Apiary: Tightly Integrating Compute and Data for Efficient, Transactional, and Observable Functions-as-a-Service

PREPRINT: DO NOT DISTRIBUTE

Abstract

Developers are increasingly using function-as-a-service (FaaS) platforms for *data-centric* applications that perform low-latency, often transactional, operations on data such as for e-commerce sites or social networks. Unfortunately, existing FaaS platforms support these applications poorly because their compute and storage layers are *disaggregated* into independent subsystems, harming performance and making it hard to provide transactional guarantees or monitoring and observability features.

We present Apiary, a novel FaaS platform for data-centric applications. Apiary *tightly integrates* compute and data by wrapping a distributed database engine and using it as a unified runtime for function execution, data management, and operational logging. Apiary guarantees functions run as ACID transactions with end-to-end exactly-once semantics and provides advanced observability capabilities like automatic data provenance capture. Despite offering more features and stronger guarantees than existing FaaS platforms, Apiary outperforms them by 7–68× on realistic microservice applications by greatly reducing communication overhead.

1 Introduction

Function-as-a-service (FaaS), or serverless, cloud offerings are becoming popular in both industry [6, 11, 54, 64, 73] and research applications [1, 2, 18, 19, 29, 30, 45, 55, 57, 58, 65, 69, 70, 72]. FaaS platforms radically reduce the operational complexity and administrative burden of cloud deployments, promising developers transparent auto-scaling and consumption-based pricing while eliminating the need to manage application servers [31].

We distinguish two broad classes of FaaS applications. The first are compute-intensive ones such as parallel analytics and video processing [19, 65]. The second, and the focus of this paper, are data-centric applications such as microservices and web serving [16, 30]. These perform low-latency, often transactional, operations on data typically stored in a separate storage system, such as confirming a travel reservation or adding items to a customer's shopping cart. Data-centric applications underlie most of the modern web, from social networks [11] to e-commerce [36] and the Internet of Things [21], so it is critical for FaaS platforms to efficiently support them.

Unfortunately, existing FaaS platforms support data-centric applications poorly because their compute and storage layers are *disaggregated* and designed independently (Figure 1a). Disaggregation is convenient for cloud vendors, who designed their FaaS platforms to work with their cloud storage offerings. However, it causes three problems for data-centric applications. First, it limits performance because state is externalized, so each operation on data must make a round trip to a storage system [29, 55, 58]. Second, it makes providing transactional guarantees for functions prohibitively complex and expensive because compute and data are logically separated [9, 25, 69]. Third, it complicates observability because traces are spread across a vast number of logs from tasks and a separate storage system, and thus difficult to interpret [10].

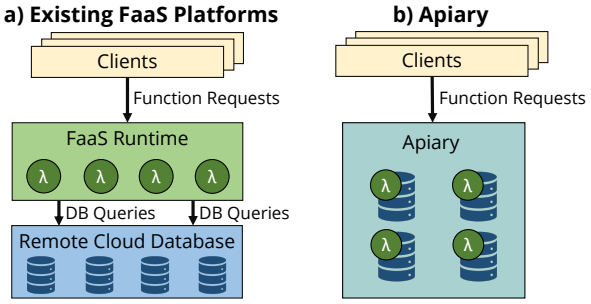

Figure 1: Existing FaaS platforms disaggregate compute and storage into separate systems, typically on physically separate servers. Apiary instead tightly integrates function execution with data management, executing functions as stored procedures inside a DBMS server.

Recently, there has been much research on building FaaS platforms for data-centric applications to tackle these challenges [29, 34, 37, 52, 55, 58, 72]. However, these systems do not deviate from the cloud vendors' original FaaS architecture: they disaggregate compute and data and only offer partial solutions. Some, such as Cloudburst [58] and Boki [29], locally cache data to improve performance, but this does nothing to provide transactions or observability. Others, like Beldi [69], use external transaction managers to provide transactions for functions using an existing storage system, but this increases already-high storage access times by as much as 3×.

In this paper, we propose Apiary, a novel FaaS platform optimized for data-centric applications. Apiary goes to an extreme not realized by existing systems. Instead of disaggregating compute and data, it *tightly integrates them* (Figure 1b), co-designing subsystems for function execution, data management, and operational logging. Api-

ary provides an easy-to-use interface to developers: they write functions in a high-level language, embedding SQL to access data stored in a relational database, then compose functions into graphs performing high-level operations. Apiary guarantees that functions run as ACID transactions, provides exactly-once semantics for end-to-end graph execution, and offers advanced observability capabilities like automatic data provenance capture. Despite offering more features and stronger guarantees than existing systems, Apiary outperforms them by up to 68×.

The key insight of Apiary is that to provide this combination of a familiar interface, strong guarantees, observability, and good performance, function execution and database operations must be handled by the same runtime. To do this, Apiary wraps an existing database management system (DBMS) and compiles user functions into *stored procedures*, routines in a non-SQL language that run natively as DBMS transactions. This design makes functions simultaneously a basic unit of control flow and atomicity. Apiary leverages this visibility into control and data flow both to co-locate compute and data and to augment the DBMS with scheduling and tracing layers that provide both end-to-end guarantees for function graph execution and advanced observability capabilities.

One major challenge that Apiary tackles is efficiently providing end-to-end guarantees, such as exactly-once semantics, for entire graphs of functions in the FaaS application. This is also a problem for existing FaaS platforms, which either require all functions to be idempotent [9, 25] or use high-overhead external transaction managers [69]. To address this problem, Apiary automatically instruments stored procedures to transactionally log their executions in the DBMS, avoiding re-execution in case of failures. Naively, this degrades performance up to 2.2×, so we develop a dataflow analysis algorithm to identify when functions can be safely re-executed, providing exactly-once semantics with <5% overhead.

A second challenge in Apiary is performing efficient and interpretable automatic tracing for observability. Apiary leverages its unified runtime to do tracing, instrumenting stored procedures to provide *data provenance capture*: recording all values read or written by each database operation. Conventional DBMSs can capture data provenance information, but it is difficult to interpret because it lacks application-specific context such as what high-level operation invoked a transaction and for what purpose [28]. Crucially, Apiary provides that context because *functions are a natural unit of control flow tracking*. Apiary groups captured data provenance information by functions, recording values read and written by each function. This novel capability makes monitoring, debugging, and auditing large-scale FaaS deployments easier, enabling queries like:

- Which operation most recently updated this record?
- Which records were produced by this data pipeline?
- Does this operation have the right to access this data?

We evaluate Apiary with microservice benchmarks representing business use cases such as social networking and e-commerce, adapted from suites such as DeathStarBench [22]. We find Apiary outperforms the popular open-source FaaS platform OpenWhisk [48] by 7–68× and the recent research system Boki [29] by up to 7.7×.

In summary, our contributions are:

- We demonstrate that the disaggregation of compute and data in FaaS platforms makes it difficult to provide transactions, observability, and good performance for increasingly popular data-centric applications.
- We propose Apiary, a novel FaaS platform designed for data-centric applications. By tightly integrating function execution and data storage using a relational DBMS, Apiary can provide transactional functions and end-to-end exactly-once semantics while also improving data-centric application performance by 7–68× compared to widely-used FaaS platforms.
- By instrumenting database operations and using functions as units of control flow tracking, Apiary automatically captures data provenance information needed for business-critical queries, incurring overhead of <15%.

2 Background and Motivation

2.1 Data-Centric FaaS Applications

Function-as-a-service (FaaS) platforms are increasingly used for short-lived, *data-centric* applications such as microservices and web serving [16, 30]. Examples include:

- A social network that allows users to create multimedia posts and view and interact with other users' posts [53].
- An e-commerce service that allows customers to retrieve information about items, add those items to a virtual shopping cart, and check out those items [24].
- A hotel reservation service that allows users to find hotels near a geographic location, retrieve information about those hotels, and reserve a room [22].

Critically, each of these applications consists largely of short-lived tasks that primarily perform operations on data, such as saving a social network post, retrieving e-commerce item information, or reserving a hotel room.

Engineers want to implement data-centric applications using FaaS because it reduces the amount of infrastructure they need to manage. For example, in the traditional three-tier web serving architecture, engineers must manage both a stateful storage backend and a stateless middle tier of web servers that implement application logic and respond to client requests. Engineers are responsible for handling failures of any of these servers and for scaling them in response to changing load. FaaS promises to lift the burden of managing these servers, replacing the middle tier with functions implementing application logic and the backend with a cloud storage system.

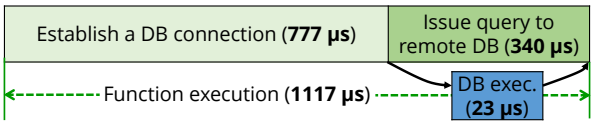

Figure 2: Latency breakdown for a function performing a point database update in the popular FaaS platform OpenWhisk. Query execution accounts for only 2% of the overall function execution time.

Unfortunately, existing FaaS platforms do not live up to this promise for data-centric applications. As we argued in the introduction, this is largely because existing platforms disaggregate compute and data into independent subsystems, causing several problems:

Performance. Data-centric applications typically perform low-latency operations on data, like point lookups and updates. In a modern high-performance DBMS, these each take at most a few tens of microseconds. However, as we show in Figure 2, in the popular FaaS platform OpenWhisk, as in other production FaaS platforms, they take more than a millisecond even if we ignore scheduling, container initialization, and other management overheads (which add an additional few milliseconds, analyzed in Section 7.5). For an OpenWhisk function to perform operations in a high-performance in-memory database, it must first establish a connection to the database (777µs), then issue queries remotely (340µs round-trip time per query). These overheads are vastly greater than the actual query execution time of 23µs for a point update. Thus, in conventional disaggregated FaaS platforms, communication overhead decreases the performance of data-centric applications by an order of magnitude.

Transactions. Data-centric applications typically require strong transactional and data integrity guarantees. For example, if someone tries to reserve a hotel room, it is important that they pay for it exactly once and that no one else can simultaneously reserve the same room. Unfortunately, because FaaS platforms are logically separated from their storage, they struggle to provide these guarantees. For example, if functions crash, existing platforms naively restart them, potentially re-executing transactions that completed before the crash. This can violate guarantees, for example by paying for a hotel room twice. To avoid such problems, developers must implement workarounds like making all functions idempotent [9, 25] or building an external transaction manager [69].

Observability. In order to debug, monitor, and audit data-centric applications, developers require information on *data provenance*: how tasks interact with data, such as what records they read and write. However, in a disaggregated FaaS platform this information is hard to collect and interpret [10]. The database can keep track of what operations occur, but does not know which high-level functions or services are responsible for them, as this control flow information is spread across traces of vast numbers of ephemeral tasks. Organizing and interpreting this data is possible with manual annotations, but this is tedious, error-prone, incurs high overhead, and may require coordination across developers of multiple services.

2.2 Current Approaches in Data-Centric FaaS

Many recent research projects seek to improve the performance and functionality of FaaS platforms on data-centric applications. Perhaps the most similar to Apiary are Cloudburst [58], Boki [29], Shredder [72], and Beldi [69]. Cloudburst [58] proposes using the autoscaling key-value store Anna [66] as a FaaS storage backend then improving data access performance with local caches. Boki [29] proposes a FaaS storage backend based on shared logs that uses local caches to improve performance. Shredder [72] allows developers to define low-latency "storage functions" that run on the same server as a key-value store hosting their data. Beldi [69] bolts an external transaction manager onto existing serverless platforms to provide guarantees for workflows.

Unlike Apiary, none of these systems address the root cause of the problems of data-centric FaaS: disaggregation. They all logically separate compute and data and thus struggle to provide transactional guarantees or observability features. Beldi provides ACID transactions, but it requires an external transaction manager, increasing the already-high overhead of remote cloud storage access by an additional 2.4–3.3×. Cloudburst and Boki cache data locally to improve performance, but both offer only key-value interfaces with weak consistency models (causal consistency for Cloudburst, monotonic reads and read-your-writes for Boki). Boki does support transactions, but uses an implementation of Beldi with similarly high overhead. Shredder runs functions directly on data servers, but offers no transactions and does not support distributing application data across multiple servers.

2.3 Non-Goals

Before discussing Apiary, we want to emphasize two objectives that are *excluded* from the scope of this paper.

Compute-Heavy Workloads. Apiary's design focuses on short-lived data-centric applications, not long-running compute-intensive workloads such as video processing [19] or batch analytics [52]. These do not require Apiary's features and guarantees, such as transactional functions and exactly-once semantics. If users wish to execute long-running tasks as part of an Apiary workflow, we expect them to leverage an external service, such as AWS Rekognition [7] for text detection in images.

Non-Relational Data Models. Apiary is tightly integrated with a relational DBMS and currently only supports a relational data model. Most comparable data-centric FaaS platforms are also restricted to relational or key-

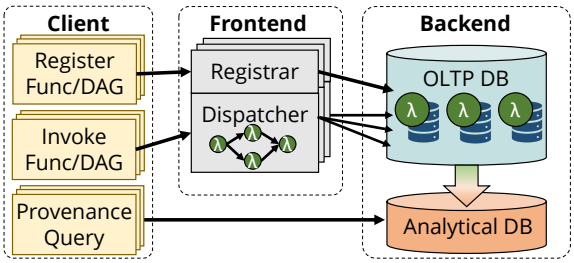

Figure 3: Architecture of Apiary.

value data [29, 55, 58, 69, 72]. We have found that most of the data-centric applications we examine are compatible with the relational model. We believe that Apiary could incorporate other data models, like Lucene indexes [12] for text search, but leave that for future work.

3 Apiary Overview

Design Rationale. To solve the challenges raised in Section 2, we design Apiary to *tightly integrate* compute and data. Apiary wraps a distributed DBMS, compiling functions to stored procedures so it can use the DBMS as a runtime for both function execution and data management. A key implication of this design is that Apiary functions are *simultaneously* basic units of control flow and atomicity. Apiary leverages this visibility into control and data flow to aggressively co-locate compute and data and to augment the DBMS with scheduling and tracing layers providing transactions, end-to-end guarantees, and advanced observability capabilities for data-centric FaaS applications. We sketch the Apiary architecture in Figure 3. It has three tiers: the clients, frontend, and backend.

Clients. Clients are end users interacting with Apiary. Users write functions and compose graphs using the Apiary programming model, which we describe in Section 4.

Frontend. Frontend servers route and authenticate requests from clients to the backend. Each frontend server has two components: a dispatcher, which manages graph execution, and a registrar, which handles graph registration and compilation. Neither component executes application logic; both are stateless and persist state in the backend. We discuss both more in Section 5.

Backend. The backend executes functions and manages data. It wraps a distributed transactional DBMS. Apiary executes functions transactionally on DBMS servers. Additionally, Apiary uses a separate distributed analytical DBMS to manage provenance data captured by functions. We describe the backend in detail in Sections 5 and 6.

4 Programming Model

Apiary gives developers an easy-to-use interface to build FaaS applications: they write functions in a high-level language and use SQL to access or modify data stored in a relational DBMS. Then, like in other FaaS platforms [8, 29, 42, 58], they compose functions into graphs comprising high-level operations. Apiary provides ACID guarantees for functions and exactly-once semantics for graphs.

Graph Interface

addLocalFunction(Function)	Add a local function to a graph.
addGlobalFunction(Function)	Add a global function to a graph.
addEdge(Func, Func, EdgeID)	Add an edge between two functions.

Function Interface

queueSQL(Query, List[Arg], Level)	Queue a SQL operation for execution, capturing provenance information at a specified level (§6.1).
executeSQL() → List[Result]	Batch execute queued operations, return results.
retrieveInput(Edge) → Object	Retrieve an input from a graph edge.
returnOutput(Object, Edge, Optional[Key])	Return an output along a graph edge; if it is to a local function, associate it with a partition key.

Figure 4: The Apiary graph and function interfaces.

4.1 Function Interface

Apiary functions are written in a high-level language (we use Java) and take in and return any number of serializable objects. Functions can embed any number of SQL queries to access or modify data in the DBMS; e.g., an e-commerce app can first check inventory then add an item to a cart. We show the function interface in Figure 4. Apiary guarantees functions execute as ACID transactions.

Apiary targets modern distributed DBMSs like VoltDB [63], SingleStore [56], and YugaByte [68], where data is divided across many *partitions*, each containing a portion of every database table. To access data in such systems, functions can be *global* or *local*. Global functions can access or modify any database state; local functions can only access or modify state on a single partition. While local functions may appear less powerful than global functions, they require no coordination and are thus much faster, so we expect that, like in most database systems [32, 59], the vast majority of functions will be local.

Each execution of a local function must be associated with a *partition key*. This determines which database partition the function execution accesses; function executions with the same partition key value are guaranteed to access the same partition. Typically, the choice of partition key corresponds to some property of a database table the function accesses. For example, a developer might decide that all functions accessing customer order data tables should use customer ID as their partition key, so for each customer, all their data is stored on the same partition.

Sometimes, developers require some information to be available to all functions, regardless of partition. For example, functions on customer order data might need information about the locations of warehouses from which orders can ship. To make this possible, we leverage the built-in replication functionalities common in distributed DBMSs to replicate a table across all partitions. Replicated tables can only be modified by global functions, but can be read by local functions from any partition.

4.2 Graph Interface

To define high-level operations in Apiary, developers compose functions into directed and acyclic *graphs*. We sketch

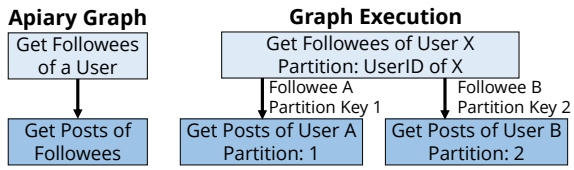

Figure 5: The Apiary graph for a social network timeline retrieval operation, which first retrieves a list of followed users then retrieves their posts. When the graph is executed, because post data is partitioned, the `getPosts` function runs multiple times, once for each followee's partition.

```
1 SQL query = new SQL("SELECT followeeID FROM
2   Followees WHERE userID=?;");
3 void getFollowees() {
4   int userID = retrieveInput(sourceEdgeID);
5   queueSQL(query, userID, ReadCapture);
6   List followees = executeSQL();
7   for (f : followees)
8     returnOutput(f, edgeID, f.ID);
9 }
```

Figure 6: Implementing `getFollowees` from Figure 5 using the Apiary interface. This is a local function accessing the `Followees` table; the partition key is `userID`.

the graph interface in Figure 4, diagram an example graph in Figure 5, and code a graph function in Figure 6.

Within a graph, nodes represent functions and edges represent data flow. Functions in a graph must associate each of their inputs and outputs with an edge, as shown in lines 4 and 8 of Fig. 6. Graph source and sink nodes have special edges receiving input from and returning output to the client. When Apiary executes a graph, the Apiary dispatcher sends each function output along its edge to become an input to later functions. Any value passed to a local function must be associated with a partition key (which can be chosen dynamically during execution) to determine which partition the local function will access.

One challenge in executing graphs is that functions often apply the same operation to multiple inputs on different partitions. For example, to retrieve a user's social network timeline, as in Figure 5, a graph first retrieves the list of the user's followees (people the user follows), then retrieves the most recent posts of each followee. Naively, we could retrieve all posts in a single global function that executes in one transaction, but that would require expensive and unnecessary coordination. Instead, Apiary allows local functions to *dynamically fan out*. If a local function is passed multiple inputs associated with different partition keys, Apiary executes it multiple times in parallel, once for each unique partition key it received. Each execution is sent only the inputs associated with its key. For example, if we wanted to retrieve the recent posts of ten different followees, we would pass our function one input per followee, each with that followee's ID as its partition key. The function would then execute ten times, with each execution retrieving data from a different partition.

Apiary not only executes each function as an ACID transaction, but also provides *exactly-once semantics* for entire graphs. Each function execution in a graph completes exactly once, regardless of failures. This eliminates the need for developers to make functions idempotent, as is necessary in existing FaaS platforms [9, 25, 69]. We implement this guarantee in Section 5.4.

5 Apiary Compilation and Execution

We implement the Apiary programming model by wrapping a distributed DBMS. We compile functions to DBMS stored procedures, then instrument them to provide end-to-end exactly-once semantics (§5.4) and data provenance capture for observability (§6). This architecture requires no modification of the underlying DBMS, so we can leverage the battle-tested infrastructure of existing DBMSs to build Apiary instead of designing a new data store from scratch. Apiary can be implemented using any distributed relational DBMS that has the following five properties:

- Shared-nothing data partitioning, with support for performing operations locally on a single partition.
- Supports ACID transactions.
- Supports running user code in a non-SQL language transactionally in stored procedures.
- Supports change data capture (for provenance, §6.2).
- Supports elastic DBMS cluster resizing.

Many DBMSs have these properties (e.g., Yugabyte [68] and SingleStore [56]); we ultimately chose VoltDB [63].

5.1 Cluster Overview

An Apiary cluster is made up of a DBMS backend and a number of stateless frontend servers that interface between clients and the backend. For isolation, we assign each application its own logical database in the backend, which is stored in many partitions on multiple servers. This is similar to how conventional FaaS platforms require a backend of long-running storage servers.

One option for auto-scaling Apiary is to use a serverless cloud database backend like AWS Athena [5], which provides managed elastic auto-scaling and charges for storage and per query. However, we instead use VoltDB because it is open-source. We scale using a simple utilization-based heuristic similar to prior work in DBMS auto-scaling [60], adding or removing servers with changes in query load. We rely on the DBMS to repartition data when scaling.

5.2 Compilation

To handle function execution and database operations using the same runtime, Apiary compiles graphs to stored procedures, routines in a non-SQL language that run natively as DBMS transactions. Each function in a graph is compiled to a separate stored procedure. Compilation happens in two steps. First, Apiary instruments each function to log its executions for exactly-once semantics (§5.4) and capture data provenance information for observability (§6). Then, Apiary registers the instrumented functions in the DBMS. Most DBMSs support compiling functions

from at least one high-level language into stored procedures; for VoltDB, this language is Java. The Apiary interface extends the DBMS stored procedure interface, so we can compile any function as long as it:

- Uses the Apiary interface (Figure 4) to communicate with the DBMS, receive inputs, and return outputs.
- Defines all SQL queries statically as parameterized prepared statements (to enable static analysis; this is a common requirement of DBMS stored procedures).
- Ensures all calls to external services (e.g., text detection) are idempotent and so can safely be re-executed.

5.3 Apiary Dispatcher and Graph Execution

Graph execution is managed by the Apiary dispatcher. It executes each function of a graph in topological order, invoking the corresponding stored procedure on the DBMS server hosting the appropriate partition, passing in its inputs, and collecting its outputs to send to later functions. If a function must fan out to multiple partitions, as discussed in Section 4, the dispatcher asynchronously launches all executions in parallel. Because dispatchers must remember the outputs of earlier functions so they can pass them as inputs to later functions (as persisting them to the DBMS is expensive), failure of a dispatcher is non-trivial. We discuss in Section 5.4 how Apiary efficiently mitigates dispatcher failures by selectively persisting function outputs to provide exactly-once semantics.

One caveat of Apiary function execution is that Apiary either executes a function locally in a single partition or requires a global lock on all partitions. This is an artifact of our VoltDB-based implementation, as VoltDB only supports single-site and global transactions. Prior database research has shown that most transactional workloads (including the microservice and web serving workloads we examined for this paper) can be expressed using mostly single-site transactions [32,59]. For workloads that cannot be expressed this way, there are many alternative concurrency control algorithms that a DBMS backend could use for efficient multi-partition transactions [27,67].

5.4 Exactly-Once Semantics

A major challenge in Apiary is efficiently providing exactly once semantics for graphs: each function execution in a graph must complete exactly once. This is hard because DBMSs provide guarantees for individual transactions but not across them, and executing a graph as a single transaction adds prohibitive coordination overhead. End-to-end guarantees are also a problem for existing FaaS platforms, which either require all functions be idempotent [9, 25] or use costly external transaction managers [69].

Two scenarios can lead to violation of exactly-once semantics: failures of DBMS servers and failures of Apiary dispatchers. Virtually every distributed DBMS can recover from failures of its own servers, typically using replication and logging. For instance, VoltDB can recover from any single server failure without loss of availability by failing over to a replica, and from failure of multiple servers without loss of data by recovering from logs.

Algorithm 1 Fault tolerance decision

```
1:  function FINDLOGGEDNODES(DAG)
2:      {f_1,...,f_n} = topoSort(DAG)        ▷ f_1 is source, f_n is sink.
3:      Logged = {}
4:      for f_i ∈ {f_n...f_1} do             ▷ Traverse from sink back to source.
5:          if hasWrite(f_i) then
6:              Logged.add(f_i)
7:          else
                ▷ Use BFS to find all paths to the logged nodes and sink.
8:              Paths = BFSFindPaths(f_i, Logged ∪ {f_n})
9:              if hasDisjointPaths(Paths) then
10:                 Logged.add(f_i)
11:     return Logged
```

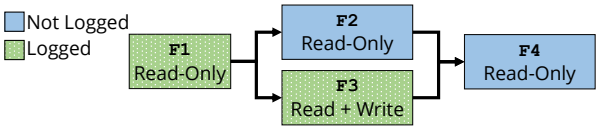

Figure 7: Example of Apiary logging rules. `F3` is logged as it performs a write. `F1` must also be logged to avoid inconsistent outputs to `F2` and `F3`.

To recover from dispatcher failures during graph execution, Apiary must ensure a new dispatcher can resume from where the failed one left off. To make this possible, Apiary automatically instruments stored procedures to transactionally log function outputs in the DBMS before returning. Then, if a dispatcher fails, clients can send the entire graph to a new dispatcher, using a per-invocation unique ID to signal a re-execution. Each re-executed function first checks for a log from the earlier execution; if it finds one it returns the logged output instead of executing.

Our experiments (§7.6) show that naively logging every function output degrades performance up to 2.2× as it requires performing multiple additional database lookups and updates in each function execution. However, we can optimize this guarantee and reduce overhead to <5% (across all workloads we tested) by recognizing that some functions *can* safely re-execute and need not be logged. For example, if an entire graph is read-only, it can be safely re-executed if its original execution failed, so we do not need to log any of its functions. Therefore, we develop a dataflow analysis algorithm (Algorithm 1) to determine, using static analysis when a graph is compiled into stored procedures, which functions must be logged and which can be safely re-executed.

We must log any function performing a DBMS write to ensure writes are not re-executed (re-executing external calls is fine as they must be idempotent). Moreover, we must log a read-only function if there exist disjoint paths from it to multiple different logged functions, or to at least

one logged function and the sink. This guarantees that two logged functions which depend on a value computed by an ancestor will always observe the same value from that ancestor, even if graph execution is restarted. If a logged local function can fan out to multiple executions (§4.2), we treat it as multiple functions for the purpose of this algorithm, to ensure all executions see consistent inputs.

We sketch our algorithm in Algorithm 1 and provide an example in Figure 7. `F3` is logged for performing writes, but `F1` is also logged despite being read-only. Suppose `F1` was not logged and a dispatcher crashed after executing F1 and F3. Upon re-execution, F1 may return a different value than it did originally because some data was changed by an unrelated transaction. If that happened, F2 would return an output based on the new output of F1, but F3 would return its logged output based on the original output of F1. This causes an inconsistency that violates exactly-once semantics, so we must log F1 to prevent it.

6 Provenance and Observability

By tightly integrating data and compute, Apiary provides an advanced observability capability for FaaS applications: automatic *data and workflow* provenance capture. Apiary records not only the full history of function execution but also all values functions read from or write to the DBMS, spooling information to an analytical database for storage and analysis. We show how provenance capture enables critical monitoring, debugging, and auditing queries that are difficult in existing FaaS platforms.

6.1 Apiary Provenance Interface

Apiary automatically captures information on all database read and write operations. While it is possible to do this in a conventional database, the captured information is difficult to interpret because it lacks application-specific context such as what high-level application invoked each database operation and for what purpose. Apiary can automatically supply that context because it tightly integrates data management with function execution: Apiary knows which function invoked each database operation.

Data Provenance Capture. When writing functions, developers can choose, for each database operation, to instrument them to capture provenance information at one of multiple levels of granularity. For read operations, developers can capture no provenance information at all, only metadata, or both metadata and data. Metadata includes the function execution ID, the tables operated on, and the timestamp. Data includes all records retrieved by the query, excluding records that are accessed incidentally (e.g., by an aggregation). For write operations, Apiary automatically captures both metadata and data, logging all updated records. This guarantees that later provenance queries can reconstruct table state from captured updates.

Workflow Capture. To make data provenance information interpretable, Apiary transparently uses instrumentation to capture information on high-level *workflows*, or graph invocations. Specifically, it records unique IDs for each function execution and graph invocation and associates them with data provenance information. This allows developers to query, for example, exactly which graph invocation created a particular database record. As we discuss in Section 8, automatic workflow capture is a novel feature of Apiary made possible by a unique characteristic of FaaS: *functions are a natural unit of control flow tracking*. In traditional server-based programs, state-of-the-art high-level workflow capture relies instead on tedious manual annotation and log parsing [4, 17, 41, 46, 49].

Provenance Query Interface. Apiary automatically spools provenance information to an analytical database (in our implementation, Vertica [62]) for long-term storage and analysis. We use a separate analytical database because it is better optimized for large data provenance queries than a transactional DBMS. Within the analytical database, provenance information is organized into tables. For captured workflow information, we create a function invocations table per application:

```
FunctionInvocations (timestamp, tx_id,
      function_name, graph_id, execution_id)
```

`tx_id` is a unique transaction ID per function execution, while all functions within a graph invocation share the same `execution_id`.

For each Apiary table used by an application, we create a corresponding event table in the analytical database for captured information on operations on that table:

```
TableEvents (timestamp, tx_id, event_type,
      [record_data...])
```

`event_type` can be `insert`, `delete`, `update`, and `read`, and `tx_id` is defined as above.

Using information stored in these tables, developers can perform data provenance queries with SQL in the analytical database. We show examples in Section 6.3.

Beyond provenance capture, Apiary also performs basic monitoring. It records monitoring information (e.g. CPU usages, function execution times, etc.) automatically captured by the transactional DBMS and exports it to the analytical database to aid in analysis; for example:

```
FunctionExecTimes (timestamp, tx_id, latency)
```

6.2 Implementing Provenance Capture

Because Apiary handles function execution and data management in the same runtime, it can interpose between functions and the database to efficiently capture data provenance information. Whenever a function performs a database operation, Apiary automatically records meta-

data such as the timestamp and transaction ID. For write operations, Apiary also records updated data. For read operations, it modifies the query to return the primary keys of all retrieved rows (in addition to the requested information), then records them to identify each accessed record. We only capture rows that are actually retrieved, not rows that are accessed incidentally (e.g. by an aggregation).

The major challenge in data provenance capture is performing it efficiently while providing guarantees about what information is captured. To capture writes, we rely on DBMS change data capture to automatically and transactionally export information. However, read capture is more difficult, both because existing DBMSs do not have built-in read capture capability and because reads are numerous so overhead may be higher. We capture reads by instrumenting the Apiary `executeSQL` function (Fig. 4). We maintain a circular buffer inside each DBMS server's memory. Whenever a read occurs in `executeSQL`, we append its information to this buffer. Periodically, we flush the buffer to the remote analytical database. This guarantees no captured information is spurious; all captured reads reflect completed transactions. Moreover, the information is safe across re-executions (§5.4) because re-executions are easily recognized and deduplicated using their shared IDs. However, captured read information may be lost if the database server crashes while information is still in the buffer. We flush the buffer every 10ms so this occurs infrequently, but if developers cannot tolerate any data loss, we optionally can place the buffer on disk, eliminating this loss at some performance cost.

6.3 Enhancing Observability with Provenance

We show the value of data provenance capture by demonstrating how Apiary can handle three business-critical queries adapted from tasks of interest to our industrial partners. Because these queries require information from multiple services, they are difficult to answer using typical FaaS platforms and monitoring schemes.

Debugging. Our first query is "What was the state of some record X when it was read by this particular function execution?" This query might be used to determine what input caused a function to run abnormally slowly or to emit a malformed output. Apiary can answer this query easily because it records in `TableEvents` all changes to data; we simply retrieve the last update to record X before the problematic function execution. Because functions are transactional, this is guaranteed to match the record state the function read. For instance, we can use a single SQL query to find the state of record X at a time T:

```
SELECT record_data FROM TableEvents
WHERE event_type IN ('insert', 'update')
  AND record_id=X AND timestamp <= T
ORDER BY timestamp DESC LIMIT 1;
```

Downstream Provenance. Our second query is "Find all records that were updated by a graph that earlier read record X." This query is useful for taint tracking, for example, if record X contains misplaced sensitive information. Since we capture both workflow and data provenance, we can answer this query in Apiary by scanning for functions that read record X, then returning the write sets of later functions in the same graphs:

```
SELECT DISTINCT(record_id)
FROM TableEvents, FunctionInvocations
WHERE event_type IN ('insert', 'update')
  AND function_name in SUCCESSOR_FUNC_NAMES
  AND execution_id in EXECUTION_IDS;
```

Auditing. Our third query is "Did this function have the legal rights to all the data it accessed?" This is a typical auditing query, especially in a post-GPDR world where many items of personally identifiable data can only be used for specific purposes. Apiary makes it easy, as we can simply query the function's read set and identify the records which are not supposed to be accessed:

```
SELECT record_id
FROM TableEvents, FunctionInvocations
WHERE event_type='read' AND function_name=FN
  AND record_id NOT IN ALLOWED_SET;
```

7 Evaluation

We evaluate Apiary with microservice workloads adapted from well-established benchmark suites [22, 24, 53]. We additionally use microbenchmarks to analyze the performance of Apiary's features and guarantees. We show that:

1. By tightly integrating compute and data, Apiary improves data-centric FaaS application performance by 7–68× compared to production FaaS systems and 7.7× compared to recent research systems (Fig. 8, 10).
2. By selectively instrumenting functions using a novel dataflow analysis algorithm, Apiary provides transactional guarantees and exactly-once semantics for functions and graphs with overhead of <5% (Fig. 12).
3. By instrumenting database operations and using functions as a unit of control flow tracking, Apiary automatically captures data provenance information critical to observability with overhead of <15% (Fig. 13).

7.1 Experimental Setup

We implement Apiary in ~10K lines of Java code[1]. We use VoltDB [63] v9.3.2 as our DBMS backend and Vertica [62] v10.1.1 as our analytics database. For communication between clients and frontend servers, we use JeroMQ [51] v0.5.2 for lightweight RPCs over TCP.

In all experiments where not otherwise noted, we run on Google Cloud using `c2-standard-8` VM instances with 8 vCPUs and 32GB DRAM. We use as a DBMS backend

[1] We will release source code with the publication of this paper.

a cluster of 40 VoltDB servers with 8 VoltDB partitions per VM. For high availability, we replicate each partition once, as is common in production. We also use the same VoltDB cluster as the storage backend for our baselines. To ensure we can fully saturate this DBMS backend, we run 45 Apiary frontend servers and perform queries using 15 client machines, each running on a `c2-standard-60` instance with 60 vCPUs and 240GB DRAM. We spool provenance data to a cluster of 10 Vertica servers. All experiments run for 300 seconds after a 5-second warmup.

7.2 Baselines

We compare Apiary to three baselines, ranging from commercial platforms to the latest research systems.

OpenWhisk. OpenWhisk (OW) [48] is a popular open-source production FaaS platform. We implement each of our workloads in the OW Java runtime, performing all business logic in an OW function but storing and querying data in an external VoltDB cluster. We coordinated with OW developers to tune our OW baseline. Since OW cannot efficiently manage multiple concurrent low-latency functions, we implement each workload in a single OW function. Moreover, we pre-warm OW function containers and only measure warm-start performance. In our experiments, we use 45 `c2-standard-8` VMs as OW workers. To maximize OW performance, we use 5 controllers, each on a `c2-standard-60` instance that manages 9 workers, load balancing between the controllers' sub-clusters.

RPC Servers. Most microservices today are deployed in long-running RPC servers with separate application and database server machines [22, 36]. We implement each of our workloads this way, performing all business logic in long-running RPC servers but using an external VoltDB cluster to store and query data. For a fair comparison, we use the same communication library as Apiary (JeroMQ, chosen because we found it outperforms alternatives like gRPC) and re-implement each microservice in Java following its original architecture. In our experiments, we use a setup identical to Apiary but with all frontend servers replaced with microservice servers.

Boki. Boki [29] is a recent research system designed for data-centric FaaS tasks. As it is co-designed with a disaggregated storage system, we use it unmodified. Since Boki depends on AWS EC2, we deploy 8 storage nodes, 3 sequencers, and 8 workers, running on EC2 `c5d.2xlarge` instances with 8 vCPUs, 16GB DRAM, and 200GB NVMe SSD. This is the same setup used in the Boki paper, and we coordinated with the Boki authors to tune our baseline. For fairness, when comparing Apiary to Boki, we use 8 VoltDB servers and 8 frontend servers.

7.3 Microservice Workloads

To evaluate the performance of Apiary on realistic and representative workloads, we use three microservice benchmarks, each performing business-critical data-centric tasks. As we show in Table 1, these workloads cover a large design space for data-centric FaaS applications.

Workload	Operation	Ratio	Read-Only?	Access Rows	RPCs for μServices	# of Func.	# of SQL Queries
Shop	Browsing	80%	Yes	8	2	1	1
	CartUpdate	10%	No	1	2	1	2
	Checkout	10%	No	5	6	3	5
Hotel	Search	60%	Yes	30	4	6	22
	Recommend	39%	Yes	1	2	1	1
	Reservation	1%	No	5	2	1	5
Retwis	GetTimeline	90%	Yes	550	3	51	51
	Post	10%	No	1	2	1	1

Table 1: Microservice benchmark information. RPCs are for the RPC Servers baseline; Apiary and OW only require one client-server RPC. Apiary only requires one DB round trip per function, but the baselines one per SQL query.

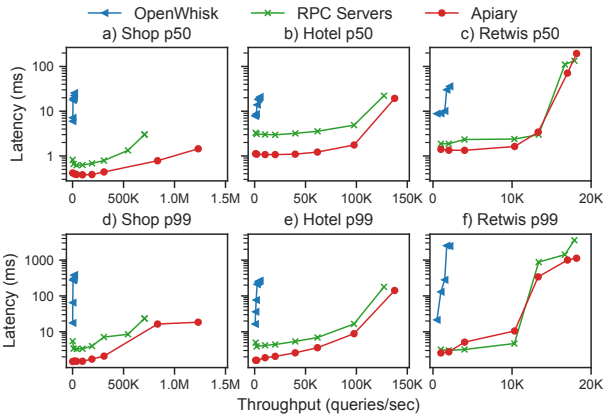

Figure 8: Throughput versus latency for Apiary (with full features enabled) and the OpenWhisk (OW) and RPC Servers baselines on all three benchmarks.

Shop. This benchmark, from Google Cloud [24], simulates an e-commerce service, where users browse an online store, update items from their shopping cart, and eventually check out items for purchase.

Hotel. This benchmark, from the DeathStarBench [22] suite, simulates searching and reserving hotel rooms.

Retwis. This benchmark, from Redis [53], simulates a Twitter-like social network, where users follow other users, make posts, and read a "timeline" of the most recent posts of all users they follow.

7.4 End-to-End Benchmarks

We first compare Apiary performance with the OW and RPC Servers baselines on all three microservice workloads, showing results in Figure 8. For each benchmark, we vary offered load (sent asynchronously following a uniform distribution) and observe throughput and latency. For all three workloads, maximum throughput achieved by Apiary is greater for Shop (1.2M RPS) than Hotel (144K RPS) and for Hotel than Retwis (20K RPS). This is because most Shop operations only access a single customer's cart, while most Hotel operations look up data for

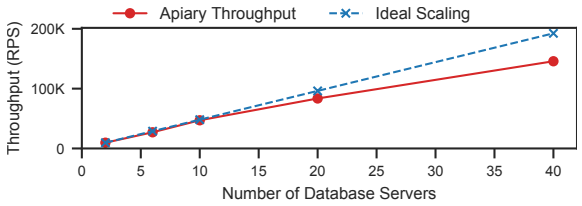

Figure 9: Maximum achievable throughput for Apiary on the Hotel benchmark with a varying number of database servers. We define "ideal scaling" by extrapolating linearly from a single replicated server (two servers total).

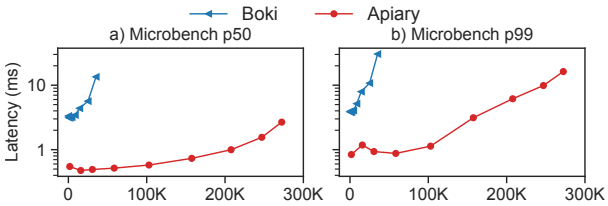

Figure 10: Throughput versus latency for Apiary (with full features enabled) and Boki on a microbenchmark.

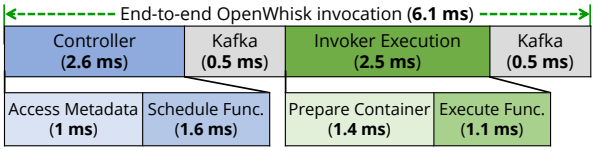

Figure 11: Latency breakdown for an OpenWhisk function invocation performing a point database update.

several hotels and most Retwis operations look up data for several dozen users ("Access Rows" in Table 1).

We find that Apiary significantly outperforms the RPC Servers baseline on two benchmarks and performs on par on the third – even though Apiary offers more features (like provenance capture) and stronger guarantees (like ACID functions and exactly-once semantics). Apiary outperforms the RPC Servers baseline due to reduced communication overhead; because it compiles services to stored procedures that run in the database server, it requires fewer round trips to perform database operations (Table 1). Moreover, Apiary eliminates RPCs between microservices because each service is implemented in Apiary functions. Apiary achieves 1.6–3.4× better median and tail latency than RPC servers on Shop and Hotel, where each function executes many database queries which each require an RTT in the baseline but not in Apiary. It matches baseline latency for Retwis, where each function executes only a single query. Apiary and RPC servers achieve similar maximum throughput for Hotel and Retwis, where throughput is bottlenecked by the database, but Apiary improves throughput by 1.75× for Shop, where the bottleneck is communication.

Apiary dramatically outperforms OW on all three benchmarks. Due to a combination of scheduling, bookkeeping, container initialization, message passing, and communication overhead (analyzed in Section 7.5), Apiary improves throughput by 7–68× and median and tail latency by 5–14× compared to OW.

The results of these experiments (and of our comparison with Boki in §7.5) establish that, for data-centric applications, disaggregating compute from storage leads to large inefficiencies. A disaggregated platform not only requires the same number of long-running storage servers as Apiary to host data, but also needs compute workers to handle application logic. However, because the application logic of a data-centric task contains many operations on data, workers are bottlenecked on communication overhead which Apiary avoids. Thus, disaggregation increases resource consumption and reduces performance, while an integrated system like Apiary reduces communication overhead and uses cluster resources more efficiently.

Scalability. We now evaluate the scalability of Apiary, measuring the maximum throughput Apiary can achieve with varying numbers of database servers. We show results for the Hotel benchmark in Figure 9, but obtained similar results for Shop and Retwis. We measure from 2 to 40 database servers (16 to 320 data partitions), beginning with 2 servers because each server needs a replica. We find that Apiary scales well; with larger numbers of servers, performance was mainly limited by VoltDB's overhead of managing a large network mesh between servers.

7.5 Microbenchmarks

Comparing with Boki. We compare Apiary and Boki performance using a simple microbenchmark that retrieves and increments a counter, similar to microbenchmarks used in the Boki paper. We implement the benchmark using Boki's durable storage API, BokiStore. We use this microbenchmark because Boki implements its own storage system, so a fair comparison on the microservice benchmarks is difficult. To improve Boki performance and reduce conflicts, we use 80K counters and have each function invocation pick a counter at random.

We show results in Figure 10. We find Apiary improves throughput by 7.7×, p50 latency by 6×, and p99 latency by 4.6× even though Apiary provides stronger guarantees (like ACID transactions) and more features (like provenance capture) than Boki. Although Boki has a local cache per worker, the cache miss rate is high because Boki does not provide affinity between data and workers. Therefore, most requests require a round trip to fetch a more recent value from remote storage (or another worker), adding communication overhead and harming performance.

OpenWhisk Performance Analysis. To explain the performance difference between Apiary and OW, we analyze OW performance on a microbenchmark consisting of a single OW function that retrieves and increments a counter stored in VoltDB. We invoke this function 100K times and measure the average latency of each step.

As we show in Figure 11, OW adds significant over-

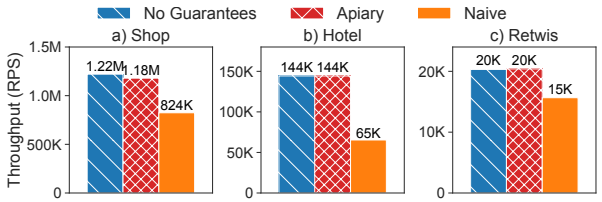

Figure 12: Maximum achievable throughput for Apiary without the exactly-once semantics guarantee, with the guarantee, and with a naive implementation of the guarantee.

head to a function invocation. Each invocation is first handled by a controller which performs basic operations such as bookkeeping and access control using function metadata stored in CouchDB (1 ms) before scheduling the invocation to an invoker/worker node (1.6 ms). OW uses Apache Kafka for controller-invoker communication, incurring 1 ms of round-trip latency. Once the invoker receives an invocation request, it needs to resume the execution of an already-warm container (1.4 ms). The function then executes in 1.1 ms. We emphasize that this high overhead is not unique to OW; other popular production FaaS systems have a similar architecture and performance characteristics. Apiary avoids this overhead because it combines compute and data and stores all state in the backend DBMS, thus reducing communication overhead and avoiding expensive external state management.

An additional factor limiting OW throughput is the low concurrency available in each OW worker. OW does not efficiently support more than 16 concurrent function containers on each of its 8 vCPU workers. Thus, maximum throughput is low because all containers are blocked on communication with the DBMS; their utilization ranges from 20% for the communication-heavy Retwis workload to 50% for the more lightweight Shop workload. By contrast, each Apiary frontend runs 128 lightweight concurrent dispatcher threads.

7.6 Apiary Performance Analysis

We now evaluate the performance impact of specific Apiary features and guarantees.

Exactly-Once Semantics Analysis. We first analyze the Apiary exactly-once semantics guarantee. As we discussed in Section 5.4, Apiary automatically instruments stored procedures to selectively log function outputs in the DBMS to guarantee consistency during failure recovery, using a novel dataflow analysis algorithm to minimize logging overhead. We evaluate the overhead of our guarantee and compare it to a more naive implementation (similar to prior work [29, 69]) that logs all function executions, showing results in Figure 12. We find that our guarantee incurs overhead of <5%, but this low overhead is only possible because we use dataflow analysis to log selectively: only 25% of Shop, 0.25% of Hotel, and 0.2% of Retwis function executions must be logged. By contrast, the naive

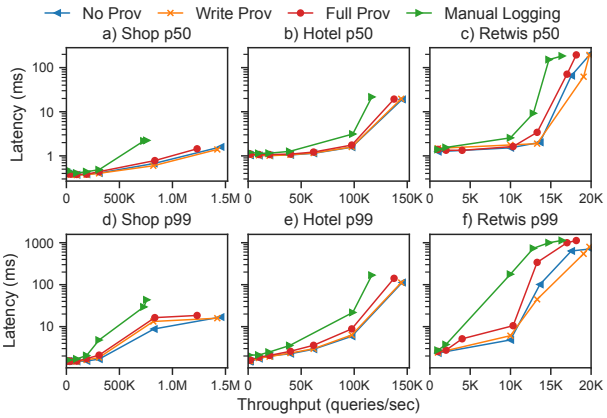

Figure 13: Throughput versus latency for Apiary with full provenance, only write capture, no provenance capture, and a manual logging baseline on all three benchmarks.

implementation reduces throughput by 1.3–2.2×.

Provenance Performance Analysis. As we discussed in Section 6, Apiary automatically instruments stored procedures to capture the read and write sets of operations, spooling this information to an analytical DBMS for long-term storage and analysis. To analyze the performance impact of provenance, we measure Apiary performance with both read and write capture enabled, with only write capture, and with no provenance capture. We also measure the performance of a "manual logging" baseline that represents how provenance information is captured in existing FaaS platforms: by logging to files on disk that are later exported by monitoring software like AWS Cloudwatch.

We show results for all experiments in Figure 13. We find that at low load, write capture has a negligible effect on latency but both read capture and manual logging increase latency (both median and tail) by up to 1.1×. At high load, write capture still has a negligible performance impact; read capture adds throughput overhead of up to 1.15× while manual logging adds overhead of up to 1.92×. Apiary provenance capture outperforms manual logging because, unlike conventional disaggregated FaaS platforms, it can buffer captured provenance data in the database's memory and export in large batches.

We next investigate whether storing and querying provenance data is practical. We execute 150M Shop operations, generating 1.2B rows of provenance data, and export this data to a single Vertica server, finding it is compressed to just 12.4GB of space on disk. We then execute all queries from Section 6.3 on this data, and find they all complete in 1.5–5.5 sec, which is reasonable for interactive querying.

8 Related Work

Function-as-a-Service Platforms. Many recent research projects seek to improve the performance and functionality of FaaS platforms on data-centric applications. We have already discussed several related systems, in-

cluding Cloudburst [58], Boki [29], Shredder [72], and Beldi [69], in Section 2. Similar systems to these include AFT [57], which uses an external transaction manager to bolt transactions onto a disaggregated FaaS platform, much like Beldi; and FaaSM [55], which allows functions to share memory regions, reducing data movement in long chains of functions, but does not support transactions.

Another set of relevant systems include Pocket [34], Locus [52], and Sonic [37], which propose multi-tier cloud storage backends designed for FaaS applications. These systems are largely designed for compute-intensive tasks on large amounts of data (e.g., batch analytics), where transactional and latency requirements are less strict. They trade these off for cost, as they can store data long-term in cheap cloud object stores like S3 while softening the performance impact by caching data in lower-latency systems like Redis. However, this tradeoff is not suitable for data-centric applications, which access smaller amounts of data but demand the low latency, strong transactional guarantees, and granular observability of Apiary.

Recently, cloud providers have released several serverless cloud database offerings, including Amazon Athena [5] and GCP Cloud Firestore [23]. These systems allow developers to store and query data without managing sever deployments. However, unlike Apiary or other FaaS platforms, they can only execute SQL queries on data and cannot perform general-purpose computation.

Data Provenance. Data provenance in relational databases is a well-studied subject [13, 28]. However, provenance information captured entirely in the database is difficult to use for debugging, monitoring, and auditing because it lacks high-level context such as which service performed what operations and for what purpose. Therefore, *workflow provenance* systems have been developed to capture this context.

Workflow provenance tracks the flow of data through a high-level application or service, typically viewed as a directed graph [14, 20, 28]. Workflow provenance is often used for scientific [3,26,43] and business [15,40,44] applications. The key challenge in these systems is capturing control flow information and linking it to data provenance information. Most existing systems rely on manual annotation and logging, requiring users to define the control flow graph then linking it to data provenance information captured through interposition. Some systems have proposed automatically capturing control flow information through kernel interposition [46] or dynamic analysis [47], but this information by itself is often too low-level and overwhelming for users [17, 49] so it must be supplemented with information from manual annotations [4, 17, 41, 46, 49].

Unlike these systems, Apiary interposes between functions and the DBMS to automatically capture workflow and data provenance without needing user annotations. While prior systems provide information flow control-based security for FaaS [2], we do not know of any prior work on data provenance tracking in FaaS environments.

9 Possible Extensions

We now describe two future directions for Apiary, leveraging tight integration of compute and data to solve problems that are difficult in existing systems.

Privacy. Both the research community and the world at large are increasingly recognizing the importance of privacy. Laws such as GDPR mandate that personally identifiable information (PII) can only be used for certain purposes and must be deleted upon request [50]. We believe that the tight integration of compute and data in Apiary can make it easier to implement privacy-preserving features. For example, using DBMS access control, a function could be associated with specific purposes, and its access to application tables containing PII is restricted based on those purposes. Moreover, using Apiary data provenance, as discussed in Section 6, organizations could audit and monitor applications for privacy violations.

Self-Adaptivity. There has been much recent research on applying machine learning (ML) to systems problems, including scheduling [39], database index construction [33,35], and DBMS configuration and tuning [61,71]. These propose using ML to automatically adapt or tune systems for particular workloads, often improving performance over existing heuristics [38]. A common challenge in these systems is collecting the data (e.g., system traces) needed for ML models and inference. However, as we have shown, the tight integration of compute and data in Apiary makes it easy to collect and analyze information on system and application behavior. For example, we may use the data provenance and cluster monitoring information captured by Apiary to enhance adaptive task scheduling using reinforcement learning [39].

10 Conclusion

In this paper, we have presented Apiary, a novel FaaS platform for data-centric applications. Apiary is the first system to *tightly integrate* function execution and data management in a FaaS context, breaking with the previous convention of disaggregating them. By using the same runtime for user code, database queries, and operational logging, Apiary gives developers a familiar high-level interface, guarantees functions run as ACID transactions with end-to-end exactly-once semantics, and offers advanced observability capabilities like automatic data provenance capture. Despite offering more features and stronger guarantees than existing FaaS platforms, Apiary outperforms them by 7–68× on realistic microservice applications by greatly reducing communication overhead and using cluster resources more efficiently.

References

[1] 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 *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, Boston, MA, July 2018. USENIX Association.

[2] Kalev Alpernas, Cormac Flanagan, Sadjad Fouladi, Leonid Ryzhyk, Mooly Sagiv, Thomas Schmitz, and Keith Winstein. Secure serverless computing using dynamic information flow control. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–26, 2018.

[3] Ilkay Altintas, Oscar Barney, and Efrat Jaeger-Frank. Provenance collection support in the kepler scientific workflow system. In *International Provenance and Annotation Workshop*, pages 118–132. Springer, 2006.

[4] Elaine Angelino, Daniel Yamins, and Margo Seltzer. Starflow: A script-centric data analysis environment. In *International Provenance and Annotation Workshop*, pages 236–250. Springer, 2010.

[5] AWS. Amazon Athena, 2021. https://aws.amazon.com/athena/.

[6] AWS. AWS Lambda Customer Case Studies, 2021. https://aws.amazon.com/lambda/resources/customer-case-studies/.

[7] AWS. AWS Rekognition, 2021. https://aws.amazon.com/rekognition/.

[8] AWS. How Do I Create a Serverless Workflow in Lambda?, 2021. https://aws.amazon.com/getting-started/hands-on/create-a-serverless-workflow-step-functions-lambda/.

[9] AWS. How do I make my Lambda function idempotent?, 2021. https://aws.amazon.com/premiumsupport/knowledge-center/lambda-function-idempotent/.

[10] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. Serverless computing: Current trends and open problems. In *Research advances in cloud computing*, pages 1–20. Springer, 2017.

[11] Jeff Barber, Ximing Yu, Laney Kuenzel Zamore, Jerry Lin, Vahid Jazayeri, Shie Erlich, Tony Savor, and Michael Stumm. Bladerunner: Stream processing at scale for a live view of backend data mutations at the edge. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 708–723, New York, NY, USA, 2021. Association for Computing Machinery.

[12] Andrzej Białecki, Robert Muir, Grant Ingersoll, and Lucid Imagination. Apache lucene 4. In *SIGIR 2012 workshop on open source information retrieval*, page 17, 2012.

[13] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. Provenance in databases: Why, how, and where. *Foundations and Trends® in Databases*, 1(4):379–474, 2009.

[14] Daniel Crawl, Jianwu Wang, and Ilkay Altintas. Provenance for mapreduce-based data-intensive workflows. In *Proceedings of the 6th workshop on Workflows in support of large-scale science*, pages 21–30, 2011.

[15] Francisco Curbera, Yurdaer Doganata, Axel Martens, Nirmal K Mukhi, and Aleksander Slominski. Business provenance–a technology to increase traceability of end-to-end operations. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 100–119. Springer, 2008.

[16] DataDog. DataDog's The State of Serverless, 2021. https://www.datadoghq.com/state-of-serverless/.

[17] Saumen Dey, Khalid Belhajjame, David Koop, Meghan Raul, and Bertram Ludäscher. Linking prospective and retrospective provenance in scripts. In *7th {USENIX} Workshop on the Theory and Practice of Provenance (TaPP 15)*, 2015.

[18] 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 *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 475–488, Renton, WA, July 2019. USENIX Association.

[19] 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 *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, Boston, MA, March 2017. USENIX Association.

[20] Juliana Freire, David Koop, Emanuele Santos, and Cláudio T Silva. Provenance for computational tasks: A survey. *Computing in Science & Engineering*, 10(3):11–21, 2008.

[21] Silvery Fu and Sylvia Ratnasamy. Dspace: Composable abstractions for smart spaces. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 295–310, New York, NY, USA, 2021. Association for Computing Machinery.

[22] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud and edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 3–18, New York, NY, USA, 2019. Association for Computing Machinery.

[23] GCP. Google Cloud Firestore, 2021. https://cloud.google.com/firestore.

[24] GCP. Google Cloud Microservices Demo (Online Boutique), 2021. https://github.com/GoogleCloudPlatform/microservices-demo.

[25] GCP. Retrying Event-Driven Functions, 2021. https://cloud.google.com/functions/docs/bestpractices/retries.

[26] Jeremy Goecks, Anton Nekrutenko, and James Taylor. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome biology*, 11(8):1–13, 2010.

[27] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. An evaluation of distributed concurrency control. *Proceedings of the VLDB Endowment*, 10(5):553–564, 2017.

[28] Melanie Herschel, Ralf Diestelkämper, and Houssem Ben Lahmar. A survey on provenance: What for? what form? what from? *The VLDB Journal*, 26(6):881–906, 2017.

[29] Zhipeng Jia and Emmett Witchel. Boki: Stateful serverless computing with shared logs. In *Symposium on Operating Systems Principles (SOSP 21)*. USENIX Association, November 2021.

[30] Zhipeng Jia and Emmett Witchel. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 152–166, 2021.

[31] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.

[32] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[33] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. Radixspline: A single-pass learned index. In *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, aiDM '20, New York, NY, USA, 2020. Association for Computing Machinery.

[34] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, Carlsbad, CA, October 2018. USENIX Association.

[35] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 489–504, New York, NY, USA, 2018. Association for Computing Machinery.

[36] Rodrigo Laigner, Yongluan Zhou, Marcos Antonio Vaz Salles, Yijian Liu, and Marcos Kalinowski. Data management in microservices: State of the practice, challenges, and research directions. *arXiv preprint arXiv:2103.00170*, 2021.

[37] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. SONIC: Application-aware data passing for chained serverless applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 285–301. USENIX Association, July 2021.

[38] Hongzi Mao, Parimarjan Negi, Akshay Narayan, Hanrui Wang, Jiacheng Yang, Haonan Wang, Ryan Marcus, ravichandra addanki, Mehrdad Khani Shirkoohi, Songtao He, Vikram Nathan, Frank Cangialosi, Shaileshh Venkatakrishnan, Wei-Hung Weng, Song Han, Tim Kraska, and Dr.Mohammad Alizadeh. Park: An open platform for learning-augmented computer systems. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[39] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 270–288, New York, NY, USA, 2019. Association for Computing Machinery.

[40] Axel Martens, Aleksander Slominski, Geetika T Lakshmanan, and Nirmal Mukhi. Advanced case management enabled by business provenance. In *2012 IEEE 19th International Conference on Web Services*, pages 639–641. IEEE, 2012.

[41] Timothy McPhillips, Tianhong Song, Tyler Kolisnik, Steve Aulenbach, Khalid Belhajjame, Kyle Bocinsky, Yang Cao, Fernando Chirigati, Saumen Dey, Juliana Freire, et al. Yesworkflow: a user-oriented, language-independent tool for recovering workflow information from scripts. *arXiv preprint arXiv:1502.02403*, 2015.

[42] Microsoft. Azure Durable Functions, 2021. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview.

[43] Paolo Missier, Khalid Belhajjame, Jun Zhao, Marco Roos, and Carole Goble. Data lineage model for taverna workflows with lightweight annotation requirements. In *International Provenance and Annotation Workshop*, pages 17–30. Springer, 2008.

[44] Luc Moreau. *The foundations for provenance on the web*. Now Publishers Inc, 2010.

[45] Ingo Müller, Renato Marroquín, and Gustavo Alonso. Lambada: Interactive data analytics on cold data using serverless cloud infrastructure. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 115–130, New York, NY, USA, 2020. Association for Computing Machinery.

[46] Kiran-Kumar Muniswamy-Reddy, David A Holland, Uri Braun, and Margo I Seltzer. Provenance-aware storage systems. In *Usenix annual technical conference, general track*, pages 43–56, 2006.

[47] Leonardo Murta, Vanessa Braganholo, Fernando Chirigati, David Koop, and Juliana Freire. noworkflow: capturing and analyzing provenance of scripts. In *International Provenance and Annotation Workshop*, pages 71–83. Springer, 2014.

[48] OpenWhisk. Apache OpenWhisk, 2021. https://openwhisk.apache.org/.

[49] João Felipe Pimentel, Saumen Dey, Timothy McPhillips, Khalid Belhajjame, David Koop, Leonardo Murta, Vanessa Braganholo, and Bertram Ludäscher. Yin & yang: demonstrating complementary provenance from noworkflow & yesworkflow. In *International Provenance and Annotation Workshop*, pages 161–165. Springer, 2016.

[50] Eugenia Politou, Efthimios Alepis, and Constantinos Patsakis. Forgetting personal data and revoking consent under the gdpr: Challenges and proposed solutions. *Journal of Cybersecurity*, 4(1):tyy001, 2018.

[51] The ZeroMQ project. JeroMQ, Pure Java implementation of libzmq, 2021. https://github.com/zeromq/jeromq.

[52] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 193–206, Boston, MA, February 2019. USENIX Association.

[53] Salvatore Sanfilippo. Retwis, 2021. https://github.com/antirez/retwis.

[54] Mohammad Shahrad, Rodrigo Fonseca, Inigo 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 *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218. USENIX Association, July 2020.

[55] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 419–433. USENIX Association, July 2020.

[56] SingleStore. SingleStore: The Single Database for All Data-Intensive Applications, 2021. https://www.singlestore.com/.

[57] Vikram Sreekanti, Chenggang Wu, Saurav Chhatrapati, Joseph E. Gonzalez, Joseph M. Hellerstein, and Jose M. Faleiro. A fault-tolerance shim for serverless computing. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[58] 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. *Proc. VLDB Endow.*, 13(12):2438–2452, July 2020.

[59] Michael Stonebraker, Samuel Madden, Daniel J Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: It's time for a complete rewrite. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*, pages 463–489. 2018.

[60] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proc. VLDB Endow.*, 8(3):245–256, nov 2014.

[61] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 1009–1024, New York, NY, USA, 2017. Association for Computing Machinery.

[62] Vertica. Vertica, 2021. https://www.vertica.com/.

[63] VoltDB. VoltDB, 2021. https://www.voltdb.com/.

[64] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, Boston, MA, July 2018. USENIX Association.

[65] Mike Wawrzoniak, Ingo Müller, Rodrigo Fraga Barcelos Paulus Bruno, and Gustavo Alonso. Boxer: Data analytics on network-enabled serverless platforms. In *11th Annual Conference on Innovative Data Systems Research (CIDR'21)*, 2021.

[66] Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. Autoscaling tiered cloud storage in anna. *Proc. VLDB Endow.*, 12(6):624–638, February 2019.

[67] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proceedings of the VLDB Endowment*, 10(7):781–792, 2017.

[68] YugabyteDB. YugabyteDB: The Global Scalable Resilient distributed SQL Database, 2021. https://www.yugabyte.com/.

[69] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1187–1204. USENIX Association, November 2020.

[70] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. Caerus: NIMBLE task scheduling for serverless analytics. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 653–669. USENIX Association, April 2021.

[71] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 415–432, New York, NY, USA, 2019. Association for Computing Machinery.

[72] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. Narrowing the gap between serverless and its state with storage functions. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 1–12, New York, NY, USA, 2019. Association for Computing Machinery.

[73] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 724–739, New York, NY, USA, 2021. Association for Computing Machinery.