From Stateless Functions to Stateful Applications

with Azure Durable Functions

SEBASTIAN BURCKHARDT
(MICROSOFT RESEARCH)

DAVID JUSTO
(MICROSOFT)

KONSTANTINOS KALLAS
(U. OF PENNSYLVANIA)

AND MANY MORE CONTRIBUTORS: **CHRIS GILLUM** (MICROSOFT), **CHRISTOPHER MEIKLEJOHN** (CARNEGIE MELLON UNIVERSITY), **ZHE YANG, JEFF HOLLAN, ANTHONY CHU, AMANDA DEEL, ANATOLI BELIAEV, CONNOR MCMAHON, KANIO DIMITROV, KATY SHIMIZU, TED HART, TSUYOSHI USHIO** (MICROSOFT)

CLOUD APPLICATIONS

- Implementing and deploying an application on the cloud is a pain
  - How many resources to allocate?
  - How to achieve reliability?
  - How to adapt to load increase?
  - What about periods of inactivity?
  - Monitoring application state?

DEVELOPERS CHOOSE

Control ⟷ Productivity

Assembly

C, C++

Java, C#

JavaScript

Python

Haskell

DEVELOPERS CHOOSE

Control ⟷ **Productivity**

Infrastructure
as a Service

Containers
as a service

Platform
as a service

Functions
as a Service
e.g.
AWS Lambda, Azure Functions

DEVELOPERS CHOOSE

SERVERLESS

Control

Productivity

Infrastructure
as a Service

Containers
as a service

Platform
as a service

Functions
as a Service
e.g.
AWS Lambda, Azure Functions

TOP-GROWING CLOUD SERVICES 2019

Place	Service	Growth	2018 Use	2019 Use
#1 (tie)	Serverless	50%	24%	36%
#1 (tie)	Stream Processing	50%	20%	30%
#3	Machine Learning	44%	18%	26%
#4	Container-as-a-Service	42%	26%	37%
#5	IoT	40%	15%	21%
#6	Data warehouse	38%	29%	40%
#7	Batch processing	38%	26%	36%

Source: Forbes, RightScale 2019 state of the cloud report

So what exactly is serverless?

SERVERLESS FUNCTIONS

```
string helloworld()
{
    return "Hello, World";
}
```

- Easy to deploy
- Elastic scale
- Load-based cost (e.g. pay per invocation)
- Free language choice, easy REST interface

```
 > curl http://my-function-app.azure.com/helloworld
Hello, World
```

COMMON MISCONCEPTION
SERVERLESS FUNCTIONS ARE NOT "PURE".
THEY CAN CALL OTHER SERVICES.

Functions can **call** external services:

key-value stores, queues, blob storage, pub-sub, databases, ...

= the "standard library" of cloud programming!

```
async void delete_all()
{
    await cloudstorage.delete_file("*");
}
```

```
async void counter_increment()
{
    var current = await cloudstorage.read("counter");
    current = current + 1;
    await cloudstorage.write("counter");
}
```

"SERVERLESS" IS NOT JUST COMPUTE

Serverless Compute	"Serverless" Storage	"Serverless" Transport
Stateless Functions	Table Storage Blob Storage	Queue Storage

Serverless is already very useful today,

but…

… THERE ARE SEVERAL PAIN POINTS AROUND STATE MANAGEMENT AND SYNCHRONIZATION.

- Sychronization

  functions can interleave and race, synchronization via storage is challenging

- Partial execution

  hosts can fail in the middle of a function, leaving behind inconsistent state

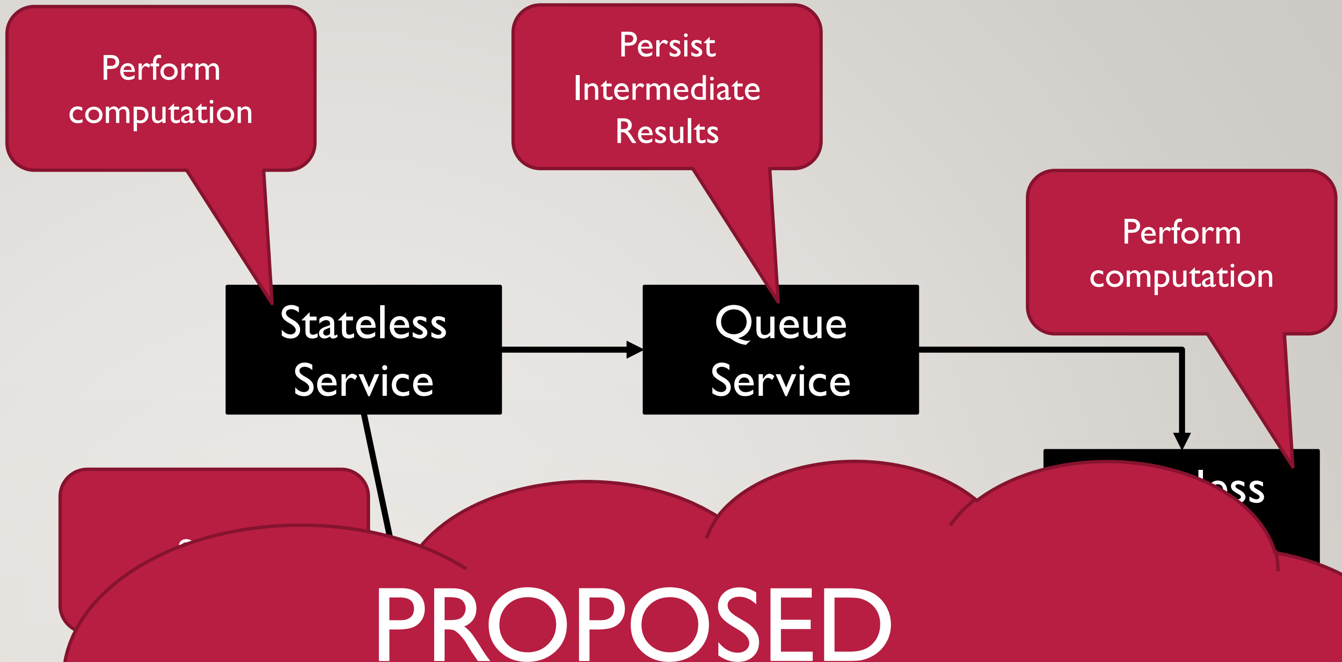
- Cost/Performance

  Double billing if a function waits for another function

  Lots of calls to storage, lots of data movement => wastes time, CPU = money

SERVERLESS APPLICATIONS

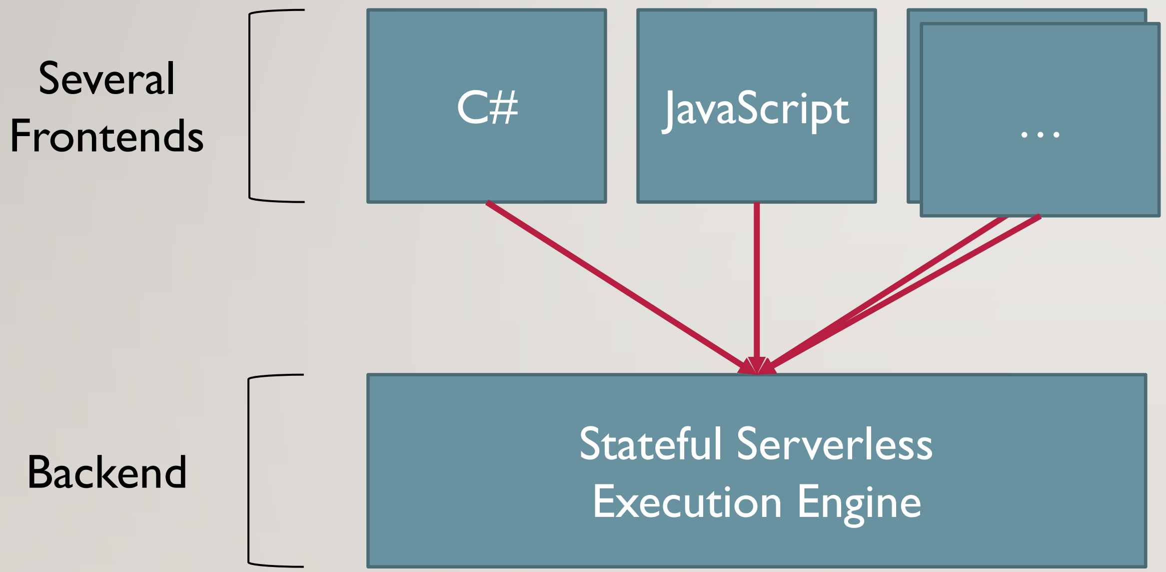
Implementing a non-trivial applications on the cloud ends up looking like this



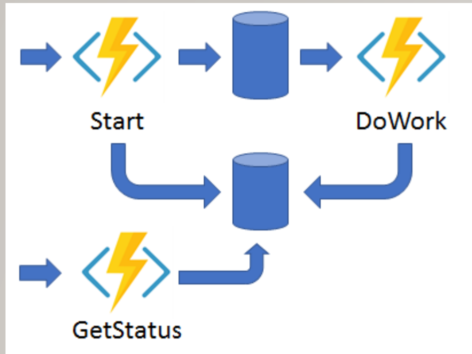
PROPOSED SOLUTION:

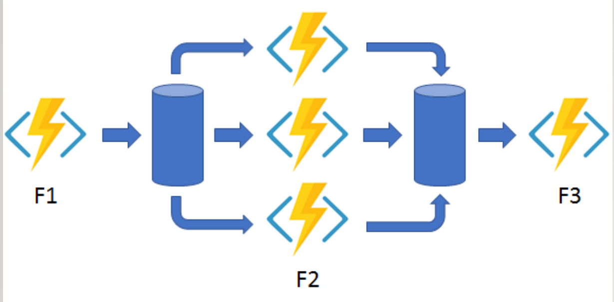
Abstractions for stateful serverless programming

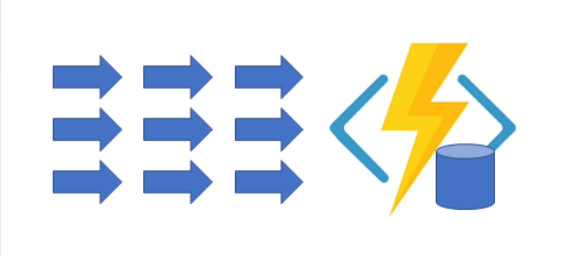
ABSTRACTION LAYERS

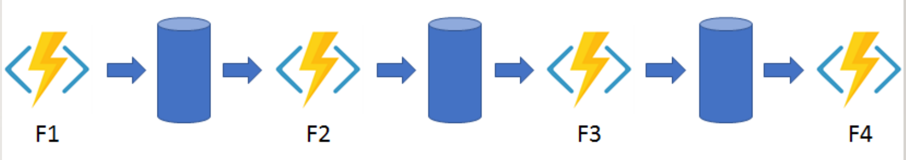


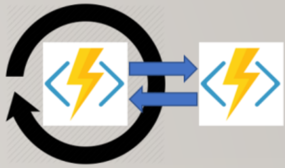
- Front End:
  - Task-Parallel Code
  - Workflows and Actors
- Back End:
  - Reliable distributed execution
  - Language agnostic











THE AZURE DURABLE FUNCTIONS PROGRAMMING MODEL

State & Synchronization for Serverless

2 NEW TYPES OF STATEFUL FUNCTIONS

Activities

≈ Stateless Functions

+

Orchestrations

≈ Workflow Functions

+

Entities

≈ Actor Functions

Activities
(≈ Stateless Functions)

Orchestrations
(≈ Workflow Functions)

- Reliably compose functions using task-parallel paradigm.
  - e.g. a sequence of functions, or multiple parallel function calls
- Advantages:
  - Expressive: very simple code for common scenarios
  - Solves the partial execution problem

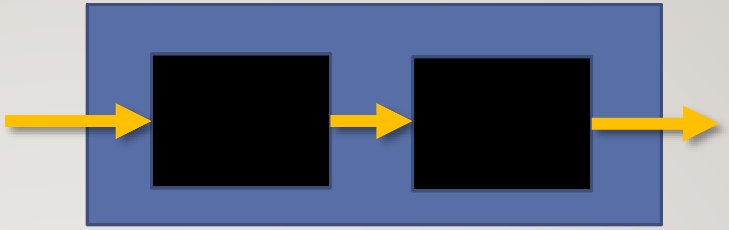
    Automatically recover state of workflow.
  - Solves the double billing problem

    Can persist execution state in storage - don't get charged while waiting

ORCHESTRATIONS:
WHAT'S NEW ABOUT IT?

- Do what was traditionally done with workflow "languages" (e.g. XML-based, or graphical designers)
- But written in task-parallel async-await style code.
- Thus, we get to enjoy the *maturity of the host language:*
  - *all of the standard sequential control flow (conditionals, loops, switches, ...)*
  - *all of the task-based asynchronous control flow (await, Task.WhenAll, Task.WhenAny, ...)*
  - *all of the exception handling (try/catch/finally)*
  - *all of the existing tooling (IDE, debugger etc.)*

EXAMPLE 1



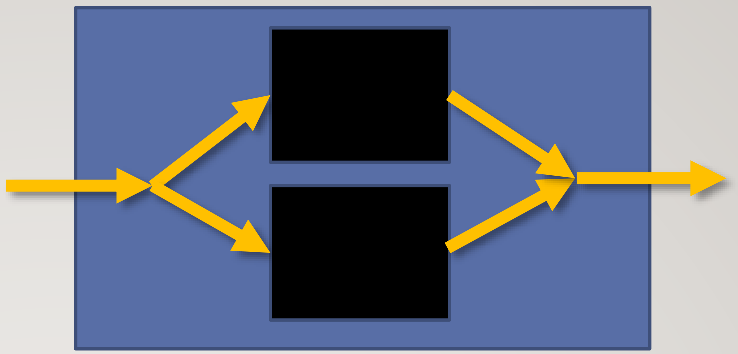
- Simple sequence: Upload file, then update index

```
void uploadImage(string name, byte[] data)
{
    await addToBlobStorage(name, data);

    await updateIndex(name);
}
```

```
void addToBlobStorage(string name, byte[] data)
{
    …
}
```

```
void updateIndex(string name)
{
    …
}
```

EXAMPLE 2



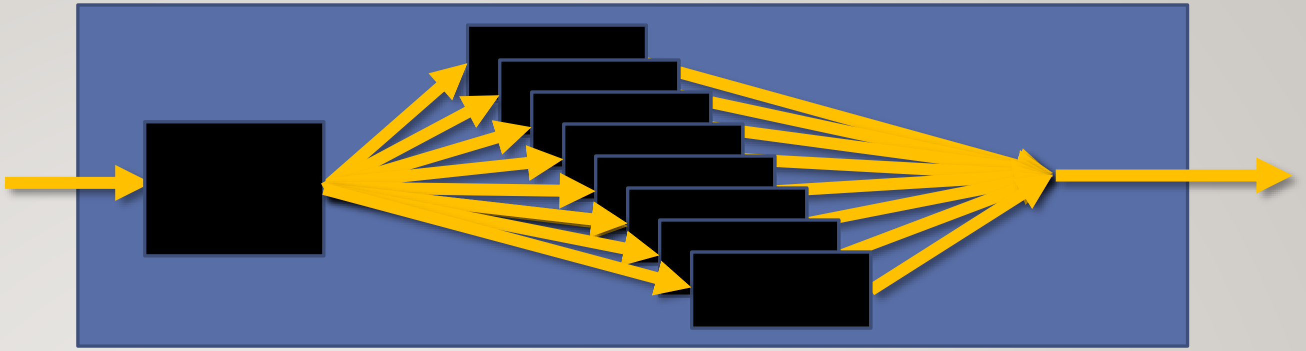
- Same **but *in parallel***

```
void uploadImage(string name, byte[] data)
{
    await Task.WhenAll(
        addToBlobStorage(name, data),
        updateIndex(name)
    );
}
```

```
void addToBlobStorage(string name, byte[] data)
{
    …
}
```

```
void updateIndex(string name)
{
    …
}
```

EXAMPLE 3



- Process all files in a directory, return sum of results

```
void processFiles(string directory)
{
    var files = await listFiles(directory);
    var tasks =  files.Select(f => process(f)).ToList();
    await Task.WhenAll(tasks);
    return tasks.Select(t => t.Result).Sum();
}
```

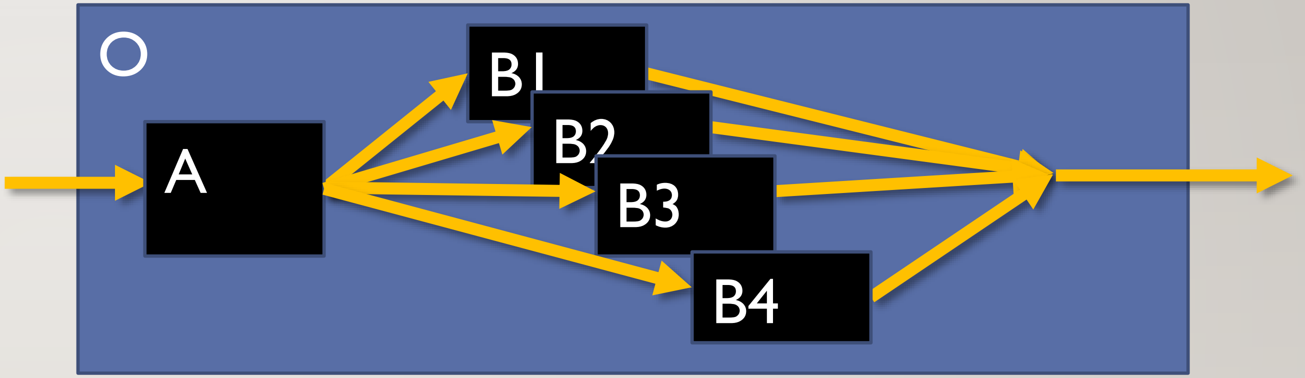
```
list<string> listFiles(string directory)
{
    …
}
```

```
int process(string file)
{
    …
}
```

RELIABLE EXECUTION

- State of workflow is persisted as *history of events.*

O started
A() started
A returned -> [f1,f2,f3,f4]
B1(f1) started
B2(f1) started
B3(f3) started
B4(f4) started
B2 returned 32
B4 returned 0
B1 returned 120
B0 returned 1
O returned 153

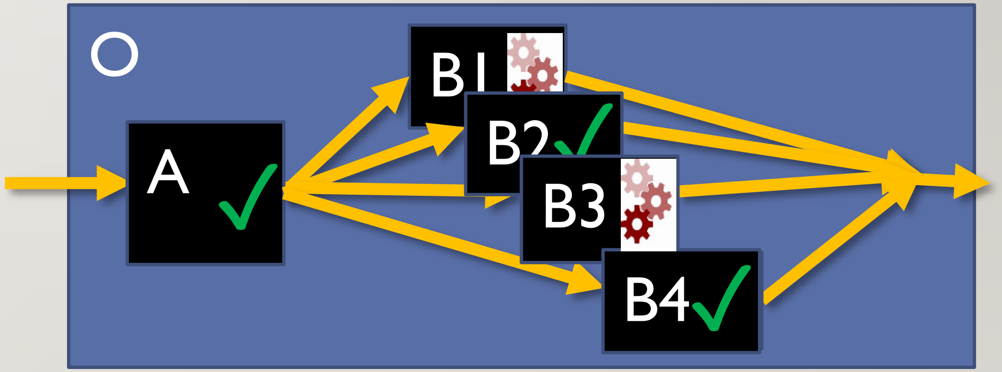


- History can be inspected in storage for debugging / monitoring purposes!
- Can rehydrate intermediate states (after crash or inactivity) from history
- Proceed in episodes, each processes batch of events, billed as 1 function inv.

EXAMPLE: PARTIAL HISTORY ≈ INTERMEDIATE STATE

O started
A() started
A returned -> [f1,f2,f3,f4]
B1(f1) started
B2(f1) started
B3(f3) started
B4(f4) started
B2 returned 32
B4 returned 0

≈



REHYDRATE STATE FROM HISTORY *BY REPLAY*

O started
A() started
A returned -> [f1,f2,f3,f4]
B1(f1) started
B2(f1) started
B3(f3) started
B4(f4) started
B2 returned 32
B4 returned 0

```
void processFiles(string directory)
{
    var files = await listFiles(directory);
    var tasks =  files.Select(f => process(f)).ToList();
    await Task.WhenAll(tasks);
    return tasks.Select(t => t.Result).Sum();
}
```

- Replay code but *do not restart activities immediately,* use placeholder task
- Substitute recorded results into placeholders during replay (A, B2, B4)
- At end of replay restart activities for remaining placeholders (B1, B3)

CAVEAT: CODE MUST SATISFY 2 REQUIREMENTS

- **Determinism of orchestrators**

Orchestrator must be deterministic, otherwise replay diverges

- **Idempotence of activities**

Activities that crash before persisting result are restarted during recovery

User responsibility : separate deterministic coordination from nondeterministic work

ACCIDENTAL NONDETERMINISM: MITIGATIONS? SOLUTIONS?

- Document common nondeterminism sources
  - time of day, random generators, I/O, global static variables
  - User must wrap these in activities, or use built-in deterministic versions
- Include static analysis tool to help find mistakes
- Some other potential ideas:
  - Use language with effect system (e.g. Daan Leijen's Koka)
  - Automatic wrapping of request handlers (JavaScript), work w/ Christopher Meiklejohn

Entities

≈ Actor Functions

ENTITIES
= DURABLE APPLICATION STATE

- Entity = smallest piece of state, a "single key-value pair", a *virtual actor* (Orleans)
- Runtime delivers "operations" (messages) to entities via *ordered async channels*
- Runtime executes operations on entities, *one at a time*. Operations can
  - read and update state
  - send messages
  - perform external calls
- Durable: All state (incl. messages) reliably kept in cloud storage

EXAMPLE ENTITY: BANK ACCOUNT

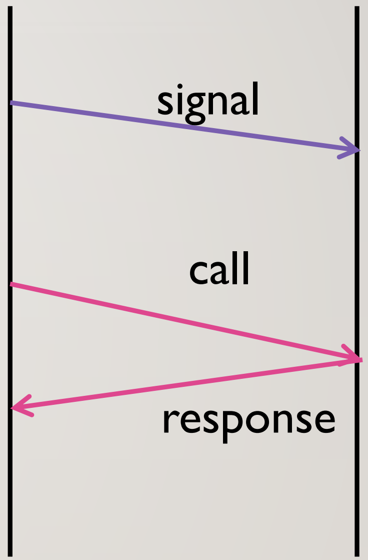
- each entity identified by a (name,key) pair, e.g. (“AccountEntity”,“32974-234093-00”)
- Accessible via interface

```
public interface IAccount
{
    Task<int> Get();
    Task Modify(int Amount);
}
```

```
public class Account : IAccount
{
    public int Balance { get; set; }

    public Task<int> Get()
    {
        return Task.FromResult(Balance);
    }

    public Task Modify(int Amount)
    {
         Balance += Amount;
         return Task.CompletedTask;
    }

    // boilerplate for class-based syntax
    [FunctionName(nameof(Account))]
    public static Task Run([EntityTrigger]
        IDurableEntityContext ctx) =>
        ctx.DispatchAsync<Account>();
}
```

CALL VS. SIGNAL

- An entity can signal another entity
  send message, fire and forget
- An orchestration can call an entity
  and wait for ack/result



- But entities cannot call entities (to prevent deadlock)
  different from virtual actors in Orleans, which can deadlock.

SYNERGY !

Orchestrations

≈ Workflow Functions

+

Entities

≈ Actor Functions

- Enables revolutionary novel synchronization construct:

!!! Critical sections !!!

just kidding of course, that's the most standard one of all;
but we can't usually do it in distributed systems because of failures!

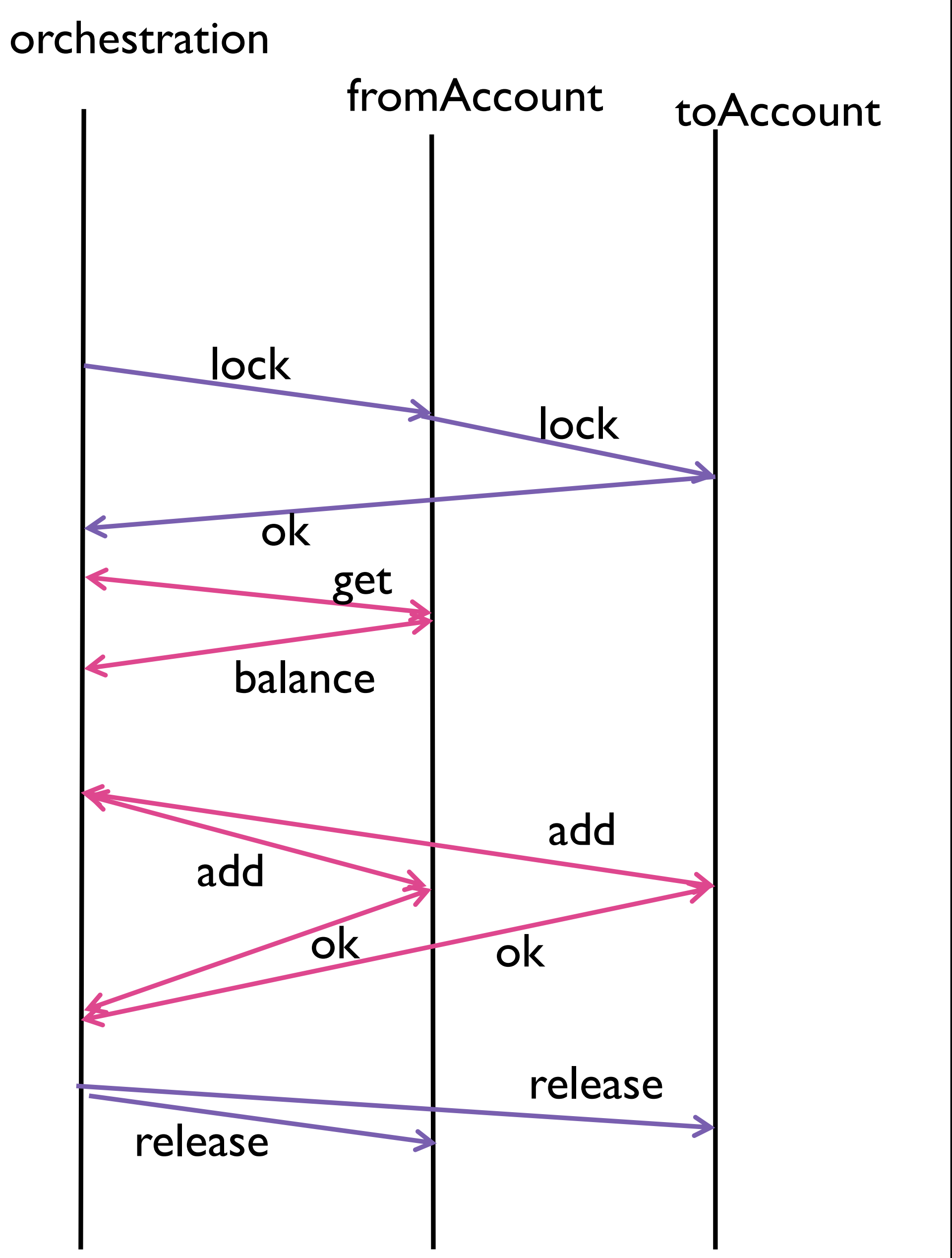
- Effective for preventing unwanted races and interleavings (doh)
- Critical sections do not require special "failure" handling, such as ability to roll back effects

EXAMPLE: TRANSFER FUNDS

```
var fromAccount = new EntityId("Account", from);
var toAccount = new EntityId("Account", to);

using (await ctx.LockAsync(fromAccount, toAccount))
{
    var source = context.CreateEntityProxy<IAccount>(fromAccount);
    var destination = context.CreateEntityProxy<IAccount>(toAccount);

    if (amount <= await source.Get())
    {
        await Task.WhenAll(
            source.Modify(-transferAmount),
            destination.Modify(transferAmount)
        );
    }
}
```

MESSAGE DIAGRAM

```
var fromAccount = new EntityId("AccountEntity", from);
var toAccount = new EntityId("AccountEntity", to);

using (await ctx.LockAsync(fromAccount, toAccount))
{
    var source = context.CreateEntityProxy<IAccount>(from
    var destination = context.CreateEntityProxy<IAccount>

    if (amount <= await source.Get())
    {
        await Task.WhenAll(
            source.Modify(-transferAmount),
            destination.Modify(transferAmount)
        );
    }
}
```



GUARANTEED DEADLOCK FREEDOM

Runtime-enforced rules prevent deadlocks:

- Runtime acquires locks in order (fixed global total order).
- Critical sections cannot be nested.
- Within a critical section:
  - can call only entities that were locked.
  - can signal only entities that were not locked.
  - cannot call the same entity more than once in parallel.

STATUS

- Azure Durable Functions have been GA for about 2 years now.
- Popular & growing: 50% of Azure Functions users use them (recent survey)
- Entities & critical sections are a new feature, shipped last year,
  (building on research w/ intern Christopher Meiklejohn)
- Much work left to be done
  - formal semantics for "stateful serverless applications"
  - build new implementation w/ more aggressive optimizations

ONGOING WORK:
SEMANTICS & OPTIMIZATIONS

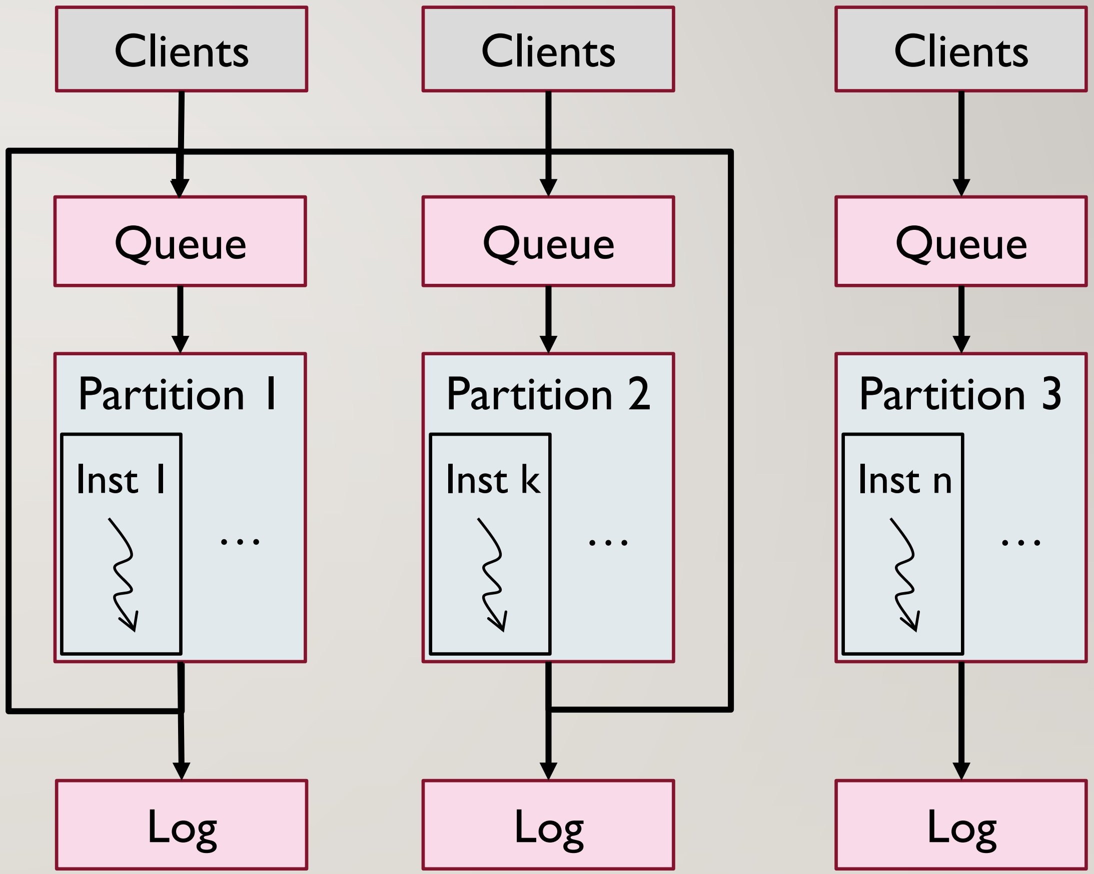
ABSTRACT SEMANTICS

- Two computation units:
  - Stateless Tasks
  - Stateful Instances
- Communication through messages
- State is event history

	$\rightarrow$	m	Client Transition
m	$\rightarrow$	$m'_1 m'_2 \ldots$	Task Transition
$h\, m_1 m_2 \ldots$	$\rightarrow$	$h' m'_1 m'_2 \ldots$	Instance Transition

IMPLEMENTATION

- Distributed – multiple partitions
- Reliable – exactly/at least once
- Executions are persisted incrementally
- Elastic – adapting to load changes



IMPLEMENTATION 2.0

- Main sources of overhead:
  - Storage Accesses
  - Network Communication
- Optimizations:
  - Speculative Message Exchange
  - In memory processing of same-partition messages
  - Message Batching
- WIP Proof of correctness

DEV TOOLING AND EXPERIENCE

A tour of the programming experience with Durable Functions

HELPING DEVS BY...

- Preventing common errors via live code analysis
- Providing common patterns to quickly scaffold solutions
- Allowing them to use their preferred PL for the job

HELPING DEVS BY...

- **Preventing common errors via live code analysis**
- Providing common patterns to quickly scaffold solutions
- Allowing them to use their preferred PL for the job

MEETING CODE CONSTRAINTS

Orchestrator

Deterministic

Activity

Activity

Activity

Activity

Activity

Non-Deterministic

LIVE CODE ANALYZER

Generating GUIDs

Reading Enviroment Variables

Reading DateTime objects

… and so on …

Constraint Violations

roslyn

Live Code Analyzer

Alerts user of constraint violations

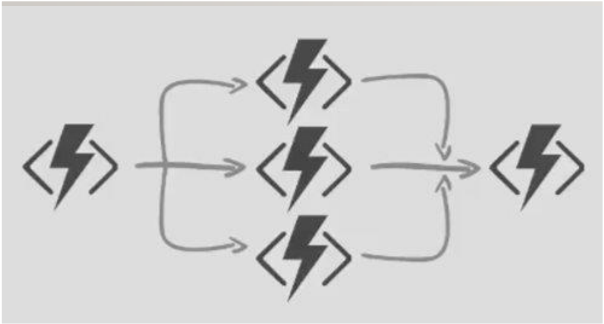
Suggests replay-safe APIs and other refactorings

Programmer Feedback

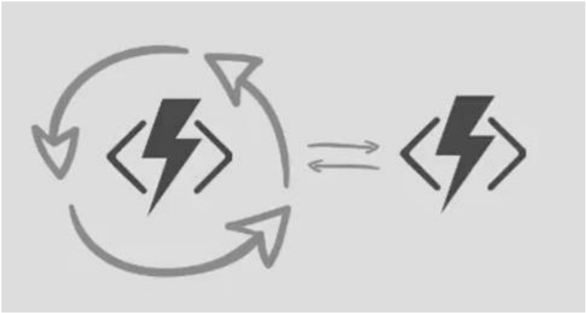
HELPING DEVS BY...

- Preventing common errors via live code analysis
- **Providing common patterns to quickly scaffold solutions**
- Allowing them to use their preferred PL for the job

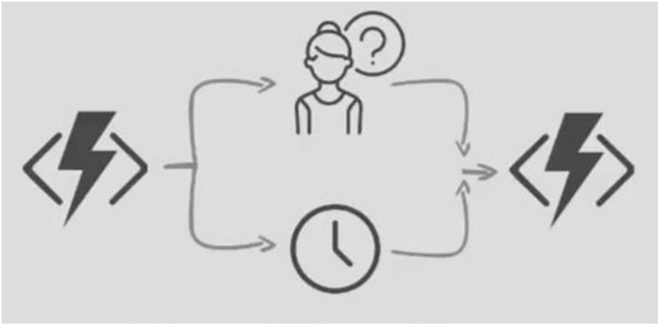
GETTING UP TO SPEED WITH DURABLE



Fan-Out Fan-In



Monitoring long-running workflows



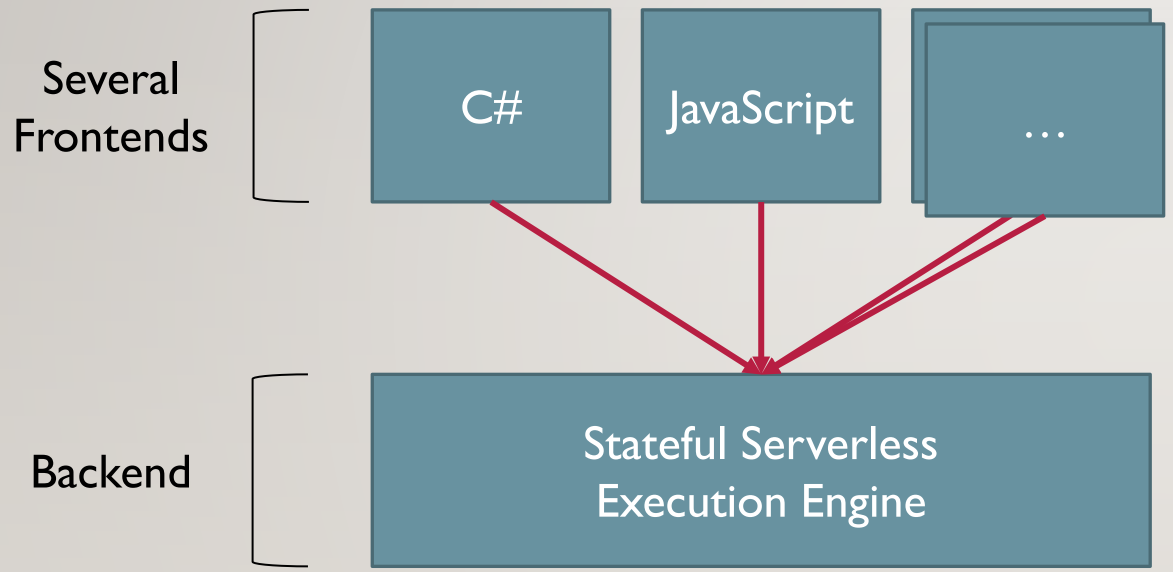
Timed Human-in-the-loop computation

Quick-start samples and templates for each host PL

HELPING DEVS BY...

- Preventing common errors via live code analysis
- Providing common patterns to quickly scaffold solutions
- **Allowing them to use their preferred PL for the job**

USE THE RIGHT PL FOR THE JOB



- Open-sourced SDKs for .NET, JavaScript, TypeScript
- *Extremely soon:* SDKs for two highly-requested host PLs
- Working to facilitate the creation of third-party SDKs

DEMO: BUILD A SERVERLESS BANK