Doing More with Less: Orchestrating Serverless Applications without an Orchestrator

David H. Liu  Amit Levy  Shadi Noghabi  Sebastian Burckhardt
The Serverless Paradigm

• Functions + Stateful Services
• Function: stateless, event-driven and short-lived
• Stateful services, e.g., object stores, databases, messaging services
The Serverless Paradigm

- Freedom from server management tasks
- Fast and fine-grained auto-scaling
- Fine-grained and pay-for-what-you-use billing
- Improved multiplexing and resource utilization
Question: How to build large applications consisting of many functions

• Stateful patterns over stateless functions
  o E.g., aggregating the results of multiple functions
• At-least-once function execution
  o Multiple diverging outputs
  o Wasted runtime costs
Orchestrators

- Additional services to existing infrastructures
- Higher-level programming interfaces
- User-deployed or provider-hosted

![Diagram showing orchestration flow with AWS Step Functions, Orchestrator, and FaaS Functions]
Orchestrators

• Logically centralized controller
  o Invoke functions and receive function results
  o Centralization simplifies stateful patterns and execution guarantees
Orchestrators

• Compromise original benefits of serverless
  o E.g., freedom from server management, fine-grained billing

• Hosting costs of resources and engineering

• Limit application performance and programmability
  o Application-specific functionalities and optimizations
Application-Level Orchestration

Can we support large serverless applications on the application-level without adding services to the infrastructure?
Application-Level Orchestration

Can we support large serverless applications on the application-level without adding services to the infrastructure?

• How far can we take the serverless model as is?
Application-Level Orchestration

*Can we support large serverless applications on the application-level without adding services to the infrastructure?*

• Original serverless benefits
• Automatically improve with underlying systems
• Providers: freedom from hosting
• Users:
  o Ability to implement application-specific functionalities and optimization
  o Easy migration and alleviate vendor lock-in
Unum: A Library Orchestrator

• No new services or changes to existing APIs
• Support programs from AWS Step Functions
• Exactly-once semantics with checkpoints
• Coordinate stateful patterns via data stores
• Better performance & costs
• Migration with no app code changes
Unum: A Library Orchestrator

High-Level App Definition \rightarrow \text{Unum Compiler} \rightarrow \text{Unum IR}

\text{Unum functions at runtime}

\text{User function} \quad \text{Unum IR} \quad \text{Unum Ingress and Egress} \quad \text{Serverless function}
Unum: A Library Orchestrator

• Where to execute orchestration logic
  o Unum Intermediate Representation: Decentralized Orchestration

• How to support stateful patterns such as aggregation
  o Coordination via Data Store

• How to ensure strong execution guarantees
  o Checkpoints via Data Store
Unum IR: Decentralize Orchestration

*Preserve higher-level programming interfaces without an orchestrator*

- Single “orchestrator” function is inefficient and limits application sizes
  - Double billing
  - Time-out limits application sizes
Unum IR: Decentralize Orchestration

*Preserve higher-level programming interfaces without an orchestrator*

- **Decentralize orchestration logic**
  - Local portion of orchestration
  - Decentralize at compile time
Unum IR: Decentralize Orchestration

• Represent high-level application definitions as directed graphs
  o Vertices are functions
  o Edges are transitions
Unum IR: Decentralize Orchestration

• Assign edges to tail nodes
  o Invoking the head node
  o Passing outputs to the head node as input

```
"States": {
  "A": {
    "Type": "Task",
    "Next": "B"
  },
  "B": {
    "Type": "Task",
    "End": true
  }
}
```

Step Functions
Unum IR: Decentralize Orchestration

• Assign edges to tail nodes
• Unum IR encodes edges as sequences of instructions
Unum IR: Decentralize Orchestration

- Assign edges to tail nodes
- Unum IR encodes edges as sequences of instructions
Unum IR: Decentralize Orchestration

- Assign edges to tail nodes
- Unum IR encodes edges as sequences of instructions
- Unum runtime executes IR instructions in-situ with user code

```
A --> B
Name: A
Next: Invoke: B

A --> B --> C
Name: A
Next: Invoke: B
- Invoke: C

Unum Runtime
```
Unum IR: Decentralize Orchestration

*The IR can be as simple or complex as the application requires*

The Unum standard library supports all patterns in Step Functions

![Diagram of Unum IR]
Coordination & Aggregation
Coordination & Aggregation

• Tail nodes write results into the data store after completion
• Each tail node checks the completion of other tail nodes by reading from the data store
Coordination & Aggregation

- Any one of the tail functions can invoke the head node
- Functions stay short-lived

Name: B
Next:
  FanIn([B, C], D)

Name: C
Next:
  FanIn([B, C], D)
Coordination & Aggregation

- Strongly consistent data stores to avoid missing invocations
- Each tail node performs
  - 1 Write
  - N - 1 Reads
Coordination & Aggregation

• Coordination Set
  o Tail nodes additionally add their names to a set data structure
  o One coordination set per head node
  o Optimize data read volume
    • 1 read vs N-1 reads

Name: C
Next: FanIn(D-Set, D)
{B, C}
Exactly-Once Execution Semantics

- Unum runtime checkpoints exactly one execution of a node
  - Checkpoints contain user code results
  - A checkpoint uniquely identifies a node
  - Concurrent-but-slower executions discard results
Exactly-Once Execution Semantics

- Only checkpointed data is propagated downstream
Exactly-Once Execution Semantics

• Only checkpointed data is propagated downstream
Exactly-Once Execution Semantics

• Only checkpointed data is propagated downstream
• Bypass user code if a checkpoint already exists

Unum Runtime
Implementation

• A standard library that includes
  o Runtime for AWS using Lambda and DynamoDB
  o Runtime for Google Cloud using Cloud Functions and Firebase

• Toolchain
  o Compiler for Step Functions to Unum IR
  o Linker that packages the user functions with the target platform’s runtime library
  o CLI tool for deploying applications

$ unum-cli compile
$ unum-cli build
$ unum-cli deploy
Evaluation

• Compare against Step Functions on AWS
• All applications are written as Step Functions
• All lambdas are similarly configured
• Unum applications are compiled from Step Functions definition
• Unum experiments use DynamoDB with on-demand provisioning
Evaluation – Single Transition in a Chain

• Latency of a single transition
  ○ From the end of the first function's user code to just before user code starts running in the second function

• Unum:
  ○ 1 storage write to create checkpoint
  ○ 1 lambda invoke
  ○ 1 storage read to check checkpoint existence

• Step Functions:
  ○ Timestamp between 1st function return and 2nd function start
Evaluation – Single Transition in a Chain

- Unum is consistently faster
- Storage writes, reads and Lambda invoke API calls make up the majority of the latency overhead

Lower is better
Evaluation – Parallel Performance

• End-to-end latency of fan-out and fan-in
• Varying the number of parallel branches
Evaluation – Parallel Performance

- Unum incurs a modest overhead (up to 200ms) at low branching degree
- Unum is up to 4x faster at higher branching degrees
## Evaluation – Applications

<table>
<thead>
<tr>
<th>App</th>
<th>Pattern(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IoT Pipeline</td>
<td>Chain</td>
<td>Process thermostat data and decide HVAC adjustment</td>
</tr>
<tr>
<td>Text Processing</td>
<td>Chain, Fan-out, Fan-in</td>
<td>Preprocess a social media post (from DeathStarBench)</td>
</tr>
<tr>
<td>Wordcount</td>
<td>Fan-out, Fan-in</td>
<td>MapReduce word count</td>
</tr>
<tr>
<td>ExCamera</td>
<td>Chain, Fan-out, Fan-in, Fold</td>
<td>Parallel video encoding</td>
</tr>
</tbody>
</table>
### Evaluation – Applications

<table>
<thead>
<tr>
<th>App</th>
<th>Latency (seconds)</th>
<th>Costs ($ per 1 mil. executions)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unum-aws</td>
<td>Unum-gcloud</td>
</tr>
<tr>
<td>IoT Pipeline</td>
<td>0.12</td>
<td>0.81</td>
</tr>
<tr>
<td>Text Processing</td>
<td>0.52</td>
<td>3.56</td>
</tr>
<tr>
<td>Wordcount</td>
<td>408.88</td>
<td>484.12</td>
</tr>
<tr>
<td>ExCamera</td>
<td>84.52</td>
<td>122.63</td>
</tr>
</tbody>
</table>

- Unum is consistently faster and cheaper
- Up to 2x faster and 9x cheaper.
Evaluation – Applications

- Step Functions makes up the majority of the costs
- Unum: DynamoDB reads and writes make up the majority of the costs
Summary

• Application-level orchestration is both possible and practical
• Higher-level programming interfaces
• Strong execution guarantees
• More flexible, cheaper and faster
Thank You!

Questions?

hao.liu@princeton.edu

Code:

- Unum library: https://github.com/LedgeDash/unum
- Unum toolchain: https://github.com/LedgeDash/unum/unum-cli
- Unum app repo: https://github.com/LedgeDash/unum-appstore