[Extended Abstract] Following the Data, Not the Function: Rethinking Function Orchestration in Serverless Computing

Minchen Yu[†] Tingjia Cao[‡] Wei Wang[†] Ruichuan Chen[§]
[†]Hong Kong University of Science and Technology
[‡]University of Wisconsin-Madison [§]Nokia Bell Labs

1 Background and Motivation

Serverless computing, with its Function-as-a-Service incarnation, is becoming increasingly popular in the cloud. It allows developers to write highly scalable, event-driven applications as a set of short-running functions. The interactions between functions are specified as *workflows*, and the serverless platform manages resource provisioning, function orchestration, autoscaling, logging, and fault tolerance for these workflows.

Serverless workflows typically consist of multiple interactive functions with diverse function-invocation and data-exchange patterns [7, 13–15, 17, 20, 21]. Ideally, a serverless platform should provide an expressive and easy-to-use function orchestration to support various interaction patterns. The orchestration should also be made efficient, enabling low-latency invocation and fast data exchange between functions. However, function orchestration in current serverless platforms is neither efficient nor easy to use with three limitations.

Limited expressiveness. Current serverless platforms take a *function-oriented* approach to orchestrating a serverless workflow: individual functions are connected by the workflow that specifies their invocation dependencies. For example, many platforms model a serverless workflow as a directed acyclic graph (DAG) [3, 4, 6, 10, 18], in which the nodes represent functions and the edges indicate the invocation dependencies between functions. Yet, this approach is oblivious to when and how data are exchanged between functions. Without such knowledge, the serverless platform assumes that the output of a function is entirely and immediately consumed by the next function(s), which is not the case in many applications, e.g., the data shuffle [17] and batch data processing [8] in Fig. 1. Consequently, funcion-oriented approach is inconvenient or incapable of expressing sophisticated function interactions, mandating developers to create their own workarounds [15].

Limited usability. As serverless functions are stateless without direct communications, developers need to implement data exchange using various options. For example, functions can exchange data either synchronously or asynchronously via a message broker or a shared storage [1,2,6,9,16,18]. They can also process data from various sources, such as nested function calls, message queues, or other cloud services [5]. Due to the lack of a single best approach to exchange data, developers may need to write complex logic to dynamically select the most efficient way at runtime, which significantly reduces the usability of serverless platforms. Fig. 2 illustrates this problem by comparing four data-passing approaches in AWS Lambda, where there is no single best approach.

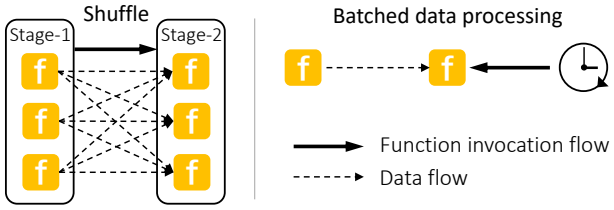

Figure 1: The shuffle operation (left) in data analytics that involves all-to-all data exchange, and the batched data processing (right) in a stream that aggregates accumulated data in a time window. The data flow is not the same as function invocation dependency.

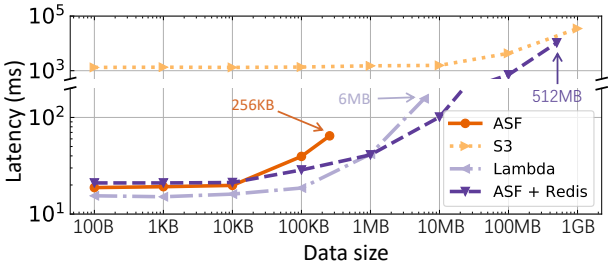

Figure 2: The interaction latency of two AWS Lambda functions under various data sizes using AWS Step Functions (ASF), S3, nested function call (Lambda), and Redis (ASF + Redis). No single approach can prevail across all scenarios.

Limited applicability. Current serverless platforms usually have a function interaction delay of multiple or tens of milliseconds (e.g., 20 ms in AWS Step Functions), and such delays accumulate as more functions are chained together in an application workflow, which can be unacceptable to latency-sensitive applications [14]. In addition, as current serverless platforms cannot efficiently support the sharing of varying-sized data between functions (Fig. 2), they are ill-suited for data-intensive applications [13, 17]. Altogether, the above characteristics substantially limit the applicability of current serverless platforms.

2 Data-Centric Function Orchestration

Key insight. We propose that function orchestration should follow the flow of data rather than the function-level invocation dependencies, thus a *data-centric approach*. We note that intermediate data are typically short-lived and im-

Table 1: Comparison between the function-oriented workflow primitives in AWS Step Functions (ASF) and the data-centric trigger primitives in Pheromone.

Invocation Patterns	ASF	Pheromone
Sequential Execution	Task	Immediate
Conditional Invocation	Choice	ByName
Assembling Invocation	Parallel	BySet
Dynamic Parallel	Map	DynamicJoin
Batched Data Processing	-	ByBatchSize ByTime
k-out-of-*n*	-	Redundant
MapReduce	-	DynamicGroup

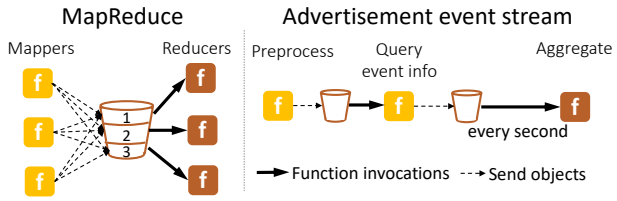

Figure 3: Usage examples of two primitives: DynamicGroup that dynamically divides data into multiple groups, each triggering a reduce function (left), and ByTime that periodically triggers aggregate function with accumulated data (right).

mutable [16, 19]: they wait to be consumed once they are generated. We therefore make data consumption explicit and enable it to trigger the target functions. Developers can specify when and how intermediate data should be consumed by the target functions and trigger their activation, thus driving the execution of an entire workflow.

The data-centric function orchestration addresses the limitations of the current practice via three key advances. First, it breaks the tight coupling between function flows and data flows, as data do not have to follow the exact order of function invocations. It therefore can enable a flexible and fine-grained control over data consumption, allowing developers to express a rich set of workflow patterns (i.e., *rich expressiveness*). Second, the data-centric function orchestration provides a unified programming interface for both function invocations and data exchange, obviating the need for developers to implement complex logic via a big mix of external services to optimize data exchange (i.e., *high usability*). Third, knowing when and how the intermediate data will be consumed provides opportunities for the scheduler to optimize the locality of functions and relevant data, and thus latency-sensitive and data-intensive applications can be supported efficiently (i.e., *wide applicability*).

Data bucket and trigger primitive. In data-centric function orchestration, we design a *data bucket* abstraction that holds the intermediate results of functions in a logical object store. The data bucket provides a rich set of data trigger primitives that developers can use to specify *when* and *how* the intermediate data are passed to the intended function(s) and trigger their execution.

We have developed a new serverless platform, Pheromone, that supports data-centric function orchestration. Table 1 lists trigger primitives supported in Pheromone, which can enable more sophisticated invocation patterns compared with ASF [3]. Fig. 3 further illustrates how the trigger primitives DynamicGroup and ByTime can be used to enable data shuffling and periodic data aggregation respectively.

3 System Design

Pheromone achieves high-performance data-centric orchestration with two key designs. First, it uses a two-tier distributed scheduling hierarchy to exploit data locality enabled by the data-centric design. Each worker node runs a local scheduler, which keeps track of workflow execution status via data buckets and locally schedules subsequent functions whenever possible, thus reducing the invocation latency. For a large workflow running across multiple workers, a global coordinator can gather bucket statuses from relevent schedulers and route next function requests to a worker with most intermediate data. Pheromone's global coordinators are sharded to ensure high scalability, where each coordinator only handles a disjoint set of workflows. Second, Pheromone trades the durability of intermediate data for fast data exchange. Functions exchange data within a node through a zero-copy shared-memory object store to fully reap the benefits of data locality. Pheromone also enables direct transfer of intermediate objects between nodes for efficient remote data exchange.

4 Experimental Results

We have evaluated Pheromone against well-established commercial and open-source serverless platforms, including AWS Lambda with Step Functions, Azure Durable Functions, Cloudburst [18], and KNIX [6]. Evaluation results show that Pheromone improves the function invocation latency by up to 10× and 450× over Cloudburst (best open-source baseline) and AWS Step Functions (best commercial baseline), respectively. Pheromone has negligible data-exchange overhead (e.g., tens of μs) thanks to its zero-copy data exchange. It can also scale well to large workflows and incurs only millisecond-scale orchestration overhead when running thousands of functions, whereas the overhead is at least a few seconds in other platforms.

We also provide case studies of two serverless applications, i.e., Yahoo! stream processing [11] and MapReduce sort [12] (examples in Fig. 3). Compared with other serverless platforms, Pheromone allows developers to easily express function interaction patterns of the complex workflows (*rich expressiveness*), requires no specific implementation to handle data exchange between functions (*high usability*), and efficiently supports both latency-sensitive and data-intensive applications (*wide applicability*).

References

[1] AWS ElastiCache. https://aws.amazon.com/elasticache/.

[2] AWS S3. https://aws.amazon.com/s3/.

[3] AWS Step Functions. https://aws.amazon.com/step-functions/.

[4] Google Cloud Composer. https://cloud.google.com/composer.

[5] Invoking AWS Lambda functions. https://docs.aws.amazon.com/lambda/latest/dg/lambda-invocation.html.

[6] KNIX Serverless. https://github.com/knix-microfunctions/knix/.

[7] Serverless applications scenarios. https://docs.aws.amazon.com/wellarchitected/latest/serverless-applications-lens/scenarios.html.

[8] Serverless reference architecture: Real-time stream processing. https://github.com/aws-samples/lambda-refarch-streamprocessing/.

[9] Use Amazon S3 ARNs instead of passing large payloads. https://docs.aws.amazon.com/step-functions/latest/dg/avoid-exec-failures.html.

[10] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards high-performance serverless computing. In *Proc. USENIX ATC*, 2018.

[11] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, and P. Poulosky. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *Proc. IEEE IPDPSW*, 2016.

[12] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *Proc. USENIX OSDI*, 2004.

[13] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *Proc. USENIX NSDI*, 2017.

[14] Zhipeng Jia and Emmett Witchel. Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proc. ACM ASPLOS*, 2021.

[15] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99%. In *Proc. ACM SoCC*, 2017.

[16] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *Proc. USENIX OSDI*, 2018.

[17] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *Proc. USENIX NSDI*, 2019.

[18] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: stateful functions-as-a-service. In *Proc. VLDB Endow.*, 2020.

[19] Yang Tang and Junfeng Yang. Lambdata: Optimizing serverless computing by making data intents explicit. In *Proc. IEEE CLOUD*, 2020.

[20] Minchen Yu, Zhifeng Jiang, Hok Chun Ng, Wei Wang, Ruichuan Chen, and Bo Li. Gillis: Serving large neural networks in serverless functions with automatic model partitioning. In *Proc. IEEE ICDCS*, 2021.

[21] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. Caerus: NIMBLE task scheduling for serverless analytics. In *Proc. USENIX NSDI*, 2021.