SJTU SAIL (Software Architecture and Infrastructure Lab)

SHANGHAI JIAO TONG UNIVERSITY

The State of Serverless Art

Yalun Lin Hsu

2020-11-15

SAIL

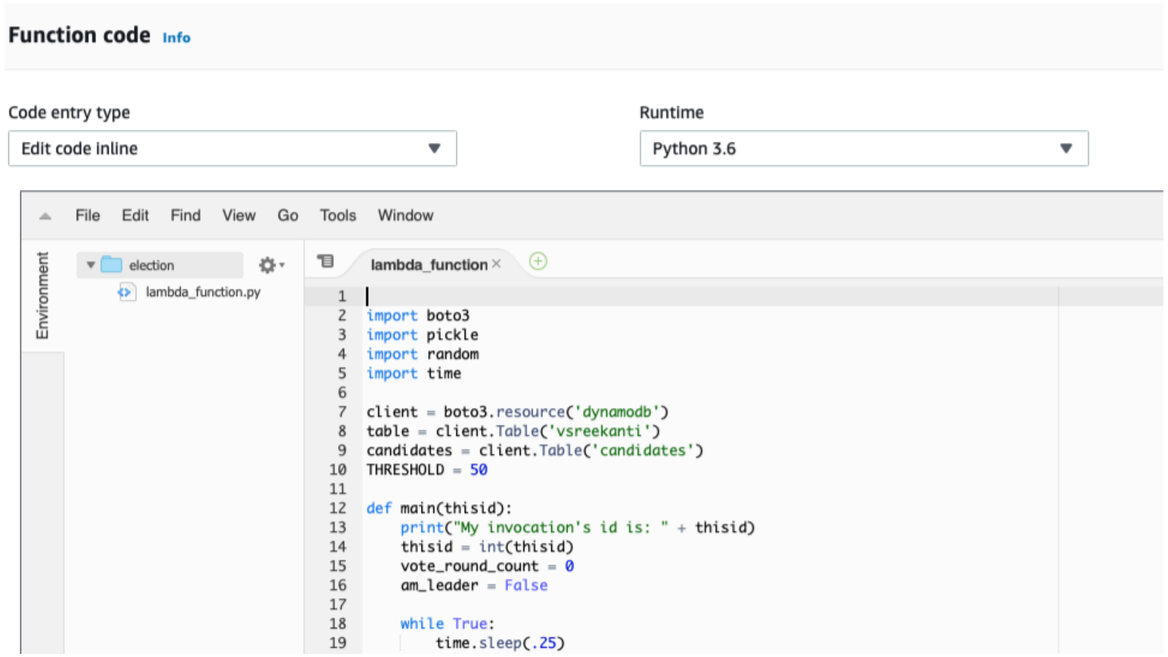
Background

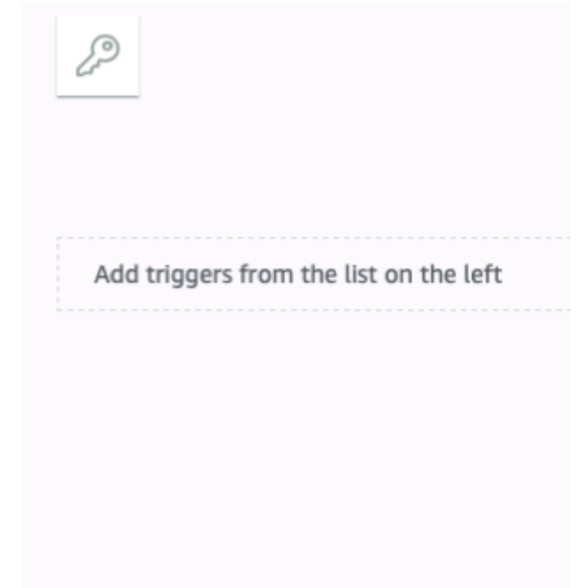
What is Serverless Computing?

Serverless computing is a programming abstraction that enables users to upload programs, run them at any scale, and pay only for resources used.

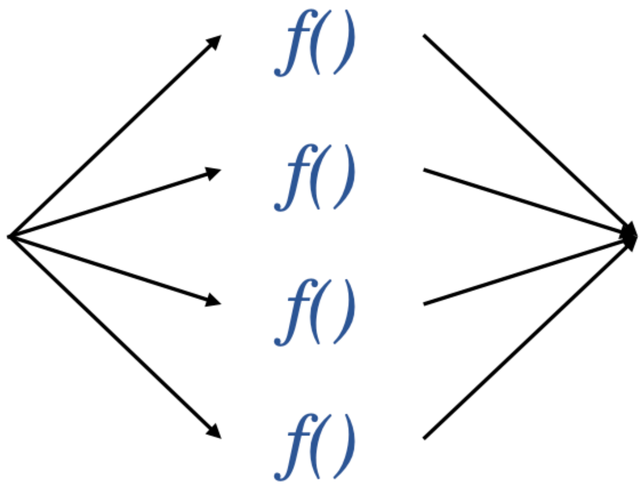
Functions-as-a-Service (FaaS)

- AWS Lambda, Google Cloud Functions, OpenWhisk (IBM), Azure Functions, OpenLambda, OpenFaaS, kNative...
- Optimized for simplicity – register functions, enable triggers, and scale transparently

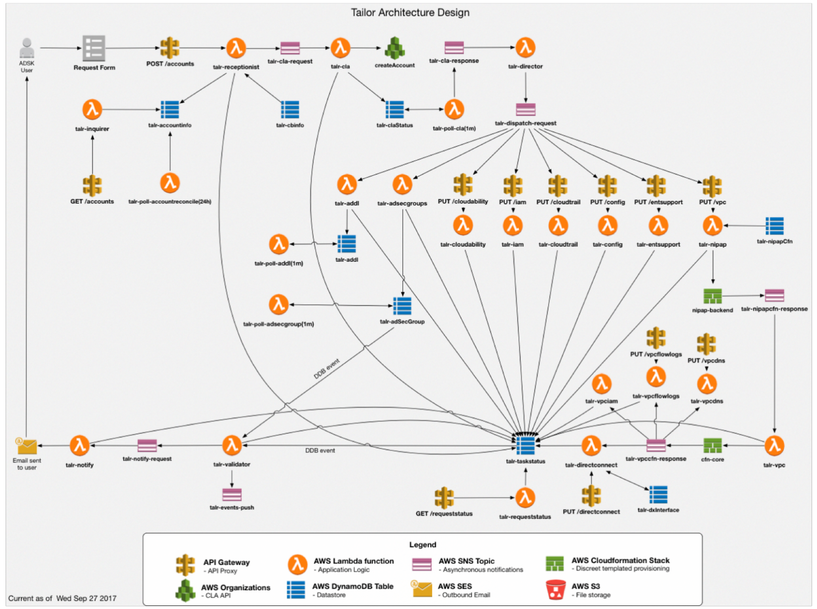




What is Serverless Good at Today?



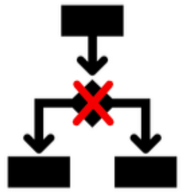
Embarrassingly parallel tasks



Workflow orchestration

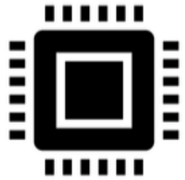
Limitations on FaaS Today

Limited execution lifetimes



No inbound network connections

IO is a bottleneck



No specialized hardware

But that's NOT okay: Everything is NOT functional!

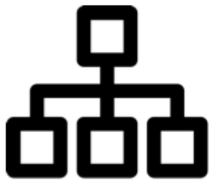
- Functional programs don't have side effects or mutable state!

- And it *is called* AWS Lambda

Dysfunction-as-a-Service

- FaaS is not designed for functional programming because *real applications share state*

$$f(g(x))$$

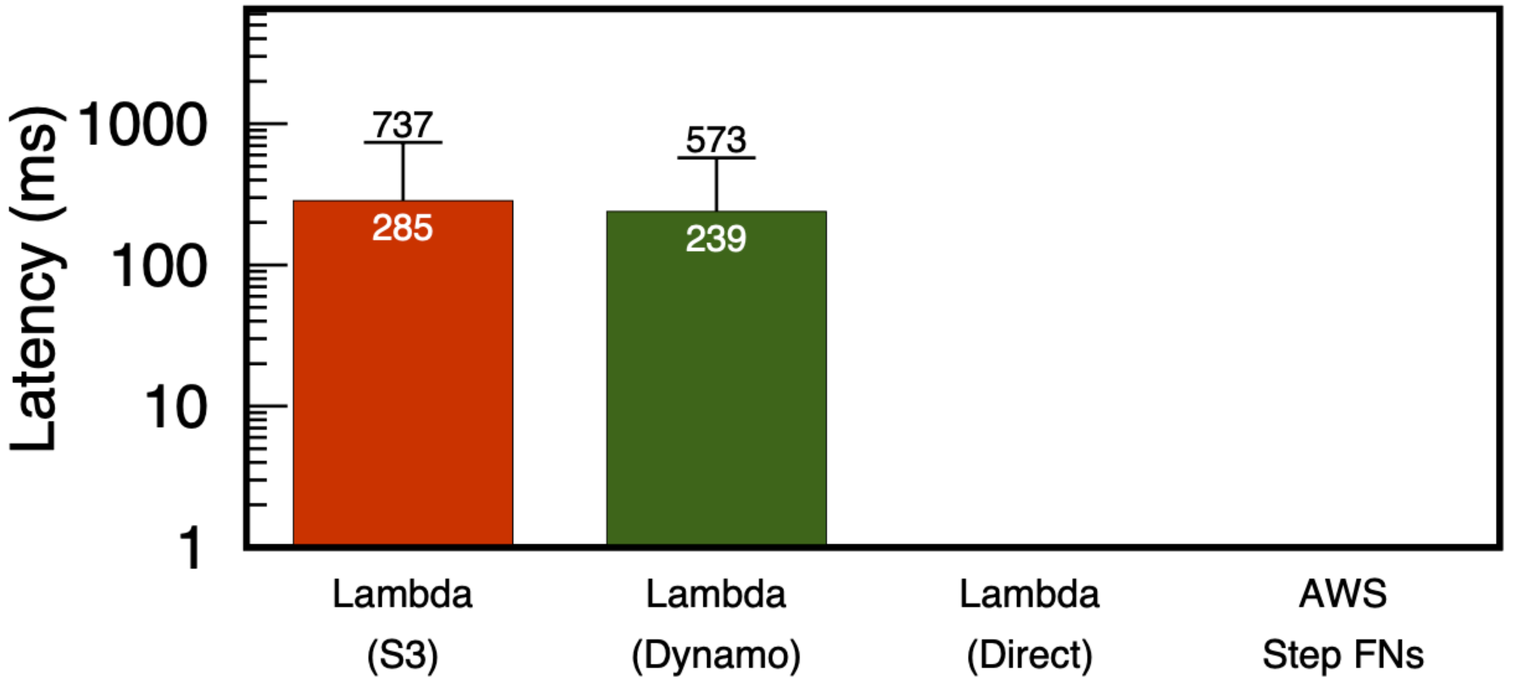


- FaaS is poorly suited for all of these

Quantifying The Pain of FaaS

Even Functional Programming is Slow!

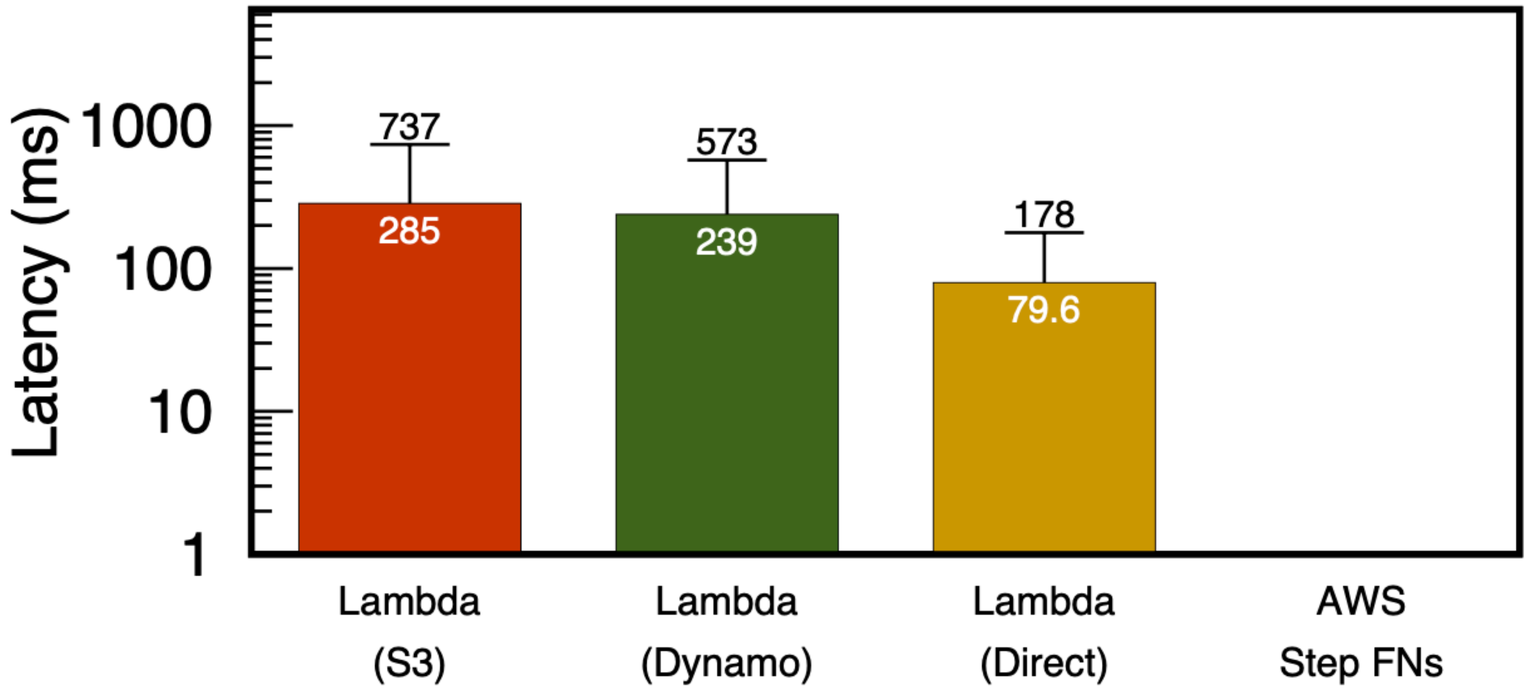
$$f(g(x))$$



Median and 99th percentile latencies for composing two arithmetic functions on AWS Lambda.

Even Functional Programming is Slow!

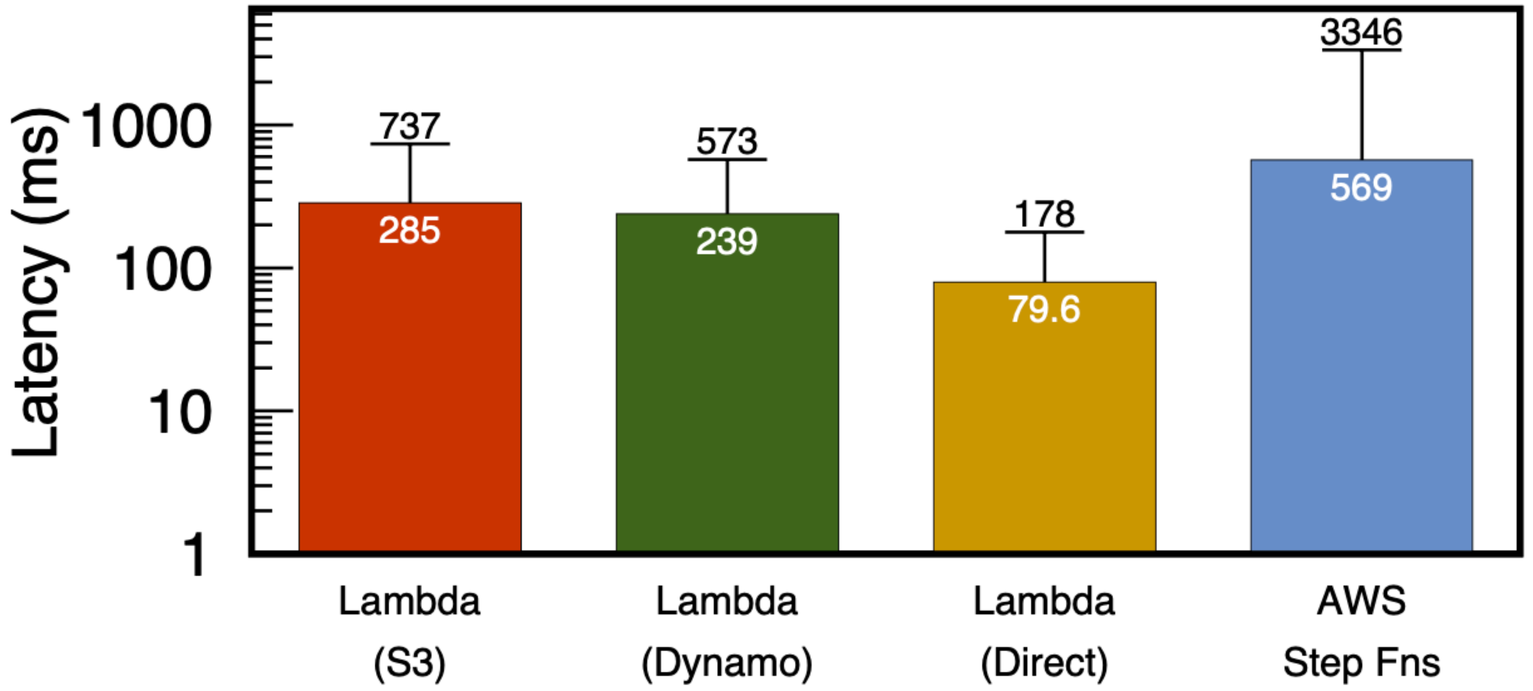
$$f(g(x))$$



Median and 99th percentile latencies for composing two arithmetic functions on AWS Lambda.

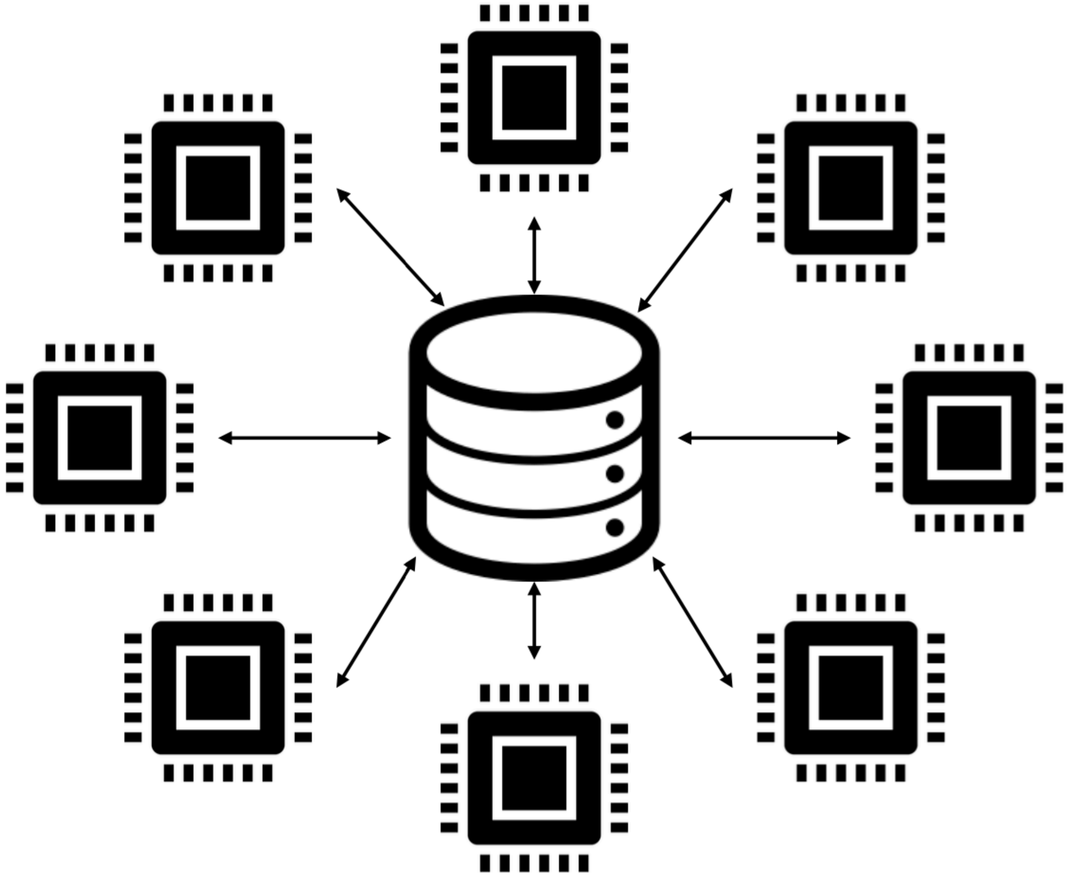
Even Functional Programming is Slow!

$$f(g(x))$$



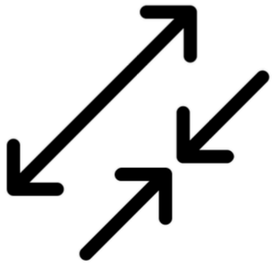
Median and 99th percentile latencies for composing two arithmetic functions on AWS Lambda.

Shared Mutable State



Shared Mutable State

Serverless Storage

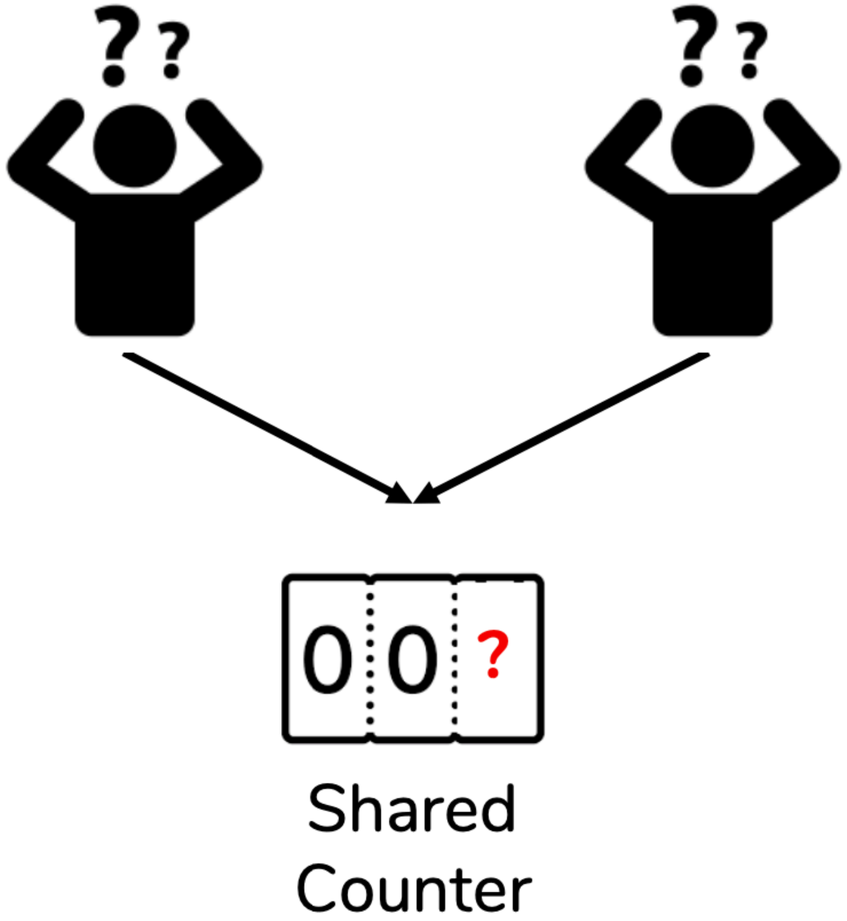


Autoscaling

TRADEOFF!

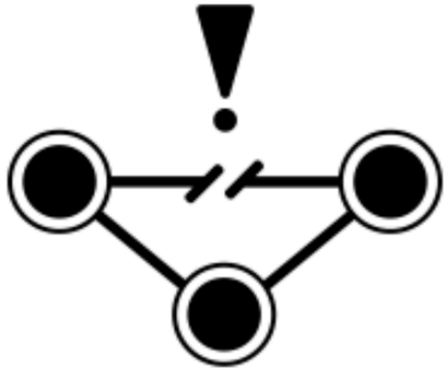
Low Latency

(In)Consistency Guarantees



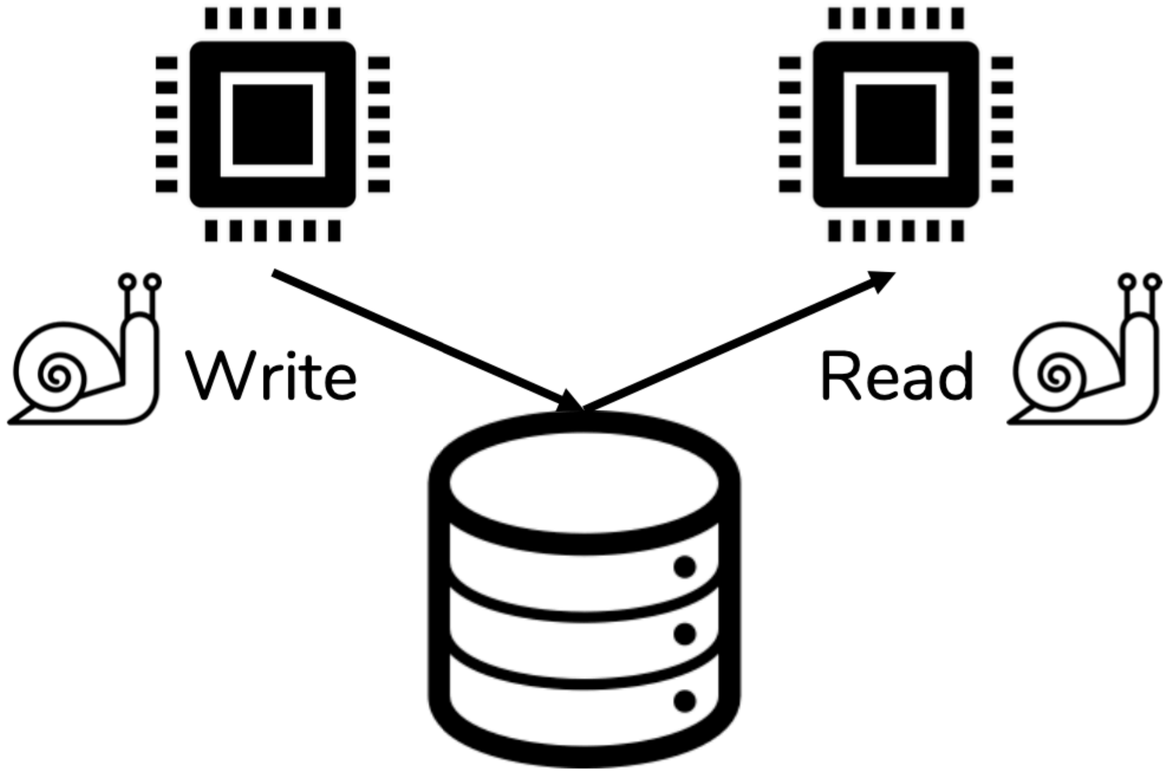
No Inbound Network Connections

Enables Process Migration



Easy Fault Tolerance

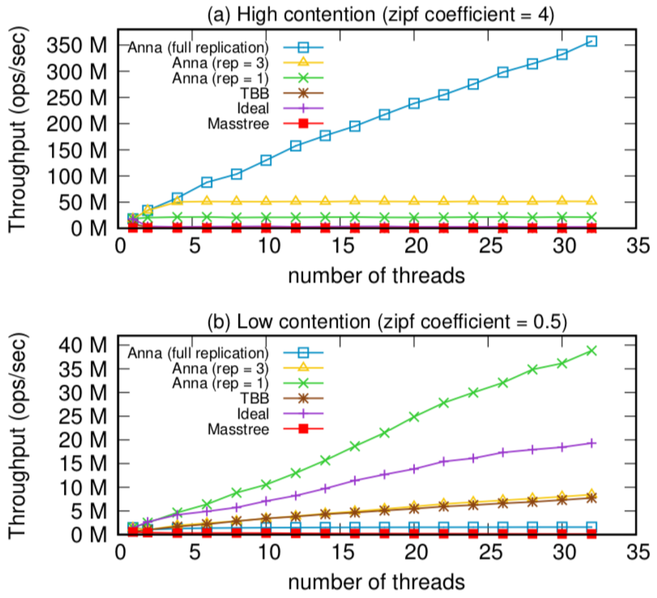
Indirect Communication

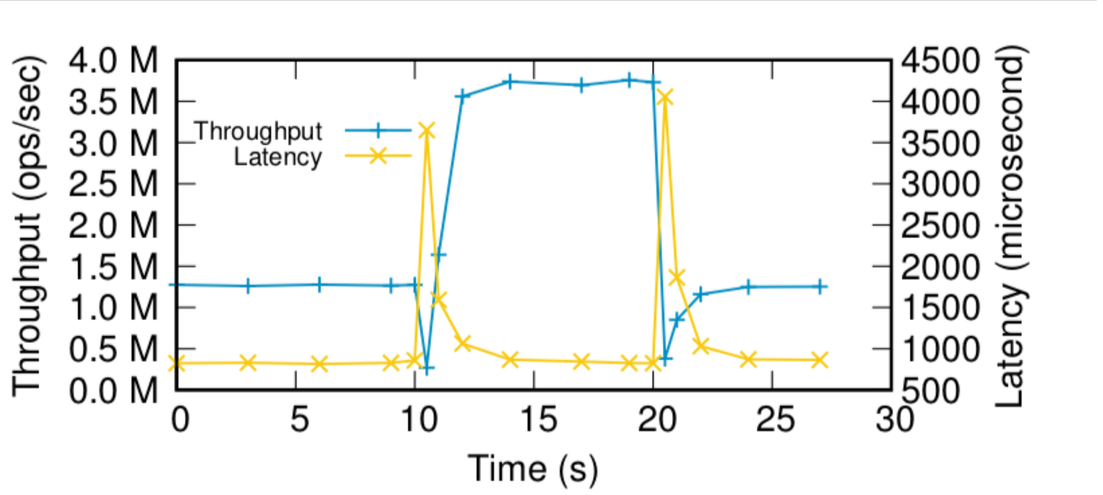


Cloudburst: Stateful FaaS

Background: Anna Overview

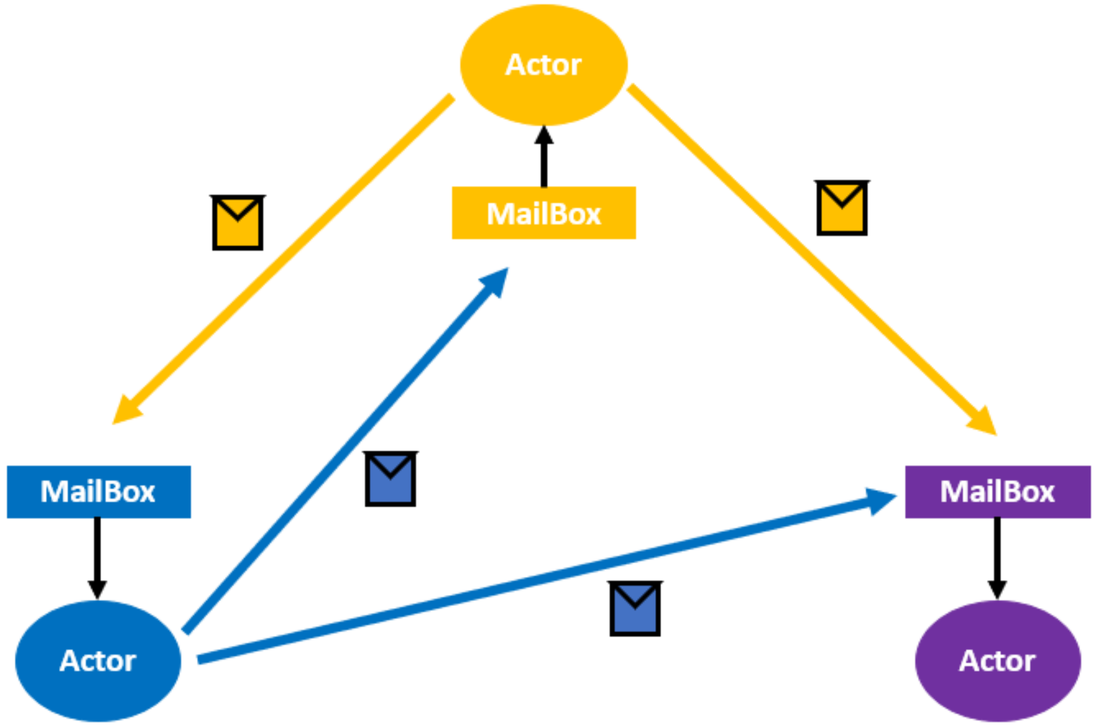
- High performance across orders of magnitude in scale
  - ✓ 10x faster than Redis/Cassandra in a geo-distributed deployment
- Autoscaling & cost-efficient
  - ✓ 500x faster than Amazon DynamoDB for the same cost





Coordination-free consistency. No atomics, no locks, no waiting ever!

Background: Actor Model



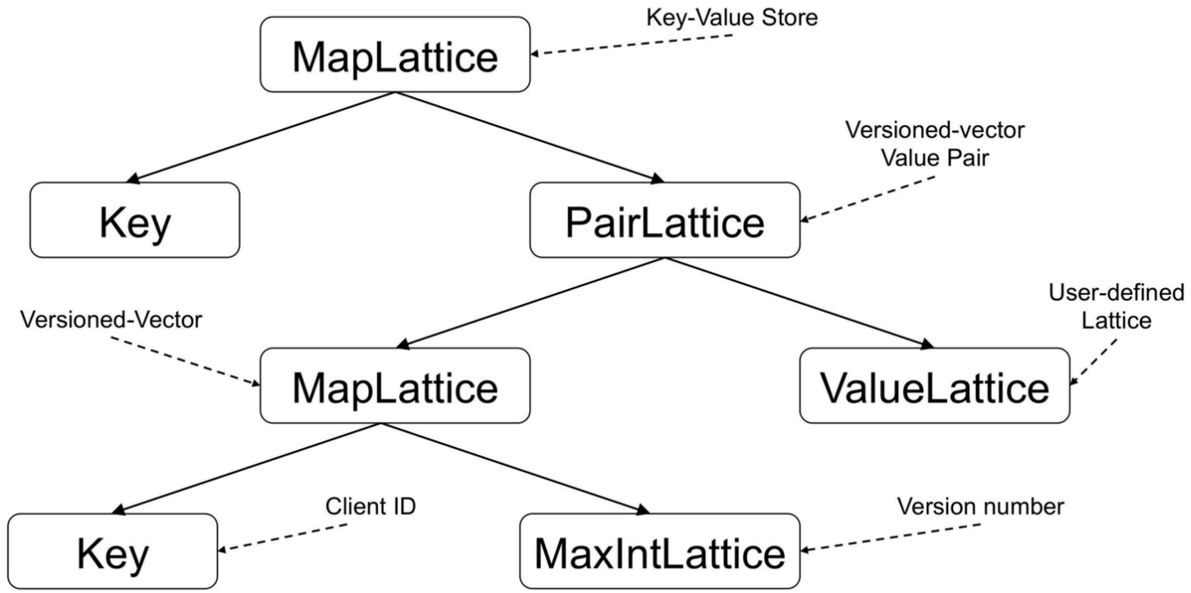
Background: Lattice

✓ Associativity: (a+b)+c = a+(b+c)

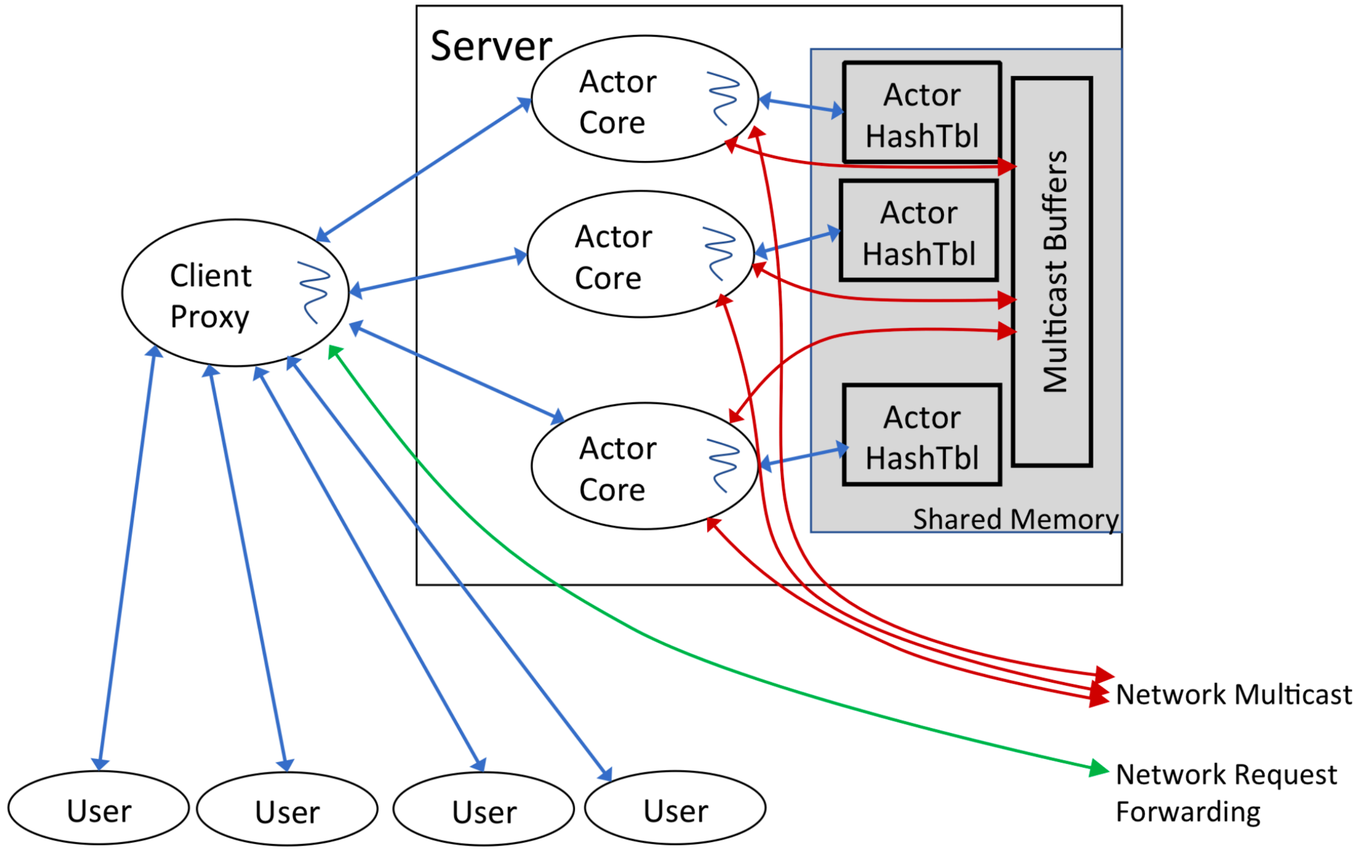
✓ Commutativity: a+b = b+a

✓ Idempotence: a+a = a

Background: Causal Lattice

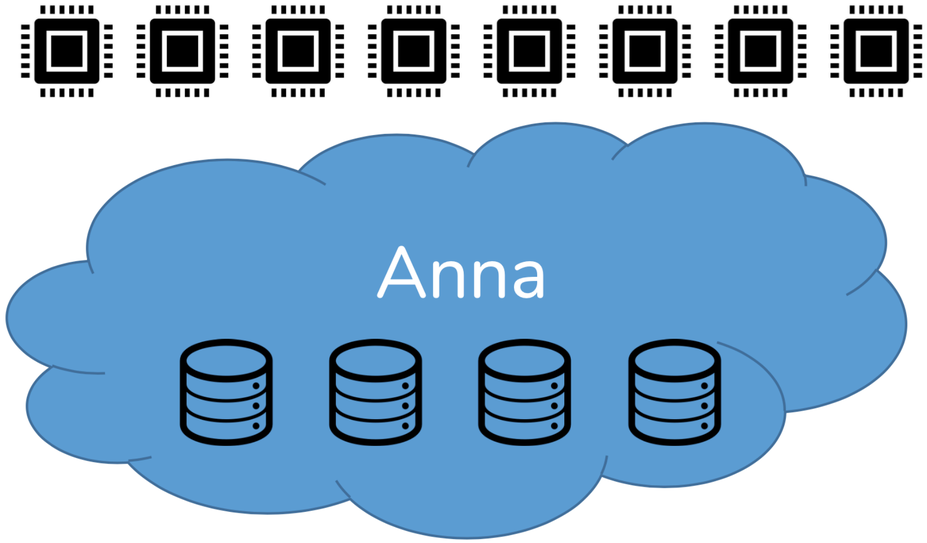


Background: Anna Architecture

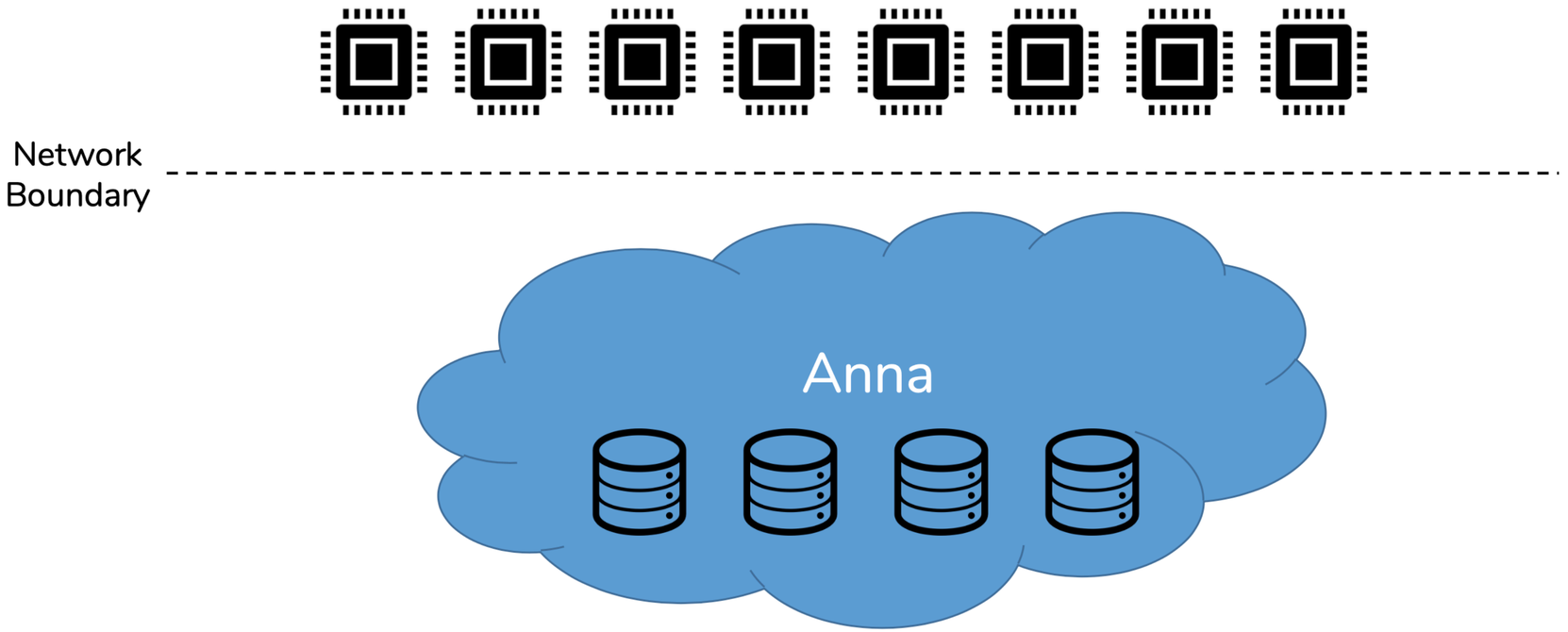


Cloudburst: FaaS over Anna

- Maintain disaggregation of compute & state
- Make serverless a viable option for stateful applications
- Use Anna for both storage and communication



Cloudburst: FaaS over Anna

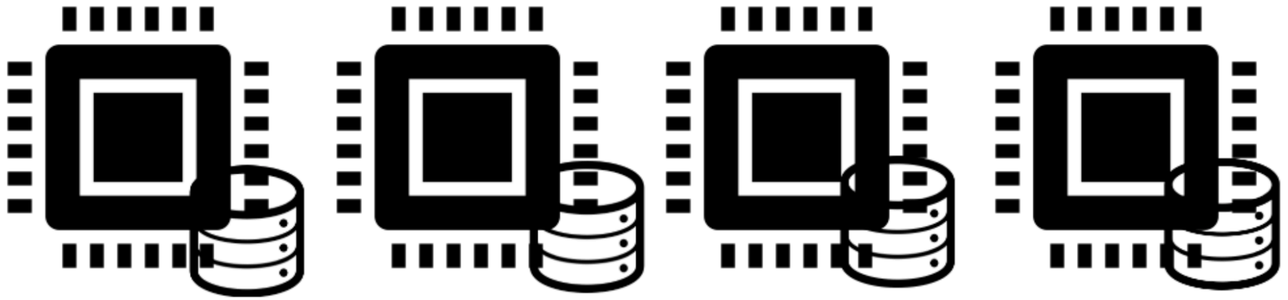


SOLUTION: Logical disaggregation with physical colocation

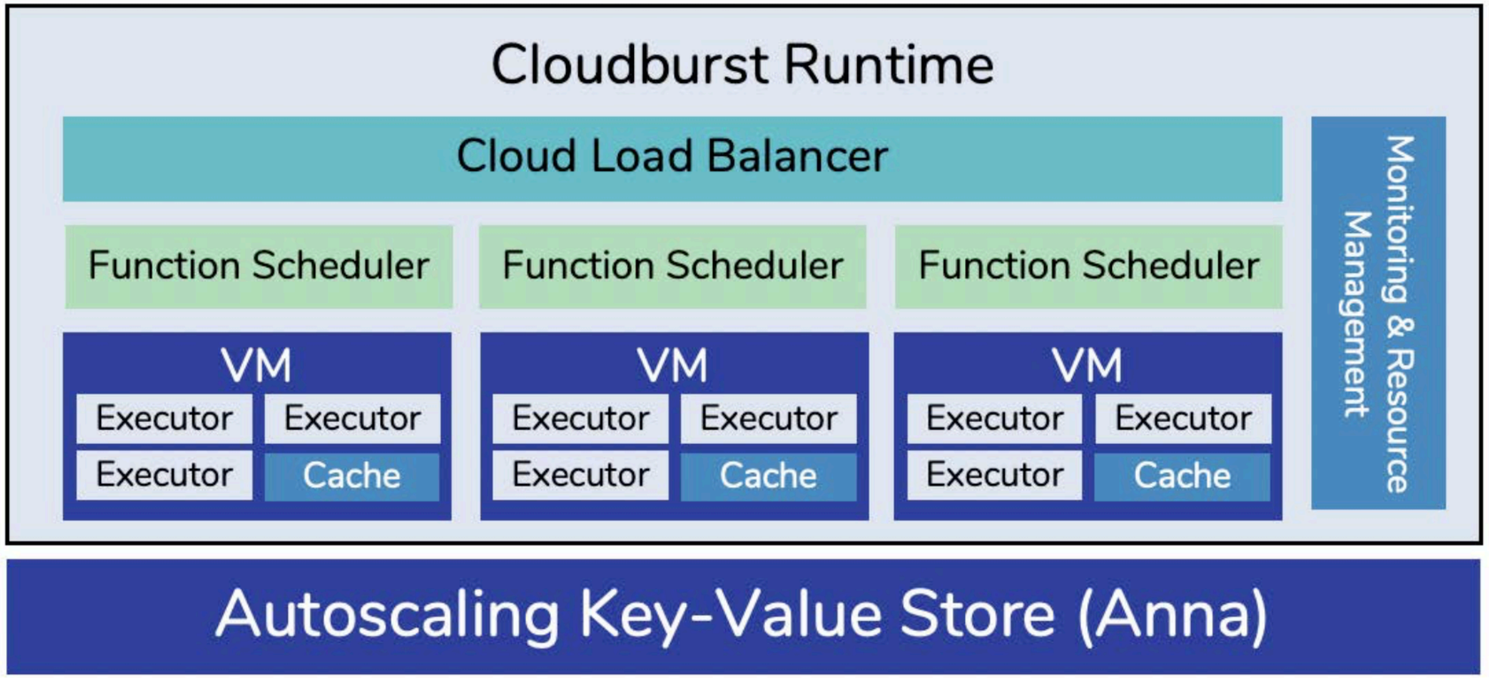
Key Idea: Caching

- Enable low-latency data access by caching data close to code execution

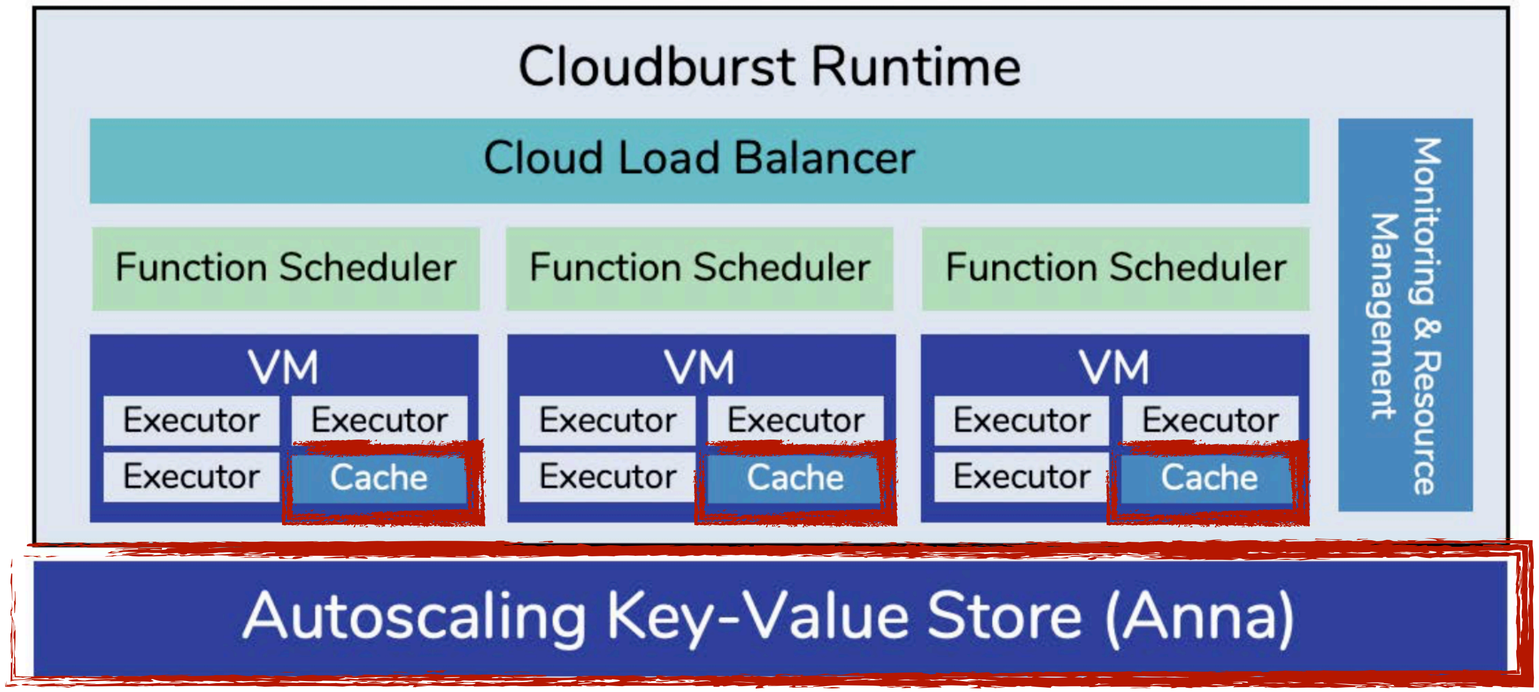
- Communication (and composition) is achieved via a fast-path on top of KVS puts and gets



Anna Architecture



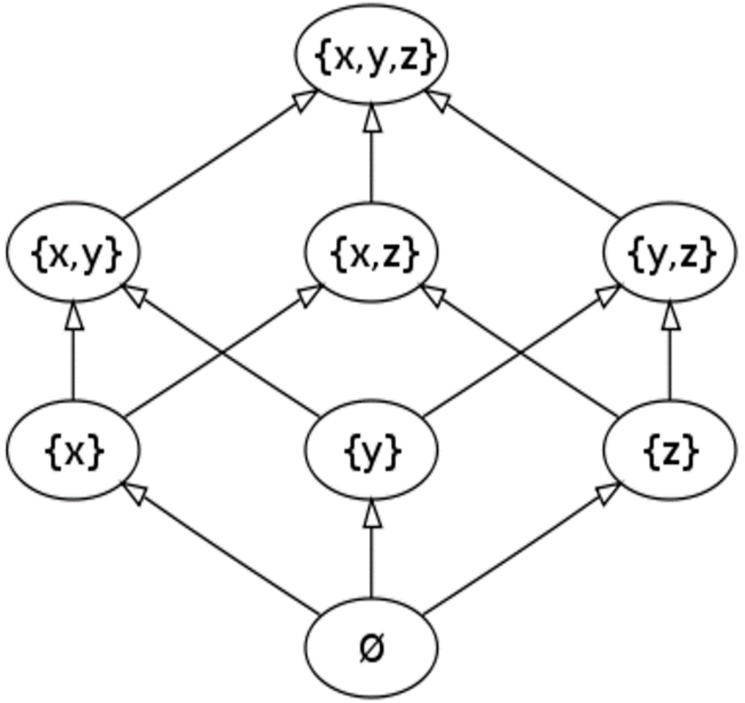
Caching is a Problem



Lattice

- Data structure that accepts incoming update in a way that is associative, commutative, and idempotent (ACI)

- Achieves eventual replica convergence



Cache Consistency in DAG

- The Scheduler creates a schedule for a DAG
- Scheduler persist DAG topologies in KVS
- DAG is a session — **CACHE CONSISTENCY NEEDED**
  - Run the entire DAG in a single thread
    - ❌ Autoscaling
    - ❌ Flexible
  - We need a distributed session consistency.

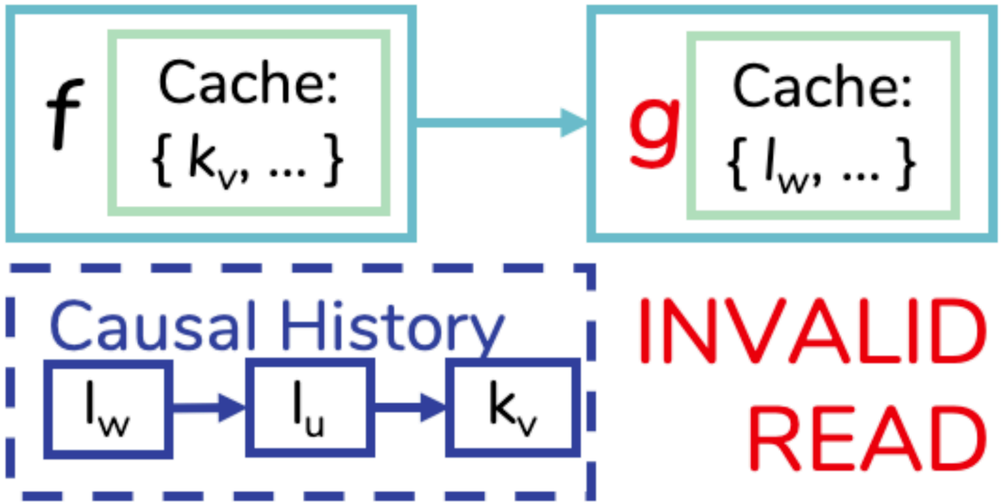
Distributed Session Repeatable Read

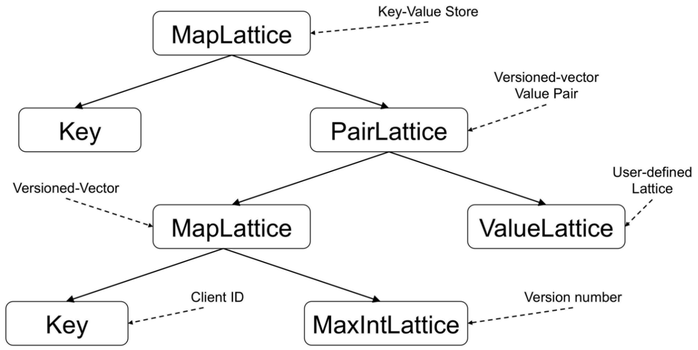
Algorithm 1 Repeatable Read

Input: k, R

1: // k is the requested key; R is the set of keys previously read by the DAG
2: **if** $k \in R$ **then**
3: $cache\_version = cache.get\_metadata(k)$
4: **if** $cache\_version == NULL \vee cache\_version != R[k].version$ **then**
5: return $cache.fetch\_from\_upstream(k)$
6: **else**
7: return $cache.get(k)$
8: **else**
9: return $cache.get\_or\_fetch(k)$

Distributed Session Causal Consistency





Distributed Session Causal Consistency

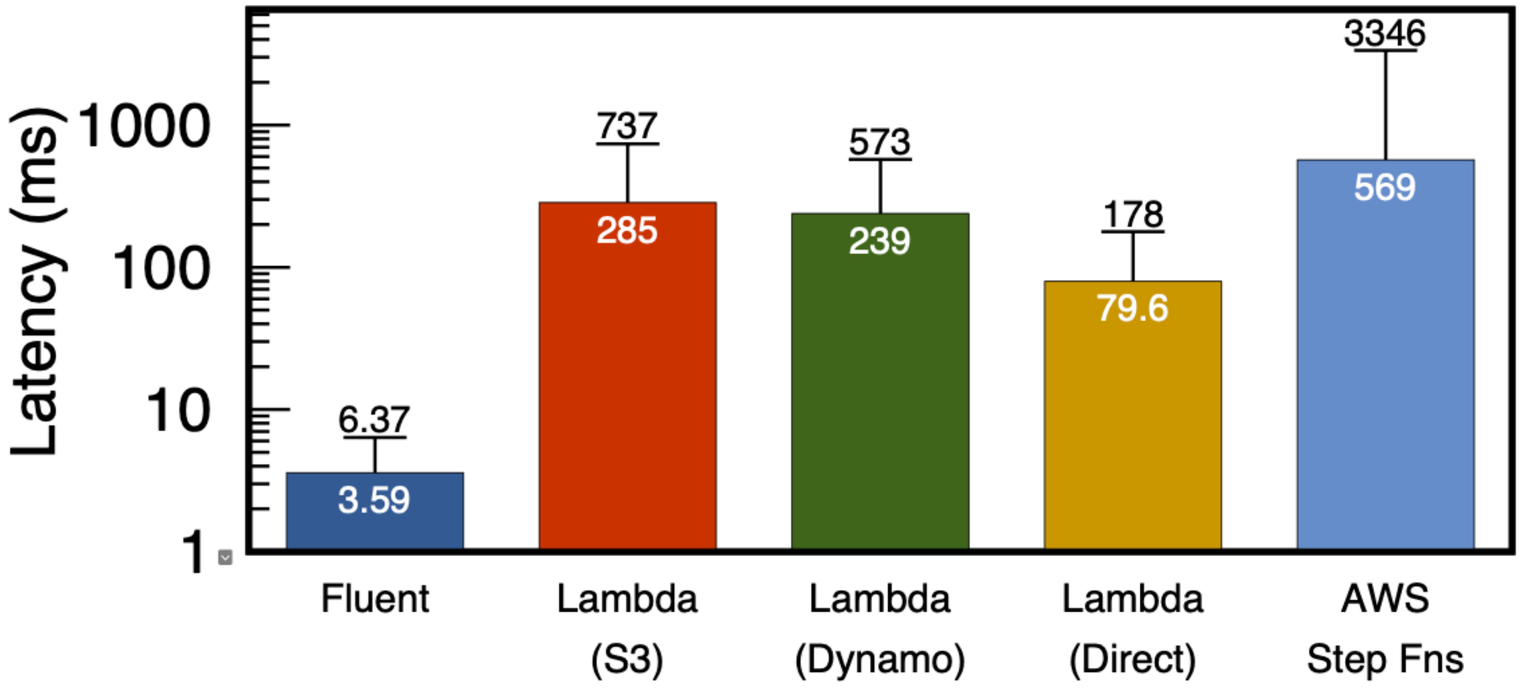
Algorithm 2 Causal Consistency

Input: k, R, $dependencies$

1: // k is the requested key; R is the set of keys previously read by the DAG; $dependencies$ is the set of causal dependencies of keys in R
2: **if** $k \in R$ **then**
3: $cache\_version = cache.get\_metadata(k)$
4: // $valid$ returns true if $k \geq cache\_version$
5: **if** $valid(cache\_version, R[k])$ **then**
6: return $cache.get(k)$
7: **else**
8: return $cache.fetch\_from\_upstream(k)$
9: **if** $k \in dependencies$ **then**
10: $cache\_version = cache.get\_metadata(k)$
11: **if** $valid(cache\_version, dependencies[k])$ **then**
12: return $cache.get(k)$
13: **else**
14: return $cache.fetch\_from\_upstream(k)$

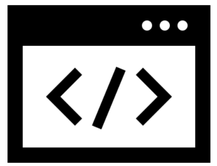
Function Composition, Revisited

$$f(g(x))$$

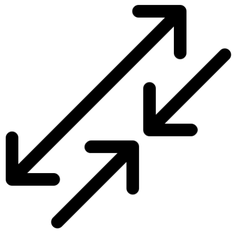


Median and 99th percentile latencies for composing two arithmetic functions on AWS Lambda.

Moving Forward from FaaS



Building
Developer
Tools



Developing
Autoscaling
Policy

Designing
SLOs & SLAs

Thanks