Exploiting Serverless Function to Build a Cost-effective Cloud Storage

DS 5110/CS 5501: Big Data Systems

Spring 2024

Lecture 10d

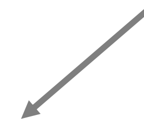
Yue Cheng

@ 2024 released for use under a CC BY-SA license.

Challenges of supporting stateful apps on FaaS

Research Question: How to overcome those limitations to embrace FaaS for stateful applications?

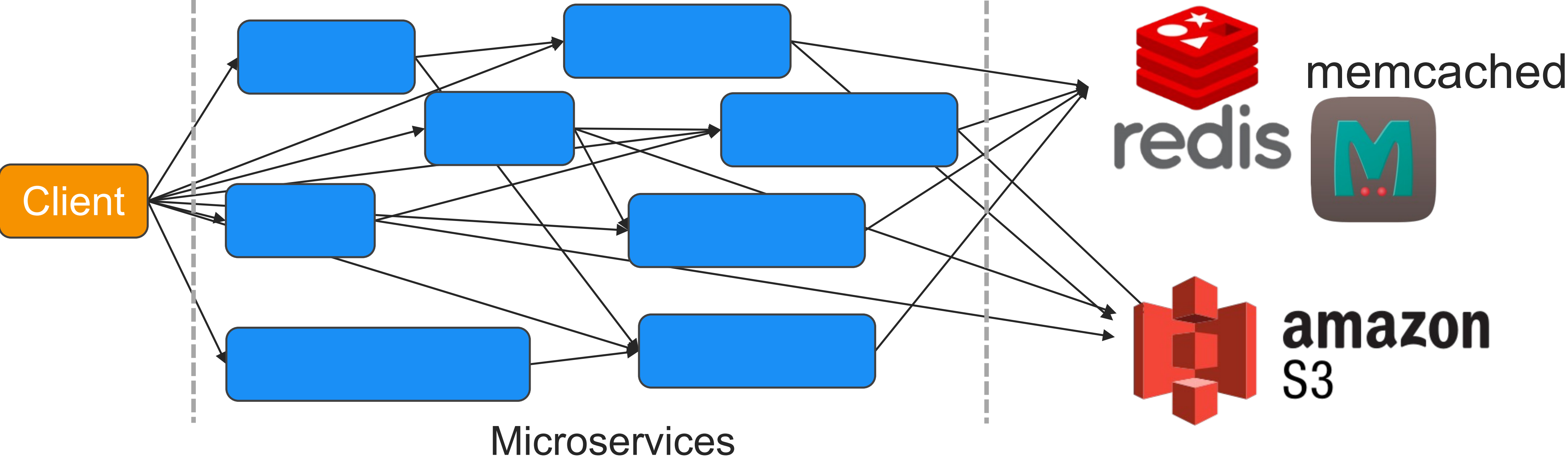
Case studies:



1. **[Parallel programming]** How to design FaaS-centric parallel computing to **enable easy programming of 10,000 CPU cores and 15,000 GBs of RAM**
2. **[Data storage]** How to exploit FaaS to reduce the $$ cost by **100X**

Today

Internet-scale web apps are storage-intensive



Example app: IBM Cloud Container Registry

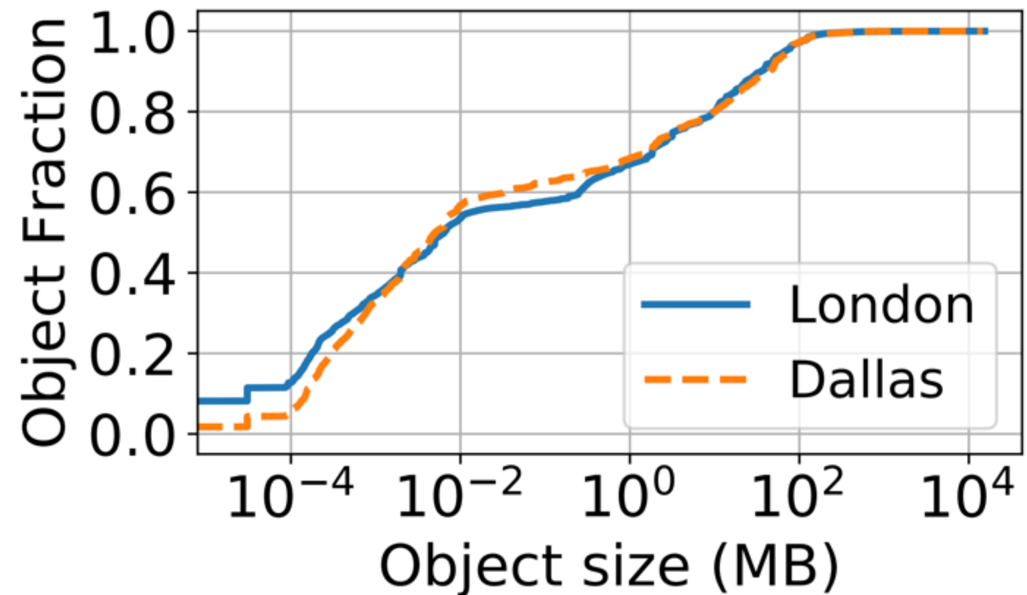
- Collected the workload traces of IBM Cloud Container Registry service for a duration of **75 days** across **seven datacenters** in 2017
- Selected datacenters: Dallas & London

Example app: IBM Cloud Container Registry

- Object size distribution
- Large objects' reuse patterns
- Storage footprint

Example app: IBM Cloud Container Registry

- Object size distribution
- Large objects' reuse patterns
- Storage footprint

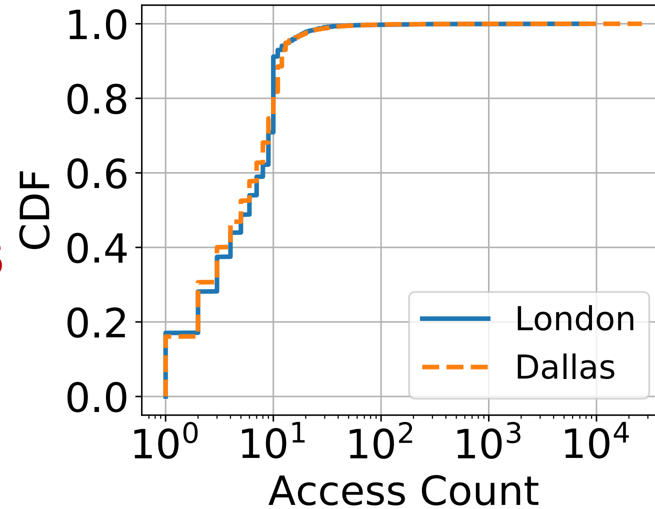


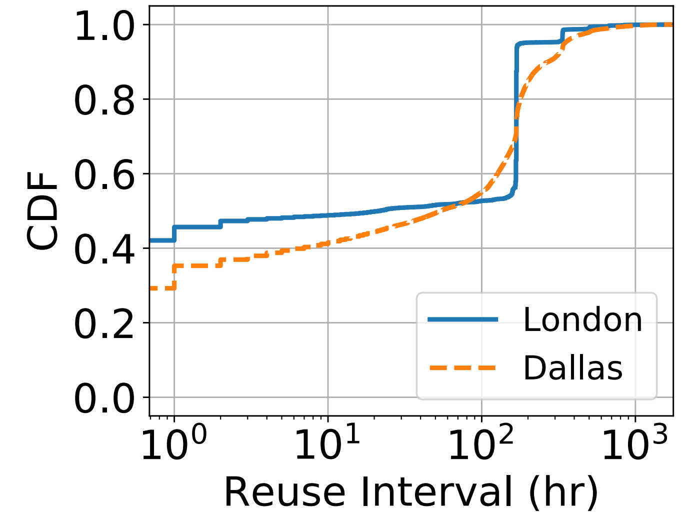
Extreme variability in object sizes:

- ➢ Object sizes span over 9 orders of magnitude
- ➢ 20% of objects > 10MB

Example app: IBM Cloud Container Registry

- Object size distribution
- Large objects' reuse patterns
- Storage footprint



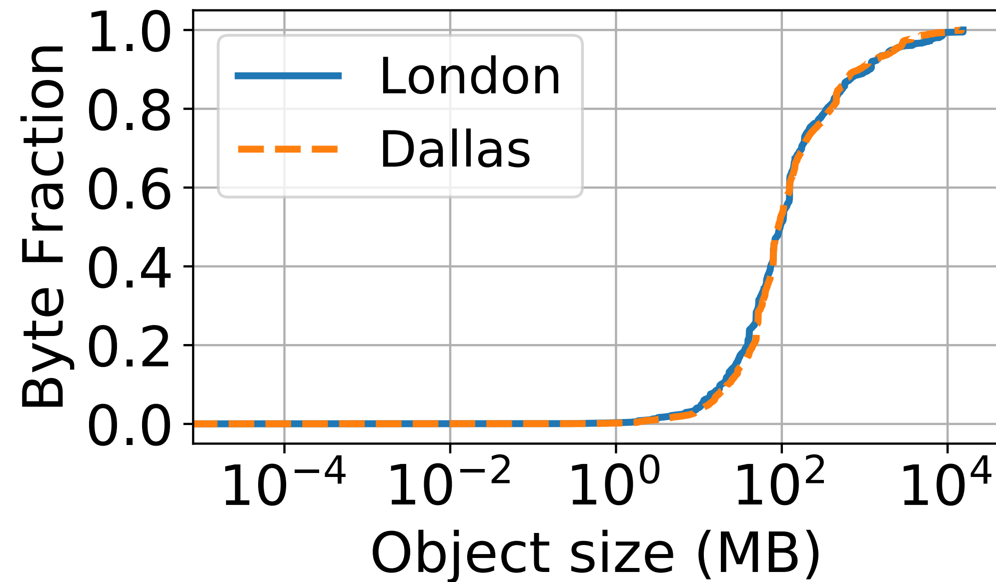


Caching large objects is beneficial:

- > 30% large object being accessed 10+ times
- Around 35-45% of them get reused within 1 hour

Example app: IBM Cloud Container Registry

- Object size distribution
- Large objects’ reuse patterns
- **Storage footprint**



Extreme tension between small and large objects:

➢ Large objects (>10MB) occupy 95% storage footprint

Today’s cloud storage landscape

Today's cloud storage landscape

Object stores are cheap but too slow

AWS S3: $0.023 per GB per month

Today's cloud storage landscape

Slow

Performance (latency)

Better

Fast

Object stores are cheap but too slow

Memory caches are fast but too expensive

AWS ElastiCache: $0.016 per GB per hour

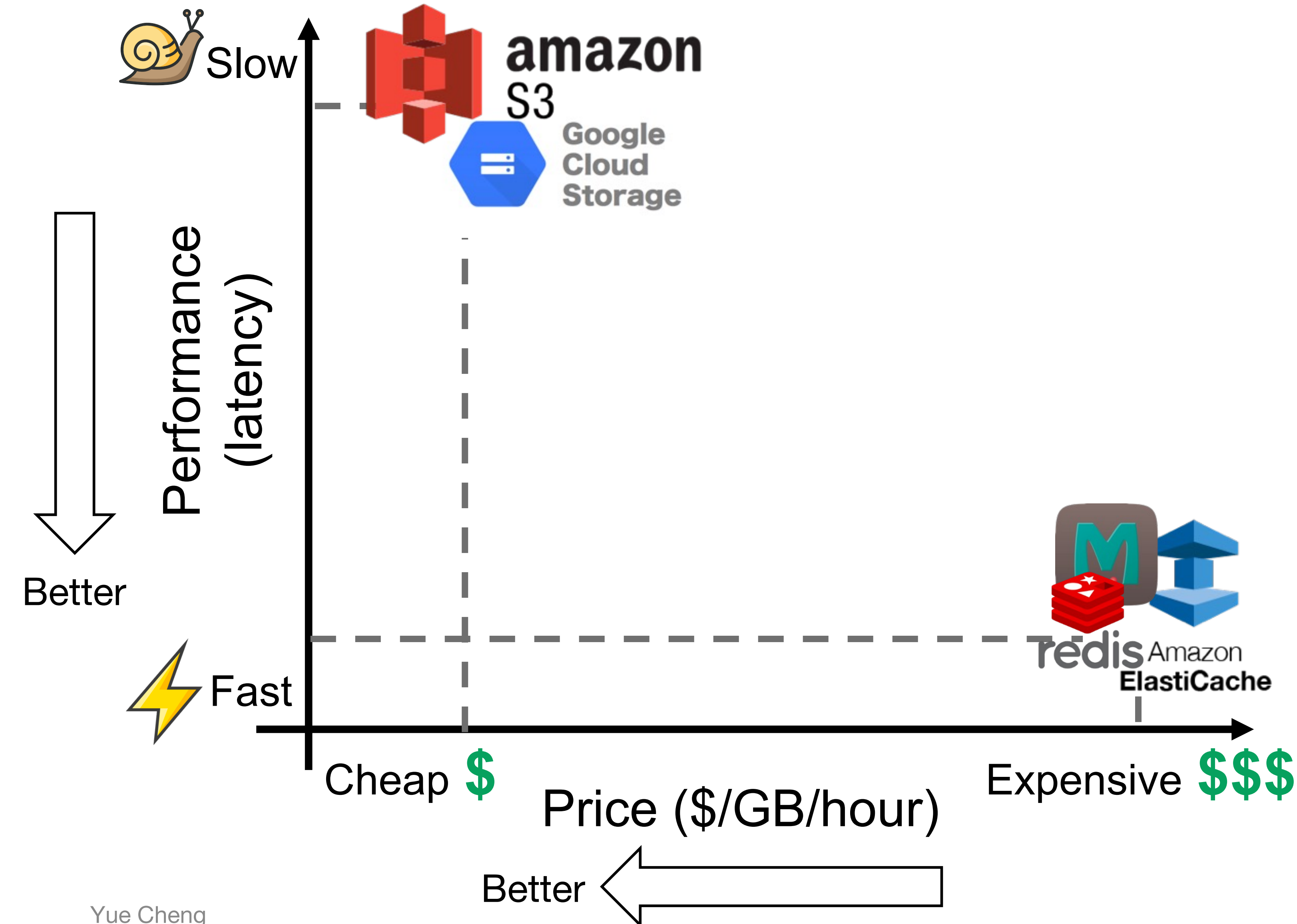
Cheap $

Expensive $$$

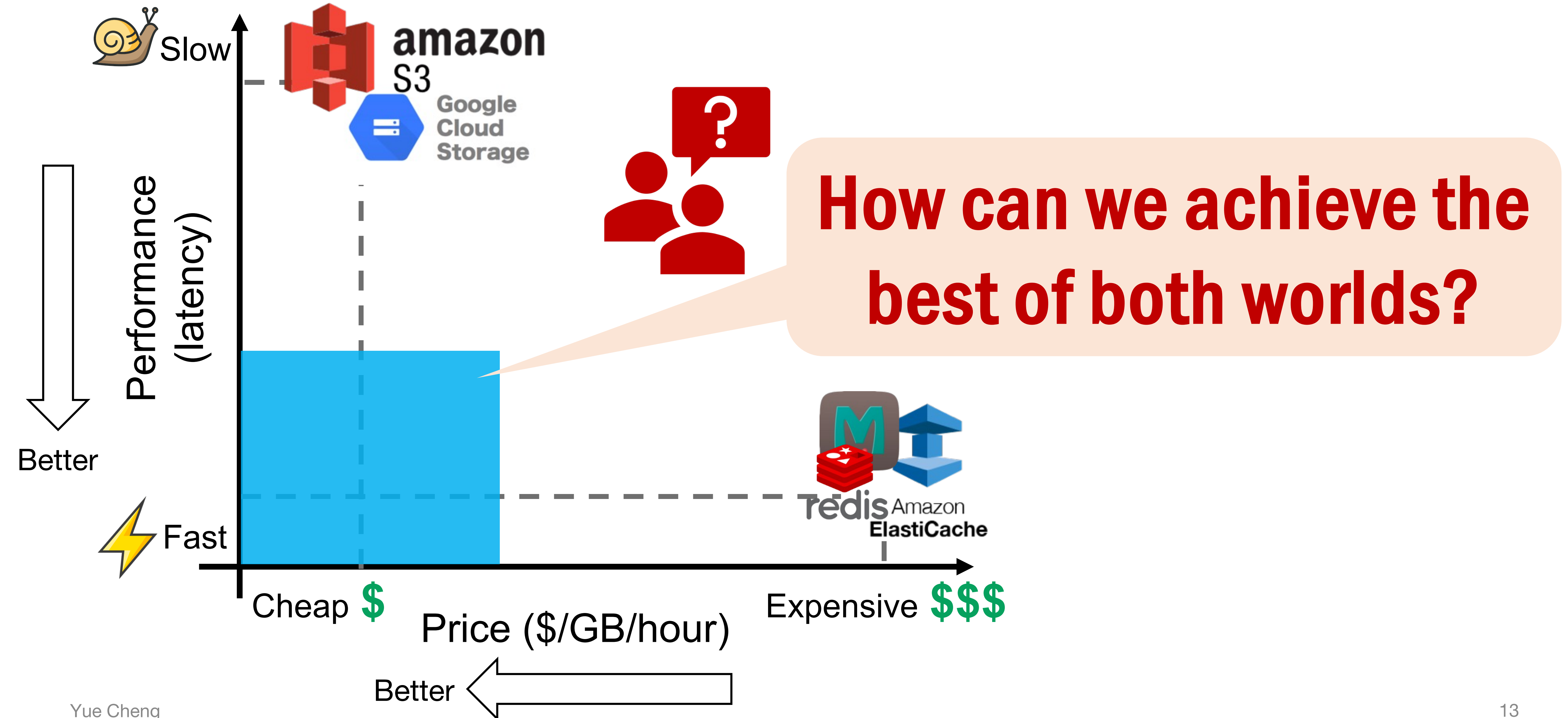
Price ($/GB/hour)

Better

- **Caching both small and large objects** is challenging
- Existing solutions **either too slow or too expensive**



- **Caching both small and large objects** is challenging
- Existing solutions **either too slow or too expensive**



InfiniCache: A cost-effective and high-performance memory cache built atop FaaS

- **Insight #1:** Serverless functions’ <CPU, RAM> resources are pay-per-use
- **Insight #2:** Serverless providers offer “free” function memory caching for tenants

InfiniCache: A cost-effective and high-performance memory cache built atop FaaS

- **Insight #1:** Serverless functions' <CPU, RAM> resources are pay-per-use → Cheap
- **Insight #2:** Serverless providers offer "free" function memory caching for tenants → Fast and cheap

Challenges to build a memory cache using serverless functions

High-level idea: Use Lambda functions to cache data objects

A strawman proposal that directly caches data objects in Lambda functions' memory may not work because of those FaaS limitations:

- No guaranteed data availability
- Banned inbound network
- Limited per-function resources

Challenges to build a memory cache using serverless functions

High-level idea: Use Lambda functions to cache data objects

A strawman proposal that directly caches data objects in Lambda functions' memory may not work because of those FaaS limitations:

⚠️ Serverless functions could be reclaimed any time

⚠️ In-memory state is lost

- No guaranteed data availability
- Banned inbound network
- Limited per-function resources

Challenges to build a memory cache using serverless functions

High-level idea: Use Lambda functions to cache data objects

A strawman proposal that directly caches data objects in Lambda functions' memory may not work because of those FaaS limitations:

- No guaranteed data availability
- Banned inbound network
- Limited per-function resources

⚠️ Serverless functions cannot run as a server

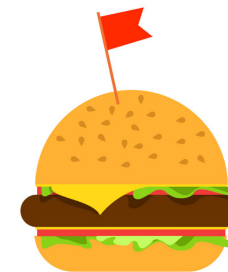
Challenges to build a memory cache using serverless functions

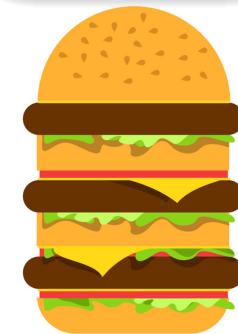
High-level idea: Use Lambda functions to cache data objects

A strawman proposal that directly caches data objects in Lambda functions' memory may not work because of those FaaS limitations:

- No guaranteed data availability
- Banned inbound network
- Limited per-function resources

⚠️ Memory up to 10 GB
⚠️ CPU up to 6 cores

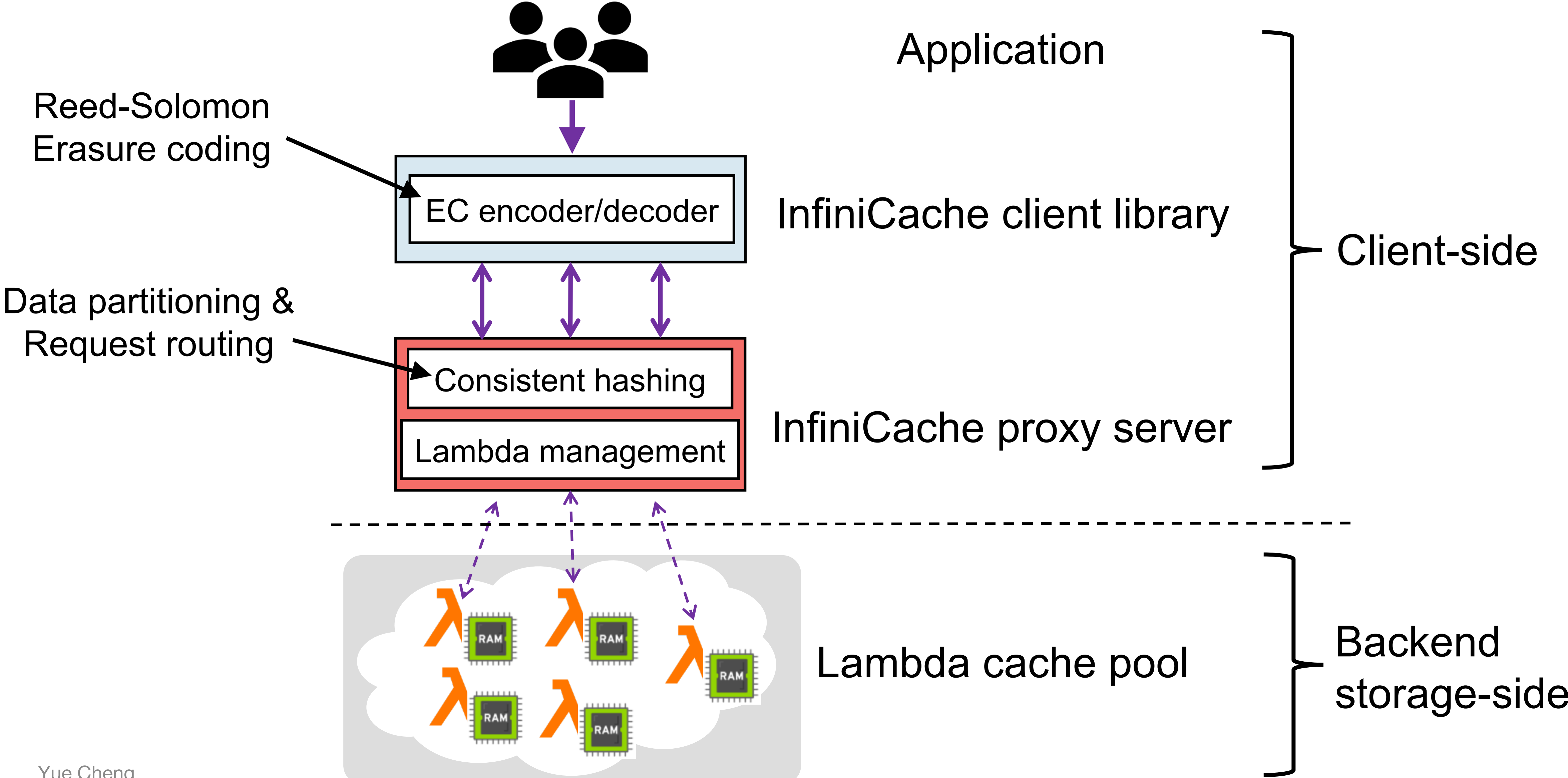




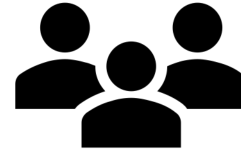
InfiniCache: The first memory cache built atop FaaS

- InfiniCache achieves high data availability by using erasure coding and delta-sync periodic data backup across functions
- InfiniCache achieves high performance by utilizing the aggregated, parallel network bandwidth of multiple functions
- InfiniCache achieves similar performance to AWS ElastiCache while reducing the $$ cost by 31-96X

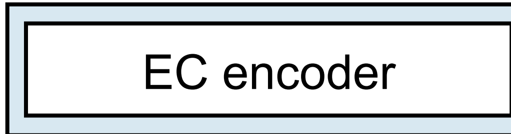
InfiniCache bird's eye view



InfiniCache: PUT path

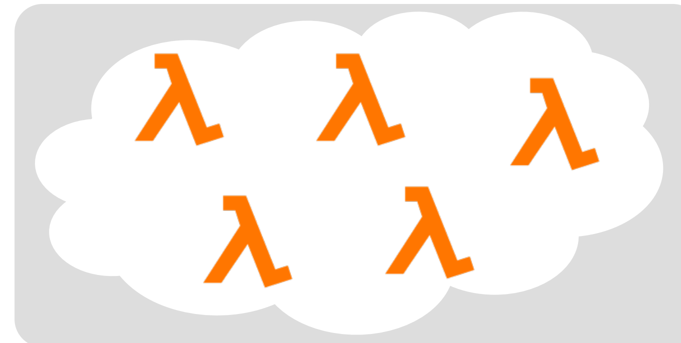


Application



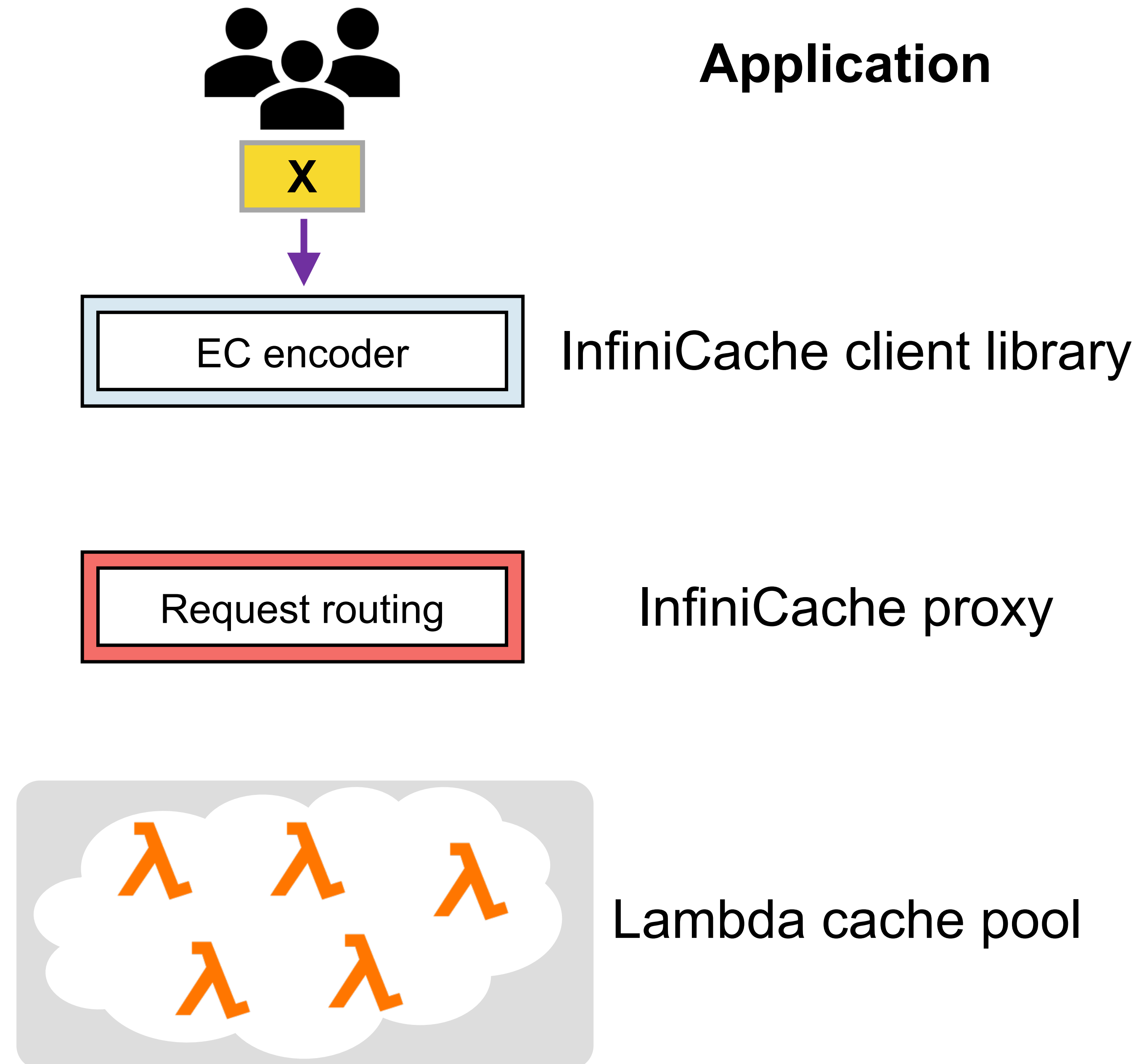
InfiniCache client library

InfiniCache proxy



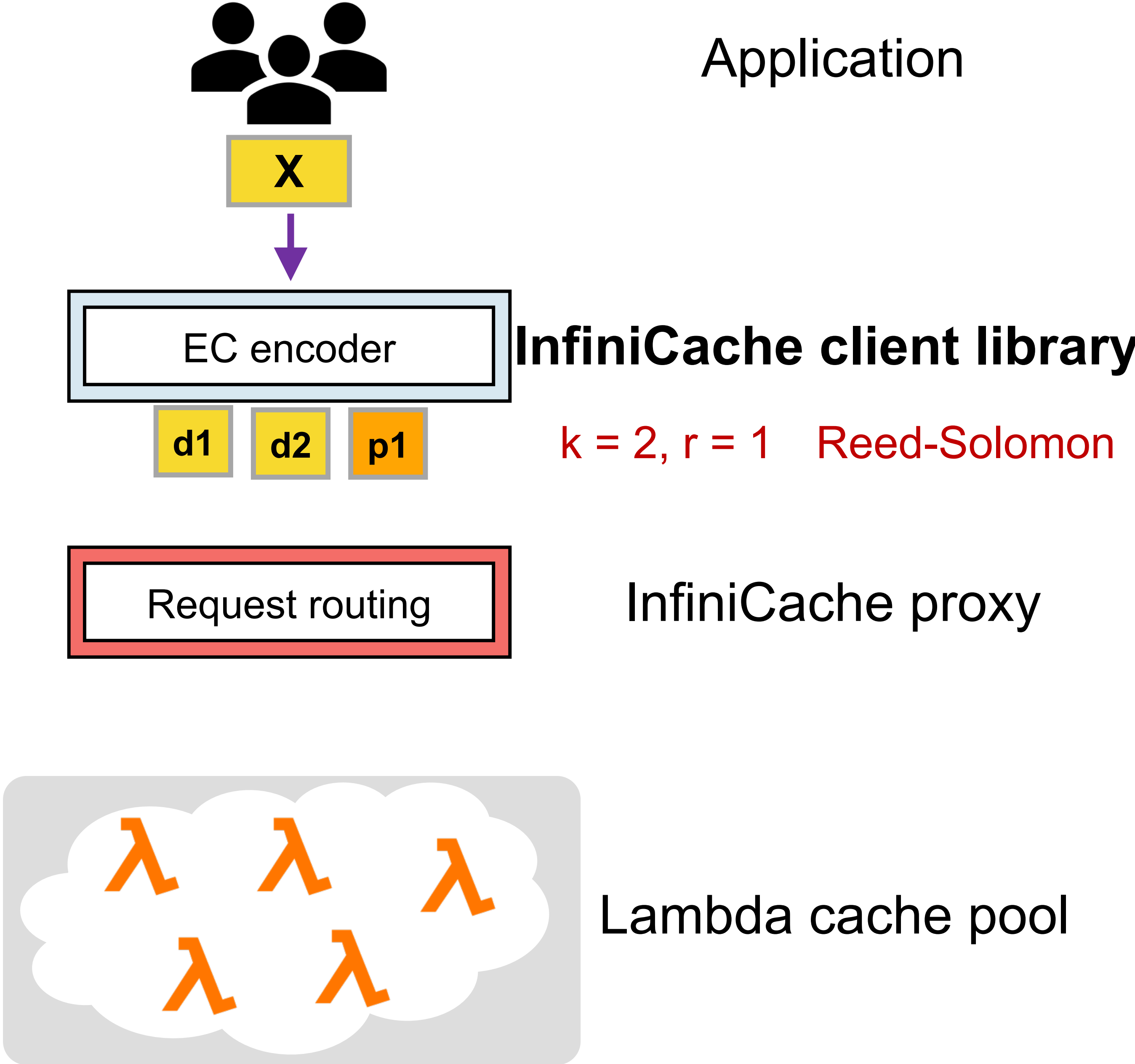
Lambda cache pool

InfiniCache: PUT path



InfiniCache: PUT path

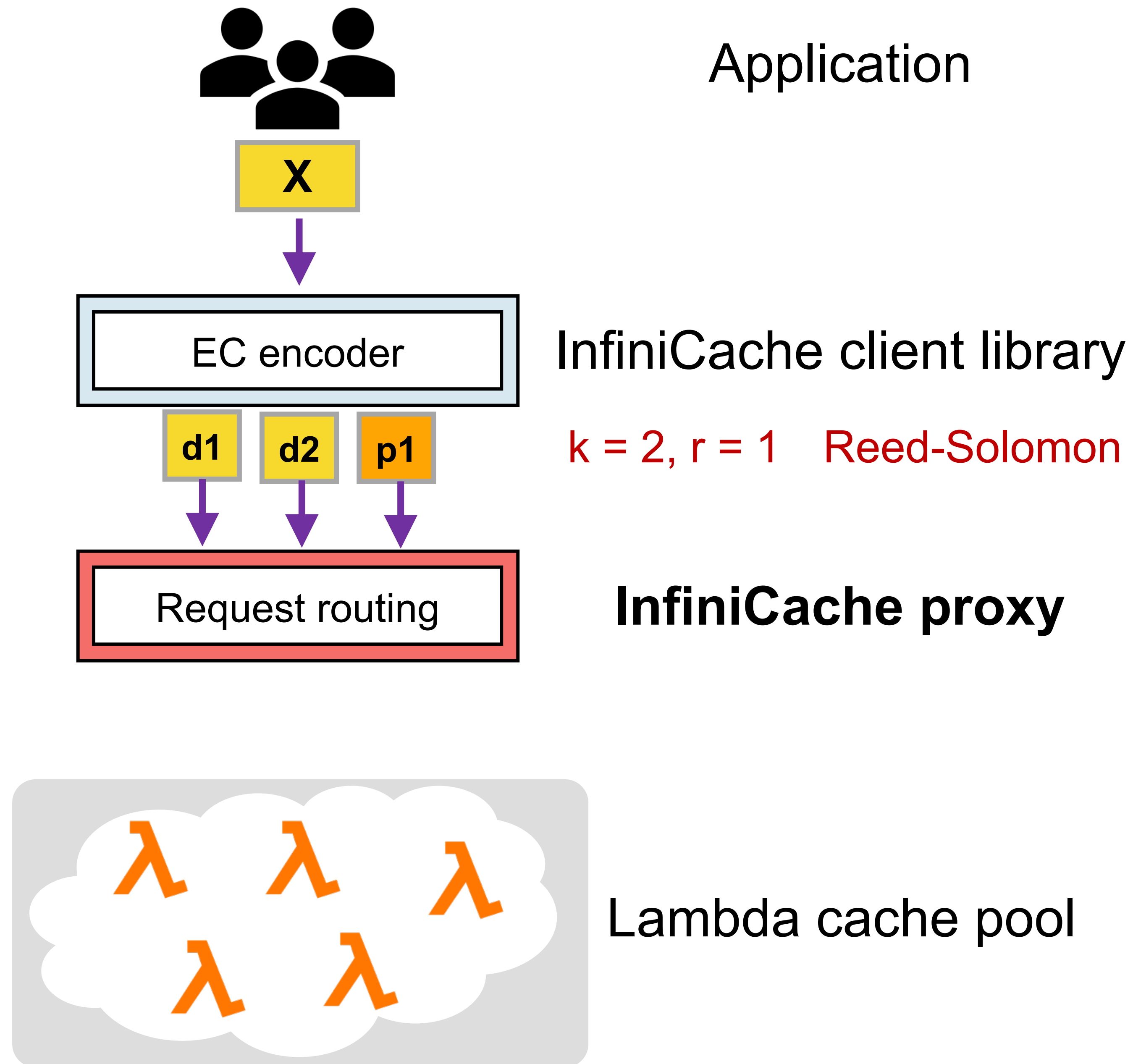
1. Object is split and encoded into k+r chunks



InfiniCache: PUT path

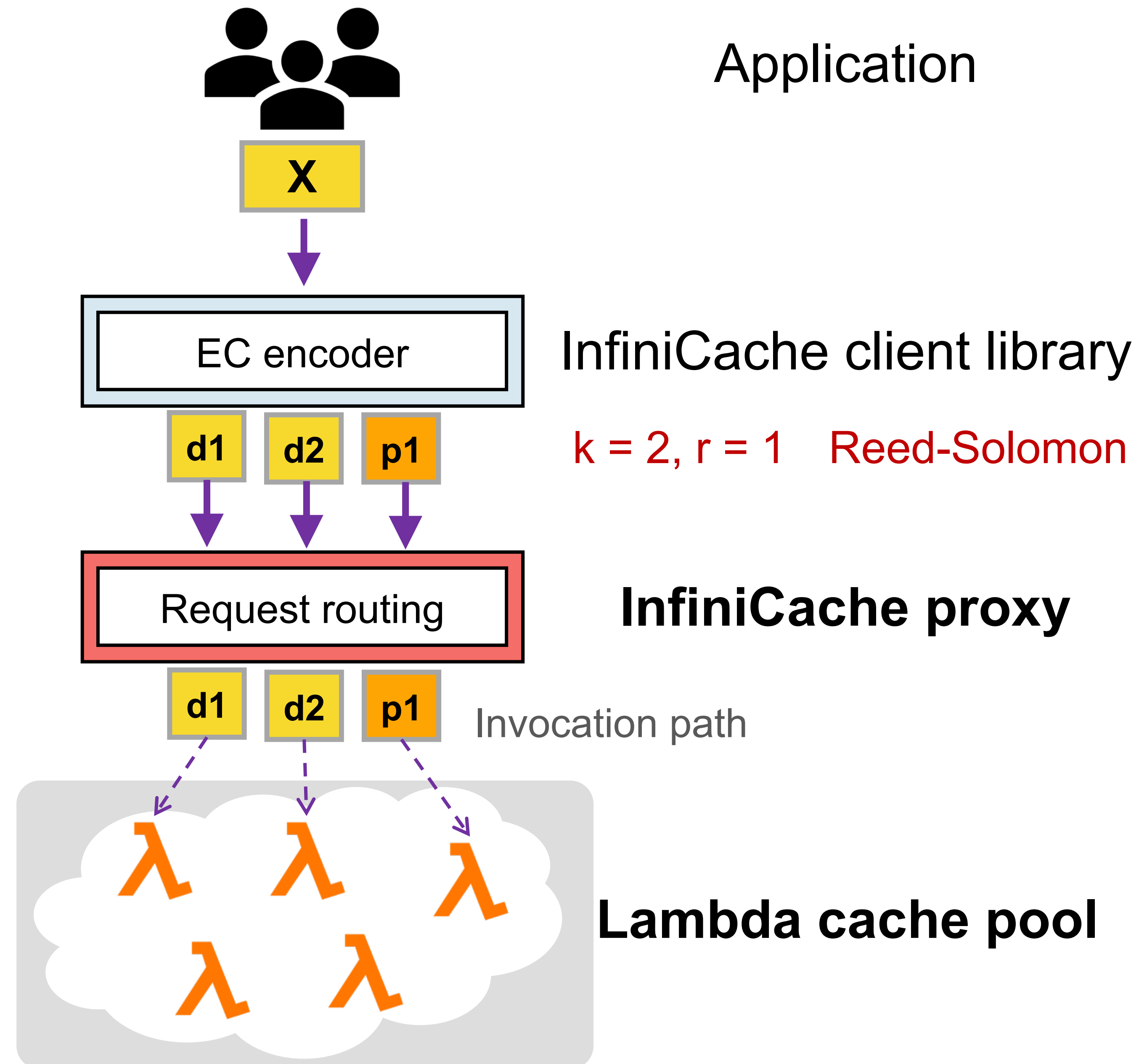
1. Object is split and encoded into k+r chunks

2. Object chunks are sent to the proxy in parallel



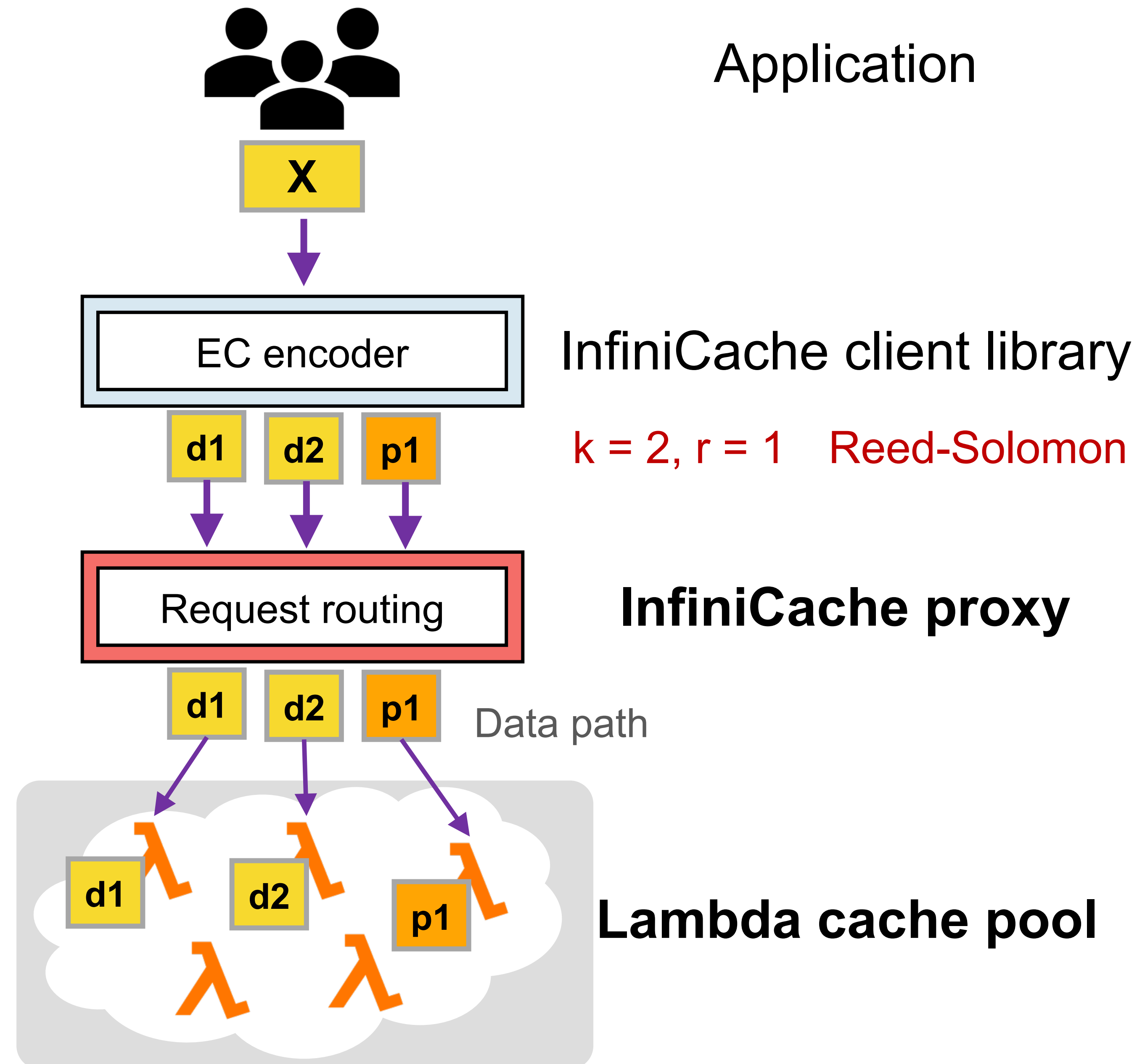
InfiniCache: PUT path

1. Object is split and encoded into k+r chunks
2. Object chunks are sent to the proxy in parallel
3. Proxy invokes Lambda cache nodes

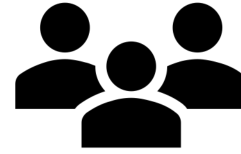


InfiniCache: PUT path

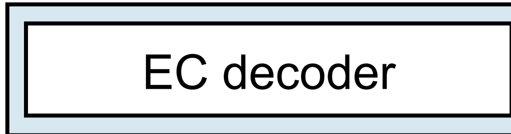
1. Object is split and encoded into k+r chunks
2. Object chunks are sent to the proxy in parallel
3. Proxy invokes Lambda cache nodes
4. Proxy streams object chunks to Lambda cache nodes



InfiniCache: GET path

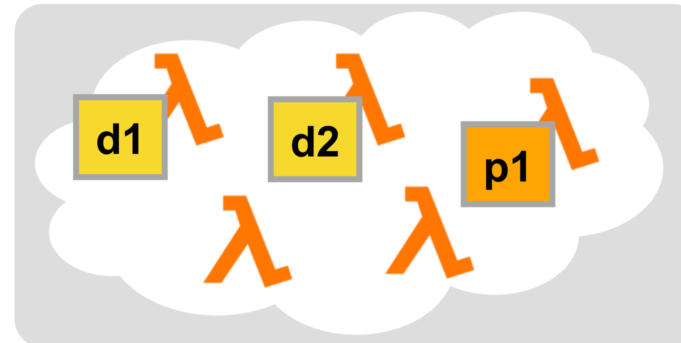


Application



InfiniCache client library

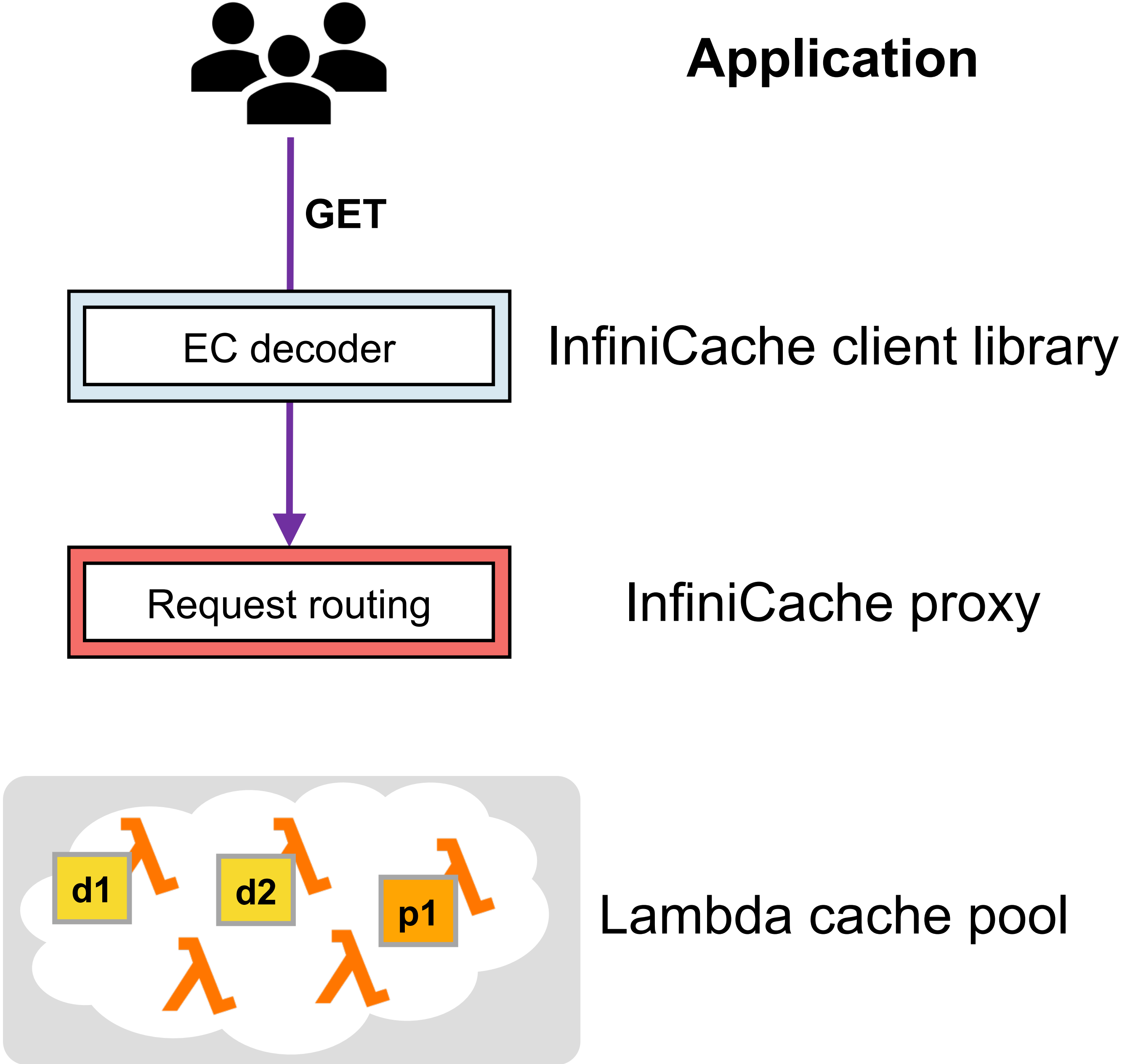
InfiniCache proxy



Lambda cache pool

InfiniCache: GET path

1. Client sends GET request

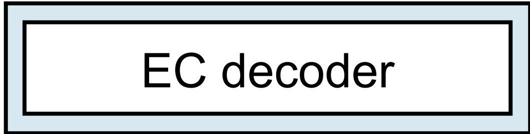


InfiniCache: GET path

Application

1. Client sends GET request

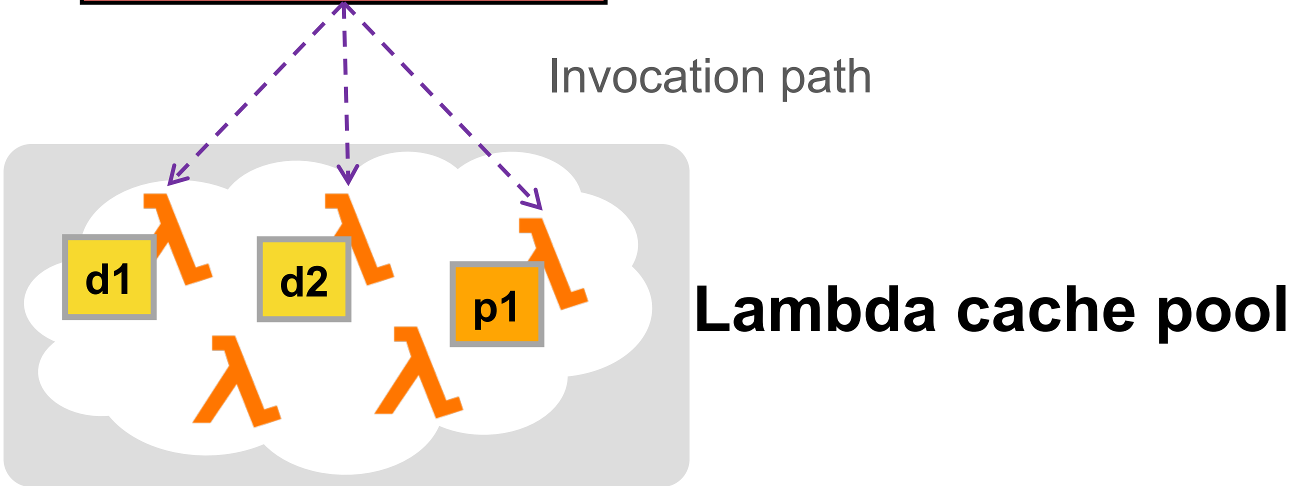
2. Proxy invokes associated Lambda cache nodes



InfiniCache client library

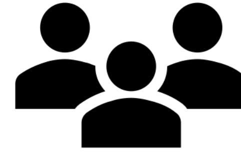
InfiniCache proxy

Invocation path



Lambda cache pool

InfiniCache: GET path

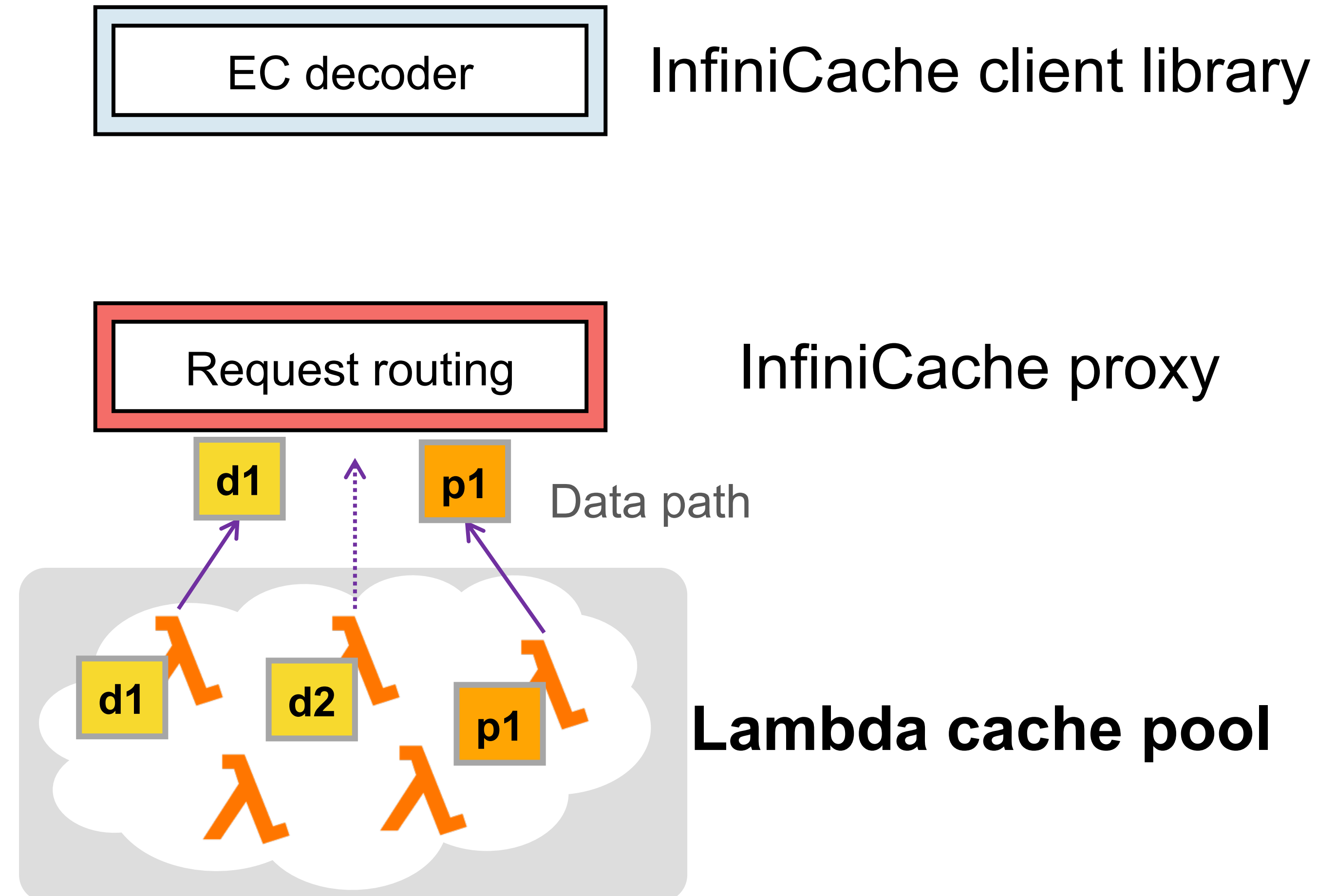


Application

1. Client sends GET request

2. Proxy invokes associated Lambda cache nodes

3. Lambda cache nodes transfer object chunks to proxy

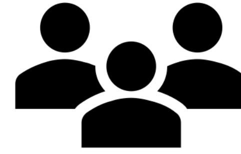


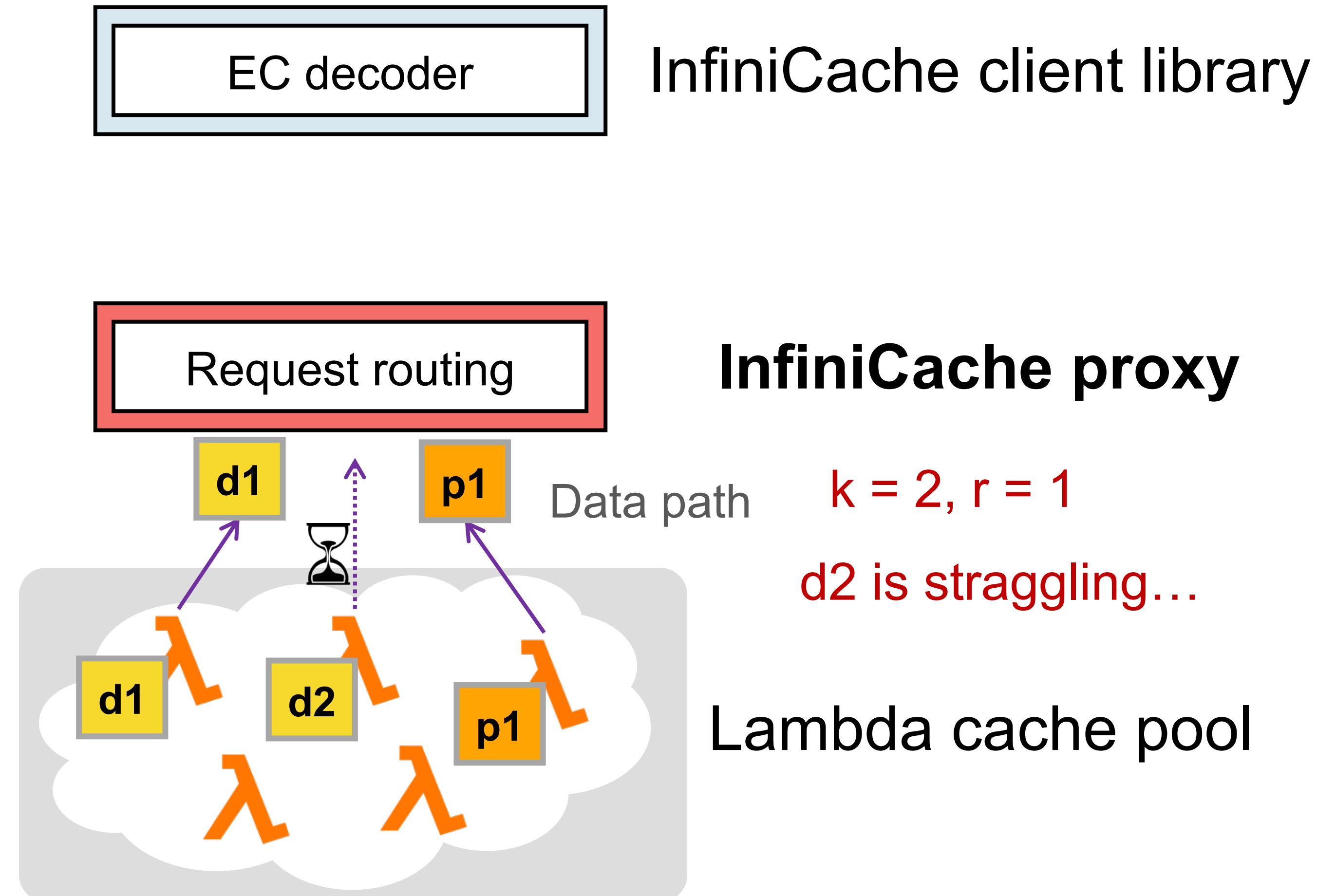
InfiniCache: GET path

1. Client sends GET request

2. Proxy invokes associated Lambda cache nodes

3. Lambda cache nodes transfer object chunks to proxy
   - **First-d optimization:** Proxy drops **straggler** Lambda





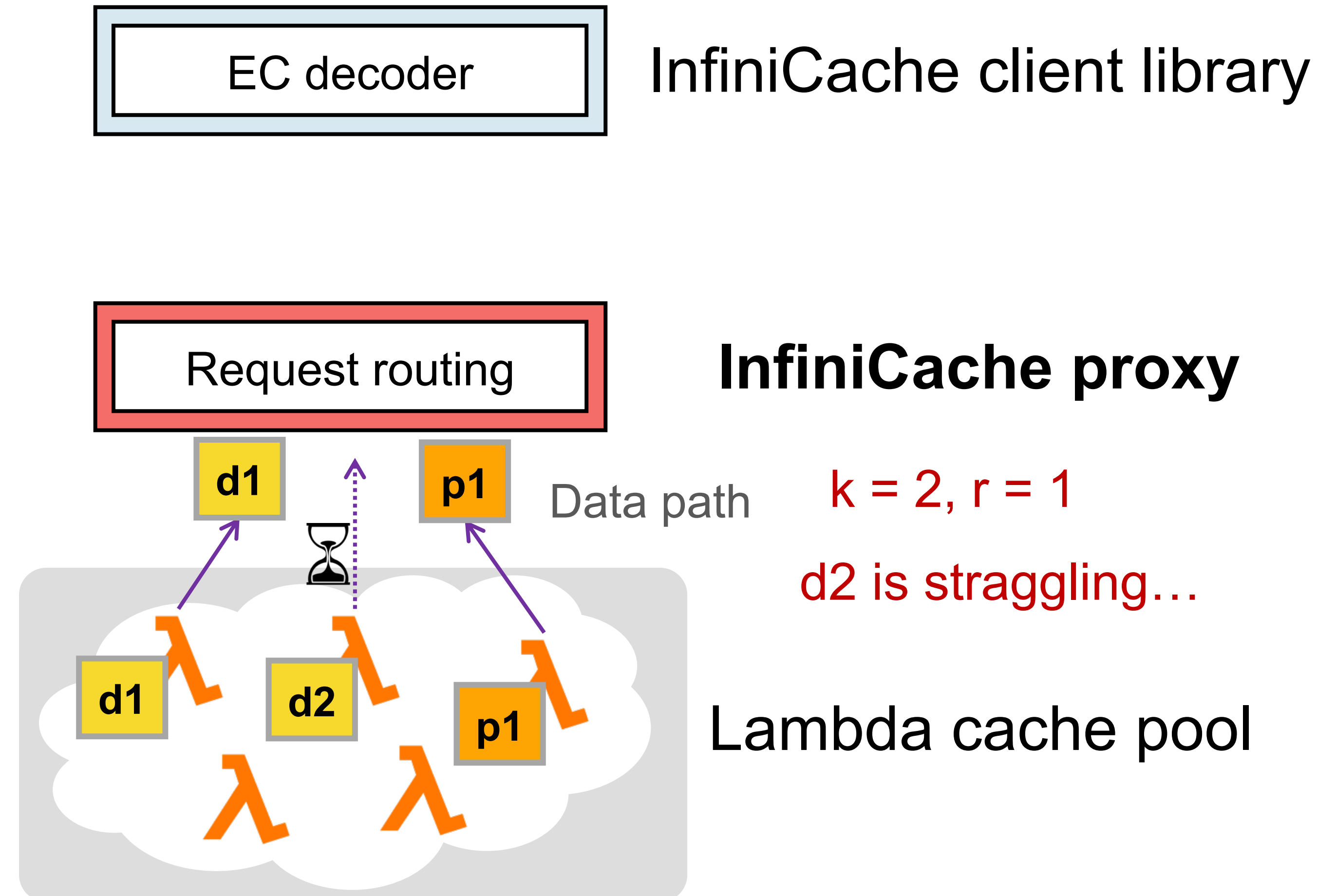
InfiniCache: GET path

Recall MapReduce uses replication to tackle **stragglers**; turns out storage-efficient redundancy technique **erasure coding** can achieve the same goal.

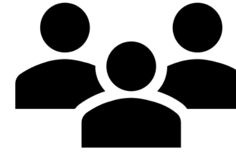
1. Client s quest

2. Proxy inv ociated Lambda ca des

3. Lambda cach odes transfer object chunks proxy
   - **First-d opti nization:** Proxy drops **straggler** Lambda

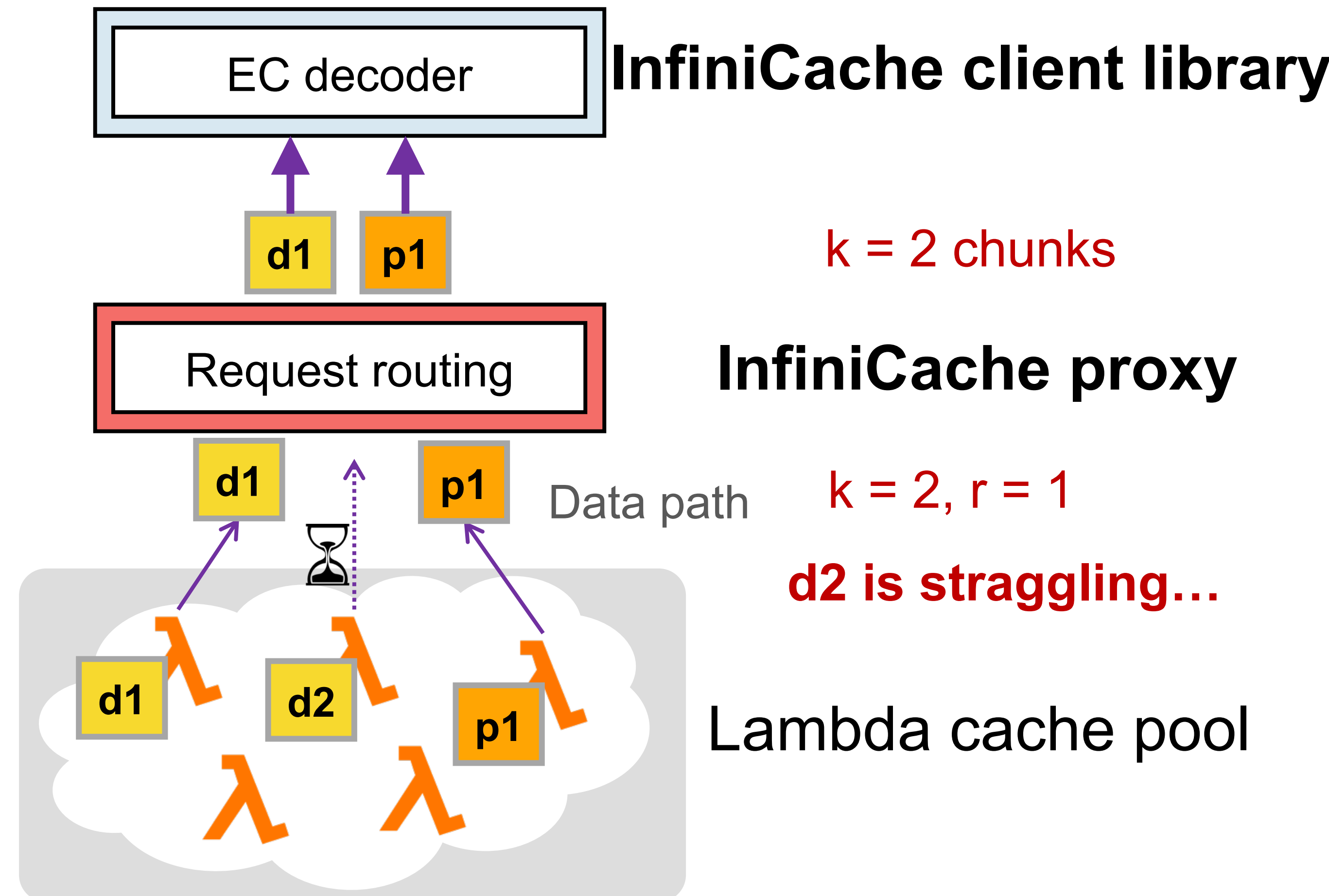


InfiniCache: GET path



Application

1. Client sends GET request
2. Proxy invokes associated Lambda cache nodes
3. Lambda cache nodes transfer object chunks to proxy
4. Proxy streams k=2 chunks in parallel to client



InfiniCache: GET path

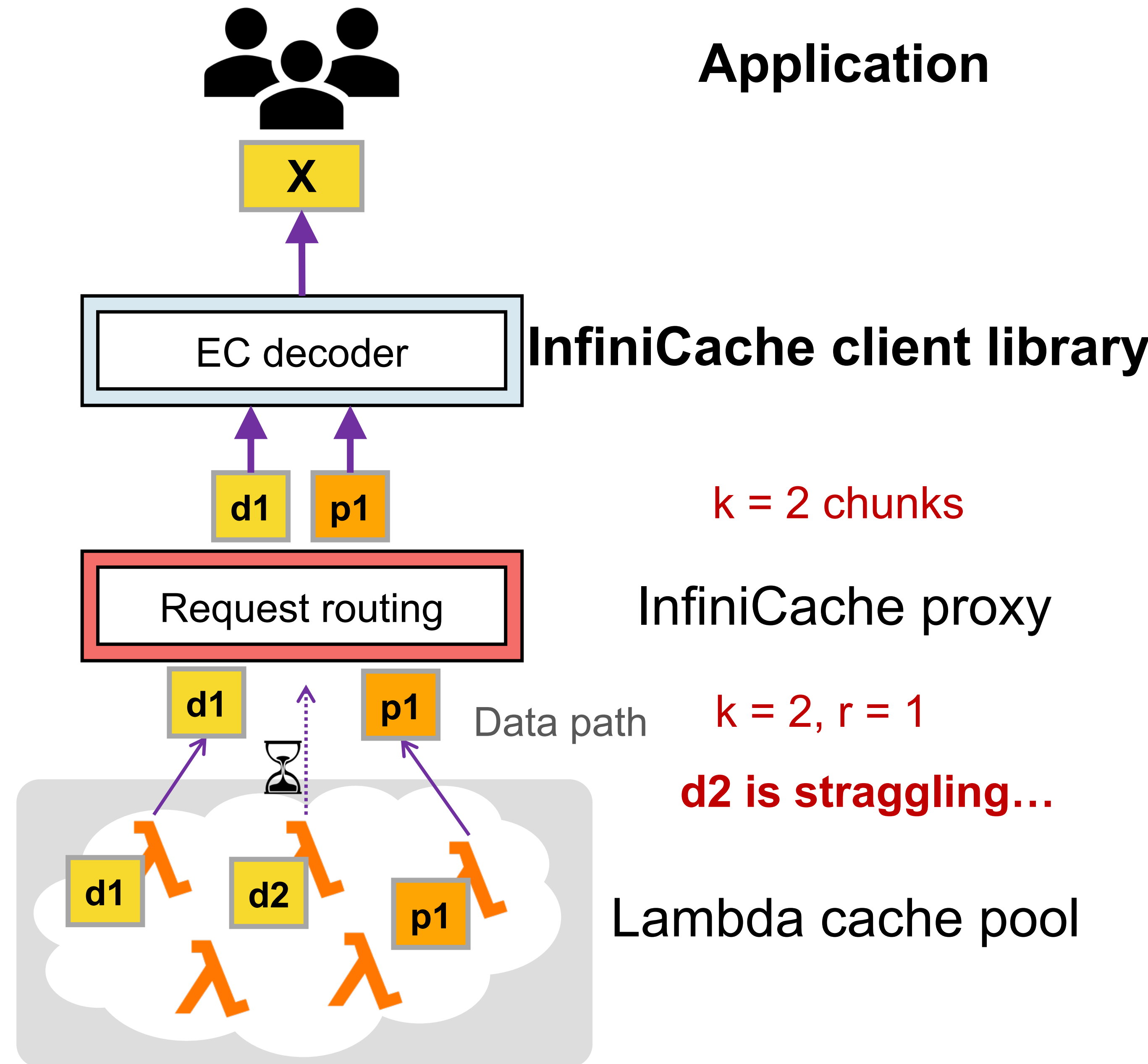
1. Client sends GET request

2. Proxy invokes associated Lambda cache nodes

3. Lambda cache nodes transfer object chunks to proxy

4. Proxy streams k=2 chunks in parallel to client

5. Client library **decodes** k chunks



InfiniCache: GET path

Application

Tradeoff: Computational cost of EC decoding **vs.** delay waiting for the straggler (typically, **computational cost < straggler delay**, thanks to the efficient implementation of modern EC libraries)

1. Client sen

2. Proxy invoke Lambda cac

3. Lambda cache ansfer object chunks t

4. Proxy streams k= hunks in parallel to client

5. Client library **decodes** k chunks

EC decoder

InfiniCache client library

d1 p1

k = 2 chunks

Request routing

InfiniCache proxy

d1 p1

Data path

k = 2, r = 1

d2 is straggling…

d1 d2 p1

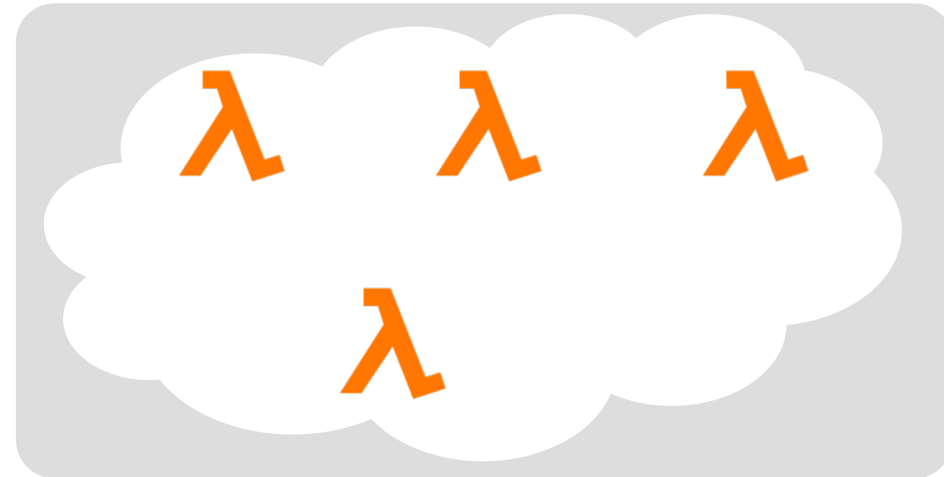
Lambda cache pool

Maximizing data availability

- Erasure-coding
- Periodic warm-up
- Smart delta-sync backup

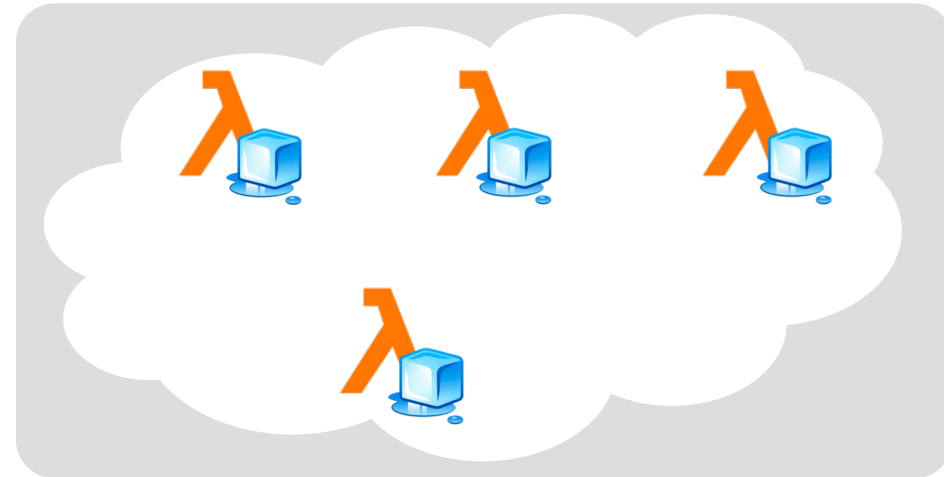
Maximizing data availability: Periodic warm-up

1. Lambda nodes are cached by AWS when not running



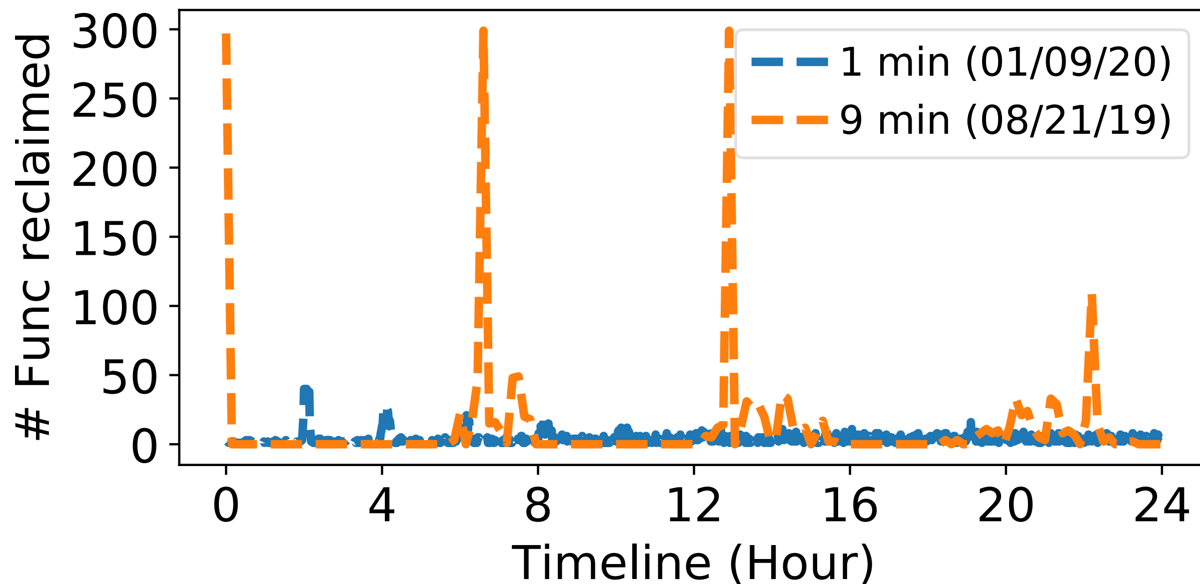
Maximizing data availability: Periodic warm-up

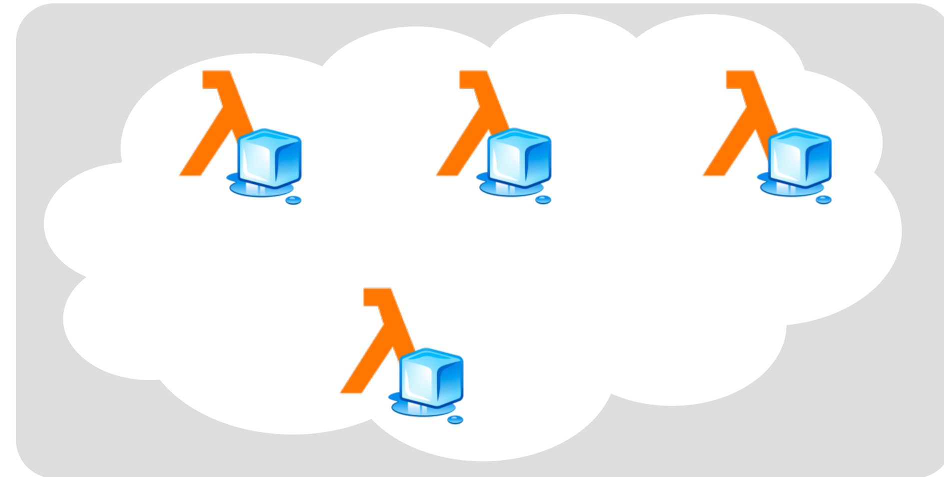
1. Lambda nodes are cached by AWS when not running
   - AWS may reclaim cold Lambda functions after they are idling for a period



Maximizing data availability: Periodic warm-up

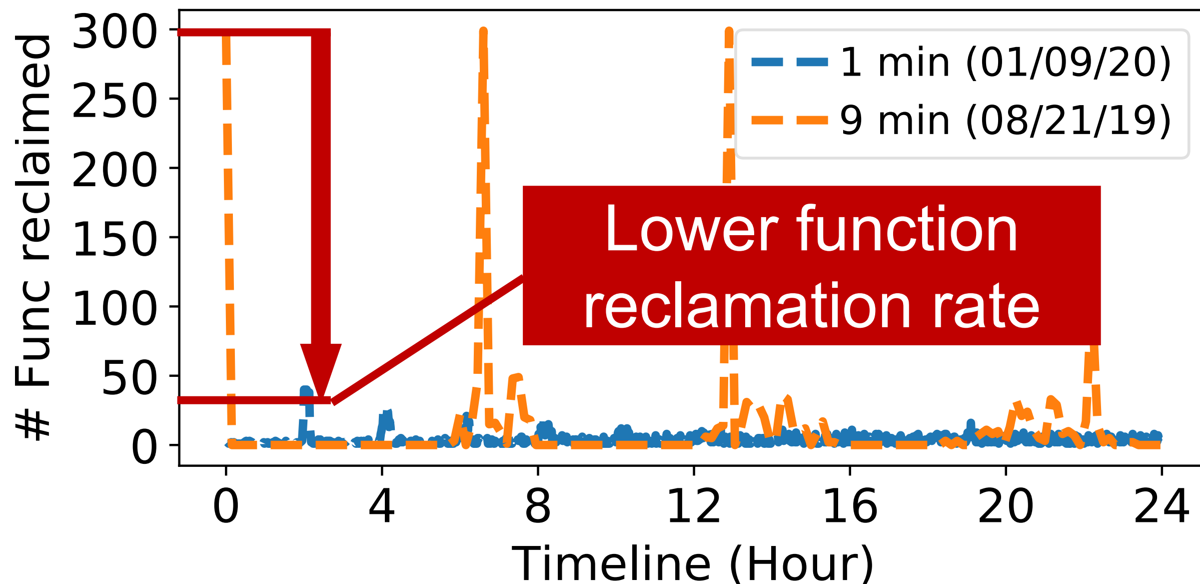
1. Lambda nodes are cached by AWS when not running
   - AWS may reclaim cold Lambda functions after they are idling for a period

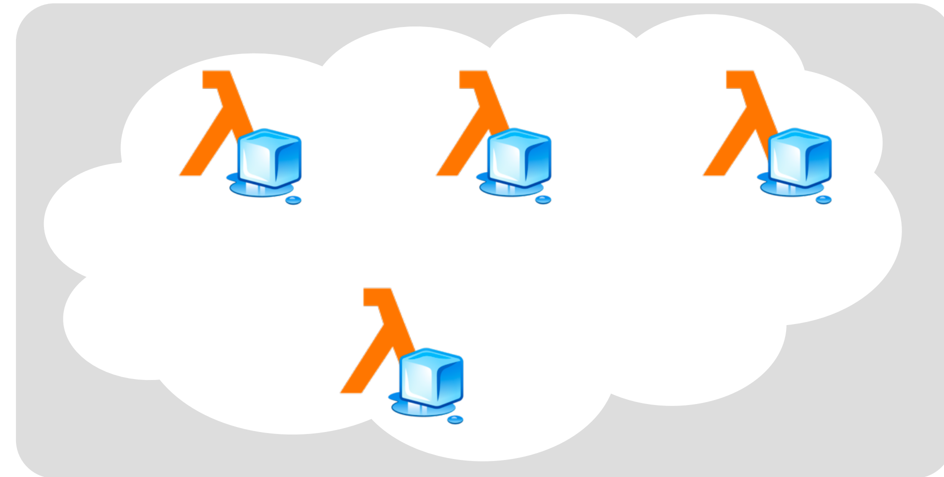




Maximizing data availability: Periodic warm-up

1. Lambda nodes are cached by AWS when not running
   - AWS may reclaim cold Lambda functions after they are idling for a period

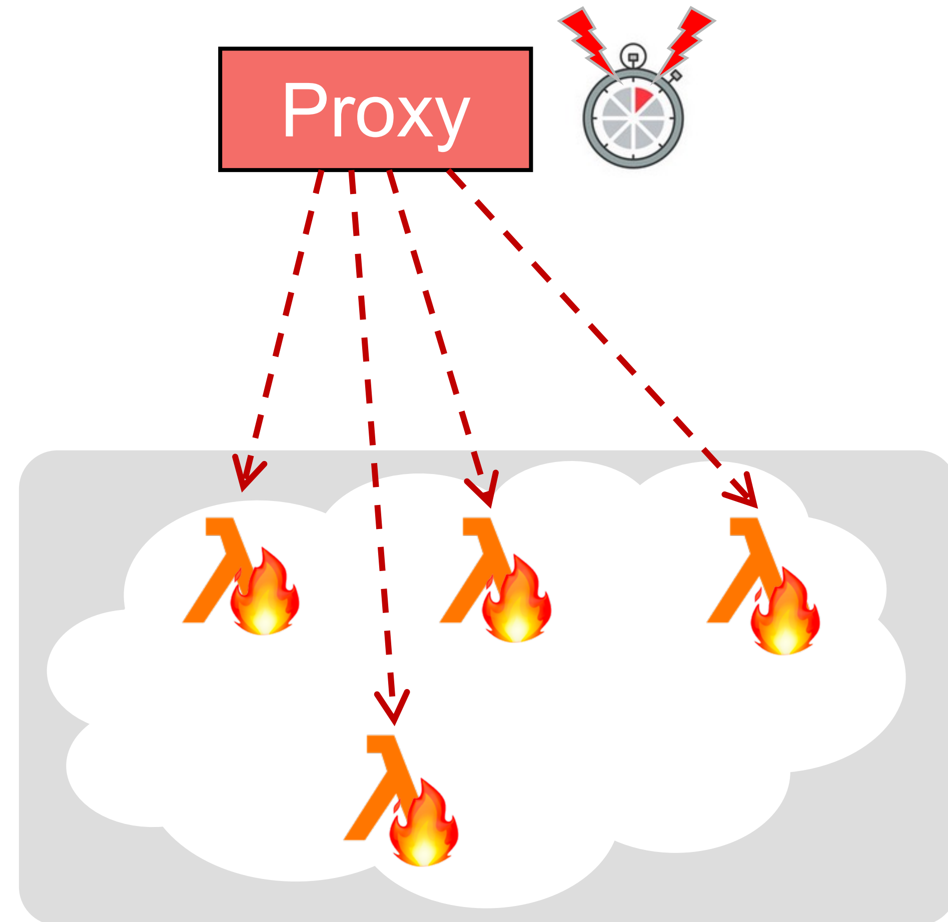




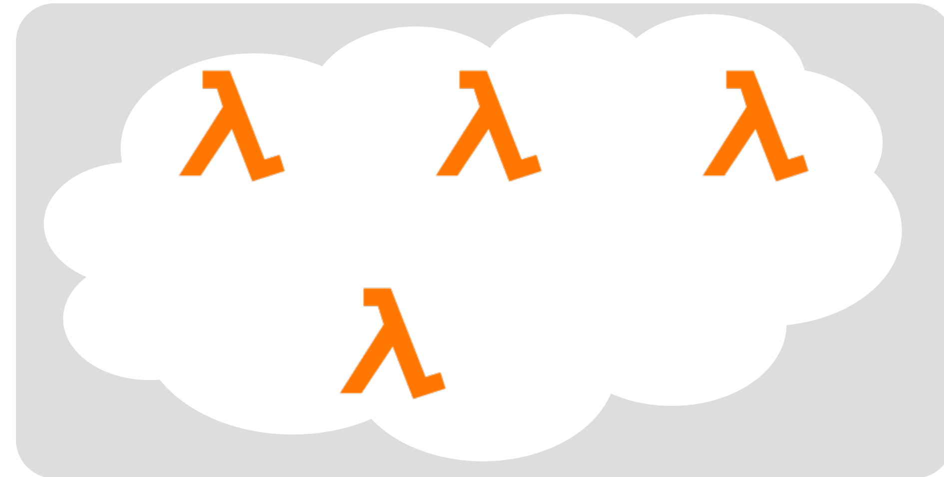
Maximizing data availability: Periodic warm-up

1. Lambda nodes are cached by AWS when not running

2. Proxy periodically invokes sleeping Lambda cache nodes to extend their lifespan



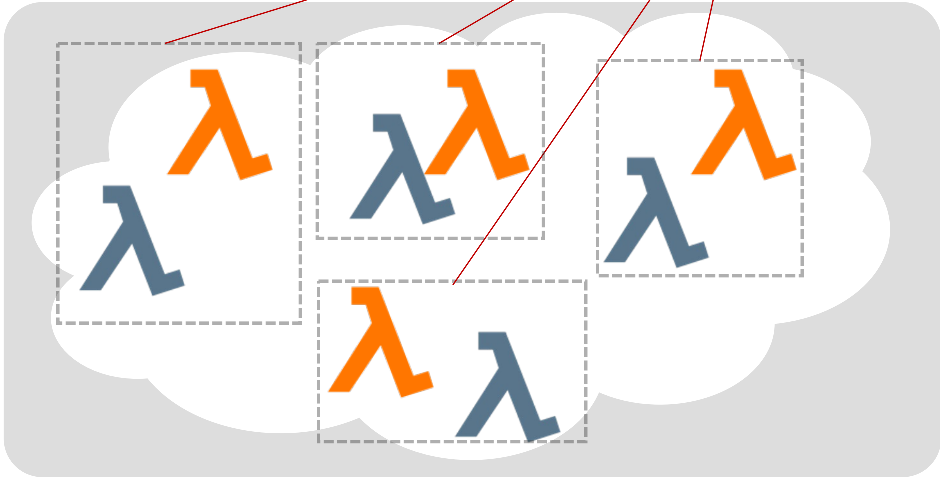
Maximizing data availability: Periodic backup



Maximizing data availability: Periodic backup

Proxy

Function deployment

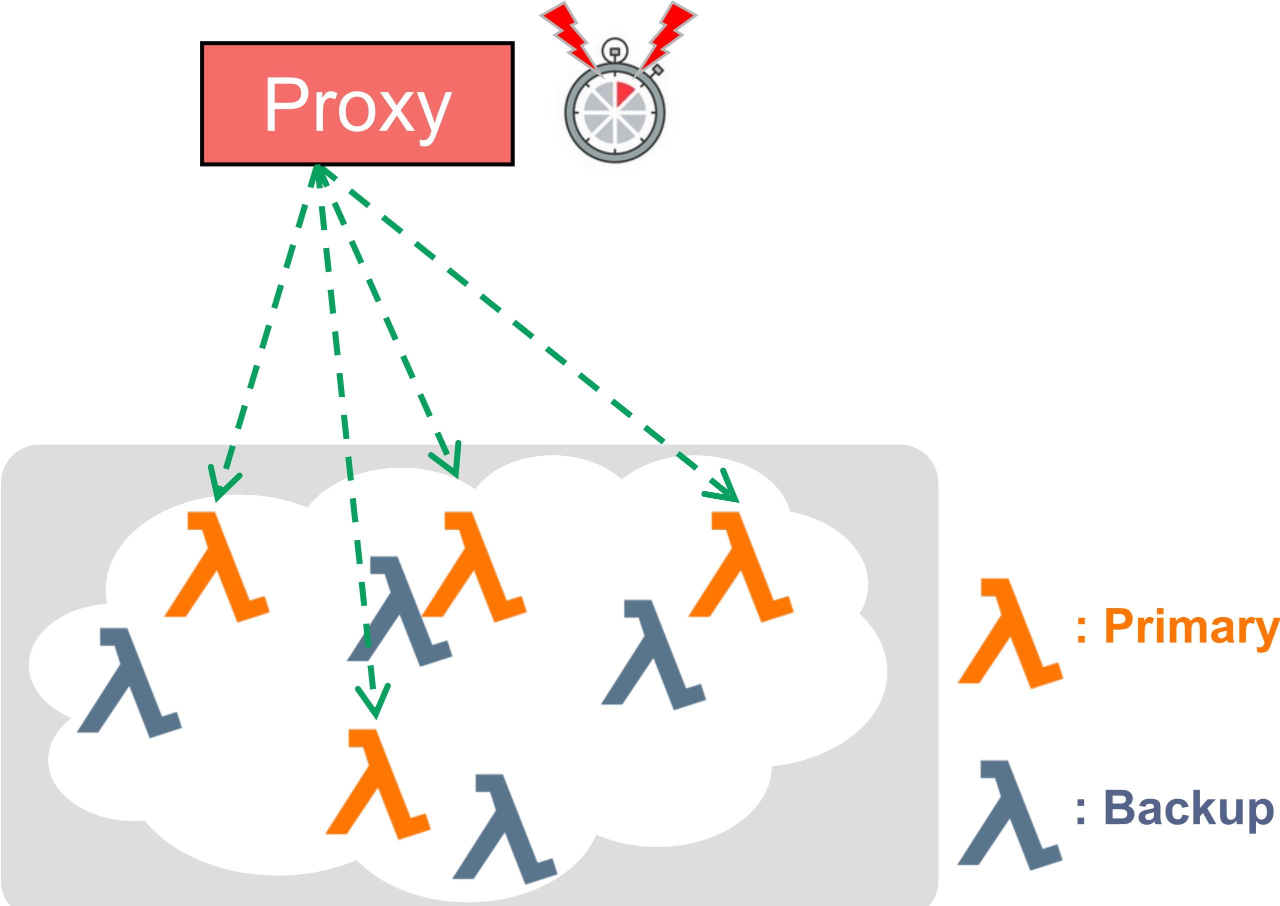


λ : Primary

λ : Backup

Maximizing data availability: Periodic backup

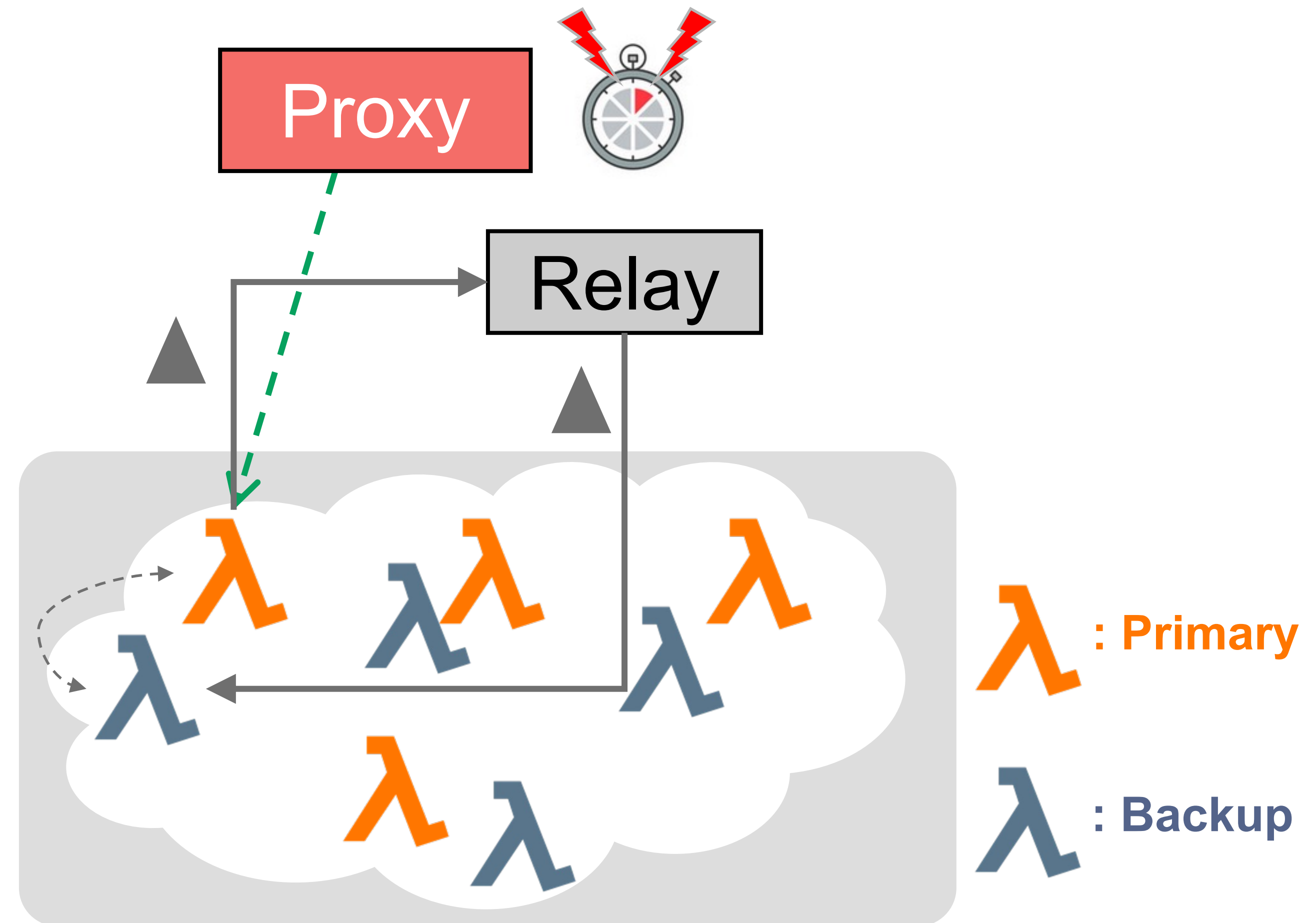
1. Proxy periodically sends out backup commands to Lambda cache nodes



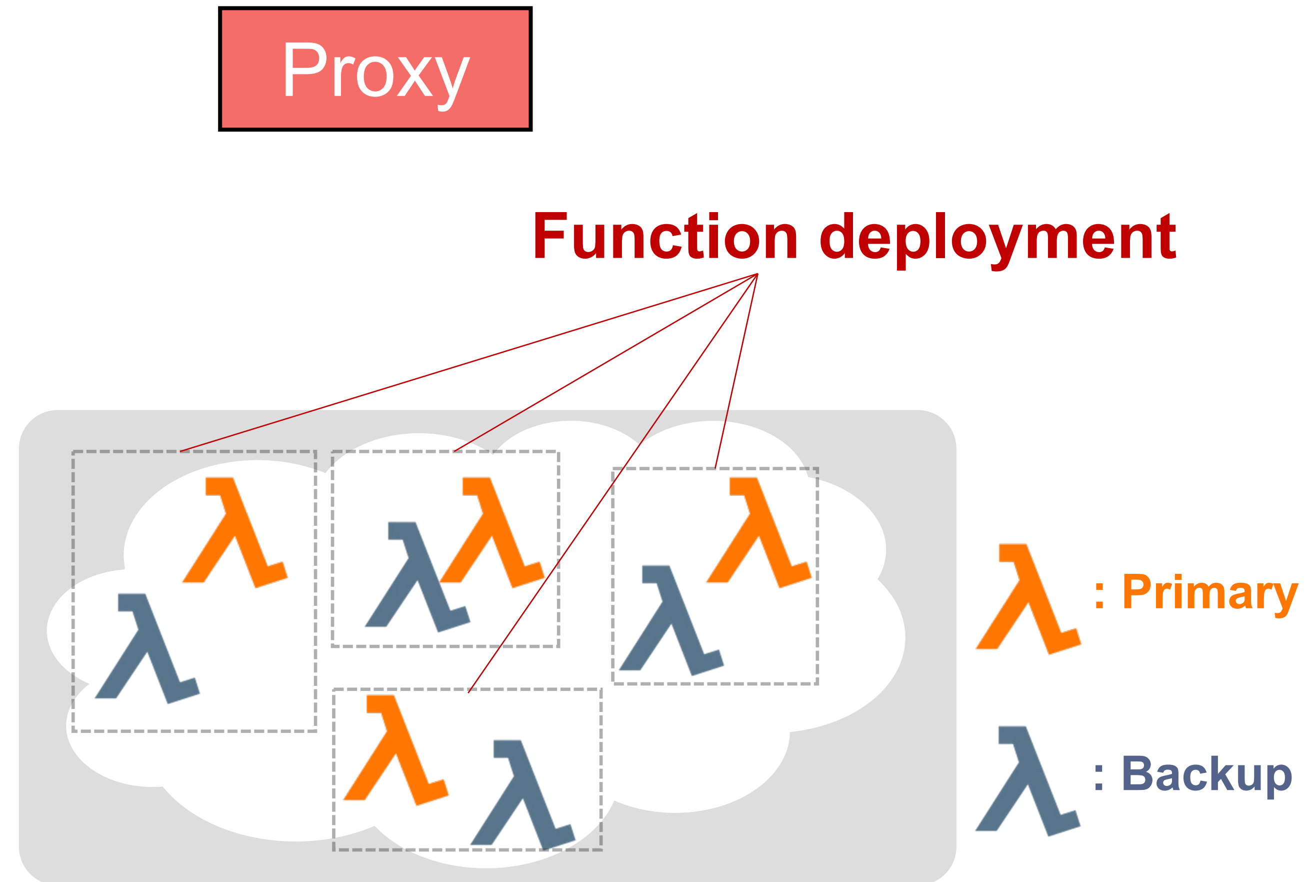
Maximizing data availability: Periodic backup

1. Proxy periodically sends out backup commands to Lambda cache nodes

2. Lambda node performs delta-sync with its peer replica
   - Source Lambda propagates delta-update ▲ to destination Lambda

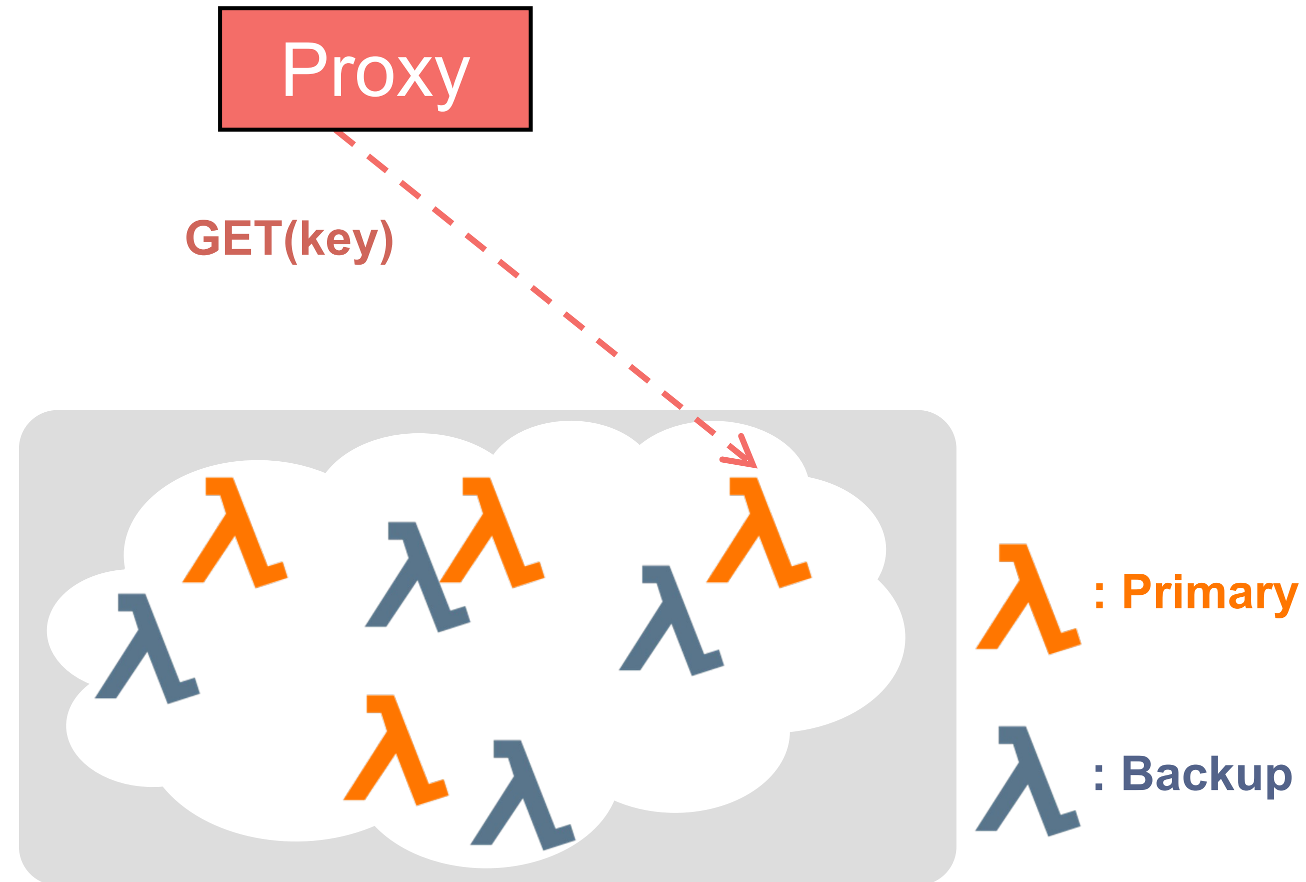


Seamless failover



Maximizing data availability: Seamless failover

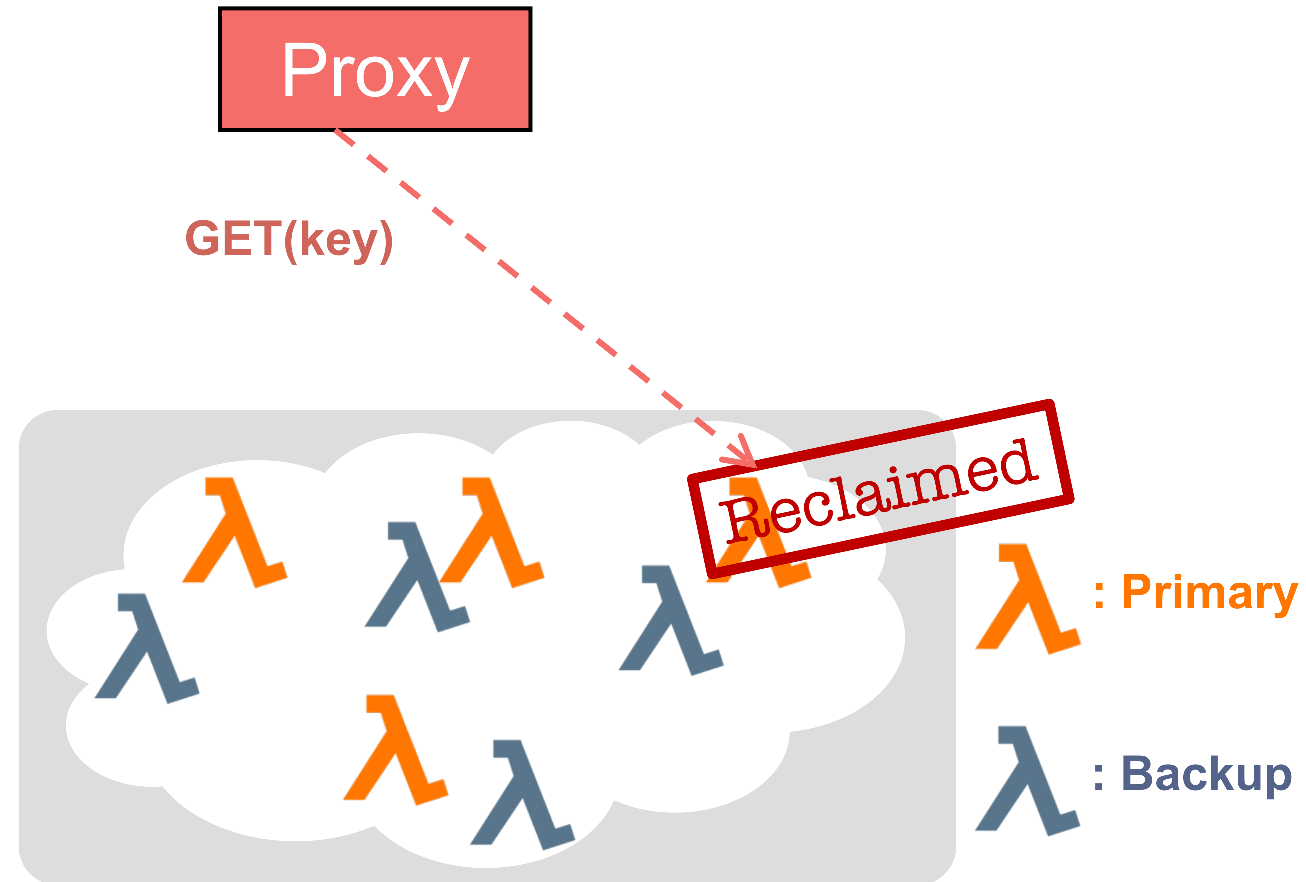
1. Proxy invokes a Lambda cache node with a GET request



Maximizing data availability: Seamless failover

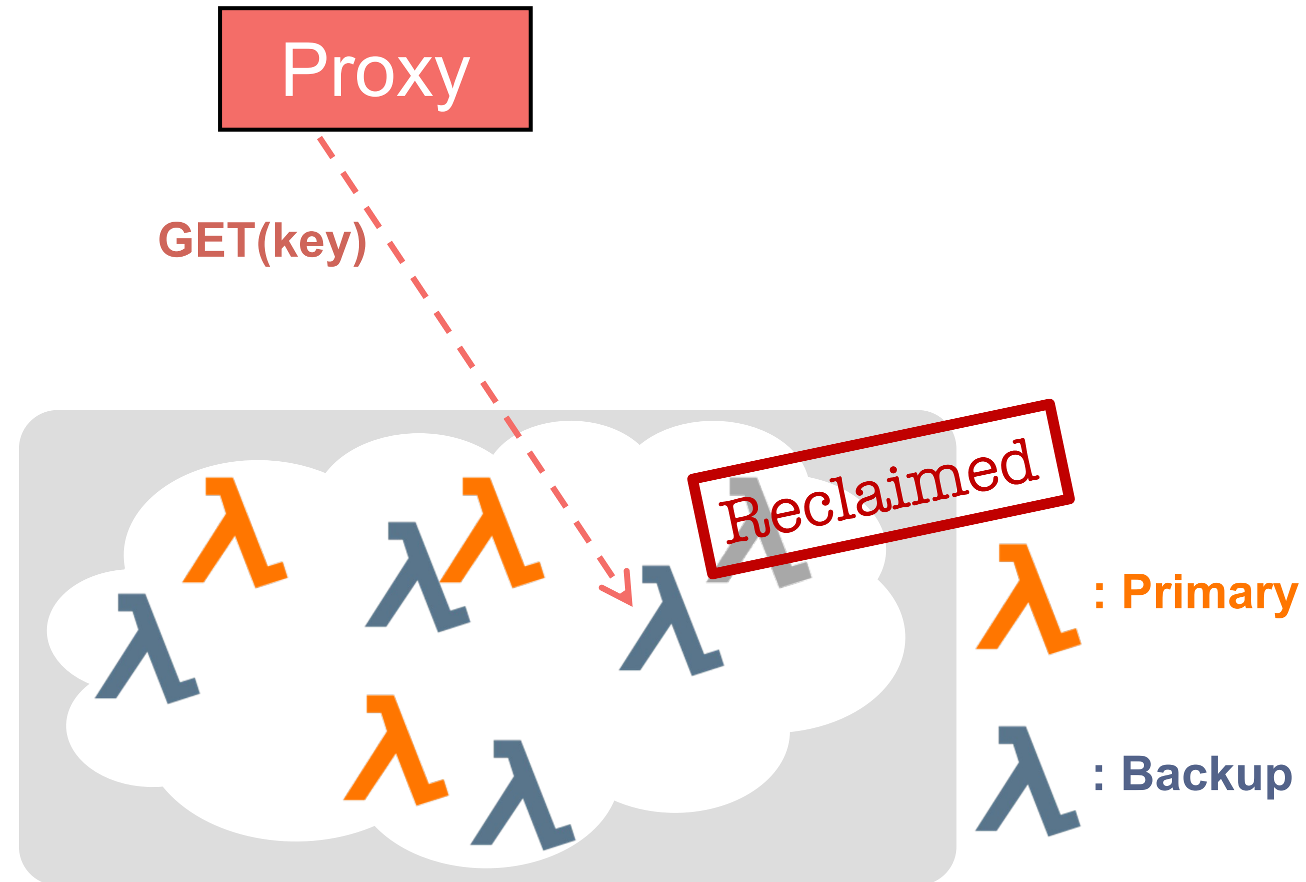
1. Proxy invokes a Lambda cache node with a GET request

2. Source Lambda gets reclaimed



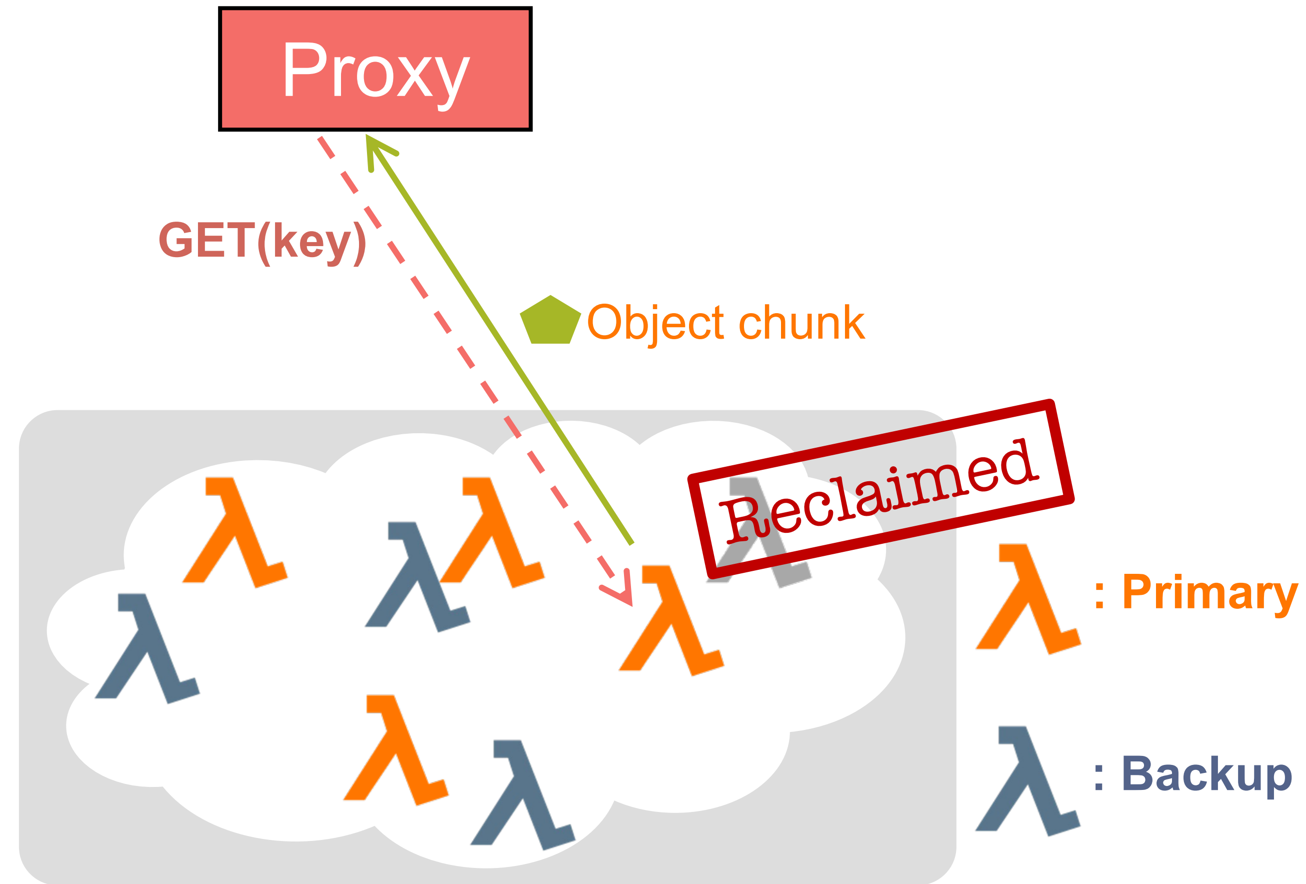
Maximizing data availability: Seamless failover

1. Proxy invokes a Lambda cache node with a GET request
2. Source Lambda gets reclaimed
3. The invocation request gets seamlessly redirected to the backup Lambda

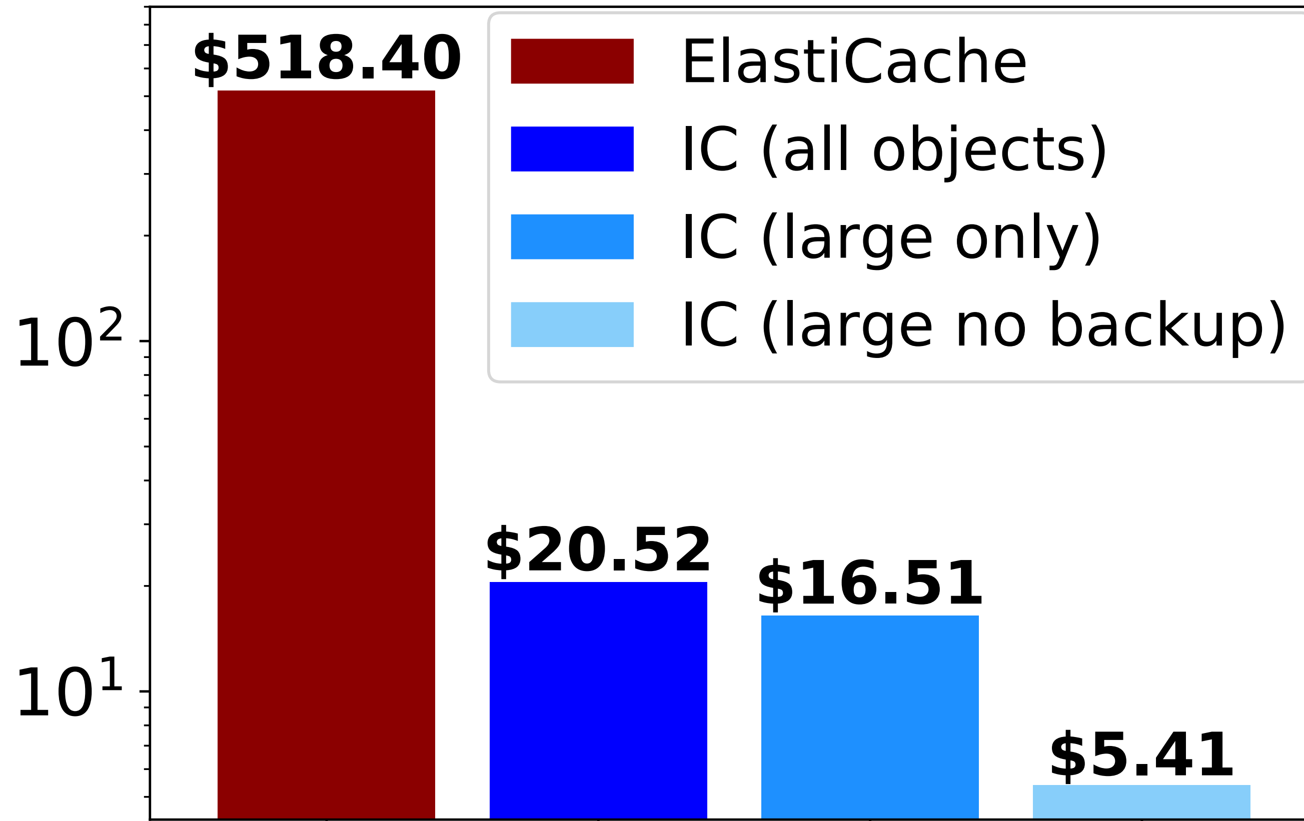


Maximizing data availability: Seamless failover

1. Proxy invokes a Lambda cache node with a GET request
2. Source Lambda gets reclaimed
3. The invocation request gets seamlessly redirected to the backup Lambda
   - Failover gets **automatically** done and the backup becomes the primary
   - By exploiting the **auto-scaling** feature of AWS Lambda



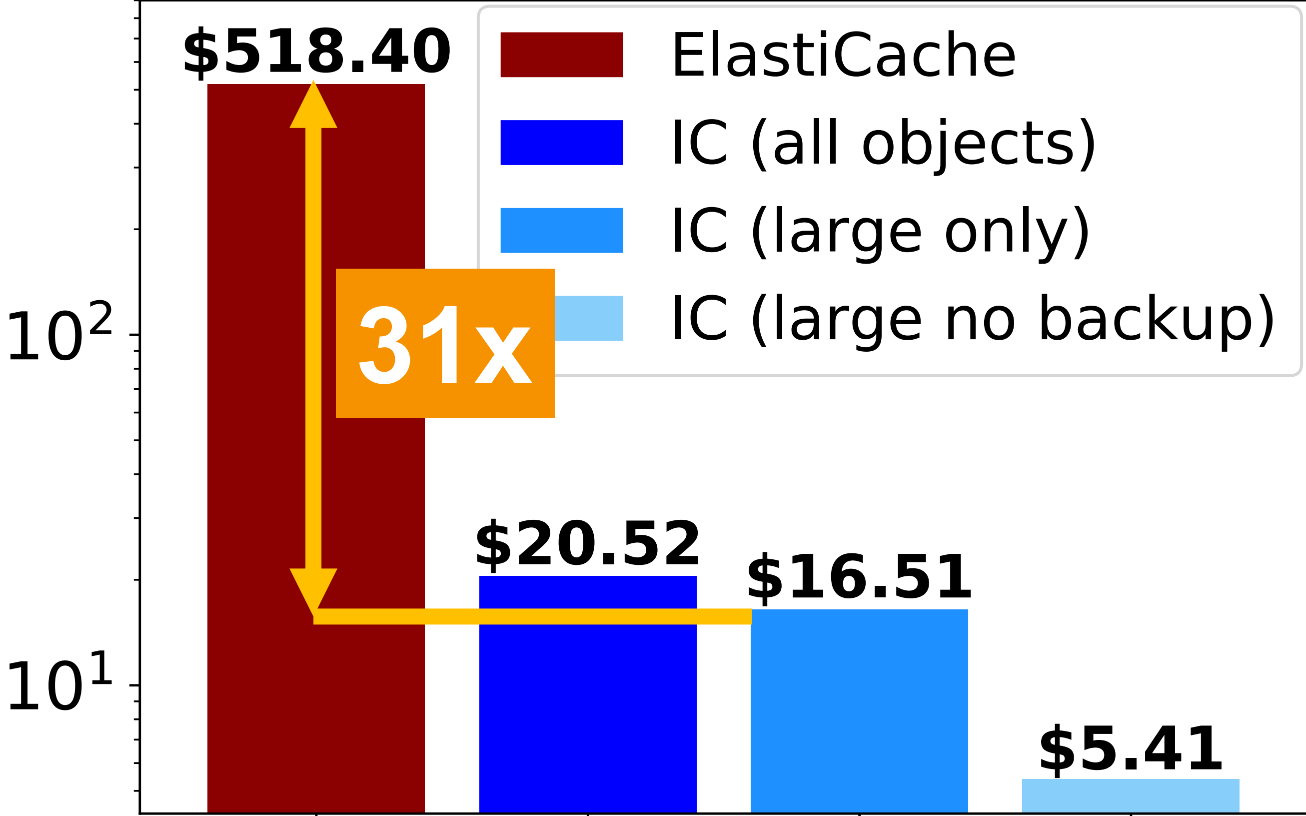
Cost effectiveness of InfiniCache



Workload setup

- All objects
- Large object only
  - Object larger than 10MB
- Large object w/o backup

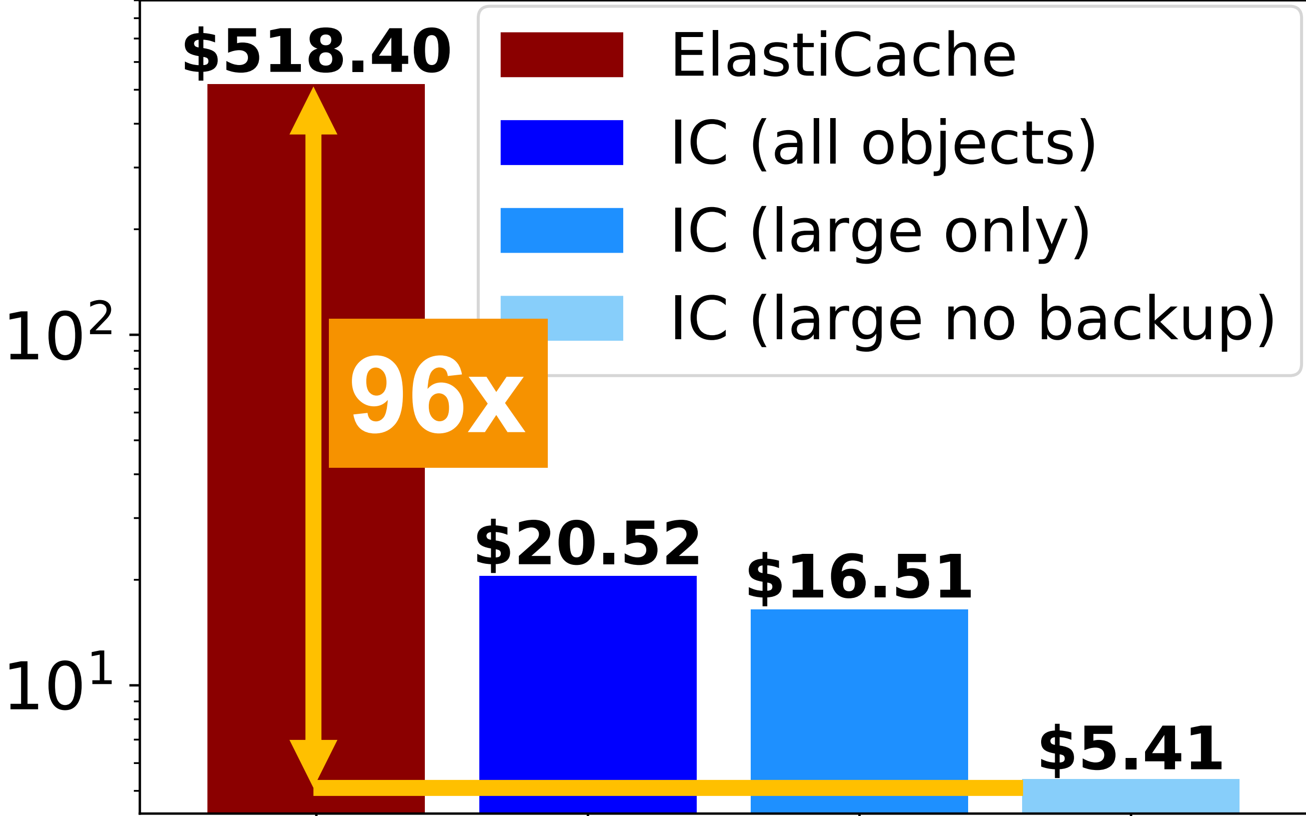
Cost effectiveness of InfiniCache



Workload setup

- All objects
- **Large object only**
  - **Object larger than 10MB**
- Large object w/o backup

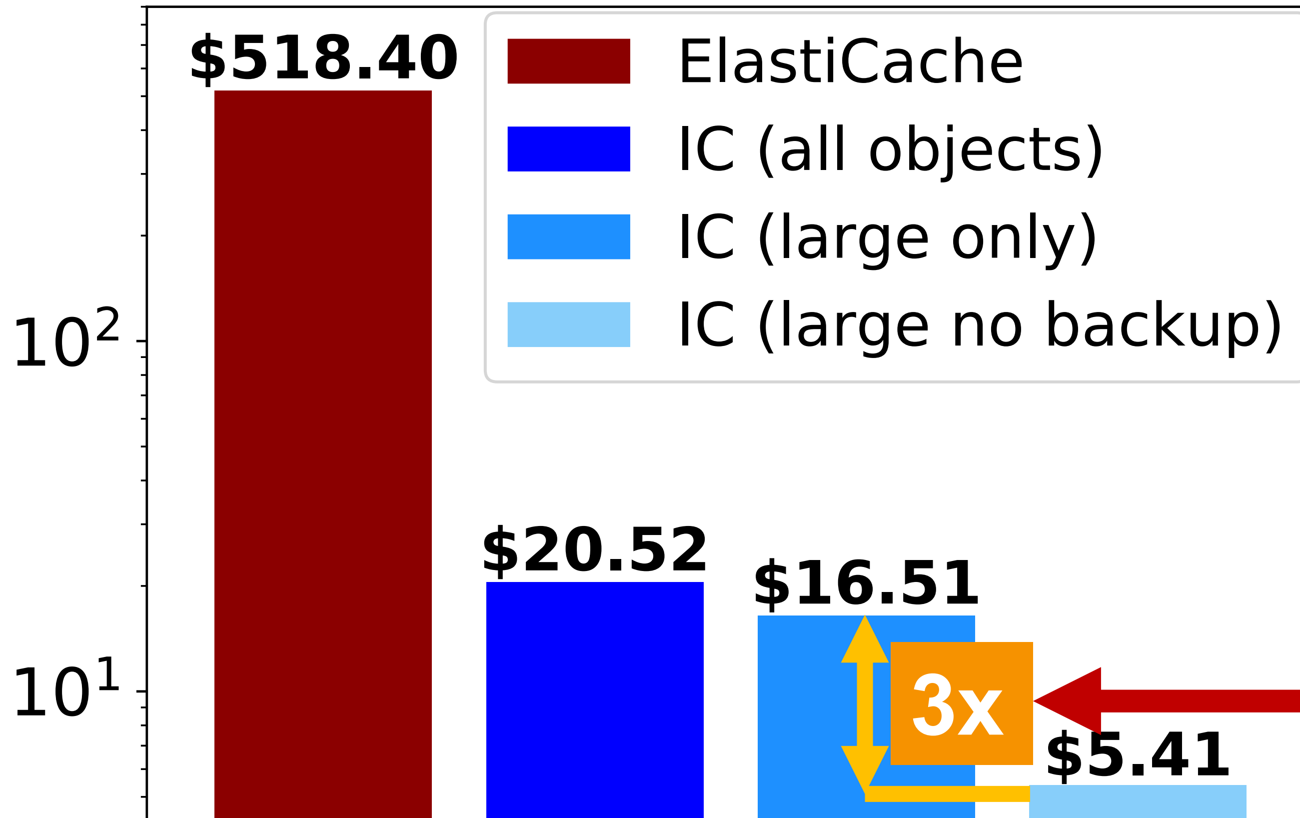
Cost effectiveness of InfiniCache



Workload setup

- All objects
- Large object only
  - Object larger than 10MB
- **Large object w/o backup**

Cost effectiveness of InfiniCache



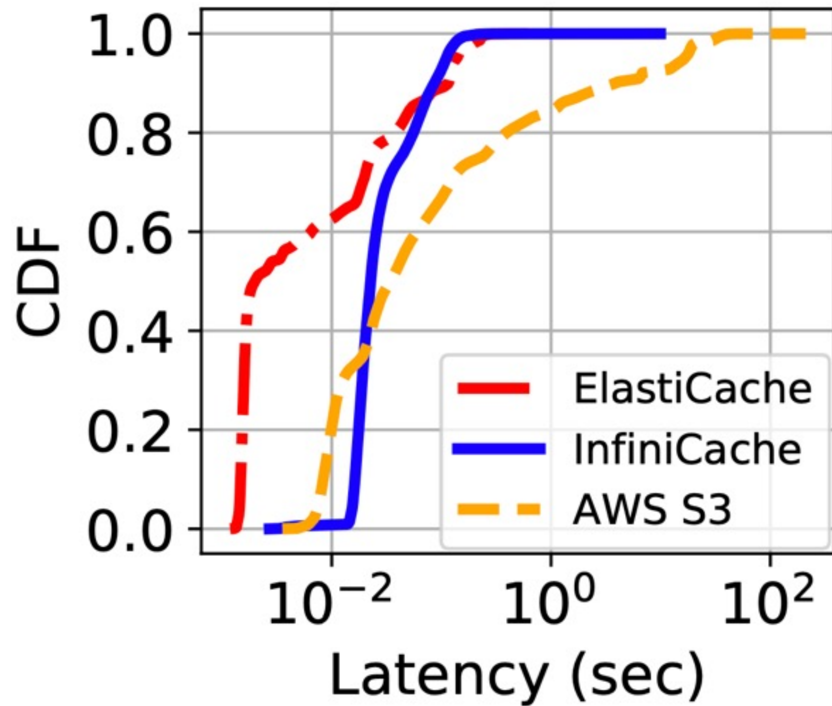
Workload setup

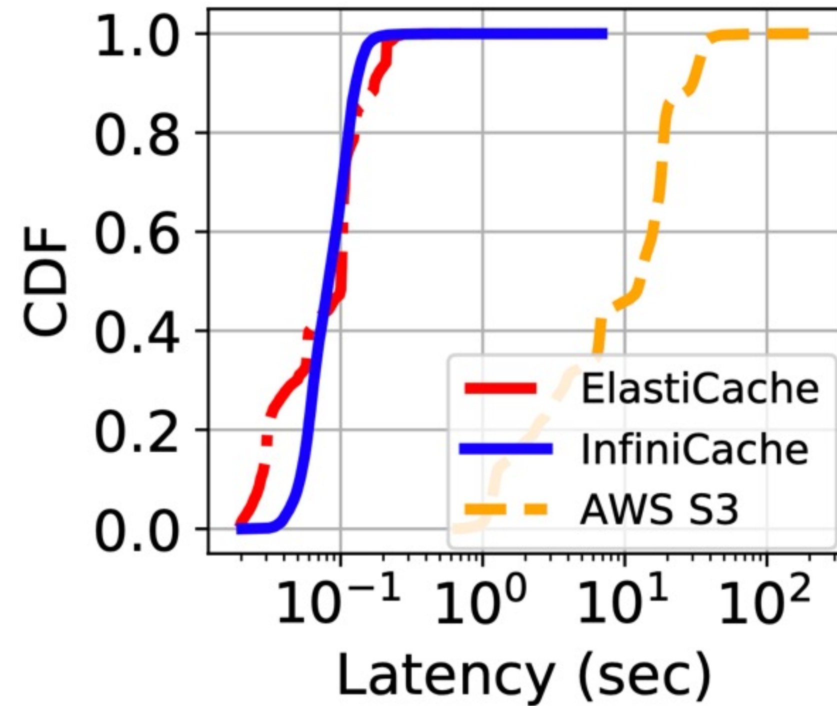
- All objects
- Large object only
  - Object larger than 10MB
- **Large object w/o backup**

Hit ratio and $$ cost tradeoff

Workload	ElastiCache	InfiniCache	InfiniCache w/o backup
All objects	67.9%	64.7%	---
Large object only	65.9%	63.6%	56.1%

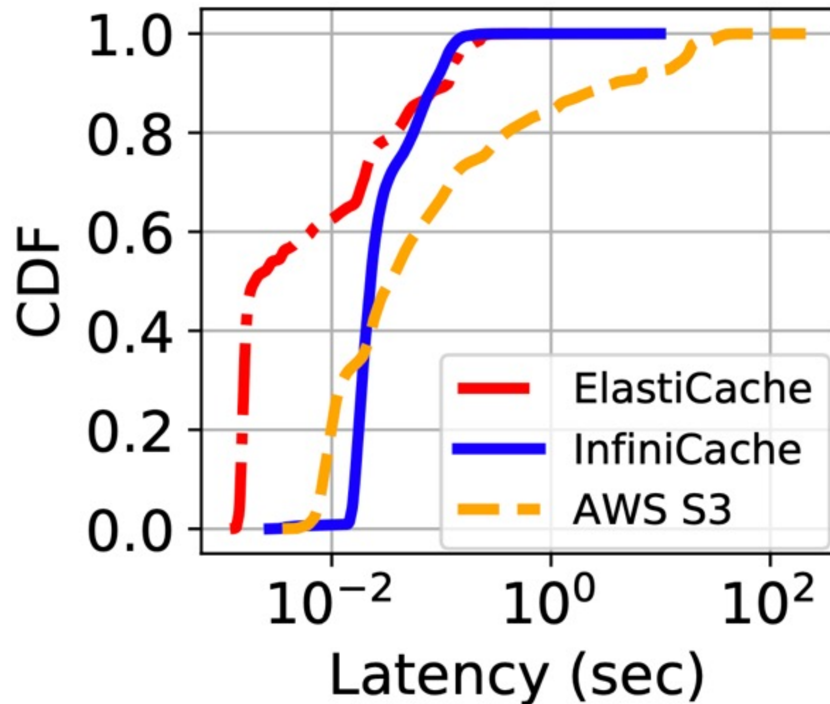
Performance of InfiniCache



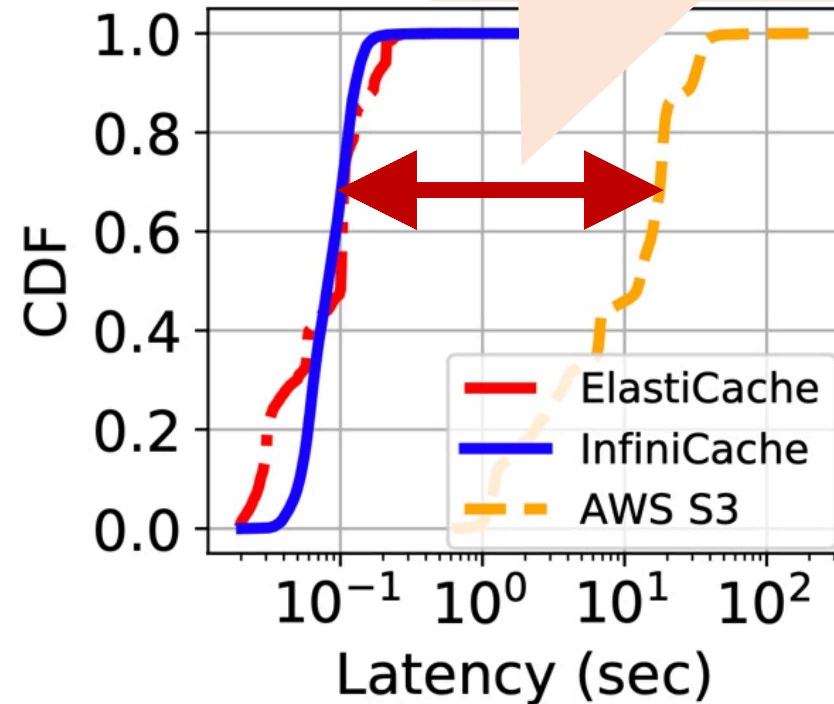


Performance of InfiniCache

> 100 times improvement



All objects



Large objects only

Exploiting FaaS for fun and profit

Q: Is FaaS well suited for stateful applications?

A: Not exactly. It requires research to make FaaS embrace state.

Q: Is FaaS poorly suited for stateful applications?

A: Not really! We need to rethink the application+system design.

Q: I'm still not convinced… Why bother?

A: More **fun** (ease-of-use, zero server management)

More **profit** (pay-per-use, no idle VMs anymore)