Comparison of FaaS Orchestration Systems

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

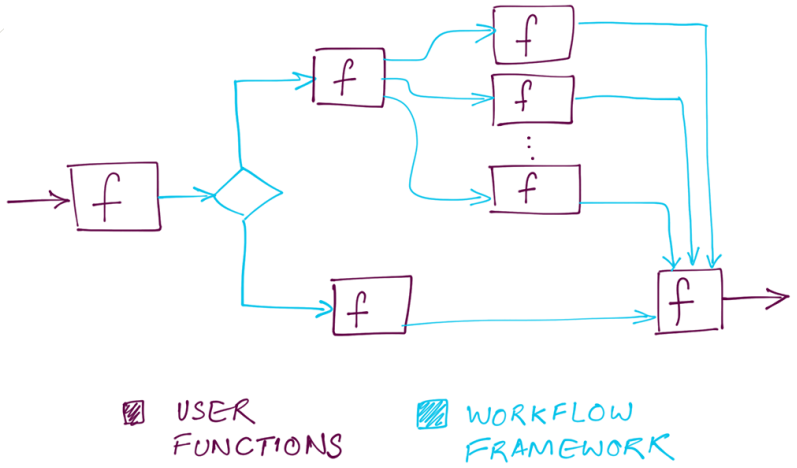
Cloud and Distributed Systems Lab

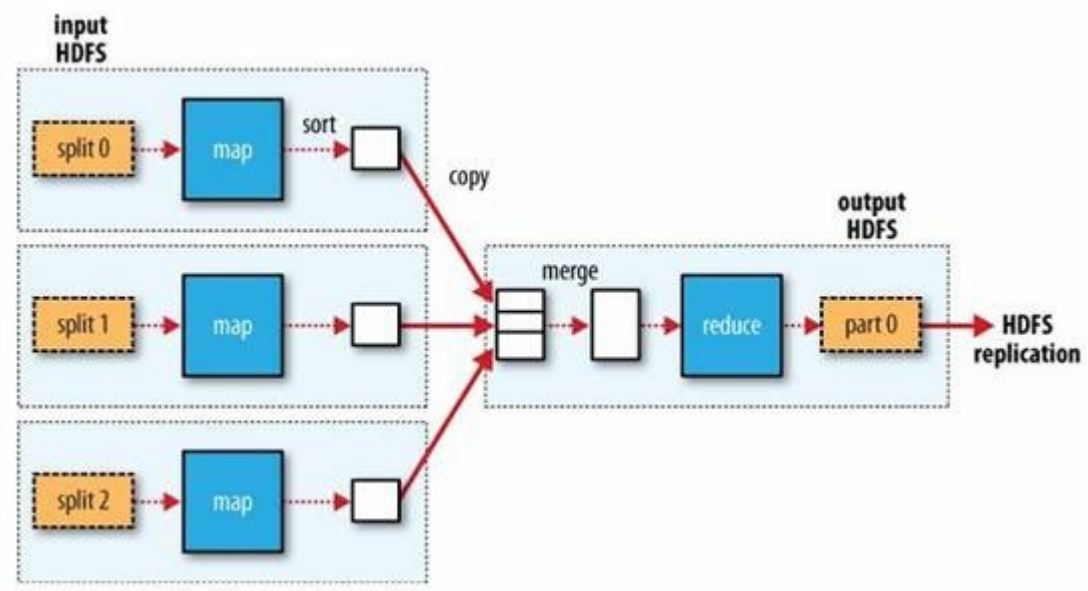
UNIVERSITAT ROVIRA i VIRGILI

CloudButton: Serverless Data Analytics

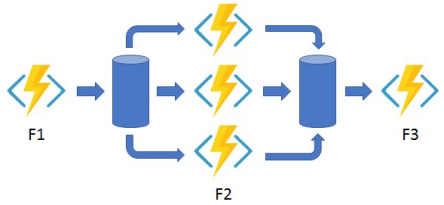
- 4.4M€ Research project
- cloudbutton.eu
- Coordinated by URV
- 2019-2021

Creating Serverless Workflows





AWS Step Functions

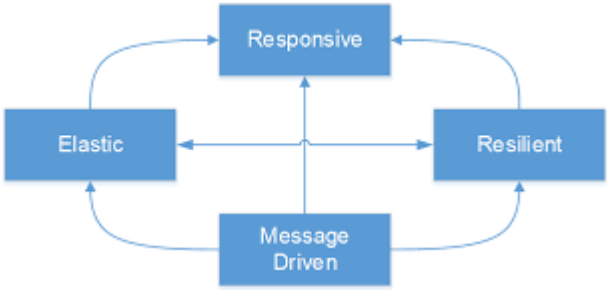


Azure Durable Functions

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**

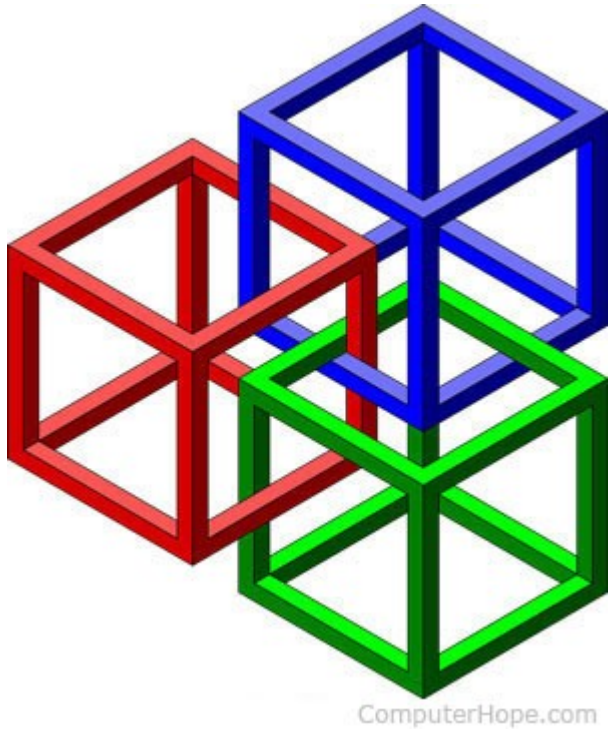
(1) Functions as Black Boxes

(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*

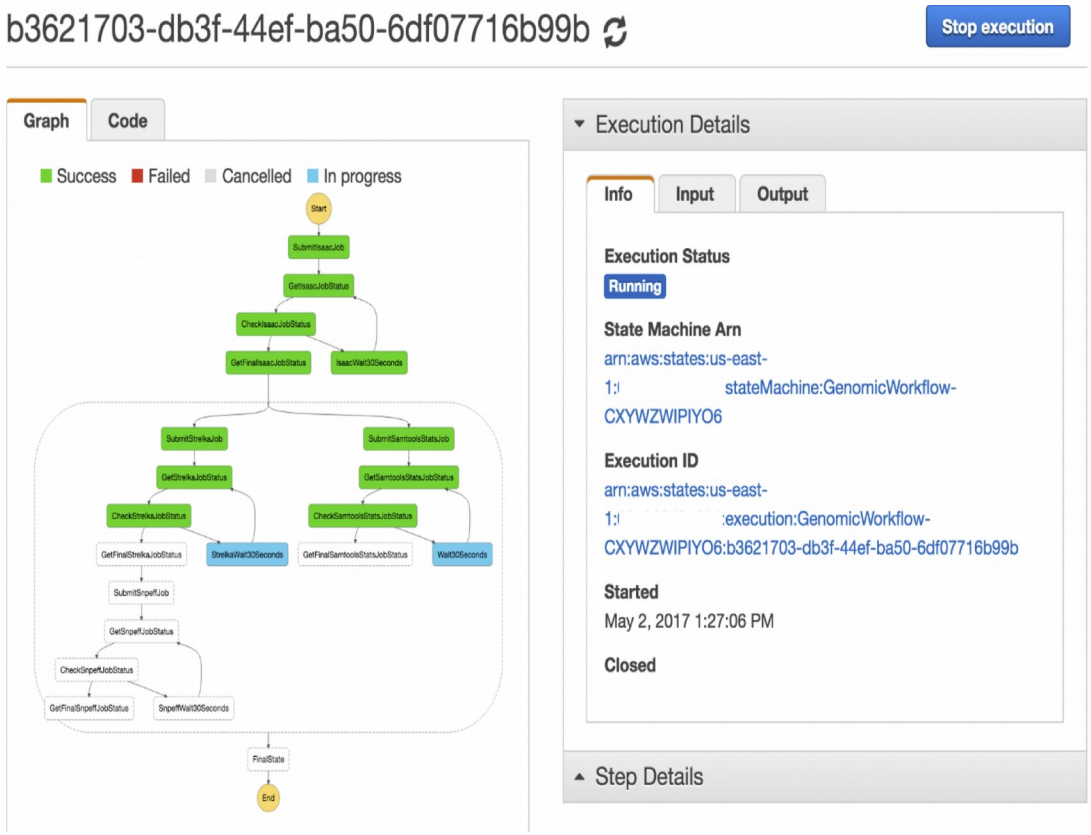


Amazon Step Functions

- *ST-Safeness* ✗ (2) composability
- *Programming model* Amazon States Language DSL
- *Parallel execution support* ✓
- *State management* 32K
- *Software packaging and repositories* ✓
- *Architecture* client scheduler
- *Billing model* 0.025USD per state transition

Amazon Step Functions


```
StateMachine.Builder stateMachineBuilder =
  stateMachine()
  .comment("A state machine with par. 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();
```

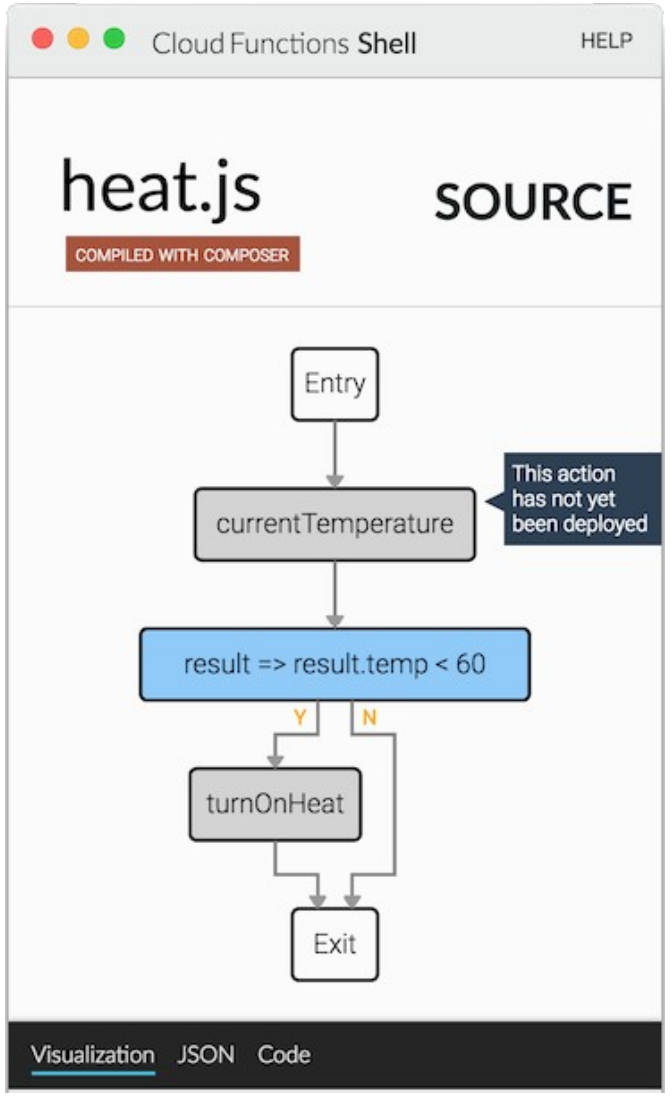


IBM Composer and Sequences

- *ST-Safeness* ✓
- *Programming model* — JavaScript Composer library
- *Parallel execution support* ✗
- *State management* — 5MB
- *Software packaging and repositories* ✓
- *Architecture* — reactive core, conductor actions
- *Billing model* — unknown, free?

IBM Composer and Sequences

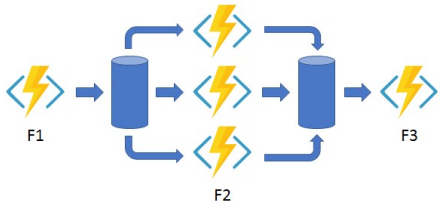
```
composer.sequence(                    // programmatic composition
  'currentTemperature',               // call cloud function or API
  composer.if(                        // conditional control flow
    result => result.temp < 60,       // mix inline JavaScript
    'turnOnHeat')                     // 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

Azure Durable Functions




```
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);
}
```

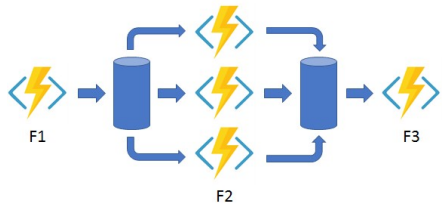
Experiment 1: Sequences

```
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();
```

AWS Step Functions

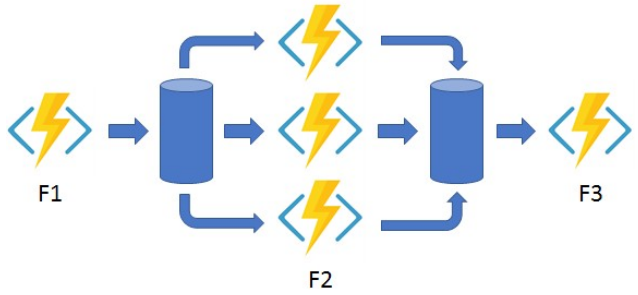
```
composer.repeat(40, 'sleepAction')
```

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



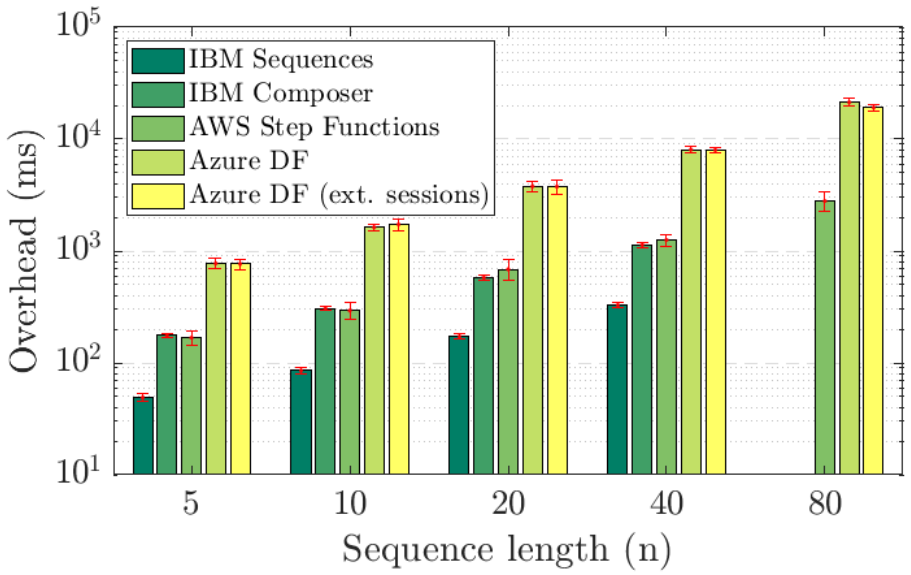
Experiment 2: Parallels

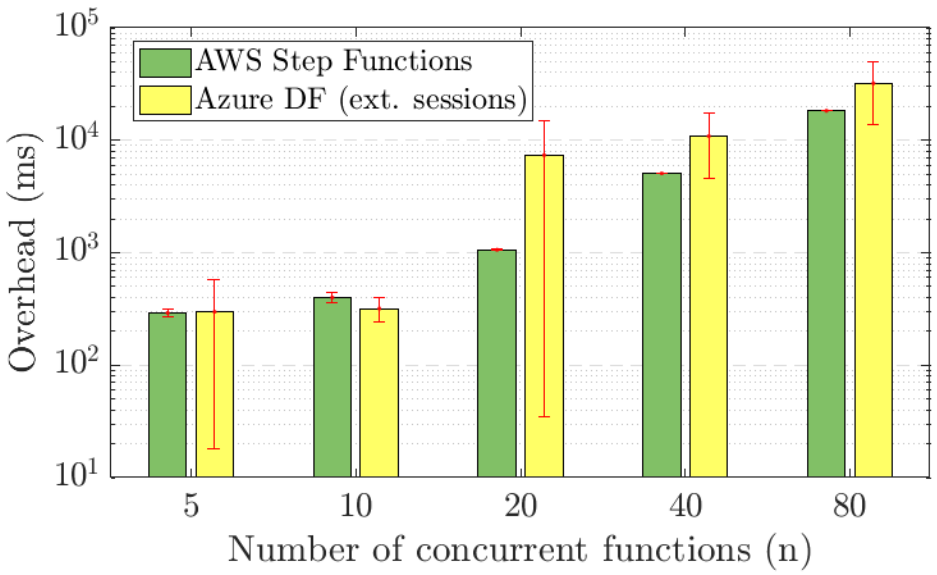
```
StateMachine.Builder stateMachineBuilder =
  stateMachine()
  .comment("A state machine with par. states.")
  .startAt("Parallel");

  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();
```



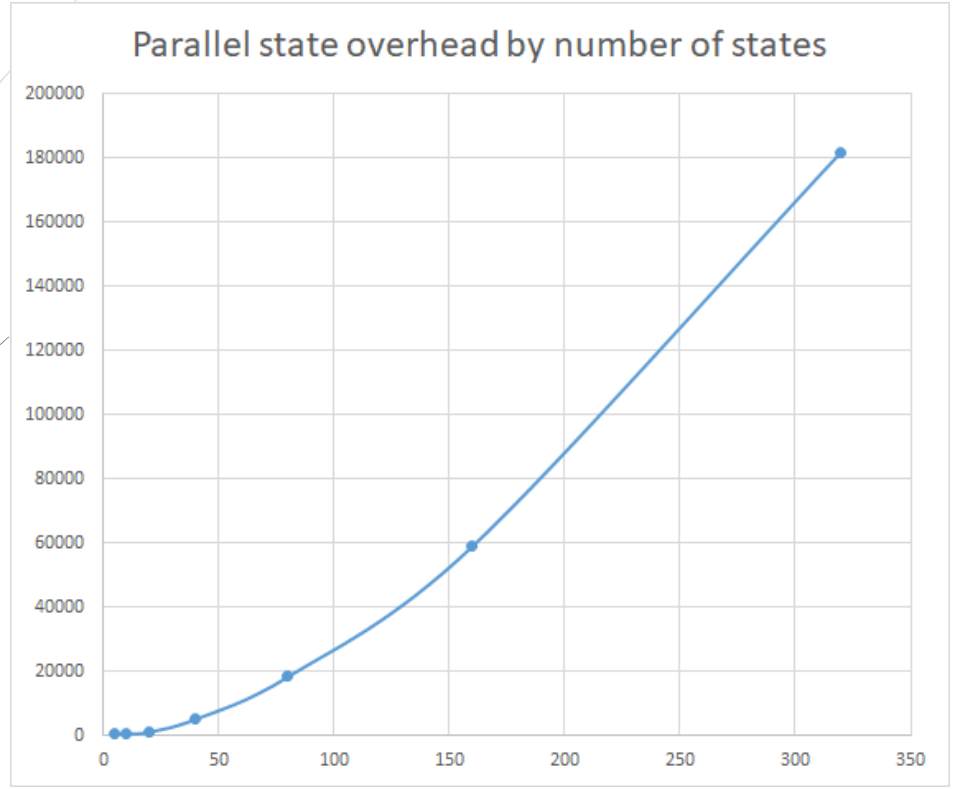

```
var tasks = new Task<long>[NSTEPS];
for (int i = 0; i < NSTEPS; i++)
{
  tasks[i] = context.CallActivityAsync<long>(
            "sleepAction");
}
await Task.WhenAll(tasks);
```

Experiments





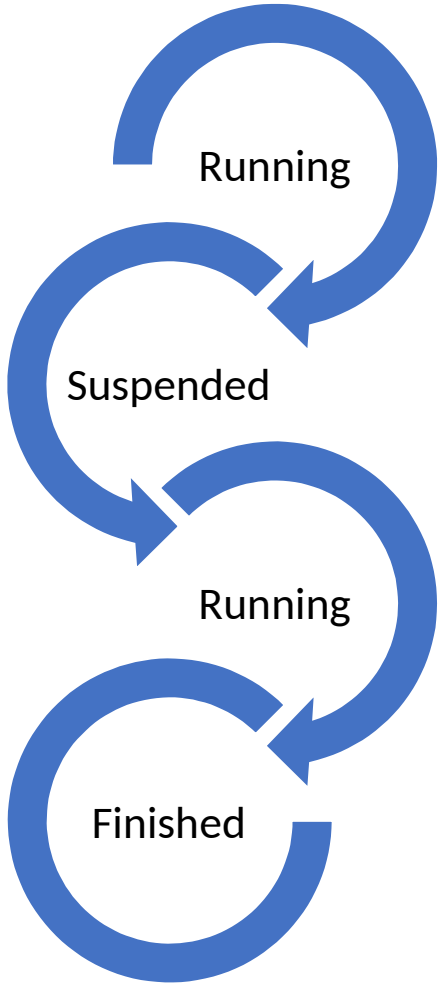
Experiments



Platform	Overhead (ms)		Increase (%)
	Without payload	With payload	
IBM Sequences	49.0	80.8	65%
IBM Composer	175.7	298.4	70%
AWS Step Functions	168.0	287.0	71%
Azure DF	766.2	859.5	12%

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

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

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*