Fault-tolerant and Transactional Stateful Serverless Workflows

Haoran Zhang, Adney Cardoza, Peter Baile Chen
Sebastian Angel, Vincent Liu
What is serverless?
What is serverless?

func MyAwesomeApp():
  return "Hello World"
What is serverless?

```
func MyAwesomeApp():
    return "Hello World"
```
What is serverless?

```go
golang
func MyAwesomeApp()
    return "Hello World"
```
What is serverless?

```
func MyAwesomeApp():
    return "Hello World"
```
What is serverless?

```
func MyAwesomeApp():
    return "Hello World"
```
What is serverless?

```
func MyAwesomeApp():
    return "Hello World"
```
What is serverless?

```
func MyAwesomeApp():
  return "Hello World"
```
What is serverless?

Workers can fail!

```
func MyAwesomeApp():
    return "Hello World"
```
How could serverless go wrong?

1. Send Request
2. Start
   - N = Read("attendees")
   - Write("attendees", N+1)
3. End
How could serverless go wrong?

Client

- Send Request
  - Receive Error / Timeout

Cloud

- Start
  - \(N = \text{Read(“attendees”)}\)
  - Write(“attendees”, \(N+1\))
  - End
How could serverless go wrong?

- **Send Request**
  - **Receive Error / Timeout**
    - **Should I Retry?**
  - **Start**
    - **N = Read(“attendees”)**
    - **Write(“attendees”, N+1)**
    - **End**
How could serverless go wrong?

1. Send Request
   - Receive Error / Timeout
     - Should I Retry?
   - Start
     - N = Read(“attendees”)
       - Write(“attendees”, N+1)
         - End

2. Cloud
3. Client
How do I make my Lambda function idempotent to prevent inconsistencies and data loss in my application?

I've heard that I should make my AWS Lambda function idempotent. What is this and how do I do it?

End

Write("a,endees",N+1)

N = Read("a,endees")

Start

Should I Retry?

Recieve Error / Timeout

Send Request

How could serverless go wrong?

Write Idempotent Functions!

Make retryable background functions idempotent

Background functions that can be retried must be idempotent. Here are some general guidelines for making a background function idempotent:

- Make sure that your code is internally idempotent. For example:
  - Make sure that mutations can happen more than once without changing the outcome.
  - Query database state in a transaction before mutating the state.
  - Make sure that all side effects are themselves idempotent.
- Impose a transactional check outside the function, independent of the code. For example, persist state somewhere recording that a given event ID has already been processed.
- Deal with duplicate function calls out-of-band. For example, have a separate clean up process that cleans up after duplicate function calls.

Cloud Client

Tips & Tricks

This document describes some tricks that you can use to solve problems while using Azure Functions.

Cloud Functions > Documentation

This section describes how to use Azure Functions to develop and deploy cloud applications.

Make retryable background functions idempotent

Background functions that can be retried must be idempotent. Here are some general guidelines for making a background function idempotent:

- Make sure that your code is internally idempotent. For example:
  - Make sure that mutations can happen more than once without changing the outcome.
  - Query database state in a transaction before mutating the state.
  - Make sure that all side effects are themselves idempotent.
- Impose a transactional check outside the function, independent of the code. For example, persist state somewhere recording that a given event ID has already been processed.
- Deal with duplicate function calls out-of-band. For example, have a separate clean up process that cleans up after duplicate function calls.

Write Idempotent Functions!

Make retryable background functions idempotent

Background functions that can be retried must be idempotent. Here are some general guidelines for making a background function idempotent:

- Make sure that your code is internally idempotent. For example:
  - Make sure that mutations can happen more than once without changing the outcome.
  - Query database state in a transaction before mutating the state.
  - Make sure that all side effects are themselves idempotent.
- Impose a transactional check outside the function, independent of the code. For example, persist state somewhere recording that a given event ID has already been processed.
- Deal with duplicate function calls out-of-band. For example, have a separate clean up process that cleans up after duplicate function calls.

Write Idempotent Functions!

Make retryable background functions idempotent

Background functions that can be retried must be idempotent. Here are some general guidelines for making a background function idempotent:

- Make sure that your code is internally idempotent. For example:
  - Make sure that mutations can happen more than once without changing the outcome.
  - Query database state in a transaction before mutating the state.
  - Make sure that all side effects are themselves idempotent.
- Impose a transactional check outside the function, independent of the code. For example, persist state somewhere recording that a given event ID has already been processed.
- Deal with duplicate function calls out-of-band. For example, have a separate clean up process that cleans up after duplicate function calls.

Write Idempotent Functions!
Beldi makes stateful serverless functions idempotent automatically!
Outline

• **Beldi’s Infrastructure**
• Linked DAAL
• Invocation with exactly-once semantics
• Evaluation
• Conclusion
Beldi’s architecture

Worker

Start

N = Read("attendees")

Write("attendees", N+1)

End

Beldi Runtime

Storage

Write("a7endees", N+1)

N = Read("attendees")
Beldi’s architecture

Worker

Start

N = Read(“attendees”)

Database API

Write(“attendees”, N+1)

End

Beldi Runtime

Storage
Beldi’s architecture

Worker

Start

N = Read("attendees")

End

Write("attendees", N+1)

Beldi Runtime

Database API

Invocation API

Storage
Beldi’s architecture

Worker

Start

N = Read("attendees")

Write("attendees", N+1)

End

Beldi Runtime

Database API

Invocation API

Transaction API

Storage
Beldi’s architecture

Start

N = Read("attendees")

Write("attendees", N+1)

End

Worker

Beldi Runtime

Storage

<table>
<thead>
<tr>
<th>Instance Id</th>
<th>Done</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Database API
- Invocation API
- Transaction API
Beldi’s architecture

Worker

Start

N = Read("attendees")

End

Write("attendees", N+1)

Beldi Runtime

Database API

Invocation API

Transaction API

Storage

Instance Id | Done
---|---

Key | Value
---|---

attendees | 10

Raw text:

Worker

Run.me

End

Write("attendees", N+1)

N = Read("attendees")

Start

Beldi's architecture
Beldi’s architecture

**Worker**

- **Start**
- **N = Read(“attendees”)**
- **Write(“attendees”, N+1)**
- **End**

**Beldi Runtime**

- **Database API**
- **Invocation API**
- **Transaction API**

**Storage**

<table>
<thead>
<tr>
<th>Instance Id</th>
<th>Done</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>attendees</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Value</th>
</tr>
</thead>
</table>
Beldi’s architecture

Worker:

Start

N = Read(“attendees”) -> Write(“attendees”, N+1)

End

Beldi Runtime:

Database API

<table>
<thead>
<tr>
<th>Instance Id</th>
<th>Done</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>attendees</td>
<td>10</td>
</tr>
</tbody>
</table>

Storage:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Value</th>
</tr>
</thead>
</table>

Beldi’s architecture

Worker

Start

N = Read(“attendees”)

Write(“attendees”, N+1)

End

Beldi Runtime

Storage

Instance Id | Done
--- | ---
d78590e | False

Key | Value
--- | ---
attendees | 10

Operation | Value
--- | ---
Worker

Start

N = Read("attendees")

Write("attendees", N+1)

End

Beldi Runtime

Progress Lambda

Database API

Instance Id | Done
-------------|--------
d78590e      | False

Key | Value
attendees | 10

Operation | Value

Storage
Worker

Start

N = Read("attendees")

Write("attendees", N+1)

End

Beldi Runtime

Progress Lambda

Database API

Storage

Instance Id

d78590e

Done

False

Key

attendees

Value

10

Operation

Value

Beldi’s architecture
Beldi’s architecture

Worker

Start

N = Read(“attendees”)

End

Beldi Runtime

Progress Lambda

Database API

Storage

Instance Id | Done
---|---
d78590e | False

Key | Value
attendees | 10

Operation | Value
d78590e-1 | 10
Worker

Start

N = Read("attendees")

Write("attendees", N+1)

End

Beldi Runtime

Progress Lambda

Database API

Instance Id | Done
-------------|------
d78590e | False

Key | Value
attendees | 10

Operation | Value
d78590e-1 | 10

Storage
Beldi’s architecture

Start

N = Read(“attendees”)

Write(“attendees”, N+1)

End

Worker

Beldi Runtime

Storage

<table>
<thead>
<tr>
<th>Instance Id</th>
<th>Done</th>
</tr>
</thead>
<tbody>
<tr>
<td>d78590e</td>
<td>False</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>attendees</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>d78590e-1</td>
<td>10</td>
</tr>
</tbody>
</table>
Beldi’s architecture

Worker

Start

N = Read(“attendees”)

Write(“attendees”, N+1)

End

Beldi Runtime

Database API

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>attendees</td>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>d78590e-1</td>
<td>10</td>
</tr>
</tbody>
</table>

Storage
Beldi’s architecture

```
Worker
N = Read("attendees")
Write("attendees", N+1)
End

Beldi Runtime
Database API
①
②

Storage
Key | Value
--- | ---
attendees | 11
Operation | Value
--- | ---
d78590e-1 | 10
d78590e-2
```

Beldi's architecture
Beldi’s architecture

Worker

Start

N = Read(“attendees”)

End

Write(“attendees”, N+1)

Beldi Runtime

Database API

①

Key | Value
---|---
attendees | 11

②

Operation | Value
---|---
d78590e-1 | 10
d78590e-2 |
Beldi’s architecture

Problem: ① and ② must be done atomically
Solution: Collocate write log with the data!
Worker

\[ N = \text{Read("attendees")} \]

\[ \text{Write("attendees", } N+1) \]

End

Beldi’s architecture

### Database API

- **Key**: attendees
- **Value**: 10

### Progress Lambda

- **Instance Id**: d78590e
- **Done**: False

### Storage

- **Operation**: d78590e-1
- **Value**: 10
Worker

Start

N = Read("attendees")

Write("attendees", N+1)

End

Beldi’s architecture

Beldi Runtime

Progress Lambda

Instance Id | Done
--- | ---
d78590e | False

Database API

Key | Value | Recent Writes
--- | --- | ---
attendees | 11 | [d78590e-2]

Storage

Operation | Value
--- | ---
d78590e-1 | 10
Beldi’s architecture

Worker

Start

N = Read(“attendees”) → Progress Lambda

Write(“attendees”, N+1) → Database API

End

Beldi Runtime

Instance Id | Done
---|---
d78590e | False

Key | Value | Recent Writes
---|---|---
attendees | 11 | [d78590e-2]

Operation | Value
---|---
d78590e-1 | 10

Storage
Technical Challenges

1. Limitation of databases
2. Federated setup
3. Transactions across multiple lambdas
Limitation of databases

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>Recent Writes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[d78590e-1, d78590e-2, …, d78590e-1000]</td>
<td>10</td>
</tr>
</tbody>
</table>

Solution: spread the log for a given key across multiple rows
## Limitation of databases

<table>
<thead>
<tr>
<th>RowId</th>
<th>Key</th>
<th>Value</th>
<th>Recent Writes</th>
<th>NextRow</th>
</tr>
</thead>
<tbody>
<tr>
<td>HEAD</td>
<td>attendees</td>
<td>10</td>
<td>[d78590e-1, d78590e-2, ..., d78590e-1000]</td>
<td>f9cec2e</td>
</tr>
<tr>
<td>f9cec2e</td>
<td>attendees</td>
<td>11</td>
<td>[d78590e-1001]</td>
<td></td>
</tr>
</tbody>
</table>
How do we traverse to the tail?
Linked DAAL
Linked DAAL
Linked DAAL

Primary Key

- HEAD
- RowId
- RowId

<table>
<thead>
<tr>
<th>RowId</th>
<th>Key</th>
<th>Value</th>
<th>Recent Writes</th>
<th>NextRow</th>
</tr>
</thead>
<tbody>
<tr>
<td>HEAD</td>
<td>Key</td>
<td>Value</td>
<td>Recent Writes</td>
<td>NextRow</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Linked DAAL
Linked DAAL

Solution: Use scan and projection to download a skeleton version of Linked DAAL

Primary Key

RowId | Key | Value | Recent Writes | NextRow
---|---|---|---|---
RowId | Key | Value | Recent Writes | NextRow
Linked DAAL

![Diagram of Linked DAAL structure with columns: RowId, Key, Value, Recent Writes, NextRow. Primary Key is indicated.]
Linked DAAL
Linked DAAL

- Primary Key

```
<table>
<thead>
<tr>
<th>RowId</th>
<th>Key</th>
<th>Value</th>
<th>Recent Writes</th>
<th>NextRow</th>
</tr>
</thead>
<tbody>
<tr>
<td>HEAD</td>
<td>Key</td>
<td>Value</td>
<td>Recent Writes</td>
<td>NextRow</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

256 Bits

DAAL
Linked DAAL

Primary Key

HEAD | Key | Value | Recent Writes | NextRow
--- | --- | --- | --- | ---
RowId | Key | Value | Recent Writes | NextRow
RowId | Key | Value | Recent Writes | NextRow
RowId | Key | Value | Recent Writes | NextRow

HEAD | NextRow
--- | ---
RowId | NextRow
RowId | NextRow
RowId | NextRow

256 Bits
Outline

• Beldi’s Infrastructure
• Linked DAAL
• **Invocation with exactly-once semantics**
• Evaluation
• Conclusion
Invocation with exactly-once semantics

Lambda 1

Call Lambda2

Lambda 2
Invocation with exactly-once semantics

Lambda 1

- Call Lambda2

<table>
<thead>
<tr>
<th>Operation</th>
<th>Callee</th>
</tr>
</thead>
<tbody>
<tr>
<td>d78590e-1</td>
<td>b97bbe0</td>
</tr>
</tbody>
</table>

Lambda 2
Invocation with exactly-once semantics

Lambda 1

Call Lambda2

Log in Progress Table

Lambda 2

<table>
<thead>
<tr>
<th>Operation</th>
<th>Callee</th>
</tr>
</thead>
<tbody>
<tr>
<td>d78590e-1</td>
<td>b97bbae0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instance Id</th>
<th>Done</th>
</tr>
</thead>
<tbody>
<tr>
<td>b97bbae0</td>
<td>False</td>
</tr>
</tbody>
</table>
Invocation with exactly-once semantics

Lambda 1

Call Lambda2

Operation: d78590e-1
Callee: b97bbe0

Lambda 2

Log in Progress Table

Instance Id: b97bbe0
Done: False

make some writes
Invocation with exactly-once semantics

Lambda 1

- Call Lambda2

Lambda 2

- Log in Progress Table
  - make some writes
  - Mark as Done

<table>
<thead>
<tr>
<th>Operation</th>
<th>Callee</th>
</tr>
</thead>
<tbody>
<tr>
<td>d78590e-1</td>
<td>b97bbe0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instance Id</th>
<th>Done</th>
</tr>
</thead>
<tbody>
<tr>
<td>b97bbe0</td>
<td>True</td>
</tr>
</tbody>
</table>
Invocation with exactly-once semantics

Lambda 1

- Call Lambda2

<table>
<thead>
<tr>
<th>Operation</th>
<th>Callee</th>
</tr>
</thead>
<tbody>
<tr>
<td>d78590e-1</td>
<td>b97bbe0</td>
</tr>
</tbody>
</table>

Lambda 2

- Log in Progress Table
- make some writes
- Mark as Done

<table>
<thead>
<tr>
<th>Instance Id</th>
<th>Done</th>
</tr>
</thead>
<tbody>
<tr>
<td>b97bbe0</td>
<td>True</td>
</tr>
</tbody>
</table>

Instance Id: b97bbe0
Done: True
Invocation with exactly-once semantics

Lambda 1

- Call Lambda2

<table>
<thead>
<tr>
<th>Operation</th>
<th>Callee</th>
</tr>
</thead>
<tbody>
<tr>
<td>d78590e-1</td>
<td>b97bbe0</td>
</tr>
</tbody>
</table>

Lambda 2

- Log in Progress Table
- make some writes
- Mark as Done

<table>
<thead>
<tr>
<th>Instance Id</th>
<th>Done</th>
</tr>
</thead>
<tbody>
<tr>
<td>b97bbe0</td>
<td>True</td>
</tr>
</tbody>
</table>
Invocation with exactly-once semantics

Lambda 1

Operation | Callee
---|---
d78590e-1 | b97bbe0

Call Lambda2

Lambda 2

Instance Id | Done
---|---
b97bbe0 | True
Invocation with exactly-once semantics

Lambda 1

Call Lambda2

Operation | Callee
---|---
d78590e-1 | b97bbe0

Lambda 2

Log in Progress Table

make some writes

Mark as Done

Instance Id | Done
---|---
b97bbe0 | True
Invocation with exactly-once semantics

Lambda 1

- Call Lambda2
- Receive Response

<table>
<thead>
<tr>
<th>Operation</th>
<th>Callee</th>
</tr>
</thead>
<tbody>
<tr>
<td>d78590e-1</td>
<td>b97bbe0</td>
</tr>
</tbody>
</table>

Lambda 2

- Log in Progress Table
- make some writes
- Mark as Done

<table>
<thead>
<tr>
<th>Instance Id</th>
<th>Done</th>
</tr>
</thead>
<tbody>
<tr>
<td>b97bbe0</td>
<td>True</td>
</tr>
</tbody>
</table>
Invocation with exactly-once semantics

Lambda 1

<table>
<thead>
<tr>
<th>Operation</th>
<th>Callee</th>
</tr>
</thead>
<tbody>
<tr>
<td>d78590e-1</td>
<td>b97bbe0</td>
</tr>
</tbody>
</table>

Call Lambda2

Lambda 2

<table>
<thead>
<tr>
<th>Instance Id</th>
<th>Done</th>
</tr>
</thead>
<tbody>
<tr>
<td>b97bbe0</td>
<td>True</td>
</tr>
</tbody>
</table>

GC
Invocation with exactly-once semantics
Invocation with exactly-once semantics

Lambda 1

Call Lambda2

Operation | Callee
----------|-------
d78590e-1 | b97bbe0

Lambda 2

Log in Progress Table

Instance Id | Done
-------------|-----
b97bbe0      | False

make some writes
Invocation with exactly-once semantics

Lambda 1

Call Lambda2

<table>
<thead>
<tr>
<th>Operation</th>
<th>Callee</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>d78590e-1</td>
<td>b97bbe0</td>
<td></td>
</tr>
</tbody>
</table>

Lambda 2
Invocation with exactly-once semantics

<table>
<thead>
<tr>
<th>Operation</th>
<th>Callee</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>d78590e-1</td>
<td>b97bbe0</td>
<td></td>
</tr>
</tbody>
</table>

Lambda 1

- Call Lambda2

Lambda 2

- Log in Intent Table
- make some writes
Invocation with exactly-once semantics

<table>
<thead>
<tr>
<th>Operation</th>
<th>Callee</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>d78590e-1</td>
<td>b97bbe0</td>
<td>result</td>
</tr>
</tbody>
</table>

Lambda 1:
- Call Lambda2
- Log in Intent Table
  - make some writes
  - Callback

Lambda 2:
- Operation
- Callee
- Result
Invocation with exactly-once semantics

<table>
<thead>
<tr>
<th>Operation</th>
<th>Callee</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>d78590e-1</td>
<td>b97bbe0</td>
<td>result</td>
</tr>
</tbody>
</table>

Lambda 1
- Call Lambda2

Lambda 2
- Log in Intent Table
- make some writes
- Callback
- Mark as Done

Operation: d78590e-1
Callee: b97bbe0
Result: result
Outline

• Beldi’s Infrastructure
• Linked DAAL
• Invocation with exactly-once semantics
  • Evaluation
• Conclusion
Evaluation

1. What are the costs of Beldi’s API operations?

2. How does Beldi perform in real-world applications?

3. What is the effect of garbage collection?
What are the costs of Beldi’s API operations?

20 rows in Linked DAAL, 2 - 4x more expensive than baseline
How does Beldi perform in real-world applications?

DeathStarBench (*ASPLOS 19*): open-source microservices benchmark
- Movie review service (Cf. IMDB)
- Travel reservation (Cf. Expedia)
- Social media site (Cf. Twitter)
How does Beldi perform in real-world applications?

- <400 req/s: 2× higher than baseline
- 700 req/s (saturation): 3.3 × higher than baseline
Outline

• Beldi’s Infrastructure
• Linked DAAL
• Invocation with exactly-once semantics
• Evaluation
• Conclusion
Conclusion

1. A framework to write transactional and fault-tolerant applications on serverless.
2. A lock-free data structure (Linked DAAL) to support fast logging and exactly-once semantics
3. A collaborative distributed transaction protocol across multiple lambdas
4. An efficient garbage collection algorithm that runs independently without affecting running lambdas or requiring any pauses.

https://github.com/eniac/beldi
Thank you!