Serverless workflows for orchestration hybrid cluster-based and serverless processing

Rustem Feyzkhanov
Instrumental
Cloud native orchestrators are convenient for uniting serverless and cluster worlds

They provide a way to shift from static cluster to on-demand cluster aka ‘serverless’ cluster

They can be used for a number of applications and can be deployed easily together
## Function as a service (FaaS)

<table>
<thead>
<tr>
<th>On premise</th>
<th>IaaS</th>
<th>PaaS</th>
<th>FaaS</th>
<th>SaaS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functions</td>
<td>Functions</td>
<td>Functions</td>
<td>Functions</td>
<td>Functions</td>
</tr>
<tr>
<td>Application</td>
<td>Application</td>
<td>Application</td>
<td>Application</td>
<td>Application</td>
</tr>
<tr>
<td>Runtime</td>
<td>Runtime</td>
<td>Runtime</td>
<td>Runtime</td>
<td>Runtime</td>
</tr>
<tr>
<td>Operating system</td>
<td>Operating system</td>
<td>Operating system</td>
<td>Operating system</td>
<td>Operating system</td>
</tr>
<tr>
<td>Virtualization</td>
<td>Virtualization</td>
<td>Virtualization</td>
<td>Virtualization</td>
<td>Virtualization</td>
</tr>
<tr>
<td>Networking</td>
<td>Networking</td>
<td>Networking</td>
<td>Networking</td>
<td>Networking</td>
</tr>
<tr>
<td>Storage</td>
<td>Storage</td>
<td>Storage</td>
<td>Storage</td>
<td>Storage</td>
</tr>
<tr>
<td>Hardware</td>
<td>Hardware</td>
<td>Hardware</td>
<td>Hardware</td>
<td>Hardware</td>
</tr>
</tbody>
</table>

On premise, IaaS, PaaS, FaaS, and SaaS.
How typical FaaS works

- **Container pool**
- **Lambda configuration**
  - Code
  - Libraries
  - Configuration (memory, max time)

**Warm container**

**Trigger**
- S3
- API
- DynamoDB
- SQS, Kinesis

**Response**
<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
<th>Limits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Easy to deploy (no docker)</td>
<td>No local debug</td>
<td>Max execution time</td>
</tr>
<tr>
<td>Easy to connect to cloud native services</td>
<td>Unpredictable warm containers</td>
<td>Max available RAM</td>
</tr>
<tr>
<td>Easy to scale</td>
<td>Logging may not be exhaustive</td>
<td>Hard disk</td>
</tr>
<tr>
<td>Relatively cheap</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Microservice architecture

A set of services which are:

- Loosely coupled
- Independently deployable
- Each one of them implements a business capability
- Highly maintainable and testable
# Microservice connectors

<table>
<thead>
<tr>
<th>Rest API</th>
<th>Event query</th>
<th>Orchestrator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous process</td>
<td>Asynchronous process</td>
<td>Asynchronous process</td>
</tr>
<tr>
<td>Short-term process</td>
<td>Long-term process</td>
<td>Long-term process</td>
</tr>
<tr>
<td>Simple intermediate logic</td>
<td>Simple intermediate logic</td>
<td>Complex intermediate logic</td>
</tr>
<tr>
<td>Doesn’t trace the whole process</td>
<td>Doesn’t trace the whole process</td>
<td>Traces the process</td>
</tr>
<tr>
<td>Cheap</td>
<td>Cheap</td>
<td>Expensive</td>
</tr>
</tbody>
</table>

#StrataData
Cloud native orchestrators

- Native support for FaaS
- Central monitoring
- Central logging and tracing
- On-demand scaling
Types of cloud native orchestrators

- AWS Step Functions
  - Integration with other AWS services
  - Static parallelization
- Azure logic apps
  - Integration with other services’ APIs
  - Dynamic parallelization
Orchestrators for hybrid architecture

- Graph-based description
- Processing nodes: FaaS or Clusters
  - Task state and waiting for the node
  - Invocation of processing node
- Logic for error handling
- Parallel execution
- Branching and loops
- Scheduler
How task state works

- Cluster makes constant requests to Step Functions service
- Receives input json with token and runs task
- Returns output with the token of specific Step function
Lambda makes constant requests to Step Functions service

- Receives input JSON with token and sends task to cluster

- Cluster returns output with the token of specific Step function
Example patterns

- Wrapper for high tier services
- Timeout for the task
- Error handling
- Retry logic
- Falling back to alternative service
Example patterns

- Tasks which require parallel execution
  - Map node for the parallel tasks
  - Error handling at each parallel branch
  - Retry logic for each branch
Example patterns

- Canary deployment
  - Redirects traffic to different nodes based on custom logic
- Gathers stats based on execution
- Can fall back to another service
‘Serverless’ cluster

- Container-as-a-Service
- On-demand cluster which scales with your consumption
- Services:
  - AWS Batch
  - AWS Fargate
## Serverless cluster comparison

<table>
<thead>
<tr>
<th></th>
<th>Lambda</th>
<th>Fargate</th>
<th>Batch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Faas</td>
<td>FaaS</td>
<td>Pure container as a service</td>
<td>Service which starts cluster and executes jobs on it</td>
</tr>
<tr>
<td>Short term processes</td>
<td>Fast startup time (~100ms)</td>
<td>Customizable instances</td>
<td>Spot instances available</td>
</tr>
<tr>
<td>Fast startup time (~10-20s)</td>
<td>Price per 100ms</td>
<td>Only CPU instances</td>
<td>Slow startup time (~1-4min)</td>
</tr>
<tr>
<td>Price per 100ms</td>
<td>Price per 1s (min 1 min)</td>
<td>Medium startup time (~10-20s)</td>
<td>Price per 1s (min 1 min)</td>
</tr>
</tbody>
</table>
Price comparison

C5 Large Instance - 2 vCPU 4GB RAM

- AWS Lambda
  - 3GB RAM \times 0.00001667 \times 3600 = 0.18\$ per hour
- AWS Fargate
  - 4GB RAM \times 0.0044 + 2\ vCPU \times 0.0404 = 0.098\$ per hour
- AWS Batch
  - C5 Large On Demand = 0.085\$ per hour
  - C5 Large Spot = 0.033\$ per hour
Examples - ML training

- Parallel training on multiple sets of hyper parameters
- Central gathering of the results
- Handling error on each branch
- Capability for feedback loop
Examples - Data pipeline

- Modular approach
- Parallel data download and parsing
- FaaS for parallel processing
- Cluster for heavy processing

Data download

Heavy CPU processing

Scalable processing

Handler
Examples - ML pipeline

- A/B testing to test performance of multiple models - either in parallel or separately
- Scalable inference which allows to run batches in parallel
- Allows modular approach (multiple frameworks)
Examples - Load testing

- Multiple parallel lambdas => low cost short-term heavy load
- Handles parallel start of multiple AWS Lambdas
- Can be scheduled (in a static or dynamic way)
Github repo + demo

- [https://github.com/ryfeus/stepfunctions2processing](https://github.com/ryfeus/stepfunctions2processing)
- Configuration for serverless framework deploys:
  - AWS Step functions
  - AWS Lambdas
  - AWS Fargate
  - AWS Batch