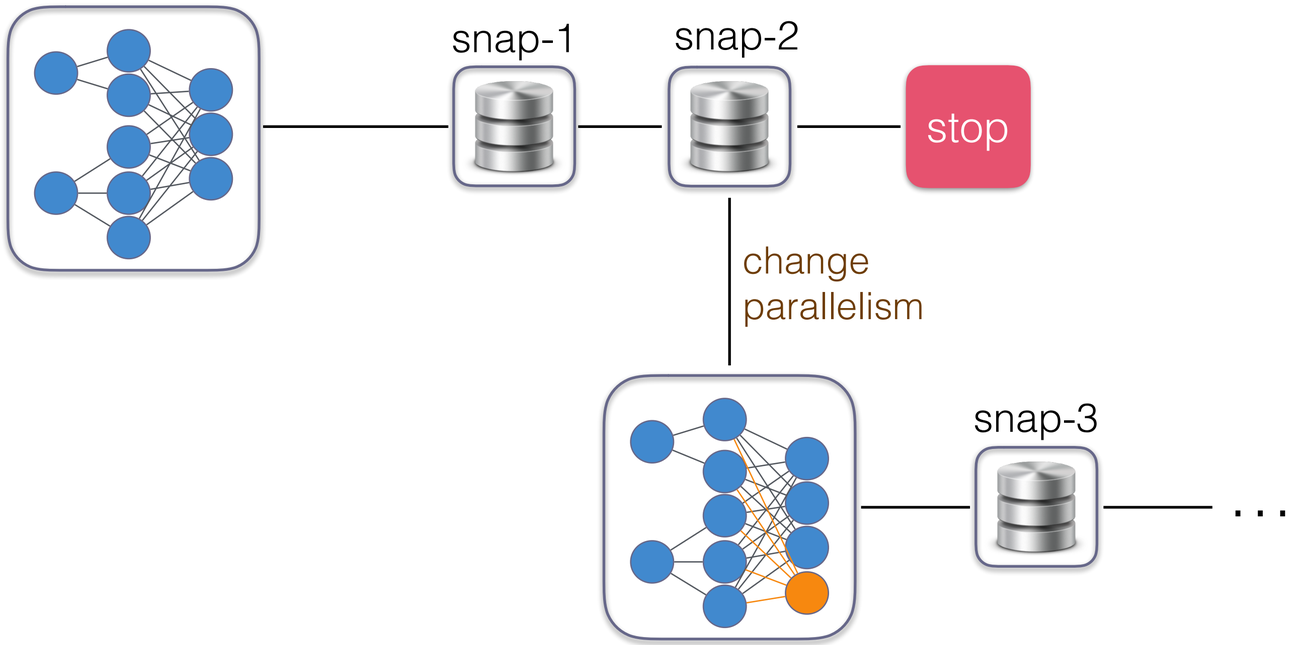
2-Phase Commit - Cheatsheet

	Start	**Pre-Commit**	**Commit**	**Abort**
Local (RocksdB)	new MemTable	flush MemTable / Snapshot SSTAbles	-	delete snapshot
Local (Heap)	-	deep copy (full snapshot)	-	delete snapshot
Kafka $\geq$0.11	new transaction	close/new transaction	commit transaction	abort transaction
Pravega	new Segment	seal Segment	commit Segment	delete Segment
HDFS	create File in tmp dir	close File (no writes)	move (atomic) file to committed Dir	truncate/delete file
DBMS (non-MVCC)	new WAL	snapshot WAL	execute WAL as transaction	drop WAL
DBMS (MVCC)	-	new version	incr version	decr version

Snapshot Usages



Dataflow Reconfiguration

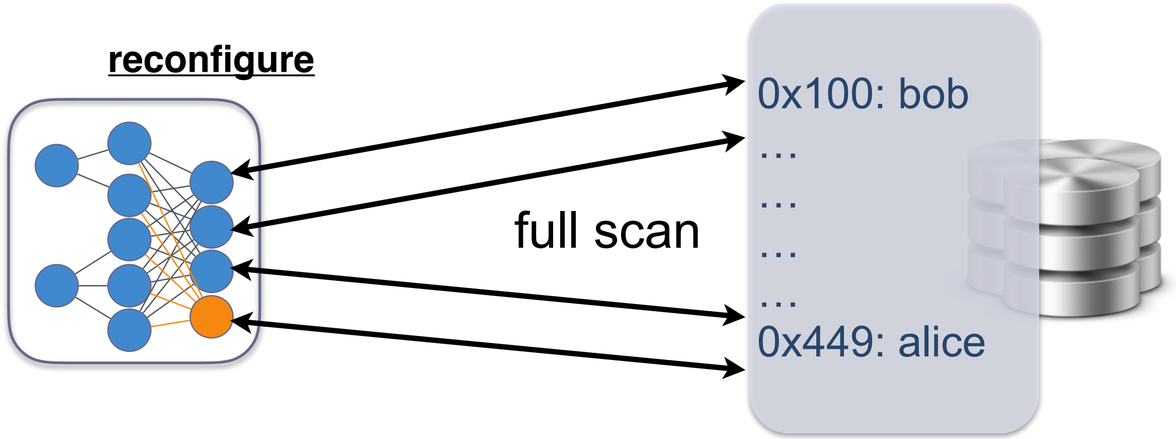


Problem: *How is state **repartitioned** from a snapshot?*

Reconfiguration: The Issue

Scan Remote Storage for Responsible Keys

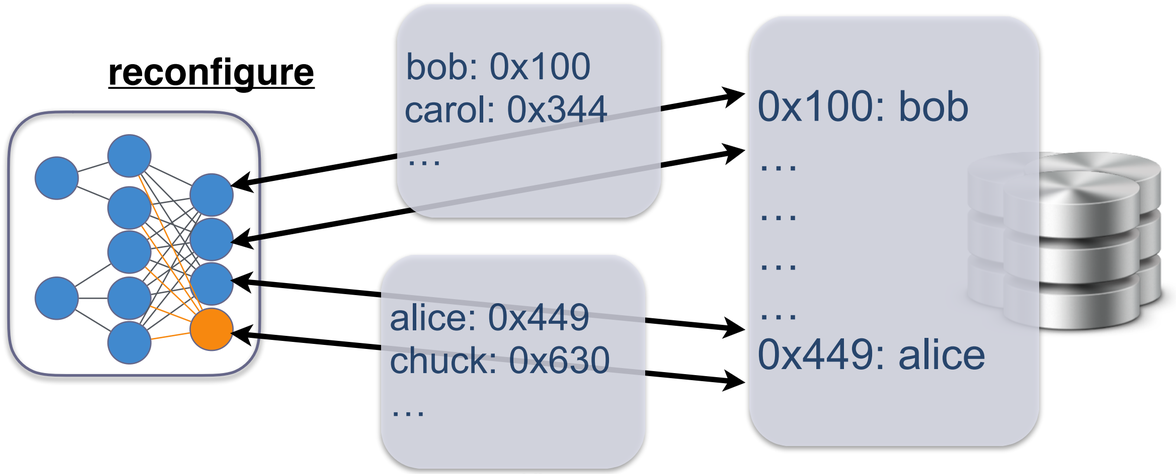
case I



too slow

Include Key Locations in Snapshot Metadata

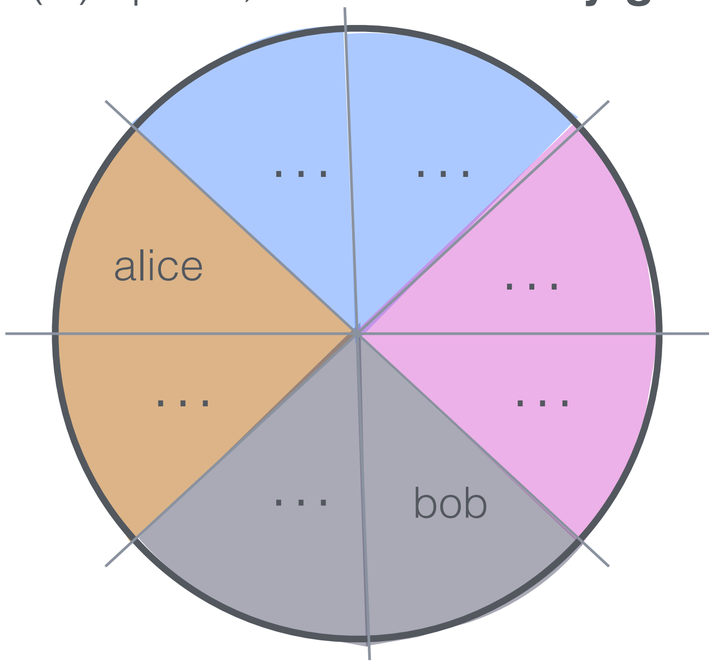
case II



too much

State Partitioning

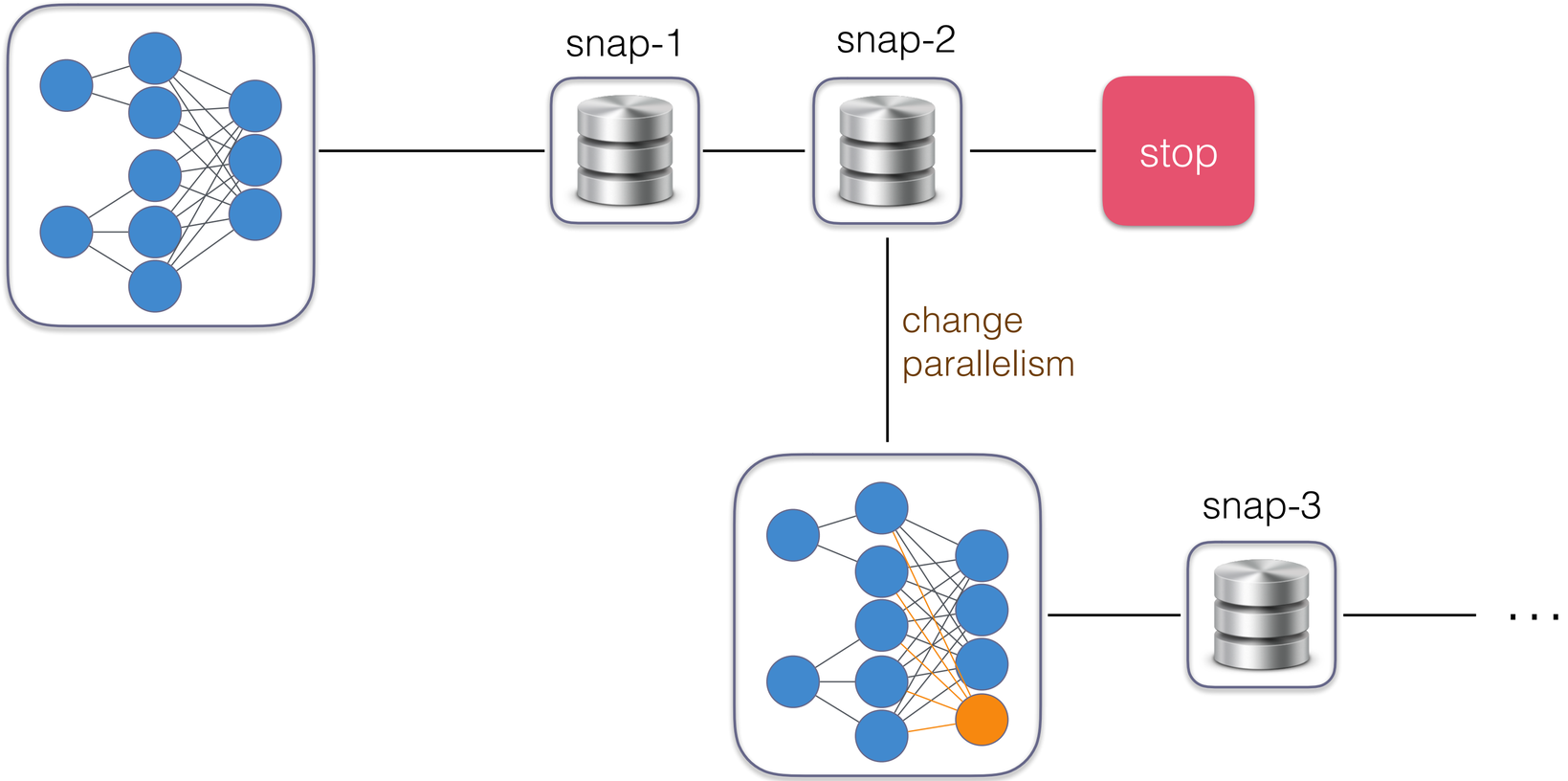
Pre-partition state in
hash(K) space, into **fixed n key-groups**



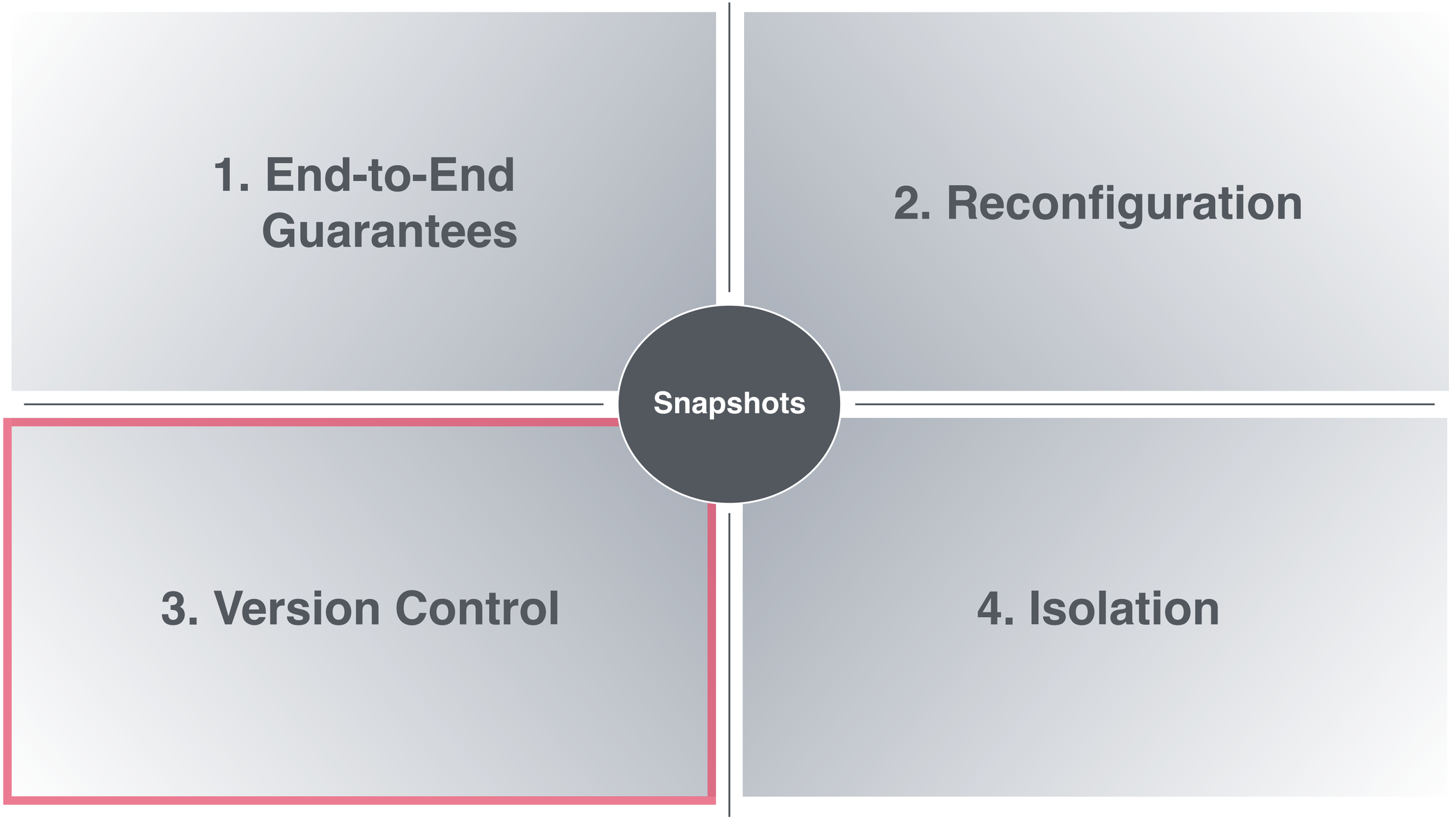
- **Snapshot Metadata:**
  Contains a reference per stored Key-Group (less metadata)
- **Reconfiguration:**
  Contiguous *key-group allocation to available tasks (less IO)*

Note: number of key groups controls trade-off between metadata to keep and reconfiguration speed

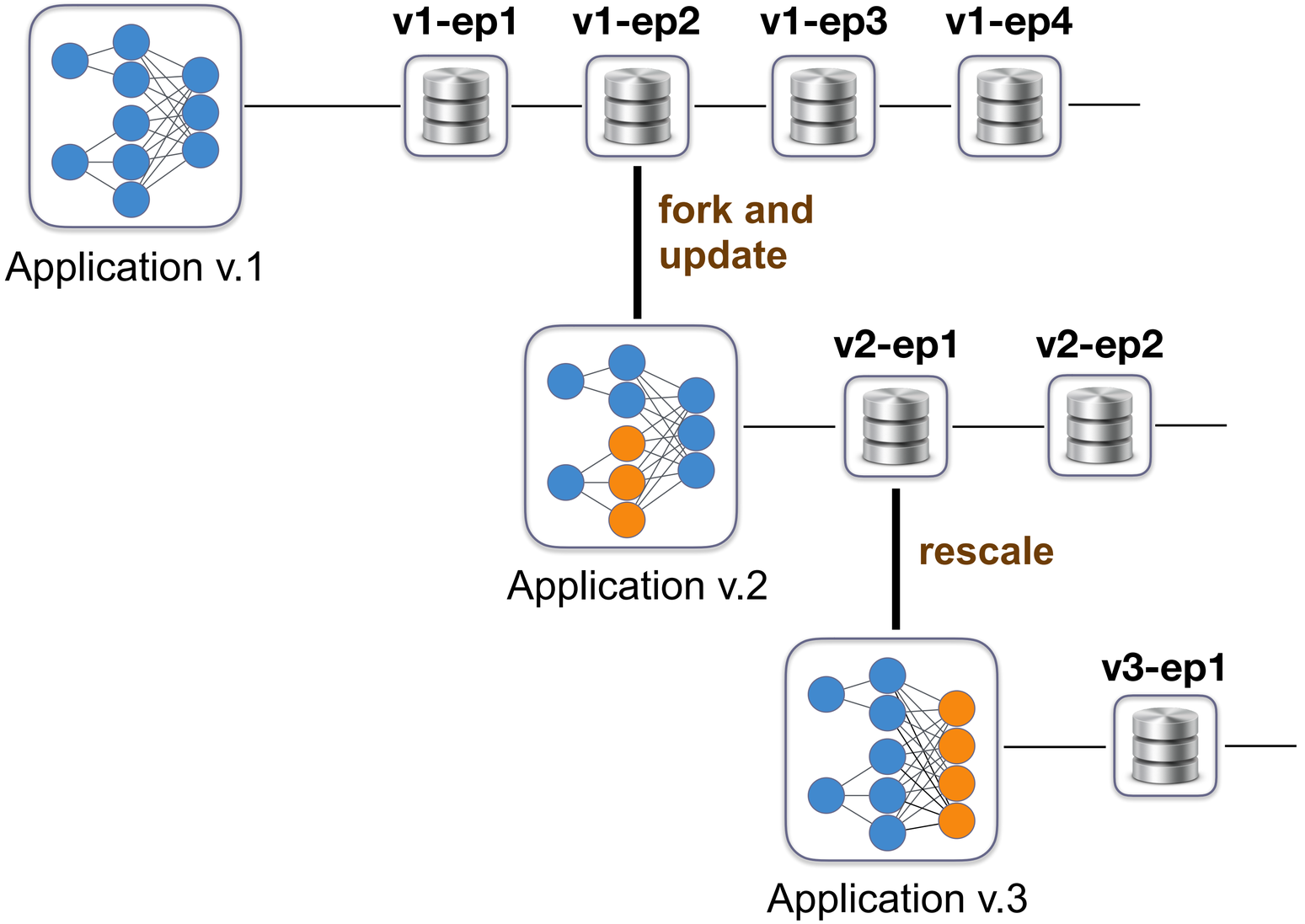
Usages:Reconfiguration



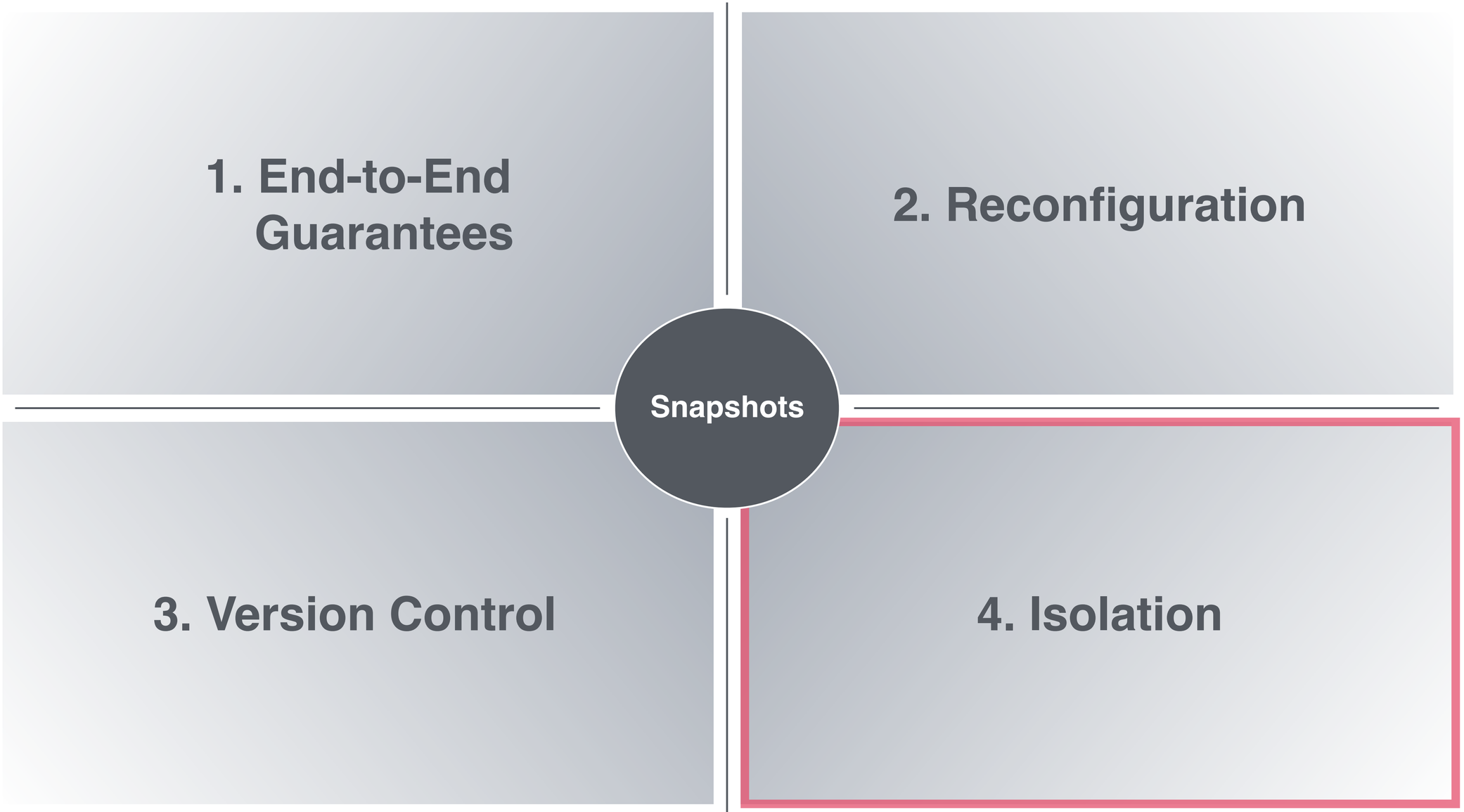
Snapshot Usages



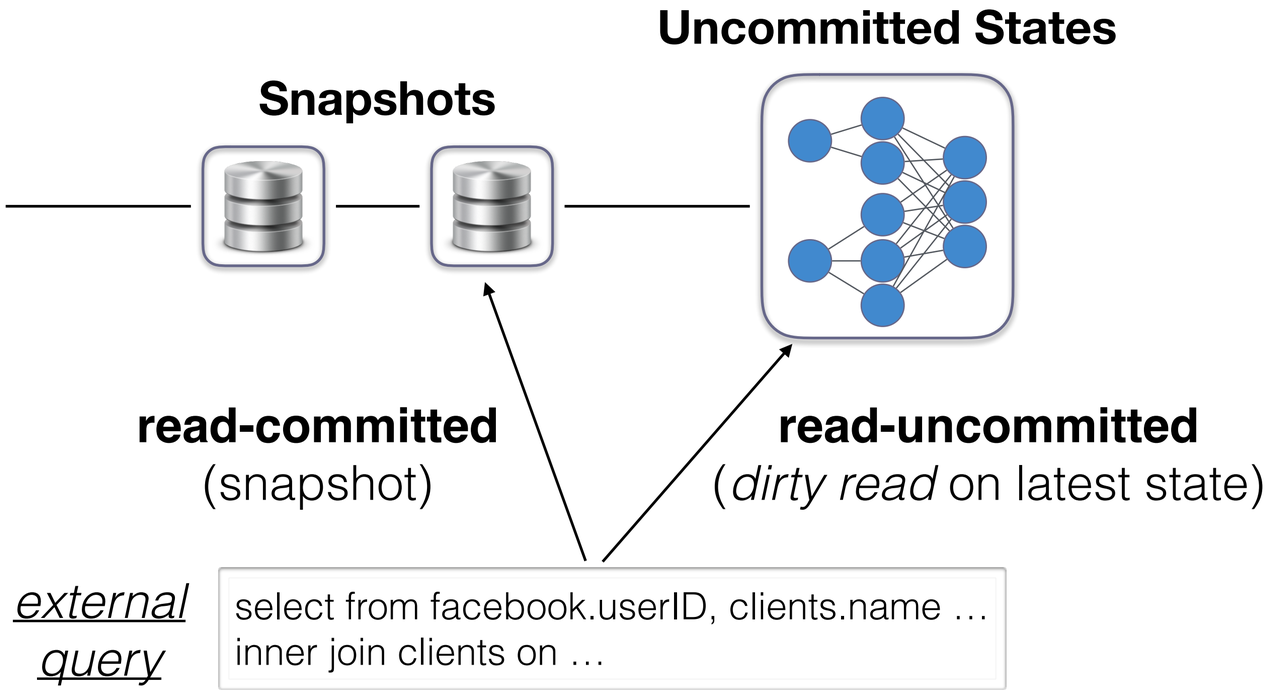
Usages: App Provenance



Snapshot Usages

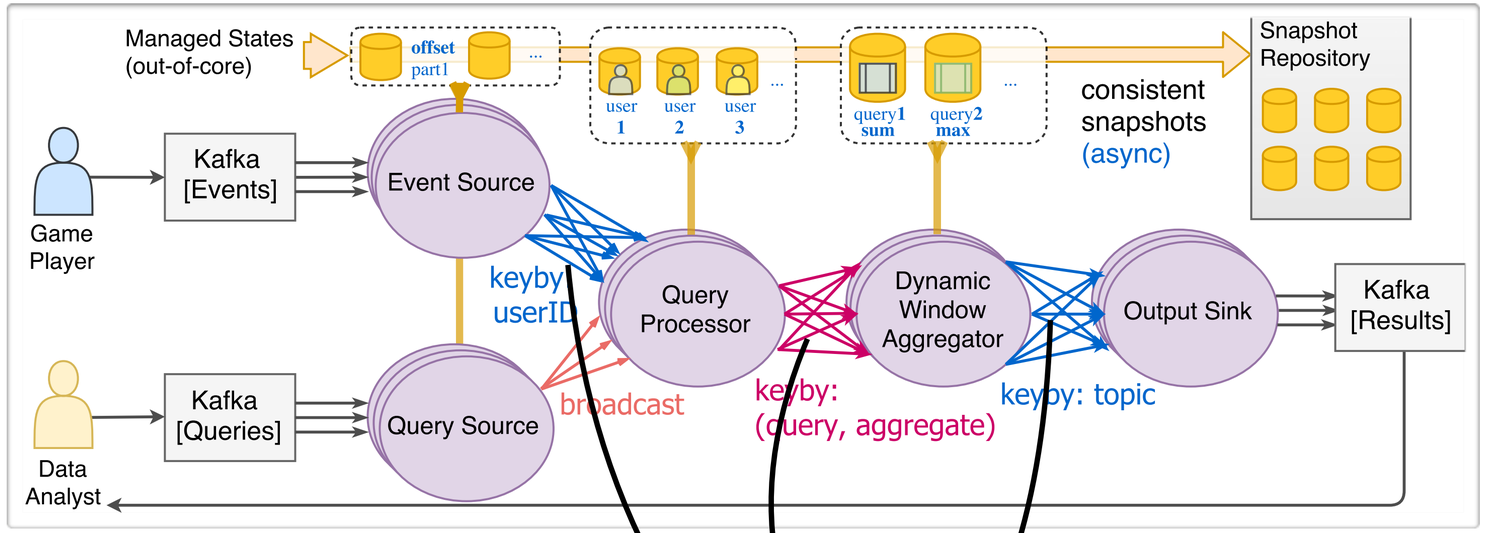


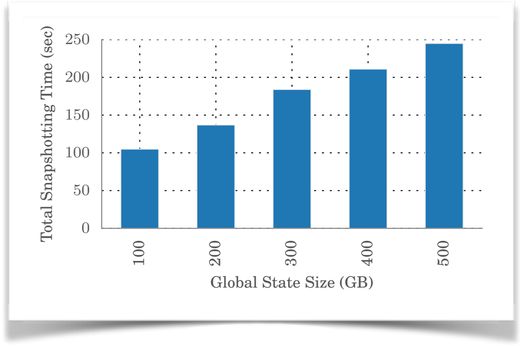
Usages: External Access Isolation



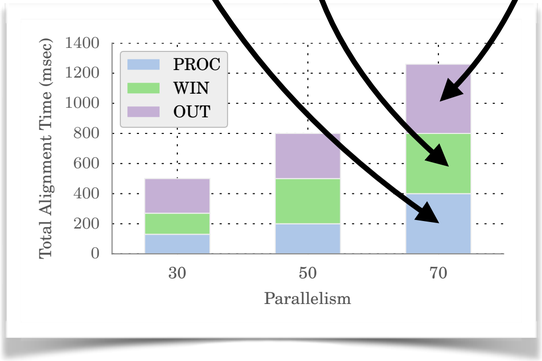
Further Optimisations

- **Asynchronous Snapshots**
  - make triggering snapshots cost-free.
- **Incremental Snapshots**
  - avoid full state copy and commit only **deltas**
  - make overhead of snapshots nearly **constant**
- **Both** are provided by Log-Structure-Merge backends, i.e. **Rocksdb.**





Total Commit Time (alignment + async copies)

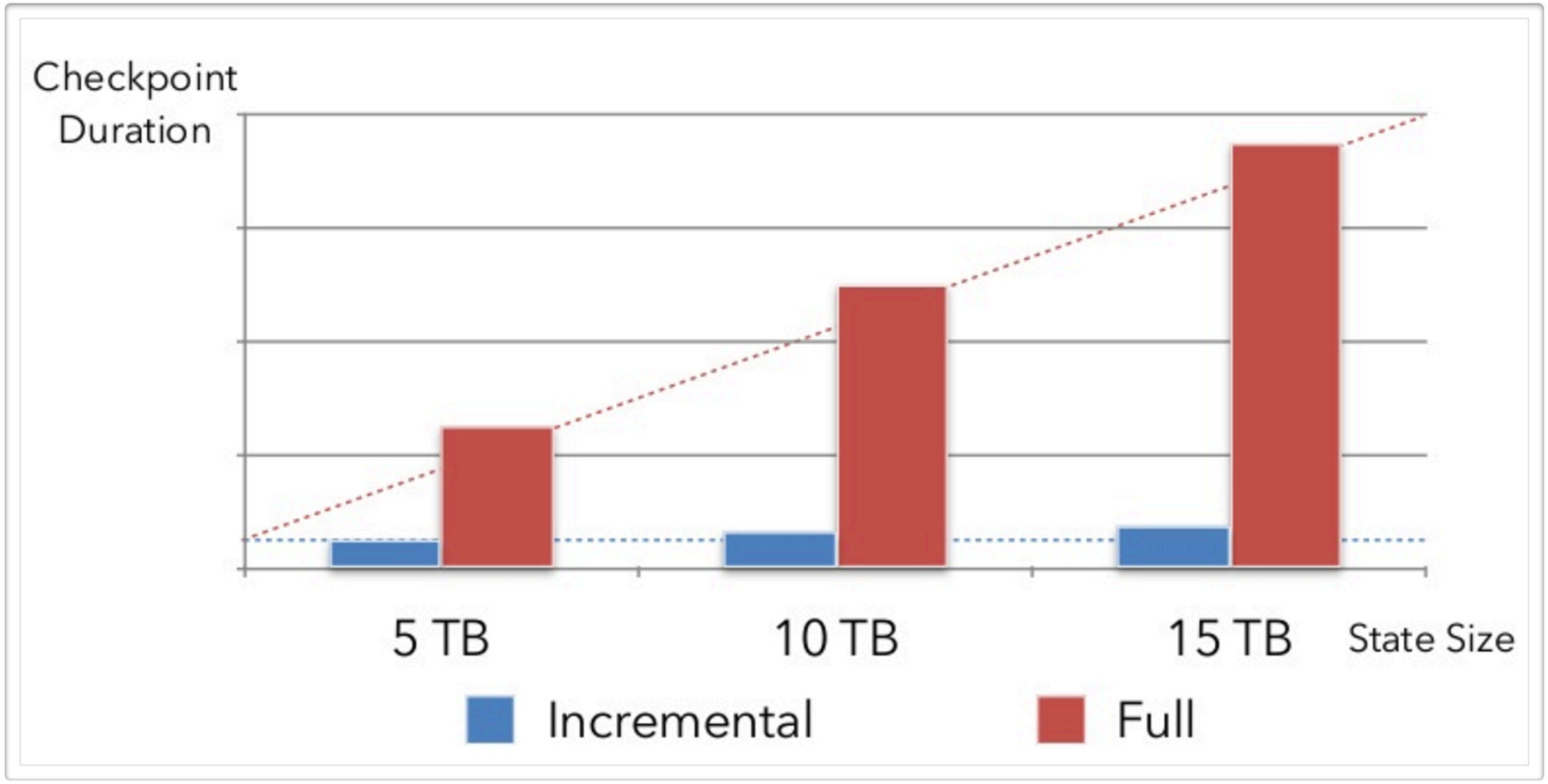


Actual Runtime Cost (alignment)

- #shuffles (keyby)
- parallelism

Flink@King

Incremental Snapshots



Further Readings

- **[Paper] State Management In Apache Flink - VLDB17**
- **[Thesis] Scalable and Reliable Data Stream Processing**
- **[Paper] The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. - VLDB15**
- **[Paper] Out-of-order processing: a new architecture for high-performance stream systems. - VLDB08**
- **[Blog] The world beyond batch: Streaming 101 by Tyler Akidau https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101**