Runtimes & Execution Engines

UNIVERSITY OF COPENHAGEN

TU Delft

Three families of systems

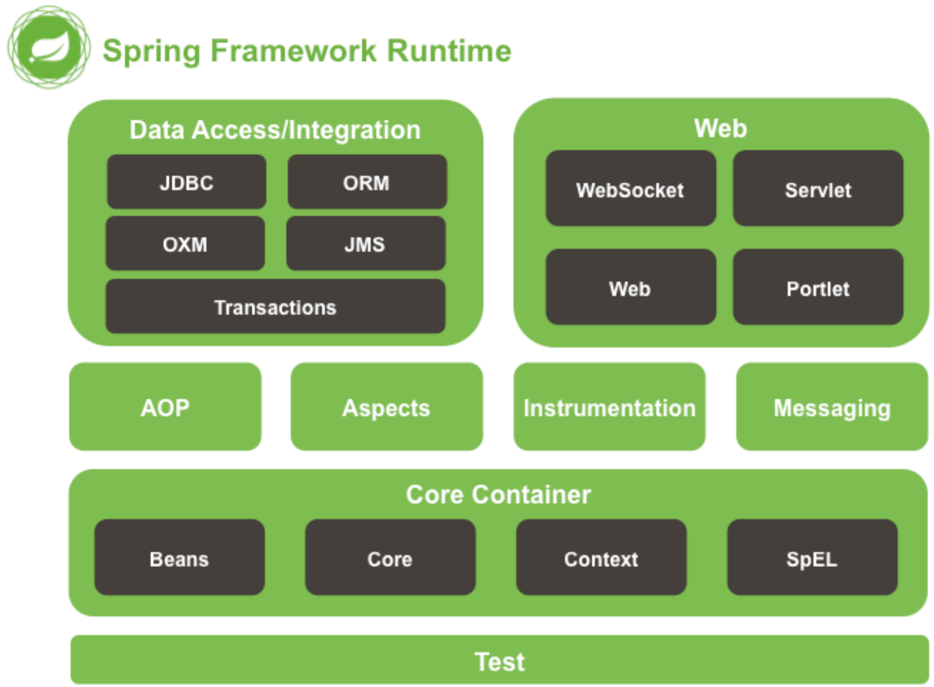
- Classic Microservices
  - **Spring**, Flask, **Dapr**, etc.
- Database + Serverless Functions
  - Boki, Beldi, Cloudburst, DBOS
- Dataflow-based
  - **Styx**
  - Statefun
- Actor-like
  - **Orleans**

Microservices

Orchestrations, service mesh, databases, etc.

Spring Framework

- Java-based web application framework
- Standardized interfaces to facilitate operational concerns
  - Security
  - Database API
  - Network
  - Modularity
  - Testing
- Industry-strength framework for cloud applications



Source: Spring Framework Docs (2025)

Spring through a “data management” perspective

ORM-heavy specifications

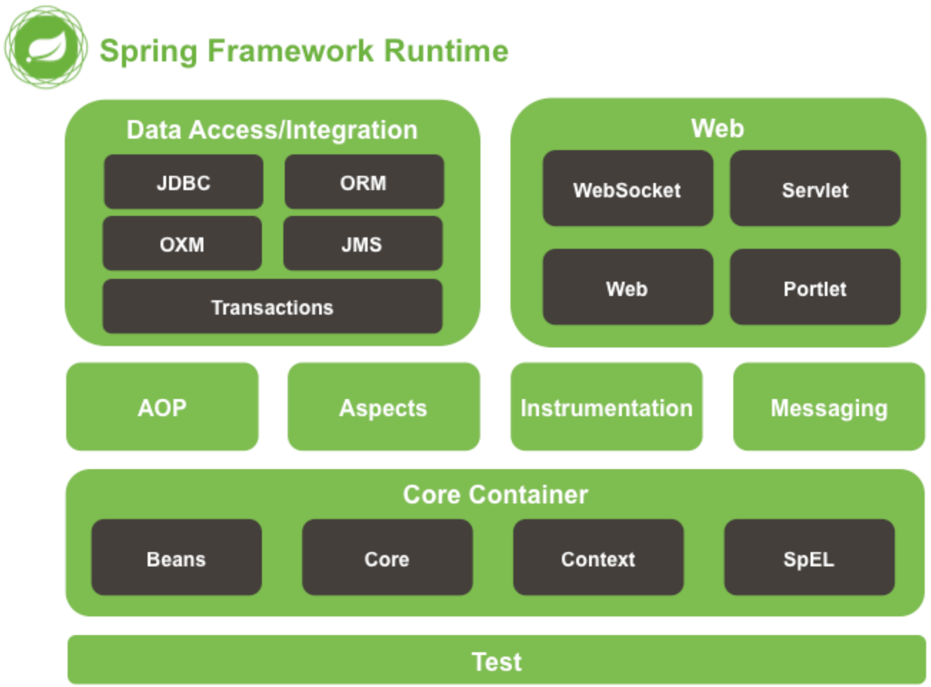
Developers specify local transactions via annotations

Limited support for distributed transactions

  Not popularly pursued

Wide library support for event-driven communication

Safety and liveness at developers' hands



Source: Spring Framework Docs (2025)

Dapr

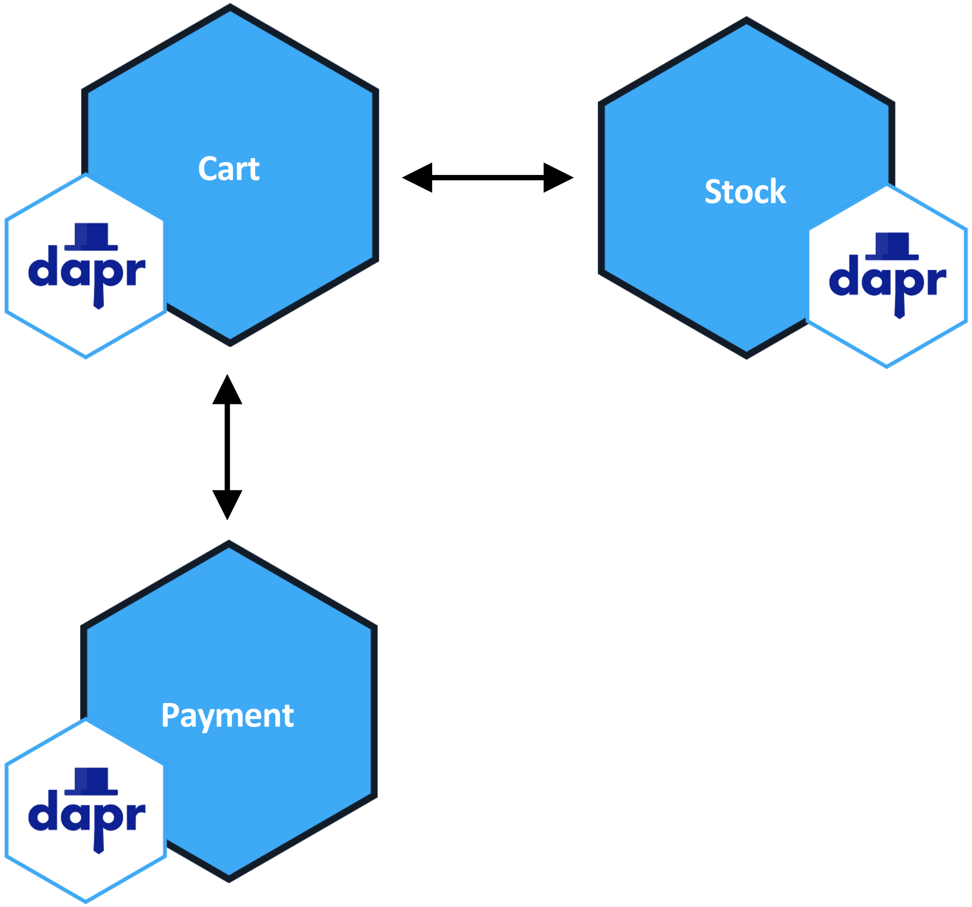
Distributed framework for event-driven applications

Standardized APIs that abstract away technologies

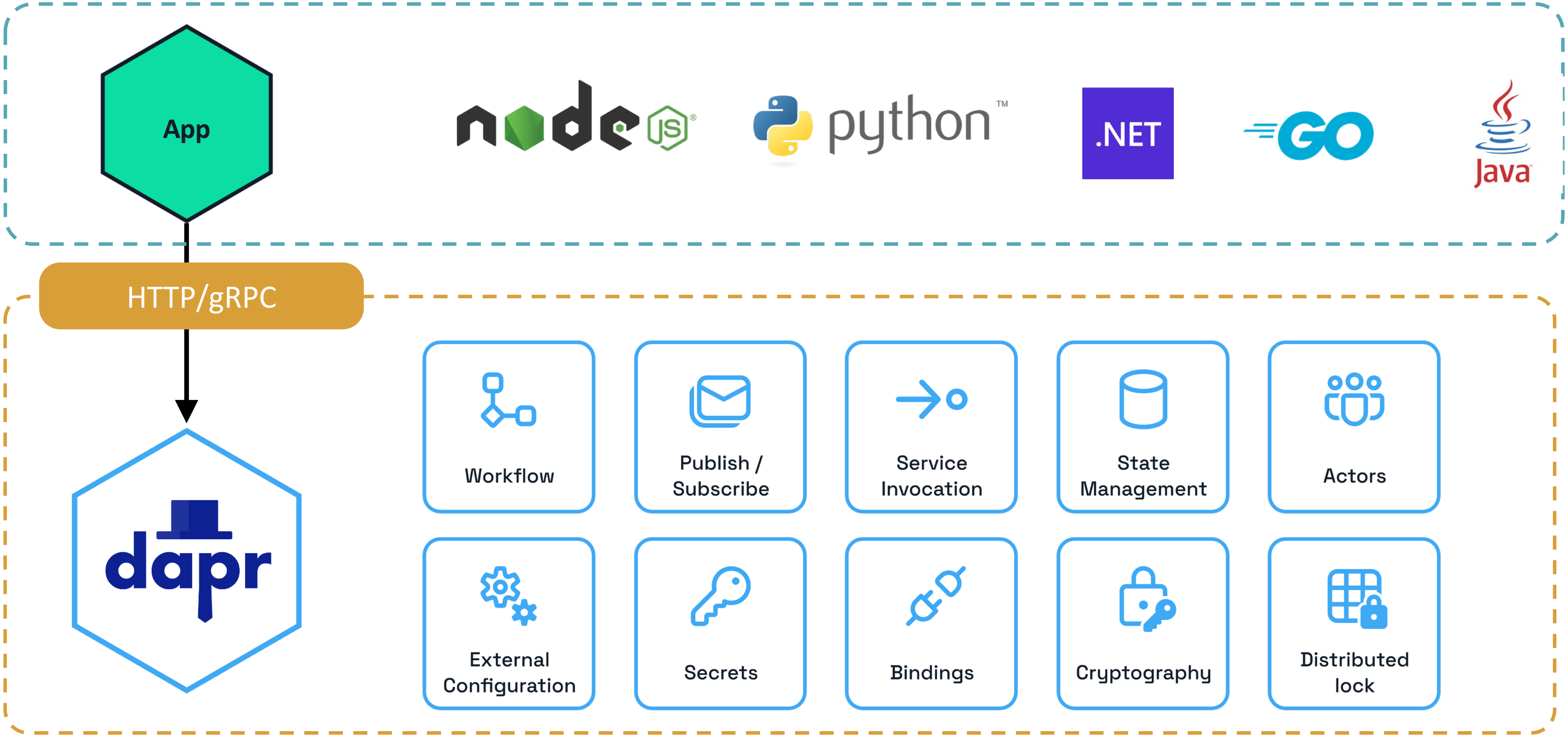
Side-car pattern

Configurable message delivery

Workflow management



A bird's eye view into Dapr



Dapr through a “data management” perspective

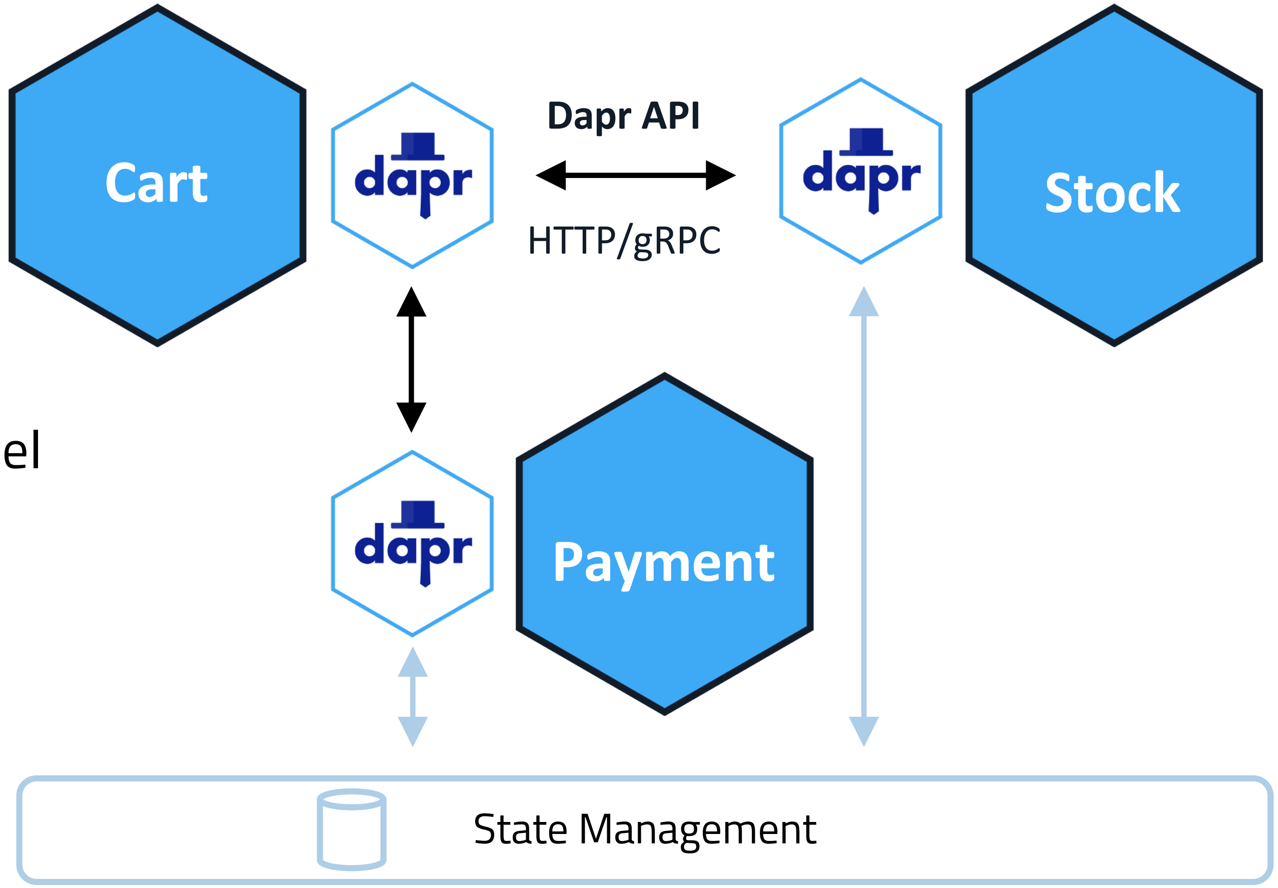
Performance overhead

Centralized state management

No relational model

No strong transactional guarantees

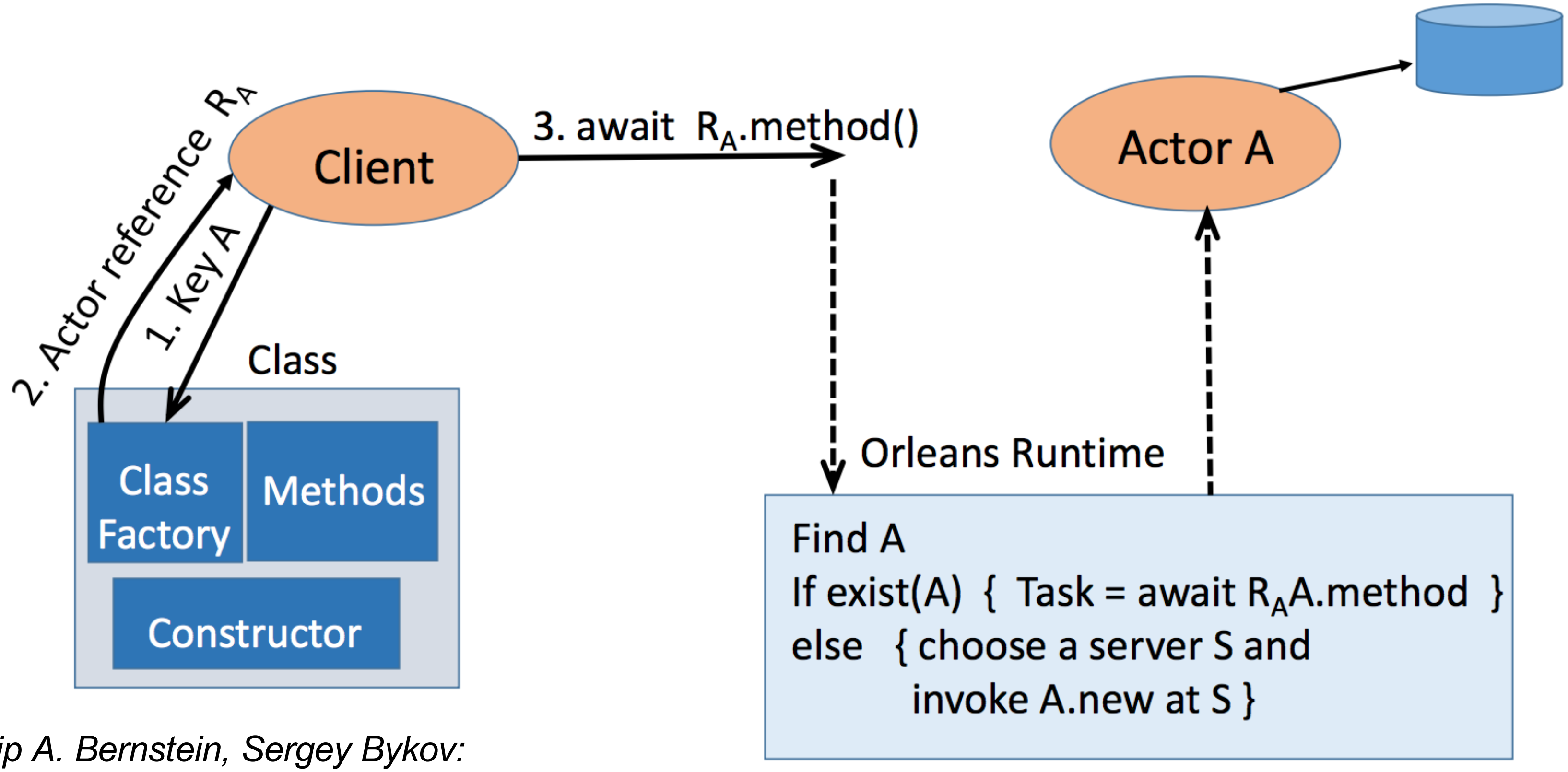
Weak delivery semantics



Actors

Akka, Orleans, etc.

Orleans: Invoking a method on actor A



Source: *Philip A. Bernstein, Sergey Bykov: Developing Cloud Services Using the Orleans Virtual Actor Model. IEEE Internet Comput. 20(5): 71-75 (2016)*

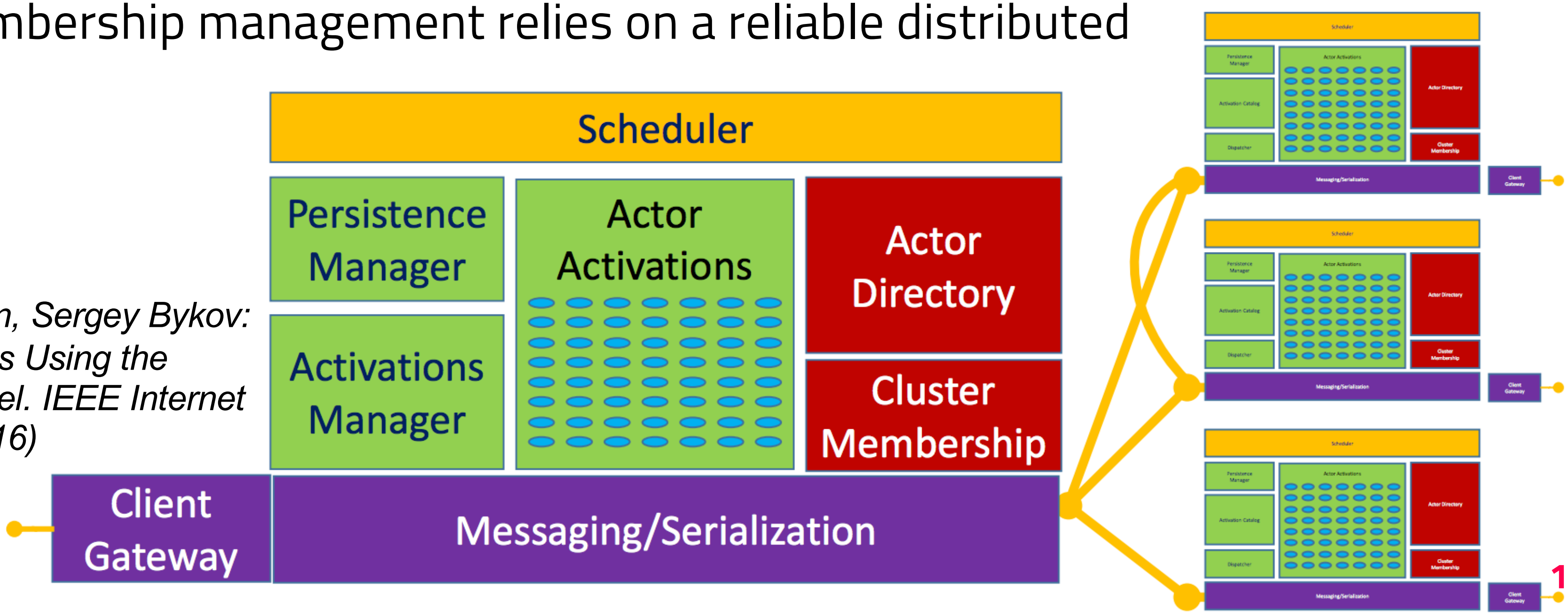
Overview of Orleans Runtime

Messaging multiplexed over a few TCP connections

Actors run on a small number of threads, one per core

Actor Directory is a custom DHT

Cluster Membership management relies on a reliable distributed database

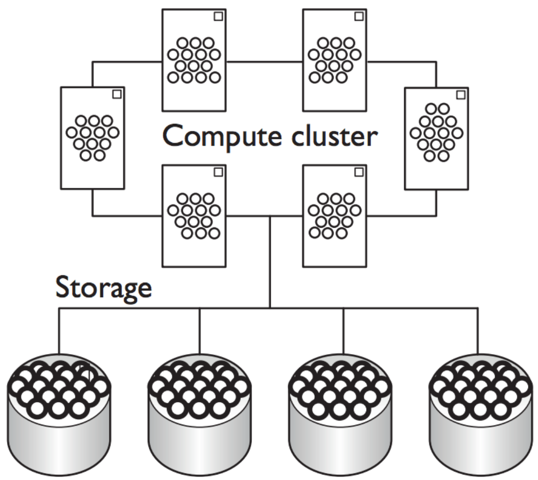


Source: *Philip A. Bernstein, Sergey Bykov: Developing Cloud Services Using the Orleans Virtual Actor Model. IEEE Internet Comput. 20(5): 71-75 (2016)*

Actor State Management

The runtime instantiates an actor and reads the actor's state from storage

```
public CartActor([PersistentState(
    stateName: "cart",
    storageName: Constants.OrleansStorage)] IPersistentState<Cart> state)
{
    this.cart = state;
}
```



Transactions in the Actor Model

Many applications require atomic operations, possibly over multiple actors, e.g.,

- A checkout workflow involving Cart, Stock, Payment and Shipment actors

Transaction management on actor abstraction is challenging.

- Multi-actor transactions are distributed transactions, even if actors are collocated on a single server

Orleans Transaction

- 2PL to achieve concurrent control
- Commit using 2PC
- Early lock release:
  - Releasing locks during phase one of 2PC
  - Reducing excessive latency caused by locking of data on cloud storage
- Deadlock can be avoided by sorting the access order of actors
  - Use timeout if sorting is not possible

Source: *Tamer Eldeeb, Sebastian Burckhardt, Reuben Bond, Asaf Cidon, Junfeng Yang, Philip A. Bernstein: Cloud Actor-Oriented Database Transactions in Orleans. Proc. VLDB Endow. 17(12): 3720-3730 (2024)*

Snapper: A transaction library on actor systems

- Transactions can be executed in two modes
  - Deterministic: PACT (Pre-declared ACtor Transaction)
  - Non-deterministic: ACT (ACtor Transaction)
- PACT: Trans are ordered and executed deterministically in each involved actor
  - App code declares the set of actors involved in a transaction
- ACT: 2PL + 2PC + no wait
- Novel **hybrid** execution mechanism:
  - Snapper allows both PACT and ACT to run simultaneously
  - A global serializability check to ensure correctness

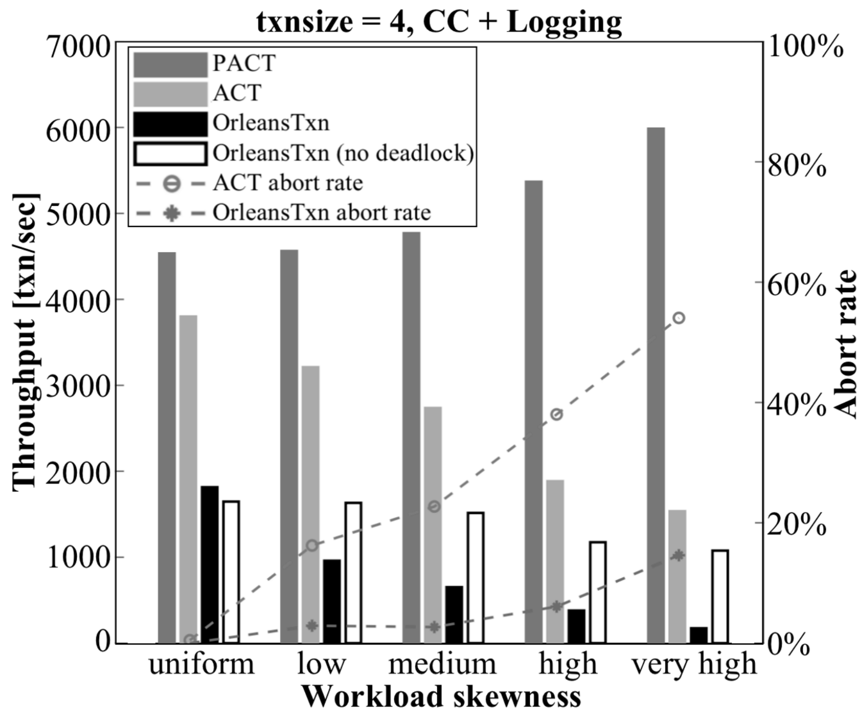
Source: *Yijian Liu, Li Su, Vivek Shah, Yongluan Zhou, Marcos Antonio Vaz Salles: Hybrid Deterministic and Nondeterministic Execution of Transactions in Actor Systems. SIGMOD Conference 2022: 65-78*

Snapper: A transaction library on actor systems

Smallbank Benchmark

Contention ↑

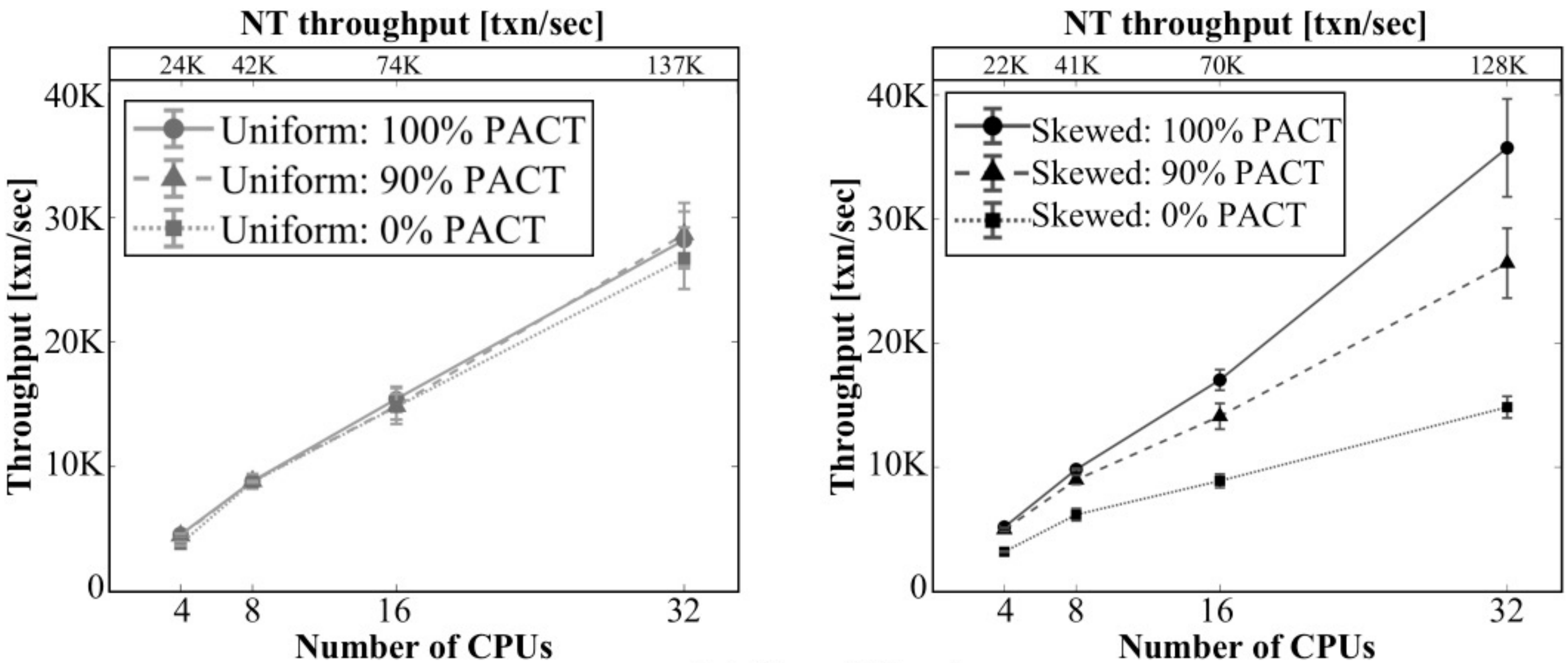
- Orleans Txn ↓
- ACT ↓
- PACT ↑



Source: *Yijian Liu, Li Su, Vivek Shah, Yongluan Zhou, Marcos Antonio Vaz Salles: Hybrid Deterministic and Nondeterministic Execution of Transactions in Actor Systems. SIGMOD Conference 2022: 65-78*

Snapper: A transaction library on actor systems

Hybrid execution:

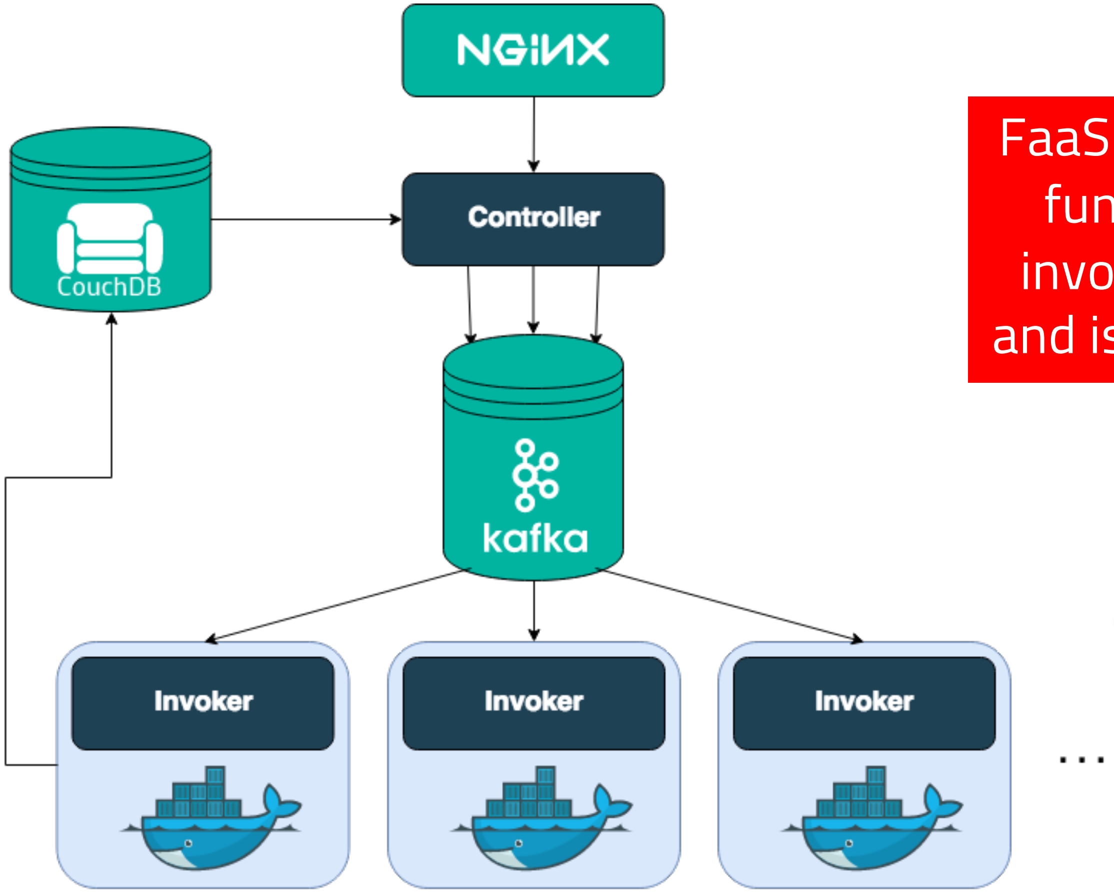


Source: *Yijian Liu, Li Su, Vivek Shah, Yongluan Zhou, Marcos Antonio Vaz Salles: Hybrid Deterministic and Nondeterministic Execution of Transactions in Actor Systems. SIGMOD Conference 2022: 65-78*

Serverless Functions

Lambda/Azure Functions, Durable Functions, Amazon Steps

Function as a Service – Typical System



FaaS model: function invocation and isolation

Functions executed via spawning new Docker container

Apache OpenWhisk (2020)

Serverless Function as a Service

Transparent application management

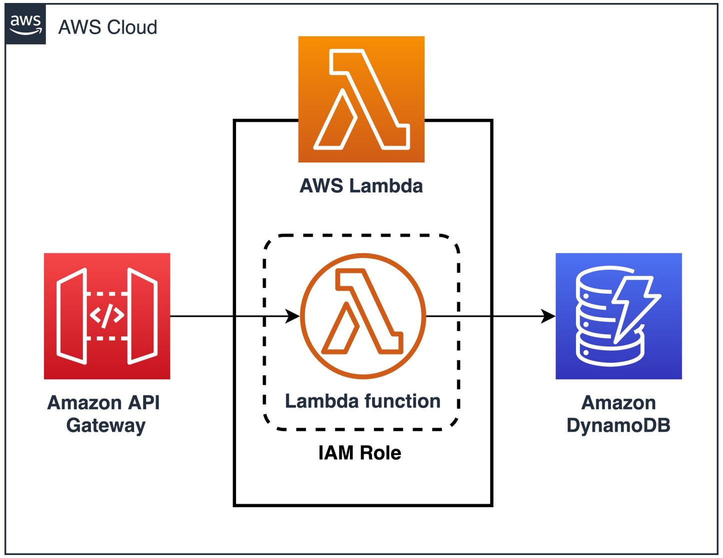
Users rely that application functions are:

- persistent (no submission of the function for every request)
- scheduled (in a timely and fair manner)
- instantiated (provision of resources and execution)
- scaled (according to application workload)
- persisted (results are stored durably)

Serverless Function as a Service - History

Lambda paradigm (e.g., AWS Lambda)

- Small to moderate I/O (storage and network) workloads
- Lack of execution guarantees intra and inter functions
- Costly shared and mutable state

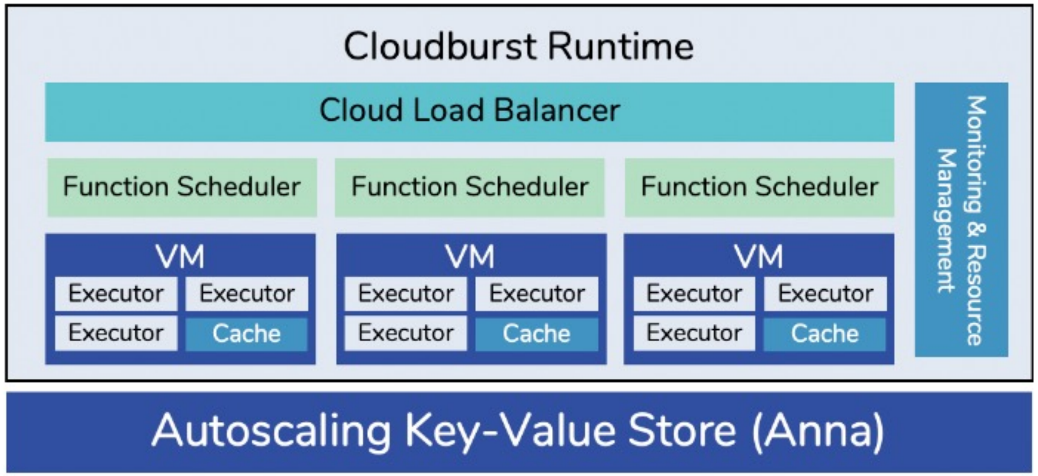


Moellering & Grunwald in AWS Architecture Blog (2020)

Serverless Function as a Service - History

Stateful FaaS (SFaaS) (1st gen)

- Specific-purpose
- autoscaling storage
- Richer data consistency
- semantics
- Weak guarantees on
- multi-function workflows



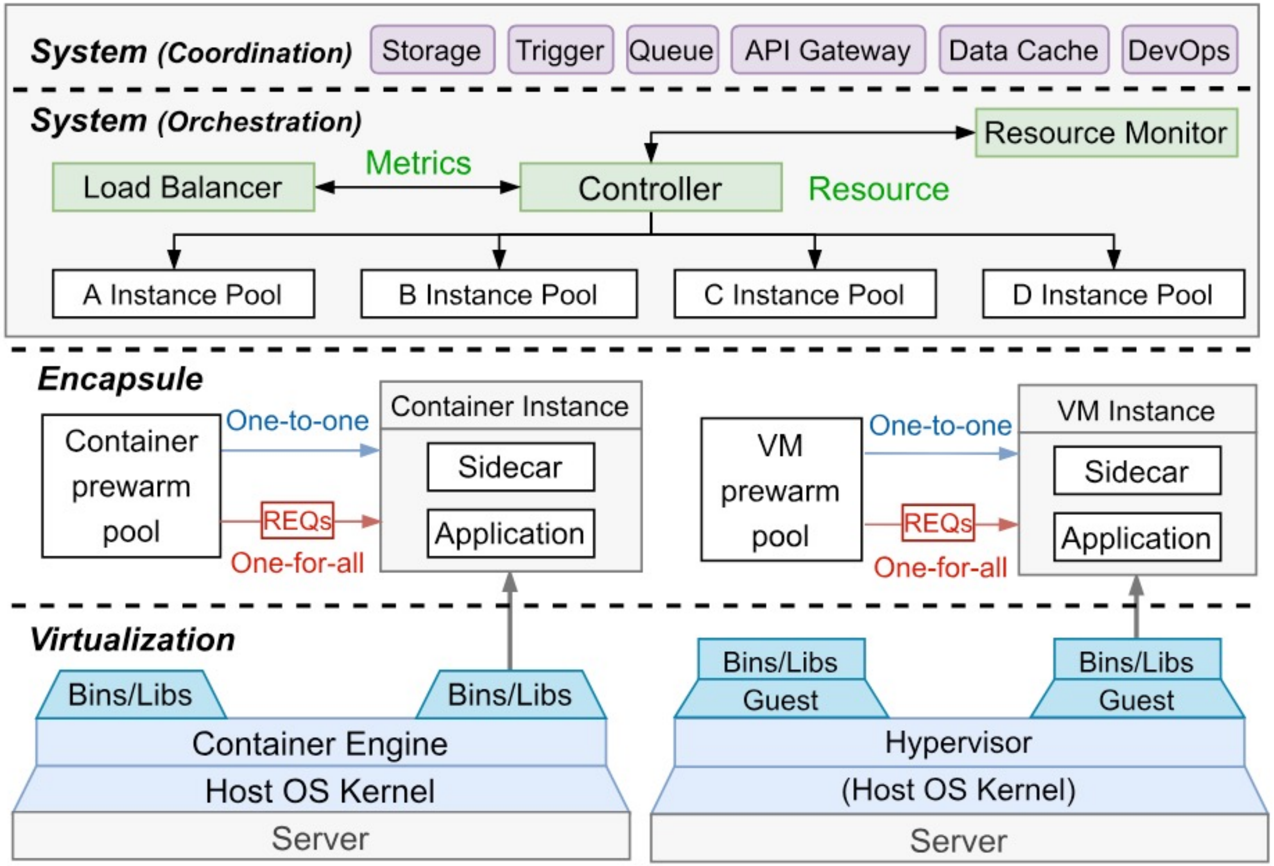
Sreekanti et al. "Cloudburst: Stateful Functions-as-a-Service." VLDB (2020)

Serverless Function as a Service - History

SFaaS (2nd generation)

- Actor-oriented programming model
- Stronger execution guarantees

Serverless Function as a Service – Typical System



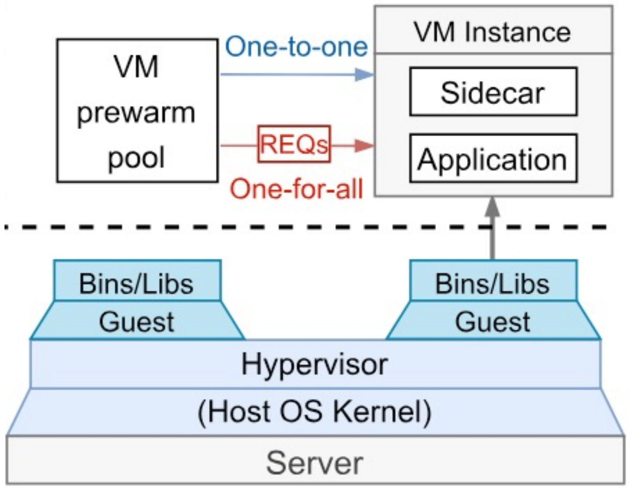
Z. Li et al. The Serverless Computing Survey: A Technical Primer for Design Architecture. ACM Comput. Surv. (2022)

Serverless Function as a Service

Function execution is multiplexed in a virtual layer

Virtualized sandboxes

1. Function is registered
2. Function is invoked
3. Upstream layers process invocation
4. Virtualized sandbox is loaded/reused
5. Sandbox executes function in isolation



Z. Li et al. The Serverless Computing Survey: A Technical Primer for Design Architecture. ACM Comput. Surv. (2022)

Serverless Function as a Service

Performance techniques

- Reuse a warm sandbox
- Pre-create a sandbox
- predicting load increase
- Distribute and batch functions
- according to previous runs

Avoid cold start

How long to keep?

Overprovision

Balance load

Increase latency

Z. Li et al. The Serverless Computing Survey: A Technical Primer for Design Architecture. ACM Comput. Surv. (2022)

Serverless Function as a Service

- FaaS is a model for executing functions
- FaaS system abstracts operational concerns to enable FaaS model
  - Resource and failure transparency to developer
  - Focus on writing application functions
- Complex, data-intensive applications may not fit well the paradigm (yet)
- Application architecture is always a trade-off

Dataflow systems

Styx


STREAMING...
STREAMING SYSTEMS EVERYWHERE!
imgflip.com

https://github.com/delftdata/styx

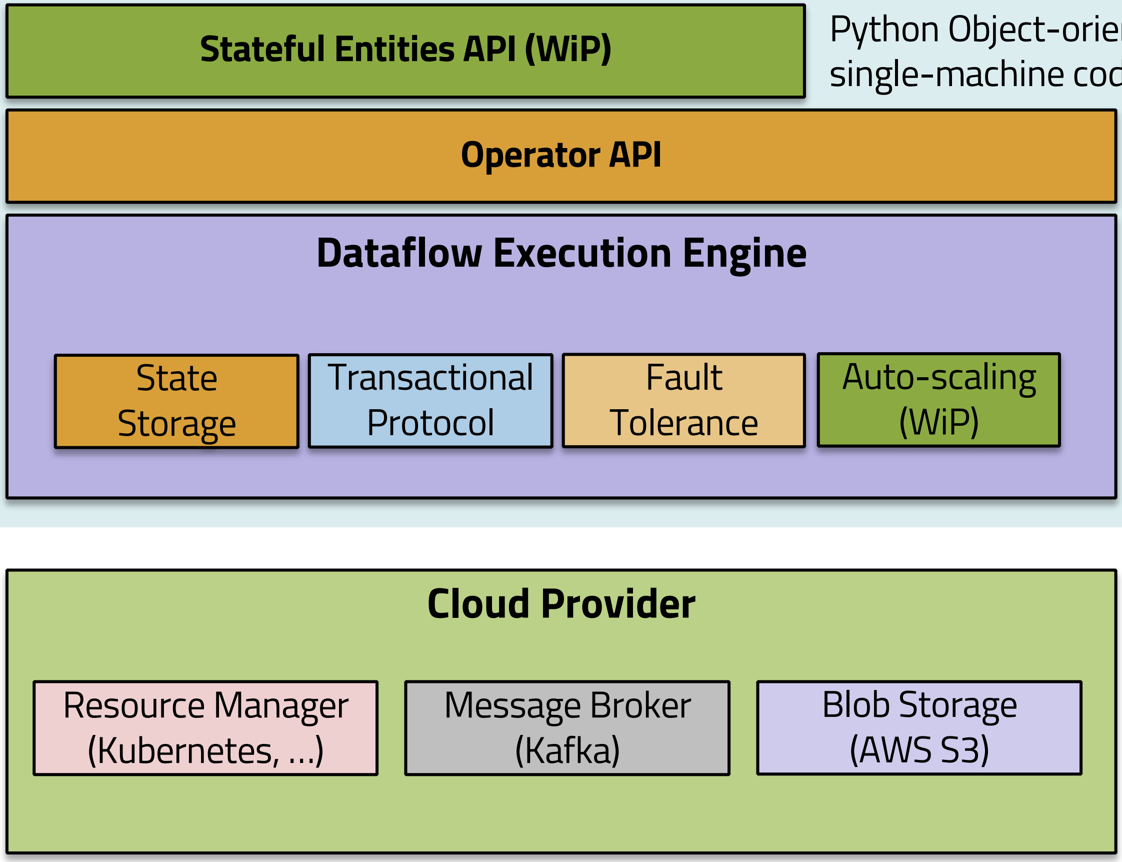
Styx: a dataflow-based transactional runtime for Cloud Applications

Design Choices

Object-oriented API	"Keep the data moving" [1]	Scalability
Dataflows are awkward. Styx offers Durable Entities [4], i.e., Python objects with arbitrary function-to-function calls.	No time for wait for 2PCs. Styx extends deterministic database concepts [2] for arbitrary function-to-function calls.	Partitioned state, collocated with function execution. Parallel execution at Entity-level granularity.
Coarse-grained Fault Tolerance	**Exactly-once output**	**Consistency**
Async checkpoints á lá Chandy-Lamport [3,4].	Uses durable queues (Kafka) for input transactions, and deduplicating outputs.	Serializable state mutations made by arbitrary function-to-function calls.

[1] Stonebraker, Çetintemel, Zdonik. **"The 8 requirements of real-time stream processing."** [*Sig. Record* 2005]
[2] Yi, Yu, Cao, and Samuel Madden. "**Aria: a fast and practical deterministic OLTP database**." [VLDB 2020]
[3] Carbone, et.al. "**State management in Apache Flink®: consistent stateful distributed stream processing**." [VLDB 2018]
[4] Silvestre, Fragkoulis, Katsifodimos. **"Clonos: Consistent causal recovery for highly-available streaming dataflows."** [SIGMOD 21]
[5] Psarakis, Zorgdrager, Fragkoulis, Salvanesci, Katsifodimos **"Stateful Entities: Object-oriented Cloud Applications as Distributed Dataflows"**, [CIDR '23, EDBT '24]
[6] Psarakis, Christodoulou, Siachamis, Fragkoulis, Katsifodimos **"Styx: Transactional Stateful Functions on Streaming Dataflows"**, [SIGMOD'25]

Enter Styx: an “ideal” Transactional Cloud Application Runtime



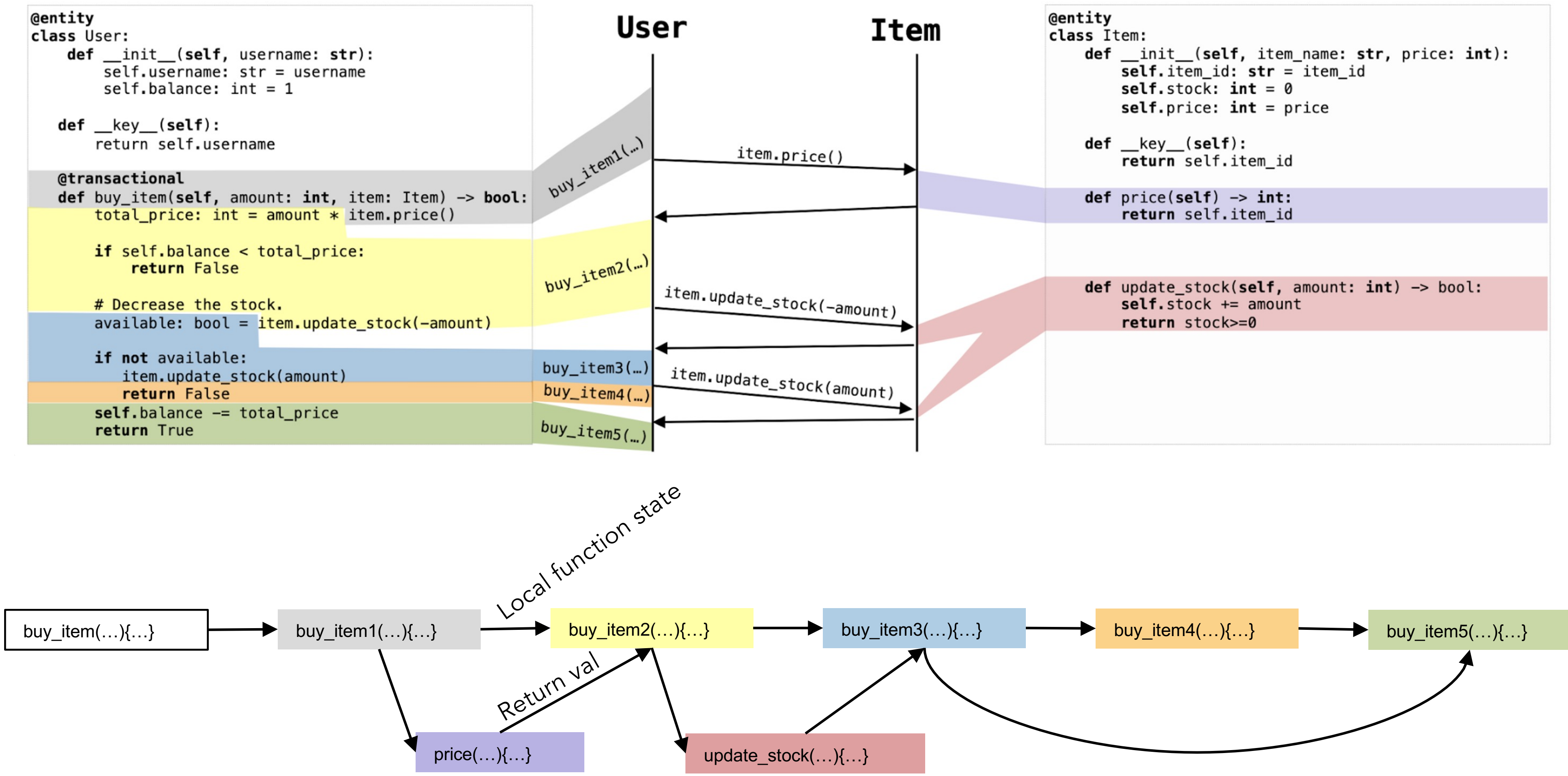
Python Object-oriented API. App code resembles single-machine code (with some conventions).

Python–based, access to lower-level operator primitives for advanced programmers.

Transactional serializability, exactly-once output, early commit-replies, low-latency processing

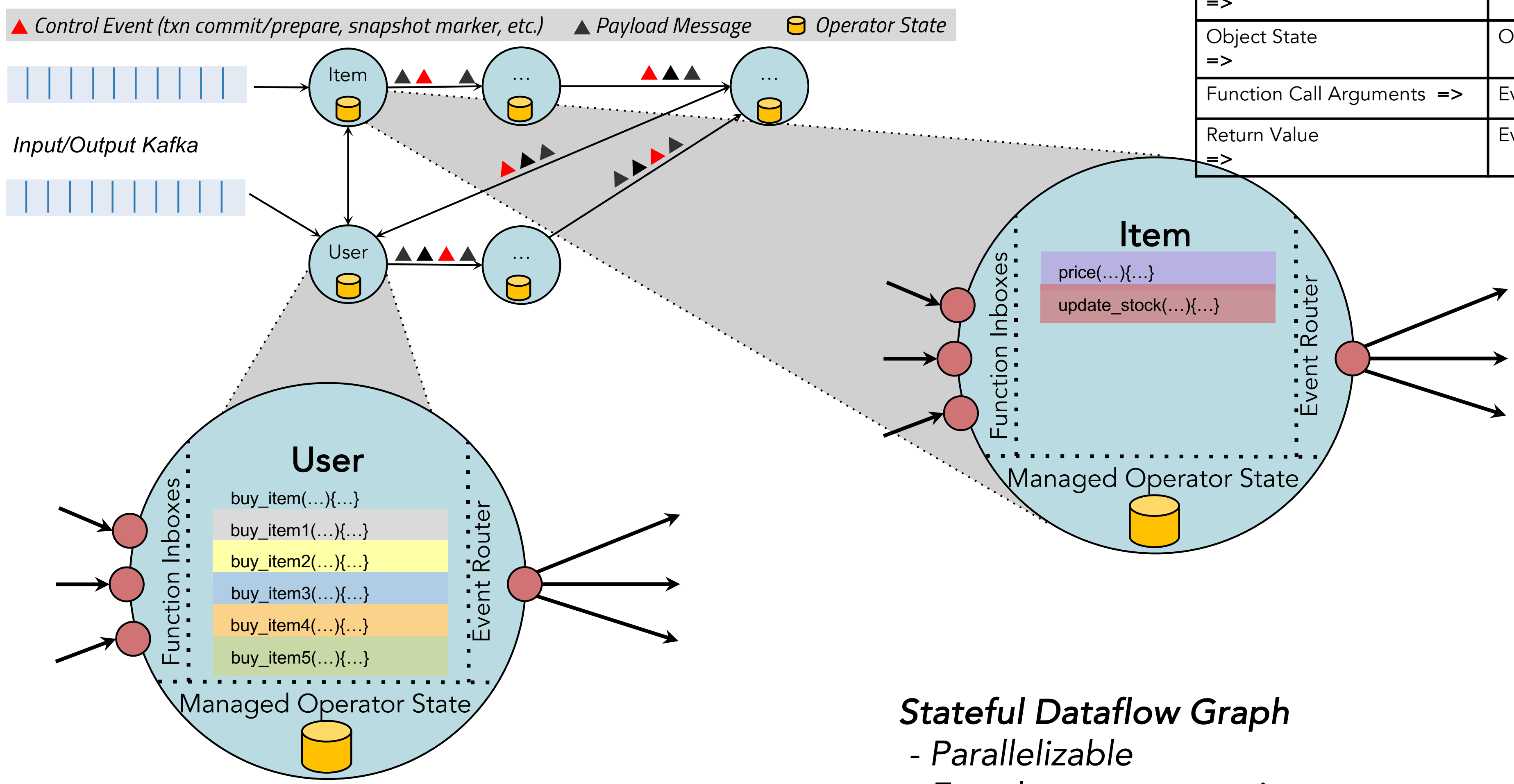
https://github.com/delftdata/styx

Stateful Entities (WiP): Object-oriented Style Programming



Psarakis, Zorgdrager, Fragkoulis, Salvanesci, Katsifodimos **"Stateful Entities: Object-oriented Cloud Applications as Distributed Dataflows"**, CIDR '23, EDBT '24]

Deployment on Styx



Python	Dataflow
Class =>	Operator
Object State =>	Operator State
Function Call Arguments =>	Event
Return Value =>	Event

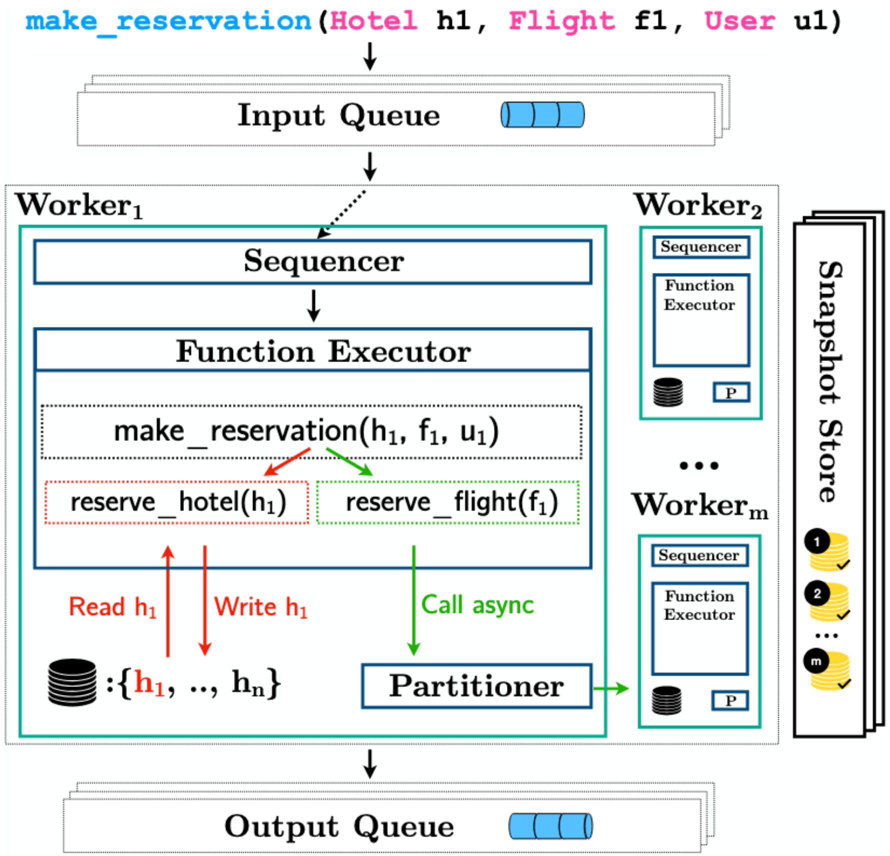
Stateful Dataflow Graph

- *Parallelizable*
- *Exactly-once processing guarantees*
- *Transactional serializability*
- *High-throughput, low-latency*

Event-driven programming in Styx (Operator API)

```python
@cart.register
def checkout(ctx: StatefulFunction):
    items, user_id, total_price, paid = ctx.get()
    for item_id, qty in items:
        ctx.call_async(operator=stock,
                       function_name='decrement_stock',
                       key=item_id,
                       params=(qty, ))
    ctx.call_async(operator=payment,
                   function_name='pay',
                   key=user_id,
                   params=(total_price, ))
    paid = True
    ctx.put((items, user_id, total_price, paid))
    return "Checkout Successful"
```

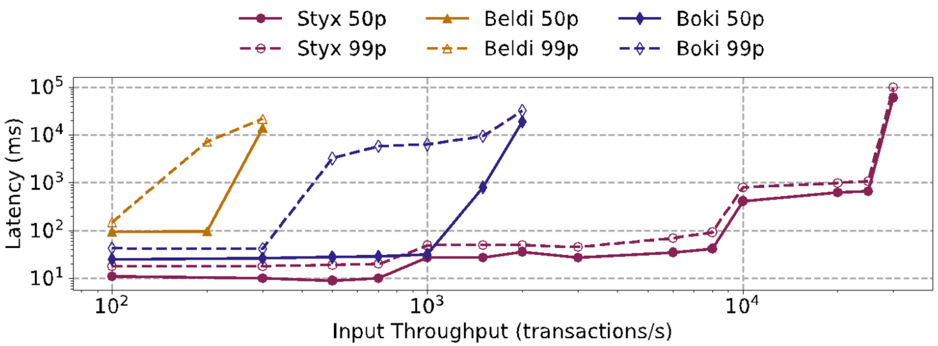
How does the architecture look like?



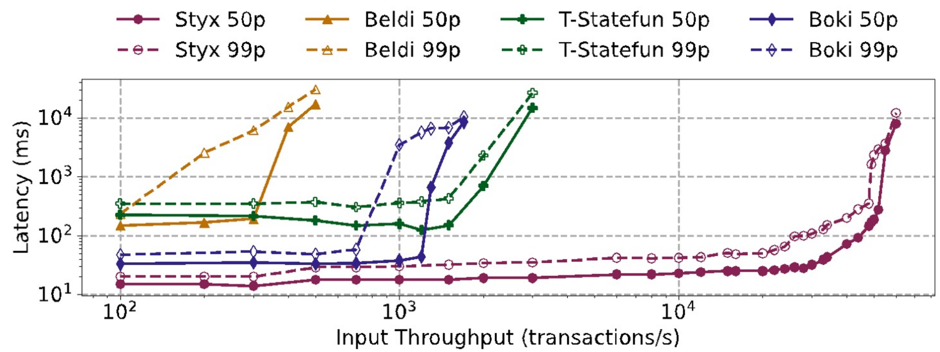
Styx is epoch-based

How does it perform?

DeathStar Throughput vs. Latency



YCSB Throughput vs. Latency

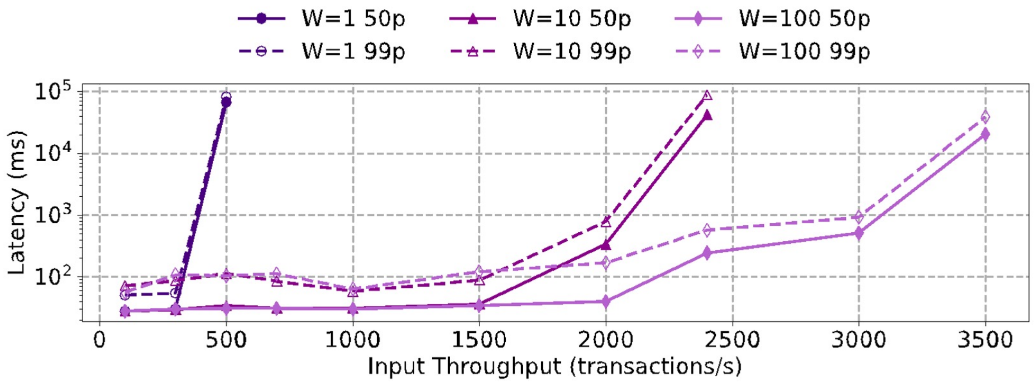


100 workers/CPUs

How does it perform?

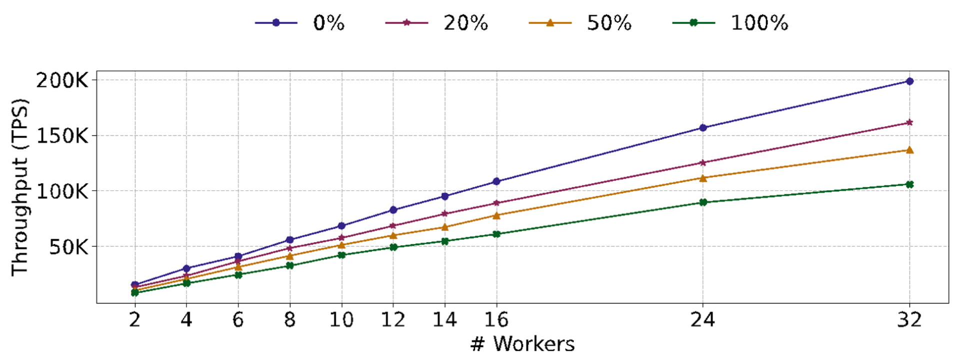
TPC-C Throughput vs. Latency

(W = Num. of Warehouses)



YCSB Scalability

(% of multipartition transactions)



100 workers/CPUs

Summary

Existing FaaS avoid important problems

State, messaging & transactions need a holistic solution

It is possible to build "ideal" runtimes