Portals: A Showcase of Multi-Dataflow Stateful Serverless

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

Chengyang Huang
KTH Royal Institute of Technology
Stockholm, Sweden
chehuang@kth.se

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

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

ABSTRACT

Serverless applications spanning the cloud and edge require flexible programming frameworks for expressing compositions across the different levels of deployment. Another critical aspect for applications with state is failure resilience beyond the scope of a single dataflow graph that is the current standard in data streaming systems. This paper presents Portals, an interactive, stateful dataflow composition framework with strong end-to-end guarantees. Portals enables event-driven, resilient applications that span across dataflow graphs and serverless deployments. The demonstration exhibits three scenarios in our multi-dataflow streaming-based system: dynamically composing a stateful serverless application; an interactive cloud and edge serverless application; and a Portals browser playground.

PVLDB Reference Format:
Jonas Spenger, Chengyang Huang, Philipp Haller, and Paris Carbone. Portals: A Showcase of Multi-Dataflow Stateful Serverless. PVLDB, 16(12): 4054 - 4057, 2023.
doi:10.14778/3611540.3611619

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/portals-project/portals.

1 INTRODUCTION

Serverless functions, such as Amazon AWS Lambda [3], have enabled developers to write and deploy functions on fully managed elastic serverless runtimes with great ease of use, allocating resources on demand. Over time, these solutions have become more sophisticated by including management of state and communication, with examples such as Cloudburst [11], Durable Functions [4, 5], Stateful Functions on Apache Flink [1, 8], and KAR [12].

In spite of these developments, the growing need for local-first processing capabilities in edge computing is not yet satisfied. Modern end-to-end applications require a new take on flexible data service composition, stepping away from cloud-managed infrastructures towards decentralized, local-first execution. To that end, the overarching challenge lies in finding an all-encompassing stateful serverless processing model that can provide both cloud-grade reliability and local-first computing flexibility.

Portals is a first step towards cloud and edge stateful serverless dataflow processing [10]. With serverless and edge execution in mind, Portals provides the following unique capabilities:

- ❶ *Multi-Dataflow Applications.* Multiple stateful dataflow streaming pipelines can *dynamically* be composed together on top of *atomic streams*, a transactional type of data streams.
- ❷ *Inter-Dataflow Services.* The *Portal* abstraction binds dataflow pipelines together to create and expose reusable services. A request-reply type of communication between pipelines is employed, by providing serverless access to remote operator states on top of a continuation-style execution.
- ❸ *Decentralized Cloud and Local Execution.* The decentralized runtime can be executed on cloud and edge devices, still providing end-to-end exactly-once processing guarantees.

This paper demonstrates the Portals framework [9, 10], showcasing its use for composing complex serverless multi-dataflow applications. Section 2 provides a background on multi-dataflow serverless applications and the Portals programming model and system. The interactive demo is discussed in section 3, building and running applications in Portals. Section 4 concludes with a summary and an outlook on future work.

2 SERVERLESS DATAFLOWS IN PORTALS

Portals is a serverless multi-dataflow processing system [10]. This section highlights three primary aspects of the programming model: (1) multi-dataflow composition of applications; (2) the *portal* service abstraction for dataflows; and (3) the decentralized cloud and local execution, with end-to-end exactly-once processing guarantees.

① Portals enables the dynamic composition of multi-dataflow applications. Its core building blocks are dataflow pipelines and atomic streams (Figure 1). Dataflow pipelines represent dataflow streaming programs with stateful operators, similar to existing stream processing engines such as Apache Flink [7]; each pipeline is a directed acyclic graph of operators. Pipelines can be connected

Proceedings of the VLDB Endowment, Vol. 16, No. 12 ISSN 2150-8097.
doi:10.14778/3611540.3611619

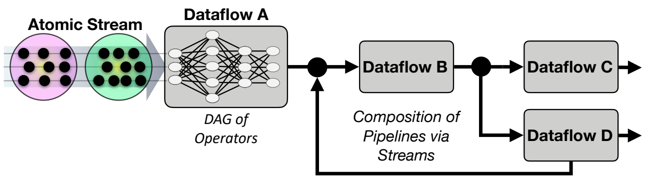


Figure 1: Core building blocks for multi-dataflow composition: dataflow pipelines (gray boxes); and atomic streams (arrows).

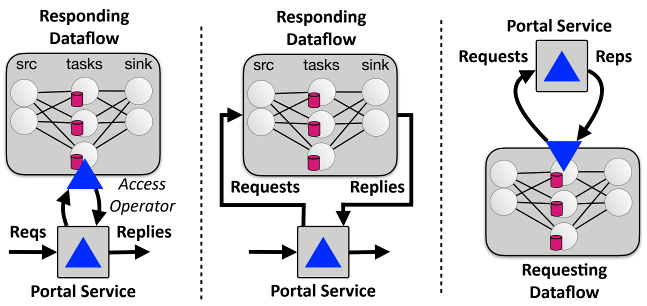


Figure 2: Exposing a service portal of a dataflow pipeline (left, middle), and using a service portal from another dataflow pipeline (right).

Listing 1: User cart as a dataflow, it consumes cart operations (AddToCart, RemoveFromCart, Checkout), produces an orders stream (Order), and communicates to the inventory via the inventory portal.

```
1  Dataflows("cart")
2      .source(cartOpsStream)
3      .requester(inventoryPortal, { x => x match
4          case AddToCart(item) =>
5              req = GetItem(item)
6              future = request(inventoryPortal, req)
7              await(future, {
8                  future.value match // handle reply
9                      case GetItemSuccess =>
10                         // per user−cart state
11                         state.update(item, state.get(item) + 1)
12                     case ... // truncated
13             })
14         case Checkout =>
15             order = Order(state.get( ))
16             emit(order)
17     }
18     .sink( )
19     .freeze( )
```

together via *atomic streams* to form reliable multi-dataflow applications. New pipelines can be added dynamically over time to the deployment, enabling a dynamically evolving application.

② *Portals* △▽, or *service portals*, are a service abstraction of dataflow pipelines (Figure 2). They allow abstracting either an operator (Figure 2, left), or the whole pipeline (middle), as a service.

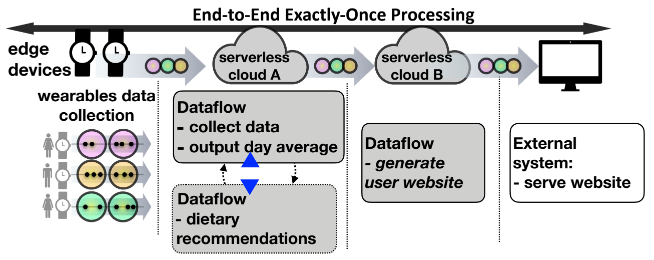


Figure 3: End-to-end IoT wearable analytics application, spanning multiple deployments, cloud and edge devices.

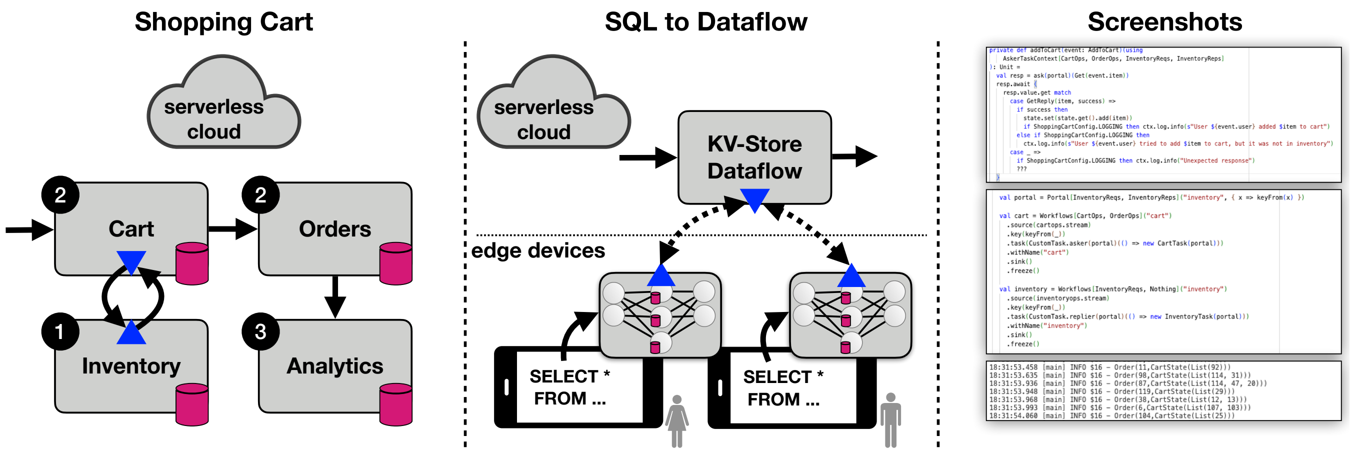
A service takes requests, processes them, and replies back to the requester. The processing may access and modify the state of the operator and may cause the processing operator to call other portals recursively, or to emit events. An exposed portal service can be used by other pipelines (right), or by external systems in an ad hoc manner (e.g., as a query service).

The core use cases of portals are to expose dataflow operators as stateful services. For example, the inventory in our demonstration scenario exposes a portal (Figure 4, left). This portal is used by the cart dataflow (Listing 1), and the interaction is with a futures-based API. The cart binds to the inventoryPortal (Listing 1, line no. 3), and sends it a request (6). This returns a future, and the cart can await (7) on the completion of the future before executing the provided continuation which handles the response (8-12): if the item was in the inventory (GetItemSuccess) it can be added to the cart state (11). The awaiting is not a blocking operation, rather, it registers a continuation which will be executed on receiving the response. Similarly to the cart, the inventory defines a responder which binds to the created inventoryPortal in order to define how to handle the portal requests. Besides the cart and inventory example as discussed here, demonstration scenario 2 showcases a SQL portal interface for a dataflow key-value store (Figure 4, middle).

③ The Portals programming model enables end-to-end exactly-once processing across cloud and edge deployments. An example of this is an IoT wearables analytics application, from trackers to dashboards (Figure 3). The application is deployed on multiple separately hosted serverless clouds, as well as on edge devices. The separately hosted dataflows connect via *remote streams*. Even in this situation, with a decentralized execution, the underlying atomic streaming mechanism ensures that the end-to-end exactly-once processing guarantees are always satisfied.

2.1 System Overview, Implementation

The Portals system consists of an API and a serverless runtime [10]. The API is built around the core multi-dataflow API. The system architecture adopts the standard out-of-order dataflow processing model as used in Google Dataflow [2] and Apache Flink [7]. Portals differ to these systems by supporting multiple decentralized deployments of dataflow pipelines, inspired by virtual and durable actor models [5, 6]. To support this, Portals features an additional management layer for the portals registry and metadata service. The end-to-end exactly-once guarantees are implemented using asynchronous two-phase commit with snapshotting [10].



```
private def addToCart(event: AddToCart)(using
    AskerTaskContext[CartOps, OrderOps, InventoryReqs, InventoryReps]
): Unit =
  val resp = ask(portal)(Get(event.item))
  resp.await {
    resp.value.get match
      case GetReply(item, success) =>
        if success then
          state.set(state.get().add(item))
          if ShoppingCartConfig.LOGGING then ctx.log.info(s"User ${event.user} added $item to cart")
        else if ShoppingCartConfig.LOGGING then
          ctx.log.info(s"User ${event.user} tried to add $item to cart, but it was not in inventory")
      case _ =>
        if ShoppingCartConfig.LOGGING then ctx.log.info("Unexpected response")
        ???
  }
```

```
  val portal = Portal[InventoryReqs, InventoryReps]("inventory", { x => keyFrom(x) })

  val cart = Workflows[CartOps, OrderOps]("cart")
    .source(cartops.stream)
    .key(keyFrom(_))
    .task(CustomTask.asker(portal)(() => new CartTask(portal)))
    .withName("cart")
    .sink()
    .freeze()

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

```
18:31:53.458 [main] INFO $16 - Order(11,CartState(List(92)))
18:31:53.635 [main] INFO $16 - Order(98,CartState(List(114, 31)))
18:31:53.936 [main] INFO $16 - Order(87,CartState(List(114, 47, 20)))
18:31:53.948 [main] INFO $16 - Order(119,CartState(List(29)))
18:31:53.968 [main] INFO $16 - Order(38,CartState(List(12, 13)))
18:31:53.993 [main] INFO $16 - Order(6,CartState(List(107, 103)))
18:31:54.060 [main] INFO $16 - Order(104,CartState(List(25)))
```

Figure 4: Overview over the demonstration scenarios: dynamically launching a shopping cart using *multi-dataflow* composition and *portals* (left, in order 1, 2, 3); SQL to Dataflow, using the *portal* SQL interface to connect to a remote KV-store dataflow instance from edge devices (middle); screenshots from the shopping cart app (right).

The *portal* services are currently implemented through a precompilation step which implements them using the existing primitives in the system. They can be constructed through a combination of internal atomic stream operators called sequencers and splitters, together with a slight alteration of the internal dataflow graphs [10].

The Portals runtime is compiled both to JVM as well as JavaScript, leveraging Scala 3 and the Scala-JS library. This way, Portals programs can run both on lightweight edge devices, as well as on serverless cloud instances, all from the same shared source code.

3 DEMONSTRATION

The demonstration exhibits running serverless multi-dataflow applications in Portals through three scenarios (Figure 4). The first scenario presents dynamically building and launching a serverless e-commerce solution written in Portals. The second scenario presents an interactive SQL Dataflow application, which runs locally on participants' web-browser devices, and can issue requests using the SQL interface of the portal service to a cloud instance. The third scenario presents the Portals playground.

Scenario 1: Complex E-Commerce Pipeline

The first scenario demonstrates, step-by-step, how to launch a serverless e-commerce solution written in Portals (Figure 4, left).

The scenario consists of four services: an inventory; cart; order; and analytics service. The *inventory* manages the inventory state in an operator, and exposes this state as a portal service. It can handle either GetItem (take an item from the inventory) or PutItem (put the item back) requests. The *user cart* interacts with the inventory from one of its operators (see Listing 1). For example, to add an item to the cart, it will have to request to get the item (GetItem) from the inventory, by calling the inventory portal, and await the response from the call. The successfully checked-out carts will be consumed by the *order* service. Lastly, there is an *analytics* service, that consumes the order history in order to provide real-time recommendations. In our example, this produces a top-100 list of item purchases, accessible as a portal service.

Demonstration experience: The demonstration will show, step-by-step, how each service is launched onto the Portals serverless platform, whilst explaining the example code. A data generator is used for simulating data, which will allow the participants to interactively inspect the various services through ad hoc querying of the portals. For example, sending custom GetItem and PutItem queries to the inventory portal, or querying the analytics dataflow for the most purchased items.

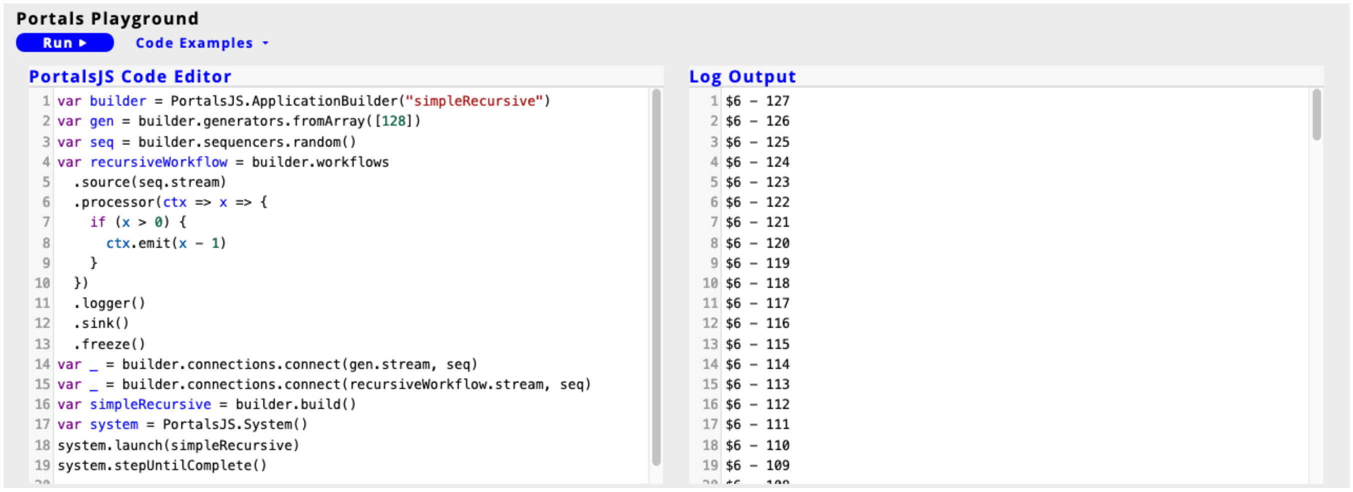
Scenario 2: Interactive SQL to Dataflow

The second scenario demonstrates a cloud and edge serverless application. The cloud runtime runs a key-value store dataflow accessible through a portal. The edge devices run locally on an instance of Portals and can connect to the remote (cloud) instance in order to send queries to the portal. For this demonstration, multiple participants will be able to query, as well as insert values into, the key-value store; the changes will be reflected by all devices connected to the portal. To note is that this application provides end-to-end exactly-once processing guarantees, spanning over all edge devices to the serverless cloud instances.

Demonstration experience: Through a URL, the participants will be able to submit ad hoc SQL queries via a prompt using a JavaScript version of the Portals framework, running the dataflow application in the browser. The queries will use the request/response capability of a portal service to submit the queries and get back the results from the active state, all with exactly-once-processing semantics.

Scenario 3: Portals Playground

The Portals playground (Figure 5) is a JavaScript-based sandbox capable of running the Portals runtime directly in the browser. It supports most of the Portals API with slight differences in the API due to the differences between JavaScript and Scala. The playground can be used to write multi-dataflow applications, connect them with other applications, and to run them locally in the browser. It hosts a code editor and prints the log output.



```
var builder = PortalsJS.ApplicationBuilder("simpleRecursive")
var gen = builder.generators.fromArray([128])
var seq = builder.sequencers.random()
var recursiveWorkflow = builder.workflows
  .source(seq.stream)
  .processor(ctx => x => {
    if (x > 0) {
      ctx.emit(x - 1)
    }
  })
  .logger()
  .sink()
  .freeze()
var _ = builder.connections.connect(gen.stream, seq)
var _ = builder.connections.connect(recursiveWorkflow.stream, seq)
var simpleRecursive = builder.build()
var system = PortalsJS.System()
system.launch(simpleRecursive)
system.stepUntilComplete()
```

```
$6 - 127
$6 - 126
$6 - 125
$6 - 124
$6 - 123
$6 - 122
$6 - 121
$6 - 120
$6 - 119
$6 - 118
$6 - 117
$6 - 116
$6 - 115
$6 - 114
$6 - 113
$6 - 112
$6 - 111
$6 - 110
$6 - 109
```

Figure 5: Portals playground available at https://www.portals-project.org/playground.

The playground is available at https://www.portals-project.org/playground for further explorations, together with example applications (Figure 5).

4 SUMMARY

We have showcased the Portals serverless multi-dataflow streaming runtime [10]. Our demonstration focused on three capabilities: *multi-dataflow*, *i.e.,* dynamically composing multiple dataflow pipelines together via atomic streams; *portal services*, *i.e.,* exposing dataflow pipelines (operators, state) as inter-dataflow services; and *cloud and edge decentralized execution*, *i.e.,* a decentralized runtime with end-to-end exactly-once processing guarantees that runs both on edge devices as well as on serverless cloud instances.

The demonstration exhibited three scenarios. The first scenario showcased how to build an analytics-powered serverless e-commerce solution in Portals . The second scenario showcased a cloud and edge runtime, with a dataflow key-value store accessed via the portal SQL interface. The third scenario showcased the Portals playground.

Future Work

In the near future, we plan to work on a dataflow optimizer for Portals , exploiting the uniquely global view over the deployment; in particular, we are interested in improving the performance of cyclic dependencies across pipelines through dataflow rewriting. Additionally, we plan to work on multi-dataflow ACID transactions.

ACKNOWLEDGMENTS

This work was partially funded by Digital Futures, the Swedish Foundation for Strategic Research under Grant No.: BD15-0006, as well as RISE AI.

REFERENCES

[1] Adil Akhter, Marios Fragkoulis, and Asterios Katsifodimos. 2019. Stateful Functions as a Service in Action. *Proc. VLDB Endow.* 12, 12 (2019), 1890–1893. https://doi.org/10.14778/3352063.3352092

[2] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *Proc. VLDB Endow.* 8, 12 (2015), 1792–1803. https://doi.org/10.14778/2824032.2824076

[3] Amazon Web Services. 2023. AWS Lambda. https://aws.amazon.com/lambda/. Accessed: 2023-03-20.

[4] Sebastian Burckhardt, Badrish Chandramouli, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, Christopher Meiklejohn, and Xiangfeng Zhu. 2022. Netherite: Efficient Execution of Serverless Workflows. *Proc. VLDB Endow.* 15, 8 (2022), 1591–1604. https://www.vldb.org/pvldb/vol15/p1591-burckhardt.pdf

[5] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S. Meiklejohn. 2021. Durable functions: semantics for stateful serverless. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–27. https://doi.org/10.1145/3485510

[6] Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and Jorgen Thelin. 2011. Orleans: cloud computing for everyone. In *ACM Symposium on Cloud Computing in conjunction with SOSP 2011, SOCC '11, Cascais, Portugal, October 26-28, 2011*, Jeffrey S. Chase and Amr El Abbadi (Eds.). ACM, 16. https://doi.org/10.1145/2038916.2038932

[7] 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. http://sites.computer.org/debull/A15dec/p28.pdf

[8] The Apache Software Foundation. 2022. Apache Flink Stateful Functions. https://nightlies.apache.org/flink/flink-statefun-docs-release-3.2/. Accessed on 2022-06-26.

[9] Portals Project Committee. 2022. Portals. https://www.portals-project.org/. Accessed: 2023-03-20.

[10] 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! 2022, Auckland, New Zealand, December 8-10, 2022*, Christophe Scholliers and Jeremy Singer (Eds.). ACM, 153–171. https://doi.org/10.1145/3563835.3567664

[11] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst: Stateful Functions-as-a-Service. *Proc. VLDB Endow.* 13, 11 (2020), 2438–2452. http://www.vldb.org/pvldb/vol13/p2438-sreekanti.pdf

[12] Olivier Tardieu, David Grove, Gheorghe-Teodor Bercea, Paul Castro, Jaroslaw Cwiklik, and Edward Epstein. 2023. Reliable Actors with Retry Orchestration. *Proc. ACM Program. Lang.* 7, PLDI, Article 159 (jun 2023), 24 pages. https://doi.org/10.1145/3591273