Pheromone: Restructuring Serverless Computing With Data-Centric Function Orchestration

Minchen Yu, *Member, IEEE*, Tingjia Cao, Wei Wang, *Member, IEEE*, and Ruichuan Chen, *Member, IEEE*

***Abstract*— Serverless applications are typically composed of function workflows in which multiple short-lived functions are triggered to exchange data in response to events or state changes. Current serverless platforms coordinate and trigger functions by following high-level invocation dependencies but are oblivious to the underlying data exchanges between functions. This design is neither efficient nor easy to use in orchestrating complex workflows – developers often have to manage complex function interactions by themselves, with customized implementation and unsatisfactory performance. Therefore, we argue that function orchestration should follow a *data-centric approach*. In our design, the platform provides a *data bucket abstraction* to hold the intermediate data generated by functions. Developers can use a rich set of data trigger primitives to control when and how the output of each function should be passed to the next functions in a workflow. By making data consumption explicit and allowing it to trigger functions and drive the workflow, complex function interactions can be easily and efficiently supported. We present `Pheromone` – a scalable, low-latency serverless platform following this data-centric design. Compared to well-established commercial and open-source platforms, `Pheromone` cuts the latencies of function interactions and data exchanges by orders of magnitude, scales to large workflows, and enables easy implementation of complex applications.**

***Index Terms*— Serverless computing, function workflow, data sharing.**

I. Introduction

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. Developers simply specify the events that trigger the activation of these functions, and let the serverless platform handle resource provisioning, autoscaling, logging, fault-tolerance, etc. Serverless computing is also economically appealing as it has zero idling cost: developers are only charged when their functions are running.

Many applications have recently been migrated to the serverless cloud [1], [2], [3], [4], [5], [6], [7], [8], [9]. These applications typically consist of multiple interactive functions with diverse function-invocation and data-exchange patterns. For example, a serverless-based batch analytics application may trigger hundreds of parallel functions for all-to-all data communication in a shuffle phase [3], [10], [11]; a stream processing application may repeatedly trigger certain functions to process dynamic data received in a recent time window. Ideally, a serverless platform should provide an *expressive* and *easy-to-use* function orchestration to support various function-invocation and data-exchange 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. It typically models a serverless application as a *workflow* that connects functions according to their invocation dependency [12], [13], [14], [15], [16], [17], [18], [19]. It specifies the order of function invocations but 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 such as the aforementioned "shuffle" operation in batch analytics and the processing of dynamically accumulated data in stream analytics. To work around these limitations, developers have to manage complex function interactions and data exchanges by themselves, using various approaches such as a message broker or a shared storage, either synchronously or asynchronously [10], [13], [14], [20], [21], [22]. As no single approach is found optimal in all scenarios, developers may need to write complex logic to dynamically select the most efficient approach at runtime (see §II-B). Current serverless platforms also incur function interaction latencies of tens of milliseconds, which may be unacceptable to latency-sensitive applications [23], particularly since this overhead accumulates as the function chain builds up.

In this paper, we argue that function orchestration should follow the flow of data rather than the function-level invocation dependency, thus a *data-centric approach*. Our key idea is to make data consumption explicit, and let it trigger functions and drive the workflow. In our design, the serverless platform exposes a *data bucket abstraction* that holds the intermediate output of functions in a logical object store. The data bucket

Received 28 May 2024; accepted 4 October 2024; approved by IEEE TRANSACTIONS ON NETWORKING Editor D. Han. Date of publication 28 November 2024; date of current version 14 February 2025. This work was supported in part by Guangdong Provincial Key Laboratory of Mathematical Foundations for Artificial Intelligence under Grant 2023B1212010001; and in part by Hong Kong Research Grants Council (RGC) under Grant GRF 16202121, Grant GRF 16210822, and Grant CRF C7004-22. *(Corresponding author: Minchen Yu.)*

Minchen Yu is with the School of Data Science, The Chinese University of Hong Kong, Shenzhen, Guangdong 518172, China (e-mail: yuminchen@cuhk.edu.cn).

Tingjia Cao is with the Department of Computer Science, University of Wisconsin–Madison, Madison, WI 53706 USA (e-mail: tcao34@wisc.edu).

Wei Wang is with the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong (e-mail: weiwa@cse.ust.hk).

Ruichuan Chen is with Nokia Bell Laboratories, 70469 Stuttgart, Germany (e-mail: ruichuan.chen@nokia-bell-labs.com).

Digital Object Identifier 10.1109/TNET.2024.3480031

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. With such a fine control of data flow, developers can express sophisticated function invocations and data exchanges, simply by configuring data triggers through a unified interface. Knowing how intermediate data will be consumed also enables the serverless platform to schedule the intended downstream functions close to the input, thus ensuring fast data exchange and low-latency function invocation.

Following this design approach, we develop Pheromone,[1] a scalable serverless platform with low-latency data-centric function orchestration. Pheromone proposes three key designs to deliver high performance. **First**, it uses a two-tier distributed scheduling hierarchy to locally execute a function workflow whenever possible. Each worker node runs a local scheduler, which keeps track of the execution status of a workflow via its data buckets and schedules next functions of the workflow onto local function executors. In case that all local executors are busy, the scheduler forwards the request to a global coordinator which then routes it to another worker node with available resources. **Second**, Pheromone trades the durability of intermediate data, which are typically short-lived and immutable, for fast data exchange. Functions exchange data within a node through a zero-copy shared-memory object store; they can also pass data to a remote function through direct data transfer. **Third**, Pheromone uses sharded global coordinators, each handling a disjoint set of workflows. With such a shared-nothing design, local schedulers only synchronize workflows' execution status with the corresponding global coordinators, which themselves require no synchronization, thus ensuring high scalability for distributed scheduling.

We evaluate Pheromone against well-established commercial and open-source serverless platforms, including AWS Lambda with Step Functions, Azure Durable Functions, Cloudburst [14], and KNIX [13]. 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 scales well to large workflows and incurs only millisecond-scale orchestration overhead when running 1k chained functions and 4k parallel functions, whereas the overhead is at least a few seconds in other platforms. Pheromone has negligible data-exchange overhead (e.g., tens of μs), thanks to its zero-copy data exchange. It can also handle failed functions through efficient re-execution. Case studies of three serverless applications, i.e., Yahoo! stream processing [24], real-time query, and MapReduce sort, further demonstrate that Pheromone can easily express complex function interaction patterns (*rich expressiveness*), require no specific implementation to handle data exchange between functions (*high usability*), and efficiently support both latency-sensitive and data-intensive applications (*wide applicability*).

[1]Pheromone is a chemical signal produced and released into the environment by an animal that triggers a social response of others of its species. We use it as a metaphor for our data-centric function orchestration approach.

II. Background and Motivation

We first introduce serverless computing and discuss the limitations of function orchestration in current serverless platforms.

A. Serverless Computing

Serverless computing, with its popular incarnation being Function-as-a-Service (FaaS), has recently emerged as a popular cloud computing paradigm that supports highly-scalable, event-driven applications [25], [26], [27]. Serverless computing allows developers to write short-lived, stateless functions that can be triggered by events. The interactions between functions are simply specified as *workflows*, and the serverless platform manages resource provisioning, function orchestration, autoscaling, logging, and fault tolerance for these workflows. This paradigm appeals to many developers as it allows them to concentrate on the application logic without having to manage server resources [28], [29] – hence the name serverless computing. In addition to the high scalability and operational simplicity, serverless computing adopts a "pay-as-you-go" billing model: developers are billed only when their functions are invoked, and the function run-time is metered at a fine granularity, e.g., 1 ms in major serverless platforms [25], [26]. Altogether, these benefits have increasingly driven a large number of traditional "serverful" applications to be migrated to the serverless platforms, including batch analytics [2], [3], [4], [11], video processing [5], [6], stream processing [1], machine learning [7], [8], microservices [23], etc.

B. Limitations of Current Platforms

Current serverless platforms take a *function-oriented* approach to orchestrating and activating the functions of a serverless workflow: each function is treated as a single and standalone unit, and the interactions of functions are separately expressed within a workflow. This workflow connects individual functions according to their invocation dependencies, such that each function can be triggered upon the completion of one or multiple upstream functions. For example, many platforms model a serverless workflow as a directed acyclic graph (DAG) [12], [13], [14], [15], [16], [17], [18], [19], in which the nodes represent functions and the edges indicate the invocation dependencies between functions. The DAG can be specified using general programming languages [17], [18], or domain-specific languages such as Amazon States Language [13], [19]. However, this approach has several limitations with regard to expressiveness, usability, and applicability.

Limited expressiveness. Although the current function-oriented orchestration supports the workflows of simple invocation patterns, it becomes inconvenient or incapable of expressing more sophisticated function interactions, as summarized in Table I. This is because the current function orchestration assumes that data flow in the same way as how functions are invoked in a workflow, and that a function passes its entire output to others by directly invoking them for immediate processing. These assumptions do not hold for many applications, hence developers resort to create workarounds.

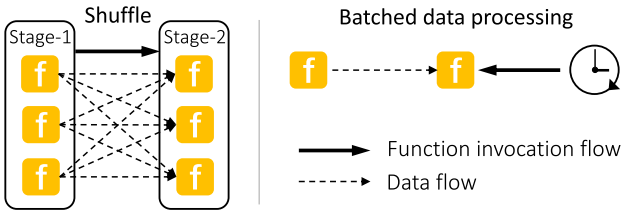


Fig. 1. The shuffle operation (left) in data analytics and the batched data processing in a stream (right).

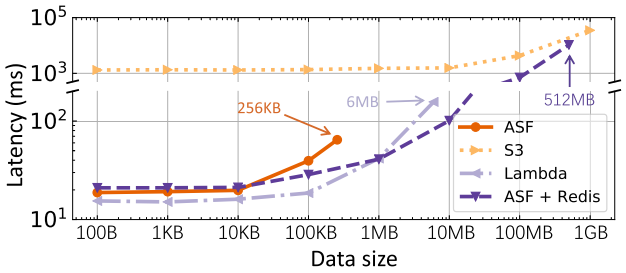


Fig. 2. The interaction latency of two AWS Lambda functions under various data sizes using four approaches.

For example, the "shuffle" operation in a data analytics job involves a fine-grained, all-to-all data exchange between the functions of two stages (e.g., "map" and "reduce" stages). As shown in Fig. 1 (left), the output data of a function in stage-1 are shuffled and selectively redistributed to multiple functions in stage-2 based on the output keys. However, the way to invoke functions is not the same as how the output data flow: only after stage-1 completes can the workflow invoke all the stage-2 functions in parallel. In current serverless platforms, developers must manually implement such a complex data shuffle invocation via external storage [3], [10], which is neither flexible nor efficient.

Another example is a batched stream analytics job which periodically invokes a function to process the data continuously received during a time window [24], [30], as shown in Fig. 1 (right). A serverless workflow cannot effectively express this invocation pattern as the function is not immediately triggered when the data arrive, and thus developers have to rely on other cloud services (e.g., AWS Kinesis [31]) to batch the data for periodic function invocations [1], [32], [33]. Note that, even with the latest stateful workflow (e.g., Azure Durable Functions [34]), an addressable function needs to keep running to receive data. As we will show in §VI-E, deploying a long-running function not only incurs extra resource provisioning cost but results in an unsatisfactory performance.

Limited usability. Current serverless platforms provide various options for data exchange between functions. Functions can exchange data either synchronously or asynchronously via a message broker or a shared storage [10], [13], [14], [20], [21], [22]. They can also process data from various sources, such as nested function calls, message queues, or other cloud services [35].

The lack of a single best approach to exchange data between functions significantly complicates the development and deployment of serverless applications, as developers must find their own ways to efficiently pass data across functions [16] which can be dynamic and non-trivial; thus, reducing the usability of serverless platforms. To illustrate this problem, we compare four data-passing approaches in AWS Lambda: a) calling a function directly (Lambda), b) using AWS Step Functions (ASF) to execute a two-function workflow,[2] c) allowing functions to access an in-memory Redis store for fast data exchange (ASF+Redis), and d) configuring AWS S3 to invoke a function upon data creation (S3) [37].

[2]We use the ASF Express Workflows in our experiments as it delivers higher performance than the ASF Standard Workflows [36].

Fig. 2 compares the latencies of these four approaches under various data volumes. Lambda is efficient for transferring small data; ASF+Redis is efficient for transferring large data; the maximum data volume supported by each approach varies considerably, and only the S3 approach can support virtually unlimited (but slow) data exchange. Thus, there is no single approach that prevails across all scenarios, and developers must carefully profile the data patterns of their applications and the serverless platforms to optimize the performance of data exchange between interacting functions.

To make matters worse, the data volume exchanged between functions depends on the workload, which may be irregular or unpredictable. Thus, there may be no best *fixed* approach to exchanging data between interacting functions, and developers have to write complex logic to select the best approach at runtime. Developers also need to consider the interaction cost. Previous work has highlighted the tricky trade-off between I/O performance and cost when using different storage to share intermediate data [3], [10], which further exacerbates the usability issue. Altogether, these common practices bring a truly *non-serverless* experience to developers as they still have to deal with server and platform characteristics.

Limited applicability. Existing serverless applications are typically not latency-sensitive. This is because current serverless platforms usually have a function interaction delay of multiple or tens of milliseconds (§VI-B), and such delays accumulate as more functions are chained together in an application workflow. For example, in AWS Step Functions, each function interaction causes a delay of more than 20 ms, and the total platform-incurred delay for a 6-function chain is over 100 ms, which may not be acceptable in many latency-sensitive applications [23]. In addition, as current serverless platforms cannot efficiently support the sharing of varying-sized data between functions (as described earlier), they are ill-suited for data-intensive applications [3], [5], [13], [14], [23], [25]. Altogether, the above characteristics substantially limit the applicability of current serverless platforms.

III. Data-Centric Function Orchestration

In this section, we address the aforementioned limitations of the function orchestration practice in current serverless platforms, with a novel data-centric approach. We will describe how this approach can be applied to develop a new serverless platform later in §IV.

Fig. 3. An overview of triggering functions in data-centric orchestration. Source functions send intermediate data to the associated bucket, which can be configured to automatically trigger target functions.

A. Key Insight

As discussed in §II-B, the current function orchestration practice only specifies the high-level invocation dependencies between functions, and thus has little fine-grained control over how these functions exchange data. In particular, the current practice assumes the tight coupling between function flows and data flows. Therefore, when a function returns its result, the workflow has no knowledge about how the result should be consumed (e.g., in full or part, directly or conditionally, immediately or later). To address these limitations, *an effective serverless platform must allow fine-grained data exchange between the functions of a workflow, while simultaneously providing a unified and efficient approach for function invocation and data exchange*.

Following this insight, we propose a new *data-centric approach* to function orchestration. We note that intermediate data (i.e., results returned by functions) are typically short-lived and immutable [10], [38]: after they are generated, they wait to be consumed and then become obsolete.[3] We therefore make data consumption explicit and enable it to trigger the target functions. Developers can thus specify when and how intermediate data should be passed to the target functions and trigger their activation, which can then drive the execution of an entire workflow. As intermediate data are not updated once they are generated [10], [38], using them to trigger functions results in no consistency issues.

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 also enables a flexible and fine-grained control over data consumption, and therefore can 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 serverless platform 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*).

B. Data Bucket and Trigger Primitives

Data bucket. To facilitate the data-centric function orchestration, we design a *data bucket* abstraction and a list of *trigger primitives*. Fig. 3 gives an overview of how functions are triggered. A serverless application creates one or multiple data buckets that hold the intermediate data. Developers can configure each bucket with triggers that specify when and how the data should invoke the target functions and be consumed by them. When executing a workflow, the source functions directly send their results to the specified buckets. Each bucket checks if the configured triggering condition is satisfied (e.g., the required data are complete and ready to be consumed). If so, the bucket triggers the target functions automatically and passes the required data to them. This process takes place across all buckets, which collectively drive the execution of an entire workflow.

We design various trigger primitives for buckets to specify how functions are triggered. The interaction patterns between functions can be broadly classified into three categories:

Direct trigger primitive (i.e., `Immediate`) allows one or more functions to directly consume data in the associated buckets. This primitive has no specified condition, and triggers the target functions immediately once the data are ready to be consumed. This primitive can easily support sequential execution or invoke multiple functions in parallel (fan-out).

Conditional trigger primitives trigger the target function(s) when the developer-specified conditions are met.

- `ByBatchSize`: It triggers the function(s) when the associated bucket has accumulated a certain number of data objects. It can be used to enable the batched stream processing [32], [33] in a way similar to Spark Streaming.
- `ByTime`: It sets up a timer and triggers the function(s) when the timer expires. All the accumulated data objects are then passed to the function(s) as input. It can be used to implement routine tasks [24], [30].
- `ByName`: It triggers the function(s) when the bucket receives a data object of a specified name. It can be used to enable conditional invocations by choice [39].
- `BySet`: It triggers functions when a specified set of data objects are all complete and ready to be consumed. It can be used to enable the assembling invocation (fan-in).
- `Redundant`: It specifies n objects to be stored in a bucket and triggers the function(s) when any k of them are available and ready to be consumed. It can be used to execute redundant requests and perform late binding for straggler mitigation and improved reliability [40], [41], [42].

Dynamic trigger primitives, unlike the previous two categories with statically-configured triggers, allow data exchange patterns to be configured at runtime.

- `DynamicJoin`: It triggers the assembling functions when a set of data objects are ready, which can be dynamically configured at runtime. It enables the dynamic parallel execution like '`Map`' in AWS Step Functions [43].
- `DynamicGroup`: It allows a bucket to divide its data objects into multiple groups, each of which can be consumed by a set of functions. The data grouping is dynamically performed based on the objects' metadata (e.g., the name of an object). Once a group of data objects are ready, they trigger the associated set of functions.

[3]For data that need durability, they can be persisted to a durable storage.

TABLE I
EXPRESSIVENESS 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

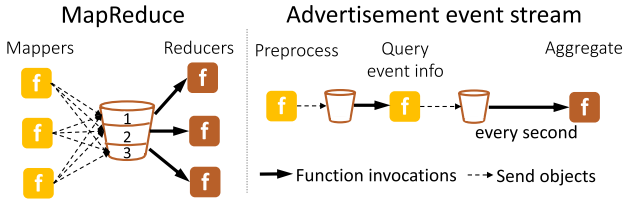


Fig. 4. Usage examples of two primitives: DynamicGroup for data shuffling in MapReduce (left), and ByTime for periodic data aggregation in the event stream processing (right).

Dynamic trigger primitives are critical to implement some widely-used computing frameworks, e.g., MapReduce (which is hard to support in current serverless platforms as it requires triggering parallel functions at every stage and optimizing the fine-grained, all-to-all data exchange between them [2], [3], [10], see §II-B). Here, our DynamicGroup primitive provides an easy solution to these issues. As shown in Fig. 4 (left), when a map function sends intermediate data objects to the associated bucket, it also specifies to which data group each object belongs (i.e., by specifying their associated keys). Once the map functions are all completed, the bucket triggers the reduce functions, each consuming a group of objects.

We have developed a new serverless platform, Pheromone, which implements the aforementioned data bucket abstraction and trigger primitives. The design of Pheromone will be detailed in §IV. Table I lists all the supported trigger primitives in current Pheromone platform. Compared to AWS Step Functions (ASF), Pheromone supports more sophisticated invocation patterns and provides richer expressiveness for complex workflows. We note that Azure Durable Functions [44] can also achieve rich expressiveness for complex workflows (§VI-A). Yet, it fails to achieve the other two desired properties, i.e., high usability and wide applicability (§VI-E).

Abstract interface. Pheromone's trigger primitives are not only limited to those listed in Table I. Specifically, we provide an abstract interface for developers to implement customized trigger primitives for their applications, if needed. Fig. 5 shows the three main methods of the trigger interface. The first method, action_for_new_object, is provided to specify how the trigger's associated target functions should be invoked. This method can be called when a new data object arrives: it checks the current data status and returns a list of functions to invoke, if any. The method can also be called periodically in a configurable time period through periodical checking (e.g., ByTime primitive). The other two methods, notify_source_func and action_for_rerun, are provided to implement the fault handling logic which re-executes the trigger's associated source functions in case of failures. In particular, through notify_source_func, a trigger can obtain the information of a source function once the function starts, including the function name, session, and arguments; Pheromone also performs the periodic re-execution checks by calling action_for_rerun, which returns a list of timeout functions, such that Pheromone can then re-execute them. The detailed fault tolerance mechanism will be described in §IV-D. We give an example of implementing a customized ByBatchSize trigger primitive via the abstract interface in our technical report [45].

```
struct BucketKey {
  string bucket_; // bucket name
  string key_; // key name
  string session_; // unique session id per request
};

abstract class Trigger {
  // Check whether to trigger functions for a new object.
  vector<TriggerAction> action_for_new_object(
      BucketKey bucket_key);

  // Notify the information of a source function.
  void notify_source_func(string function_name,
      string session, vector<string> function_args);

  // Check whether to re-execute source functions.
  vector<TriggerAction> action_for_rerun(string session);
};
```

Fig. 5. Three main methods of the trigger interface.

```
int handle(UserLibraryInterface* library,\
  int arg_size, char** arg_values);
```

Fig. 6. Function interface.

C. Programming Interface

Our Pheromone serverless platform currently accepts functions written in C++, with capabilities to support more languages (see §VII). Pheromone also provides a Python client through which developers can program function interactions.

Function interface. Following the common practice, developers implement their functions through the handle() interface (see Fig. 6), which is similar to the C++ main function except that it takes a user library as the first argument. The user library provides a set of APIs (see Table II) that allow developers to operate on intermediate data objects. These APIs enable developers to create intermediate data objects (EpheObject), set their values, and send them to the buckets. A data object can also be persisted to a durable storage by setting the output flag when calling send_object(). When a bucket receives objects and decides to trigger next function(s), it automatically packages relevant objects as the function arguments (see Fig. 6). A function can also access other objects via the get_object() API.

Bucket trigger configuration. Developers specify how the intermediate data should trigger functions in a workflow via our Python client. The client creates buckets and configures

TABLE II
THE APIS OF USER LIBRARY WHICH DEVELOPERS USE TO OPERATE ON INTERMEDIATE DATA OBJECTS AND DRIVE THE WORKFLOW EXECUTION

Class	API	Description
EpheObject	`void* get_value()`	Get a pointer to the value of an object.
	`void set_value(value, size)`	Set the value of an object.
UserLibrary	`EpheObject* create_object(bucket, key)`	Create an object by specifying its bucket and key name.
	`EpheObject* create_object(function)`	Create an object by specifying its target function.
	`EpheObject* create_object()`	Create an object.
	`void send_object(object, output=false)`	Send an object to its bucket, and set the output flag if it needs to persist.
	`EpheObject* get_object(bucket, key)`	Get an object by specifying its bucket and key name.

```
1 app_name = 'event-stream-processing'
2 bucket_name = 'by_time_bucket'
3 trigger_name = 'by_time_trigger'
4 prim_meta = {'function':'aggregate', 'time_window':1000}
5 re_exec_rules = ([('query_event_info', EVERY_OBJ)], 100)
6 client.create_bucket(app_name, bucket_name)
7 client.add_trigger(app_name, bucket_name, trigger_name, \
8         BY_TIME, prim_meta, hints=re_exec_rules)
```

Fig. 7. Configuring a bucket trigger to periodically invoke a function in a stream processing workflow.

triggers on the buckets using the primitives described in §III-B. Functions can then interact with the buckets by creating, sending and getting objects using the APIs listed in Table II.

As an example, we refer to a stream processing workflow [24] as shown in Fig. 4 (right). This workflow first filters the incoming advertisement events (i.e., `preprocess`) and checks which campaign each event belongs to (i.e., `query_event_info`). It then stores the returned results into a bucket and periodically invokes a function (i.e., `aggregate`) to count the events per campaign every second. Fig. 7 gives a code snippet of configuring a bucket trigger that periodically invokes the `aggregate` function, where a `ByTime` trigger is created with the primitive metadata that specifies both the target function and the triggering time window (line 4). Developers can optionally specify a re-execution rule in case of function failures, e.g., by re-executing the `query_event_info` function if the bucket has not received `query_event_info`'s output in 100 ms (line 5). We will describe the fault tolerance and re-execution in §IV-D. A full script of deploying this workflow is given in our technical report [45].

To summarize, our data bucket abstraction, trigger primitives, and programming interface facilitate the data-centric function orchestration, and enable developers to conveniently implement their application workflows and express various types of data patterns and function invocations. In addition, the unified programming interface also obviates the need to make an ad-hoc selection from many APIs provided by various external services, such as a message broker, in-memory database, and persistent storage.

IV. PHEROMONE DESIGN

This section presents the design of `Pheromone`, a new serverless platform that supports data-centric function orchestration.

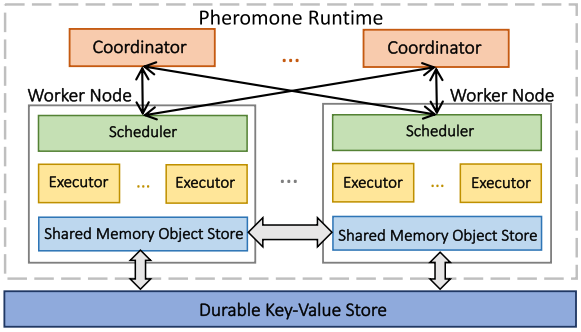


Fig. 8. An architecture overview of `Pheromone`.

A. Architecture Overview

`Pheromone` runs on a cluster of machines. Fig. 8 shows an architecture overview. Each worker node follows instructions from a local scheduler, and runs multiple executors that load and execute the user function code as needed. A worker node also maintains a shared-memory object store that holds the intermediate data generated by functions. The object store provides a data bucket interface through which functions can efficiently exchange data within a node and with other nodes. It also synchronizes data that must persist with a remote durable key-value store, such as Anna [46]. When new data are put into the object store, the local scheduler checks the associated bucket triggers. If the triggering conditions are satisfied, the local scheduler invokes the target function(s) either locally, or remotely with the help of a global coordinator that runs on a separate machine and performs cross-node coordination with a global view of bucket statuses.

B. Function Request Scheduling

In this section, we first introduce the scheduling problem and algorithm of `Pheromone`, and then illustrate how we realize scalable distributed scheduling.

Scheduling algorithm. `Pheromone` aims to exploit the data locality of function execution for enhanced workflow performance. Therefore, the objective of scheduling is to minimize the amount of data transmitted across worker nodes, thereby improving the overall efficiency of function interactions. Let $M = \{1, \ldots, m\}$ be a batch of function requests that are required to be scheduled onto a list of worker nodes $N = \{1, \ldots, n\}$. Let e_i be a number of idle function executors at node i. The total number of available executors is sufficient to accommodate all function requests, i.e., $\sum_i e_i \geq m$. For

Algorithm 1 Scheduling Algorithm

– M: a batch of function requests
– N: a list of worker nodes
– e_i: available executors at node i
– d_i^j: amount of data accessed by request j at node i

```
1:  function SCHEDULE
2:      T ← list of nodes in N with executors available, i.e., e_i > 0
3:      for request j ∈ M do
4:          t ← Null
5:          s ← −∞
6:          for node i ∈ T do
7:              if d_i^j > s then
8:                  s ← d_i^j
9:                  t ← i
10:         Schedule request j to node t
11:         e_t ← e_t − 1
12:         if e_t = 0 then
13:             remove node t from T
```

each node i, we denote d_i^j to be its amount of data accessed by request j during execution. Note that $\{d_i^j\}$ can be easily obtained before execution in our data-centric function orchestration. We define a binary variable x_i^j to indicate whether request j is scheduled to node i, i.e., 1 if scheduled and 0 otherwise. We formalize the scheduling problem as follows.

$$\begin{aligned} \max \quad & \sum_i \sum_j d_i^j x_i^j \\ \text{s.t.} \quad & \sum_i x_i^j = 1, \quad \forall j \in M \\ & \sum_j x_i^j \le e_i, \quad \forall i \in N \\ & x_i^j \in \{0,1\} \end{aligned} \tag{1}$$

The objective of Equation 1 is to schedule all requests in the batch (x_i^j) to maximize the amount of local data, which in turn minimizes cross-node data transmission. The constraints ensure that each function request is scheduled to exactly one node and that the number of scheduled requests at each node does not exceed its number of idle executors. Clearly, finding the optimal solution to this problem is NP-hard, which is a variant of the bin packing problem. We therefore resort to a greedy scheduling algorithm.

Algorithm 1 presents the scheduling design of `Pheromone`. In particular, it keeps track of a list of worker nodes with available executors (line 2), and iterates over all function requests in a batch and schedules each request to the node with the most data to be accessed (line 3-10). The algorithm then updates the number of available executors at the scheduled node (line 11) and removes the node from the list if it has no more available executors (line 12-13).

We realize the scheduling algorithm following a two-tier, distributed system design to deliver low-latency function invocations and ensure high scalability.

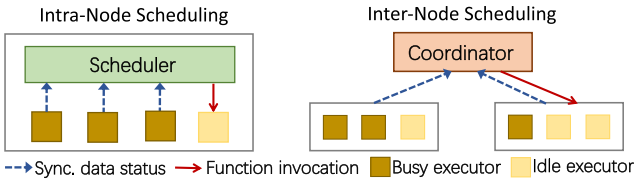


Fig. 9. Intra-node (left) and inter-node (right) scheduling.

Realization of distributed scheduling. In `Pheromone`, a workflow request first arrives at a global coordinator, which routes the request to a local scheduler on a worker node. The local scheduler invokes subsequent functions to locally execute the workflow whenever possible, thus reducing the invocation latency and incurring no network overhead.

Intra-node scheduling. The local scheduler uses bucket triggers to invoke functions *as locally as possible*. The scheduler starts the first function of a workflow and tracks its execution status via its bucket. The downstream functions are triggered immediately on the same node when their expected data objects are put into the associated buckets and ready to be consumed. As no cross-node communication is involved, it reduces the function invocation latency and enables efficient consumption of data objects in a local workflow execution. Fig 9 (left) shows how the local scheduler interacts with executors when running a workflow locally. The executors synchronize the data status (e.g., the readiness of local objects in buckets) with the local scheduler, which then checks the associated bucket triggers and invokes downstream functions if the triggering conditions are met. The low-latency message exchange between the scheduler and executors is enabled via an on-node shared-memory object store.

Local invocations occur when the node possesses all the data required by downstream functions, aligning with Algorithm 1. The scheduler simply routes subsequent requests to idle executors that have no running tasks, avoiding concurrent invocations and resource contention in each executor (similar to the concurrency model in AWS Lambda [47]). When the executor receives a request for the first time, it loads the function code from the local object store and persists it in memory for reuse in subsequent invocations. In case of multiple idle executors, the scheduler prioritizes those with function code already loaded to enable a warm start.[4]

If the requests received by a local scheduler exceed the capacity of local executors, the scheduler forwards them to a global coordinator, which routes them to other worker nodes with sufficient resources. Instead of forwarding the exceeding requests immediately, the scheduler waits for a configurable short time period: if any local executors become available during this period, the requested functions start and the requests are served locally. The rationale is that it typically takes little time for executors to become available as most serverless functions are short-lived [53], plus `Pheromone` has microsecond-scale invocation overhead (§VI-B). Such a

[4]Many techniques have been proposed to deal with cold starts of executors [48], [49], [50], [51], [52], [53], [54], which can be applied directly in `Pheromone`.

delayed scheduling has proven effective for improving data locality [55].

Inter-node scheduling. A global coordinator not only forwards requests from overloaded nodes to non-overloaded nodes, but also drives the execution of a large workflow which needs to run across multiple worker nodes that collectively host many functions of the workflow. This cannot be orchestrated by individual local schedulers without a global view.

As Fig. 9 (right) shows, a coordinator gathers the associated bucket statuses of the functions of a large workflow from multiple worker nodes, and triggers the next functions as needed. Each node immediately synchronizes local bucket status with the coordinator upon any change, such that the coordinator maintains an up-to-date global view. When the coordinator decides to trigger functions, it also updates this message to relevant workers, which reset local bucket status accordingly. This ensures a function invocation is neither missed nor duplicated. Note that, some bucket triggers (e.g., `ByTime`) can only be performed at the coordinator with its global view; here, worker nodes only update their local statuses to the coordinator without checking trigger conditions.

The coordinator employs Algorithm 1 in the inter-node scheduling to enhance data locality. It tracks the node-level knowledge from local schedulers, including the number of idle executors and the amount of intermediate data objects. When scheduling request batches, the coordinator can reduce data movement across nodes, thereby improving overall function interaction performance.

Scaling distributed scheduling with sharded coordinators. `Pheromone` employs a *shared-nothing* model to significantly reduce synchronization between local schedulers and global coordinators, thus attaining high scalability. Specifically, it partitions the workflow orchestration tasks across *sharded coordinators*, each of which manages a disjoint set of workflows. When executing a workflow, the responsible coordinator sends the relevant bucket triggers to a selected set of worker nodes and routes the invocation requests to them. A worker node may run functions of multiple workflows. For each workflow, its data and trigger status are synchronized with the responsible coordinator only. This design substantially reduces communication and synchronization overheads, and can be achieved by running a standard cluster management service (e.g., ZooKeeper [56], [57]) that deals with coordinator failures and allows a client to locate the coordinator of a specific workflow. The client can then interact with this coordinator to configure data triggers and send requests. This process is automatically done by the provided client library and is transparent to developers.

C. Bucket Management and Data Sharing

We next describe how `Pheromone` manages data objects in buckets, and enables fast data sharing between functions.

Bucket management. `Pheromone` uses an on-node shared-memory object store to maintain data objects, such that functions can directly access them via pointers (i.e., `EpheObject` in Table II). A data object is marked ready when the source function puts it into a bucket via `send_object()`. The bucket can be distributed across its responsible coordinator and a number of worker nodes, where each worker node tracks local data status while the coordinator holds a global view (§IV-B). Bucket status synchronization is only needed between the responsible coordinator and workers, as local statuses at different workers track their local objects only and are disjoint.

`Pheromone` garbage-collects the intermediate objects of a workflow execution after the associated invocation request has been *fully* served along the workflow. In case a workflow is executed across multiple worker nodes, the responsible coordinator notifies the local scheduler on each node to remove the associated objects from its object store.

When a worker node's local object store runs out of memory, a remote key-value store is used to hold the newly generated data objects at the expense of an increased data access delay.[5] Later, when more memory space is made available (e.g., via garbage collection), the node remaps the associated buckets to the local object store. In case a data object is lost due to system failures, `Pheromone` automatically re-executes the source function(s) to get it recovered (details in §IV-D).

Fast data sharing. `Pheromone` further adopts optimizations to fully reap the benefits of data locality enabled by its data-centric design. As intermediate data are typically short-lived and immutable [10], [38], we trade their durability for fast data sharing and low resource footprint. With an on-node shared-memory object store, `Pheromone` enables *zero-copy* data sharing between local functions by passing only the pointers of data objects to the target functions. This avoids the significant data copying and serialization overheads, and substantially reduces the latency of accessing local data objects.

To efficiently pass data to remote functions, `Pheromone` also enables the *direct* transfer of data objects between nodes. A function packages the metadata (e.g., locator) of a data object into a function request being sent to a remote node. The target function on the remote node uses such metadata to directly retrieve the required data object. Compared with using a remote storage for cross-node data sharing, our direct data transfer avoids unnecessary data copying, and thus leads to reduced network and storage overheads. While the remote-storage approach can ensure better data durability and consistency [13], [14], [58], there is no such need for intermediate data objects. Only when data are specified to persist will `Pheromone` synchronize data objects with a durable key-value store (see `send_object()` in Table II).

Note that, `Pheromone`'s data-centric design can expose details of intermediate data (e.g., the size of each data object), therefore we can further optimize cross-node data sharing. For large data objects, they are sent as raw byte arrays to avoid serialization-related overheads, thus significantly improving the performance of transferring large objects (see Fig. 13 in §VI-B). For small data objects, Pheromone implements a shortcut to transfer them between nodes: it piggybacks small

[5]Our current implementation does not support spilling in-memory objects to disk, which we leave for future work.

objects on the function invocation requests forwarded during the inter-node scheduling (see §IV-B). This shortcut saves one round trip as the target function does not need to additionally retrieve data objects from the source function. In addition, Pheromone runs an I/O thread pool on each worker node to improve cross-node data sharing performance.

D. Fault Tolerance

Pheromone sustains various types of system component failures. In case an executor fails or a data object is lost, Pheromone restarts the failed function to reproduce the lost data and resume the interrupted workflow. This is enabled by using the data bucket to re-execute its source function(s) if the expected output has not been received in a configurable timeout. This fault handling approach is a natural fit for data-centric function orchestration and brings two benefits. First, it can simplify the scheduling logic as data buckets can autonomously track the runtime status of each function and issue re-execution requests whenever necessary, without needing schedulers to handle function failures. Second, it allows developers to customize function re-execution rules when configuring data buckets, e.g., timeout. Fig. 7 gives an example of specifying re-execution rules (line 5). Fig. 5 shows the interface to implement the logic of function re-execution for a bucket trigger (`notify_source_func` and `action_for_rerun`).

Pheromone also checkpoints the scheduler state (e.g., the workflow status) to the local object store, so that it can quickly recover from a scheduler failure on a worker node. In case that an entire worker node crashes, Pheromone re-executes the failed workflows on other worker nodes. Pheromone can also handle failed coordinators with a standard cluster management service, such as ZooKeeper, as explained in §IV-B.

V. Implementation

We have implemented Pheromone atop Cloudburst [14], a lightweight, performant serverless platform. We heavily re-architected Cloudburst and implemented Pheromone's key components (Fig. 8) in 5k lines of C++ code. These components were packaged into Docker [59] images for ease of deployment. Pheromone's client was implemented in 400 lines of Python code. Like Cloudburst, Pheromone runs in a Kubernetes [60] cluster for convenient container management, and uses Anna [46], [61], an autoscaling key-value store, as the durable key-value storage. On each worker node, we mount a shared in-memory volume between containers for fast data exchange and message passing. The executor loads function code as dynamically linked libraries, which is pre-compiled by developers and uploaded to Pheromone. The entire codebase of Pheromone is open-sourced at [62].

VI. Evaluation

In this section, we evaluate Pheromone via a cluster deployment in AWS EC2. Our evaluation answers three questions:

- How does Pheromone improve function interactions (§VI-B) and ensure high scalability (§VI-C)?
- Can Pheromone effectively handle failures (§VI-D)?
- Can developers easily implement real-world applications with Pheromone and deliver high performance (§VI-E)?

A. Experimental Setup

Cluster settings. We deploy Pheromone in an EC2 cluster. The coordinators run on the `c5.xlarge` instances, each with 4 vCPUs and 8 GB memory. Each worker node is a `c5.4xlarge` instance with 16 vCPUs and 32 GB memory. The number of executors on a worker node is configurable and we tune it based on the requirements of our experiments. We deploy up to 8 coordinators and 51 worker nodes, and run clients on separate instances in the same `us-east-1a` EC2 zone.

Baselines. We compare Pheromone with four baselines.

1) Cloudburst: As an open-source platform providing fast state sharing, Cloudburst [14] adopts *early binding* in scheduling: it schedules all functions of a workflow before serving a request, and enables direct communications between functions. It also uses function-collocated caches. As Pheromone's cluster setting is similar to Cloudburst's, we deploy the two platforms using the same cluster configuration and resources.

2) KNIX: As an evolution of SAND [12], KNIX [13] improves the function interaction performance by executing functions of a workflow as processes in the same container. Message passing and data sharing can be done either via a local message bus or via a remote persistent storage.

3) AWS Step Functions (ASF): We use ASF Express Workflows [36] to orchestrate function instances as it achieves faster function interactions than the ASF Standard Workflows [36]. As ASF has a size limit of transferring intermediate data (see Fig. 2), we use Redis [20], a fast in-memory storage service, to share large data objects between functions.

4) Azure Durable Functions (DF): Compared with ASF, DF provides a more flexible support for function interactions. It allows developers to express workflows in program code and offers the Entity Functions [34] that can manage workflow states following the actor model [63], [64]. We include DF to study whether this expressive orchestration approach can achieve satisfactory performance.

Here, Cloudburst, KNIX and ASF focus more on optimizing function interactions of a workflow, while DF provides rich expressiveness. Note that, for the two commercial platforms, i.e., ASF and DF, we cannot control their orchestration runtime. To make a fair comparison, we configure their respective Lambda and Azure functions such that the number of function instances matches that of executors in Pheromone. The resource allocations of each function instance and executor are also maintained the same. In our experiments, functions are all warmed up to avoid cold starts in all platforms.

B. Function Interaction

Function invocation under various patterns. We first evaluate the overhead of invoking no-op functions without any payload. We consider three common invocation patterns:

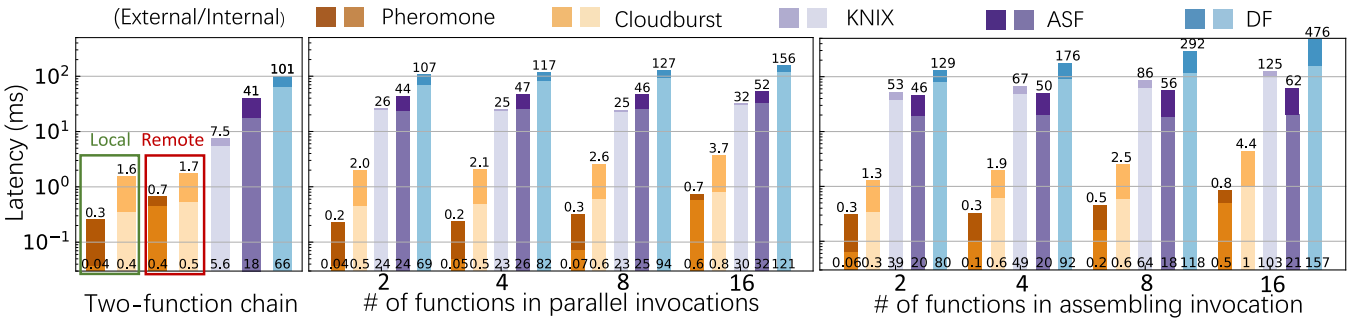


Fig. 10. Latencies of invoking no-op functions under three interaction patterns: function chain, parallel and assembling invocations. Each bar is broken into two parts which measure the latencies of external (darker) and internal (lighter) invocations, respectively. The overall latency value is given at the top of the bar, and the internal invocation latency is given at the bottom.

sequential execution (e.g., a two-function chain), parallel invocation (fan-out), and assembling invocation (fan-in). We vary the number of involved functions for parallel and assembling invocations to control the degree of parallelism. Fig. 10 shows the latencies of invoking no-op functions under these three patterns. Each latency bar is further broken down into the overheads of external and internal invocations. The former measures the latency between the arrival of a request and the complete start of the workflow, and the latter measures the latency of internally triggering the downstream function(s) following the designated pattern. In Pheromone, the external invocation latency is mostly due to the overhead of request routing which takes about 200 μs [65]. Note that, functions can be invoked locally or remotely in Pheromone and Cloudburst, thus we measure them respectively in Fig. 10. In our experiments, we report the average latency over 10 runs.

Fig. 10 (left) compares the invocation latencies of a two-function chain measured on five platforms. Pheromone substantially outperforms the others. In particular, Pheromone's shared memory-based message passing (§IV-C) only incurs an overhead of less than 20 μs, reducing the local invocation latency to 40 μs, which is 10× faster than Cloudburst. The latency improvements become significantly more salient compared with other platforms (e.g., 140× over KNIX, 450× over ASF). When invoking a remote function, both Pheromone and Cloudburst require network transfer, leading to a similar internal invocation latency. Yet, Cloudburst incurs higher overhead than Pheromone for external invocations as it needs to schedule the entire workflow's functions before serving a request (early binding), thus resulting in worse overall performance.

Fig. 10 (center) and (right) show the invocation latencies under parallel and assembling invocations, respectively. We also evaluate the cross-node function invocations in Pheromone and Cloudburst by configuring 12 executors on each worker, thus forcing remote invocations when running 16 functions. Pheromone constantly achieves the best performance and incurs only sub-millisecond latencies in all cases, even for cross-node function invocations. In contrast, Cloudburst's early-binding design incurs a much longer latency for function invocations as the number of functions increases. Both KNIX and ASF incur high invocation overheads in the parallel and assembling scenarios. DF yields the worst performance, and we exclude it from the experiments hereafter.

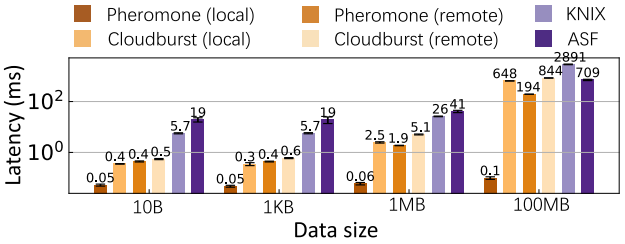


Fig. 11. Latencies of a two-function chain invocation under various data sizes.

Data transfer. We next evaluate the interaction overhead when transferring data between functions. Fig. 11 shows the invocation latencies of a two-function chain with various data sizes in Pheromone, Cloudburst, KNIX, and ASF. We evaluate both local and remote data transfer for Pheromone and Cloudburst. For KNIX and ASF where the data transfer can be done via either a workflow or a shared storage (i.e., Riak and Redis), we report the best of the two choices.

For local data transfer, Pheromone enables zero-copy data sharing, leading to extremely low overheads regardless of the data size, e.g., 0.1 ms for 100 MB data. In comparison, Cloudburst needs the data copying and serialization, causing much longer latencies especially for large data objects. For remote data transfer, both Pheromone and Cloudburst support direct data sharing across worker nodes. Pheromone employs an optimized implementation without (de)serialization, making it more efficient than Cloudburst. Collectively, compared with Pheromone, the serialization overhead in Cloudburst dominants the latencies of both local and remote invocations under large data exchanges, which diminishes the performance benefits of data locality: saving the cost of transferring 100 MB data across network only reduces the latency from 844 ms to 648 ms. Fig. 11 also shows that KNIX and ASF incur much longer latencies. While KNIX outperforms ASF when data objects are small, ASF becomes more efficient for passing large data because it is configured in our experiments to use the fast Redis in-memory storage for large data transfer.

We further evaluate the overhead of data transfer under parallel and assembling invocations. For parallel invocation,

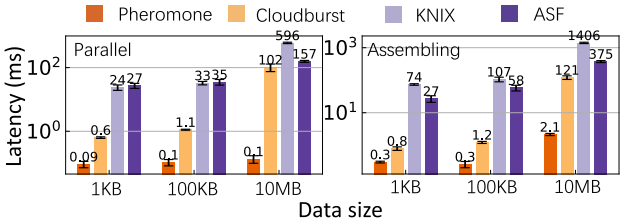


Fig. 12. Latencies of parallel (left) and assembling (right) invocations under various data sizes, using 8 functions.

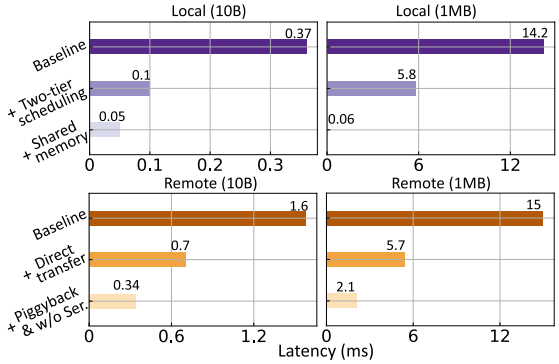


Fig. 13. Improvement breakdown for local (top) and remote (bottom) invocations. Each case includes transferring 10 B (left) and 1 MB (right) of data in function invocations.

we measure the latency of a function invoking parallel downstream functions and passing data to all of them; for assembling invocation, we measure the latency between the transfer of the first object and the reception of all objects in the assembling function. Fig. 12 shows the latencies of these two invocation patterns under various data sizes. Similarly, Pheromone constantly achieves faster data transfer compared with all other platforms for both invocation patterns.

Improvement breakdown. To illustrate how each of our individual designs contributes to the performance improvement, we break down Pheromone's function invocation performance and depict the results in Fig. 13. Specifically, for local invocation, "Baseline" uses a central coordinator to invoke downstream functions (i.e., no local schedulers), which is today's common practice [19]; "Two-tier scheduling" uses our local schedulers for fast invocations on the same worker node (§IV-B), where intermediate data objects are cached in the scheduler's memory and get copied to next functions; "Shared memory" further optimizes the performance via zero-copy data sharing (§IV-C). Fig. 13 (top) shows that applying two-tier scheduling can reduce network round trips and achieve up to 3.7× latency improvement over "Baseline". Applying shared memory avoids data copy and serialization, further speeding up the data exchange especially for large objects (e.g., 1 MB) by two orders of magnitude.

For remote invocation, "Baseline" uses a durable key-value store (i.e., Anna [46]) to exchange intermediate data among cross-node functions; "Direct transfer" reduces the communication overhead by allowing direct data passing between nodes (§IV-C), where raw data objects on a node are serialized into a protobuf [66] message and then sent to downstream functions; "Piggyback & w/o Ser." further optimizes the data exchange

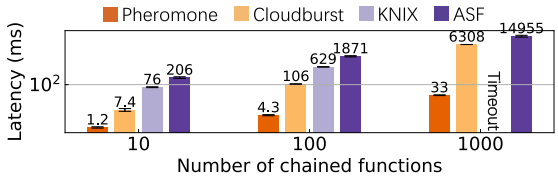


Fig. 14. Latencies of function chains of different lengths.

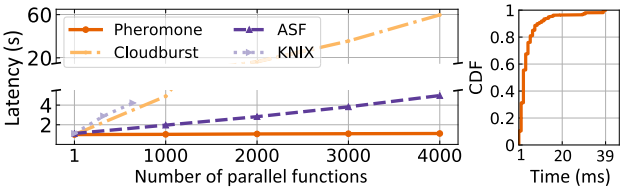


Fig. 15. End-to-end latencies with various numbers of parallel functions (left), and the distribution of function start times when executing 4k functions in Pheromone (right).

by piggybacking small objects on forwarded function requests and eliminating serialization (§IV-C). As shown in Fig. 13 (bottom), direct data transfer avoids interactions with the remote storage and improves the performance by up to 2.6× compared with baseline. The piggyback without serialization further speeds up the remote invocations with small (10 B) and large (1 MB) objects by 2× and 2.7×, respectively.

C. Scalability

We next evaluate the scalability of Pheromone with regard to internal function invocations and external user requests.

Long function chain. We start with a long function chain that sequentially invokes a large number of functions [67]. Here, each function simply increments its input value by 1 and sends the updated value to the next function, and the final result is the total number of functions. As shown in Fig. 14, we change the number of chained functions, and Pheromone achieves the best performance at any scale. Moreover, Cloudburst suffers from poor scalability due to its early-binding scheduling, causing significantly longer latencies when the number of chained functions increases; KNIX cannot host too many function processes in a single container, making it ill-suited for long function chains; ASF incurs the longest latencies due to its high overhead of function interactions.

Parallel functions. Fig. 15 (left) evaluates the end-to-end latencies of invoking various numbers of parallel functions, where each function sleeps 1 second. We run 80 function executors per node in Pheromone and Cloudburst. Pheromone only incurs a negligible latency in large-scale parallel executions, while ASF and Cloudburst incur much higher latencies, e.g., seconds or tens of seconds. KNIX suffers from severe resource contention when running all workflow functions in the same container, and fails to support highly parallel function executions. To further illustrate Pheromone's behavior in parallel invocations, Fig. 15 (right) shows the distribution of function start times, where Pheromone can quickly launch all 4k functions within 40 ms.

User request throughput. Fig. 16 shows the user request throughput when serving requests to no-op functions using

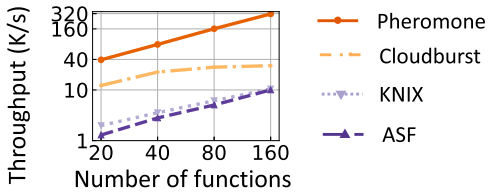


Fig. 16. Request throughput when serving requests to no-op functions under various numbers of functions or executors.

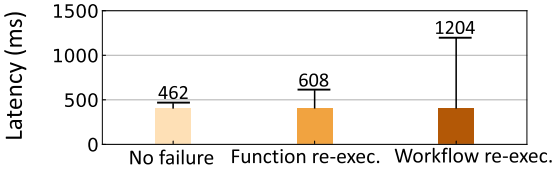


Fig. 17. Median and 99^{th} tail latencies of a four-function workflow with no failure, function- and workflow-level re-executions. The numbers indicate the tail latencies.

various numbers of executors. We configure 20 executors on each node in Pheromone and Cloudburst. We observe that Cloudburst's schedulers can easily become the bottleneck under a high request rate, making it difficult to fully utilize the executor's resources; KNIX suffers from a similar problem that limits its scalability. While ASF has no such an issue, it leads to low throughput due to its high invocation overhead (Fig. 10). Compared with these platforms, Pheromone ensures better scalability with the highest throughput.

D. Fault Tolerance

We evaluate Pheromone's fault tolerance mechanism (§IV-D). We execute a workflow that chains four sleep functions (each sleeps 100 ms), and each running function is configured to crash at a probability of 1%. Fig. 17 shows the median and 99^{th} tail latencies of the workflow over 100 runs using Pheromone's function- and workflow-level re-executions after a configurable timeout. In particular, the timeout is configured as twice of the normal execution, i.e., 200 ms for each individual function and 800 ms for the workflow. We also include the normal scenario where no failure occurs. Compared with the common practice of workflow re-execution, Pheromone's data-centric mechanism allows finer-grained, function-level fault handling, which cuts the tail latency of the workflow from 1204 ms to 608 ms, thus significantly reducing the re-execution overhead.

E. Case Studies

We evaluate three representative applications atop Pheromone: Yahoo's streaming benchmark for advertisement events [24], a real-time query task, and a data-intensive MapReduce sort.

Advertisement event stream processing. This application filters incoming advertisement events (e.g., click or purchase), checks which campaign each event belongs to, stores them into storage, and periodically counts the events per campaign every second. As shown in Fig. 1 (right) and discussed in §II-B, the key to enabling this application in serverless platforms is to periodically invoke a function to process the events

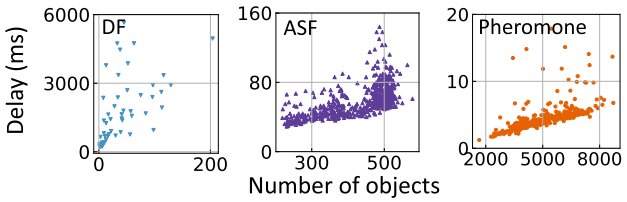


Fig. 18. Delays of accessing the accumulated data objects in the advertisement event stream processing. Lower delays and more objects are better.

accumulated during the past one second. In Pheromone, this is straightforward by using the ByTime primitive (§III-C and Fig. 7). This application can also be implemented easily in DF by using an addressable Entity function for aggregation [68]. However, it is non-trivial in ASF and we have to resort to a "serverful" workaround: one workflow does the regular "filter-check-store" for each event and sends the event ID to an external, serverful coordinator; a separate workflow is set up to get triggered every second by the accumulated event IDs sent from the external coordinator, so that it can access and count the associated events per campaign.

Fig. 18 compares the performance on Pheromone, ASF, and DF. We measure the delays of accessing accumulated data objects (i.e., advertisement events), where the lower delays and more objects are better. For DF, data are not accessed in batches, and thus we measure the queuing delay between the reset request being issued and the Entity function receiving it. We use up to 40 functions in all these platforms. DF results in a significant overhead with high and unstable queuing delays, as its Entity function can easily become a bottleneck. Among the three platforms, Pheromone performs the best: it can access substantially more accumulated data objects in a much smaller delay. In summary, Pheromone not only simplifies the design and deployment for such a stream processing application, but also delivers high performance.

Real-time query. To evaluate Pheromone's ability to efficiently support latency-critical tasks, we employ a real-time query application as a representative example. This application workflow consists of three functions. The first function accepts user queries and extracts the data field to be retrieved from the data store. It then passes the extracted information to the second function, which searches a local in-memory data store for the corresponding value associated with the requested field. Finally, the third function returns a formatted message to the user with the retrieved data value.

We deploy the real-time query workflow on Pheromone, Cloudburst, KNIX, and ASF, and compare their performance in Fig. 19 (left). Among all the platforms, Pheromone achieves the best performance and reduces the end-to-end latency by 3.5×, 23×, and 79× compared with Cloudburst, KNIX, and ASF, respectively. We further break down the end-to-end latency into function execution time and interaction overhead, as shown in Fig. 19 (right). The numbers indicate the latency of function interaction. Pheromone incurs only microsecond-scale interaction overheads compared to Cloudburst, resulting in significantly lower end-to-end latency.

MapReduce sort. We next evaluate how Pheromone's data-centric orchestration can easily facilitate MapReduce sort,

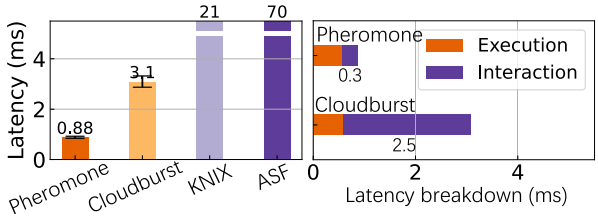


Fig. 19. Latencies of the real-time query workflow on various platforms (left) and the latency breakdown (right).

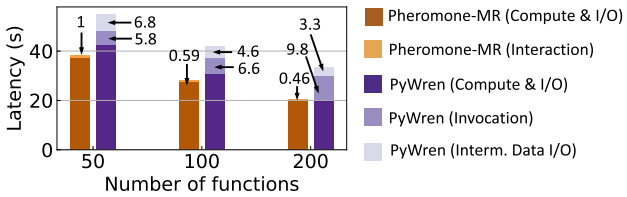


Fig. 20. Latencies of sorting 10 GB data using various numbers of functions on PyWren and Pheromone-MR. The latency is broken down into: the interaction latency (for PyWren, the invocation and intermediate data I/O), and the latency for compute and I/O. The numbers indicate the former.

a typical data-intensive application. We have built a MapReduce framework atop Pheromone, called Pheromone-MR. Using the DynamicGroup primitive, Pheromone-MR can be implemented in only 500 lines of code, and developers can program standard mapper and reducer [69] without operating on intermediate data (§III-B). We compare Pheromone-MR with PyWren [2], a specialized serverless analytics system built atop AWS Lambda. Compared with Pheromone-MR, PyWren is implemented in about 6k lines of code and supports the map operator only, making it complicated to deploy a MapReduce application: developers need to explicitly transfer the intermediate data via a shared storage (e.g, Redis) to simulate the reducer, and carefully configure the storage cluster for improved data exchange. Even with these optimizations, PyWren still suffers from limited performance (and usability).

We evaluate the performance of Pheromone-MR and PyWren with MapReduce sort over 10 GB data, where 10 GB intermediate objects are generated in the shuffle phase. We allocate each Pheromone executor and each Lambda instance the same resource, e.g., 1 vCPU. We also configure a Redis cluster for PyWren to enable fast data exchange. We measure the end-to-end latencies on Pheromone-MR and PyWren using various numbers of functions, and break down the results into the function interaction latency and the latency for compute and I/O. The former measures the latency between the completion of mappers and the start of reducers. For PyWren, the interaction latency consists of two parts: 1) the invocation latency of triggering all reducers after mappers return, and 2) the I/O latency of sharing intermediate data via Redis. As shown in Fig. 20, running more functions in PyWren improves the I/O of sharing intermediate data, but results in a longer latency in parallel invocations. Compared with PyWren, Pheromone-MR has a significantly lower interaction latency (e.g., less than 1s), thus improving the end-to-end performance by up to 1.6×.

We note that, the limitations of AWS Lambada make PyWren less efficient. First, because Lambda does not support large-scale map by default [43], it needs to implement this operation but in an inefficient way which incurs high invocation overheads. Second, Lambda has a limited support for data sharing, forcing developers to explore an external alternative that incurs high overheads even though using a fast storage (i.e., Redis). Unlike AWS Lambda, Pheromone supports rich patterns of function executions while enabling fast data sharing, such that developers can easily build a MapReduce framework and achieve high performance.

VII. Discussion and Related Work

Isolation in Pheromone. Pheromone provides the container-level isolation between function invocations, while functions running on the same worker node share in-memory data objects (§IV-C). Commercial container-based serverless platforms often do not co-locate functions from different users to enhance security [70]. In this setting, functions on the same worker node can be trusted; hence, it is safe to trade strict isolation for improved I/O performance. We notice that current serverless platforms have made various trade-offs between performance and isolation. For example, AWS Lambda runs functions in MicroVMs for strong isolation [48]; KNIX isolates a workflow's functions using processes in the same container for better performance [12]; recent work proposes lightweight isolation for privacy-preserving serverless applications [71]. Pheromone can explore these different trade-offs, which we leave for future work.

Supported languages. Pheromone currently supports functions written in C++, but it can be straightforward to support other programming languages. Specifically, Pheromone's executor exposes data trigger APIs (Tabel II) and interacts with other system components, and can serve as a proxy for functions written in different languages. That being said, Pheromone's optimization on fast data exchange via shared memory may not apply to all language runtimes – only those allowing the direct consumption of byte arrays without (de)serialization, e.g., Python ctype, can benefit from zero-copy data sharing. The other Pheromone designs are still effective regardless of language runtimes.

Data exchange in serverless platforms. Data exchange is a common pain point in today's serverless platforms. One general approach is to leverage shared storage to enable and optimize data exchange among functions [3], [5], [10], [72], [73], [74], [75]. One other approach is to exploit data locality to improve performance, e.g., by placing workflow functions on a single machine [14], [15], [16], [23], [38], [58]. Moreover, OFC [76] and Faa$T [77] provide the autoscaling cache for individual applications. Shredder [78] and Zion [79] push the function code into storage. Wukong [4] enhances the locality of DAG-based parallel workloads at the application level. Lambdata [38] makes the intent of a function's input and output explicit for improved locality; however, it does not provide a unified programming interface for expressive and simplified function interactions, and its performance is heavily bound to Apache OpenWhisk [15], [80].

VIII. Conclusion

This paper revisits the function orchestration in serverless computing, and advocates a new design paradigm that a serverless platform needs to: 1) break the tight coupling between function flows and data flows, 2) allow fine-grained data exchange between functions of a workflow, and 3) provide a unified and efficient approach for both function invocations and data exchange. With this data-centric paradigm, we have designed and developed `Pheromone`, a new serverless platform which achieves all the desired properties, namely, rich expressiveness, high usability, and wide applicability. `Pheromone` is open-sourced, and outperforms current commercial and open-source serverless platforms by orders of magnitude in terms of the latencies of function invocation and data exchange.

References

[1] *Serverless Applications Scenarios*. Accessed: May 1, 2024. [Online]. Available: https://docs.aws.amazon.com/wellarchitected/latest/serverless-applications-lens/scenarios.html

[2] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in *Proc. ACM SoCC*, 2017, pp. 445–451.

[3] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, fast and slow: Scalable analytics on serverless infrastructure," in *Proc. USENIX NSDI*, 2019, pp. 193–206.

[4] B. Carver, J. Zhang, A. Wang, A. Anwar, P. Wu, and Y. Cheng, "Wukong: A scalable and locality-enhanced framework for serverless parallel computing," in *Proc. ACM SoCC*, 2021, pp. 1–15.

[5] S. Fouladi et al., "Encoding, fast and slow: Low-latency video processing using thousands of tiny threads," in *Proc. USENIX NSDI*, 2017, pp. 363–376.

[6] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter, "Sprocket: A serverless video processing framework," in *Proc. ACM SoCC*, 2018, pp. 263–274.

[7] M. Yu, Z. Jiang, H. C. Ng, W. Wang, R. Chen, and B. Li, "Gillis: Serving large neural networks in serverless functions with automatic model partitioning," in *Proc. IEEE ICDCS*, Jul. 2021, pp. 138–148.

[8] C. Zhang, M. Yu, W. Wang, and F. Yan, "MArk: Exploiting cloud services for cost-effective, SLO-aware machine learning inference serving," in *Proc. USENIX ATC*, 2019, pp. 1049–1062.

[9] H. Tian, S. Li, A. Wang, W. Wang, T. Wu, and H. Yang, "Owl: Performance-aware scheduling for resource-efficient function-as-a-service cloud," in *Proc. ACM SoCC*, 2022, pp. 78–93.

[10] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic ephemeral storage for serverless analytics," in *Proc. USENIX OSDI*, 2018, pp. 427–444.

[11] H. Zhang, Y. Tang, A. Khandelwal, J. Chen, and I. Stoica, "Caerus: NIMBLE task scheduling for serverless analytics," in *Proc. USENIX NSDI*, 2021, pp. 653–669.

[12] I. E. Akkus et al., "SAND: Towards high-performance serverless computing," in *Proc. USENIX ATC*, 2018, pp. 923–935.

[13] *KNIX Serverless*. Accessed: May 1, 2024. [Online]. Available: https://github.com/knix-microfunctions/knix/

[14] V. Sreekanti et al., "Cloudburst: Stateful functions-as-a-service," in *Proc. VLDB Endowment*, 2020, pp. 1–15.

[15] S. Kotni, A. Nayak, V. Ganapathy, and A. Basu, "Faastlane: Accelerating function-as-a-service workflows," in *Proc. USENIX ATC*, 2021, pp. 805–820.

[16] A. Mahgoub, K. Shankar, S. Mitra, and A. Klimovic, "SONIC: Application-aware data passing for chained serverless applications," in *Proc. USENIX ATC*, 2021, pp. 285–301.

[17] *Apache OpenWhisk Composer*. Accessed: May 1, 2024. [Online]. Available: https://github.com/apache/openwhisk-composer/

[18] *Google Cloud Composer*. Accessed: May 1, 2024. [Online]. Available: https://cloud.google.com/composer

[19] *AWS Step Functions*. Accessed: May 1, 2024. [Online]. Available: https://aws.amazon.com/step-functions/

[20] *AWS ElastiCache*. Accessed: May 1, 2024. [Online]. Available: https://aws.amazon.com/elasticache/

[21] *AWS S3*. Accessed: May 1, 2024. [Online]. Available: https://aws.amazon.com/s3/

[22] *Use Amazon S3 ARNs Instead of Passing Large Payloads*. Accessed: May 1, 2024. [Online]. Available: https://docs.aws.amazon.com/step-functions/latest/dg/avoid-exec-failures.html

[23] Z. Jia and E. Witchel, "Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices," in *Proc. ACM ASPLOS*, 2021, pp. 152–166.

[24] S. Chintapalli et al., "Benchmarking streaming computation engines: Storm, flink and spark streaming," in *Proc. IEEE IPDPSW*, May 2016, pp. 1789–1792.

[25] *AWS Lambda*. Accessed: May 1, 2024. [Online]. Available: https://aws.amazon.com/lambda/

[26] *Azure Functions*. Accessed: May 1, 2024. [Online]. Available: https://azure.microsoft.com/en-us/services/functions/

[27] *Google Cloud Functions*. Accessed: May 1, 2024. [Online]. Available: https://cloud.google.com/functions

[28] E. Jonas et al., "Cloud programming simplified: A Berkeley view on serverless computing," 2019, *arXiv:1902.03383*.

[29] J. Schleier-Smith et al., "What serverless computing is and should become: The next phase of cloud computing," *Commun. ACM*, vol. 64, no. 5, pp. 76–84, Apr. 2021.

[30] L. Xu, S. Venkataraman, I. Gupta, L. Mai, and R. Potharaju, "Move fast and meet deadlines: Fine-grained real-time stream processing with cameo," in *Proc. USENIX NSDI*, 2021.

[31] *AWS Kinesis*. Accessed: May 1, 2024. [Online]. Available: https://aws.amazon.com/kinesis/

[32] *Serverless Stream-based Processing for Real-Time Insights*. Accessed: May 1, 2024. [Online]. Available: https://aws.amazon.com/blogs/architecture/serverless-stream-based-processing-for-real-time-insights/

[33] *Serverless Reference Architecture: Real-time Stream Processing*. Accessed: May 1, 2024. [Online]. Available: https://github.com/aws-samples/lambda-refarch-streamprocessing/

[34] *Azure Entity Functions*. Accessed: May 1, 2024. [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-entities/

[35] *Invoking AWS Lambda Functions*. Accessed: May 1, 2024. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/lambda-invocation.html

[36] *AWS Step Functions Standard Vs. Express Workflows*. Accessed: May 1, 2024. [Online]. Available: https://docs.aws.amazon.com/step-functions/latest/dg/concepts-standard-vs-express.html

[37] *Using AWS Lambda With Amazon S3*. Accessed: May 1, 2024. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/with-s3.html

[38] Y. Tang and J. Yang, "Lambdata: Optimizing serverless computing by making data intents explicit," in *Proc. IEEE 13th Int. Conf. Cloud Comput. (CLOUD)*, Oct. 2020, pp. 294–303.

[39] *AWS Step Functions—Choice*. Accessed: May 1, 2024. [Online]. Available: https://docs.aws.amazon.com/step-functions/latest/dg/amazon-states-language-choice-state.html

[40] A. Vulimiri, O. Michel, P. B. Godfrey, and S. Shenker, "More is less: Reducing latency via redundancy," in *Proc. ACM HotNet*, 2012, pp. 13–18.

[41] K. V. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran, "EC-cache: Load-balanced, low-latency cluster caching with online erasure coding," in *Proc. USENIX OSDI*, 2016, pp. 401–417.

[42] J. Kosaian, K. V. Rashmi, and S. Venkataraman, "Parity models: Erasure-coded resilience for prediction serving systems," in *Proc. ACM SOSP*, 2019, pp. 30–46.

[43] *AWS Step Functions—Map*. Accessed: May 1, 2024. [Online]. Available: https://docs.aws.amazon.com/step-functions/latest/dg/amazon-states-language-map-state.html

[44] *Azure Durable Functions*. Accessed: May 1, 2024. [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/durable/

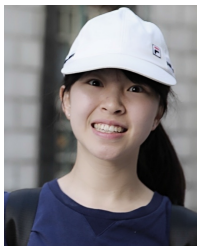
[45] M. Yu, T. Cao, W. Wang, and R. Chen, "Following the data, not the function: Rethinking function orchestration in serverless computing," 2021, *arXiv:2109.13492*.

[46] C. Wu, J. Faleiro, Y. Lin, and J. Hellerstein, "Anna: A KVS for any scale," in *Proc. IEEE 34th Int. Conf. Data Eng. (ICDE)*, Apr. 2018, pp. 401–412.

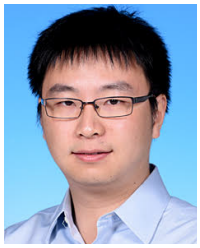
[47] *AWS Lambda Execution Model*. Accessed: May 1, 2024. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/runtimes-context.html

[48] A. Agache et al., "Firecracker: Lightweight virtualization for serverless applications," in *Proc. USENIX NSDI*, 2020, pp. 419–434.
[49] D. Du et al., "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting," in *Proc. ACM ASPLOS*, 2020, pp. 467–481.
[50] A. Wang et al., "FaaSNet: Scalable and fast provisioning of custom serverless container runtimes at Alibaba cloud function compute," in *Proc. USENIX ATC*, 2021, pp. 443–457.
[51] E. Oakes et al., "SOCK: Rapid task provisioning with serverless-optimized containers," in *Proc. USENIX ATC*, 2018, pp. 57–70.
[52] J. Cadden, T. Unger, Y. Awad, H. Dong, O. Krieger, and J. Appavoo, "SEUSS: Skip redundant paths to make serverless fast," in *Proc. ACM EuroSys*, 2020, pp. 1–15.
[53] M. Shahrad et al., "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proc. USENIX ATC*, 2020, pp. 205–218.
[54] A. Fuerst and P. Sharma, "FaasCache: Keeping serverless computing alive with greedy-dual caching," in *Proc. ACM ASPLOS*, 2021, pp. 386–400.
[55] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proc. ACM EuroSys*, 2010, pp. 265–278.
[56] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for internet-scale systems," in *Proc. USENIX ATC*, 2010, pp. 1–14.
[57] *Apache ZooKeeper*. Accessed: May 1, 2024. [Online]. Available: https://zookeeper.apache.org/
[58] S. Shillaker and P. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in *Proc. USENIX ATC*, 2020, pp. 419–433.
[59] *Docker*. Accessed: May 1, 2024. [Online]. Available: https://www.docker.com
[60] *Kubernetes: Production-grade Container Orchestration*. Accessed: May 1, 2024. [Online]. Available: http://kubernetes.io
[61] C. Wu, V. Sreekanti, and J. M. Hellerstein, "Autoscaling tiered cloud storage in anna," *Proc. VLDB Endow.*, vol. 12, no. 6, pp. 624–638, 2019.
[62] *Pheromone Codebase*. Accessed: May 1, 2024. [Online]. Available: https://github.com/MincYu/pheromone
[63] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin, "Orleans: Cloud computing for everyone," in *Proc. ACM SoCC*, 2011, pp. 1–14.
[64] P. Moritz et al., "Ray: A distributed framework for emerging AI applications," in *Proc. USENIX OSDI*, 2018, pp. 561–577.
[65] *Firecracker Network Performance Numbers*. Accessed: May 1, 2024. [Online]. Available: https://github.com/firecracker-microvm/firecracker/blob/main/docs/network-performance.md
[66] *Protocol Buffers—Google's Data Interchange Format*. Accessed: May 1, 2024. [Online]. Available: https://github.com/protocolbuffers/protobuf
[67] T. Yu et al., "Characterizing serverless platforms with serverlessbench," in *Proc. ACM SoCC*, 2020, pp. 30–44.
[68] *Azure Durable Functions Aggregator Pattern*. Accessed: May 1, 2024. [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview
[69] *Apache Hadoop*. Accessed: May 1, 2024. [Online]. Available: https://hadoop.apache.org
[70] *Alibaba Cloud Function Compute*. Accessed: May 1, 2024. [Online]. Available: https://www.alibabacloud.com/product/function-compute
[71] M. Li, Y. Xia, and H. Chen, "Confidential serverless made efficient with plug-in enclaves," in *Proc. ACM/IEEE ISCA*, Jun. 2021, pp. 306–318.
[72] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "Cirrus: A serverless framework for end-to-end ML workflows," in *Proc. ACM SoCC*, 2019, pp. 13–24.
[73] S. Fouladi et al., "From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers," in *Proc. USENIX ATC*, 2019, pp. 475–488.
[74] I. Müller, R. Marroquín, and G. Alonso, "Lambada: Interactive data analytics on cold data using serverless cloud infrastructure," in *Proc. ACM SIGMOD*, 2020, pp. 115–130.
[75] M. Perron, R. C. Fernandez, D. DeWitt, and S. Madden, "Starling: A scalable query engine on cloud functions," in *Proc. ACM SIGMOD*, 2020, pp. 131–141.
[76] D. Mvondo et al., "OFC: An opportunistic caching system for FaaS platforms," in *Proc. ACM EuroSys*, 2021, pp. 228–244.
[77] F. Romero et al., "Faa$t: A transparent auto-scaling cache for serverless applications," in *Proc. ACM SoCC*, 2021, pp. 122–137.
[78] T. Zhang, D. Xie, F. Li, and R. Stutsman, "Narrowing the gap between serverless and its state with storage functions," in *Proc. ACM SoCC*, 2019, pp. 1–12.
[79] J. Sampé, M. Sánchez-Artigas, P. García-López, and G. París, "Data-driven serverless functions for object storage," in *Proc. ACM/IFIP/USENIX Middleware*, 2017, pp. 121–133.
[80] *Apache OpenWhisk*. Accessed: May 1, 2024. [Online]. Available: http://openwhisk.apache.org/

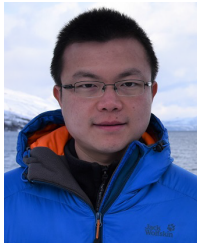
Minchen Yu (Member, IEEE) received the B.Eng. degree from Nanjing University in 2018 and the Ph.D. degree from The Hong Kong University of Science and Technology in 2023. He is currently an Assistant Professor with the School of Data Science, The Chinese University of Hong Kong, Shenzhen. He is also affiliated with Shenzhen Research Institute of Big Data. His research interests include cloud computing and distributed systems, with a special focus on serverless computing, big data, and machine learning systems. His work has received the Best Paper Runner-Up Award at IEEE ICDCS 2021.



Tingjia Cao is currently pursuing the Ph.D. degree with the Computer Sciences Department, University of Wisconsin–Madison, under the guidance of Prof. Andrea Arpaci-Dusseau and Prof. Remzi Arpaci-Dusseau. Her research interests include operating systems and scheduling.



Wei Wang (Member, IEEE) received the B.Engr. and M.Engr. degrees in electrical engineering from Shanghai Jiao Tong University in 2007 and 2010, respectively, and the Ph.D. degree in electrical and computer engineering from the University of Toronto in 2015. In 2015, he joined the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology (HKUST), where he is currently an Associate Professor. He is also affiliated with the HKUST Big Data Institute. His current research interests include distributed and cloud systems, with the recent focus on serverless computing, machine learning systems, and large-scale cluster management in production AI clouds. His research has received the Best Paper Award at ACM SoCC 2023 and two Best Paper Runner-Up Awards at IEEE ICDCS 2021 and USENIX ICAC 2013.



Ruichuan Chen (Member, IEEE) received the Ph.D. degree in computer science from Peking University, Beijing, China, in 2009. He was a Post-Doctoral Researcher with the Max Planck Institute for Software Systems, Kaiserslautern, Germany. He is currently a Distinguished Member of the Technical Staff and the Tech Lead of Nokia Bell Laboratories, Stuttgart, Germany. His work has been published at various prestigious venues, including SIGCOMM, NSDI, SOSP, OSDI, CoNEXT, and USENIX ATC. His current research interests include the areas of networking and distributed systems, in particular cloud computing and privacy-preserving technologies.