Scala Days

Crossing the Boundaries of Stateful Streaming and Actors using Serverless Portals

Jonas Spenger
RISE Research Institutes of Sweden
KTH Royal Institute of Technology

Part 1: Stateful Serverless
Part 2: Stateful Streaming and Actors
Part 3: The Portals Framework

Stateful Serverless

- **Serverless** simplifies building cloud applications
  - FaaS: *Stateless* Functions and Triggers
  - Serverless frameworks **fully manage the function execution**
- Challenges with traditional FaaS:
  - Functions are stateless, functions cannot call other functions
  - Consistency is the applications responsibility
- **Recent development: Stateful Serverless**
  - **Fully manages compute, state, messaging**
  - Consistency is the frameworks responsibility
  - Challenge: ensure end-to-end consistency in spite of failures
- Desirable properties
  - **Strong execution guarantees**
    - Exactly-once processing guarantees
  - **Good performance**
    - High-throughput, low-latency
  - **Expressive** enough for intended applications

Execution Guarantees - Message Processing, 3 Ways

Execution guarantees provided by message processing frameworks:

Exactly-once processing

```
A:
  send x to B

B:
  on receive x do
    state = state + x
```

A message is consumed, processed, and side-effecting **exactly-once**

Or, **processing a message is a transactional step** in which: 1) the message is consumed; 2) processed; 3) and any of its side-effects produced/published.

Stateful Serverless

At-least-once processing

```
A:
  send x to B

B:
  on receive x do
    transaction:
      if !rcvdMsgs.contains(x) then
        rcvdMsgs.add(x)
        state = state + x
```

A message is consumed and processed at least once

(Stateless) Serverless

At-most-once processing

```
A:
  repeat
    send x to B
  until receive `Ack` from B

B:
  on receive x do
    transaction:
      resp `Ack`
      if !rcvdMsgs.contains(x) then
        rctxMsgs.add(x)
        state = state + x
```

A message is consumed and processed at most once

Actor Frameworks

Execution Guarantees - Message Processing, 3 Ways

- Programs in exactly-once processing frameworks **contain solely application logic**
- Other execution models require extensive failure-handling logic
  - **=> Likely to introduce bugs**
- **End-to-end exactly-once processing make programs significantly easier to write and reason about**

Part 1: Stateful Serverless

Part 2: Stateful Streaming and Actors

Part 3: The Portals Framework

Stateful Stream Processing

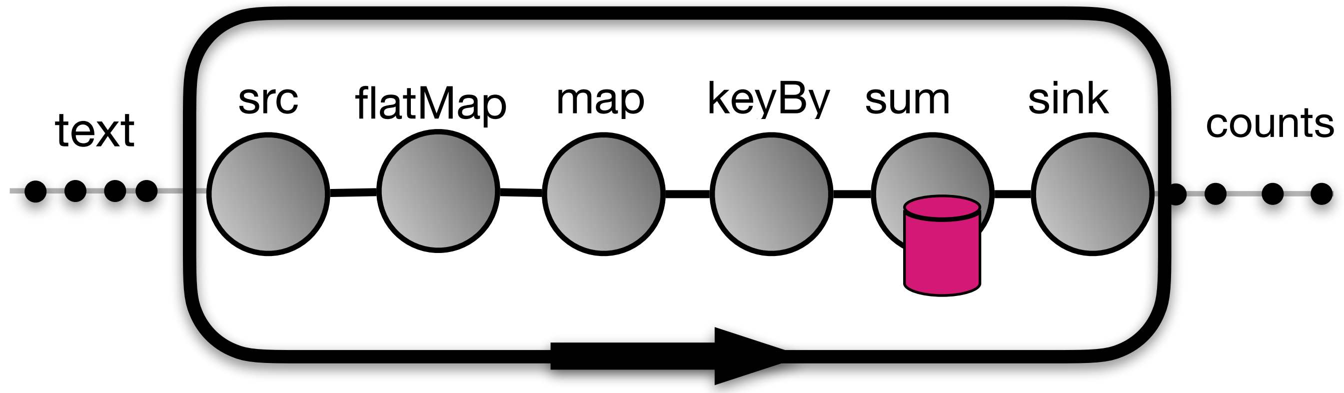
WordCount

```
val text: DataStream[String] = ...

val counts = text
  .flatMap { w => w.split("\\s") }
  .map { w => (w, 1) }
  .keyBy { x => x._1 }
  .sum { x => x._2 }
```

Flink

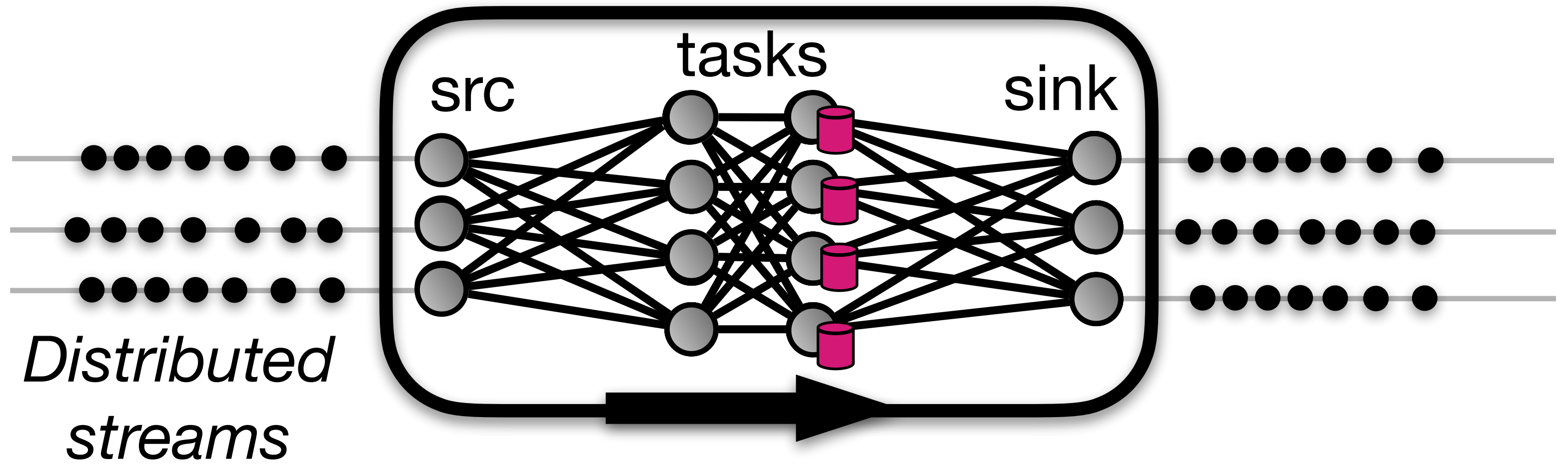
1. Program written in streaming API

WordCount Pipeline



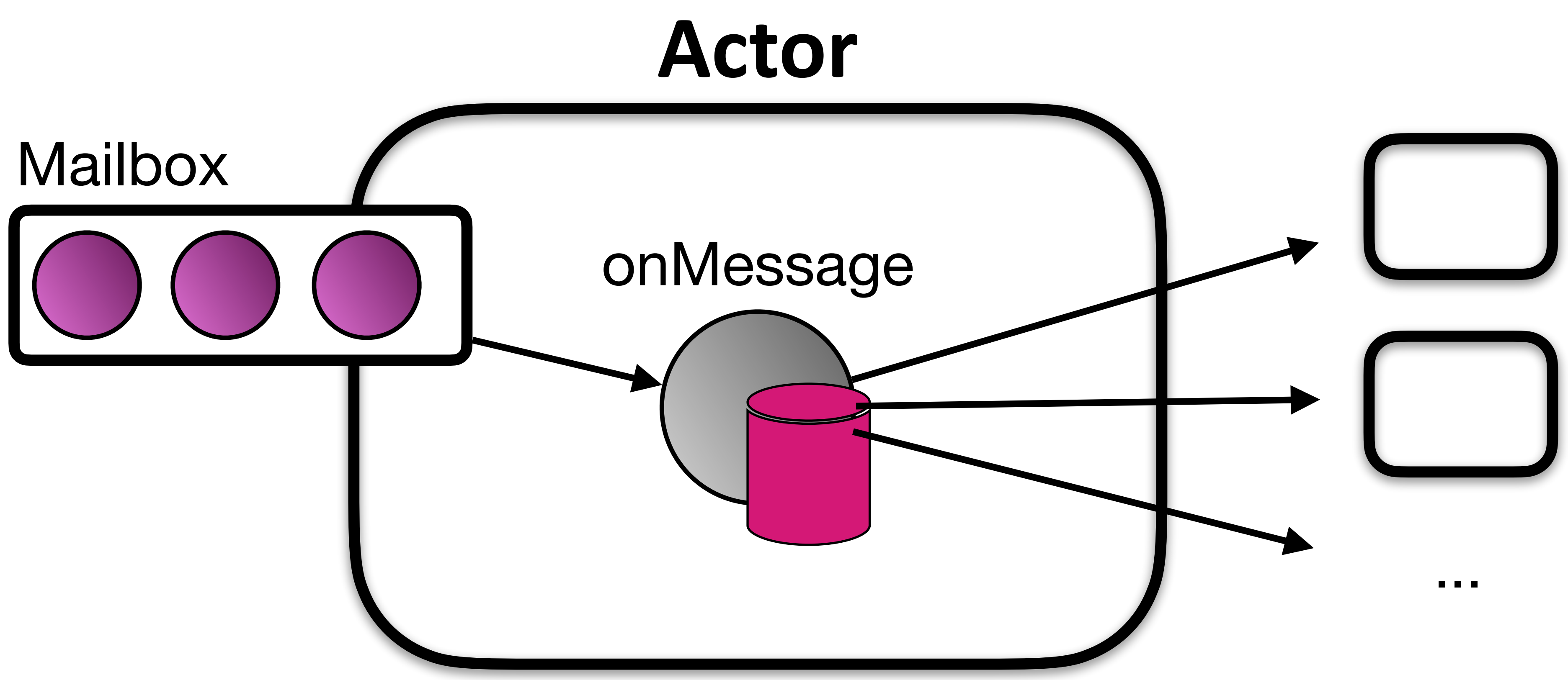
2. Logical representation, acyclic graph of stateful tasks

Physical WordCount Pipeline



3. Physical representation, optimizations

The Actor Model



- Actors can
  - **Send messages to other actors**
    - Connect to new actors through exchanging actor references
  - **Create new actors**
  - **Modify local state**

Comparison of Stateful Streaming and Actors

Stateful Streaming Systems

+ High-throughput, low-latency, suitable for real-time, (data-parallelism, pipeline-parallelism)

- Limited expressiveness to static acyclic graphs of tasks
  - No request/reply interaction with a stream pipeline, nor with a pipeline tasks.
  - Not dynamic, no cycles

+ Exactly-once processing guarantees
  - Illusion of failure-free execution

Actor Systems

+ Low-latency, low-overhead, real-time (task-parallelism)

+ Very expressive, can express general concurrent computations
  - However, this comes with concurrency problems such as deadlocks, livelocks

- No exactly-once processing guarantees
  - Low-level, used to implement fault-tolerant services manually

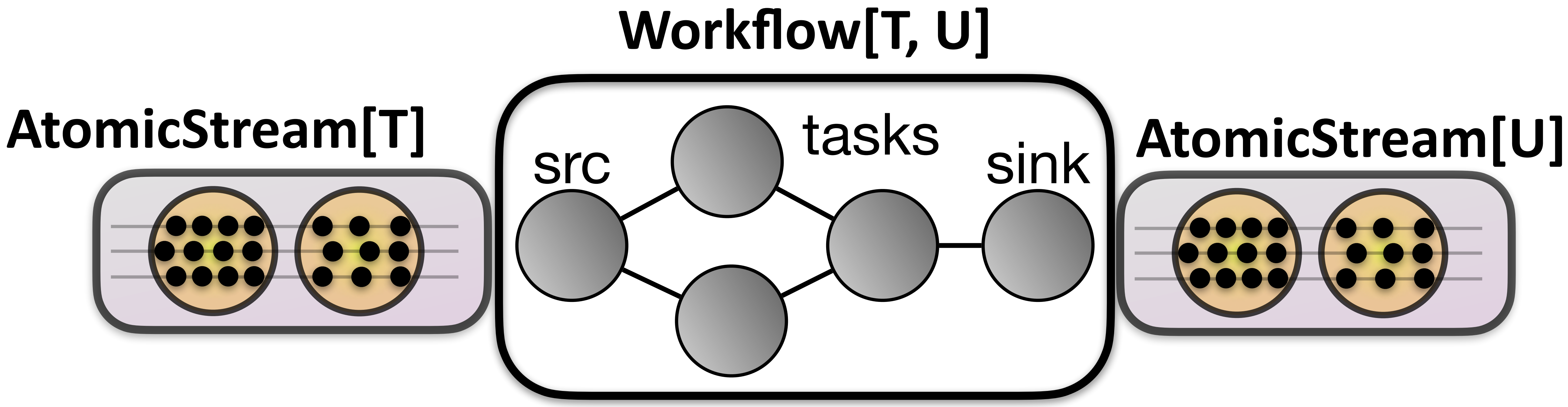
Part 1: Stateful Serverless
Part 2: Stateful Streaming and Actors
Part 3: The Portals Framework

The Portals Programming Model

- **Workflows**
  - Stream processing pipelines
- **Atomic streams**
  - Transactional streams, compose workflows together
- **Portals**
  - Actor-like communication, request/reply messaging
- End-to-end exactly-once processing

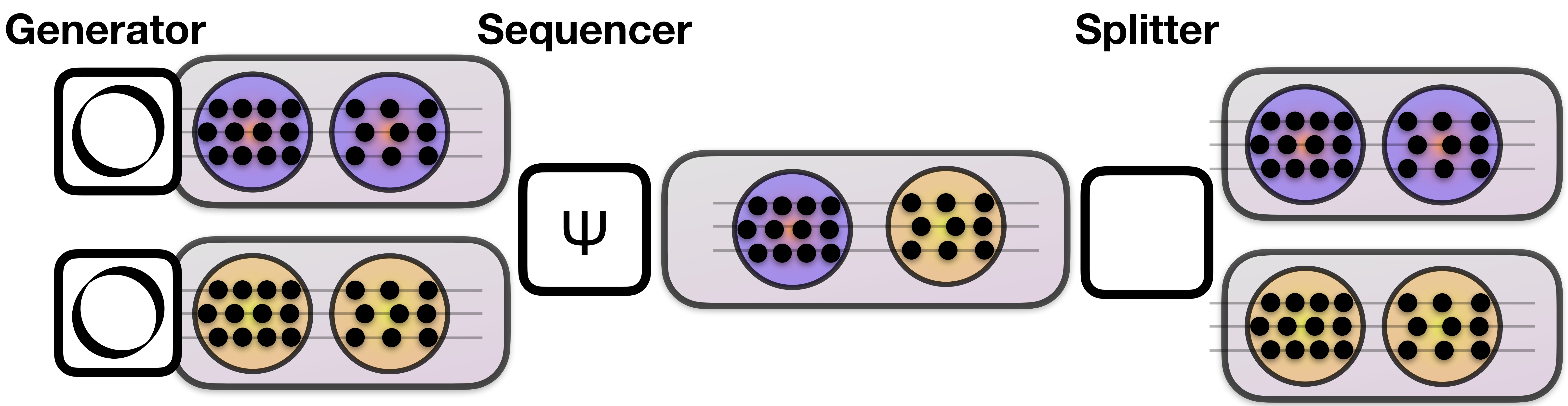
Workflows

- Consume and produce atomic streams
- DAG of stateful tasks



Atomic Streams

- **Transactional distributed streams:**
  - Transport atoms (batch of events)
  - Atoms are totally ordered on a stream
  - Connect workflows



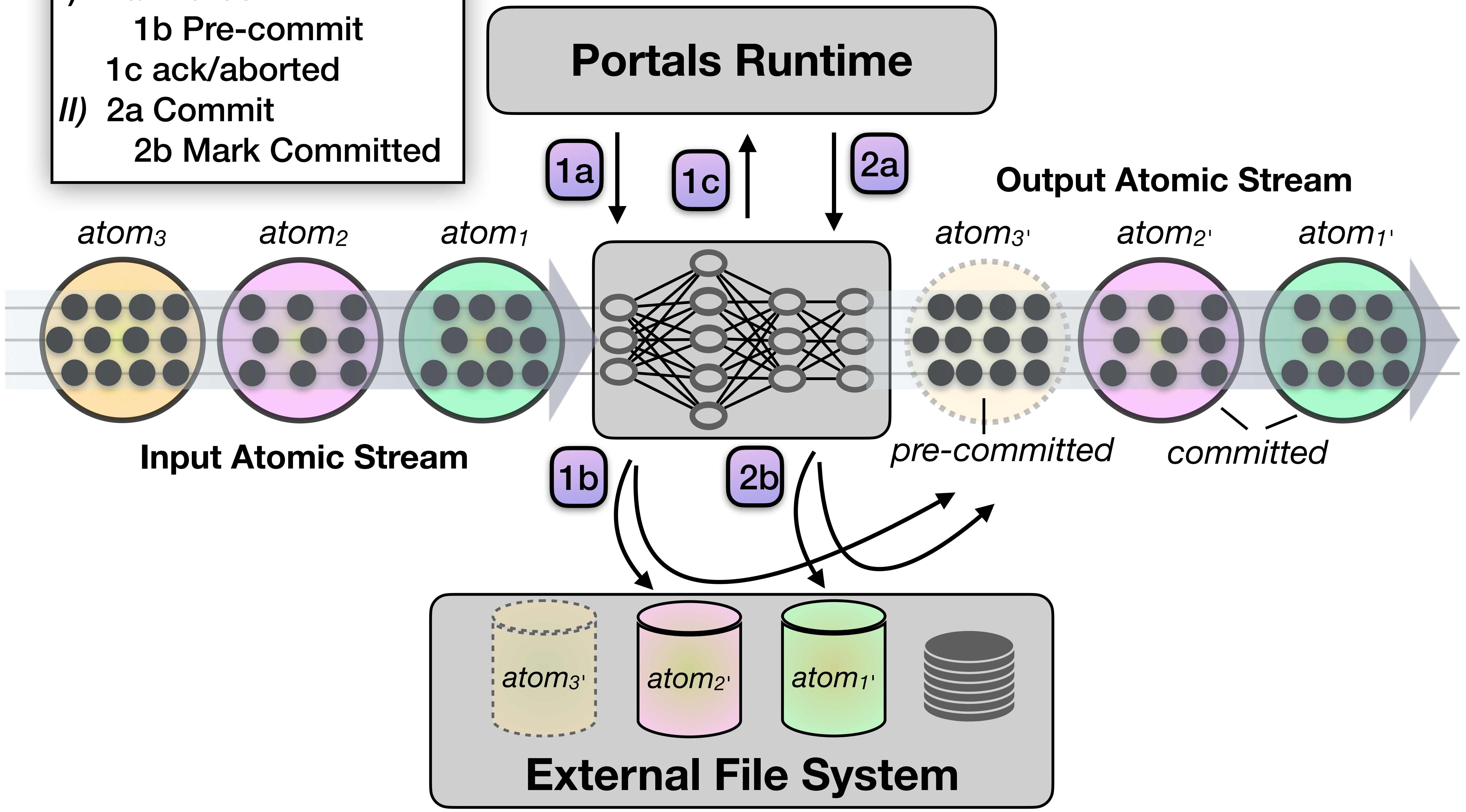
Atomic Processing

Atom Commit Protocol
I) 1a Pre-commit
1b Pre-commit
1c ack/aborted
II) 2a Commit
2b Mark Committed

Atomic Processing Contract:

Processing through atomic (transactional) steps:

- Consume an atom ("batch of events")
- Processes the whole atom
- Produce the side-effects (new events, state updates)



In general, implemented via rollback-recovery techniques and 2PC*

**E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. 2002. A survey of rollback-recovery protocols in message-passing systems. ACM Comput. Surv. 34, 3 (September 2002), 375–408. https://doi.org/10.1145/568522.568525*

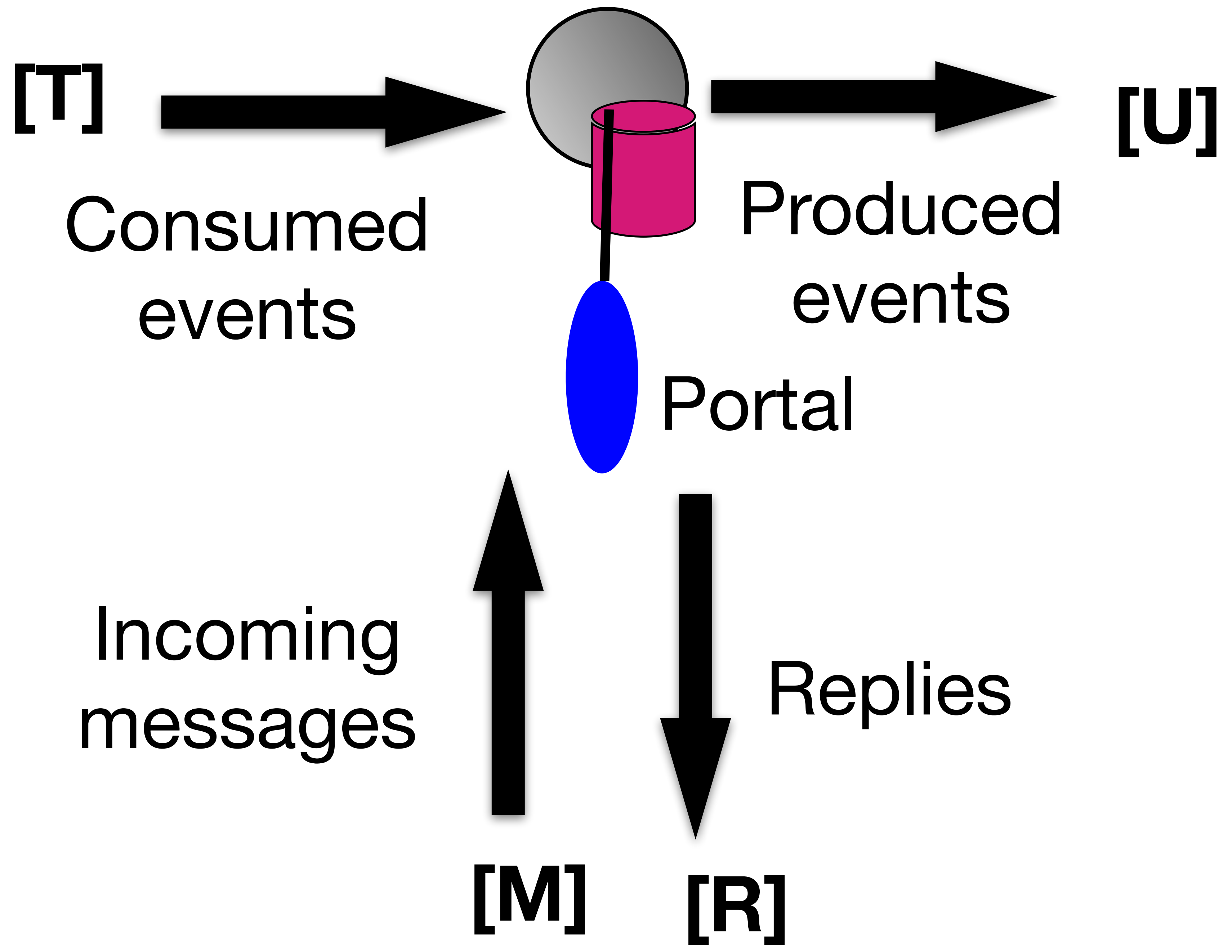
See also: Spenger, Jonas, Paris Carbone, and Philipp Haller. "Portals: An extension of dataflow streaming for stateful serverless.", 2022, ONWARD'22.

New Concept: *Portals*

- ***Portals* enable actor-like communication**
- **Communication restrictions**
  - **1) Connections are statically defined**, dynamically reconfigurable
    - Actor-refs can only be used if connection was defined
    - Connections are uni-directional
  - **2) No dynamic *creation* of workflows, tasks**
  - **1, 2 imply static topology**
- **Messages can be replied to**
  - Replier does not need a reference to requester, limits no. connections

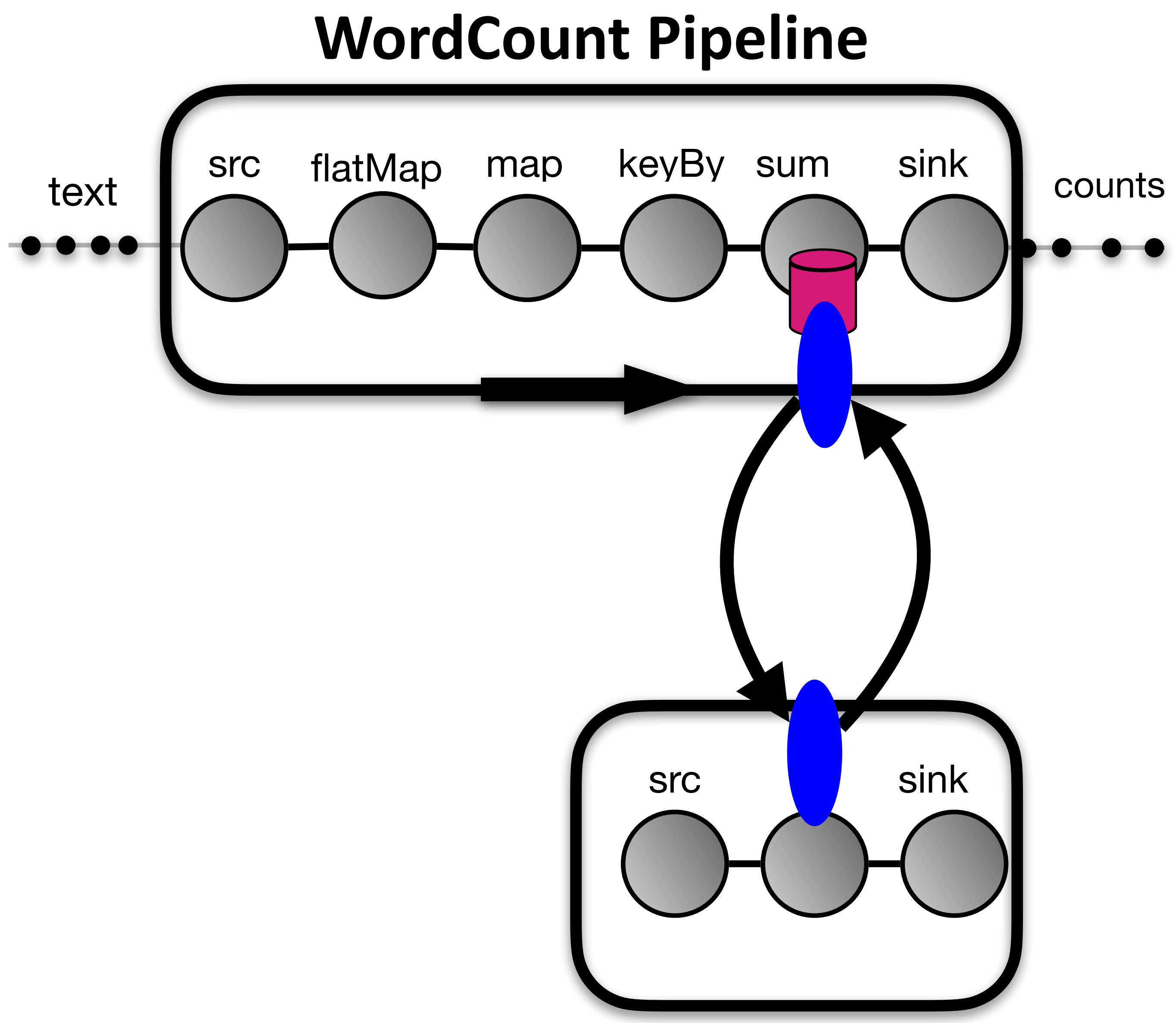
Tasks with Portals

- **PortalTask[T, U, M, R]** as a task/actor hybrid
- **T, U** are stream input/output types
- **M, R** message and reply type
- **Portal[M, R]** as a named mailbox
- Tasks statically connect to Portals as senders or receivers
    - Every Portal has exactly one receiving task



Word Count Portal Example

- Sum task connects to portal as receiver
- Replies to requests (*words*) with *count*
- Other task connects to portal, sends *requests*



Word Count Portal Example

Responding Task

```
...
  val portal = Portal[String, (String, Int)]("wordcount")
  ...
    .taskWithReplier(portal)(...): msg =>
      val state = PerKeyState[Int]("count").withDefault(0)
      val wordCount = (msg, state.get())
      reply(wordCount)
```

Requesting Task

```
...
  val portalRef = Registry.portals[String, (String, Int)]("/
  WordCount/portals/wordcount")
  ...
    .taskWithRequester(portalRef): event =>
      ...
      val request = word
      val future = ask(portalRef)(request)
      future.onComplete:
        case Success((word, count)) =>
          ...
          emit((word, count))
```

Continuations in Portals

- On invocation of onComplete
- Store continuation and metadata to task's persistent storage
  - Safety with Spores3 library https://github.com/phaller/spores3
- When reply arrives
  - Load continuation, restore context from metadata, execute
  - Execution serialized with other events
- **This ensures that continuations are persistent and not ephemeral**

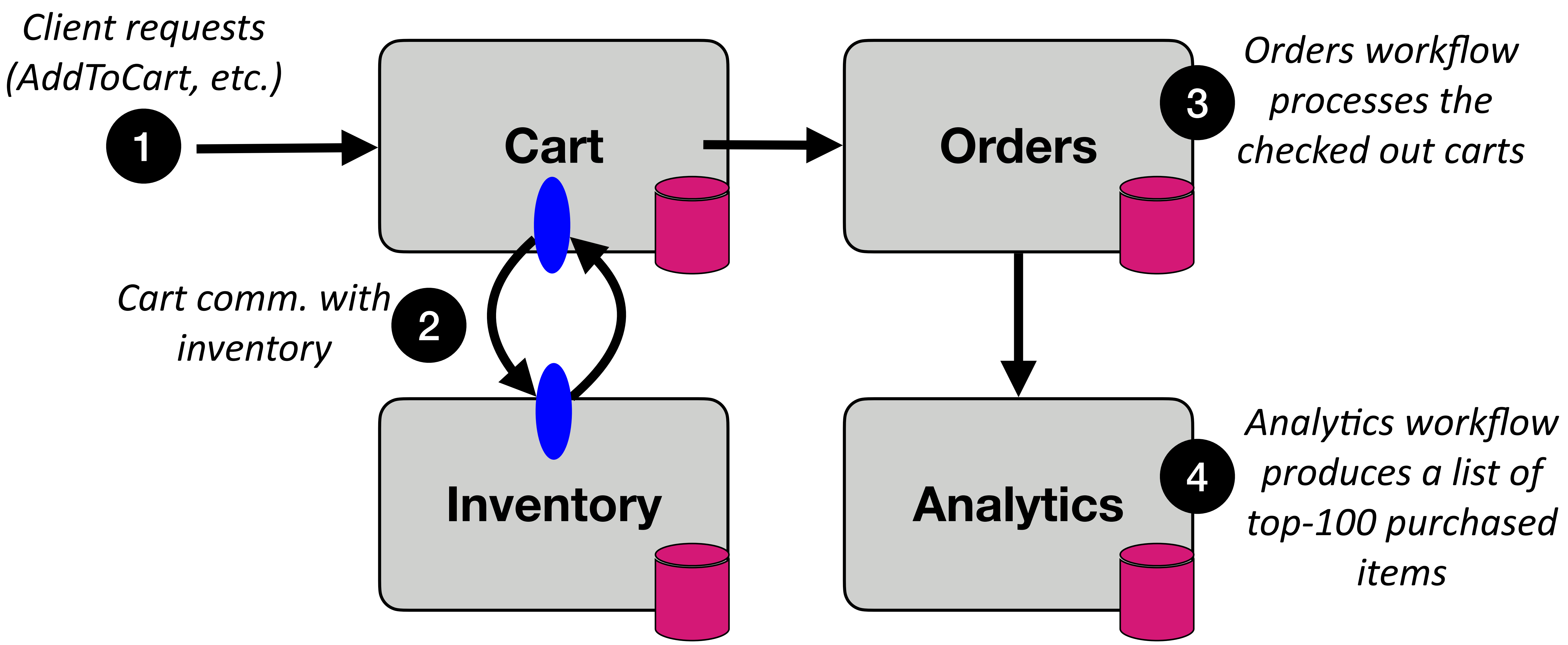
Implementation

- **Exactly-once processing**
    - In workflows: similar to Flink/Kafka
    - For Portals: we can use similar mechanism **because topology is static** (uses *reply streams)*
- **Performance**
    - Leverage performance of stream processing systems
- **All built on streams**
    - **Atomic streams**: single-producer multi-consumer
    - **Reply streams:** atomic streams which can be replied to; multi-producer single-consumer

Examples

- **1) Shopping Cart**
  - Compositions of workflows
  - Microservices request/reply with Portals
  - Futures
- **2) Implementing the Actor Model using Cyclic Workflows**
  - Iterative programming models

Shopping Cart Example



- Framework guarantees end-to-end guarantees, across all services
- Check out examples @ https://github.com/portals-project/portals

Shopping Cart Example: Inventory

```scala
PortalsApp("Inventory"):
  val inventoryOpsGenerator = Generators
          .generator(ShoppingCartData.inventoryOpsGenerator)

  val portal = Portal[InventoryReqs, InventoryReps]("inventory", keyFrom)

  val inventory = Workflows[InventoryReqs, Nothing]("inventory")
    .source(inventoryOpsGenerator.stream)
    .key(keyFrom(_))
    .task(InventoryTask(portal))
    .withName("inventory")
    .sink()
    .freeze()
```

Shopping Cart Example: Inventory

```
object InventoryTask:
  def apply(portal: PortalRef): Task =
    Tasks.taskWithReplier(portal)(onNext)(onMessage)

  private final val state: PerKeyState[Int] =
    PerKeyState[Int]("state", 0)

  private def onMessage(msg: InventoryReqs)(using RepContext): Unit = msg match
    case e: Get => get_req(e)
    case e: Put => put_req(e)

  private def get_req(e: Get)(using RepContext): Unit =
   state.get() match
      case x if x > 0 =>
        reply(GetReply(e.item, true))
        state.set(x - 1)
      case _ =>
        reply(GetReply(e.item, false))
  ...
_
```

Shopping Cart Example: Cart

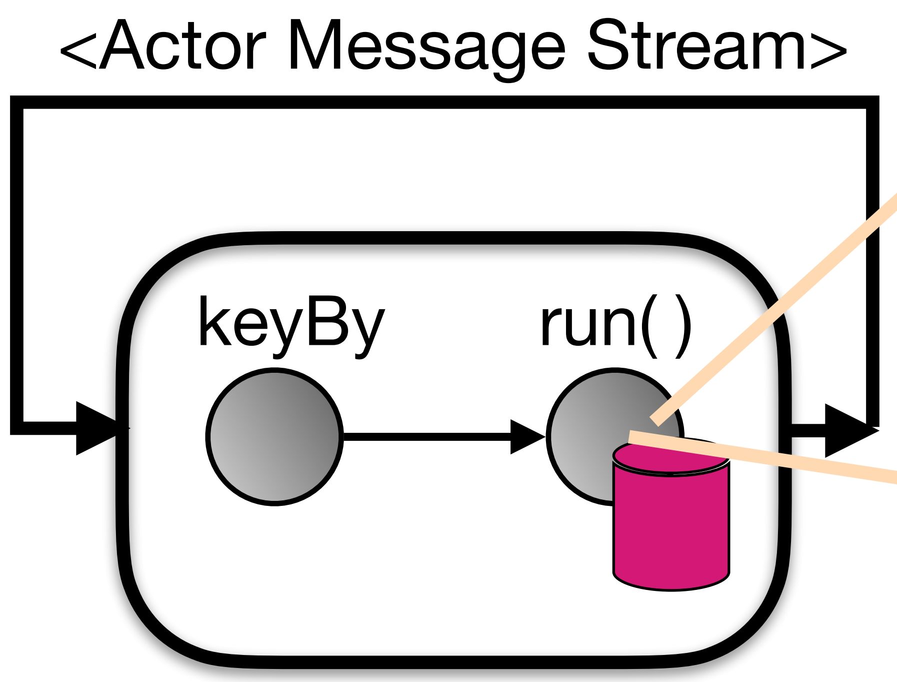
```
PortalsApp("Cart"):
  val cartOpsGenerator = Generators
    .generator(ShoppingCartData.cartOpsGenerator)

  val portal = Registry
    .portals
    .get[InventoryReqs, InventoryReps]("/Inventory/portals/inventory")

  val cart = Workflows[CartOps, OrderOps]("cart")
    .source(cartOpsGenerator.stream)
    .key(keyFrom(_))
    .task(CartTask(portal))
    .withName("cart")
    .sink()
    .freeze()
_
```

Shopping Cart Example: Cart

```
object CartTask:
  ...
  def apply(portal: PortalRef): Task =
    Tasks.taskWithRequester(portal)(onNext(portal))

  private final val state: PerKeyState[CartState] =
    PerKeyState[CartState]("state", CartState.zero)

  private def onNext(portal: PortalRef)(event: CartOps)(using Context): Unit =
    event match
      case event: AddToCart => addToCart(event, portal)
      case event: RemoveFromCart => removeFromCart(event, portal)
      case event: Checkout => checkout(event)

  private def addToCart(event: AddToCart, portal: PortalRef)(using Context): Unit =
    val request = Get(event.item)
    val response = ask(portal)(request)
    response.onComplete:
      case Success(GetReply(item, true)) =>
        state.set(state.get().add(item))
      case Success(GetReply(item, false)) => ...
      case _ => ...
```

Example Library: The Classic Actor Model Implemented on Portals

Simple to implement with Cyclic Workflows



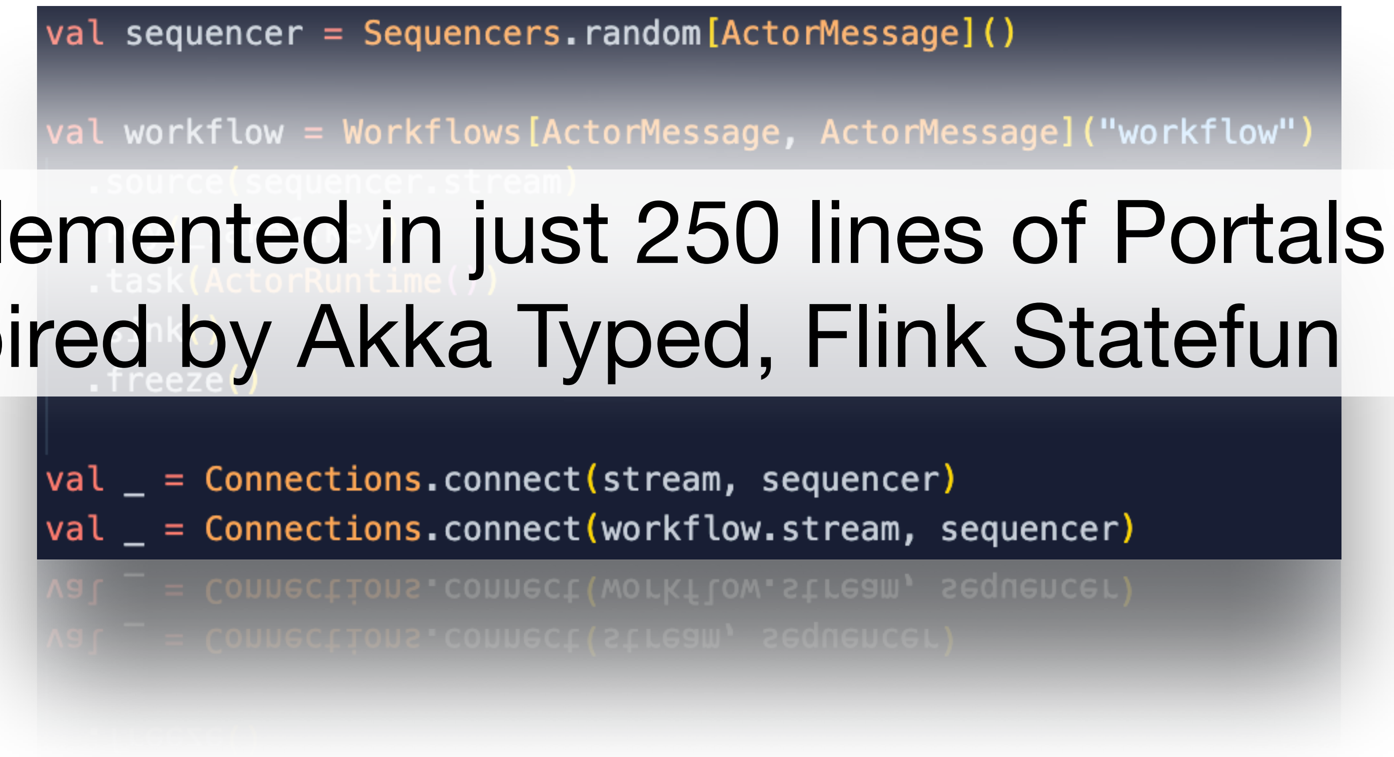

```
def run(msg, ctx):
  actx = ActorCtx(ctx, msg.id)
  actor = state.load(msg.id)
  newActor = actor
    .run(msg.event, actx)
  state.save(msg.id, newActor)
  actx.emitMessages()
```

Simplified Runtime Illustration

- Messages are cycled back, distributed
- Messages are routed by Actor Identity (keyBy)
- Actors are run virtually by the operators

Guarantees, performance

- Inherits exactly-once processing guarantees
  - Remember: difficult with actors
- Performance, data-parallel



- Implemented in just 250 lines of Portals code
- Inspired by Akka Typed, Flink Statefun

- Check out library @ https://github.com/portals-project/portals

Example: Classic Actor Model

```scala
object FibActors:
    val fibBehavior: ActorBehavior[FibCommand] =
      ...
      val fibValue = ValueTypedActorState[Int]("fibValue")
      val fibCount = ValueTypedActorState[Int]("fibCount")
      val fibReply = ValueTypedActorState[ActorRef[FibReply]]("fibReply")
      ActorBehaviors.receive {
        case Fib(replyTo, i) =>
          i match
            case 0 =>
              ctx.send(replyTo)(FibReply(0))
              ActorBehaviors.same
            case 1 => ...
            case n =>
              fibValue.set(0); fibCount.set(0); fibReply.set(replyTo)
              ctx.send(ctx.create(fibBehavior))(Fib(ctx.self, n - 1))
              ctx.send(ctx.create(fibBehavior))(Fib(ctx.self, n - 2))
              ActorBehaviors.same
        case FibReply(i) =>
          ...
      }
    }
_
```

Example: Classic Actor Model

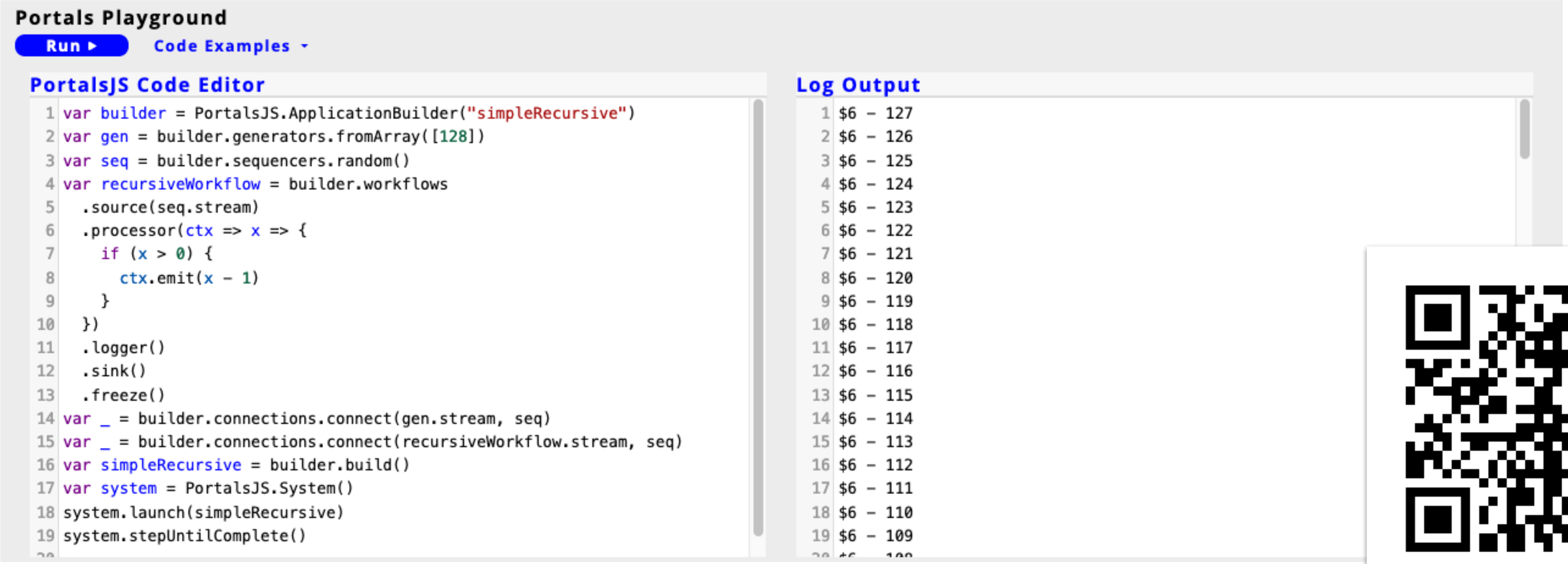
```
object ActorWorkflow:
  ...
    val sequencer = Sequencers.random[ActorMessage]()

    val workflow = Workflows[ActorMessage, ActorMessage]("workflow")
      .source(sequencer.stream)
      .key(_.aref.key)
      .task(ActorRuntime(config))
      .sink()
      .freeze()

    val _ = Connections.connect(stream, sequencer)
    val _ = Connections.connect(workflow.stream, sequencer)

    workflow
_
```

Example: Classic Actor Model

```
object ActorRuntime:
  def apply(config: ActorConfig): Task[ActorMessage, ActorMessage] =
      ...
      val behavior = PerKeyState[ActorBehavior[Any]]("behavior", NoBehavior)
      ...
          case ActorSend(aref, msg) => {
           behavior.get() match
              case NoBehavior =>
                ...
              case ReceiveActorBehavior(f) =>
                f(actx)(msg) match
                  case b @ ReceiveActorBehavior(f) => behavior.set(b)
                  case b @ StoppedBehavior => behavior.set(b)
                  ...
          }
          case ActorCreate(aref, newBehavior) => {
            ...
          }
        }
      }
    }
_
```

The Portals Playground

- Portals compiled to Javascript with Scala.js
- Run Portals apps in the browser
- https://www.portals-project.org/playground/



Portals Project Information

- Portals is an open-source framework, Apache 2.0 license
- Stateful serverless framework
- Combines guarantees and performance of stream processing with the flexibility of actors
- Guarantees end-to-end exactly-once processing
- **Written in Scala 3**
- Ongoing work on the distributed runtime
- Planning release 2023/2024
- https://github.com/portals-project/portals
- Jonas Spenger, Paris Carbone, and Philipp Haller. "*Portals: An extension of dataflow streaming for stateful serverless.*" ONWARD'22 @ SPLASH'22 https://doi.org/10.1145/3563835.3567664

The Portals Framework

- **Stateful serverless framework**
- **Combines guarantees and performance of stream processing with the flexibility of actors**
- **Guarantees end-to-end exactly-once processing**
- **Flexible programming model**
    - **Compositions of workflows** using **Atomic Streams**, cycles, dynamically reconfigurable
    - **Actor-like communication with Portals**, request/reply interaction with streams

Thanks to the core team members: **Jonas Spenger, Paris Carbone, Philipp Haller**; and thanks to all contributors: **Aleksey Veresov; Maxi Kurzawski; Chengyang Huang; Gabriele Morello; Siyao Liu**.

We warmly welcome contributions!

https://www.portals-project.org/contribute

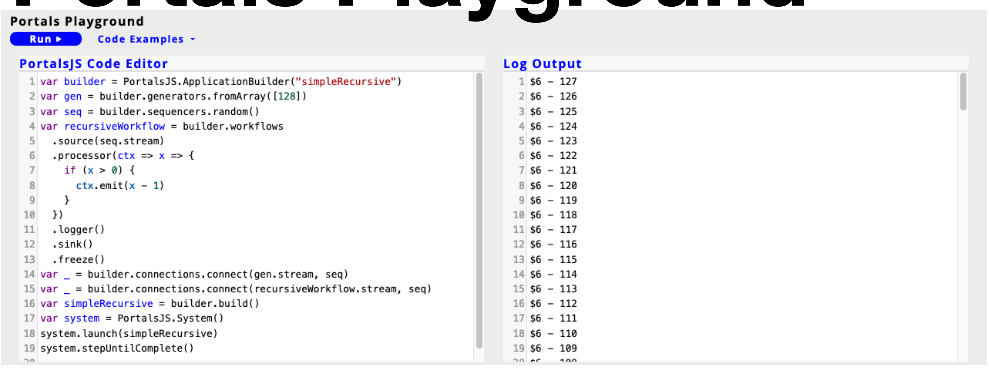
Portals Tutorial

https://www.portals-project.org/learn/tutorial

Portals Repo⭐

https://github.com/portals-project/portals

Portals Playground



https://www.portals-project.org/playground/

Related Work

- Durable Functions
- IBM KAR
- Flink Stateful Functions
- Stateflow
- Orleans
- Kalix
- Ray
- Cloudburst
- Apache Flink, Google Dataflow, Timely Dataflow
- Akka, Erlang