Apiary: A DBMS-Backed Transactional Function-as-a-Service Framework

Anonymous Authors

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 e-commerce sites or social networks. Unfortunately, existing FaaS platforms support these applications poorly because they separate application logic, executed in cloud functions, from data management, done in ad-hoc transactions accessing a remote storage system. This separation harms performance and makes it hard to provide applications with transactional guarantees or monitoring and observability features.

We present Apiary, a novel transactional FaaS framework for data-centric applications. Apiary *tightly integrates* application logic and data management, building a unified runtime for function execution, data and transaction management, and operational logging by wrapping a distributed database engine and its stored procedures. Apiary augments the DBMS with scheduling and tracing layers, providing a graph-based programming model for composing functions into larger programs with end-to-end exactly-once semantics and advanced observability capabilities like automatic data provenance capture. In addition to offering more features and stronger guarantees than existing FaaS platforms, Apiary outperforms them by 7–68× on realistic microservice applications by greatly reducing communication and coordination overhead and using cluster resources more efficiently.

ACM Reference Format:
Anonymous Authors. 2023. Apiary: A DBMS-Backed Transactional Function-as-a-Service Framework. In *Proceedings of SIGMOD (SIGMOD '23).* ACM, New York, NY, USA, 15 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Function-as-a-service (FaaS), or serverless, cloud offerings are becoming popular in both industry [9, 15, 65, 77, 87] and research applications [3, 4, 25, 26, 37, 38, 55, 59, 67, 70, 71, 78, 83, 84, 86]. 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 [39].

FaaS is especially promising for building *data-centric* applications: low-latency and transactional applications such as a travel reservation web service or an e-commerce microservices application. In fact, these applications were the original motivation for FaaS [7] as they underlie much of the web. Unfortunately, existing FaaS platforms support data-centric applications poorly because they separate application logic, executed in cloud functions, from data management, done in ad-hoc transactions accessing a remote database. This 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 remote storage system [37, 67, 71]. Second, it makes providing transactional guarantees for functions difficult because the FaaS platform may arbitrarily re-execute ad-hoc transactions [12, 30, 83]. Third, it complicates observability, which is critical for application developers [35], because traces must span many tasks and an entirely separate storage system and thus are difficult to collect and interpret [14].

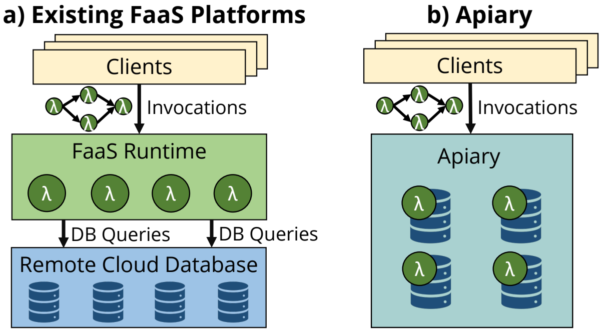

Figure 1: Existing FaaS platforms separate application logic, executed in cloud functions, from data management, done in ad-hoc transactions accessing a remote database. Apiary instead tightly integrates application logic and data management, executing functions in DBMS stored procedures.

Recently, there has been much research on building FaaS platforms for data-centric applications to tackle these problems [37, 43, 47, 63, 67, 71, 86]. However, these systems still separate function execution and data management and thus only offer partial solutions. Some, like Cloudburst [71] and Boki [37], locally cache data to improve performance, but this does nothing to provide transactions or observability. Others, like Beldi [83], provide transactions using an external transaction manager bolted onto remote storage, increasing already-high storage access times by as much as 3×.

To address these problems, we propose Apiary, a novel transactional FaaS framework optimized for data-centric applications. Instead of separating application logic and data management, Apiary *tightly integrates* them (Figure 1b) by leveraging a classic idea from database systems: it compiles user functions to database *stored procedures* to make functions transactional and dramatically improve their performance relative to other FaaS systems. However, naively using stored procedures is insufficient to build realistic FaaS programs, which consist of graphs of many functions and require end-to-end guarantees and tracing, because traditional DBMSs do not efficiently support *composing* stored procedures into larger programs. Therefore, Apiary *augments* the DBMS with scheduling and tracing layers that can execute complex graph-based programs

SIGMOD '23, June 18–23, 2023, Seattle, WA

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

with end-to-end guarantees and track information across function boundaries to provide advanced observability capabilities. Apiary provides a conventional FaaS interface to developers: they write functions in a high-level language, then compose them into graphs describing larger programs. Because of its novel data-centric design, Apiary can provide features and guarantees not present in existing FaaS platforms, like transactional functions, end-to-end exactly-once semantics for program execution, and advanced observability capabilities like automatic data provenance capture, while outperforming them by 7-68×.

Designing a scheduling layer for Apiary is challenging because it must efficiently execute large FaaS programs comprised of many functions while providing end-to-end guarantees such as exactly-once semantics. Commercial FaaS systems provide these guarantees by requiring developers to make all functions idempotent so they can be freely re-executed [12, 30], but this is an onerous requirement [83]. Naively, we could execute large FaaS programs in a conventional DBMS by nesting transactions so an entire program runs as one large transaction, but this imposes prohibitive coordination costs. Therefore, to efficiently support realistic FaaS programs in Apiary, we implement a graph-based programming model for composing functions and a frontend service for executing graphs. Then, to provide exactly-once semantics, we automatically instrument functions to transactionally record their executions in the DBMS so the frontend can safely re-execute them. However, naively instrumenting all functions is prohibitively expensive, degrading performance up to 2.2×. Therefore, we develop a novel dataflow analysis algorithm to identify when functions can be safely re-executed, providing exactly-once semantics with <5% overhead.

Apiary also tackles another common challenge faced by FaaS users: performing efficient and interpretable tracing for observability. Apiary leverages its unified runtime to do automatic tracing, instrumenting stored procedures to provide *data provenance capture*: recording all reads from and writes to each database record. Conventional DBMSs can capture data provenance information, but this information is hard to interpret as it lacks application-specific context such as what high-level operation invoked a transaction and for what purpose [34]. Crucially, Apiary provides that context because *functions are a natural unit of control flow tracking*. Apiary groups captured data provenance information by function, recording values read and written by each function. This 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 representative of business use cases such as social networking and e-commerce, adapted from sources such as DeathStarBench [27] and Google Cloud [29]. We find that by tightly integrating function execution with data management and reducing communication overhead, Apiary outperforms the popular open-source FaaS platform OpenWhisk [58] by 7–68× and the recent performance-optimized research system Boki [37] by up to 7.7×. Moreover, Apiary reduces the cost of deploying a FaaS application by 2.2–2.8× compared with OpenWhisk and Google Cloud Functions, even for a realistic time-varying workload, by using cluster resources more efficiently.

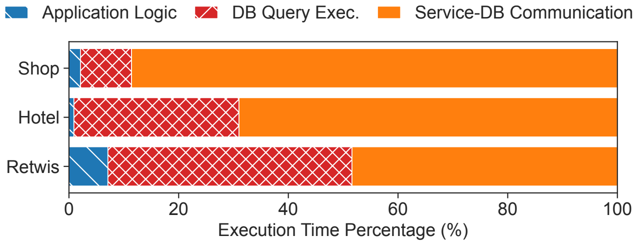

Figure 2: Average execution time breakdown for three data-centric applications. All three applications spend the vast majority of their runtime communicating with the database (over long-lived connections) or executing database operations.

In summary, our contributions are:

- We demonstrate that the separation of function execution and data management in FaaS platforms makes it difficult to provide transactions, observability, and good performance for increasingly popular data-centric applications.
- We propose Apiary, a novel transactional FaaS framework for data-centric applications. By leveraging DBMS stored procedures to tightly integrate function execution and data management, Apiary provides transactional guarantees for functions and exactly-once semantics for whole programs 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 observability 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 [21, 38]. Three representative example workloads, which we also use in our evaluation, include:

- **Shop:** 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 [29].
- **Hotel:** A hotel reservation service that allows users to find hotels near a geographic location, retrieve information about those hotels, and reserve a room [27].
- **Retwis:** A social network that allows users to create multimedia posts and view and interact with other users' posts [64].

Critically, much like other data-centric applications, all three workloads consist 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. To make this point more concrete, in Figure 2 we break down the runtime of each application implemented with a conventional microservice architecture performing application logic in long-running RPC servers. As we can see, all three data-centric applications spend the vast majority of their runtime either communicating with the

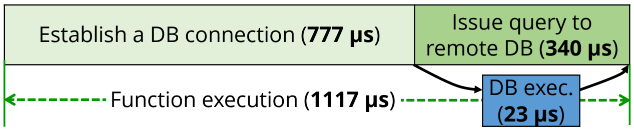
Establish a DB connection (777 µs)
Issue query to remote DB (340 µs)
Function execution (1117 µs)
DB exec. (23 µs)

Figure 3: 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.

DBMS or executing DBMS operations; application logic other than DB operations accounts for only 1-7% of runtime.

2.2 Issues of Existing FaaS Platforms

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 cloud storage.

Unfortunately, existing FaaS platforms do not live up to this promise for data-centric applications. As we argued earlier, this is largely because existing platforms separate function execution from data management, causing several problems:

Performance. Data-centric applications typically perform low-latency operations on data, like point lookups and updates. In a high-performance in-memory DBMS, each of these operations usually takes at most a few tens of microseconds. However, as we show in Figure 3, 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, this can be reduced by a connection pool but OpenWhisk does not support these natively), 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 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 manage data using ad hoc transactions, 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 [12, 30] or building an external transaction manager [83].

Observability. In order to debug, monitor, and audit data-centric applications, developers require information on *data provenance*: what operations occurred on each database record. However, in existing FaaS platforms this information is hard to collect and interpret [14]. 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 many 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.

2.3 Current Approaches in Data-Centric FaaS

Many recent research projects seek to improve the performance and functionality of FaaS platforms on data-centric applications. The most relevant to Apiary are Cloudburst [71], Boki [37], Shredder [86], and Beldi [83]. Cloudburst [71] proposes using the autoscaling key-value store Anna [79] as a FaaS storage backend then improving data access performance with local caches. Boki [37] proposes a FaaS storage backend based on shared logs that uses local caches to improve performance. Shredder [86] allows developers to define low-latency "storage functions" that run on the same server as a key-value store hosting their data. Beldi [83] 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: the separation of function execution and data management. Thus, they struggle to provide transactional guarantees or observability features. Beldi and Boki provide transactions, but require a costly external transaction manager, increasing the overhead of remote cloud storage access by an additional 2.4–3.3×. Cloudburst and Boki cache data locally to improve performance, but offer only key-value interfaces with weak consistency models (causal consistency for Cloudburst, monotonic reads and read-your-writes for Boki). Shredder runs functions on data servers, but offers no transactions and does not support distributing application data across multiple servers.

2.4 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 [26] or batch analytics [63]. 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 [10] 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-value data [37, 67, 71, 83, 86]. We have found that most of the data-centric applications we examine are compatible with the relational model. We believe that Apiary

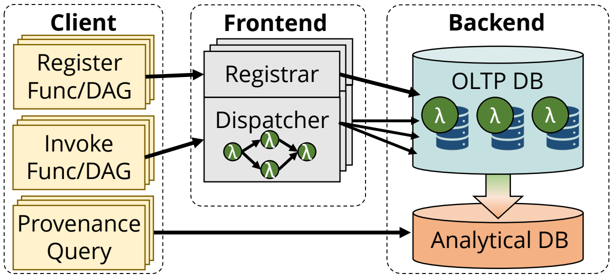

Figure 4: Architecture of Apiary.

could incorporate other data models, like Lucene indexes [16] for text search, but leave that for future work.

3 Apiary Overview

To solve the challenges raised in Section 2, we design Apiary to *tightly integrate* function execution and data management. Apiary wraps a distributed DBMS, compiling functions to stored procedures so it can leverage the DBMS to build a unified 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 4. It has three layers: 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, manages data, and handles operational logging. It wraps a distributed transactional DBMS and its stored procedures. Apiary executes functions transactionally on DBMS servers and automatically captures provenance data. 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 [11, 37, 52, 71], they compose functions into graphs comprising high-level operations. Apiary provides ACID guarantees for functions and exactly-once semantics for graphs.

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 5. Functions are the basic unit of atomicity in Apiary: we guarantee functions execute as ACID transactions and expect developers will put operations that must run transactionally in the same function.

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 5: The Apiary graph and function interfaces.

Apiary targets modern distributed DBMSs like VoltDB[76], SingleStore[69], and YugaByte[82], 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. Because local functions require no coordination, they are significantly faster than global functions, so we expect that, as in many other database systems [40, 72], most 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 functionality 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

Realistic applications consist of many functions composed together into a larger program and often require end-to-end guarantees for the entire program. In Apiary, developers can build these higher-level programs by composing functions into directed and acyclic *graphs* with a guarantee of exactly-once semantics for graph execution. We sketch the graph interface in Figure 5, diagram an example graph in Figure 6, and code a graph function in Figure 7.

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 Figure 7. 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

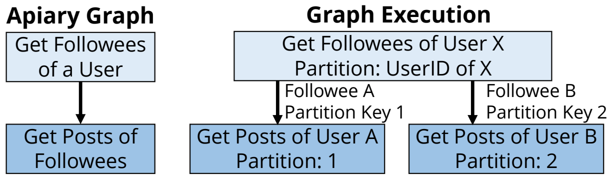
Apiary Graph
Get Followees of a User
Get Posts of Followees
Graph Execution
Get Followees of User X Partition: UserID of X
Followee A Partition Key 1
Followee B Partition Key 2
Get Posts of User A Partition: 1
Get Posts of User B Partition: 2

Figure 6: 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.

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

Figure 7: Implementing getFollowees from Figure 6 using the Apiary interface. This is a local function accessing the Followees table; the partition key is userID.

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 6, 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 their ID as its partition key. The function would then execute ten times, each execution retrieving data from a different partition.

To build complex programs, many applications require end-to-end exactly-once semantics: the guarantee that each function execution in a graph completes exactly once, regardless of failures. This is needed for important application design patterns like sagas [13] as well as for reliable messaging. Commercial FaaS platforms only provide exactly-once semantics if all functions are idempotent [12, 30] (an onerous requirement for developers [83]), while conventional databases do not provide cross-transaction guarantees like exactly-once semantics. Apiary automatically provides exactly-once semantics for graphs without imposing requirements on developers; we discuss how 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 (Section 5.4) and data provenance capture for observability (Section 6). Apiary builds on top of existing DBMSs so we can leverage their battle-tested infrastructure instead of designing a new data store from scratch. Apiary can be implemented using any distributed relational DBMS that has the following four properties:

- Supports ACID transactions.
- Supports running user code in a non-SQL language transactionally in stored procedures.
- Supports change data capture (for provenance, Section 6.2).
- Supports elastic DBMS cluster resizing.

Many DBMSs have these properties (e.g., Yugabyte [82] and SingleStore [69]); we ultimately chose VoltDB [76].

5.1 Cluster Overview

An Apiary cluster is made up of a DBMS backend and an automatically-scaled number of stateless frontend servers that interface between clients and the backend. For isolation, we provide each application with its own backend, which consists of a distributed database whose servers are each located in their own container.

The Apiary backend relies on its underlying DBMS for autoscaling. For example, our implementation utilizes VoltDB's elastic scaling capabilities, resizing each backend cluster using a utilization-based heuristic. We rely on DBMS auto-scaling because the data-centric applications we target are computationally bottlenecked by database operations. As we show in Figure 2, they spend 93-99% of their runtime communicating with the database or executing database operations. Thus, the separation of function execution and data management in other FaaS platforms is counterproductive: as we demonstrate in Sections 7.4 and 7.7, the additional communication overhead incurred by offloading application logic to another server harms performance and increases the cost of deployment.

5.2 Compilation

To improve performance and provide transactional guarantees, Apiary compiles all user functions to stored procedures, routines in a non-SQL language that run natively as DBMS transactions. Compilation happens in two steps. First, Apiary instruments each function to record its executions in the database for exactly-once semantics (Section 5.4) and to capture provenance information for observability (Section 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. Apiary extends the DBMS stored procedure interface, so we can compile any function that:

- Uses the Apiary interface (Figure 5) 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. We can pass function inputs and outputs through the dispatcher because, in our experience, they are typically small. 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 nontrivial. We discuss in Section 5.4 how Apiary efficiently mitigates dispatcher failures by selectively persisting function outputs to provide exactly-once semantics.

5.4 Exactly-Once Semantics

Many FaaS applications require an end-to-end guarantee of exactly-once semantics for graph execution, for example to support important design patterns like sagas [13] or to provide reliable messaging. However, providing exactly-once semantics efficiently is challenging because conventional DBMSs do not provide guarantees across separate transactions and executing a graph as a single transaction adds prohibitive coordination overhead. Providing exactly-once semantics is also a problem for existing FaaS platforms, which either require all functions be idempotent [12, 30] or use costly external transaction managers [83].

Two scenarios can lead to violation of exactly-once semantics: failures of DBMS servers and failures of Apiary dispatchers. Most distributed DBMSs can recover from failures of their 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.

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 *record* function outputs in the DBMS before returning. Then, if a dispatcher fails, any clients currently communicating with that dispatcher can send pending graph invocations to another dispatcher, using a client-generated per-invocation unique ID to signal a re-execution. In case of client failures, Apiary can also record this unique ID and graph metadata ahead-of-time in the database so dispatchers can automatically re-execute partially executed graphs without client involvement. Each re-executed function first checks for a record from the earlier execution; if it finds one it returns the recorded output instead of executing.

Our experiments (Section 7.6) show that naively recording 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 recorded. 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 record 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 recorded and which can be safely re-executed.

Algorithm 1 Fault tolerance decision

```
function FINDRECORDEDNODES($DAG$)
    $\{f_1,...,f_n\}$ = topoSort($DAG$)                ▷ $f_1$ is source, $f_n$ is sink.
    $Recorded$ = {}
    for $f_i \in \{f_n...f_1\}$ do                     ▷ Traverse from sink back to source.
        if hasWrite($f_i$) then
            $Recorded$.add($f_i$)
        else
            ▷ Use BFS to find all paths to the recorded nodes and sink.
            $Paths$ = BFSFindPaths($f_i$, $Recorded \cup \{f_n\}$)
            if hasDisjointPaths($Paths$) then
                $Recorded$.add($f_i$)
    return $Recorded$
```

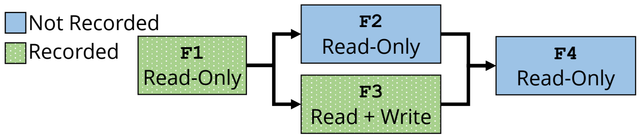

Figure 8: Example of Apiary recording rules. F3 is recorded as it performs a write. F1 must also be recorded to avoid inconsistent outputs to F2 and F3.

We must record 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 record a read-only function if there exist disjoint paths from it to multiple different recorded functions, or to at least one recorded function and the sink. This guarantees that two recorded 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 recorded local function can fan out to multiple executions (Section 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 8. F3 is recorded for performing writes, but F1 is also recorded despite being read-only. Suppose F1 was not recorded 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 recorded output based on the original output of F1. This causes an inconsistency that violates exactly-once semantics, so we must record F1 to prevent it. The worst-case complexity of this algorithm is $O(n^2)$ where n is the number of functions in a graph. Since we only run this algorithm once and off the critical path (when the graph is registered) and the number of functions in a graph is typically at most a few hundred [27], our algorithm is practical and would not affect graph invocation latencies.

6 Provenance and Observability

Developers often need information on data *provenance* to debug, monitor, and audit their systems. For example, they may wish to know who was the last user to modify a record, or what the state of a record was when a function reading it crashed. However, this information is difficult to collect both in existing FaaS systems, where it is spread across a vast number of ephemeral tasks and a separate storage system, and in conventional DBMSs, which lack high-level context about applications.

In this section, we show how Apiary solves these problems by augmenting a distributed DBMS with a tracing layer that provides automatic *data and workflow* provenance capture. Apiary records not only the full history of function execution but also traces of all DBMS operations, spooling information to an analytical database for storage and analysis. Therefore, Apiary can automatically supply provenance data along with its high-level application workflow context. We provide several examples of critical monitoring, debugging, and auditing queries that are difficult in existing FaaS platforms but straightforward in Apiary.

6.1 Apiary Provenance Interface

Provenance Capture. Apiary automatically captures the *data provenance* of each database record: the set of all database operations that accessed or modified it. 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.

To make data provenance information interpretable, Apiary augments it with *workflow provenance*. Apiary transparently instruments user functions to record their executions (along with related metadata, such as the higher-level program the function belonged to and the user who invoked it) and associate them with data provenance information. Thus, from a user perspective, Apiary captures data provenance information at the function level, recording all values functions read from and write to the DBMS. This makes many business-critical queries easy; for example, it is simple to look up who was the last user to update a record and how they updated it. This 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 [6, 23, 51, 56, 60].

Provenance Query Interface. Apiary automatically spools provenance information to an analytical database (in our implementation, Vertica [75]) 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. Long-term storage policies such as data retention rules are implemented by this database. 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 an event table 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 metadata 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 (Figure 5). 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 (Section 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 [27, 29, 64]. We additionally use microbenchmarks to analyze the performance of Apiary's features and guarantees. We show that:

(1) By tightly integrating function execution and data management, Apiary improves data-centric FaaS application performance by 7–68× compared to production FaaS systems and 7.7× compared to recent research systems (Figures 9, 11).

(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% (Figure 13).

(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% (Figure 14).

7.1 Experimental Setup

We implement Apiary in ~10K lines of Java code[1]. We use VoltDB [76] v9.3.2 as our DBMS backend and Vertica [75] v10.1.1 as our analytics database. For communication between clients and frontend servers, we use JeroMQ [62] v0.5.2 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 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) [58] 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. Additionally, 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 sub-clusters.

RPC Servers. Most microservices today are deployed in long-running RPC servers with separate application and database server machines [27, 45]. We implement each of our workloads this way, performing all business logic in long-running RPC servers but storing data in an external VoltDB cluster. 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 [37] is a recent research system designed for data-centric FaaS. As it is co-designed with a disaggregated storage system, we use it unmodified. We use the experimental setup described in the Boki paper, deploying 8 storage nodes, 3 sequences, and 8 workers, each on an AWS EC2 `c5d.2xlarge` instance with 8 vCPUs and 16GB DRAM. 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.

Shop. This benchmark, from Google Cloud [29], simulates an e-commerce service, where users browse an online store, update items from their shopping cart, and check out items for purchase.

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

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.

Hotel. This benchmark, from the DeathStarBench [27] suite, simulates searching and reserving hotel rooms.

Retwis. This benchmark, from Redis [64], 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 our three microservice workloads, showing results in Figure 9. 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 access a single customer's cart, while most Hotel operations look up data for several hotels and most Retwis operations access 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 Section 7.5) establish that, for data-centric applications, separating function execution from data management leads to large inefficiencies. A conventional FaaS 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 is computationally bottlenecked by database operations (as shown in Figure 2), these workers contribute little but add significant communication overhead which Apiary avoids. Thus, the tightly integrated architecture of Apiary reduces communication overhead, uses resource more efficiently, and, as we will show in Section 7.7, reduces the cost of deployment.

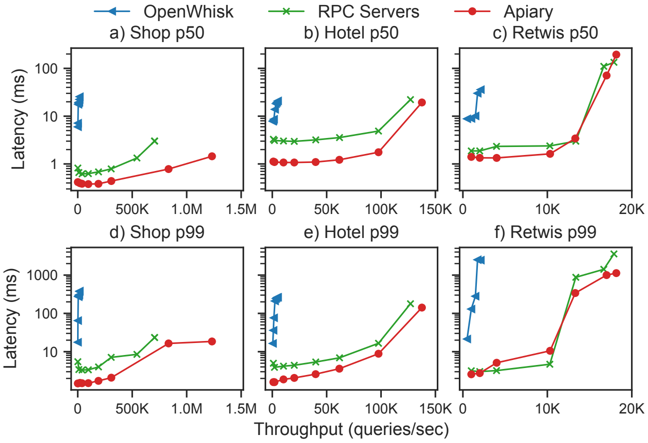

Figure 9: Throughput versus latency for Apiary (with full features enabled) and the OpenWhisk (OW) and RPC Servers baselines on all three benchmarks.

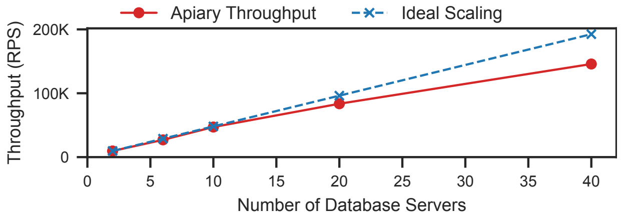

Figure 10: Maximum achievable throughput for Apiary on the Hotel benchmark with a varying number of database servers. We define "ideal scaling" by extrapolating linearly from a single replicated server (two servers total).

Scalability. We also 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 10, 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 the microbenchmarks used in the Boki paper. We implement the benchmark using Boki's durable storage API, BokiStore. We use this microbenchmark because Boki

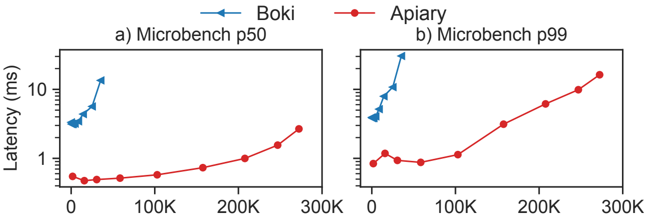

Figure 11: Throughput versus latency for Apiary (with full features enabled) and Boki on a microbenchmark.

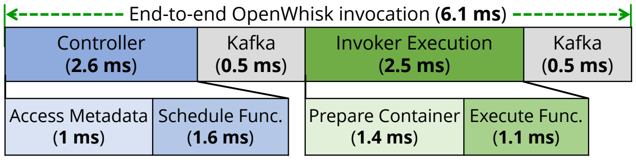

Figure 12: Latency breakdown for an OpenWhisk function invocation performing a point database update.

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 11. 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 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 12, OW adds significant overhead 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 integrates function execution and data management 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.

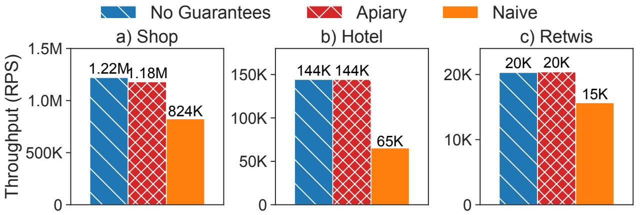

Figure 13: Maximum achievable throughput for Apiary without the exactly-once semantics guarantee, with the guarantee, and with a naive implementation of the guarantee.

7.6 Apiary Performance Analysis

Exactly-Once Semantics Analysis. We now analyze the performance impact of the Apiary exactly-once semantics guarantee. As we discussed in Section 5.4, Apiary automatically instruments stored procedures to selectively record function outputs in the DBMS to guarantee consistency during failure recovery, using a novel dataflow analysis algorithm to minimize recording overhead. We evaluate the overhead of our guarantee and compare it to a more naive implementation (similar to prior work [37, 83]) that records all function executions, showing results in Figure 13. We find that our guarantee incurs overhead of <5%, but this low overhead is only possible because we use dataflow analysis to record selectively: only 25% of Shop, 0.25% of Hotel, and 0.2% of Retwis function executions must be recorded. By contrast, the naive 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 14. 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 existing 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 compresses 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, reasonable for interactive querying.

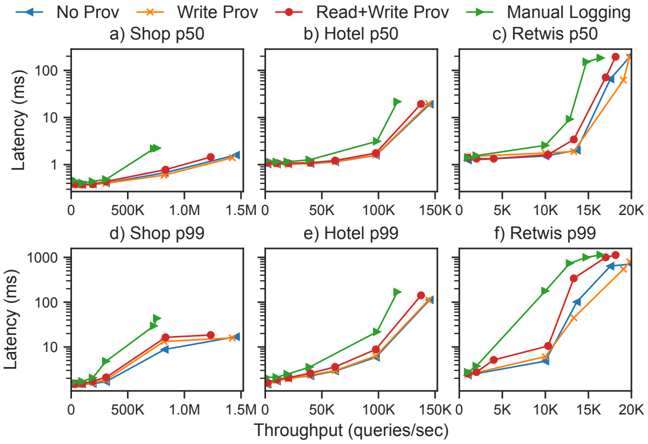

Figure 14: Throughput versus latency for Apiary with both read and write capture enabled, only write capture, no provenance capture, and a manual logging baseline on all three benchmarks.

System	Low Load 100 QPS	Mid Load 10K QPS	High Load 1M QPS	Mixed Load
OW + VoltDB	$1,221	$16,312	$1,521,253	$34,986
RPC S. + VoltDB	$917	$1,831	$36,204	$12,888
GCF+Firestore	**$204**	$18,595	$1,857,859	$27,792
Apiary + VoltDB	$917	**$1,526**	**$25,915**	**$12,635**

Table 2: Estimated monthly total cost for OpenWhisk, RPC Servers, Google Cloud Function (GCF) with Firestore, and Apiary serving the Shop workload on GCP[2], under different loads.

7.7 Cost Analysis

Finally, we evaluate the cost of deploying Apiary to the cloud. In Table 2, we estimate the monthly total cost of serving the Shop workload (we observe similar trends for Hotel and Retwis) on GCP for Apiary and the OW and RPC Servers baselines. We also wanted to compare with a commercial FaaS platform using a serverless database, so we chose Google Cloud Functions using Google's serverless database Firestore [28] as a backend. We evaluate four different load patterns: low, medium, and high patterns consisting of a uniform 100 QPS, 10K QPS, and 1M QPS, as well as a mixed pattern of 50% low load, 49% medium load, and 1% high load, representing diurnal variance with unpredictable spikes. For all systems except GCF+Firestore (Firestore is pay-per-request), we provision the database cluster based on the peak load of the given query pattern and conservatively assume no DBMS scaling during the month. We assume OW can dynamically scale its workers and controllers to minimize cost at a given load, RPC Servers can scale its stateless microservice workers, Apiary can scale its stateless frontend servers, and GCF and Firestore are pay-per-request.

As expected, we find that Apiary minimizes cost compared to both OpenWhisk and RPC Servers because it reduces communication overhead and uses resources more efficiently. Even though our analysis conservatively assumes Apiary does not scale its database, we find OW and GCF+Firestore are actually costlier than RPC Servers or Apiary. At low load, OW is on par with Apiary and GCF+Firestore is cheaper because it can scale to near-zero. However, at medium and high load OW is 8.9–58.7× costlier and GCF+Firestore is 12.2–71.7× costlier because their high overhead, documented in earlier sections, means they require far more resources than Apiary to support the same load. Even for mixed load, OW is 2.8× costlier and GCF+Firestore is 2.2× costlier because this overhead outweighs any benefit derived from separating function execution and data management Therefore, in a realistic web serving scenario, Apiary not only is faster and provides more features, but is also more cost-efficient than comparable systems.

8 Discussion

We now describe potential future directions for Apiary that further explore tight integration with the DBMS to solve problems that are difficult in existing systems, then discuss how future DBMSs could better support FaaS programming models.

8.1 Possible Extensions

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 [61]. There has already been much work on using DBMSs for privacy [36, 66, 73], and we believe the tight integration of data management with function execution in Apiary can make it even easier to implement end-to-end 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 [49], database index construction [42, 44], and DBMS configuration and tuning [74, 85]. These propose using ML to automatically adapt or tune systems for particular workloads, often improving performance over existing heuristics [48]. 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 function execution and data management 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 [49].

8.2 DBMS Support for FaaS

Multi-Tenant Databases. Currently, Apiary provides isolation between applications by running each application in its own separate distributed database. However, we may potentially increase resource utilization and provide powerful cross-application guarantees by running different applications in a multi-tenant database. This would require solving two interesting challenges. First, performance isolation: leveraging prior work on fair scheduling in multi-tenant systems [20, 68] and real-time databases [1, 33] to ensure all FaaS applications receive only their fair share of cluster resources. Second, security isolation: leveraging sandboxing research [2, 3, 81, 86] to isolate stored procedures containing arbitrary user code from one another and from the database.

[2] Prices were retrieved from the Google Cloud Pricing Calculator on 2022-04-11, using the us-west1 region as it was the cheapest: https://cloudpricingcalculator.appspot.com/.

Multi-Partition Stored Procedures. Apiary dramatically improves the performance of functions by compiling them to stored procedures. Existing DBMSs (including VoltDB) optimize stored procedures for functions that only access a single partition of data; they do not provide such large performance benefits for distributed stored procedures. This is acceptable because our target microservice and web serving workloads, like many transactional workloads [40, 72], are expressed using mostly single-partition operators, but we would still like to see more research into high-performance distributed stored procedures to better support other workloads.

9 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, including Cloudburst [71], Boki [37], Shredder [86], and Beldi [83], in Section 2. Similar systems to these include AFT [70], which uses an external transaction manager to bolt transactions onto a FaaS platform, much like Beldi; and FaaSM [67], which allows functions to share memory regions, reducing data movement in long chains of functions, but does not support transactions. Also related are systems like Hydrocache [80] and FaaSTCC [46] which provide transactional causal consistency for graphs of functions so they read from a consistent snapshot of data. This is a useful cross-function guarantee, but it comes at the cost of allowing serious anomalies such as write-write conflicts and stale reads; Apiary by contrast guarantees function serializability (where functions are the basic unit of consistency and atomicity), but retains high performance.

Another set of relevant systems include Pocket [43], Locus [63], and Sonic [47], 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 [8] and GCP Cloud Firestore [28]. There have also been recent research systems in this space, like Starling [59]. 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.

Provenance Tracking. Recent surveys [34] distinguish two types of provenance tracking in data management systems: *data provenance* [18], which traces the provenance of individual data items, and *workflow provenance* [22], which traces the flow of data through different units of computation (e.g., functions or system modules). Data provenance information provided alone without workflow provenance information is difficult to interpret because it lacks high-level context such as which service performed what operations and for what purpose. Moreover, providing fine-grained data provenance requires detailed knowledge of the semantics of individual operators (e.g., to determine which input tuples witness each output tuple, as in why-provenance [17]) and thus is difficult for general-purpose functions like those in Apiary. Therefore, Apiary provides both workflow provenance and coarse-grained data provenance: it tracks which functions execute, what high-level programs they belong to, and what data they read, write, and return.

Many existing frameworks provide workflow provenance, sometimes in combination with coarse-grained data provenance, for scientific [5, 31, 53] and business [19, 50, 54] applications. The key challenge in these systems is capturing control flow information and linking it to data flow information. Most existing systems rely on manual annotation and logging, requiring users to define the control flow graph then linking it to data flow information. Some systems have proposed automatically capturing control flow information through kernel interposition [56] or dynamic analysis [57], but this information by itself is often too low-level and overwhelming for users [23, 60] so it must be supplemented with information from manual annotations [6, 23, 51, 56, 60].

Unlike existing provenance systems, Apiary interposes between functions and the DBMS and leverages control flow information inherent in its FaaS programming model to automatically capture workflow and data provenance without needing user annotations. While prior systems provide information flow control-based security for FaaS [4], we do not know of any prior work on data provenance tracking in FaaS environments.

In-Database Computation. Most relational DBMSs can run procedural code as either user-defined functions (UDFs) or stored procedures [32]. UDFs are pure functions that cannot modify persistent state but can be invoked from within SQL queries. Stored procedures are less restrictive and can run arbitrary procedural code as part of a database transaction. Apiary therefore compiles user functions to database stored procedures to run them natively as performant transactions on database servers. While many DBMSs support running user code natively in stored procedures [24, 32, 41, 76], to our knowledge none support composing them together into larger programs with end-to-end guarantees and tracing like Apiary.

10 Conclusion

In this paper, we presented Apiary, a novel transactional FaaS framework for data-centric applications. Apiary is the first system to *tightly integrate* function execution and data management in a FaaS context, breaking with the convention of separating them. By wrapping a distributed DBMS and its stored procedures with new scheduling and tracing layers, Apiary gives developers a familiar FaaS interface and a graph-based programming model for composing functions into larger programs and also guarantees functions run as ACID transactions, provides end-to-end exactly-once semantics for graphs, and offers advanced observability capabilities like automatic data provenance capture. In addition to offering more features and stronger guarantees than existing FaaS platforms, Apiary outperforms them by 7–68× on realistic microservice applications by greatly reducing communication and coordination overhead and using cluster resources more efficiently.

References

[1] Robert K Abbott and Hector Garcia-Molina. 1992. Scheduling real-time transactions: A performance evaluation. *ACM Transactions on Database Systems (TODS)* 17, 3 (1992), 513–560.

[2] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 419–434. https://www.usenix.org/conference/nsdi20/presentation/agache

[3] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 923–935. https://www.usenix.org/conference/atc18/presentation/akkus

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

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

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

[7] AWS. 2014. Amazon Web Services Announces AWS Lambda. https://press.aboutamazon.com/news-releases/news-release-details/amazon-web-services-announces-aws-lambda.

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

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

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

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

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

[13] Microsoft Azure. 2022. Saga distributed transactions pattern. https://docs.microsoft.com/en-us/azure/architecture/reference-architectures/saga/saga.

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

[15] Jeff Barber, Ximing Yu, Laney Kuenzel Zamore, Jerry Lin, Vahid Jazayeri, Shie Erlich, Tony Savor, and Michael Stumm. 2021. 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* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 708–723. https://doi.org/10.1145/3477132.3483572

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

[17] Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. 2001. Why and where: A characterization of data provenance. In *International conference on database theory*. Springer, 316–330.

[18] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. 2009. Provenance in Databases: Why, How, and Where. *Foundations and Trends® in Databases* 1, 4 (2009), 379–474. https://doi.org/10.1561/1900000006

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

[20] Sudipto Das, Vivek R. Narasayya, Feng Li, and Manoj Syamala. 2013. CPU Sharing Techniques for Performance Isolation in Multi-Tenant Relational Database-as-a-Service. *Proc. VLDB Endow.* 7, 1 (sep 2013), 37–48. https://doi.org/10.14778/2732219.2732223

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

[22] Susan B. Davidson and Juliana Freire. 2008. Provenance and Scientific Workflows: Challenges and Opportunities. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 1345–1350. https://doi.org/10.1145/1376616.1376772

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

[24] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 1243–1254. https://doi.org/10.1145/2463676.2463710

[25] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 475–488. https://www.usenix.org/conference/atc19/presentation/fouladi

[26] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. 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)*. USENIX Association, Boston, MA, 363–376. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi

[27] 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. 2019. 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* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 3–18. https://doi.org/10.1145/3297858.3304013

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

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

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

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

[32] Surabhi Gupta and Karthik Ramachandra. 2021. Procedural Extensions of SQL: Understanding Their Usage in the Wild. *Proc. VLDB Endow.* 14, 8 (apr 2021), 1378–1391. https://doi.org/10.14778/3457390.3457402

[33] Jayant R Haritsa, Miron Livny, and Michael J Carey. 1991. *Earliest deadline scheduling for real-time database systems*. Technical Report. University of Wisconsin-Madison Department of Computer Sciences.

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

[35] Kasun Indrasiri and Prabath Siriwardena. 2018. Observability. In *Microservices for the Enterprise*. Springer, 373–408.

[36] Zsolt István, Soujanya Ponnapalli, and Vijay Chidambaram. 2021. Software-Defined Data Protection: Low Overhead Policy Compliance at the Storage Layer is within Reach! *Proc. VLDB Endow.* 14, 7 (mar 2021), 1167–1174. https://doi.org/10.14778/3450980.3450986

[37] Zhipeng Jia and Emmett Witchel. 2021. Boki: Stateful Serverless Computing with Shared Logs. In *Symposium on Operating Systems Principles (SOSP 21)*. USENIX Association.

[38] Zhipeng Jia and Emmett Witchel. 2021. 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*. 152–166.

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

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

[41] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow.* 1, 2 (aug 2008), 1496–1499. https://doi.org/10.14778/1454159.1454211

[42] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: A Single-Pass Learned Index. In *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management* (Portland, Oregon) *(aiDM '20)*. Association for Computing Machinery, New York, NY, USA, Article 5, 5 pages. https://doi.org/10.1145/3401071.3401659

[43] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 427–444. https://www.usenix.org/conference/osdi18/presentation/klimovic

[44] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 489–504. https://doi.org/10.1145/3183713.3196909

[45] Rodrigo Laigner, Yongluan Zhou, Marcos Antonio Vaz Salles, Yijian Liu, and Marcos Kalinowski. 2021. Data Management in Microservices: State of the Practice, Challenges, and Research Directions. *arXiv preprint arXiv:2103.00170* (2021).

[46] Taras Lykhenko, Rafael Soares, and Luis Rodrigues. 2021. FaaSTCC: Efficient Transactional Causal Consistency for Serverless Computing. In *Proceedings of the 22nd International Middleware Conference* (Québec city, Canada) *(Middleware '21)*. Association for Computing Machinery, New York, NY, USA, 159–171. https://doi.org/10.1145/3464298.3493392

[47] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. 2021. SONIC: Application-aware Data Passing for Chained Serverless Applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 285–301. https://www.usenix.org/conference/atc21/presentation/mahgoub

[48] 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. 2019. Park: An Open Platform for Learning-Augmented Computer Systems. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2019/file/f69e505b08403ad2298b9f262659929a-Paper.pdf

[49] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning Scheduling Algorithms for Data Processing Clusters. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 270–288. https://doi.org/10.1145/3341302.3342080

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

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

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

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

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

[55] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 115–130. https://doi.org/10.1145/3318464.3389758

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

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

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

[59] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. 2020. Starling: A Scalable Query Engine on Cloud Functions. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 131–141. https://doi.org/10.1145/3318464.3380609

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

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

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

[63] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 193–206. https://www.usenix.org/conference/nsdi19/presentation/pu

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

[65] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad

[66] Supreeth Shastri, Vinay Banakar, Melissa Wasserman, Arun Kumar, and Vijay Chidambaram. 2020. Understanding and Benchmarking the Impact of GDPR on Database Systems. *Proc. VLDB Endow.* 13, 7 (mar 2020), 1064–1077. https://doi.org/10.14778/3384345.3384354

[67] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 419–433. https://www.usenix.org/conference/atc20/presentation/shillaker

[68] David Shue, Michael J. Freedman, and Anees Shaikh. 2012. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, Hollywood, CA, 349–362. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/shue

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

[70] Vikram Sreekanti, Chenggang Wu, Saurav Chhatrapati, Joseph E. Gonzalez, Joseph M. Hellerstein, and Jose M. Faleiro. 2020. A Fault-Tolerance Shim for Serverless Computing. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 15, 15 pages. https://doi.org/10.1145/3342195.3387535

[71] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst: Stateful Functions-as-a-Service. *Proc. VLDB Endow.* 13, 12 (July 2020), 2438–2452. https://doi.org/10.14778/3407790.3407836

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

[73] Michael Stonebraker, Timothy Mattson, Tim Kraska, and Vijay Gadepally. 2020. Poly'19 Workshop Summary: GDPR. *SIGMOD Rec.* 49, 3 (dec 2020), 55–58. https://doi.org/10.1145/3444831.3444842

[74] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-Scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1009–1024. https://doi.org/10.1145/3035918.3064029

[75] Vertica. 2021. Vertica. https://www.vertica.com/.

[76] VoltDB. 2021. VoltDB. https://www.voltdb.com/.

[77] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 133–146. https://www.usenix.org/conference/atc18/presentation/wang-liang

[78] Mike Wawrzoniak, Ingo Müller, Rodrigo Fraga Barcelos Paulus Bruno, and Gustavo Alonso. 2021. Boxer: Data Analytics on Network-enabled Serverless Platforms. In *11th Annual Conference on Innovative Data Systems Research (CIDR'21)*.

[79] Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. 2019. Autoscaling Tiered Cloud Storage in Anna. *Proc. VLDB Endow.* 12, 6 (Feb. 2019), 624–638. https://doi.org/10.14778/3311880.3311881

[80] Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. 2020. Transactional Causal Consistency for Serverless Computing. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 83–97. https://doi.org/10.1145/3318464.3389710

[81] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *2009 30th IEEE Symposium on Security and Privacy*. 79–93. https://doi.org/10.1109/SP.2009.25

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

[83] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. 2020. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 1187–1204. https://www.usenix.org/conference/osdi20/

presentation/zhang-haoran

[84] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. 2021. Caerus: NIMBLE Task Scheduling for Serverless Analytics. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 653–669. https://www.usenix.org/conference/nsdi21/presentation/zhang-hong

[85] 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. 2019. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 415–432. https://doi.org/10.1145/3299869.3300085

[86] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. 2019. Narrowing the Gap Between Serverless and Its State with Storage Functions. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3357223.3362723

[87] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. 2021. Faster and Cheaper Serverless Computing on Harvested Resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 724–739. https://doi.org/10.1145/3477132.3483580