Apiary: Tightly Integrating Compute and Data for Efficient, Transactional, and Observable FaaS

Peter Kraft, Qian Li, and many others at Stanford, MIT, CMU, Wisconsin, BCG, VMWare, and Google
Question: Why do we need yet another FaaS Platform???
Answer: Data-Centric Applications

- Apps performing low-latency, transactional operations on data.
- Most microservices and web services are data-centric -> extremely important and ideal targets for FaaS.
- Examples: Confirming a travel reservation, adding an item to a customer’s online shopping cart, posting to a social network.
FaaS Can’t Handle Data-Centric Apps

- FaaS platforms *disaggregate* compute and data into separate subsystems.
- Example: AWS Lambda for compute, S3 or DynamoDB for data.
Why is Disaggregation Bad?

- Poor performance: Every data access requires an RTT.
Why is Disaggregation Bad?

- An OpenWhisk function performing a point database update. Query execution accounts for only 2% of the overall execution time.
Why is Disaggregation Bad?

- Poor performance: Every data access requires an RTT.
- Poor guarantees: Functions aren’t transactional, failures handled through naive retrying.
Why is Disaggregation Bad?

- Poor performance: Every data access requires an RTT.
- Poor guarantees: Functions aren’t transactional, failures handled through naive retrying.
- Poor observability: Traces spread across vast number of logs from tasks and a separate storage system.
A Radical Solution: Tight Integration

- Instead of disaggregating, tightly integrate compute and data!
- Use the same runtime for function execution, data management, and operational logging.
How Do We Tightly Integrate?

- Build Apiary on top of an in-memory relational distributed database!
- Compile user functions to database stored procedures.
- Instrument stored procedures to add an observability/monitoring layer and provide end-to-end guarantees like exactly-once semantics for function composition.
A Radical Solution: Tight Integration

a) Existing FaaS Platforms

Clients → Function Requests → FaaS Runtime

FaaS Runtime → DB Queries → Remote Cloud Database

b) Apiary

Clients → Function Requests → Apiary

Apiary → DB Queries
Why Do We Tightly Integrate?

- Performance: Functions access data locally, so >10x better performance than popular FaaS platforms.
Why Do We Tightly Integrate?

- **Performance:** Functions access data locally, so >10x better performance than popular FaaS platforms.
- **Strong Guarantees:** Functions are ACID transactions, we provide exactly-once semantics for function composition.
Why Do We Tightly Integrate?

- Performance: Functions access data locally, so >10x better performance than popular FaaS platforms.
- Strong Guarantees: Functions are ACID transactions, we provide exactly-once semantics for function composition.
- Observability: Easily trace function calls and what data each function accesses.
Apiary Programming Model

• **Familiar interface**: Developers write functions in a non-SQL language (currently Java) and embed SQL queries.
Example Function

```java
final SQLStmt checkExist = new SQLStmt(
    "SELECT Username FROM WebsiteLogins WHERE Username=\?;"
);

final SQLStmt addItem = new SQLStmt(
    "INSERT INTO WebsiteLogins VALUES (\?, \?);"
);

public boolean register (String username, String password) {
    String existUser = executeSQL(checkExist, username);
    if (existUser != null) {
        return false;
    }
    executeSQL(addItem, username, password);
    return true;
}
```
Apiary Programming Model

- **Familiar interface:** Developers write functions in a non-SQL language (currently Java) and embed SQL queries.
- To compose functions together, developers define *graphs* of many functions (similar to AWS Step Functions).
Apiary Programming Model

- **Familiar interface**: Developers write functions in a non-SQL language (currently Java) and embed SQL queries.
- To compose functions together, developers define *graphs* of many functions (similar to AWS Step Functions).
- We’re currently working on improving function composition and optimizing function scheduling and data transfer between functions.
End-to-End Guarantees

- Hard to provide end-to-end exactly-once semantics for graphs.
- DBMSs only guarantee individual transactions but not across them.
End-to-End Guarantees

- Hard to provide end-to-end exactly-once semantics for graphs.
- DBMSs only guarantee individual transactions but not across them.
- Existing FaaS platforms either:
  - Require all functions to be idempotent, simply re-execute (AWS Lambda, GCP Cloud Functions) -> hard to write applications.
  - Or use external transaction managers (Beldi OSDI'20, Boki OSDI'21) -> costly and slow.
End-to-End Guarantees

- **An obvious solution**: transactionally log function outputs in the DBMS before a function returns.
- During failure recovery, check for the log to avoid violation of exactly-once semantics.
End-to-End Guarantees

- **An obvious solution:** transactionally log function outputs in the DBMS before a function returns.
- During failure recovery, check for the log to avoid violation of exactly-once semantics.
- However, this approach harms performance up to 2.2x due to additional database lookups and updates for every execution.
End-to-End Guarantees

- **Our solution**: we recognize that some functions can safely re-execute and need not be logged. E.g., a read-only graph.
- Less than 5% overhead across all of our workloads.
End-to-End Guarantees

- **Intuition**: Always log writes; only log a read-only function if re-executing it could cause inconsistencies.
End-to-End Guarantees

- **Intuition**: Always log writes; only log a read-only function if re-executing it could cause inconsistencies.

  ![Flowchart](chart.png)

- **F3** is logged as it performs a write. **F1** must also be logged so it only executes once.
Apiary Observability

- **Automatically** instrument DB and functions to capture data provenance and full history of function executions.
- **Comprehensive** provenance because *functions are a natural unit of control flow tracking*. We provide high-level contexts for provenance.
Apiary Observability

- **Automatically** instrument DB and functions to capture data provenance and full history of function executions.
- **Comprehensive** provenance because *functions are a natural unit of control flow tracking*. We provide high-level contexts for provenance.
- All logged information spooled to an analytical database like AWS Redshift or Vertica.
- **Easily** query it with SQL, no specialized tools required!
Captured Data Provenance Information

- **Execution history**: what operation executed and when.
  
  \texttt{FunctionInvocations}(timestamp, tx\_id, function\_name, ...)

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

  \texttt{TableEvents}(timestamp, tx\_id, event\_type, [record\_data...])
Example Data Provenance Query

- What was the state of a table when this particular operation accessed it?
- Useful if the operation failed -- debug the failure!
Example Data Provenance Query

- What was the state of a table when this particular operation accessed it?
- Useful if the operation failed -- debug the failure!

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

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

- A cluster of ~100 VMs on GCP. Realistic microservice workloads.
Evaluation

- A cluster of ~100 VMs on GCP. Realistic microservice workloads.
- Outperform OpenWhisk (a popular production FaaS system) by 7--68x due to a combination of scheduling, container init, and communication.
- Apiary with full features enabled: fault tolerance, provenance.
Evaluation

- Compare with Boki (SOSP'21, a research system for stateful FaaS).
- Improve throughput by \(7.7x\), p50 latency by \(6x\), and p99 latency by \(4.6x\).
  - Boki disaggregates compute and data; remote data access.
- Apiary provides stronger guarantees.
Evaluation

- Apiary provenance capture adds overhead < 15% across workloads.
Conclusion

- We presented Apiary, a novel FaaS platform for data-centric applications.
  - Provide a familiar high-level interface;
  - Guarantee ACID transactions with end-to-end exactly-once semantics;
  - Offer advanced observability;
  - Outperform existing systems by 7--68x despite offering more features.
- Project website: [https://dbos.stanford.edu/](https://dbos.stanford.edu/)

Questions?
{qianli, kraftp}@cs.stanford.edu