Following the Data, Not the Function:
Rethinking Function Orchestration in Serverless Computing

Minchen Yu[†]   Tingjia Cao[‡*]   Wei Wang[†]   Ruichuan Chen[§]

[†]Hong Kong University of Science and Technology
[‡]University of Wisconsin-Madison   [§]Nokia Bell Labs

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.

In this paper, 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.

1 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 [28, 35, 39, 43, 48, 59, 68, 74, 77]. 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 [49, 59, 78]; 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 [4, 11, 21, 24, 34, 51, 53, 66]. 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 [6, 10, 24, 31, 49, 66]. 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 §2.2). Current serverless platforms also incur function interaction latencies of tens of milliseconds, which may be unacceptable to latency-sensitive applications [46], 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 provides a rich set of data trigger primitives that developers can use to specify *when* and *how* the intermediate

1

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 [66], and KNIX [24]. Evaluation results show that Pheromone improves the function invocation latency by up to $10\times$ and $450\times$ 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 two serverless applications, i.e., Yahoo! stream processing [40] 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.

2 Background and Motivation

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

2.1 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 [8, 18, 22]. 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 [47, 63] – 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 [8,18]. Altogether, these benefits have increasingly driven a large number of traditional "serverful" applications to be migrated to the serverless platforms, including batch analytics [39,48,59,78], video processing [35,43], stream processing [28], machine learning [74,77], microservices [46], etc.

2.2 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) [4, 11, 21, 24, 34, 51, 53, 66], in which the nodes represent functions and the edges indicate the invocation dependencies between functions. The DAG can be specified using general programming languages [4, 21], or domain-specific languages such as Amazon States Language [11, 24]. 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 1. This is because the current function orchestration assumes that data flow in the same way as how
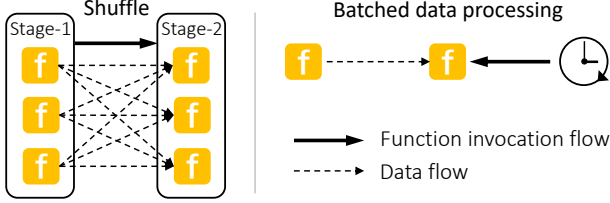
Figure 1: The shuffle operation (left) in data analytics and the batched data processing in a stream (right).

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.

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 [49, 59], 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 [40, 73], 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 [7]) to batch the data for periodic function invocations [28–30]. Note that, even with the latest stateful workflow (e.g., Azure Durable Functions [17]), an addressable function needs to keep running to receive data. As we will show in §6.5, 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 [6,10,24,31,49,66]. They can also process data from various sources, such as nested function calls, message queues, or other cloud services [23].

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 [53] 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 work-
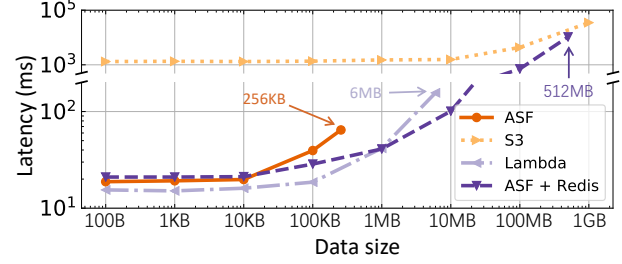

Figure 2: The interaction latency of two AWS Lambda functions under various data sizes using four approaches.

flow[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) [32]. 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 tradeoff between I/O performance and cost when using different storage to share intermediate data [49, 59], 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 (§6.2), 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 [46]. 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 [8, 24, 43, 46, 59, 66]. Altogether, the above characteristics substantially limit the applicability of current serverless platforms.

---

[2]We use the ASF Express Workflows in our experiments as it delivers higher performance than the ASF Standard Workflows [14].

3 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 §4.

3.1 Key Insight

As discussed in §2.2, 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 [49, 67]: 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 [49, 67], 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*).

---

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

Figure 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.

3.2 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 [29, 30] 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 [40, 73].
- `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 [12].
- `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

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

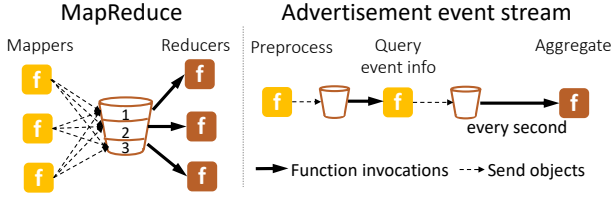| Invocation Patterns	ASF	Pheromone
Sequential Execution	Task	Immediate
Conditional Invocation	Choice	ByName
Assembling Invocation	Parallel	BySet
Dynamic Parallel	Map	DynamicJoin
Batched Data Processing	-	ByBatchSize ByTime
k-out-of-n	-	Redundant
MapReduce	-	DynamicGroup

Figure 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).

available and ready to be consumed. It can be used to execute redundant requests and perform late binding for straggler mitigation and improved reliability [50, 60, 69].

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 [13].
- 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.

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 [48, 49, 59], see §2.2). 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 abstrac-

```
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);
};
```

Figure 5: Three main methods of the trigger interface.

tion and trigger primitives. The design of Pheromone will be detailed in §4. Table 1 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 [15] can also achieve rich expressiveness for complex workflows (§6.1). Yet, it fails to achieve the other two desired properties, i.e., high usability and wide applicability (§6.5).

Abstract interface. Pheromone's trigger primitives are not only limited to those listed in Table 1. 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 §4.4. Appendix A.1 gives an example of implementing the ByBatchSize trigger primitive via the interface.

3.3 Programming Interface

Our Pheromone serverless platform currently accepts functions written in C++, with capabilities to support more languages (see §7). Pheromone also provides a Python client

Table 2: 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.

```
int handle(UserLibraryInterface* library,\
    int arg_size, char** arg_values);
```
Figure 6: Function interface.

```
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)
```

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

Figure 8: An architecture overview of Pheromone.

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 2) 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 triggers on the buckets using the primitives described in §3.2. Functions can then interact with the buckets by creating, sending and getting objects using the APIs listed in Table 2.

As an example, we refer to a stream processing workflow [40] 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 tar-
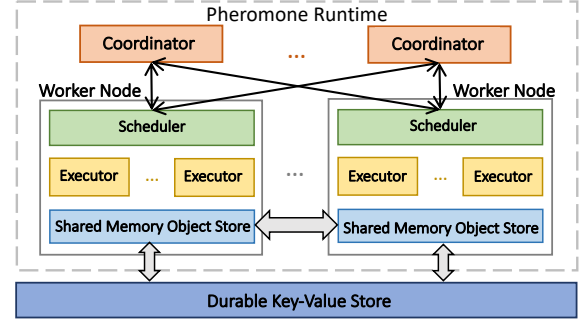
get function and the triggering time window (line 4). Developers can optionally specify a re-execution rule in case of function failures, e.g., by re-executing the query_event_info function if the bucket has not received query_event_info's output in 100 ms (line 5). We will describe the fault tolerance and re-execution in §4.4. Appendix A.2 gives a full script of deploying this workflow.

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.

4 Pheromone System Design

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

4.1 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 ef-
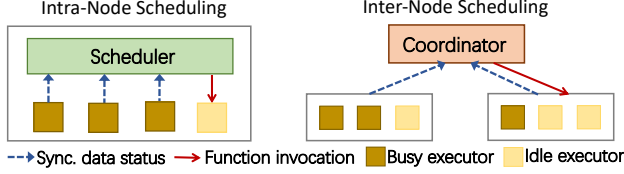
Figure 9: Intra-node (left) and inter-node (right) scheduling.

ficiently 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 [71]. 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.

4.2 Scalable Distributed Scheduling

We design a two-tier, distributed scheduling scheme to exploit data locality and ensure high scalability, enabled by the data-centric approach. Specifically, 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. In Pheromone, a 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.

A local scheduler makes scheduling decisions based on the status of executors. The scheduler only routes function 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 [9]). 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]

Delayed request forwarding from overloaded nodes. 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 [64], plus Pheromone has microsecond-scale invocation overhead (§6.2). Such a delayed scheduling has proven effective for improving data locality [76].

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 data-centric orchestration improves data locality in the inter-node scheduling. The coordinator makes scheduling decisions using the node-level knowledge reported by local schedulers, including cached functions, the number of idle executors, and the number of objects relevant to the workflow. It then schedules a request to a worker node with sufficient warm executors and the most relevant data objects.

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

---

[4]Many techniques have been proposed to deal with cold starts of executors [33, 37, 41, 44, 57, 64, 70], which can be applied directly in Pheromone.

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 [5, 45]) 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.

4.3 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 2). 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 (§4.2). 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. Appendix A.5 gives an evaluation of the garbage collection mechanism.

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 §4.4).

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 [49, 67], 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

---

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

---

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 [24, 65, 66], 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 2).

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 §6.2). For small data objects, Pheromone implements a shortcut to transfer them between nodes: it piggybacks small objects on the function invocation requests forwarded during the inter-node scheduling (see §4.2). 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.

4.4 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 §4.2.
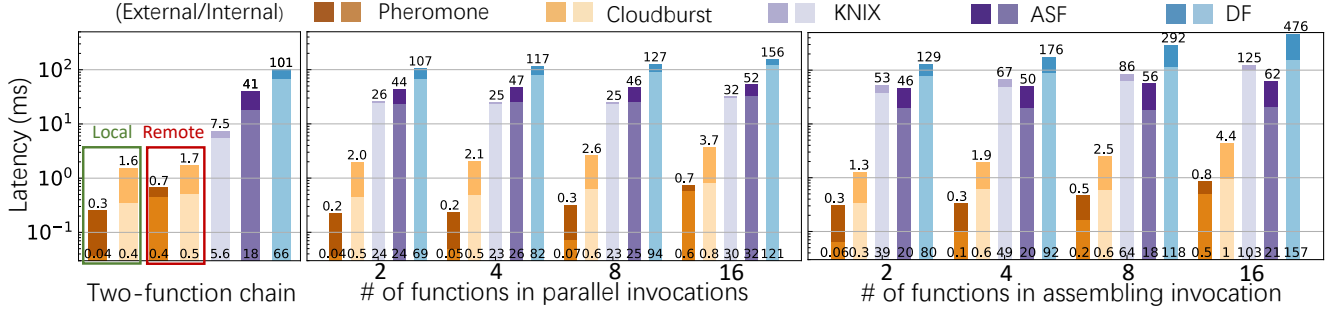
Figure 10: Latencies of invoking no-op functions under three interaction patterns: function chain, parallel and assembling invocations. Each bar is broken into two parts which measure the latencies of external (darker) and internal (lighter) invocations, respectively. The overall latency value is given at the top of the bar, and the internal invocation latency is given at the bottom.

5   Implementation

We have implemented `Pheromone` atop Cloudburst [66], 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 [19] images for ease of deployment. `Pheromone`'s client was implemented in 400 lines of Python code. Like Cloudburst, `Pheromone` runs in a Kubernetes [25] cluster for convenient container management, and uses Anna [71, 72], 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 code-base of `Pheromone` is open-sourced at [26].

6   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 (§6.2) and ensure high scalability (§6.3)?
- Can `Pheromone` effectively handle failures (§6.4)?
- Can developers easily implement real-world applications with `Pheromone` and deliver high performance (§6.5)?

6.1   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 [66] 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 [34], KNIX [24] 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 [14] to orchestrate function instances as it achieves faster function interactions than the ASF Standard Workflows [14]. As ASF has a size limit of transferring intermediate data (see Fig. 2), we use Redis [6], 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 [17] that can manage workflow states following the actor model [36, 54]. 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.

6.2   Function Interaction

Function invocation under various patterns.   We first evaluate the overhead of invoking no-op functions without any
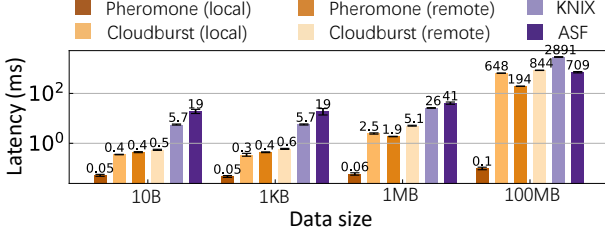
9

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

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

payload. We consider three common invocation patterns: 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 [20]. 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 (§4.3) 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
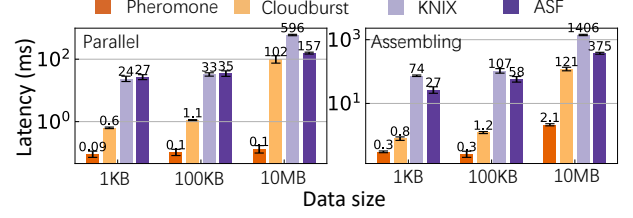
we exclude it from the experiments hereafter.

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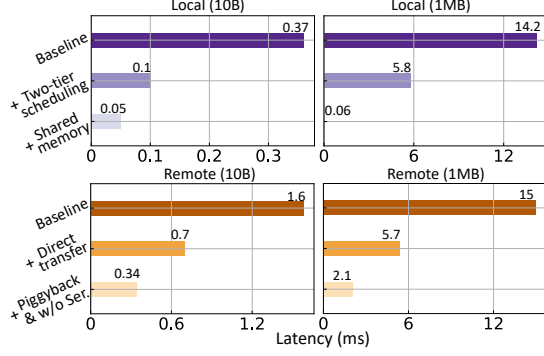
Figure 13: Improvement breakdown for local (top) and remote (bottom) invocations. Each case includes transferring 10 B (left) and 1 MB (right) of data in function invocations.
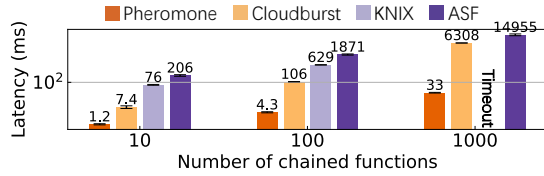

Figure 14: Latencies of function chains of different lengths.

voke downstream functions (i.e., no local schedulers), which is today's common practice [11]; "Two-tier scheduling" uses our local schedulers for fast invocations on the same worker node (§4.2), 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 (§4.3). 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 [71]) to exchange intermediate data among cross-node functions; "Direct transfer" reduces the communication overhead by allowing direct data passing between nodes (§4.3), where raw data objects on a node are serialized into a protobuf [27] message and then sent to downstream functions; "Piggyback & w/o Ser." further optimizes the data exchange by piggybacking small objects on forwarded function requests and eliminating serialization (§4.3). 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.

6.3 Scalability

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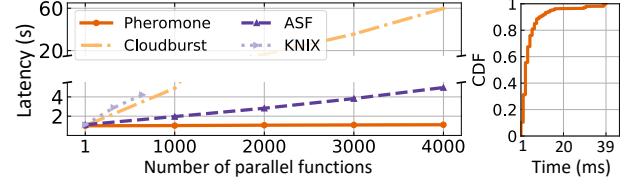

Figure 15: End-to-end latencies with various numbers of parallel functions (left), and the distribution of function start times when executing 4k functions in Pheromone (right).
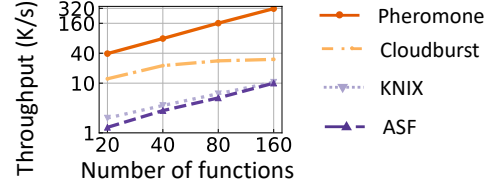

Figure 16: Request throughput when serving requests to no-op functions under various numbers of functions or executors.

Long function chain. We start with a long function chain that sequentially invokes a large number of functions [75]. 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 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
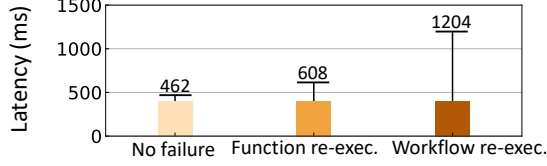
11

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

leads to low throughput due to its high invocation overhead (Fig. 10). Compared with these platforms, Pheromone ensures better scalability with the highest throughput.

6.4 Fault Tolerance

We evaluate Pheromone's fault tolerance mechanism (§4.4). 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 99th 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.

6.5 Case Studies

We evaluate two representative applications atop Pheromone: Yahoo's streaming benchmark for advertisement events [40], 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 §2.2, the key to enabling this application in serverless platforms is to periodically invoke a function to process the events accumulated during the past one second.

In Pheromone, this is straightforward by using the ByTime primitive (§3.3 and Fig. 7). This application can also be implemented easily in DF by using an addressable Entity function for aggregation [16]. 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
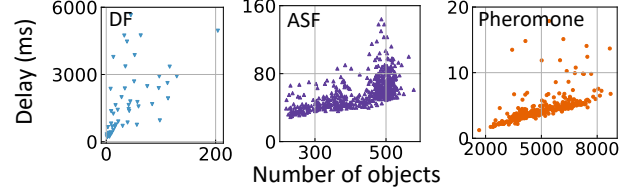
it can access and count the associated events per campaign.

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.

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

MapReduce sort. We next evaluate how Pheromone's data-centric orchestration can easily facilitate MapReduce sort, 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 [2] without operating on intermediate data (§3.2). Appendix A.3 gives an example of writing a MapReduce program in Pheromone-MR. We compare Pheromone-MR with PyWren [48], 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 Py-Wren 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
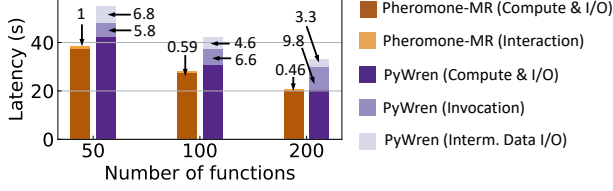
Figure 19: Latencies of sorting 10 GB data using various numbers of functions on PyWren and `Pheromone-MR`. The latency is broken down into: the interaction latency (for PyWren, the invocation and intermediate data I/O), and the latency for compute and I/O. The numbers indicate the former.

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. 19, 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 [13], 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.

7   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 (§4.3). Commercial container-based serverless platforms often do not co-locate functions from different users to enhance security [1]. 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 [33]; KNIX isolates a workflow's functions using processes in the same container for better performance [34]; recent work proposes lightweight isolation for privacy-preserving serverless applications [52]. `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 2) 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 [38,42,43,49,56,58,59]. One other approach is to exploit data locality to improve performance, e.g., by placing workflow functions on a single machine [46, 51, 53, 65–67]. Moreover, OFC [55] and Faa$T [61] provide the autoscaling cache for individual applications. Shredder [79] and Zion [62] push the function code into storage. Wukong [39] enhances the locality of DAG-based parallel workloads at the application level. Lambdata [67] 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 [3, 51].

8   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.

Acknowledgments

```
1  s: the batch size specified by developers
2  t: the target function of this trigger
3  B: a buffer of objects for next invocation
4  L: a set of statuses of running source functions
5
6  action_for_new_object(obj){
7    A ← {}
8    remove the source function of obj from L
9    B.add(obj)
10   if (B.size() >= s){
11     a ← trigger action for function t with objects in B
12     A.add(a)
13     B.clear()
14   }
15   return A
16 }
17
18 notify_source_func(info of source function){
19   add relevant info of source function to L
20 }
21
22 action_for_rerun(session){
23   A ← {}
24   if (L contains functions in this session){
25     A ← trigger actions for relevant functions in L
26   }
27   return A
28 }
```

Figure A.1: A pseudocode implementation of `ByBatchSize` primitive using the provided trigger interfaces.

A Appendix

A.1 Trigger Example

Pheromone provides a common interface for developers to implement customized trigger primitives. There are three main methods of the interface (§3.2, Fig. 5). Fig. A.1 provides an example of using the interface to implement the `ByBatchSize` trigger primitive.

A.2 Advertisement Event Stream

Fig. A.2 shows the script of deploying the event stream processing application (case study #1 in §6.5) in Pheromone. The application has three functions, `preprocess`, `query_event_info`, and `aggregate`, chained through two buckets (see Fig. 4, right). The script first registers the application with the name and three functions (lines 1-3), and then configures two bucket triggers, `Immediate` and `ByTime`, to invoke functions `query_event_info` (lines 6-11) and `aggregate` (lines 14-20), respectively. Fig. A.3 shows the code snippets of how the two producer functions `preprocess` and `query_event_info` interact with the two buckets. Each function creates an empty object by specifying its target bucket and key name (lines 2 and 7). Once the object value is computed (omitted in the code snippet) and the object is ready to be consumed, the function sends it to the bucket via the `send_object` method, thereby triggering the next function.

```
1  app_name = 'event-stream-processing'
2  functions = ['preprocess','query_event_info','aggregate']
3  client.register_app(app_name, functions)
4
5  # configure the first bucket trigger.
6  bck_name = 'immed_bck'
7  trig_name = 'immediate_trigger'
8  prim_meta = {'function':'query_event_info'}
9  client.create_bucket(app_name, bck_name)
10 client.add_trigger(app_name, bck_name, trig_name, \
11         IMMEDIATE, prim_meta)
12
13 # configure the second bucket trigger.
14 bck_name = 'by_time_bck'
15 trig_name = 'by_time_trigger'
16 prim_meta = {'function':'aggregate', 'time_window':1000}
17 re_exec_rules = ([('query_event_info', EVERY_OBJ)], 100)
18 client.create_bucket(app_name, bck_name)
19 client.add_trigger(app_name, bck_name, trig_name, \
20       BY_TIME, prim_meta, hints=re_exec_rules)
```

Figure A.2: Deploying the stream processing workflow.

```
1  // specify the bucket in function preprocess
2  obj = library->create_object("immed_bck", key_name);
3  ... // operate on obj
4  library->send_object(obj);
5
6  // specify the bucket in function query_event_info
7  obj = library->create_object("by_time_bck", key_name);
8  ... // operate on obj
9  library->send_object(obj);
```

Figure A.3: Code snippets of interacting with buckets.

Pheromone also provides a function-oriented interface for application deployment, which we detail in Appendix A.4.

A.3 MapReduce atop `Pheromone`

Pheromone's data-centric function orchestration provides a natural support for implementing data-flow applications in the serverless cloud. We have developed a MapReduce framework atop Pheromone using the system-provided `DynamicGroup` primitive and other APIs, which we call `Pheromone-MR`. In particular, `Pheromone-MR` configures `DynamicGroup` bucket triggers for a MapReduce program. It can accept the code snippets of mapper and reducer from clients, and package them into two Pheromone functions, i.e., map and reduce. The Pheromone map functions execute the user-provided code, and then automatically send the intermediate objects to the pre-configured bucket and specify which data group each object belongs to. When map functions all complete, reduce functions get invoked by the bucket trigger to consume intermediate objects and output the final results. Fig. A.4 shows a classical WordCount example in `Pheromone-MR`. Note that our implementation of `Pheromone-MR` includes only 380 lines of C++ function wrappers and 120 lines of Python interfaces. In comparison, PyWren [48], an equivalent system built on Lambda, requires about 6k lines of Python for executing map-only operations.

```
1  void map_function(const char* input, size_t data_size) {
2      string input_str{input, data_size};
3      vector<string> words;
4      split_string(input_str, ' ', words);
5      for (auto &w : words) emit(w, 1);
6  }
7
8  void reduce_function(string key, vector<int> &values) {
9      int sum = 0;
10     for (auto& v : values) sum += v;
11     emit(key, sum);
12 }
```

Figure A.4: A WordCount example in `Pheromone-MR`. Users simply write map (line 1-6) and reduce (line 8-12) functions.

```
1  app_name = 'event-stream-processing'
2  functions = ['preprocess','query_event_info','aggregate']
3  # dependency: (source_functions, target_functions, type)
4  dep1 = (['preprocess'],['query_event_info'],DIRECT)
5  # Include extra metadata (1000 ms timeout) in dependency
6  dep2 = (['query_event_info'],['aggregate'],PERIODIC,1000)
7  client.register_app(app_name, functions, [dep1, dep2])
8  # The two functions preprocess and query_event_info can
9  # directly create objects w/o specifying the bucket name:
10 #     library->create_object();
```

Figure A.5: Deploying the stream processing workflow using function-oriented interface.

A.4 Function-Oriented Interface

Pheromone also provides a function-oriented interface that encapsulates the data trigger APIs. The interface hides the detailed data consumption and allows developers to directly express function interactions without specifying buckets for workflow functions. Fig. A.5 gives an example of using this interface to deploy the aforementioned event stream processing workflow (case study #1 in §6.5). Developers simply specify the dependencies between functions (lines 4 and 6), which can be interpreted as data triggers. That is, the DIRECT and PERIODIC dependencies are translated into the Immediate and ByTime trigger primitives (Fig. A.2), respectively. It is worth mentioning that the function-oriented interface cannot support complex workflows that need the function-side knowledge to perform fine-grained data exchange (e.g., MapReduce), for which data trigger APIs are needed.

A.5 Evaluation of Garbage Collection

Pheromone garbage-collects in-memory data objects after the associated invocation request has been fully served (§4.3). We evaluate the effectiveness of garbage collection (GC) with a synthesized workflow. The workflow contains two chained functions, which generates 1 MB dummy data and keeps alive for a random period of time (e.g., sleep). The keep-alive period ranges from 1 ms to 1000 ms and is sampled following the log-normal distribution, a typical distribution of function execution time in a production platform [64]. We concurrently run this workflow in Pheromone to fully occupy the executor resources, and measure the memory usage over
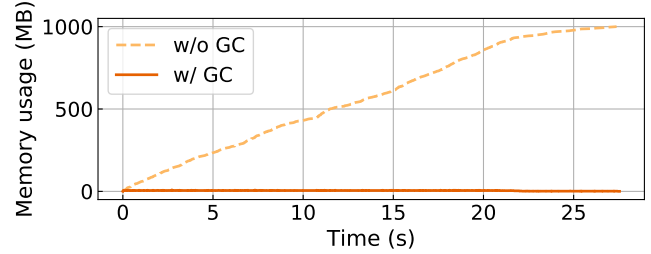

Figure A.6: Memory usage of shared-memory object store with and without garbage collection.

time with and without GC. As shown in Fig. A.6, enabling GC in Pheromone can timely evict unused data objects from the in-memory store, significantly reducing the memory footprint.

References

[1] Alibaba Cloud Function Compute. https://www.alibabacloud.com/product/function-compute.

[2] Apache Hadoop. https://hadoop.apache.org.

[3] Apache OpenWhisk. http://openwhisk.apache.org/.

[4] Apache OpenWhisk Composer. https://github.com/apache/openwhisk-composer/.

[5] Apache ZooKeeper. https://zookeeper.apache.org/.

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

[7] AWS Kinesis. https://aws.amazon.com/kinesis/.

[8] AWS Lambda. https://aws.amazon.com/lambda/.

[9] AWS Lambda execution model. https://docs.aws.amazon.com/lambda/latest/dg/runtimes-context.html.

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

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

[12] AWS Step Functions - Choice. https://docs.aws.amazon.com/step-functions/latest/dg/amazon-states-language-choice-state.html.

[13] AWS Step Functions - Map. https://docs.aws.amazon.com/step-functions/latest/dg/amazon-states-language-map-state.html.

[14] AWS Step Functions Standard vs. Express Workflows. https://docs.aws.amazon.com/step-functions/latest/dg/concepts-standard-vs-express.html.

[15] Azure Durable Functions. https://docs.microsoft.com/en-us/azure/azure-functions/durable/.

[16] Azure Durable Functions aggregator pattern. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview#aggregator.

[17] Azure Entity Functions. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-entities/.

[18] Azure Functions. https://azure.microsoft.com/en-us/services/functions/.

[19] Docker. https://www.docker.com.

[20] Firecracker network performance numbers. https://github.com/firecracker-microvm/firecracker/blob/main/docs/network-performance.md.

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

[22] Google Cloud Functions. https://cloud.google.com/functions.

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

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

[25] Kubernetes: Production-grade container orchestration. http://kubernetes.io.

[26] Pheromone codebase. https://github.com/MincYu/pheromone.

[27] Protocol Buffers - Google's data interchange format. https://github.com/protocolbuffers/protobuf.

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

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

[30] Serverless stream-based processing for real-time insights. https://aws.amazon.com/blogs/architecture/serverless-stream-based-processing-for-real-time-insights/.

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

[32] Using AWS Lambda with Amazon S3. https://docs.aws.amazon.com/lambda/latest/dg/with-s3.html.

[33] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *Proc. USENIX NSDI*, 2020.

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

[35] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. Sprocket: A serverless video processing framework. In *Proc. ACM SoCC*, 2018.

[36] Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and Jorgen Thelin. Orleans: cloud computing for everyone. In *Proc. ACM SoCC*, 2011.

[37] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. SEUSS: skip redundant paths to make serverless fast. In *Proc. ACM EuroSys*, 2020.

[38] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. Cirrus: A serverless framework for end-to-end ML workflows. In *Proc. ACM SoCC*, 2019.

[39] Benjamin Carver, Jingyuan Zhang, Ao Wang, Ali Anwar, Panruo Wu, and Yue Cheng. Wukong: a scalable and locality-enhanced framework for serverless parallel computing. In *Proc. ACM SoCC*, 2021.

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

[41] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proc. ACM ASPLOS*, 2020.

[42] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *Proc. USENIX ATC*, 2019.

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

[44] Alexander Fuerst and Prateek Sharma. FaasCache: keeping serverless computing alive with greedy-dual caching. In *Proc. ACM ASPLOS*, 2021.

[45] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proc. USENIX ATC*, 2010.

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

[47] Eric Jonas, Anurag Khandelwal, Karl Krauth, Johann Schleier-Smith, Qifan Pu, Neeraja Yadwadkar, Ion Stoica, Vikram Sreekanti, Vaishaal Shankar, Joseph E Gonzalez, David A Patterson, Chia-Che Tsai, Joao Carreira, and Raluca Ada Popa. Cloud programming simplified: A Berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.

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

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

[50] Jack Kosaian, K. V. Rashmi, and Shivaram Venkataraman. Parity models: erasure-coded resilience for prediction serving systems. In *Proc. ACM SOSP*, 2019.

[51] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. Faastlane: Accelerating function-as-a-service workflows. In *Proc. USENIX ATC*, 2021.

[52] Mingyu Li, Yubin Xia, and Haibo Chen. Confidential serverless made efficient with plug-in enclaves. In *Proc. ACM/IEEE ISCA*, 2021.

[53] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, and Ana Klimovic. SONIC: Application-aware data passing for chained serverless applications. In *Proc. USENIX ATC*, 2021.

[54] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *Proc. USENIX OSDI*, 2018.

[55] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël De Palma, Bernabé Batchakui, and Alain Tchana. OFC: an opportunistic caching system for FaaS platforms. In *Proc. ACM EuroSys*, 2021.

[56] Ingo Müller, Renato Marroquín, and Gustavo Alonso. Lambada: Interactive data analytics on cold data using serverless cloud infrastructure. In *Proc. ACM SIGMOD*, 2020.

[57] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In *Proc. USENIX ATC*, 2018.

[58] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. Starling: A scalable query engine on cloud functions. In *Proc. ACM SIGMOD*, 2020.

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

[60] K. V. Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. EC-cache: Load-balanced, low-latency cluster caching with online erasure coding. In *Proc. USENIX OSDI*, 2016.

[61] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. Faa$t: A transparent auto-scaling cache for serverless applications. In *Proc. ACM SoCC*, 2021.

[62] Josep Sampé, Marc Sánchez-Artigas, Pedro García-López, and Gerard París. Data-driven serverless functions for object storage. In *Proc. ACM/IFIP/USENIX Middleware*, 2017.

[63] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J Yadwadkar, Raluca Ada Popa, Joseph E Gonzalez, Ion Stoica, and David A Patterson. What serverless computing is and should become: The next phase of cloud computing. *Commun. ACM*, 64(5):76–84, 2021.

[64] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *Proc. USENIX ATC*, 2020.

[65] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *Proc. USENIX ATC*, 2020.

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

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

[68] Huangshi Tian, Suyi Li, Ao Wang, Wei Wang, Tianlong Wu, and Haoran Yang. Owl: Performance-aware scheduling for resource-efficient function-as-a-service cloud. In *Proc. ACM SoCC*, 2022.

[69] Ashish Vulimiri, Oliver Michel, P. Brighten Godfrey, and Scott Shenker. More is less: reducing latency via redundancy. In *Proc. ACM HotNet*, 2012.

[70] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. FaaS-Net: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute. In *Proc. USENIX ATC*, 2021.

[71] Chenggang Wu, Jose Faleiro, Yihan Lin, and Joseph Hellerstein. Anna: A KVS for any scale. In *Proc. IEEE ICDE*, 2018.

[72] Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. Autoscaling tiered cloud storage in anna. In *Proc. VLDB Endow.*, 2019.

[73] Le Xu, Shivaram Venkataraman, Indranil Gupta, Luo Mai, and Rahul Potharaju. Move fast and meet deadlines: Fine-grained real-time stream processing with cameo. In *Proc. USENIX NSDI*, 2021.

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

[75] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing serverless platforms with serverlessbench. In *Proc. ACM SoCC*, 2020.

[76] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proc. ACM EuroSys*, 2010.

[77] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. MArk: Exploiting cloud services for cost-effective, SLO-aware machine learning inference serving. In *Proc. USENIX ATC*, 2019.

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

[79] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. Narrowing the gap between serverless and its state with storage functions. In *Proc. ACM SoCC*, 2019.