Portals:

An Extension of Dataflow Streaming for Stateful Serverless

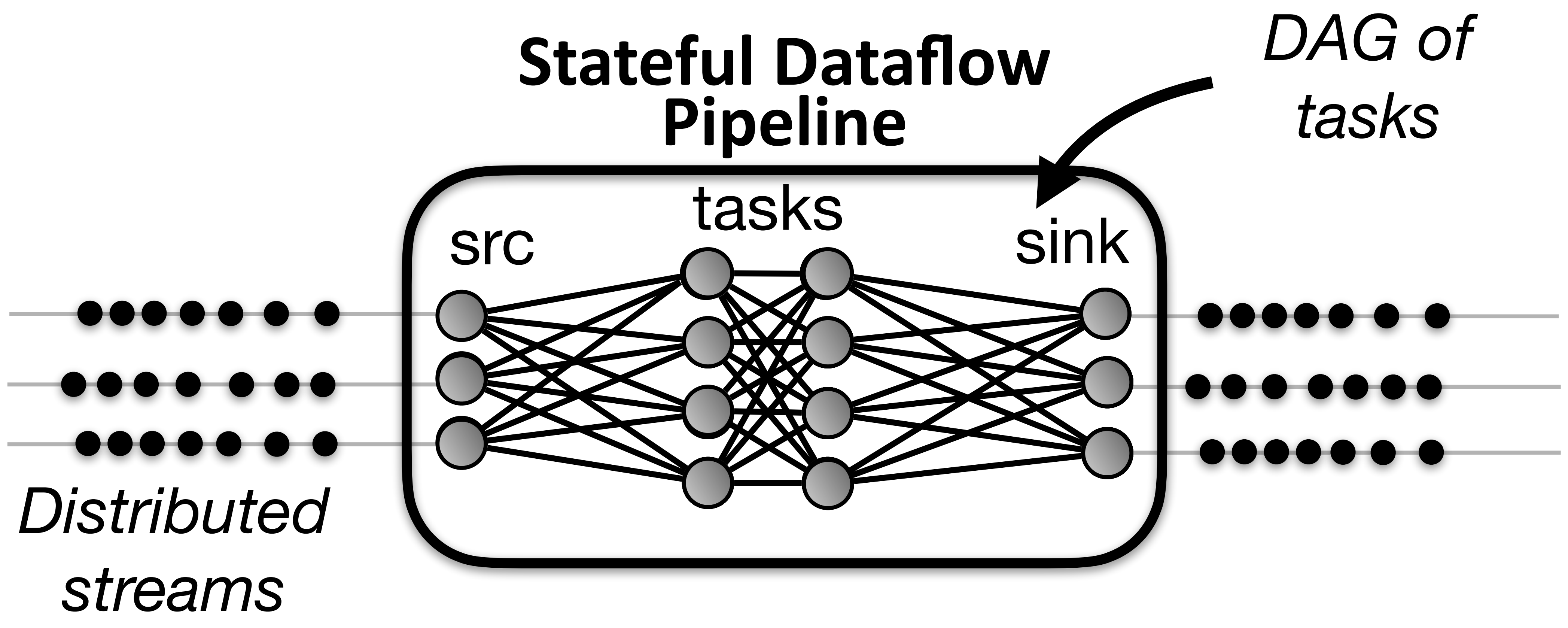
Jonas Spenger[1,2]
Paris Carbone[1,2]
Philipp Haller[1]

[1]EECS KTH Royal Institute of Technology, Stockholm, Sweden
[2]RISE Research Institutes of Sweden, Stockholm, Sweden

Onward! '22, December 8-10, Auckland, New Zealand

Dataflow Streaming

- Apache Flink; Google Dataflow; Kafka Streaming; Timely Dataflow; etc.
- Distributed
- Exactly-once processing guarantees

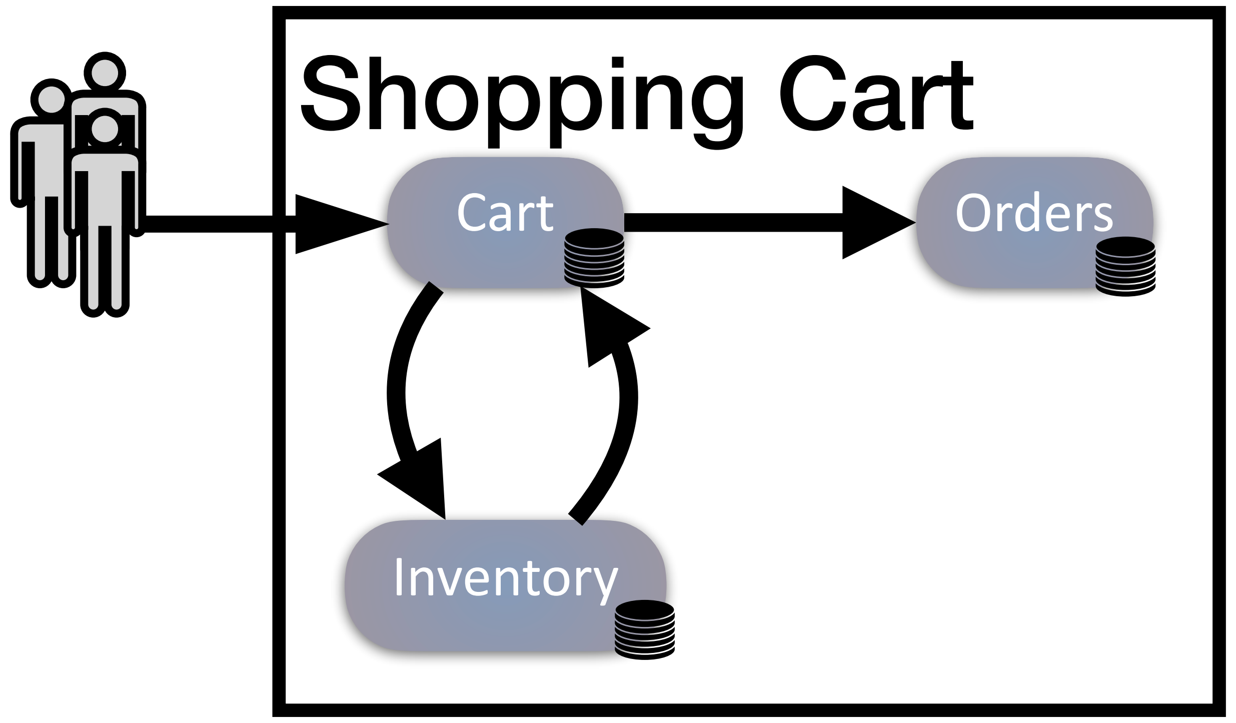


```
Pipeline()
  .source()
  .map( ... )
  .shuffle( ... )
  .map( ... )
  .sink()
```

1/3 Motivation

Modern Distributed Services

- Power critical infrastructure: Google Services; Microsoft; Uber; Netflix; Spotify; etc.
- Complex composition of communicating services.



Requirements:
- guarantees
- performance
- programming flexibility

Building Distributed Services is Difficult

- **Failures:** computers crash, messages get lost...
- **Scalability, response time**: workloads increase or decrease; services require low latency
- **Cloud and edge**: execution in heterogeneous environments
- **Privacy**: systems manage sensitive regulated data (GDPR, CCPA)

Building Distributed Services is Difficult

- **Failures:** computers crash, messages get lost...
- **Scalability, response time**: workloads increase or decrease; services require low latency
- **Cloud and edge**: execution in heterogeneous environments
- **Privacy**: systems manage sensitive regulated data (GDPR, CCPA)

We are asking too much of distributed software programmers

Stateful Serverless

- Microsoft Azure Durable Functions; Apache Flink Stateful Functions; Cloudburst; Beldi; Kalix/Cloudstate; etc.
- Stateful: system manages state
- Serverless:
  - The programmer should only need to write business logic
  - The stateful serverless system should fully manage all the other parts: reliability; scalability; execution; privacy; state.
- Exactly-once processing guarantees

Current Stateful Serverless Systems

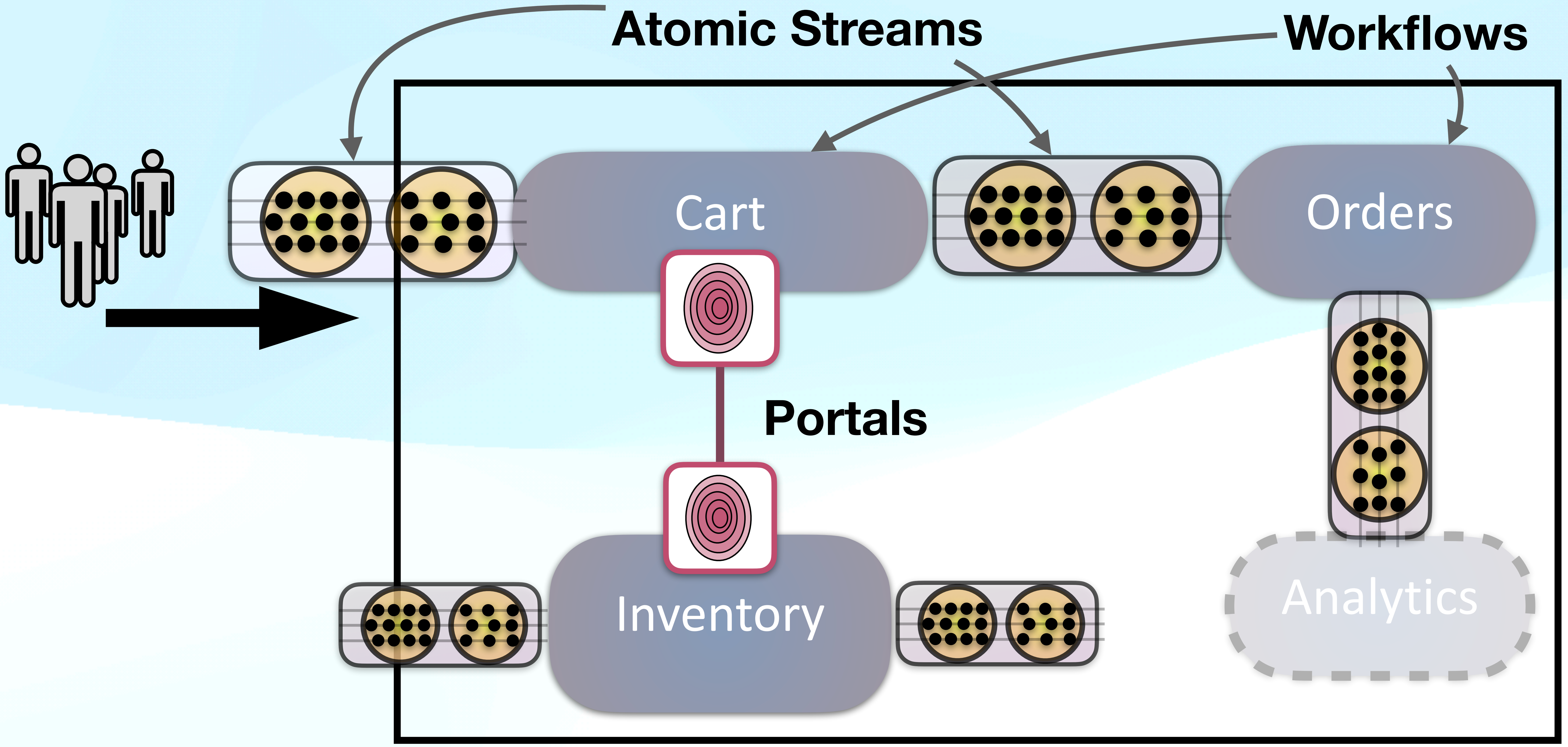
- There are many great systems: Durable Functions; Flink Stateful Functions; etc.
    - Exactly-once processing; dynamic/decentralized topology; parallelism;
- What can be improved?
    - Dataflow streaming style composition
    - Event ordering guarantees
    - Cyclic dependencies; RPC + futures

2/3 Portals

Portals

- Builds on Dataflow Streaming, harnessing: *exactly-once processing guarantees, performance, scalability*
- With some extensions for: *multiple services; dynamic topology; cycles; RPCs*

Portals Overview

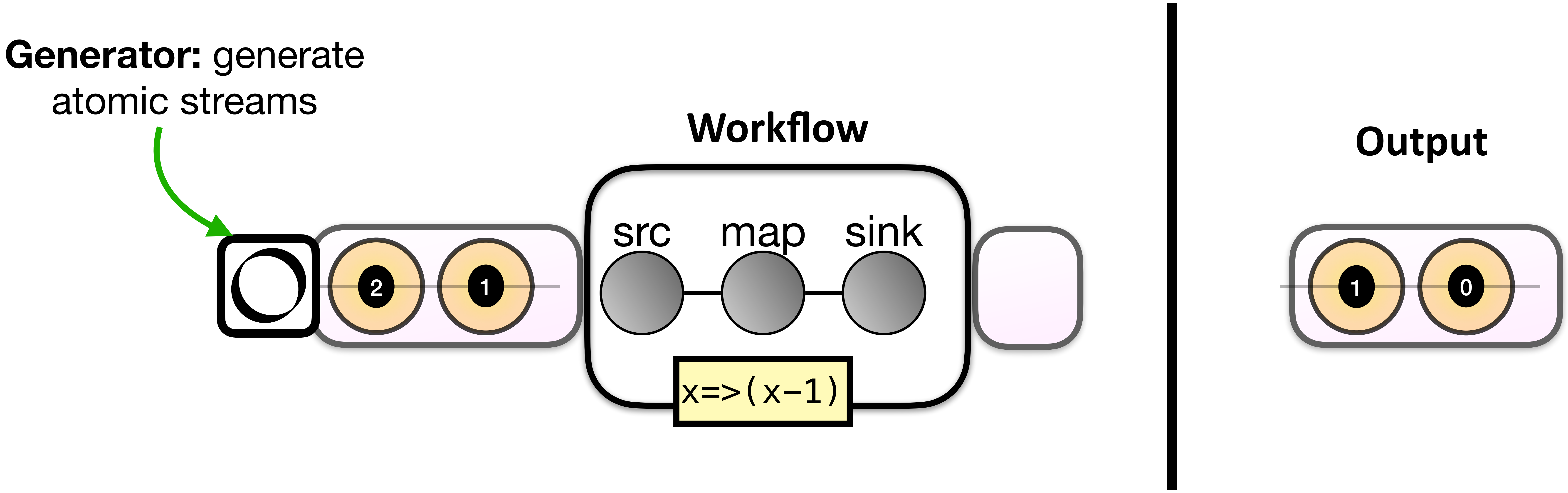


(cycles between workflows allowed, cycles within workflows not allowed)

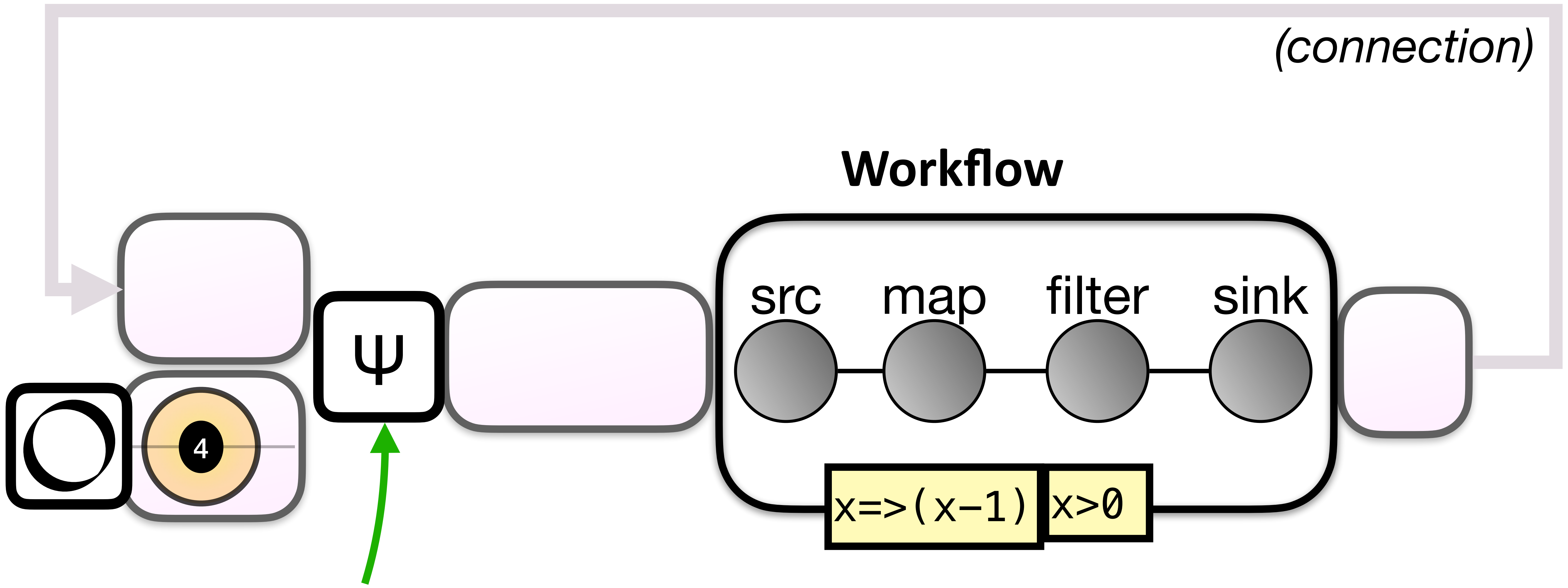
Portals abstractions:

1. *Atomic Streams*
2. *Workflows + Tasks*
3. *Portals*
4. *Applications + Registry*

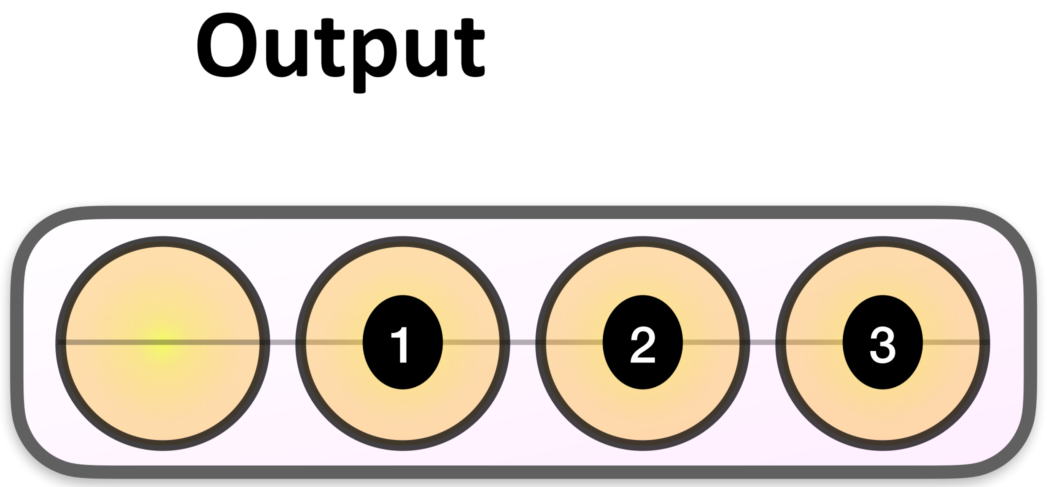
Example 1


Generator: generate atomic streams
Workflow
src
map
sink
x=>(x−1)
2
1
Output
1
0

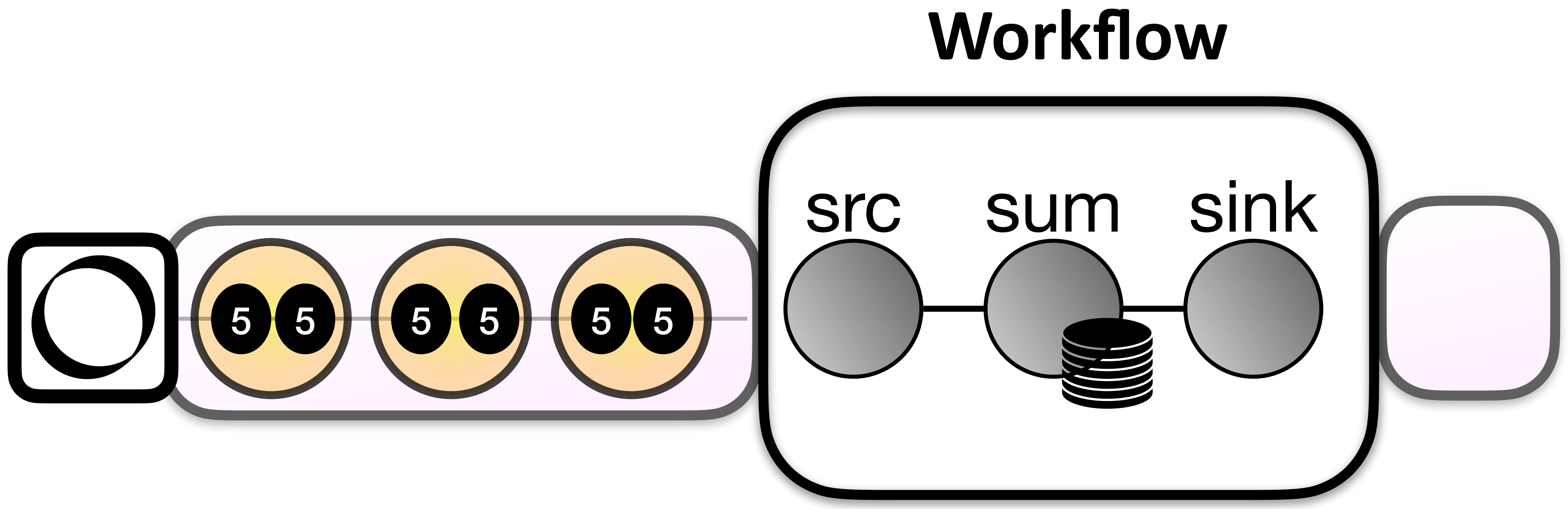
Example 2

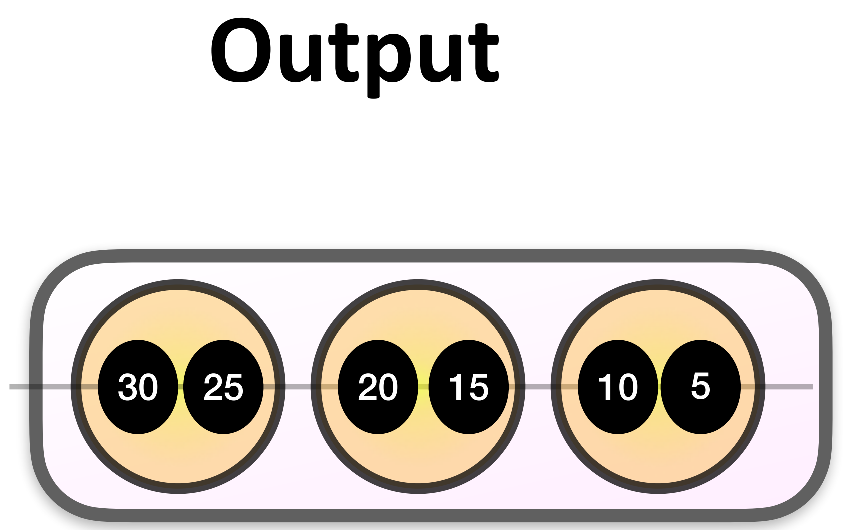


Sequencer: sequence atomic streams



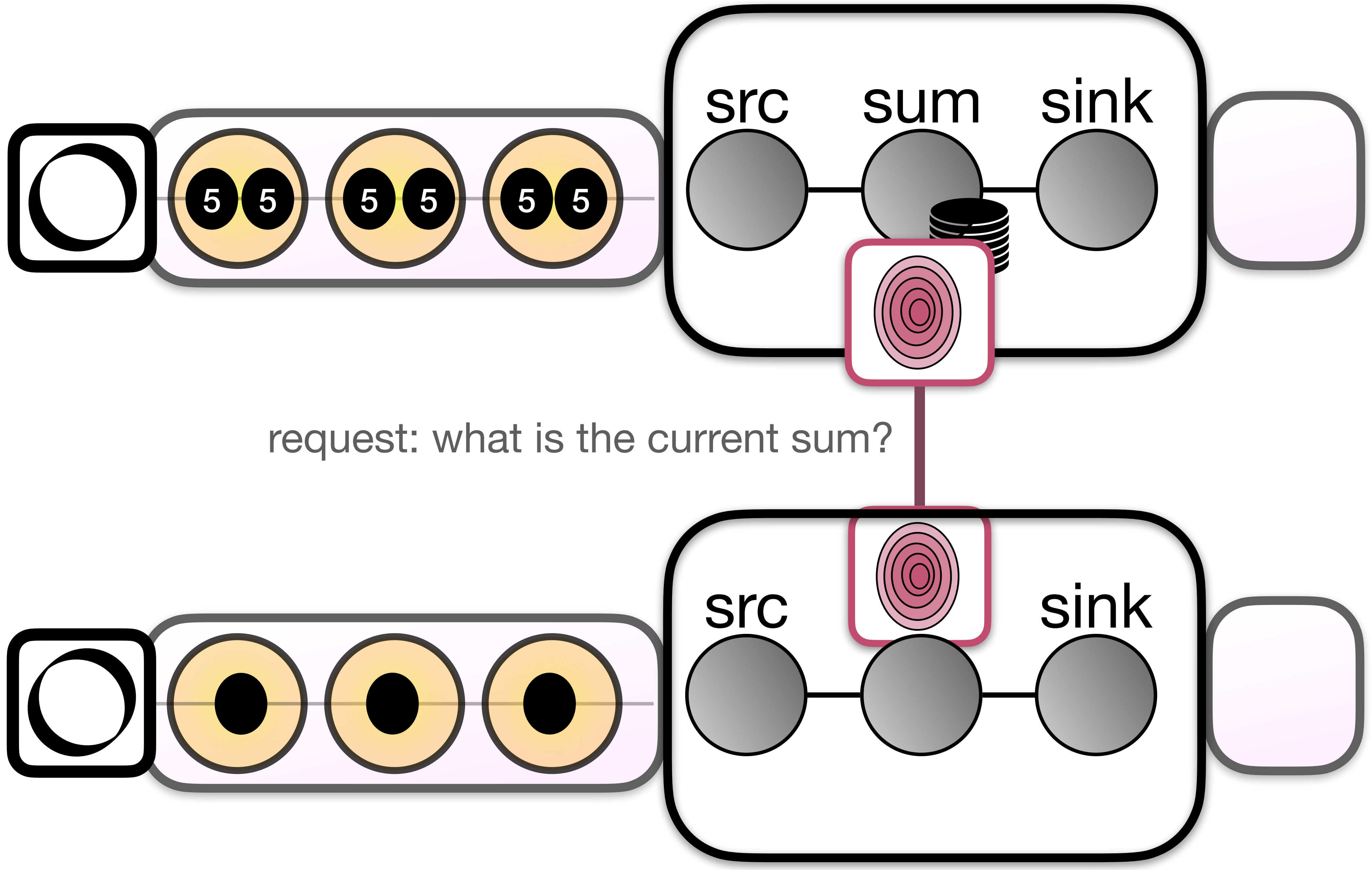
Example 3





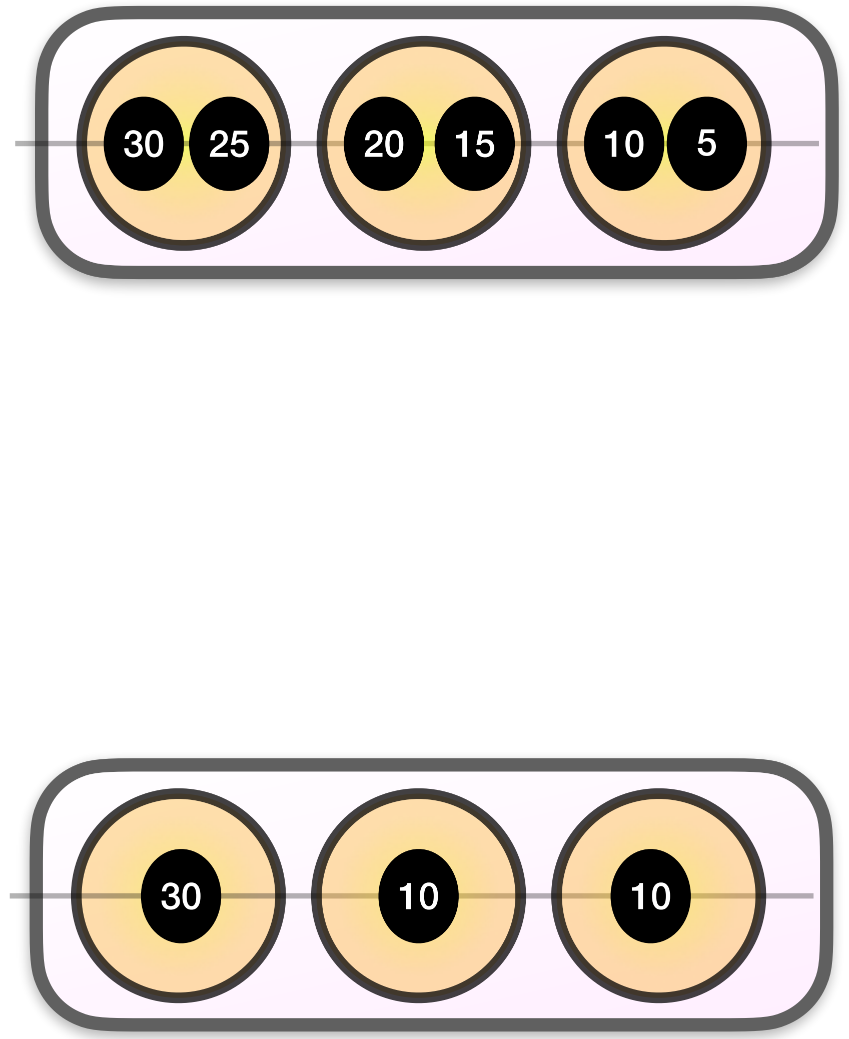
Example 4

Workflow



request: what is the current sum?

Output



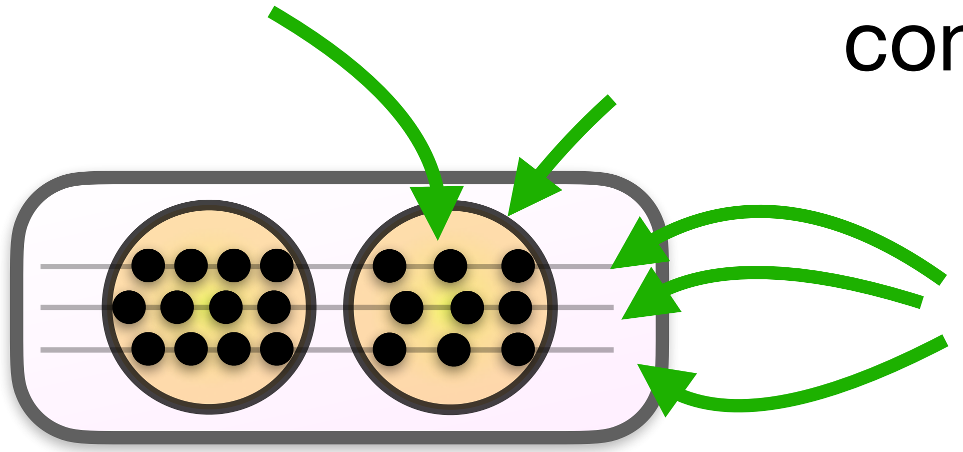
- Output is not deterministic;
- Only sums divisible by 10 are observed

Atomic Streams

Events (black dots): application events

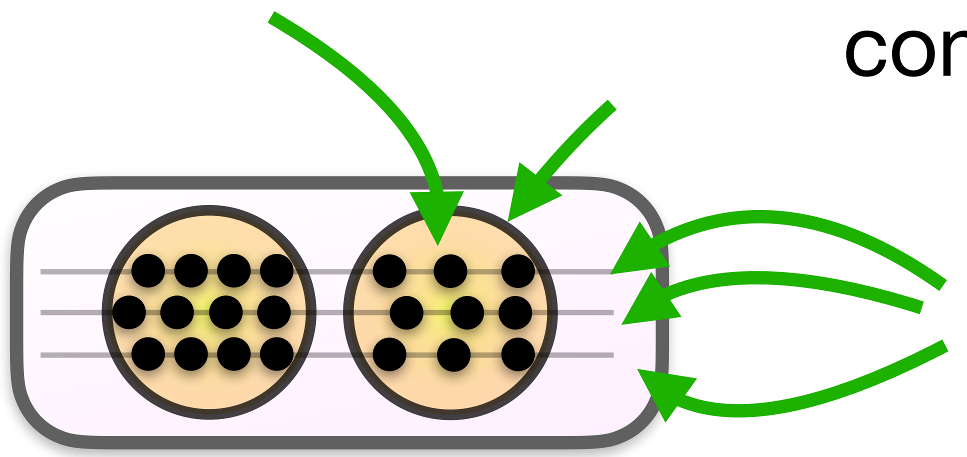
Atom (big circles): Sequence of events, transactional unit of computation.

Atomic Stream: totally ordered, distributed stream of atoms



Partitions: distributed/sharded streams of events

Atomic Streams

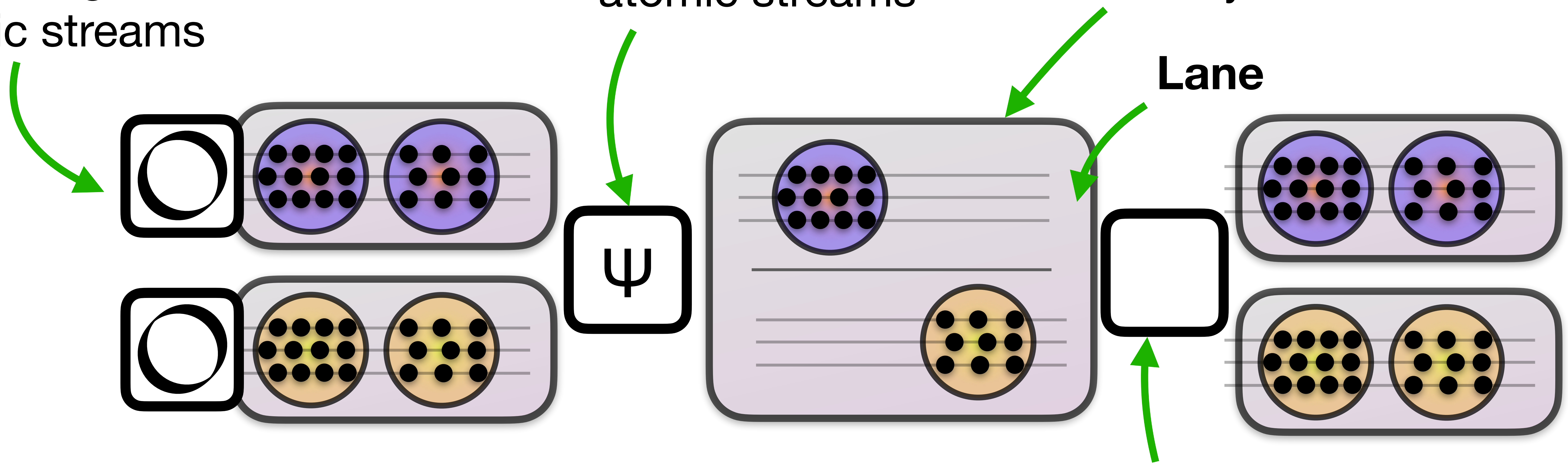


Events (black dots): application events

Atom (big circles): Sequence of events, transactional unit of computation.

Atomic Stream: totally ordered, distributed stream of atoms

Partitions: distributed/sharded streams of events



Generator: generate atomic streams

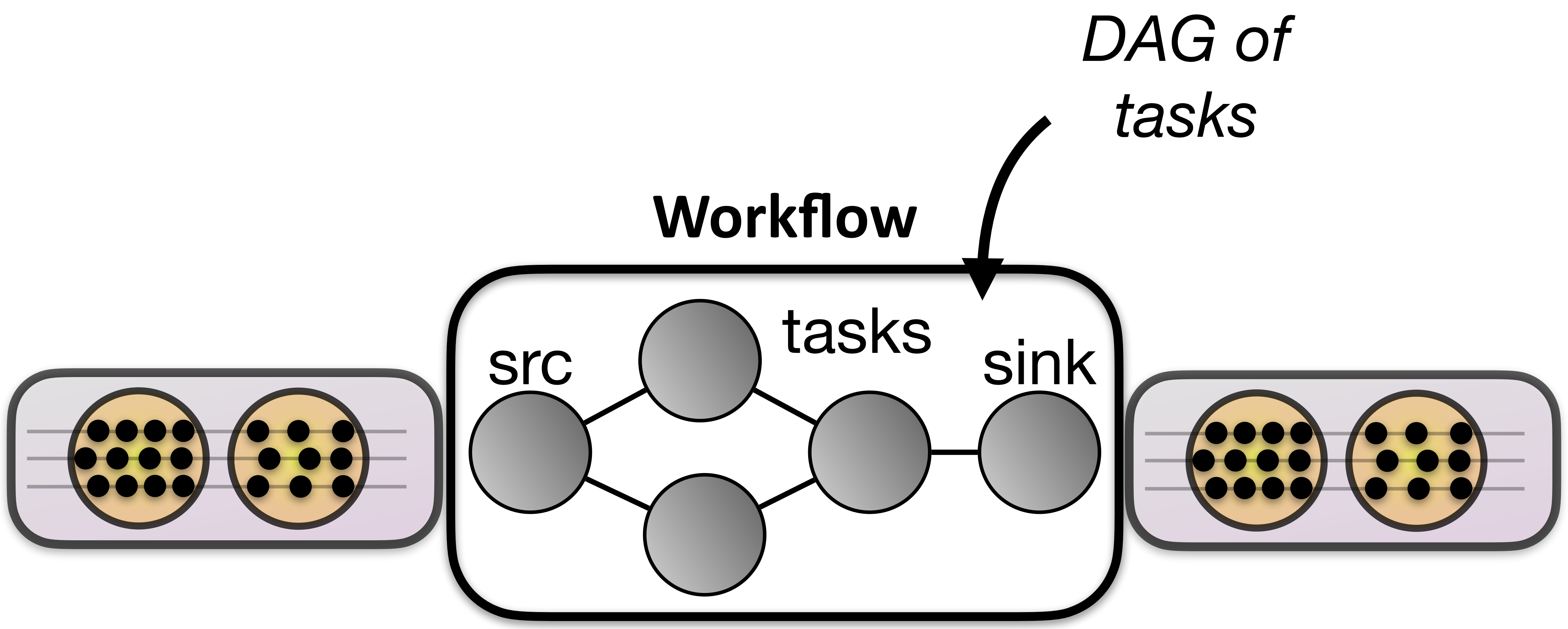
Sequencer: sequence atomic streams

Composite atomic stream: totally-ordered across lanes

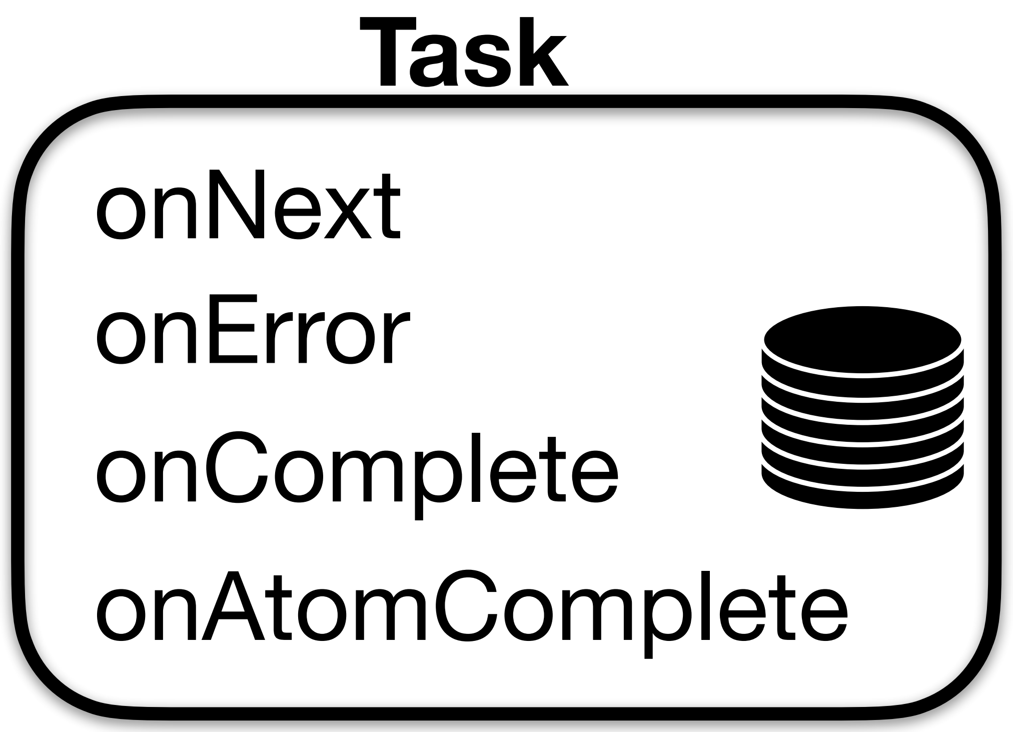
Lane

Splitter: split atomic streams

Workflows and Tasks

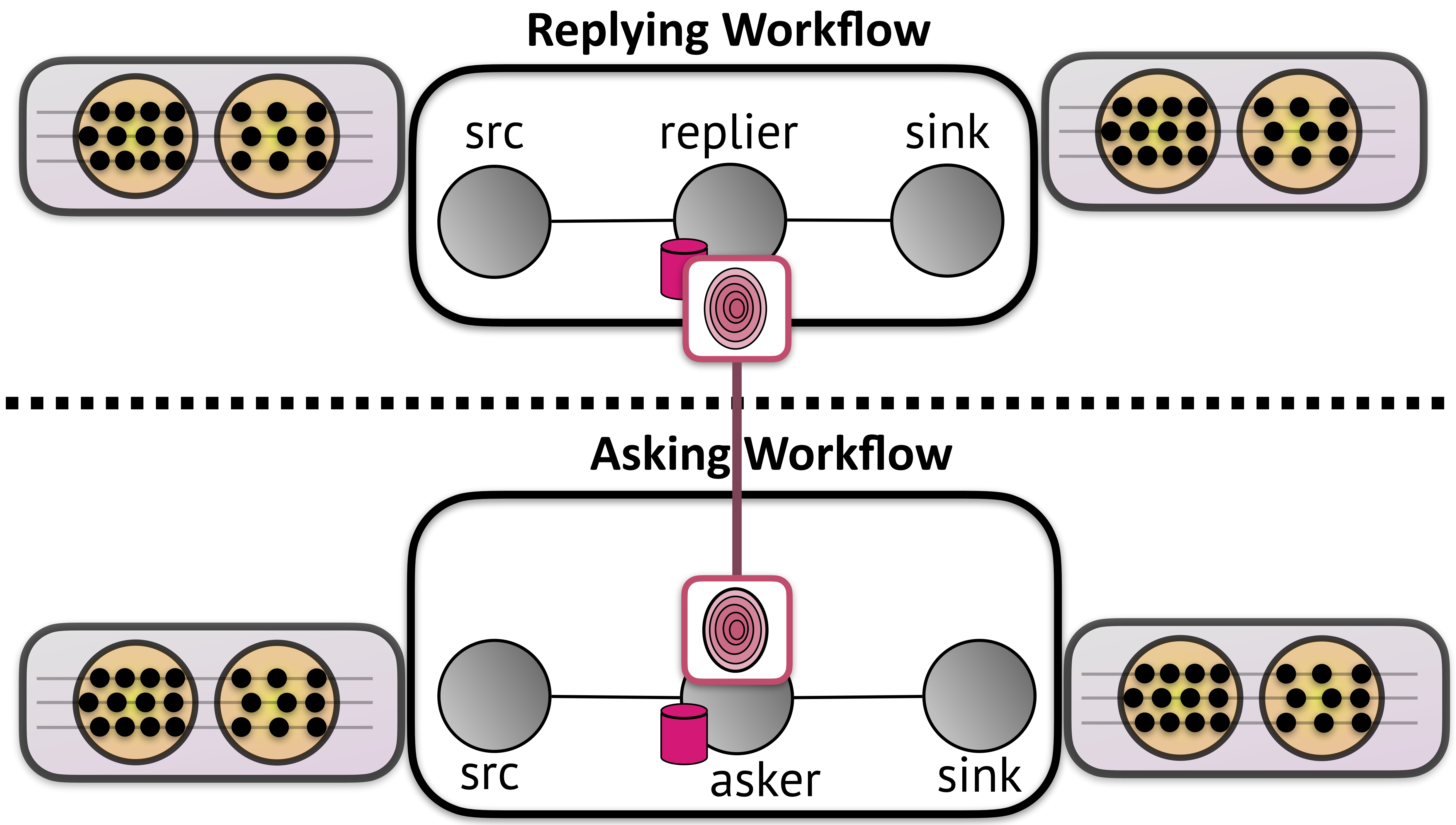


Workflow: consumes and produces atomic streams, represents a service; distributed, sharded over key-space



Task: stateful computational logic; can access state, emit events, etc.

Portals



Portal: request reply on streams, service portal

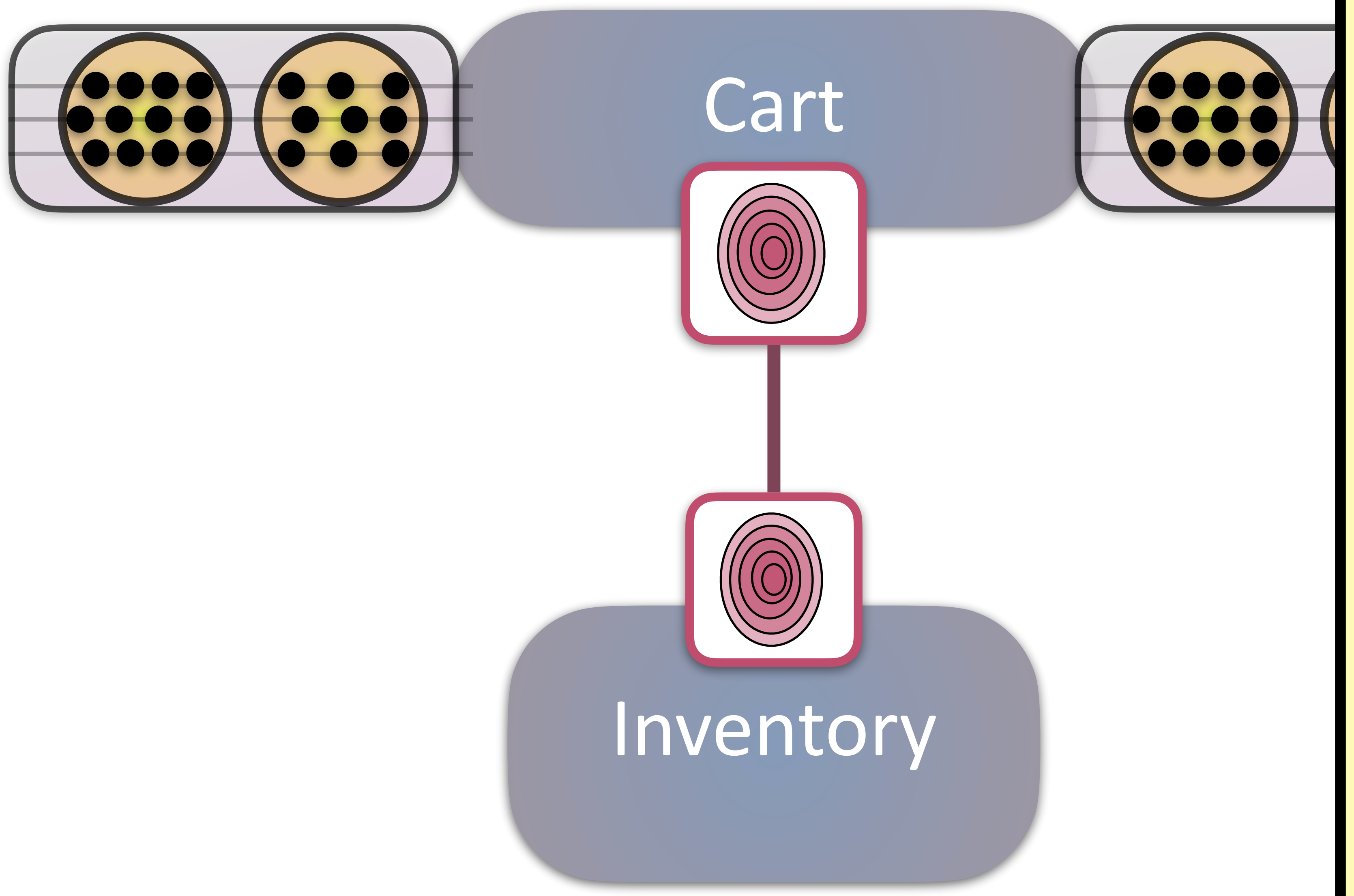
```
val portal = portals[Req, Rep]("portalName")

// Replying Workflow
...
.replier(portal)
  { /* handle events */ }
  { /* handle requests */ }
```

```
val portal = registry
  .portals.get[Req, Rep]]("portalName")

// Asking Workflow
...
.asker(portal) { event =>
  val request = ...
  val future = ask(portal)(request)
  await(future) { /* continuation */ }}
```

Portals

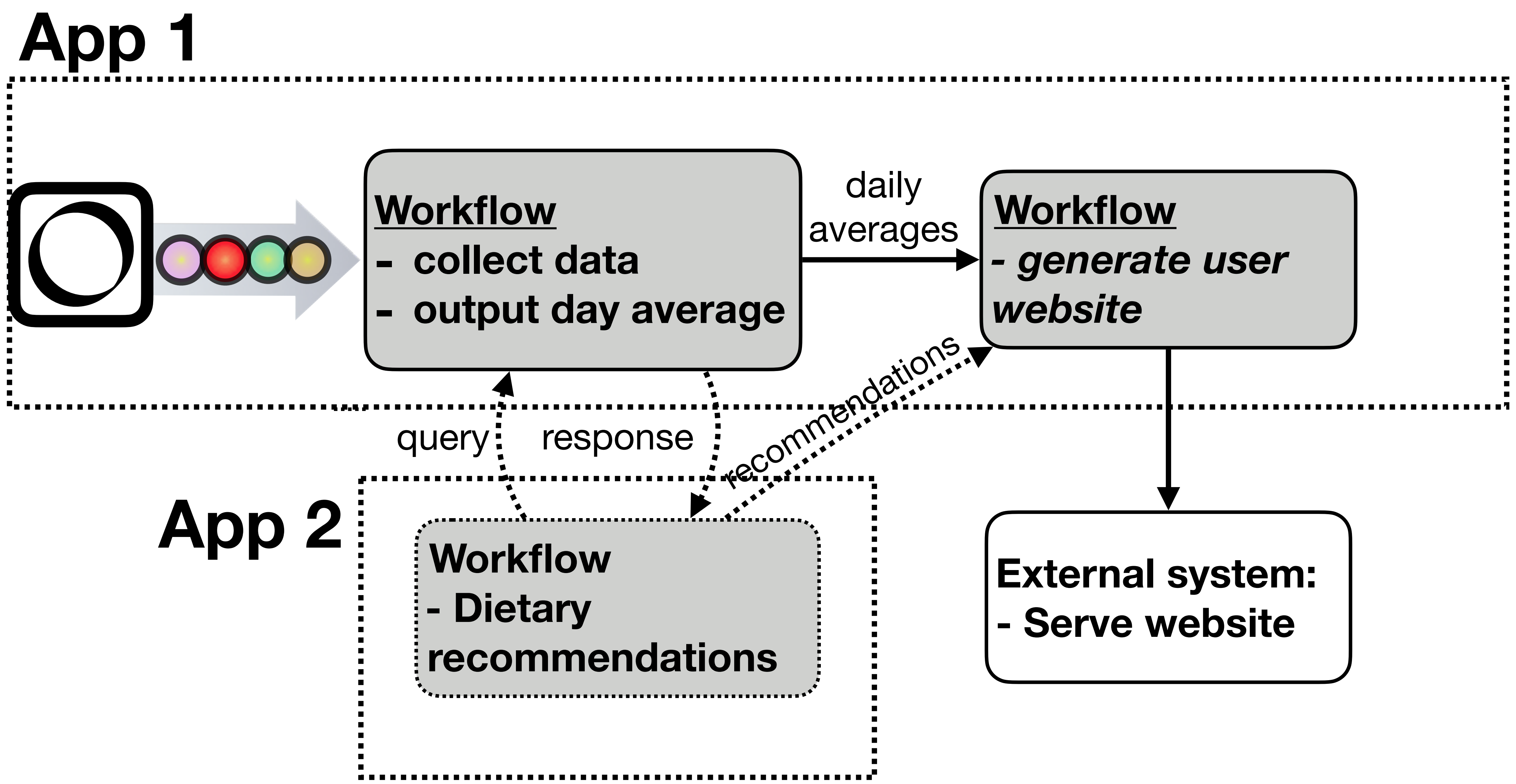


```
// Cart Workflow
val cart = Workflows[ClientReqs, Orders]()
   .source(clientStream)
   .asker(portal) {
      case AddToCart(item) =>
         val cartState = PerKeyState(Map.empty)
         val f = ask(portal)( GetItem(item) )
         Await(f) {
            f.value match
               case GetItemSuccess =>
                  cartState += item -> (cartState(item) + 1)
               case GetItemFail =>
                  () // do nothing
         }
      case ...
   }
   .sink()
   .freeze()
```

Portals

- Use Cases
  - Dynamically query the state of another workflow
  - Update, modify the state of another workflow
- Many workflows can connect / send queries to the same portal

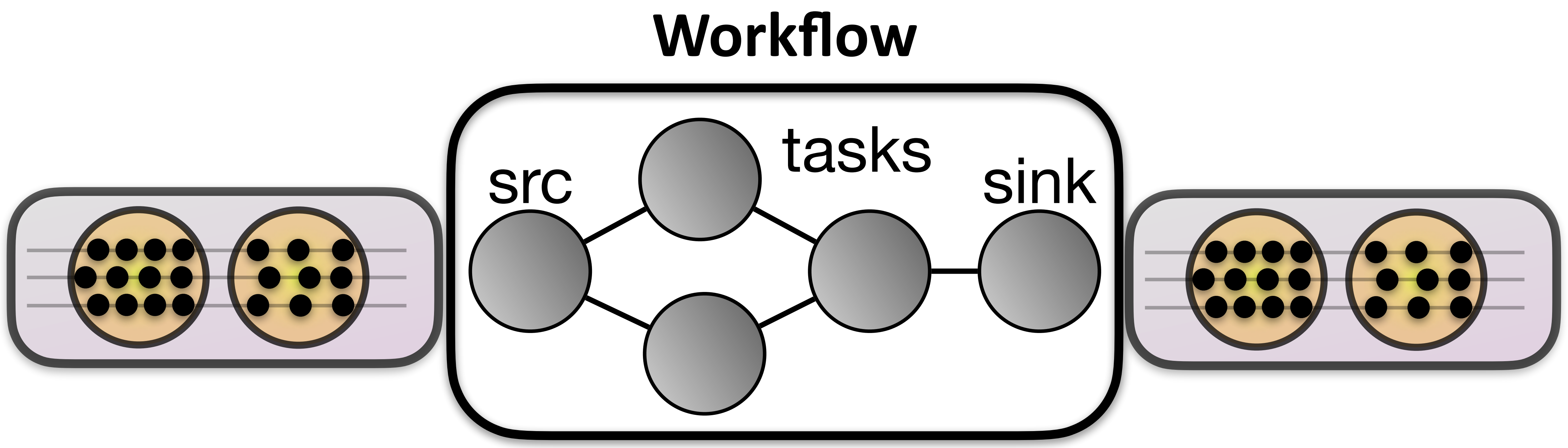
Applications, Registry



Application: Set of portals, workflows, streams, generators, etc. encapsulated as one application.

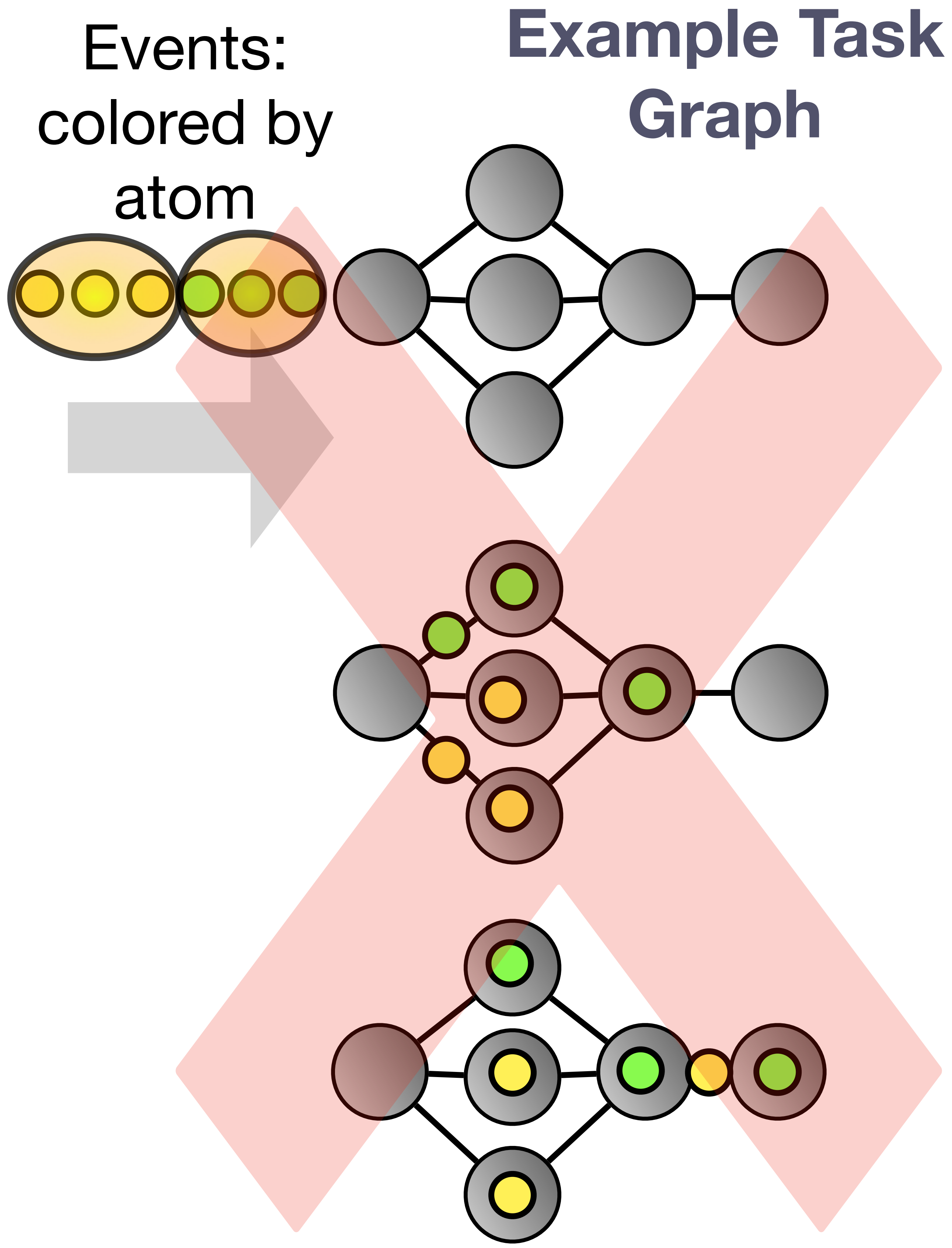
Registry: finding existing streams, portals, workflows, etc., from other apps, dependencies.

Atomic Processing



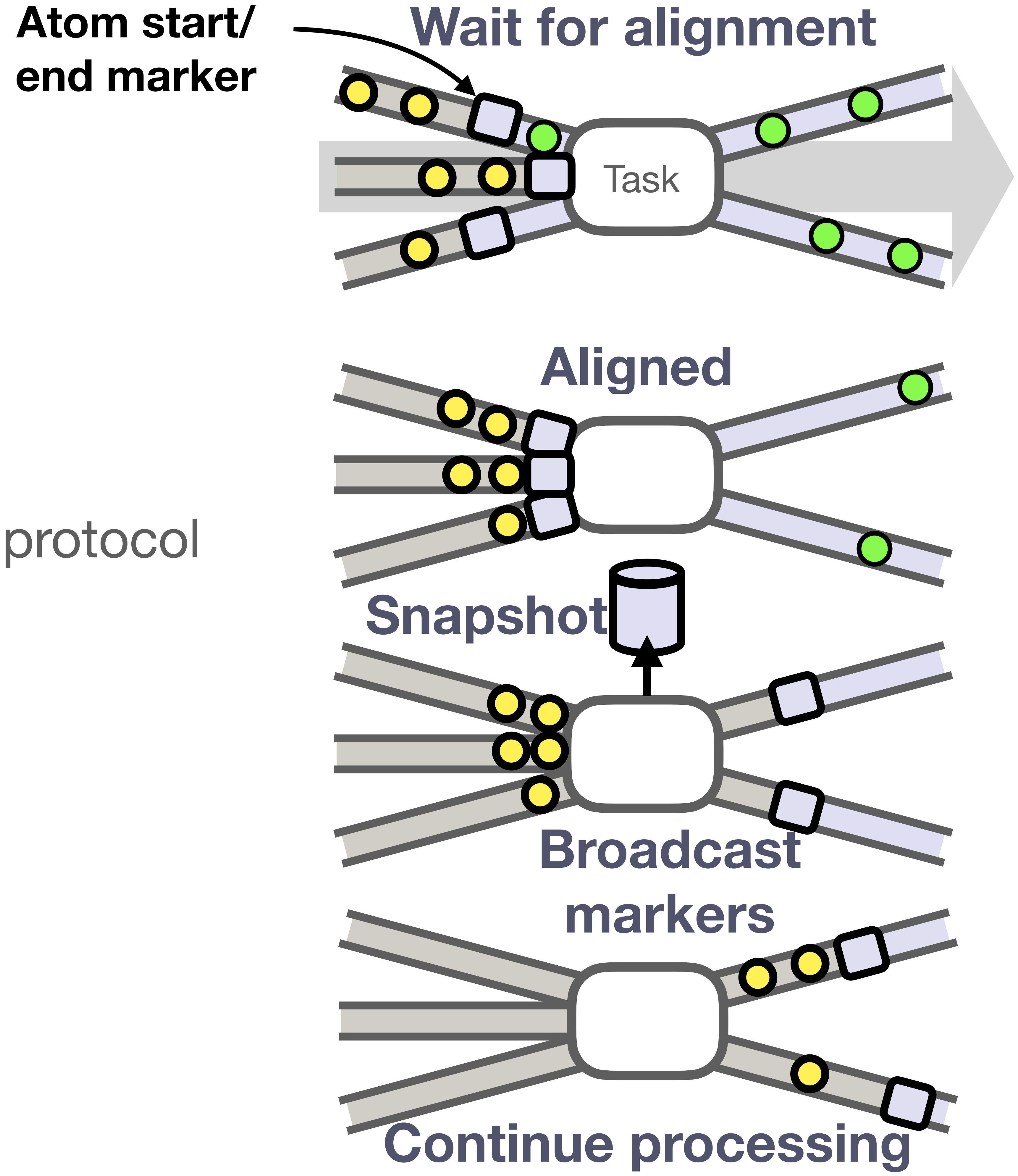
Atomic processing:
- Take atom
- Process atom until completion
- Commit to output
- Repeat

Alignment Protocol



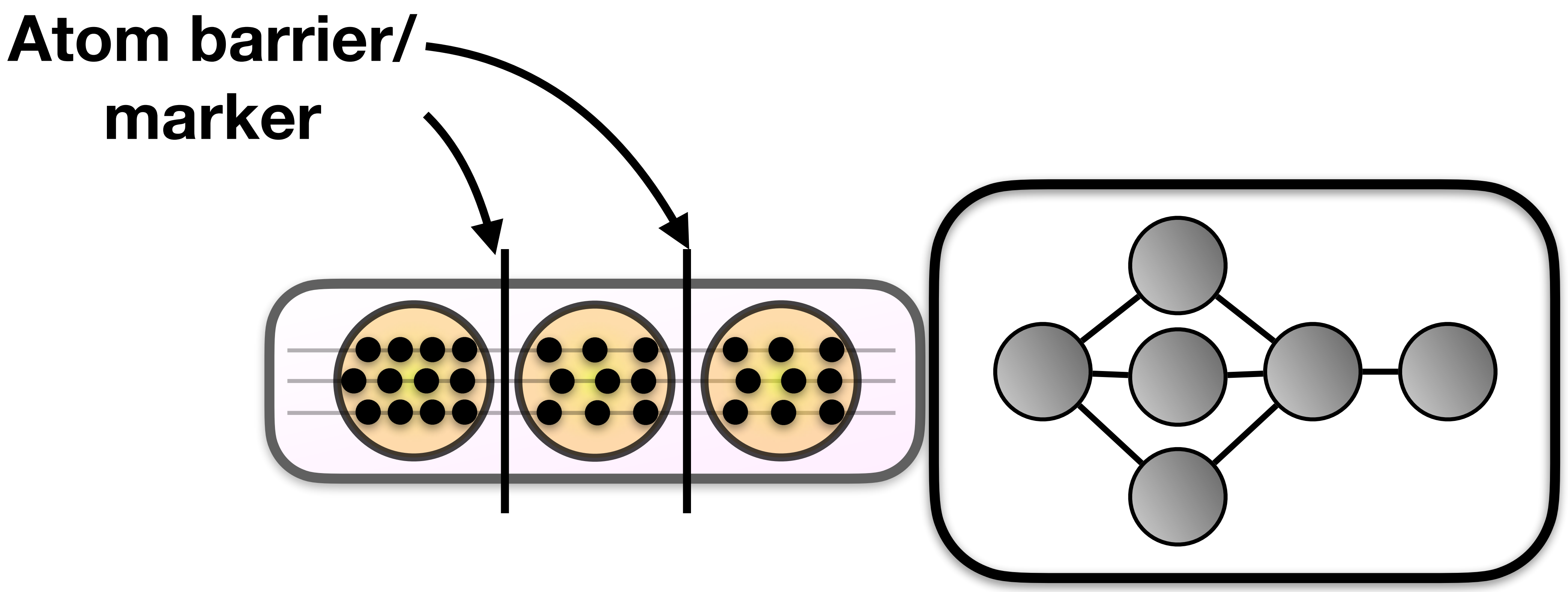
Problem: if we process two atoms, the events might reorder across atoms!

Solution: alignment protocol



Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. IEEE Data Eng. Bull. 38, 4 (2015), 28–38.

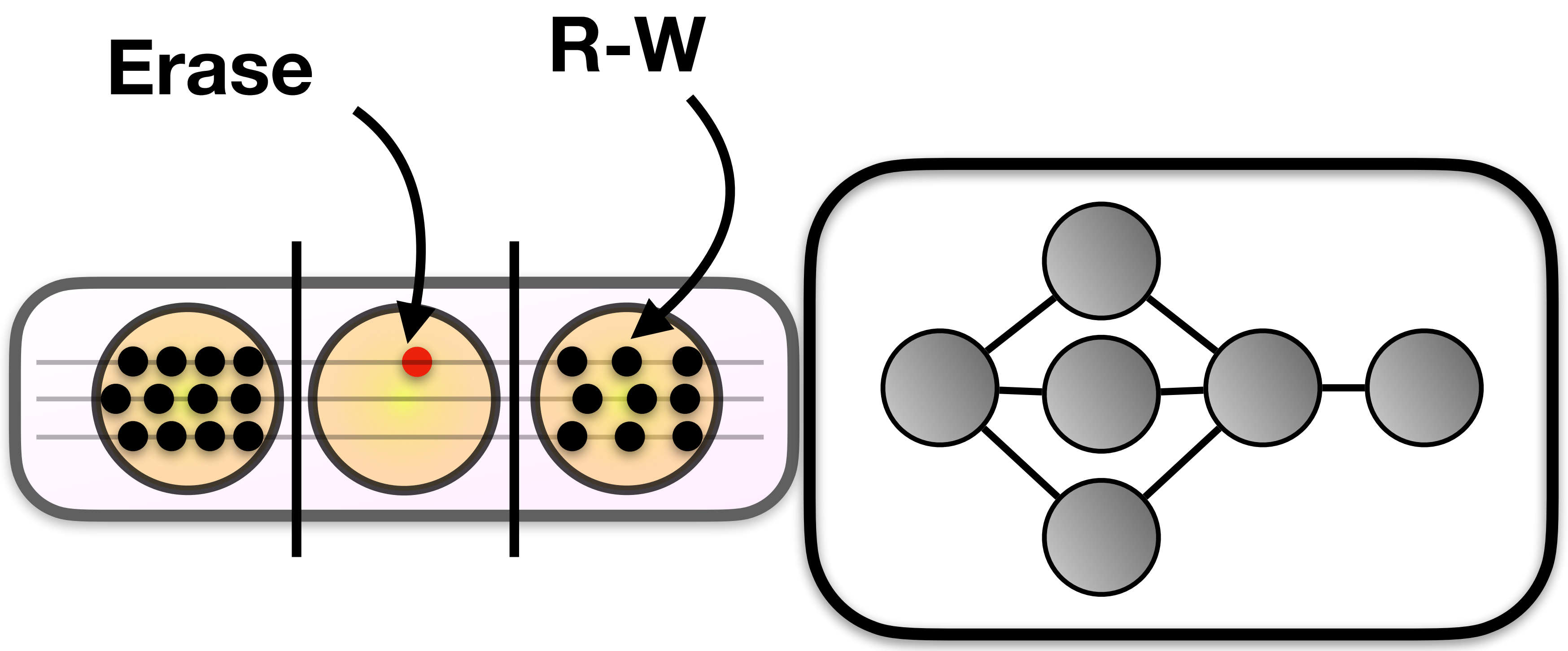
Event Ordering

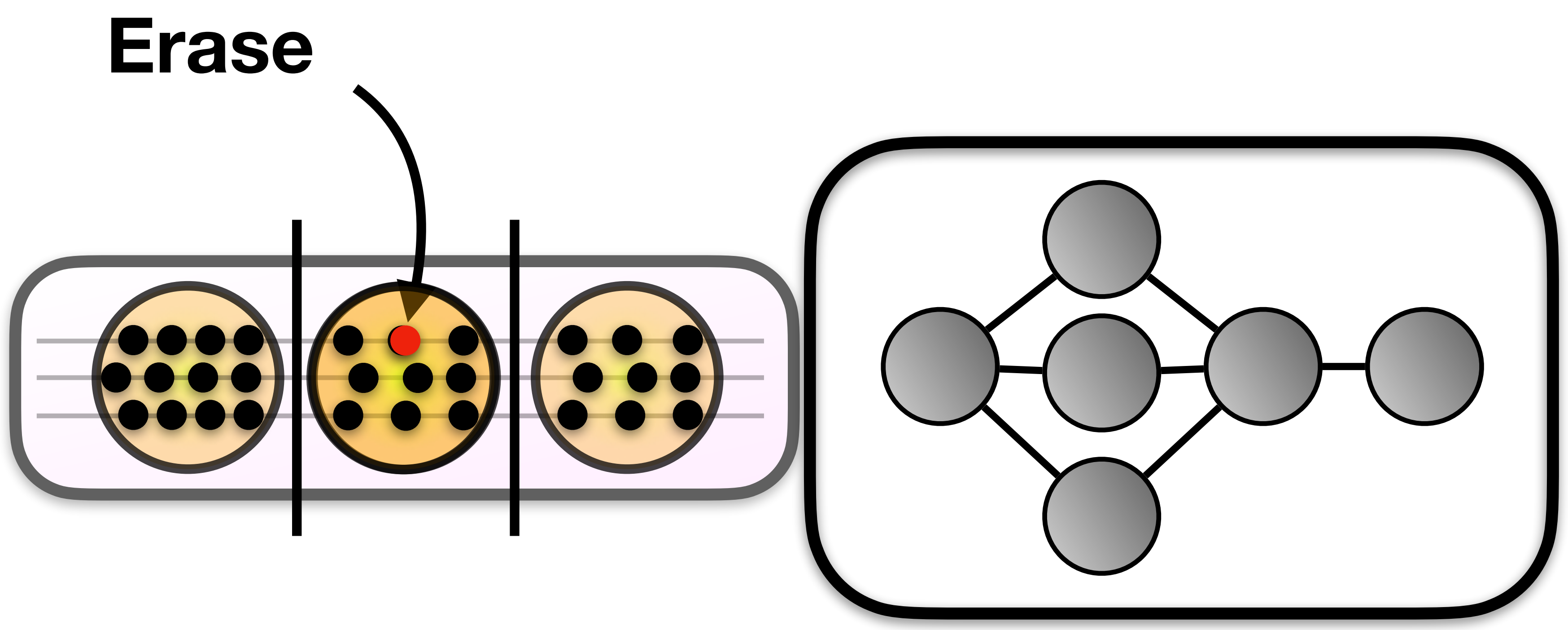


*The **onAtomComplete** event is triggered uniformly on all tasks by the passing atom barrier, this event is in a total order to all other events on the workflow*

- Atoms on Atomic Stream are totally-ordered (*event ordering guarantees**)
- => Events from two different atoms are in a strict order
- => Atom barriers / markers are totally-ordered

Event Ordering Examples



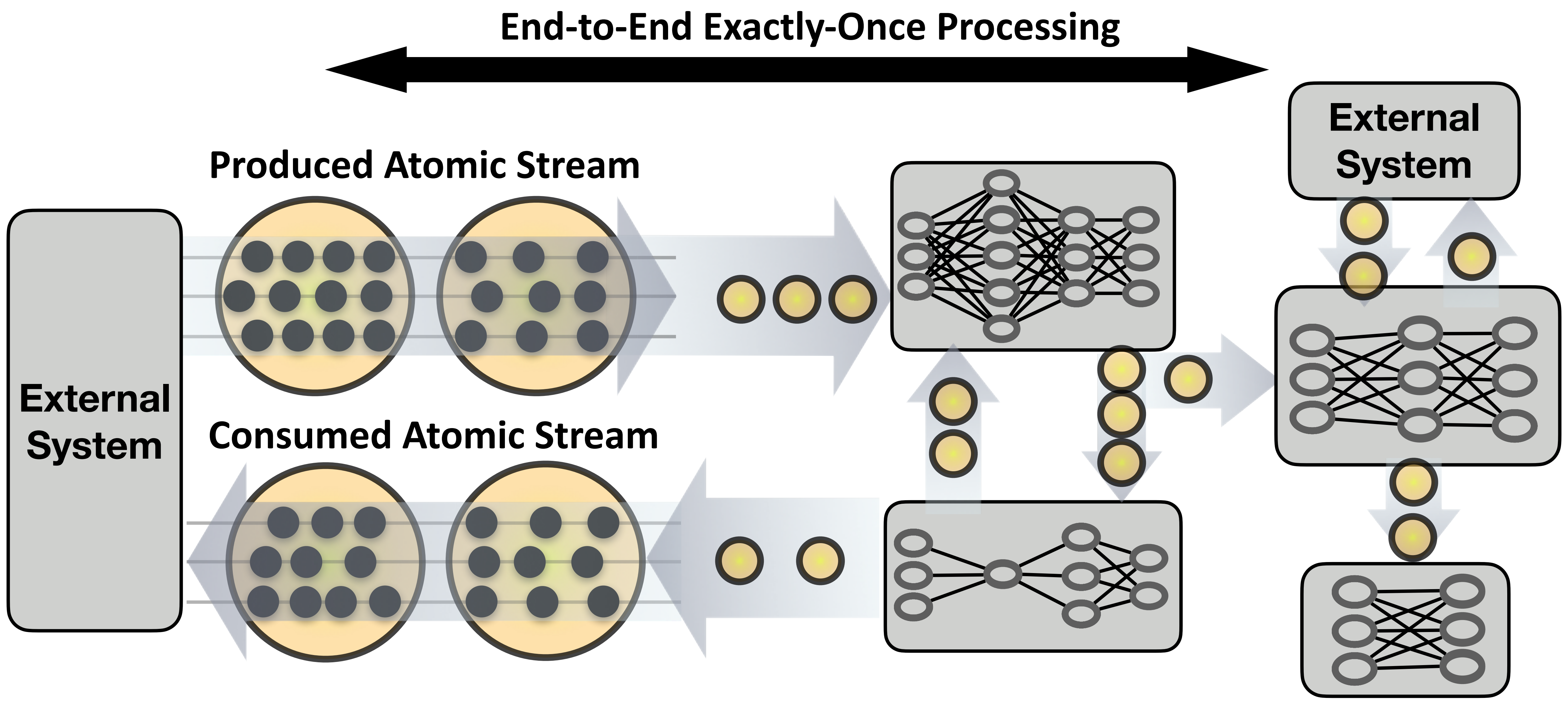


defer execution of erasure to the barrier

*The **onAtomComplete** event is triggered uniformly on all tasks by the passing atom barrier, this event is in a total order to all other events on the workflow*

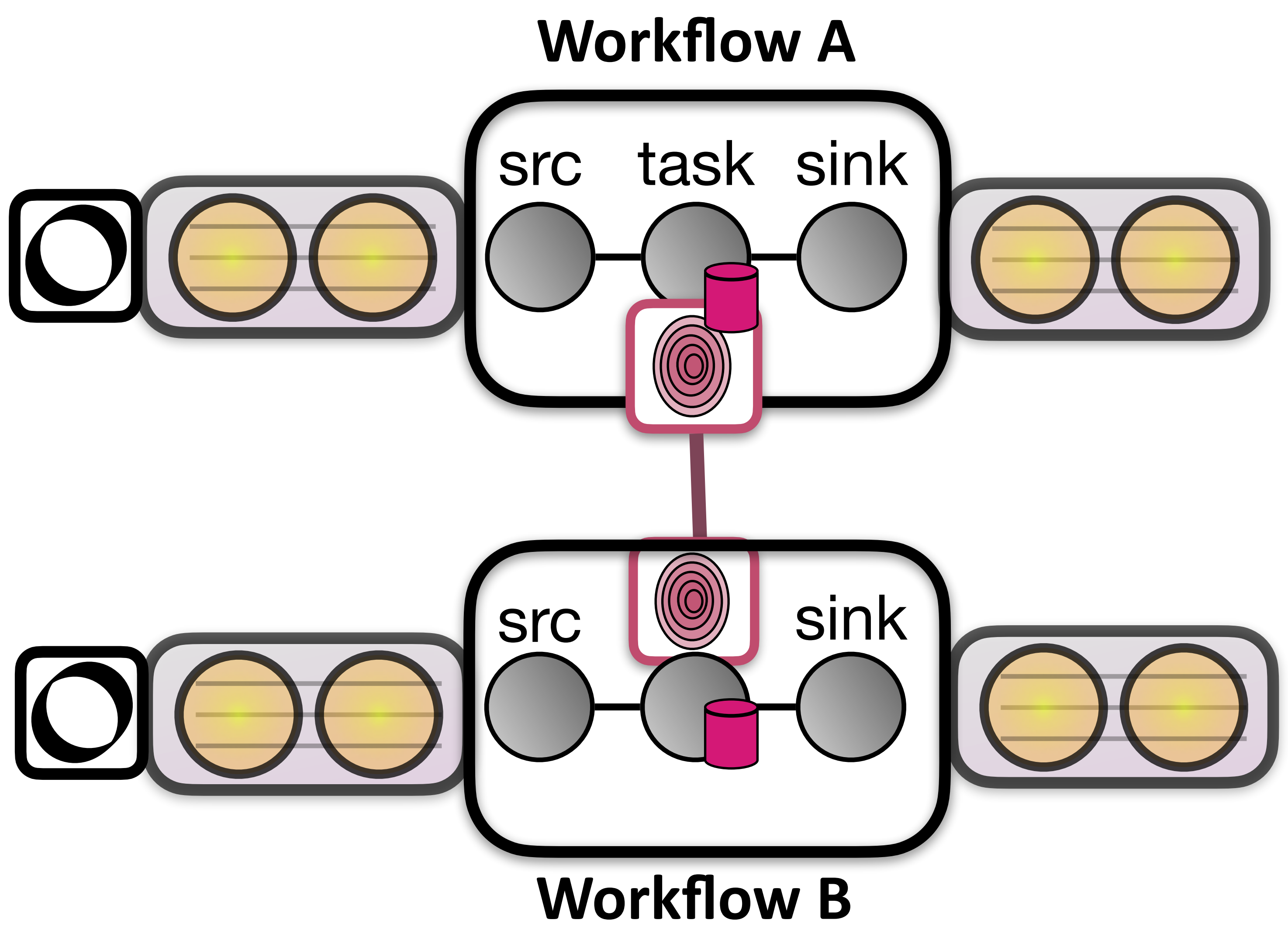
The Atomic Processing Contract

The Atomic Processing Contract: Atoms must be processed one-at-a-time, only committed & failure-free results may be observable/produced.

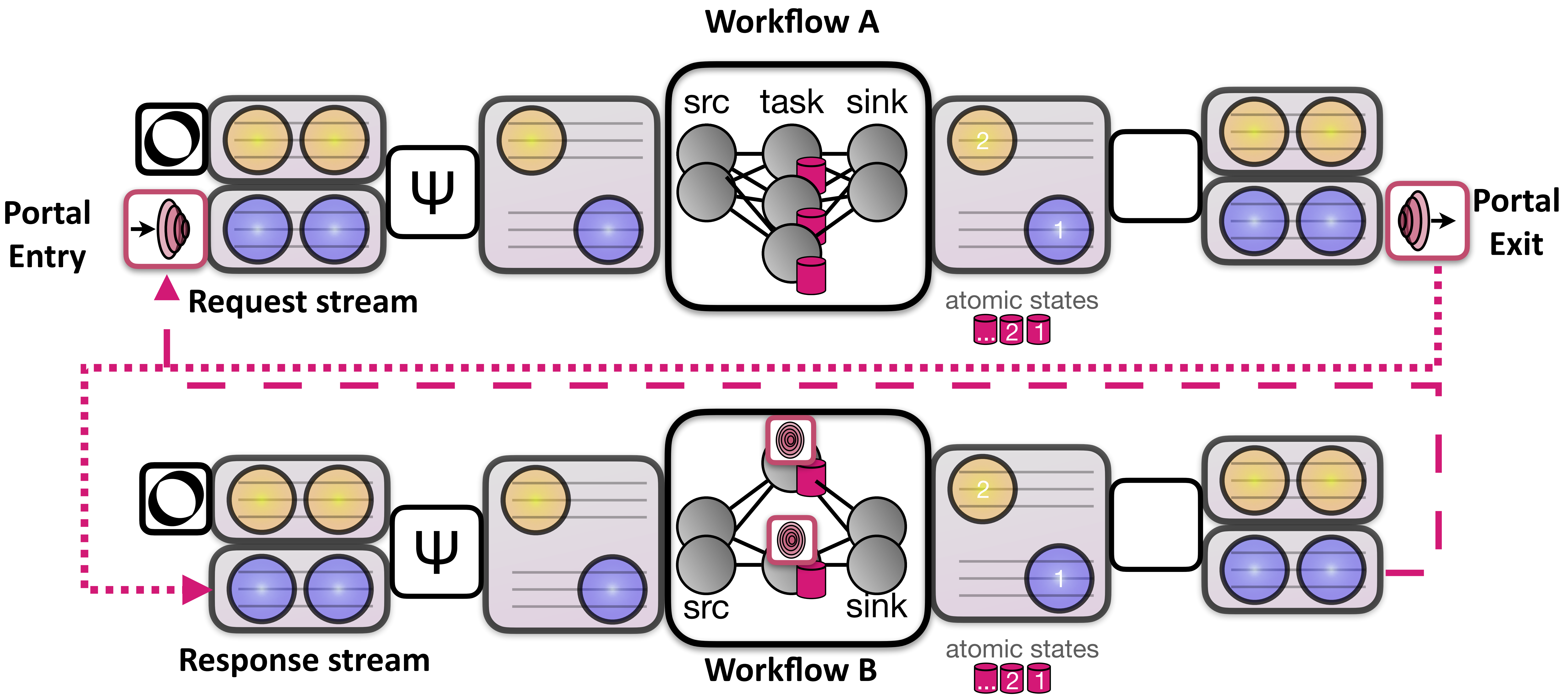


Logical View / Physical View

Logical



Physical



Events (black dots) omitted for clarity

3/3 Conclusion

Conclusion

- The **Portals programming model** extends dataflow streaming for stateful serverless applications:
  - **Dataflow streaming** provides exactly-once processing guarantees, performance, scalability
  - **Atomic streams** ensure end-to-end exactly-once processing guarantees, enable dynamic decentralized deployments, principled approach to cycles
  - **Portals** enable request/reply-style communication with futures, dynamic services

More in the Paper ...

- Programming model
- **Exactly-once processing mechanism**
- Prototype implementation in Scala 3
- Evaluation
- Use cases
- Related work

Portals: An Extension of Dataflow Streaming for Stateful Serverless

Jonas Spenger
Digital Futures
RISE Research Institutes of Sweden &
KTH Royal Institute of Technology
Stockholm, Sweden
jspenger@kth.se

Paris Carbone
Digital Futures
RISE Research Institutes of Sweden &
KTH Royal Institute of Technology
Stockholm, Sweden
parisc@kth.se

Philipp Haller
Digital Futures
KTH Royal Institute of Technology
Stockholm, Sweden
phaller@kth.se

Abstract

Portals is a serverless, distributed programming model that blends the exactly-once processing guarantees of stateful dataflow streaming frameworks with the message-driven compositionality of actor frameworks. Decentralized applications in Portals can be built dynamically, scale on demand, and always satisfy strict atomic processing guarantees that are natively embedded in the framework's principal elements of computation, known as atomic streams. In this paper, we describe the capabilities of Portals and demonstrate its use in supporting several popular existing distributed programming paradigms and use-cases. We further introduce all programming model invariants and the corresponding system methods used to satisfy them.

CCS Concepts: • **Software and its engineering** → **Distributed programming languages**; **Data flow languages**.

Keywords: dataflow streaming, stateful serverless, exactly-once processing.

ACM Reference Format:
Jonas Spenger, Paris Carbone, and Philipp Haller. 2022. Portals: An Extension of Dataflow Streaming for Stateful Serverless. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '22), December 8–10, 2022, Auckland, New Zealand.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3563835.3567664

1 Introduction

Decentralized stateful applications support most of the critical services in use today. This includes financial data transactions, transportation, e-commerce, healthcare, data monitoring systems as well as gaming and social networking services. Regardless of their importance, the programming frameworks we have at our disposal are ill-equipped for the complete, end-to-end job and often make compromises that are detrimental to either processing guarantees, scalability or programming flexibility. Thus, a great deal of mental effort is necessary to compose complex decentralized services with all guarantees and challenges in mind. Making them fault-tolerant, scalable, with arbitrarily complex and dynamic dependencies is a demanding multidisciplinary task that falls at the hands of the developer today. In this work, we investigate the potential of an all-encompassing solution to the problem of building and running decentralized stateful services that oversees the following challenges: I) processing guarantees (i.e., exactly-once transactional processing, live consistent updates), II) on-demand scalability and III) compositional, intuitive programming semantics.

Existing programming technologies in use today partially satisfy some, but not all, challenges behind decentralized applications. The most dominant being distributed actor frameworks [5, 9, 15, 25, 33, 41], serverless cloud programming services (e.g. Function as a Service - FaaS [4]) and dataflow streaming systems (e.g., Flink Streaming [12], Kafka Streams [51], etc.). Actor frameworks such as Akka [33] offer great flexibility in manually composing and scaling services through direct actor communication and passing of actor references. However, despite their ease of distributed programming, actors do not offer any guarantees for stateful processing, such as transactions and exactly-once processing. Similarly, serverless programming services such as AWS Lambda [4] were designed with simplicity of use and data-driven scalability in mind, yet, they collectively lack stateful processing semantics and guarantees.

On the other end of the spectrum, we are witnessing an increasing number of applications and services developed on top of dataflow streaming frameworks [3, 12, 42]. Dataflow streaming systems gained popularity during the last decade, and have met high adoption due to their exceptionally strong reliability guarantees (challenge I). In the dataflow streaming setting the dependencies between computational tasks are explicit and this is therefore a trivial task. At the same time, dataflow tasks can be executed in a parallel fashion over sharded state using consistent hashing (challenge II). These attributes make dataflow streaming systems a convenient platform to write applications, at the expense of serious

Onward! '22, December 8–10, 2022, Auckland, New Zealand
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9909-8/22/12.
https://doi.org/10.1145/3563835.3567664

Future Work

- **Implementation:** *distributed, decentralized, reduce overhead*
- **Portals formalization + proofs**
- **Further extensions:** *dynamically splitting atoms; actor-like references; optimistic execution; transactions*

Thanks!

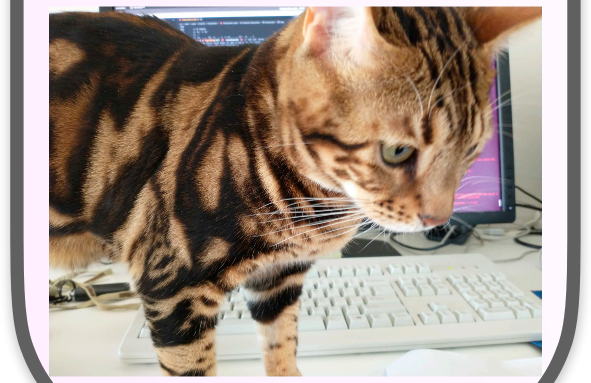
www.portals-project.org

This work was partially funded by the Swedish Foundation for Strategic Research (SSF grant no. BD15-0006), by Digital Futures, and by RISE AI.

Jonas Spenger
jspenger@kth.se
(KTH, RISE)

Paris Carbone
(KTH, RISE)

Philipp Haller
(KTH)



House Cat

Key takeaways:

- The **Portals programming model** extends dataflow streaming for stateful serverless applications:
  - **Atomic streams** ensure end-to-end exactly-once processing guarantees, enable dynamic decentralized deployments, principled approach to cycles
  - **Portals** enable request/reply-style communication with futures, dynamic services