Comparison of FaaS Orchestration Systems

Pedro García López, Marc Sánchez-Artigas, Gerard París, Daniel Barcelona Pons, Álvaro Ruiz Ollobarren, and David Arroyo Pinto

Cloud and Distributed Systems Lab
CloudButton: Serverless Data Analytics

- 4.4M€ Research project
- cloudbutton.eu
- Coordinated by URV
- 2019-2021
Creating Serverless Workflows

- Azure Durable Functions
- AWS Step Functions
- IBM Function Composer
The Serverless Trilemma

- If the serverless runtime is limited to a **reactive core**, i.e. one that deals only with dispatching functions in response to events, then these constraints form the serverless trilemma.

- IBM Sequences are **ST-Safe**

  1. Functions as Black Boxes
  2. Substitution principle
  3. Double billing
Evaluation framework

- ST-Safeness
- Programming model
- Parallel execution support
- State management
- Software packaging and repositories
- Architecture
- Billing model
- Overhead
Amazon Step Functions

- **ST-Safeness**
- **Programming model**
- **Parallel execution support**
- **State management**
- **Software packaging and repositories**
- **Architecture**
- **Billing model**

(2) composability

Amazon States Language DSL

- 32K

- client scheduler

0.025USD per state transition
StateMachine.Builder stateMachineBuilder =
  stateMachine()
  .comment("A state machine with parallel states")
  .startAt("Parallel");

Branch.Builder[] branchBuilders =
  new Branch.Builder[NSTEPS];

for (int i = 0; i < NSTEPS; i++) {
  branchBuilders[i] = branch()
    .startAt(String.valueOf(i + 1))
    .state(String.valueOf(i + 1))
    .taskState()
    .resource(arnTask).transition(end());
}

stateMachineBuilder.
state("Parallel",
  parallelState().branches(branchBuilders)
  .transition(end()));

final StateMachine stateMachine =
  stateMachineBuilder.build();
IBM Composer and Sequences

- ST-Safeness
- Programming model
- Parallel execution support
- State management
- Software packaging and repositories
- Architecture
- Billing model

> JavaScript Composer library

5MB

reactive core, conductor actions

unknown, free?
IBM Composer and Sequences

```javascript
composer.sequence(
    'currentTemperature', // programmatic composition
    composer.if(          // call cloud function or API
        result => result.temp < 60,  // conditional control flow
        'turnOnHeat')                 // mix inline JavaScript
    ) // interface to 3rd party services
```
Azure Durable Functions

- ST-Safeness
- Programming model: C# async/await, Task Framework
- Parallel execution support
- State management: Unlimited, compressed
- Software packaging and repositories
- Architecture: reactive core, event sourcing
- Billing model: unknown, unexpected storage costs
public static async Task Run(DurableOrchestrationContext ctx)
{
    var parallelTasks = new List<Task<int>>();

    // get a list of N work items to process in parallel
    object[] workBatch = await ctx.CallActivityAsync<object[]>("F1");
    for (int i = 0; i < workBatch.Length; i++)
    {
        Task<int> task = ctx.CallActivityAsync<int>("F2", workBatch[i]);
        parallelTasks.Add(task);
    }

    await Task.WhenAll(parallelTasks);

    // aggregate all N outputs and send result to F3
    int sum = parallelTasks.Sum(t => t.Result);
    await ctx.CallActivityAsync("F3", sum);
}
Experiment 1: Sequences

```java
StateMachine.Builder stateMachineBuilder =
    stateMachine()
    .comment("A_sequence_state_machine")
    .startAt("1");
for (int i = 1; i <= NSTEPS; i++) {
    stateMachineBuilder.state(String.valueOf(i),
                            taskState().resource(arnTask)
                            .transition((i != NSTEPS) ?
                                next(String.valueOf(i + 1)) : end()));
}
StateMachine stateMachine =
    stateMachineBuilder.build();

composer.repeat(40, 'sleepAction')

for (int i = 0; i < NSTEPS; i++) {
    await context;
    CallActivityAsync("sleepAction", null);
}
```
Experiment 2: Parallels

```java
StateMachine.Builder stateMachineBuilder =
    stateMachine()
    .comment("A state machine with parallel states.")
    .startAt("Parallel");

Branch.Builder[] branchBuilders =
    new Branch.Builder[NSTEPS];

for (int i = 0; i < NSTEPS; i++) {
    branchBuilders[i] = branch()
        .startAt(String.valueOf(i + 1))
        .state(String.valueOf(i + 1),
            taskState()
                .resource(arnTask),
            transition(end()));
}

stateMachineBuilder.state("Parallel",
    parallelState().branches(branchBuilders)
        .transition(end()));
final StateMachine stateMachine =
    stateMachineBuilder.build();

var tasks = new Task<long>[NSTEPS];
for (int i = 0; i < NSTEPS; i++)
{
    tasks[i] = context.CallActivityAsync<long>(
        "sleepAction");
}
await Task.WhenAll(tasks);
```
Experiments

![Graph 1: Overhead vs Sequence length (n)]

- IBM Sequences
- IBM Composer
- AWS Step Functions
- Azure DF
- Azure DF (ext. sessions)

![Graph 2: Overhead vs Number of concurrent functions (n)]

- AWS Step Functions
- Azure DF (ext. sessions)
Experiments

Parallel state overhead by number of states

<table>
<thead>
<tr>
<th>Platform</th>
<th>Overhead (ms)</th>
<th>Increase (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Without payload</td>
<td>With payload</td>
</tr>
<tr>
<td><strong>IBM Sequences</strong></td>
<td>49.0</td>
<td>80.8</td>
</tr>
<tr>
<td><strong>IBM Composer</strong></td>
<td>175.7</td>
<td>298.4</td>
</tr>
<tr>
<td><strong>AWS Step Functions</strong></td>
<td>168.0</td>
<td>287.0</td>
</tr>
<tr>
<td><strong>Azure DF</strong></td>
<td>766.2</td>
<td>859.5</td>
</tr>
</tbody>
</table>
Suspend API

- Suspend function until event is received
- Passivation and state should be handled by the Function
- Requires a pure **reactive core** enabling **custom events**
- It would enable the creation of custom orchestrators
## Discussion

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Amazon Step Functions</th>
<th>IBM Composer</th>
<th>Azure Durable Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST-safe [1]</td>
<td>No (compositions are not functions)</td>
<td>Yes (composition as functions)</td>
<td>Yes (composition as functions)</td>
</tr>
<tr>
<td><strong>Programming model</strong></td>
<td>DSL (JSON)</td>
<td>Composition library (Javascript)</td>
<td>async/await (C#)</td>
</tr>
<tr>
<td>Reflective API</td>
<td>Yes (limited)</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Parallel execution support</td>
<td>Yes (limited)</td>
<td>No</td>
<td>Yes (limited)</td>
</tr>
<tr>
<td>Software packaging and repositories</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes (no repo)</td>
</tr>
<tr>
<td><strong>Billing model</strong></td>
<td>$0.025 per 1,000 state transitions</td>
<td>Orchestrator function execution</td>
<td>Orchestrator function execution + storage costs</td>
</tr>
<tr>
<td><strong>Architecture</strong></td>
<td>Synchronous client scheduler</td>
<td>Reactive scheduler</td>
<td>Reactive scheduler</td>
</tr>
</tbody>
</table>
Conclusions

- Amazon Step Functions is the most mature project
- Microsoft ADF is the more advanced in programmability, IBM Composer wins in simplicity
- None of them support parallel tasks efficiently
- Orchestration must have a cost if it is fault-tolerant
- Reactive core, custom events and suspend API
- Early immature projects with high potential for the future