The Design of Apiary: A Programming Environment for DBOS

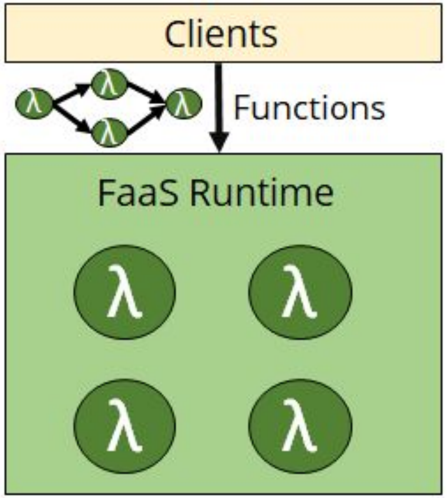
Peter Kraft and Qian Li

Question: How should developers program in DBOS?

Answer: DBOS should provide a function-as-a-service (FaaS) programming model!

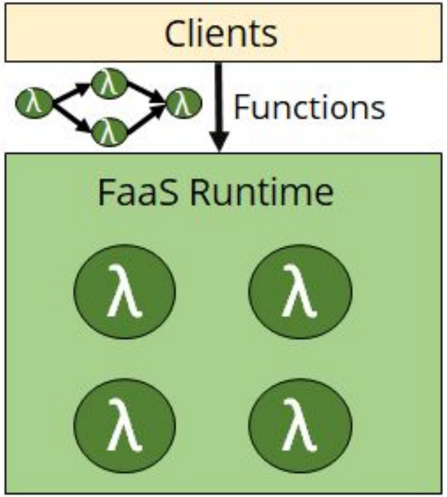
What is FaaS?

In the function-as-a-service (FaaS) model, users submit functions to a remote runtime which manages and executes them.



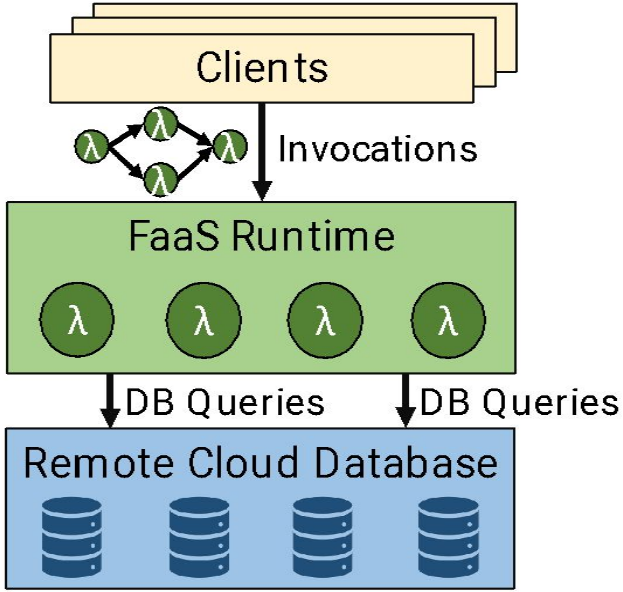
Why FaaS?

- FaaS abstracts away the need to manage your own servers and infrastructure—transparent failure recovery and auto-scaling!
- Reduces cost because you only pay for what compute you use.
- Our prototype targets applications–web services and microservices.

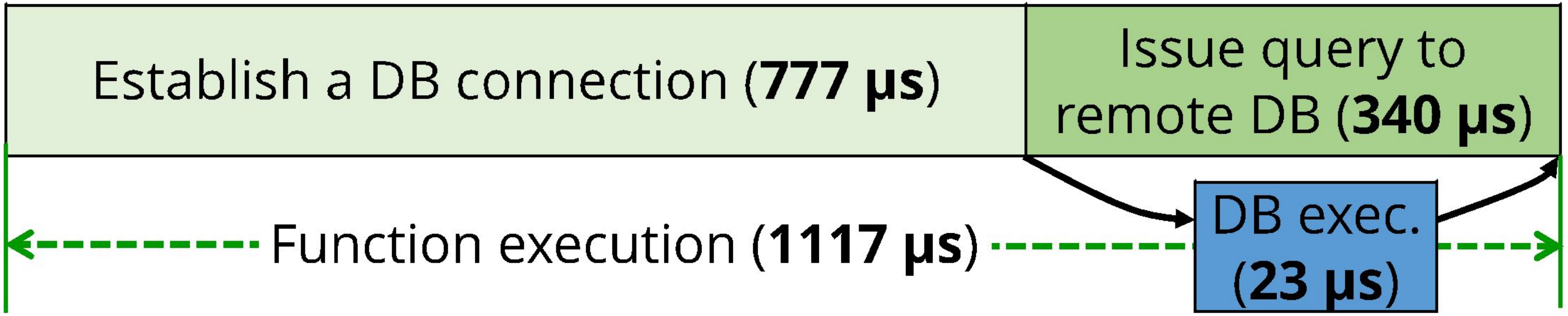


Existing FaaS Platforms Don’t Follow DBOS Ideas

- Existing FaaS platforms *separate* application logic (executed in cloud functions) and data management (done in interactive transactions).
- This is the opposite of DBOS.



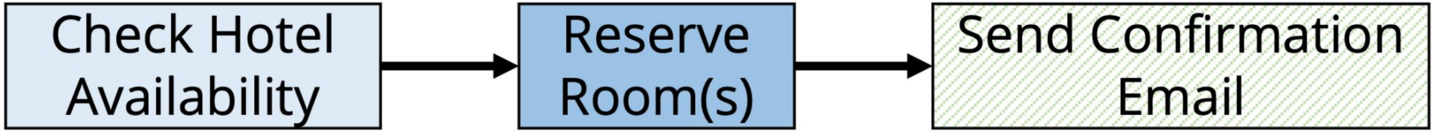
Issue #1: High Overhead on DB Operations


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

An OpenWhisk function performing a point update in an in-memory DB.
Query execution accounts for only **2%** of the overall execution time.

Issue #2: Weak Guarantees for Data Management

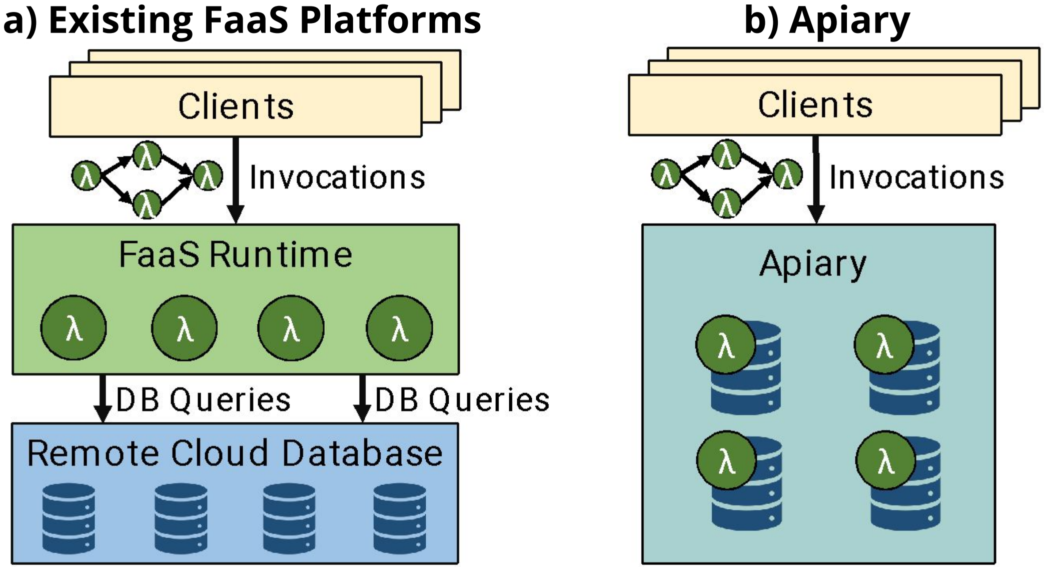
- Functions aren’t transactional, developers instead must manage interactive transactions in a remote database.
- No cross-function transactional guarantees.
- Functions are naively re-executed on failure, potentially replaying completed transactions and leading to unexpected errors.
    - Example: You may pay for a reservation twice 😭



Apiary: A DBOS-Inspired FaaS Platform

- Apiary tightly integrates function execution and data management: it wraps a distributed DBMS and executes functions transactionally as stored procedures.

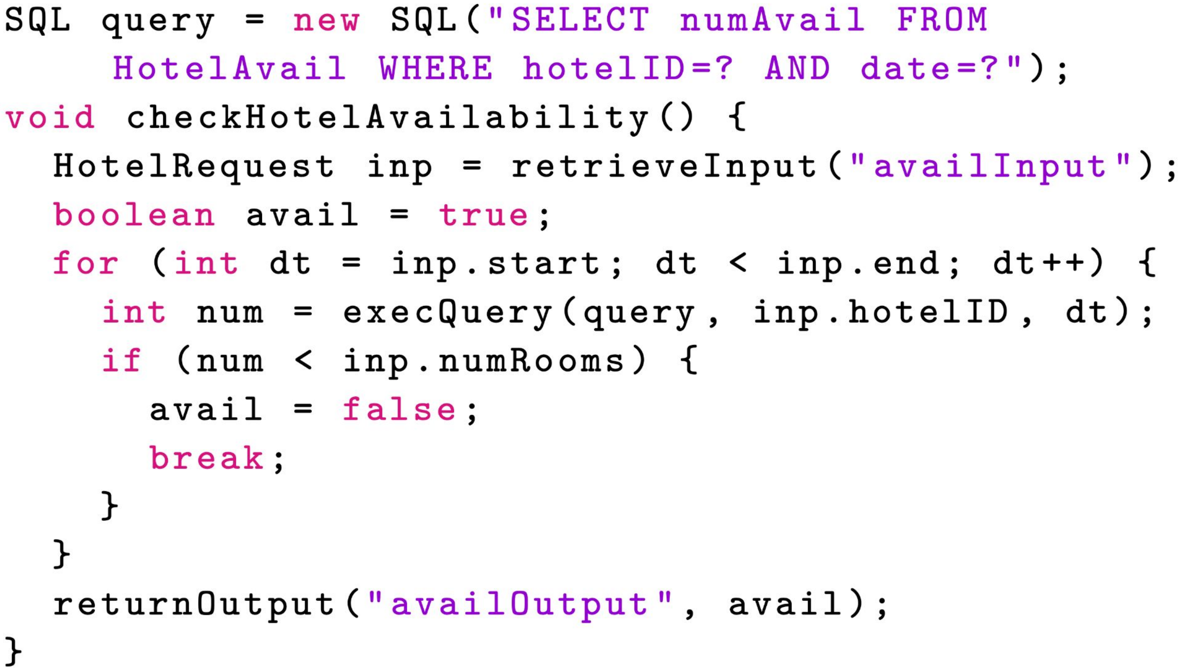
💡 **A classic idea from database systems!**



a) Existing FaaS Platforms

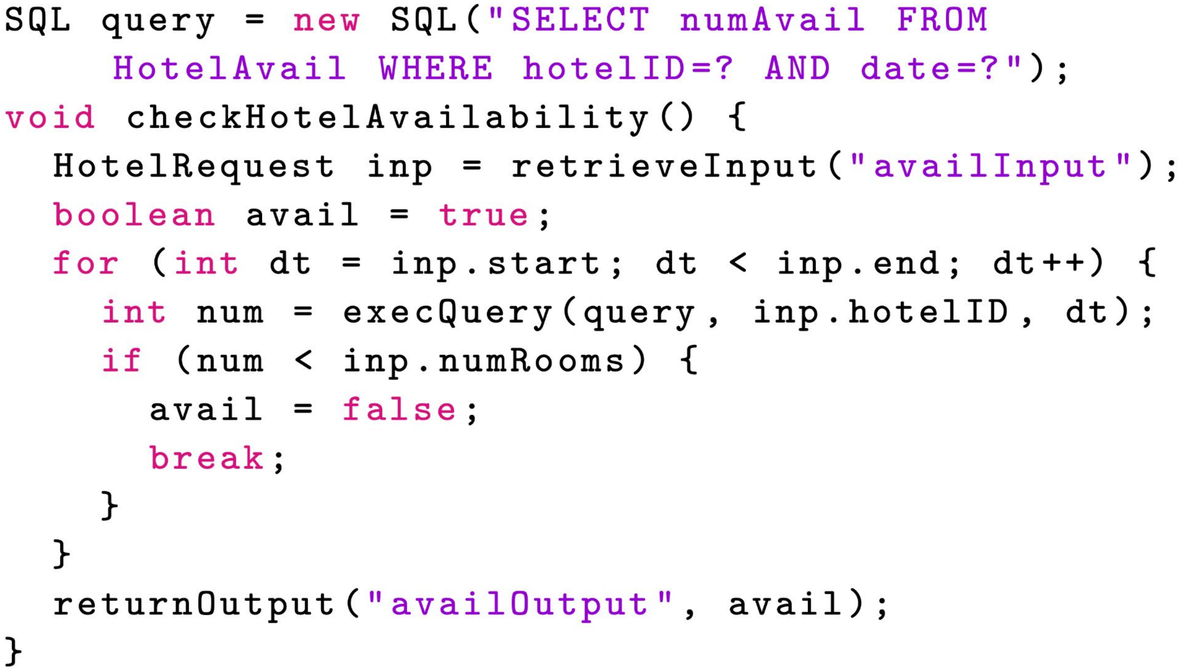
b) Apiary

Apiary Provides a Familiar Programming Interface

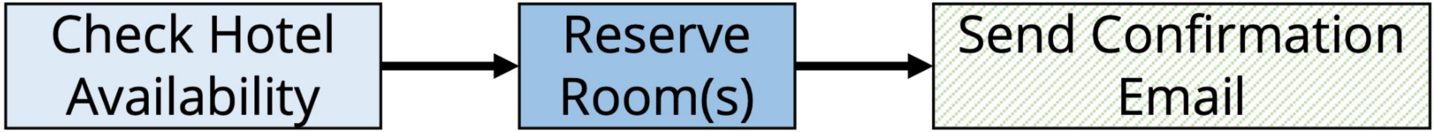


```
SQL query = new SQL("SELECT numAvail FROM
    HotelAvail WHERE hotelID=? AND date=?");
void checkHotelAvailability() {
  HotelRequest inp = retrieveInput("availInput");
  boolean avail = true;
  for (int dt = inp.start; dt < inp.end; dt++) {
    int num = execQuery(query, inp.hotelID, dt);
    if (num < inp.numRooms) {
      avail = false;
      break;
    }
  }
  returnOutput("availOutput", avail);
}
```

Apiary Functions are Composed into Larger Programs



```
SQL query = new SQL("SELECT numAvail FROM
     HotelAvail WHERE hotelID=? AND date=?");
void checkHotelAvailability() {
  HotelRequest inp = retrieveInput("availInput");
  boolean avail = true;
  for (int dt = inp.start; dt < inp.end; dt++) {
    int num = execQuery(query, inp.hotelID, dt);
    if (num < inp.numRooms) {
      avail = false;
      break;
    }
  }
  returnOutput("availOutput", avail);
}
```



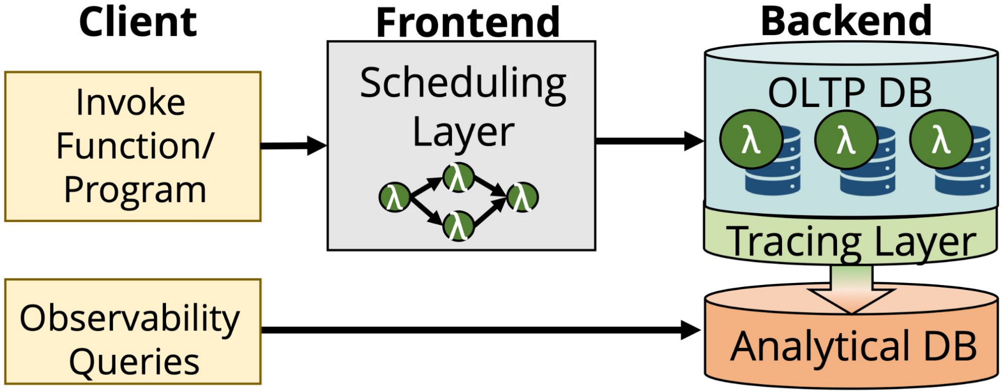
Check Hotel Availability

Reserve Room(s)

Send Confirmation Email

Apiary Builds Service Layers on top of the DBMS

- Scheduling layer: Executes programs, provides end-to-end guarantees (multi-function txns, exactly-once semantics).
- Tracing layer: Provides observability through data provenance tracking.



We’ll Discuss Three New Apiary Features

- Transactional guarantees.
- Exactly-once function execution semantics.

Scheduling Layer

- Automatic provenance capture for observability.

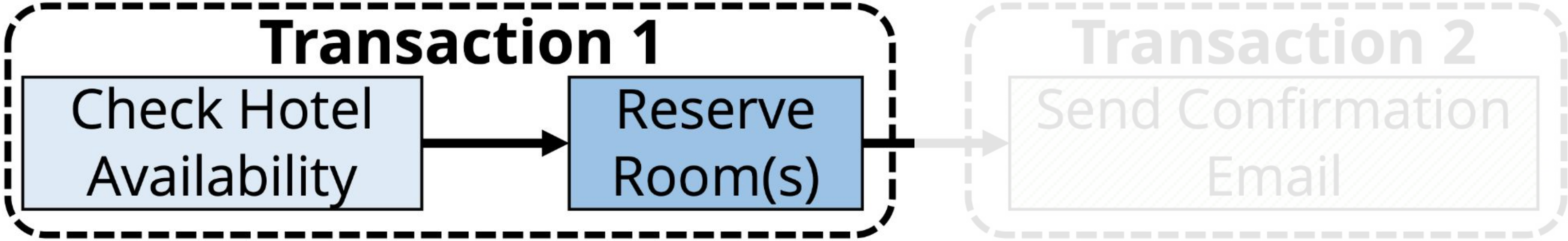
Tracing Layer

Apiary Functions are Database Transactions

- Apiary functions run transactionally as database stored procedures.
- Workflows are not transactional: transactions from separate workflows may interleave.

Apiary Provides Multi-Function Transactions

- Apiary functions run transactionally as database stored procedures.
- Workflows are not transactional: transactions from separate workflows may interleave.
- We provide multi-function transactions:
  - Example: first check room availability then reserve it.
  - We compile multiple functions into a single stored procedure.

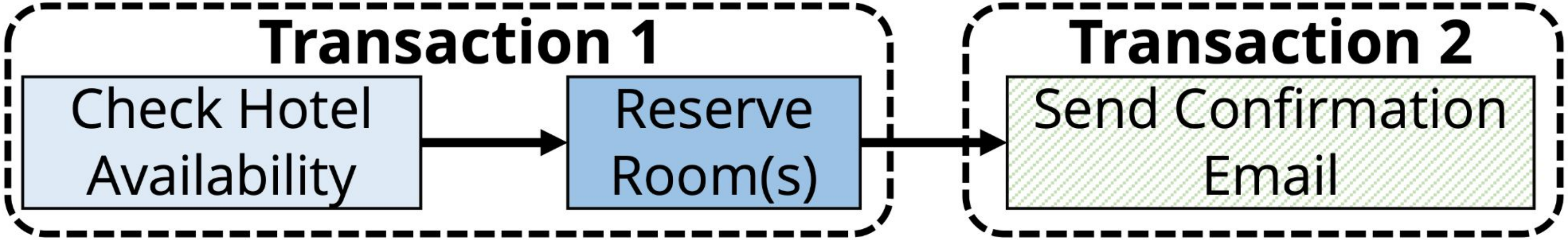


Apiary Executes Functions Exactly Once

- To guarantee reliable workflow executions, we need exactly-once function execution semantics.
- Example: We must guarantee that:

1) A room is only reserved once

2) Once reserved, a confirmation email is sent only once.



Apiary Guarantees Exactly-Once Using Transactions

- Our solution: transactionally record function outputs in the DBMS before a function returns.
- During failure recovery, check for the record in the database to avoid violation of exactly-once semantics.

Apiary Guarantees Exactly-Once Using Transactions

- Our solution: transactionally record function outputs in the DBMS before a function returns.
- During failure recovery, check for the record in the database to avoid violation of exactly-once semantics.
- Some functions can safely re-execute and need not be recorded. E.g., a read-only workflow.
- Through selective instrumentation, reduce runtime overhead from **2.2x** to **5%**

Apiary Enhances Observability Through Data Provenance

- Automatically instrument DB and functions to capture data provenance and full history of function executions.
- All logged information spooled to an analytical database like Amazon Redshift or Vertica, queried with SQL.

Captured Data Provenance Information

- **Execution history:** what operation executed and when.

  FunctionInvocations(timestamp, tx_id, function_name, ...)

- **Data access history:** what records did each transaction read from and write to the database?

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

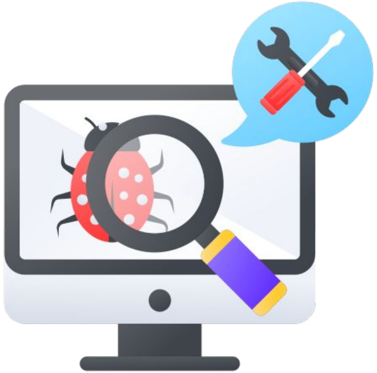
Example Data Provenance Query

- Downstream Provenance: Find all changes made by a request that earlier read sensitive information.

```
SELECT DISTINCT(record_id)
FROM TableEvents AS T,
     FunctionInvocations AS F
   ON T.func_id = F.func_id
WHERE T.event_type IN ('insert', 'update')
   AND F.function_name IN SUCCESSOR_FUNC_NAMES
   AND F.execution_id IN EXECUTION_IDS;
```

Extending Apiary Observability

- Building a transaction-oriented debugger.
- Everything is a transaction, enabling exciting debugging features:
  - Always-on tracing
  - Declarative debugging
  - Faithful replay
  - Retroactive programming

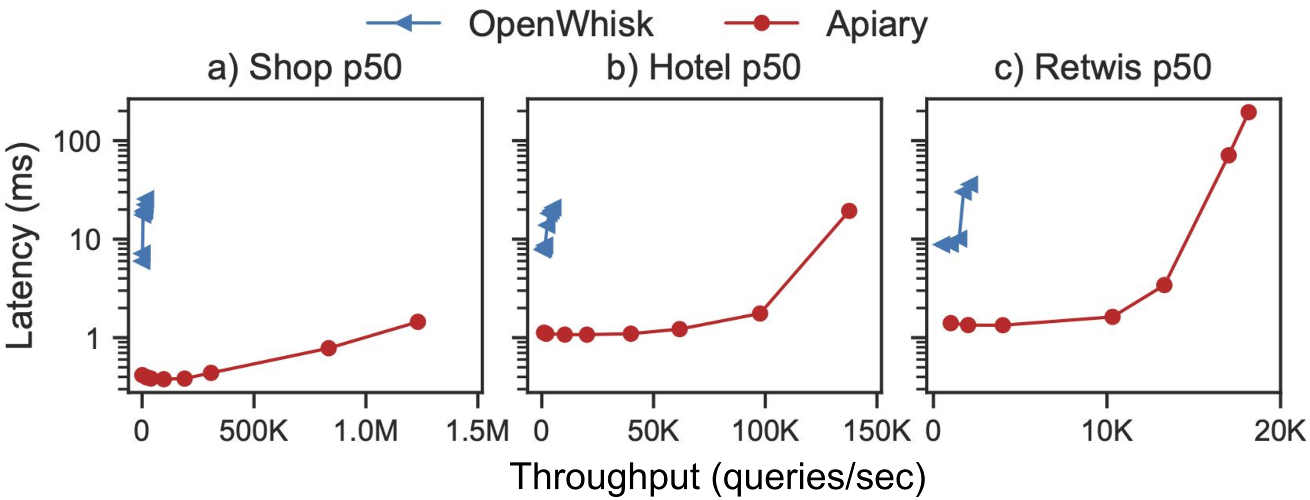


Faithful Replay and Retroactive Programming

- Insight: if functions are deterministic, and access shared state only transactionally, we can faithfully replay any past execution by:
  - Re-executing its code normally but…
  - Restoring the database before each transaction.
- Developers can **modify** their code and test it on past events.
- Eliminate most Heisenbugs :)

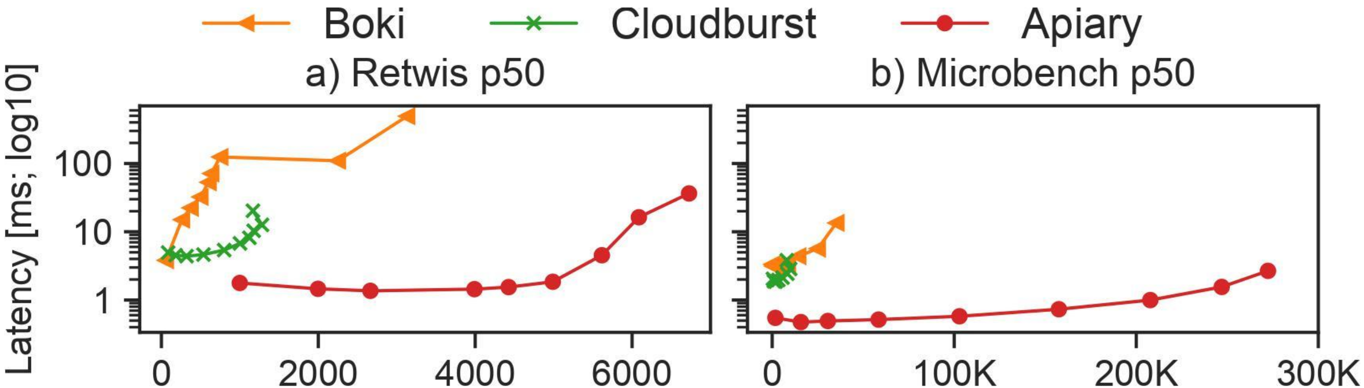
Evaluation

- A cluster of ~100 VMs on GCP. Microservice workloads.
- Outperform OpenWhisk (a popular production FaaS system) by **7--68x**: due to a combination of scheduling, container init, and communication.



Evaluation

- Compare with Cloudburst (VLDB’21) and Boki (SOSP'21), research systems for stateful FaaS.
- Improve performance by **2-27x** using stored procedures to minimize communication overhead.
- Apiary also provides stronger guarantees and observability.



https://github.com/DBOS-project/apiary