WOSCx June 16, 2022

Netherite: Efficient Execution of Serverless Workflows

Sebastian Burckhardt
Microsoft Research

in collaboration with Badrish Chandramouli (MSR), Chris Gillum (Microsoft Azure), David Justo (Microsoft Azure), Konstantinos Kallas (UPenn), Connor McMahon (Microsoft Azure), Christopher Meiklejohn (CMU), Zhu Xiangfeng (UW)

Agenda

Motivation

Using serverless workflows (e.g. Durable Functions) for composition and coordination of services

Netherite Architecture

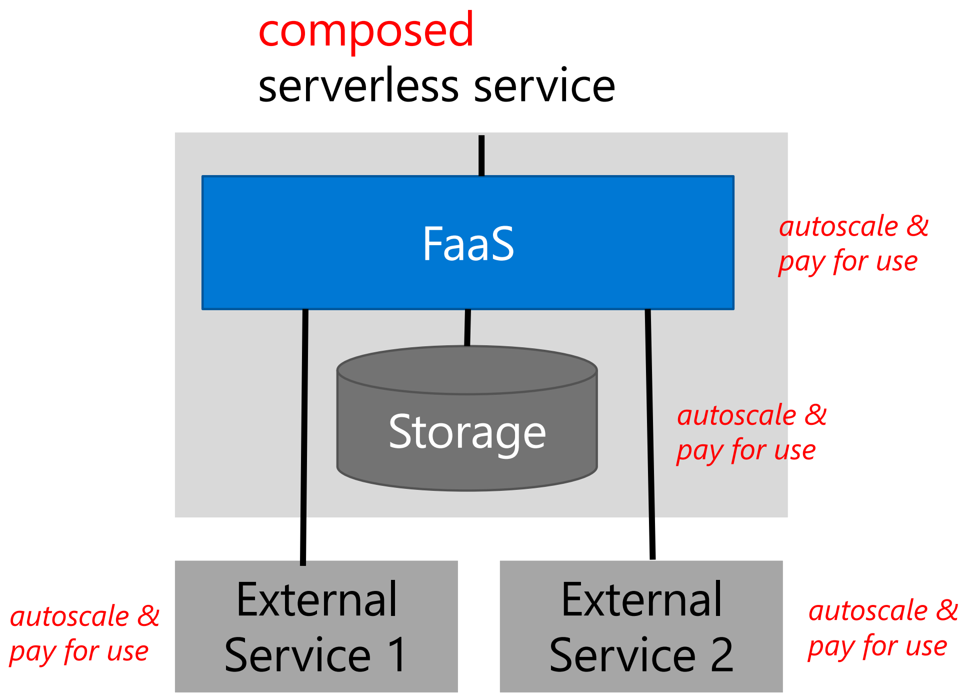
Efficient execution of workflows on an elastic cluster

Serverless is about Developer Productivity

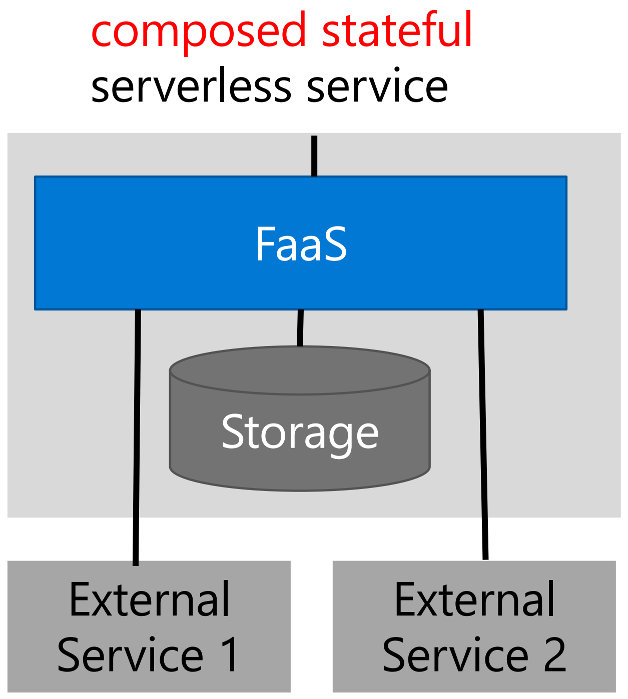
- Simplify development of cloud services by delegating server management
  - to commercial provider (e.g. Azure, AWS, Google, IBM ...)
  - or just to lower layer of the stack (e.g. Kubernetes + KEDA scaler)

FaaS is not just another component. It's the glue!

With FaaS, you can build a serverless service entirely from serverless components:



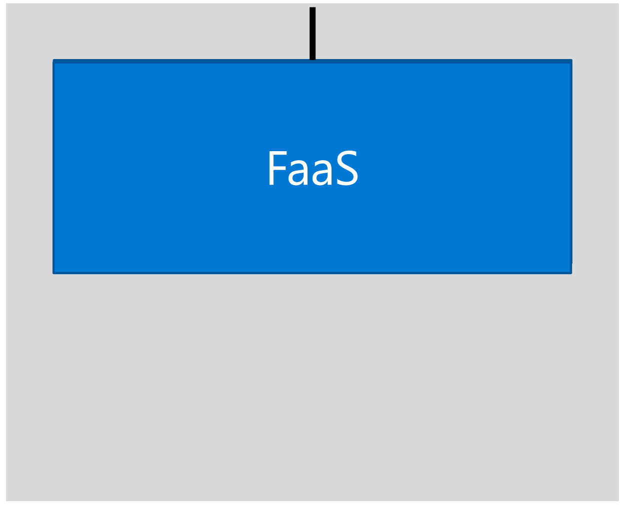
Problem: Explicit State and Synchronization Are Hard

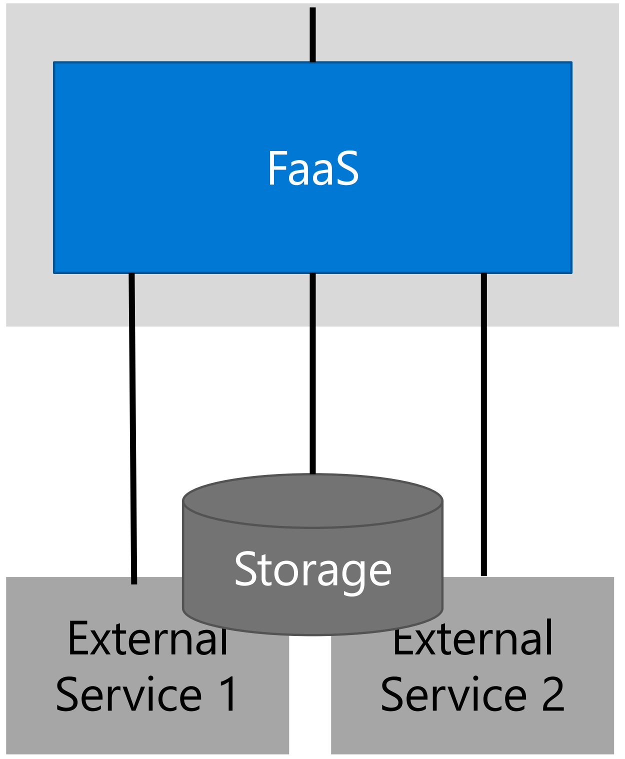


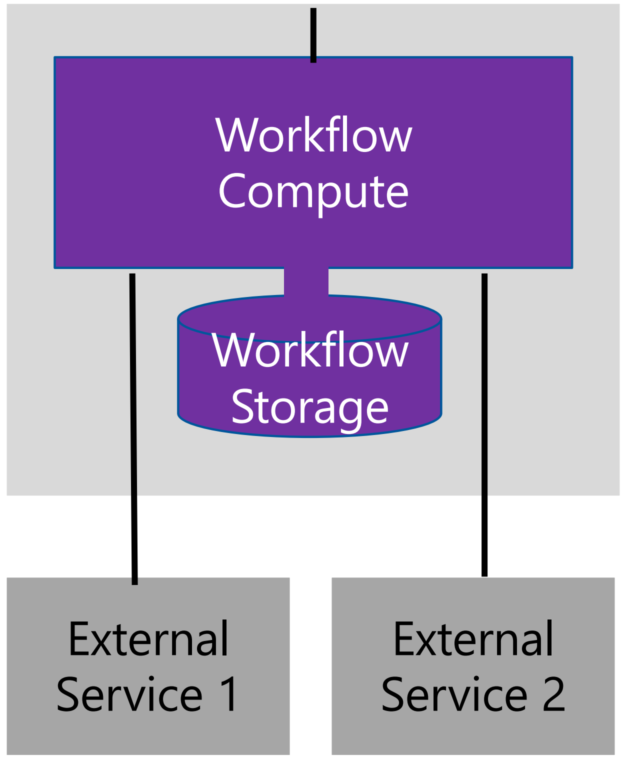
- C1 -- Execution Progress
  - What if function fails in the middle of execution?
  - What if function times out?
- C2 -- Persistent Application State
  - Functions do not have local storage
  - All persistent state needs to be saved explicitly
- C3 -- Exactly-Once Processing
  - Functions may process event multiple times
  - Developers must make functions idempotent
- C4 -- Concurrency
  - Synchronization via storage is difficult (e.g. optimistic e-tags, pessimistic leases)

Solution: evolve the programming model

	Stateless FaaS	FaaS + Storage + Services	Serverless Workflows
state management, synchronization	none	explicit (via storage)	implicit (programming model)

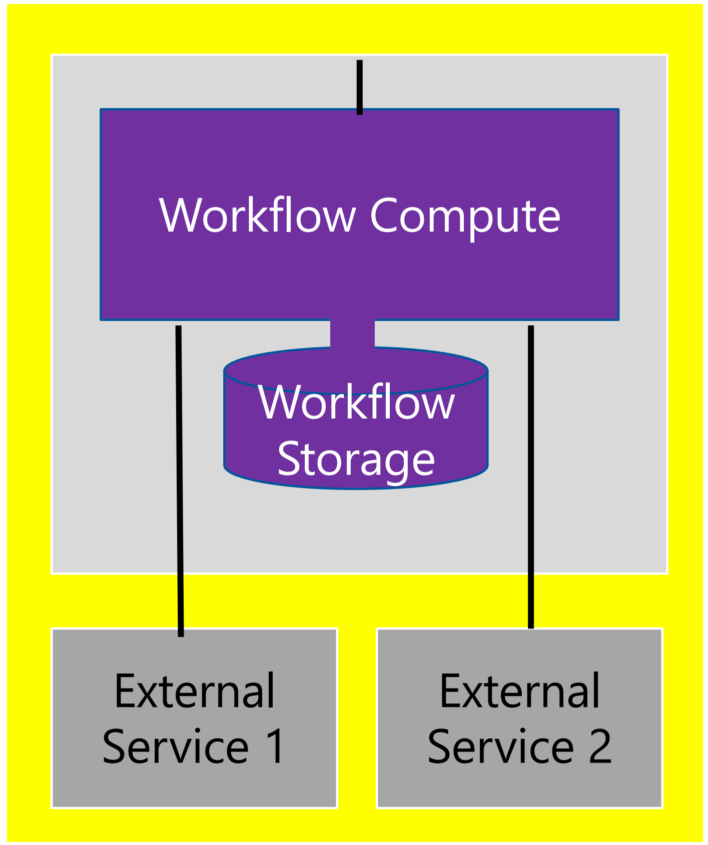






Serverless Workflows

We anticipate that the majority of cloud services will be built in this way in the future.

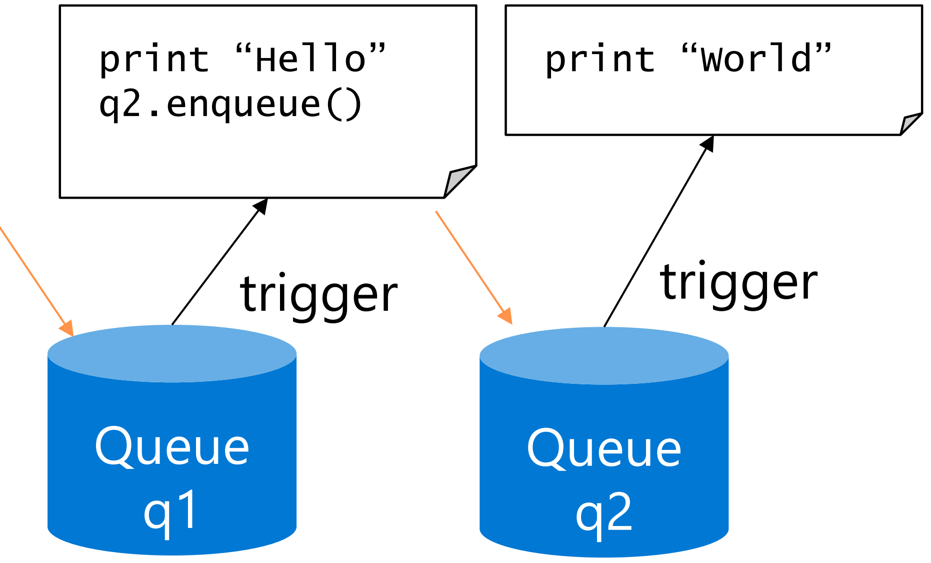


Two Research Questions

- How to express workflows?
- How to execute workflows?

How to express workflows?

FaaS + Storage	Serverless Workflows



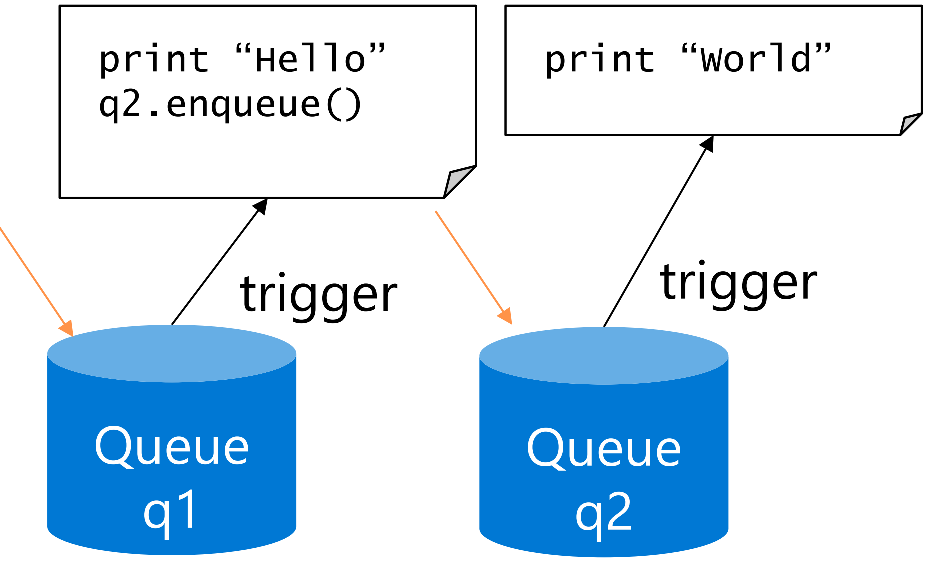
```
print "Hello"
q2.enqueue()
```

```
print "World"
```

Example:
FaaS w/ storage triggers

How to express workflows?

FaaS + Storage



Example:
FaaS w/ storage triggers

Serverless Workflows

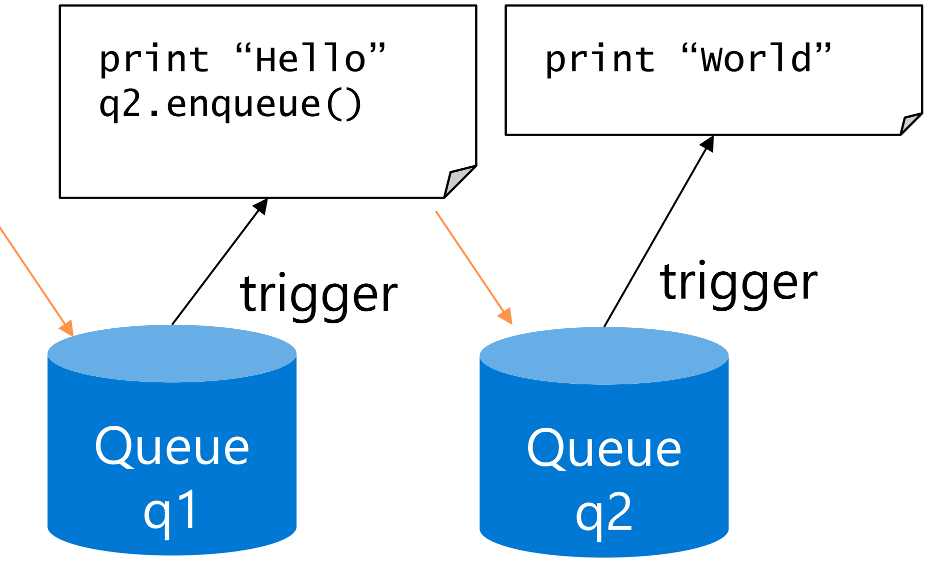
Declarative

```
{
  "StartAt": "Hello",
  "States": {
    "Hello": {
      "Type": "Pass",
      "Next": "World"
    },
    "World": {
      "Type": "Pass",
      "Result": "World",
      "End": true
    }
  }
}
```

Examples:
AWS step functions
Azure Logic Apps

How to express workflows?

FaaS + Storage



Example:
FaaS w/ storage triggers

Serverless Workflows

Declarative

```
{
  "StartAt": "Hello",
  "States": {
    "Hello": {
      "Type": "Pass",
      "Next": "World"
    },
    "World": {
      "Type": "Pass",
      "Result": "World",
      "End": true
    }
  }
}
```

Examples:
AWS step functions
Azure Logic Apps

Workflows-as-code

```
workflow start_trial()
{
    await timer(days: 30);
    if (!this.was_cancelled)
    {
        if (await charge_money())
        {
          await extend();
          return;
        }
    }
    await cancelsubscription();
}
```

Examples:
Azure Durable Functions
Temporal Workflows
Ray

Two Research Questions

- How to express workflows?

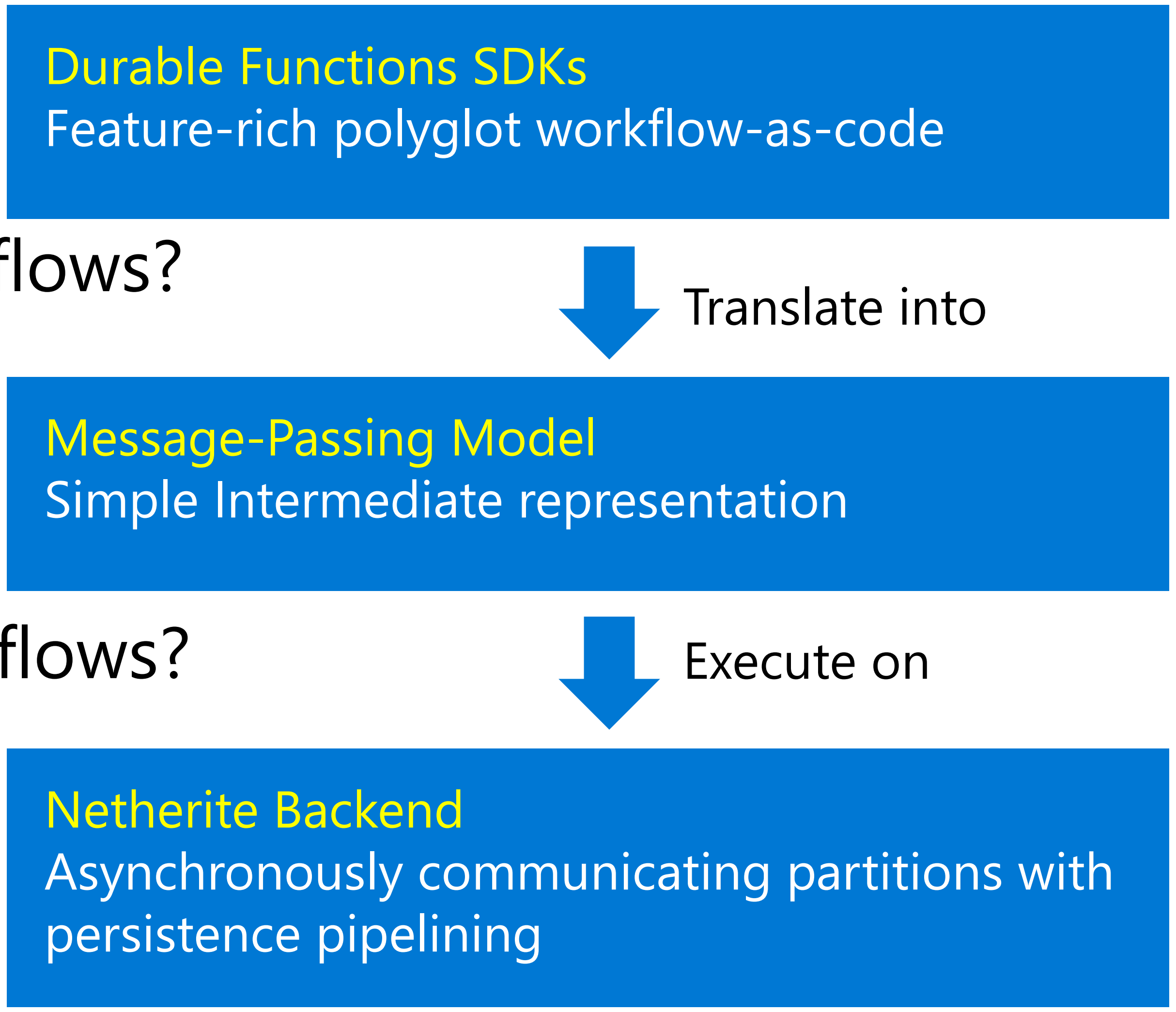
How to execute workflows?

Challenges:

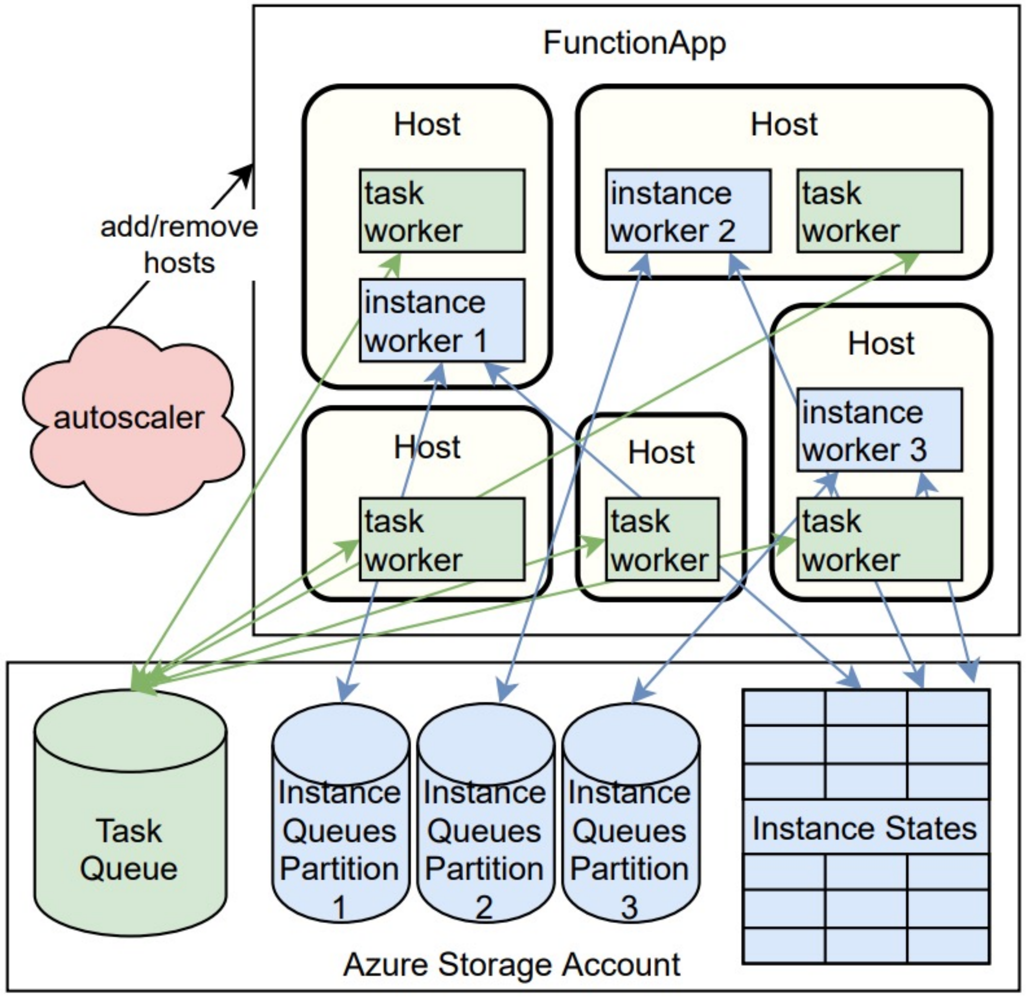
- Fault tolerance, distribution, and elastic scale
- Continuous persistence without excessive storage traffic

Our approach

- How to express workflows?
- How to execute workflows?

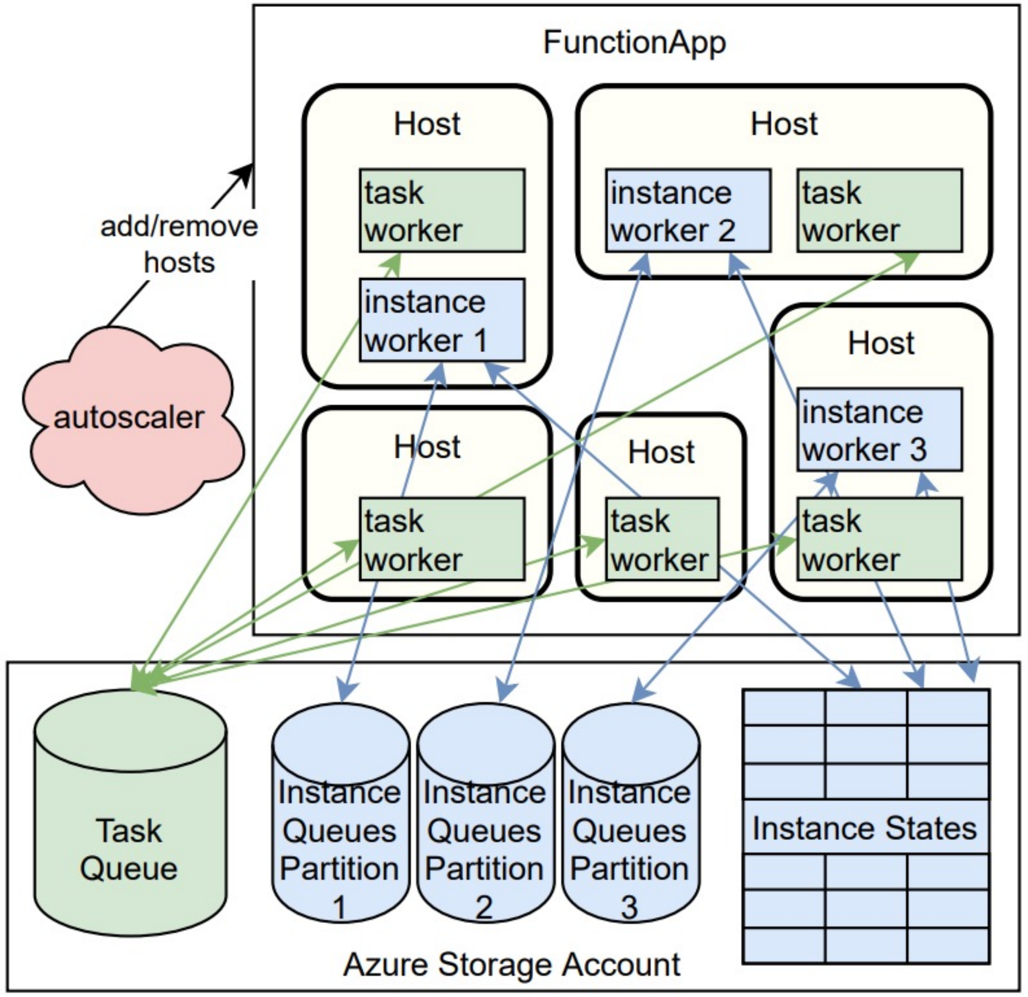

Durable Functions SDKs
Feature-rich polyglot workflow-as-code
Translate into
Message-Passing Model
Simple Intermediate representation
Execute on
Netherite Backend
Asynchronously communicating partitions with persistence pipelining

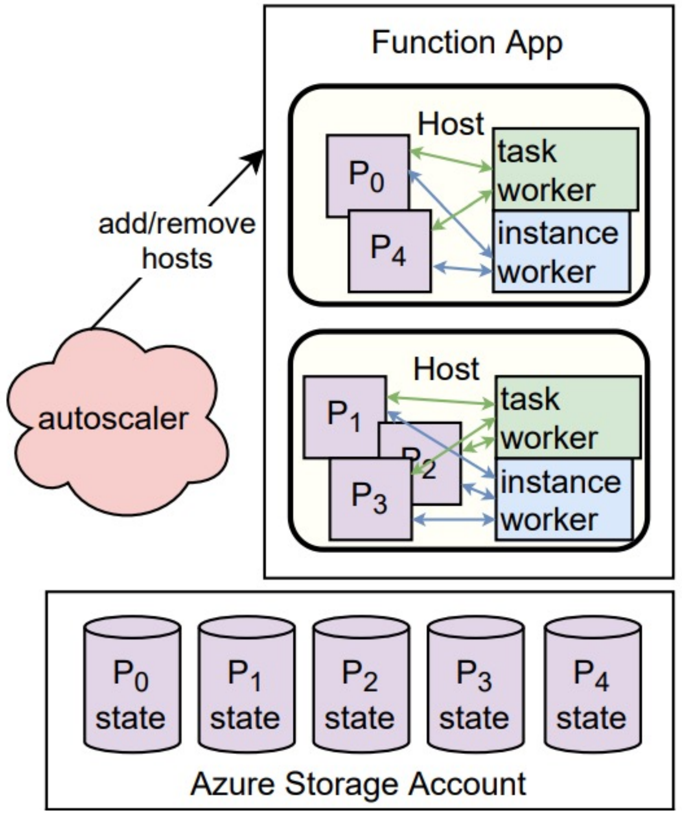
Original DF Implementation



Throughput bottleneck:
too many storage accesses

Original DF Implementation vs. Netherite

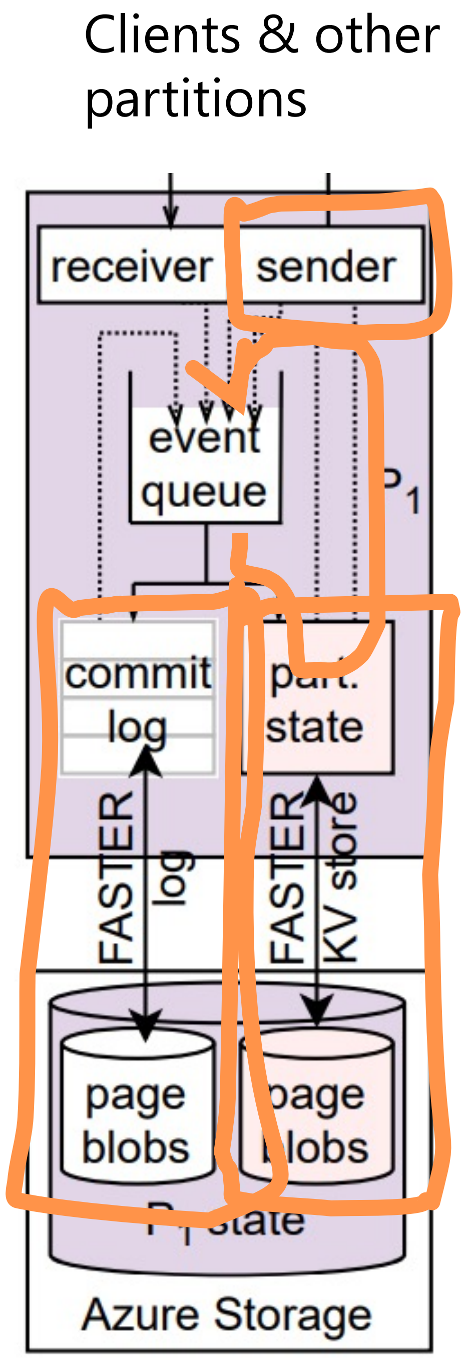




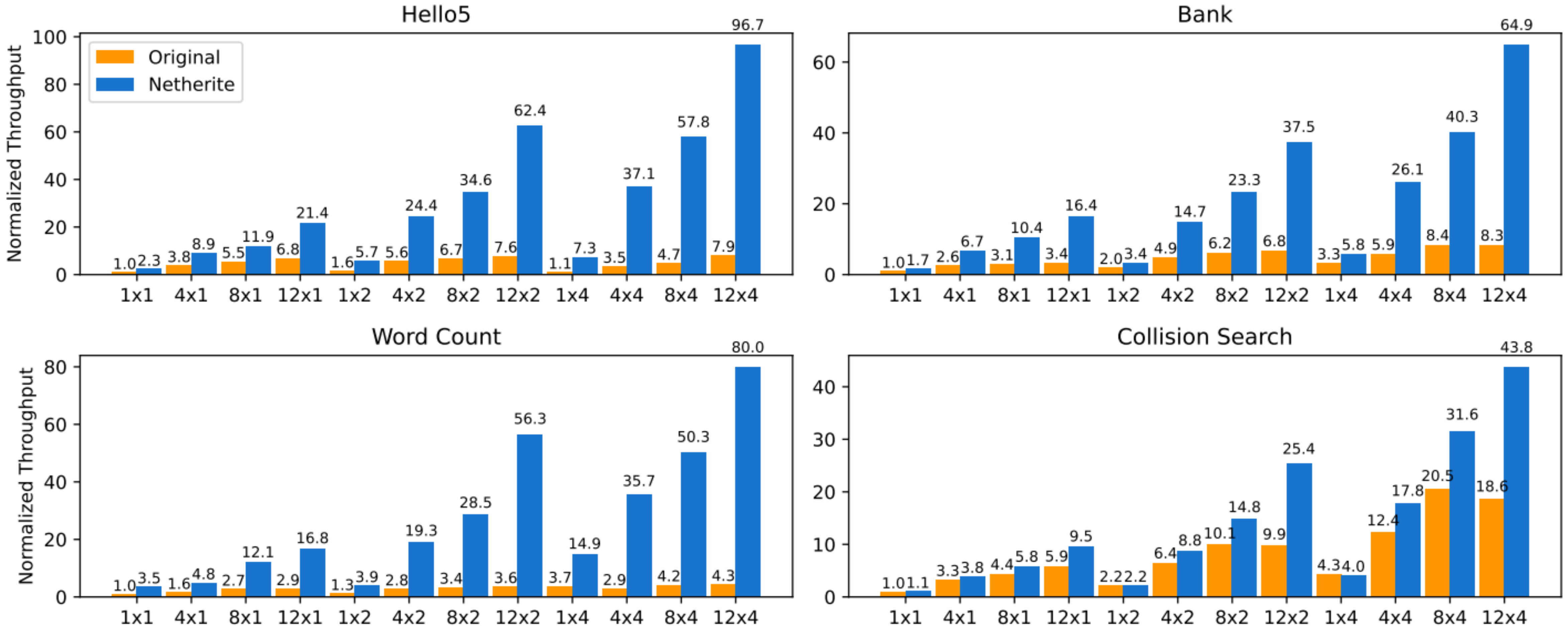
Partitions are largely autonomous
Communicate via asynchronous ordered channels
(no need for distributed 2-phase commit)

Partition Persistence Optimizations

- Commit log
  - commit many transitions with a single storage write (cf. group commit)
- Persistence Pipelining
  - Allow local dependency on uncommitted transitions (cf. early lock release)
  - Local only: outgoing messages wait for commit
- FASTER Key-Value Store enables
  - Larger-than-memory instance store
  - LRU cache for instance states
  - Asynchronous, incremental checkpointing
  - Instant Recovery (lazy loading)



Improved Throughput (Scalability)



Status

Microsoft Product

- *Azure Durable Functions* widely used, strong growth
- *Netherite Execution Engine* currently in public preview

Research Publications

- Early preprint ArXiV Februrary 2021
- DF semantics paper OOPSLA 2021
- Netherite paper VLDB 2022

Microsoft

Thank you!