

Exploiting Serverless Function to Build a Cost-effective Cloud Storage

CS 4740: Cloud Computing

Fall 2024

Lecture 14d

Yue Cheng



Rule-breaking approach

* Quoted from “*A Rule-Breaking Approach to Research*”,
by Todd Austin, MICRO'24.

- A rule-breaking approach is effective and exciting
 - Identify a rule no one breaks
 - Invent a way to break that rule
 - See what happens!
- You will often find yourself in fertile ground
 - The “rules” are typically learned early or based on “conventional wisdom”
 - The “rules” create dogma that hide opportunity
- 50% will be intrigued with your crazy idea
- 50% will think your crazy idea will never work
- Embrace the pushback, it will inform and sharpen

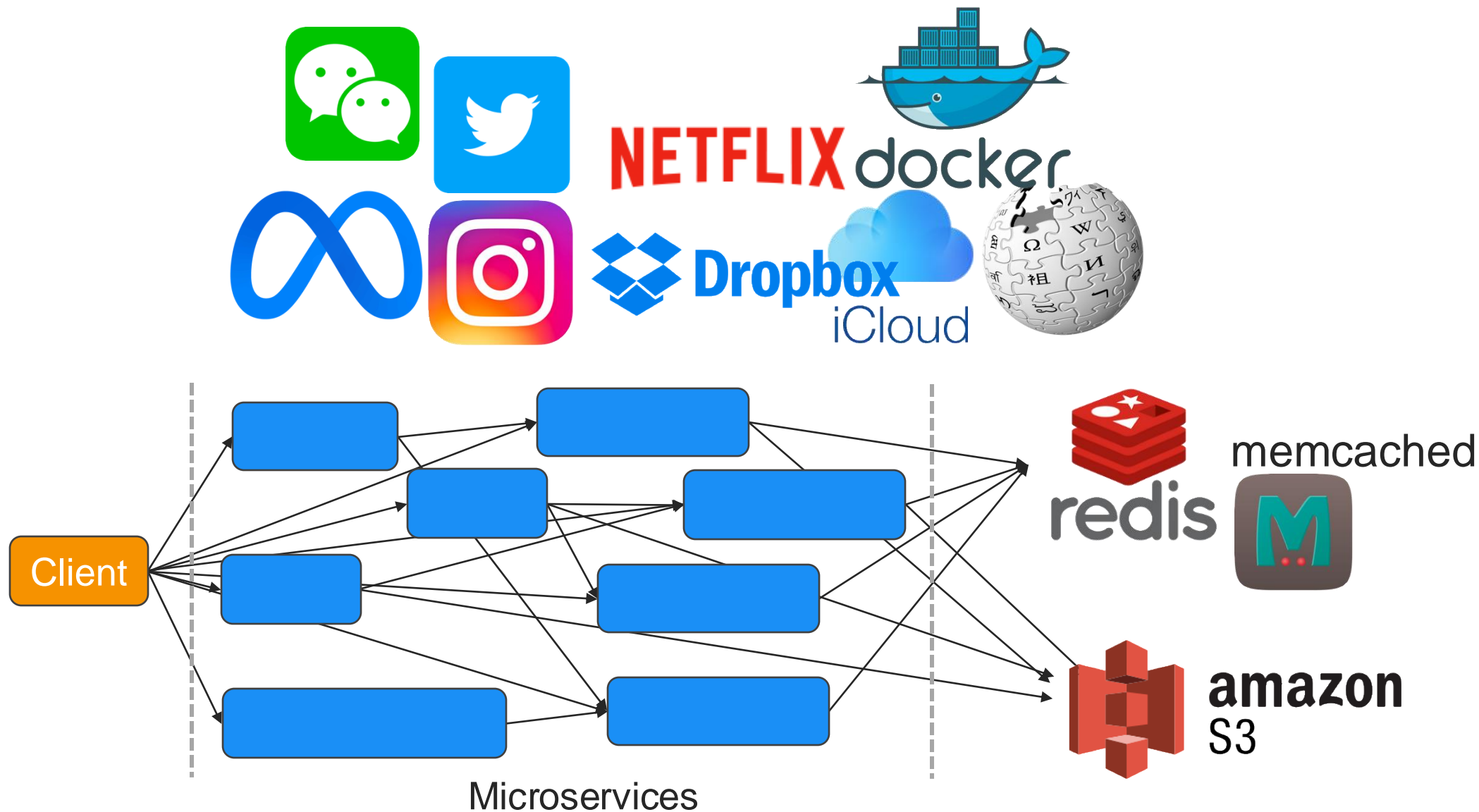


Breaking rules in serverless

- **Rule:** Serverless functions are stateless and can never work as storage
- **Rule-breaking idea:** Use functions as a brand-new storage medium to build a first-of-its-kind cloud storage system
 - Exploiting provider's function caching to retain data between func invocations
 - Erasure coding + replication to improve availability and performance
 - Reasonable performance+availability while being **extremely cost-effective** for not-too-busy storage workloads
 - Case study: **IBM Docker registry**



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



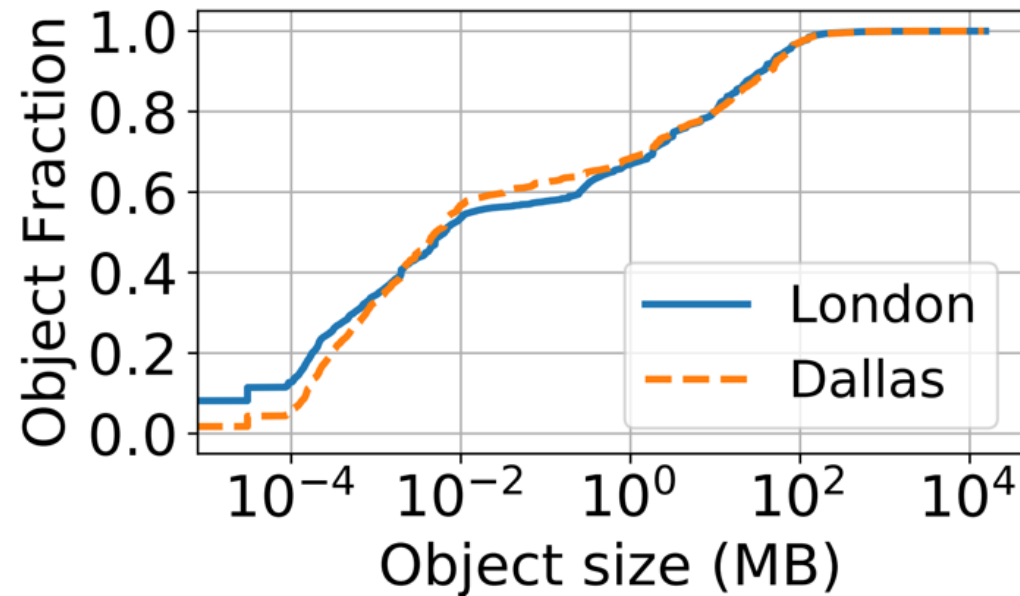
**IBM Cloud
Container Registry**

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

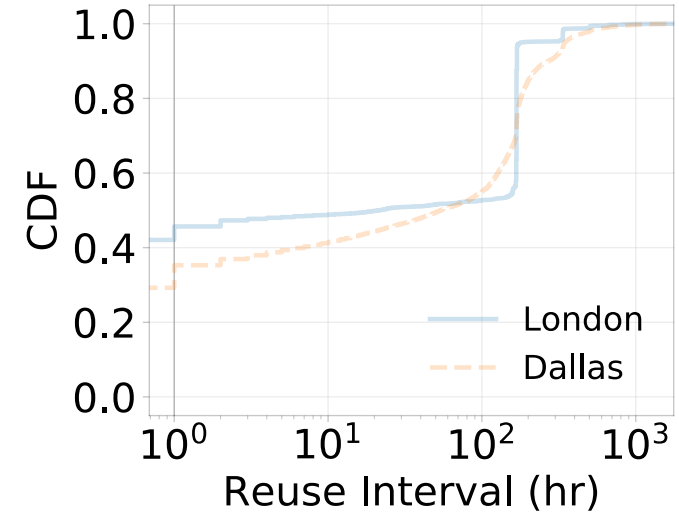
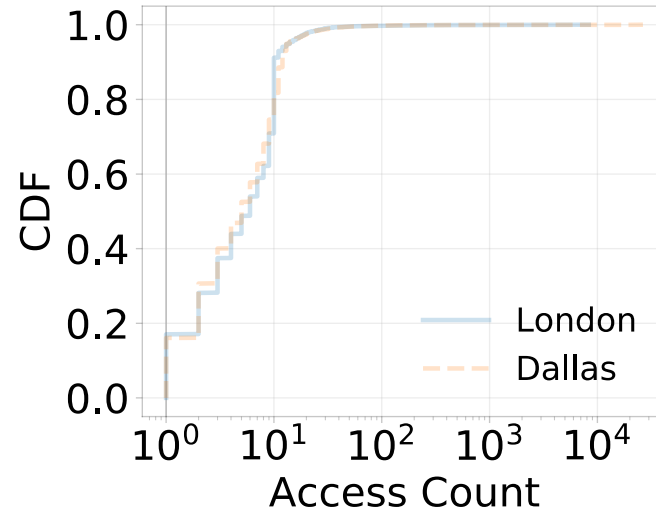


Extreme variability in object sizes:

- 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

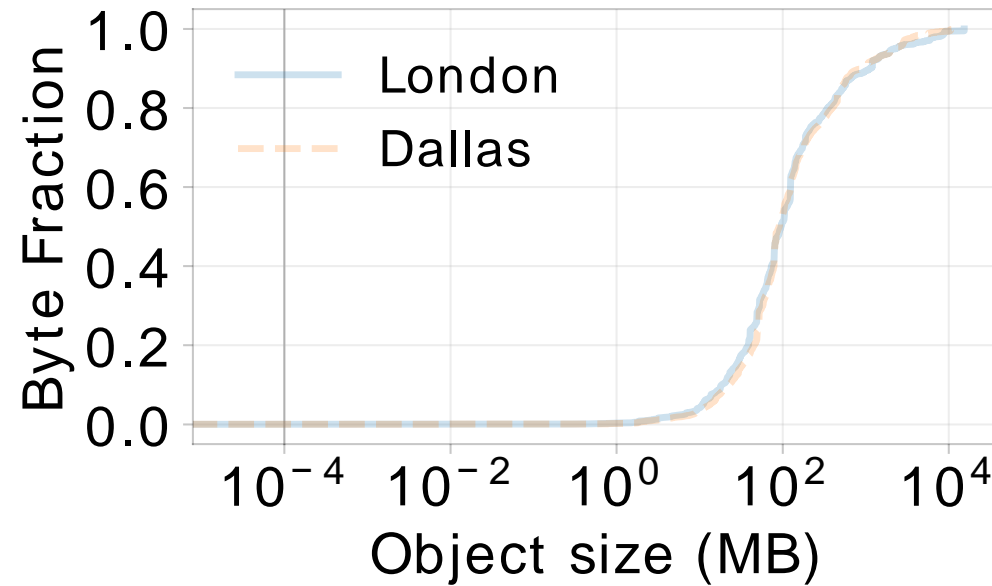


Caching large objects is beneficial:

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

Example app: IBM Cloud Container Registry

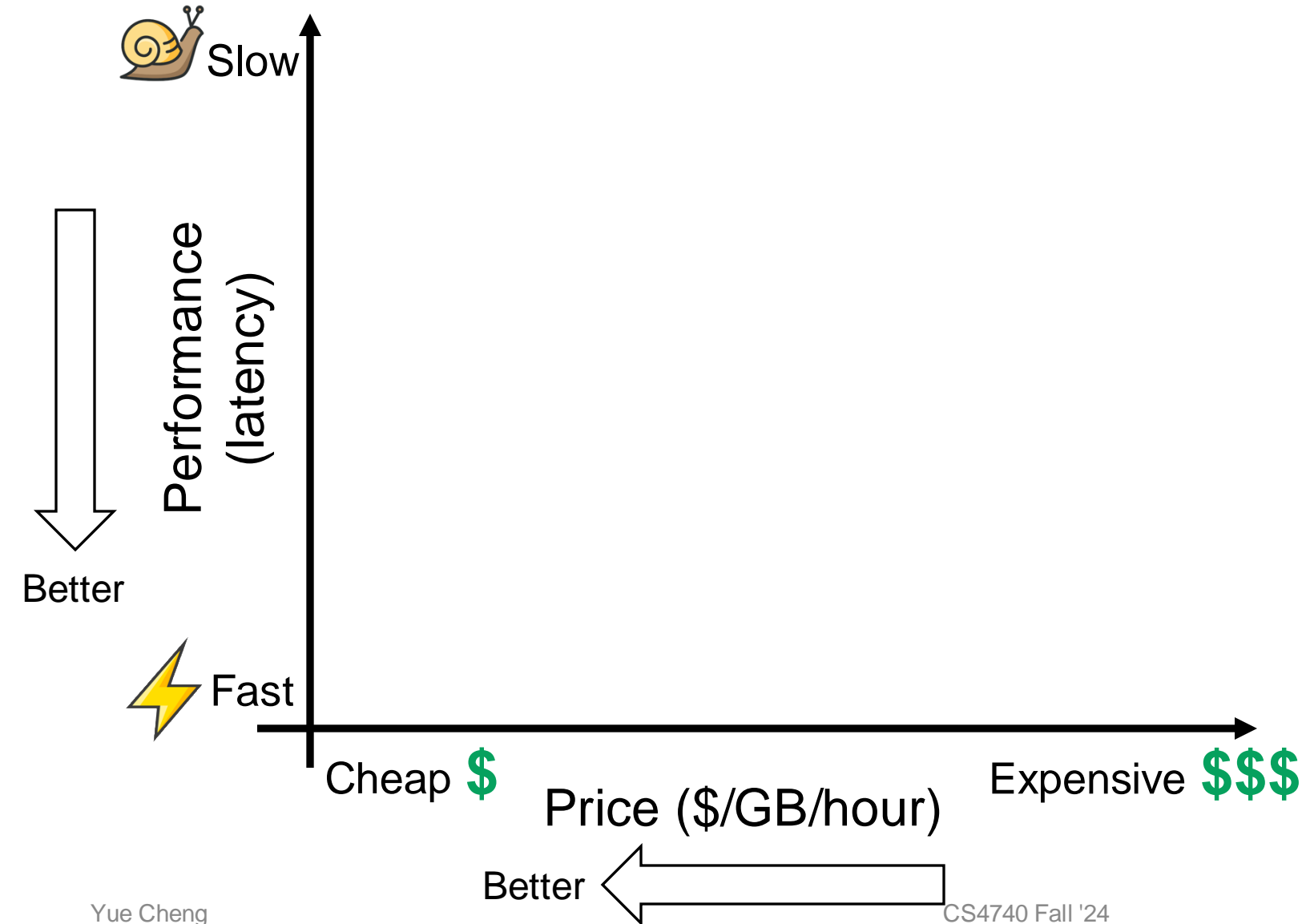
- Object size distribution
- Large objects' reuse patterns
- **Storage footprint**



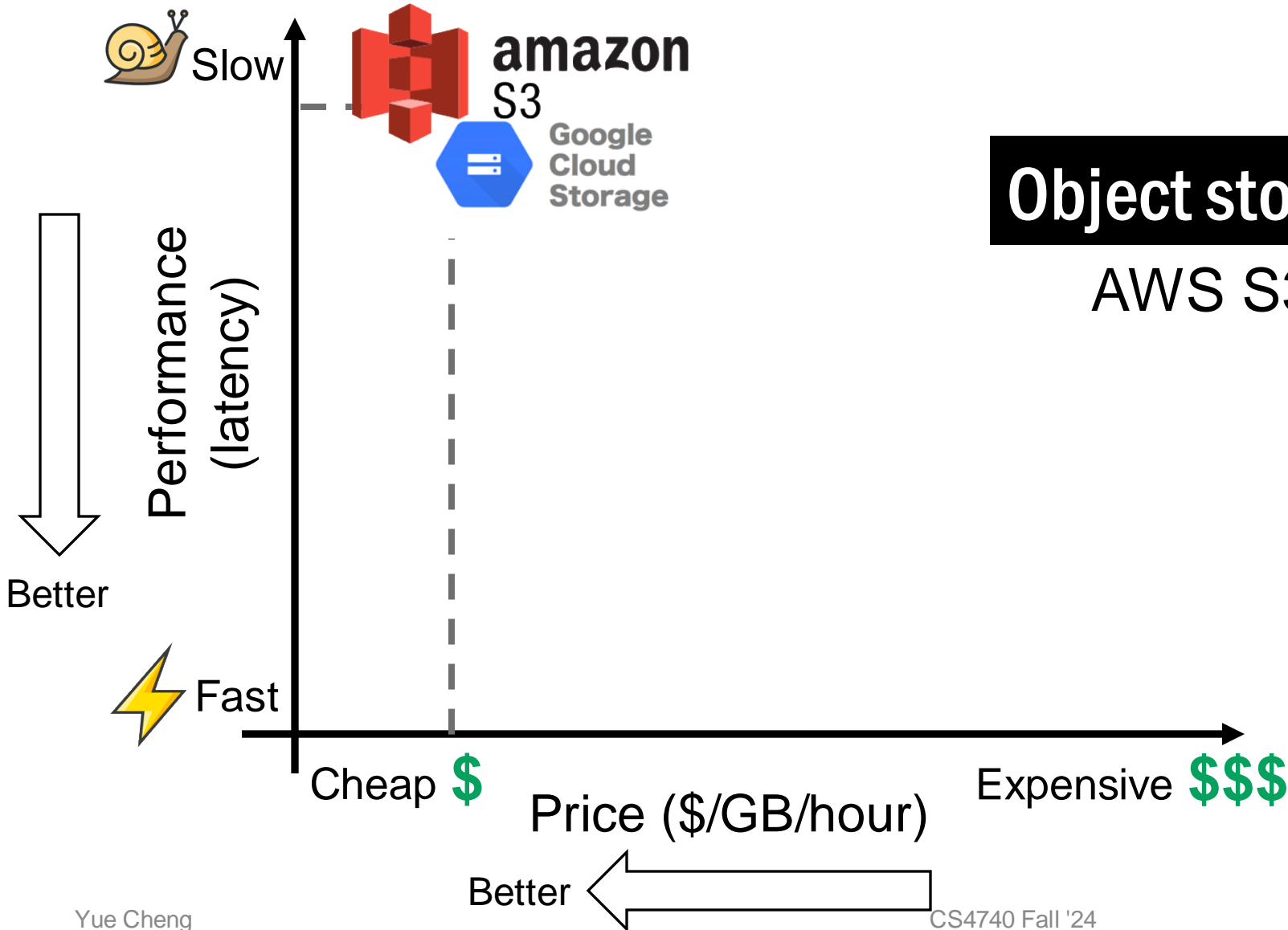
Extreme tension between small and large objects:

- Large objects ($>10\text{MB}$) occupy **95%** storage footprint

Today's cloud storage landscape



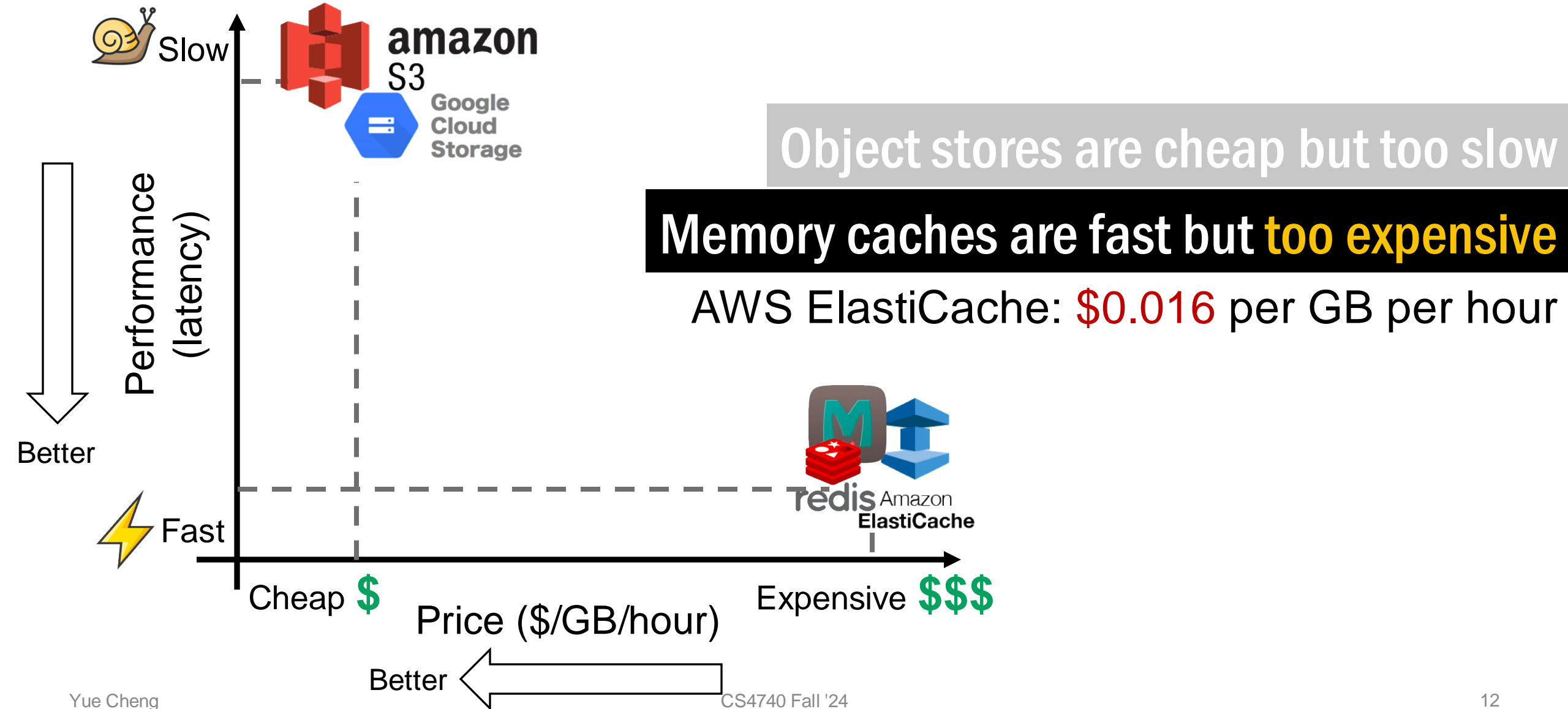
Today's cloud storage landscape



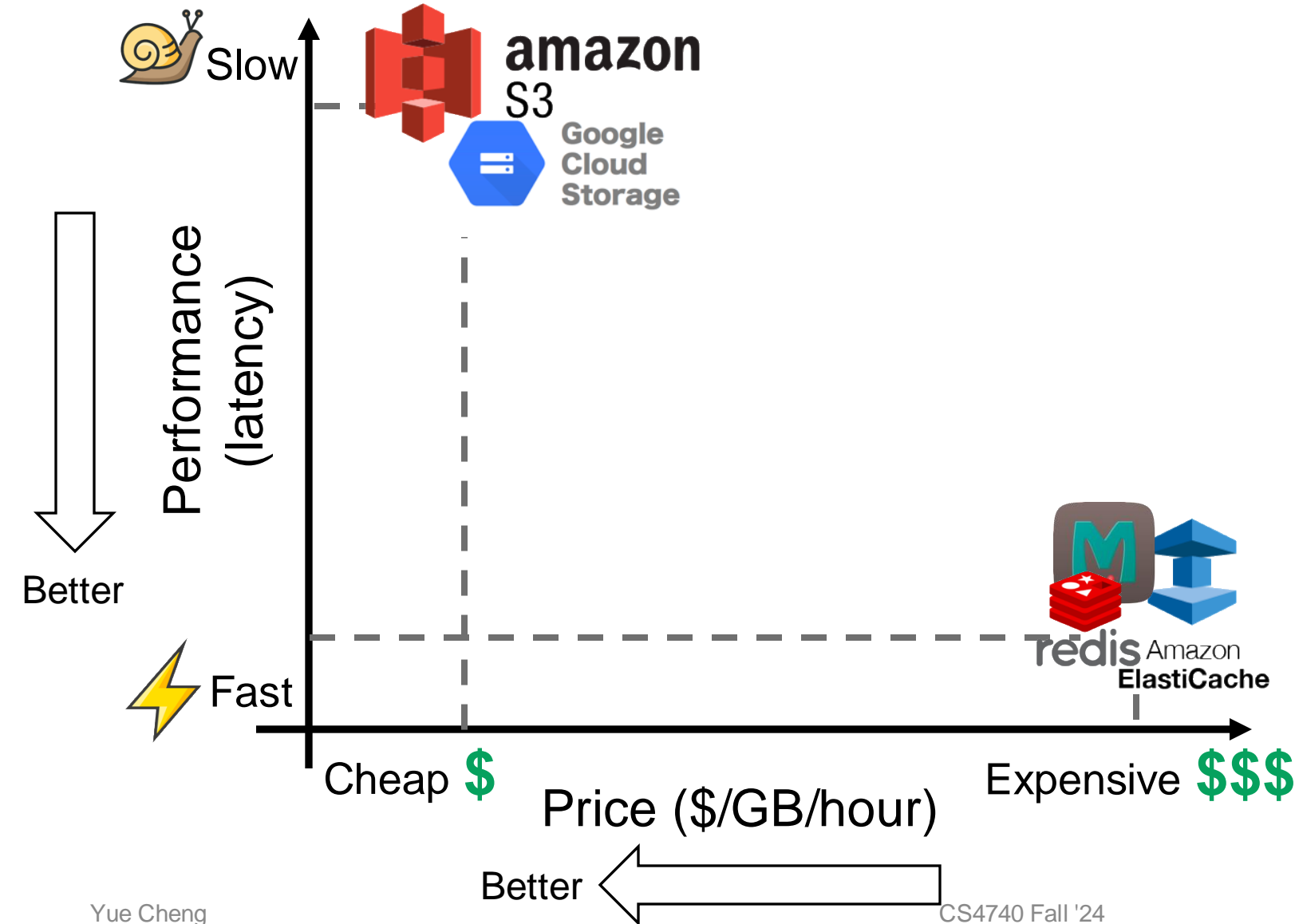
Object stores are cheap but **too slow**

AWS S3: **\$0.023** per GB per month

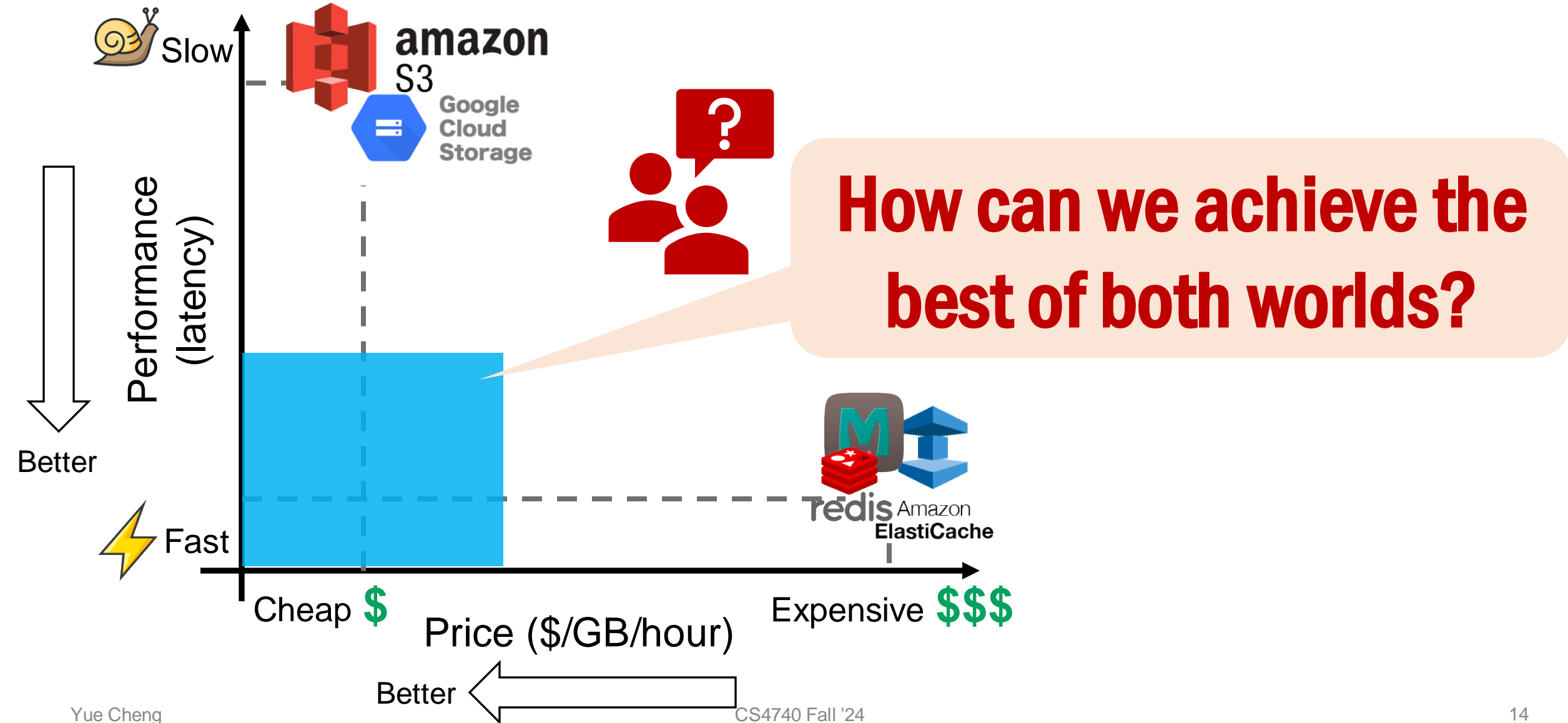
Today's cloud storage landscape



- **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:

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

- ⚠ Serverless functions could be reclaimed any time
- ⚠ In-memory state is lost



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

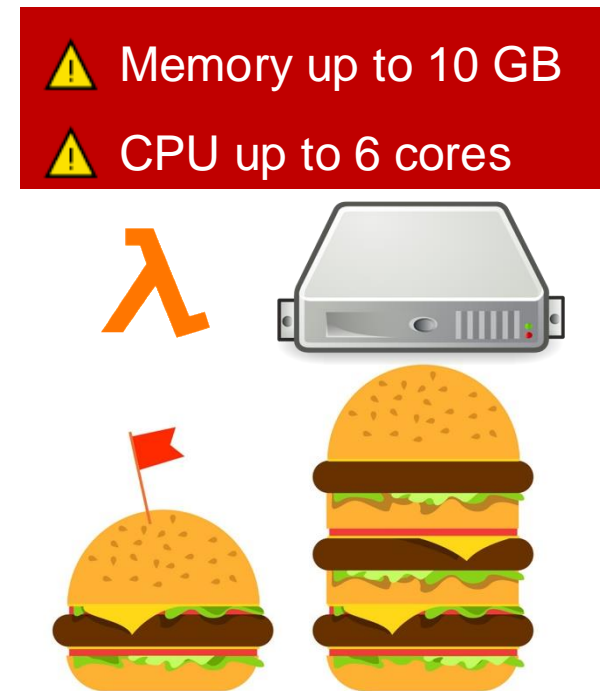


Challenges to build a memory cache using serverless functions

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:

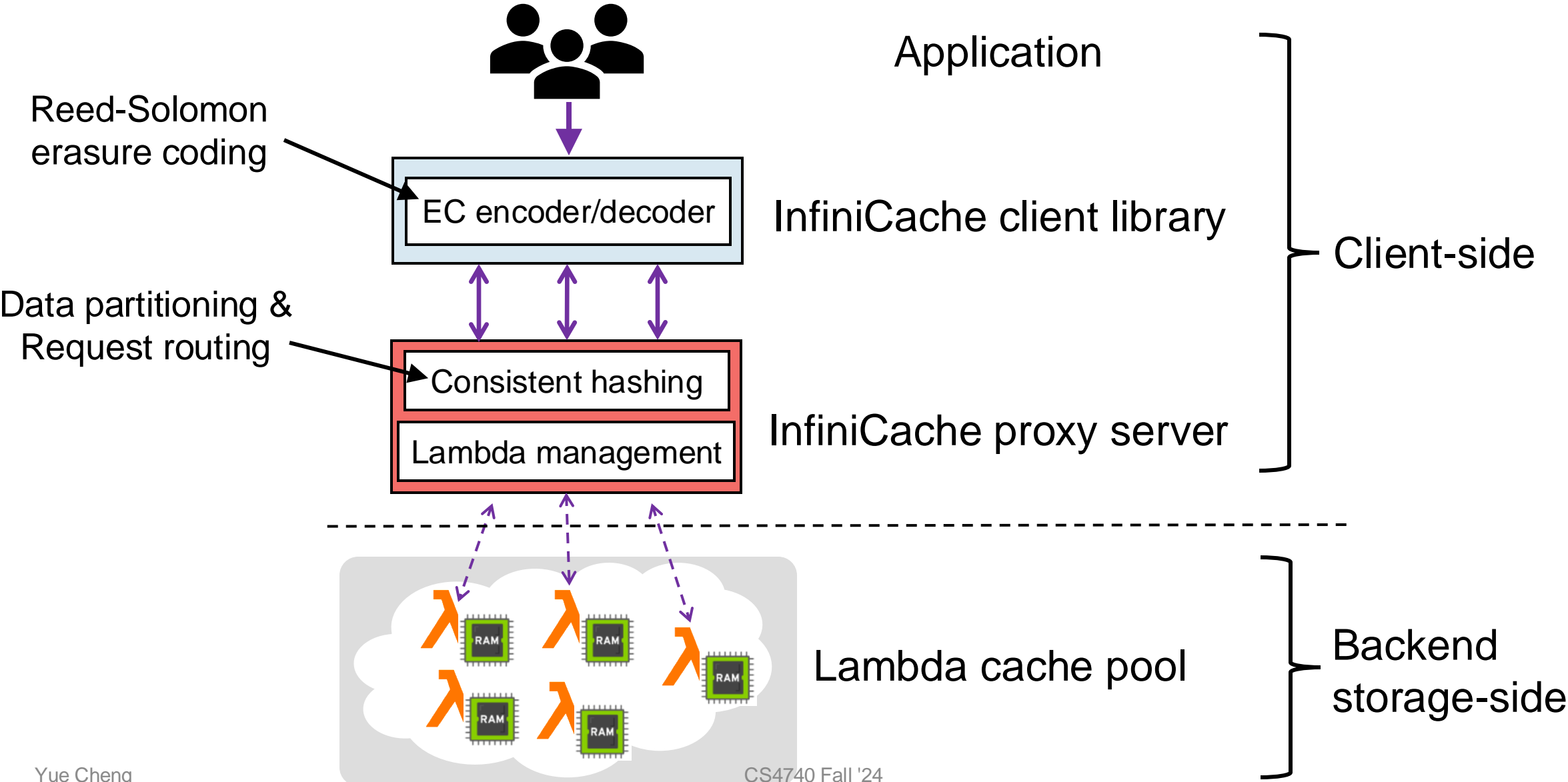
- **No** guaranteed data availability
- **Banned** inbound network
- **Limited** per-function resources



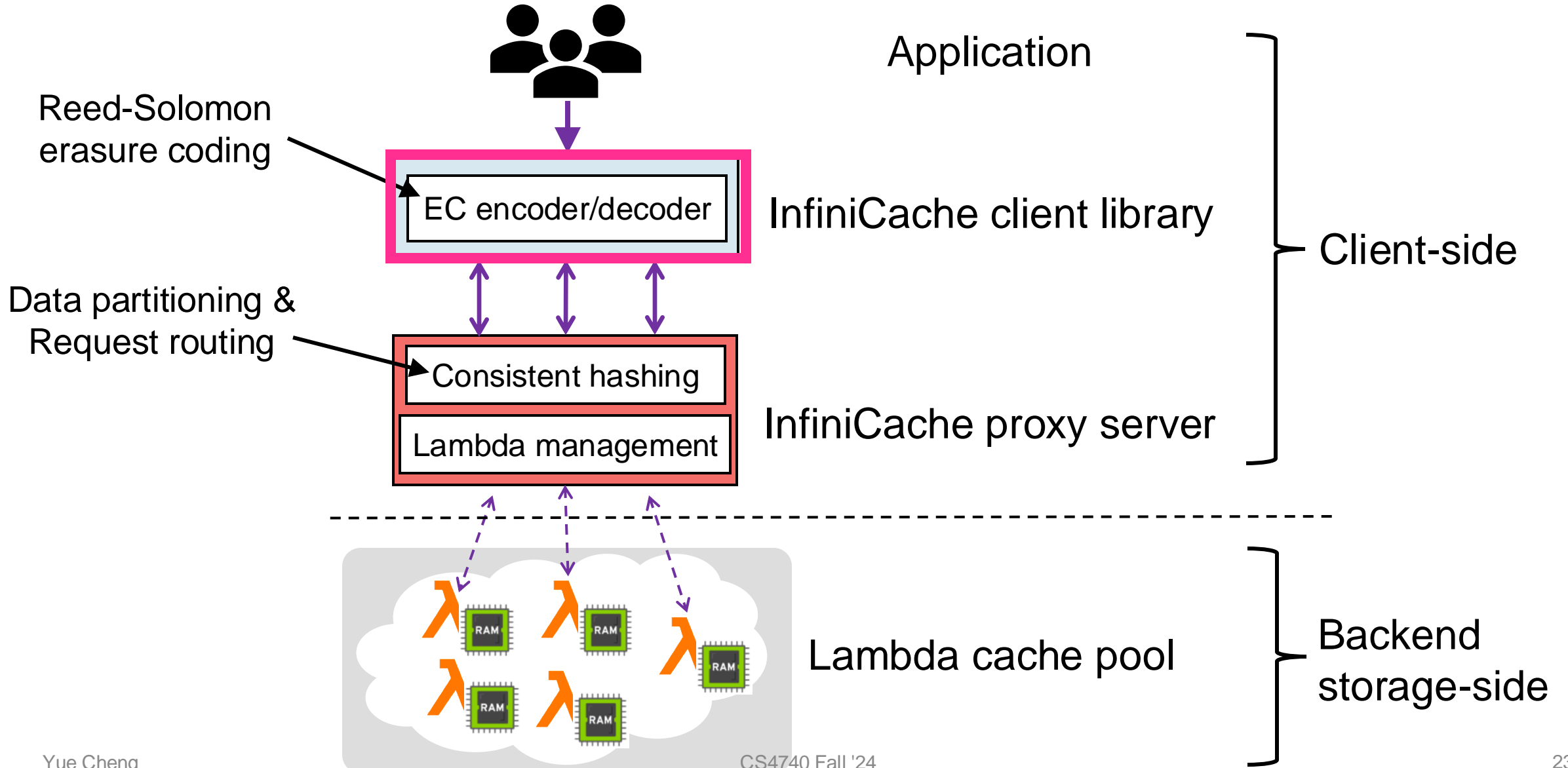
InfiniCache: The first memory cache built atop FaaS

- 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



Let's look at RAID and Reed-Solomon EC first



RAID: Redundant Array of Inexpensive Disks

Wish List for a Disk

- Wish it to be **faster**
 - I/O is always the performance bottleneck

Wish List for a Disk

- Wish it to be **faster**
 - I/O is always the performance bottleneck
- Wish it to be **larger**
 - More and more data needs to be stored

Wish List for a Disk

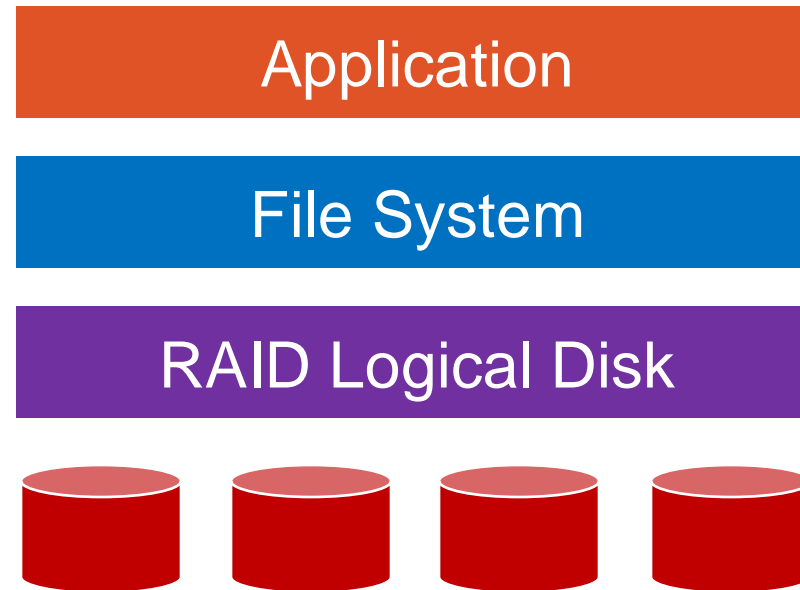
- Wish it to be **faster**
 - I/O is always the performance bottleneck
- Wish it to be **larger**
 - More and more data needs to be stored
- Wish it to be **more reliable**
 - We don't want our valuable data to be gone

Only One Disk?

- Sometimes we want many disks
 - For higher performance
 - For larger capacity
 - For better reliability
- **Challenge:** Most file systems work on only one disk

Solution: RAID

RAID: Redundant Array of Inexpensive Disks



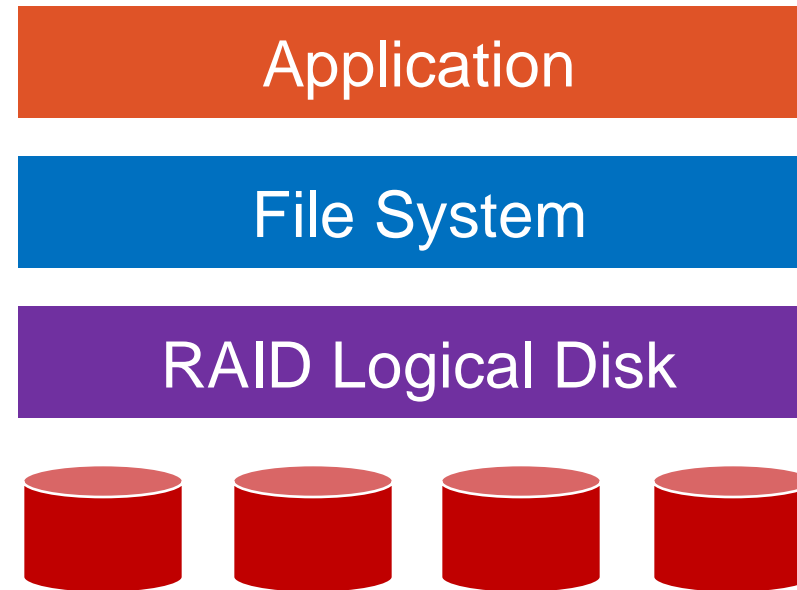
Build a logical disk from many physical disks

Solution: RAID

RAID: Redundant Array of Inexpensive Disks

RAID is

- Transparent
- Deployable



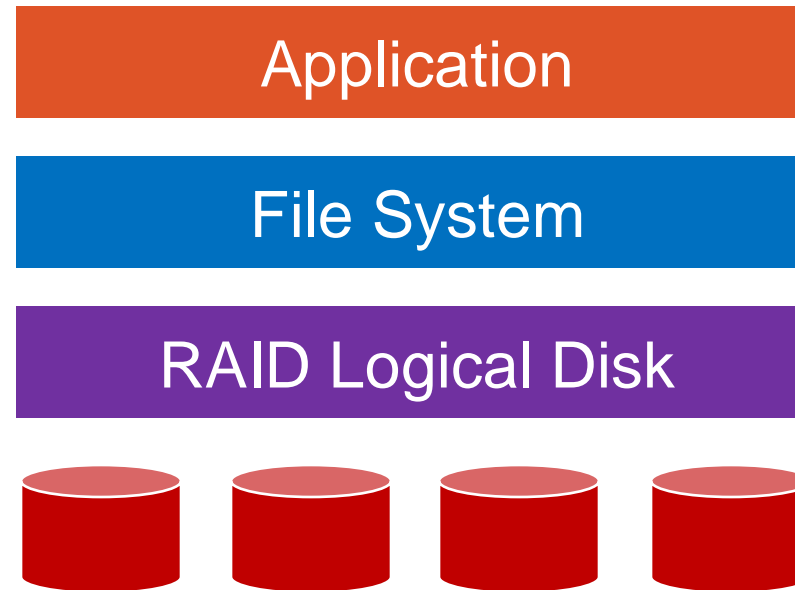
Build a logical disk from many physical disks

Solution: RAID

RAID: Redundant Array of Inexpensive Disks

RAID is

- Transparent
- Deployable



Logical disk gives

- Performance
- Capacity
- Reliability

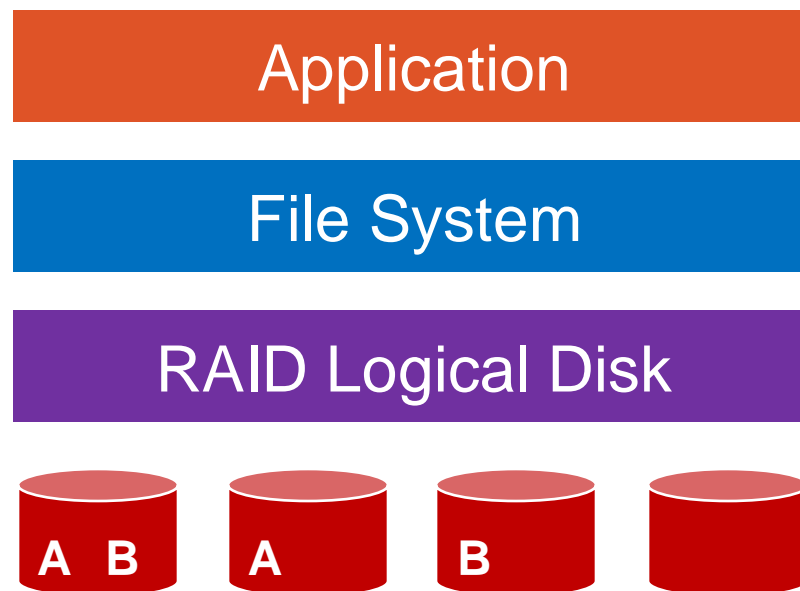
Build a logical disk from many physical disks

Solution: RAID

RAID: Redundant Array of Inexpensive Disks

RAID is

- Transparent
- Deployable



Logical disk gives

- Performance
- Capacity
- Reliability

Build a logical disk from many physical disks

Why Inexpensive Disks?

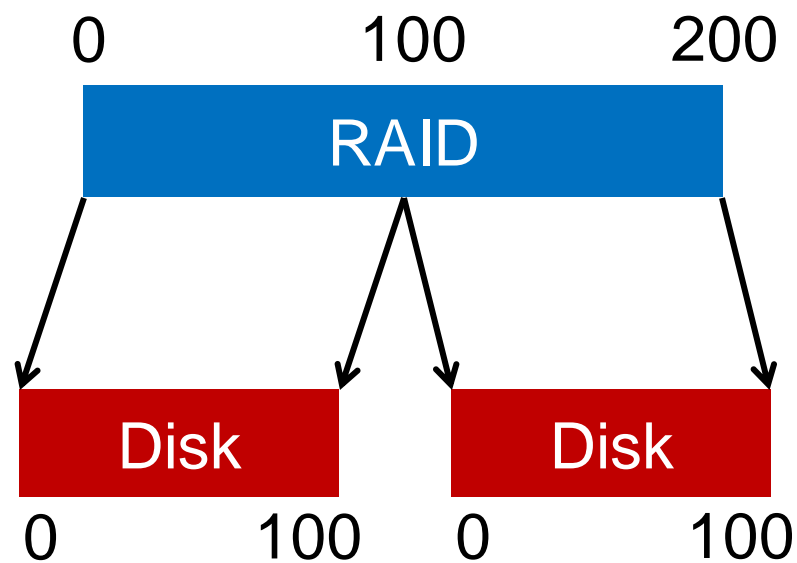
- Economies of scale! Cheap disks are popular
- You can often get **many commodity** hardware components for the same price as a **few expensive** components

Why Inexpensive Disks?

- Economies of scale! Cheap disks are popular
- You can often get **many commodity** hardware components for the same price as a **few expensive** components
- Strategy: Write software to **build high-quality logical devices from many cheap devices**
 - Tradeoff: To compensate poor properties of cheap devices

General Strategy

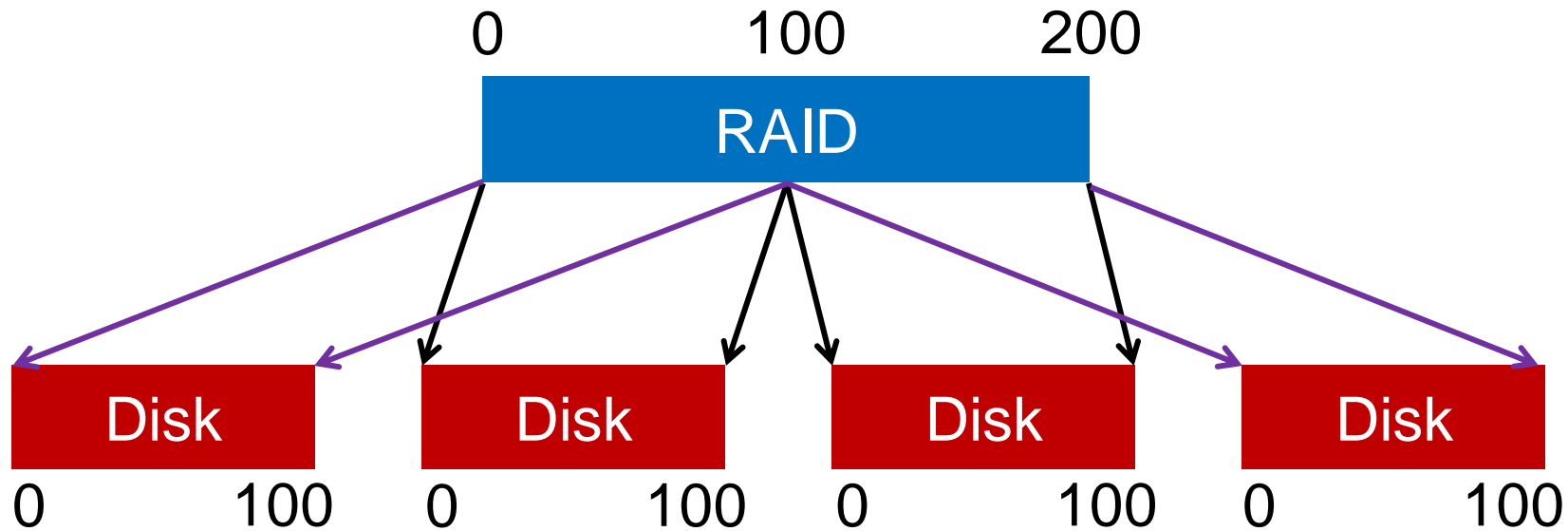
Build fast and large disks from smaller ones



General Strategy

Build fast and large disks from smaller ones

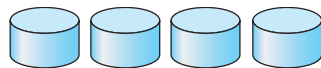
Add more disks for **reliability++**!



RAID Metrics

- Capacity
 - How much space can apps use?
- Reliability
 - How many disks can we safely lose?
 - Assume **fail-stop** model!

RAID Levels



(a) RAID 0: non-redundant striping.



(b) RAID 1: mirrored disks.



(c) RAID 2: memory-style error-correcting codes.



(d) RAID 3: bit-interleaved parity.



(e) RAID 4: block-interleaved parity.

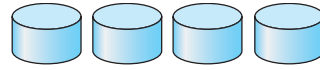


(f) RAID 5: block-interleaved distributed parity.



(g) RAID 6: P + Q redundancy.

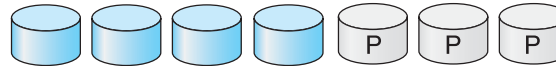
RAID Level 0



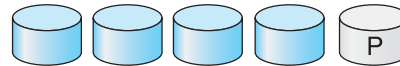
(a) RAID 0: non-redundant striping.



(b) RAID 1: mirrored disks.



(c) RAID 2: memory-style error-correcting codes.



(d) RAID 3: bit-interleaved parity.



(e) RAID 4: block-interleaved parity.



(f) RAID 5: block-interleaved distributed parity.

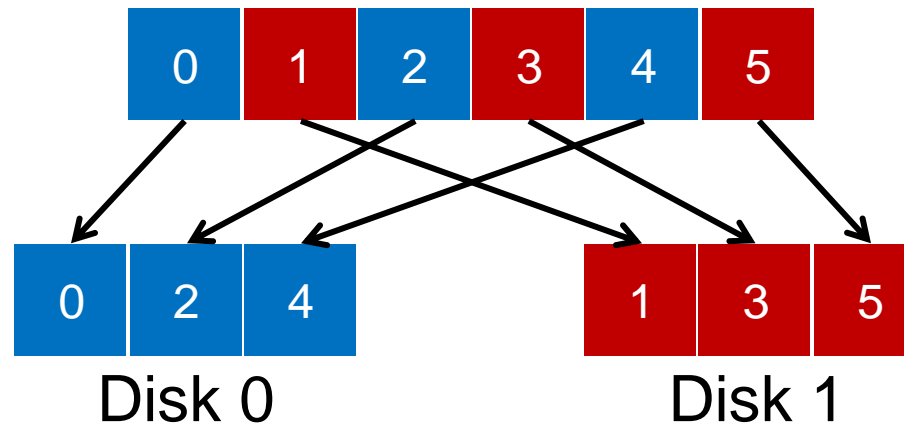


(g) RAID 6: P + Q redundancy.

RAID-0: Striping

- No redundancy
- Serves as **upper bound** for
 - Performance
 - Capacity

Logical blocks



4 Disks

Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

4 Disks

	Disk 0	Disk 1	Disk 2	Disk 3
	0	1	2	3
stripe:	4	5	6	7
	8	9	10	11
	12	13	14	15

How to Map?

- Given logical address A:
 - Disk = ...
 - Offset = ...

Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

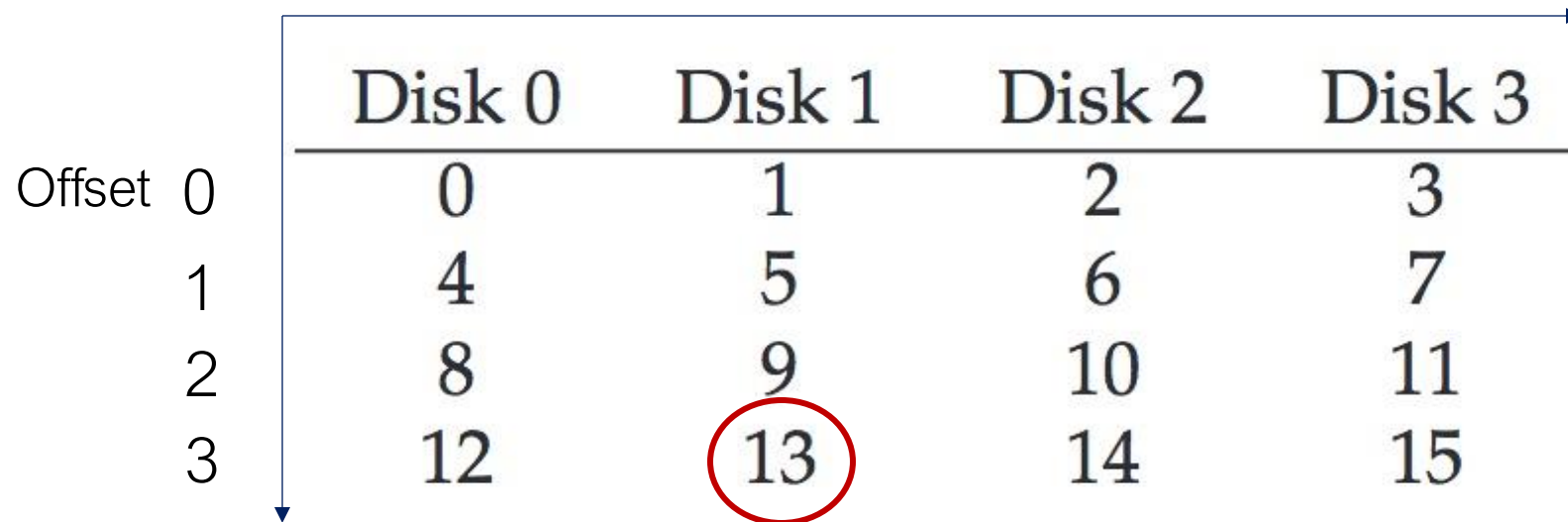
How to Map?

- Given logical address A:
 - **Disk** = $A \% \text{disk_count}$
 - **Offset** = $A / \text{disk_count}$

Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Mapping Example: Find Block 13

- Given logical address 13:
 - Disk** = $13 \% 4 = 1$
 - Offset** = $13 / 4 = 3$



	Disk 0	Disk 1	Disk 2	Disk 3
Offset 0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15

Chunk Size = 1

Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Chunk Size = 1

Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Chunk Size = 2

Disk 0	Disk 1	Disk 2	Disk 3	
0	2	4	6	chunk size: 2 blocks
1	3	5	7	
8	10	12	14	
9	11	13	15	

Chunk Size = 1


Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

In all following examples, we assume chunk size of 1

Chunk Size = 2

Disk 0	Disk 1	Disk 2	Disk 3	
0	2	4	6	chunk size: 2 blocks
1	3	5	7	
8	10	12	14	
9	11	13	15	

RAID-0 Analysis

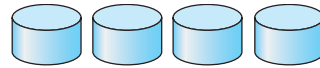
1. What is capacity? $N * C$ 

N is the number of disks

C is the capacity of each disk

2. How many disks can fail? 0

RAID Level 1



(a) RAID 0: non-redundant striping.



(b) RAID 1: mirrored disks.



(c) RAID 2: memory-style error-correcting codes.



(d) RAID 3: bit-interleaved parity.



(e) RAID 4: block-interleaved parity.



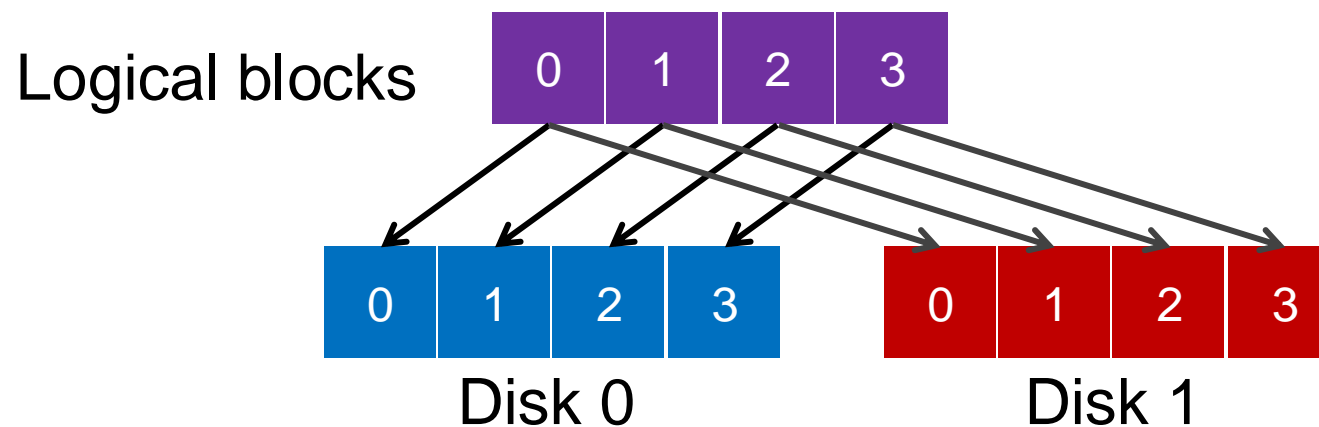
(f) RAID 5: block-interleaved distributed parity.



(g) RAID 6: P + Q redundancy.

RAID-1: Mirroring

- RAID-1 keeps two copies of each block



Assumption

- Assume disks are **fail-stop**
 - Two states
 - They work or they don't
 - We know when they don't work

4 Disks

Disk 0	Disk 1	Disk 2	Disk 3
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

4 Disks

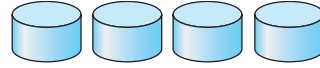
Disk 0	Disk 1	Disk 2	Disk 3
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

How many disks can fail?

RAID-1 Analysis

1. What is capacity? $N/2 * C$
2. How many disks can fail? 1 or maybe $N / 2$

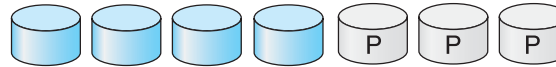
RAID Level 4



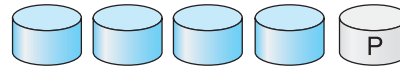
(a) RAID 0: non-redundant striping.



(b) RAID 1: mirrored disks.



(c) RAID 2: memory-style error-correcting codes.



(d) RAID 3: bit-interleaved parity.



(e) RAID 4: block-interleaved parity.

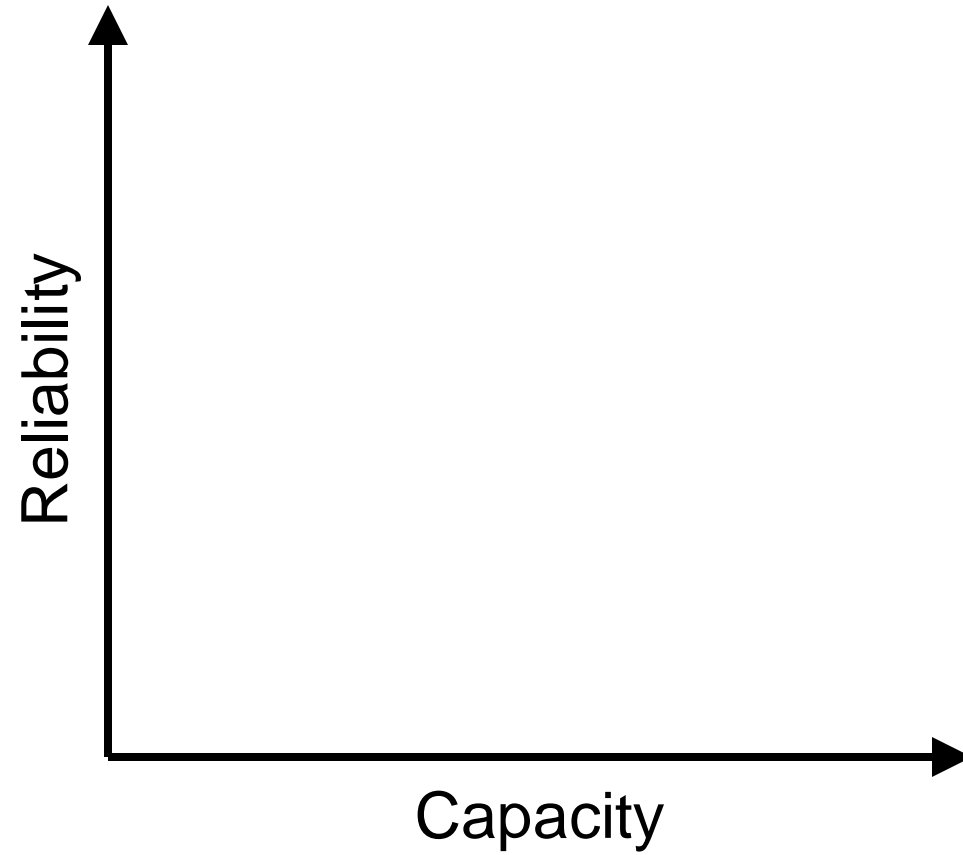


(f) RAID 5: block-interleaved distributed parity.

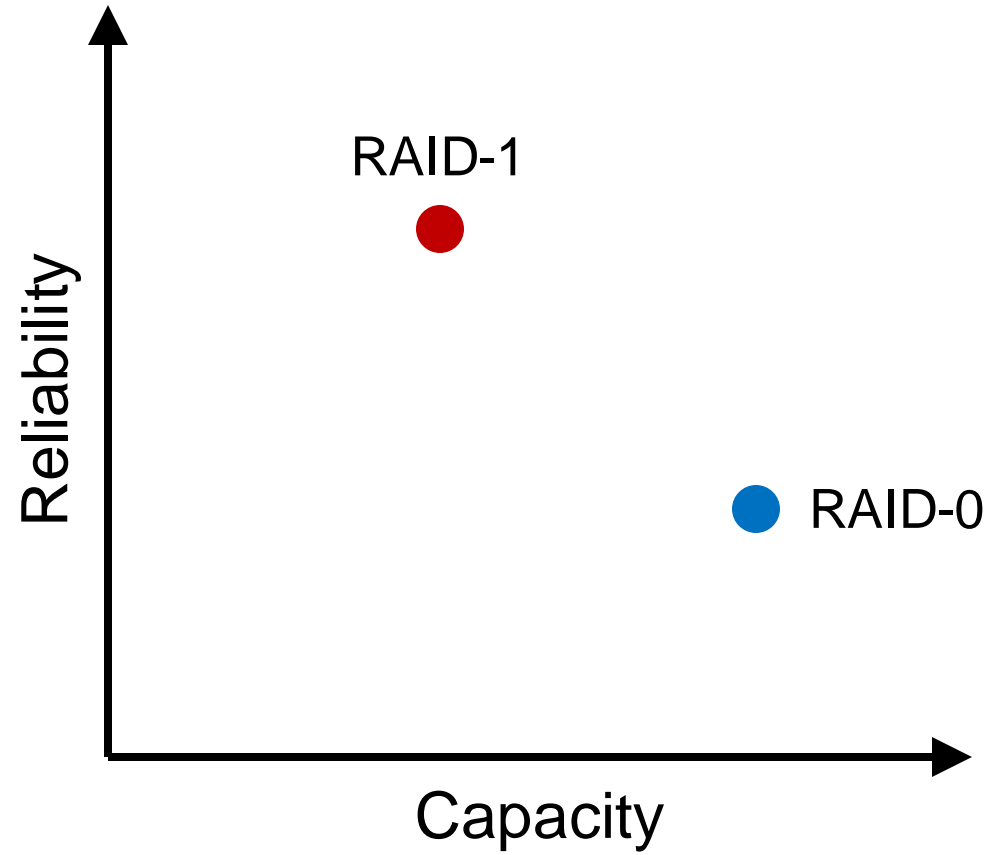


(g) RAID 6: P + Q redundancy.

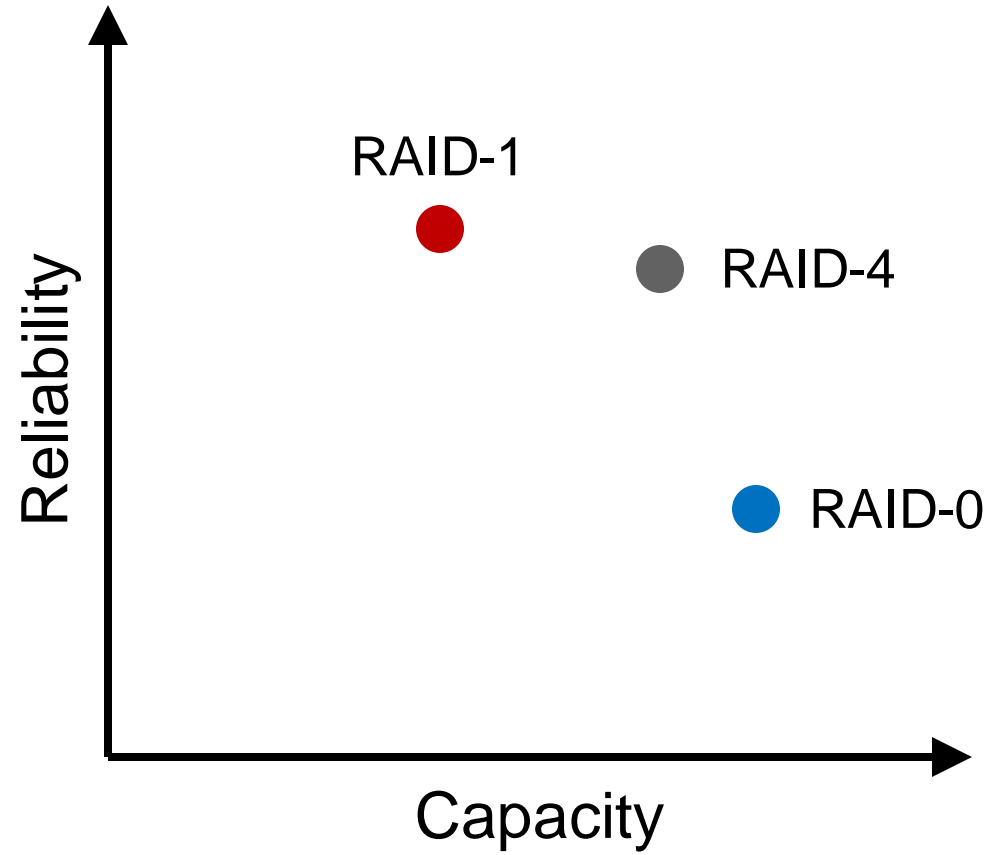
RAID-4



RAID-4



RAID-4

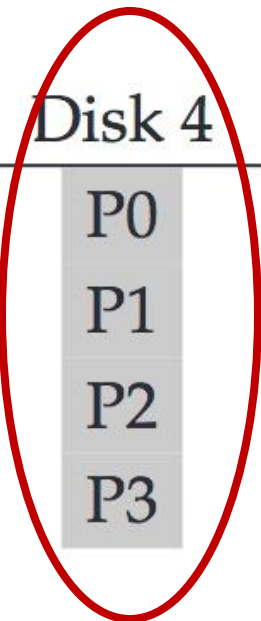


RAID-4 Strategy

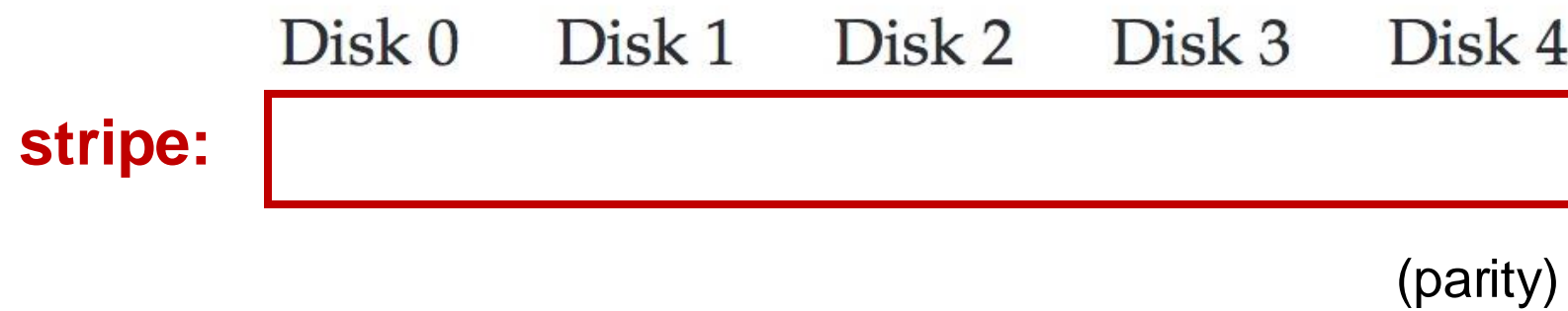
- Use **parity** disk
- In algebra, if an **equation** has N variables, and $N-1$ are known, you can also solve for the unknown
- Treat the sectors/blocks across disks in a stripe as an equation

5 Disks

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3



Example



Example

	Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
stripe:	4	3	0	2	

(parity)

Example

	Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
stripe:	4	3	0	2	9

(parity)

Example

	Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
stripe:	X	3	0	2	9

(parity)

Example

	Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
stripe:	4	3	0	2	9

(parity)

Parity Function: XOR Example

C0	C1	C2	C3	P
0	0	1	1	$\text{XOR}(0,0,1,1) = 0$
0	1	0	0	$\text{XOR}(0,1,0,0) = 1$

Parity Function: XOR Example

C0	C1	C2	C3	P
0	0	1	1	$\text{XOR}(0,0,1,1) = 0$
0	1	0	0	$\text{XOR}(0,1,0,0) = 1$

XOR function:

- $P = 0$: The number of 1 in a stripe must be an even number
- $P = 1$: The number of 1 in a stripe must be an odd number

Parity Function: XOR Example

	Block0	Block1	Block2	Block3	Parity
stripe:	00	10	11	10	11
	10	01	00	01	10

XOR function:

- $P = 0$: The number of 1 in a stripe must be an even number
- $P = 1$: The number of 1 in a stripe must be an odd number

Parity Function: XOR Example

	Block0	Block1	Block2	Block3	Parity
stripe:	<div><div>×</div></div>	10	11	10	11
	10	01	00	01	10

XOR function:

- $P = 0$: The number of 1 in a stripe must be an even number
- $P = 1$: The number of 1 in a stripe must be an odd number

Parity Function: XOR Example

	Block0	Block1	Block2	Block3	Parity
stripe:	<div><div>×</div></div>	10	11	10	11
	10	01	00	01	10

$$\text{Block0} = \text{XOR}(10, 11, 10, 11) = 00$$

XOR function:

- $P = 0$: The number of 1 in a stripe must be an even number
- $P = 1$: The number of 1 in a stripe must be an odd number

Parity Function: XOR Example

	Block0	Block1	Block2	Block3	Parity
stripe:	00	10	11	10	11
	10	01	00	01	10

$$\text{Block0} = \text{XOR}(10, 11, 10, 11) = \mathbf{00}$$

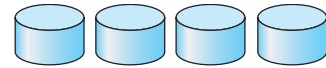
XOR function:

- $P = 0$: The number of 1 in a stripe must be an even number
- $P = 1$: The number of 1 in a stripe must be an odd number

RAID-4 Analysis

1. What is capacity? $(N-1) * C$
2. How many disks can fail? 1

RAID Level 5



(a) RAID 0: non-redundant striping.



(b) RAID 1: mirrored disks.



(c) RAID 2: memory-style error-correcting codes.



(d) RAID 3: bit-interleaved parity.



(e) RAID 4: block-interleaved parity.



(f) RAID 5: block-interleaved distributed parity.



(g) RAID 6: P + Q redundancy.

RAID-5: Rotating Parity

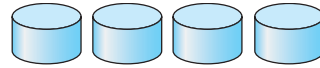
Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14
P4	16	17	18	19

RAID-5 works almost identically to RAID-4, except that it rotates the parity block across drives

RAID-5 Analysis

1. What is capacity? $(N-1) * C$
2. How many disks can fail? 1

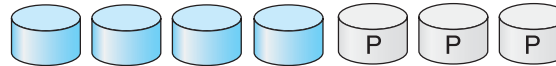
RAID Level 6



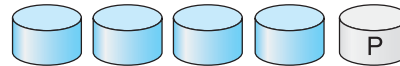
(a) RAID 0: non-redundant striping.



(b) RAID 1: mirrored disks.



(c) RAID 2: memory-style error-correcting codes.



(d) RAID 3: bit-interleaved parity.



(e) RAID 4: block-interleaved parity.

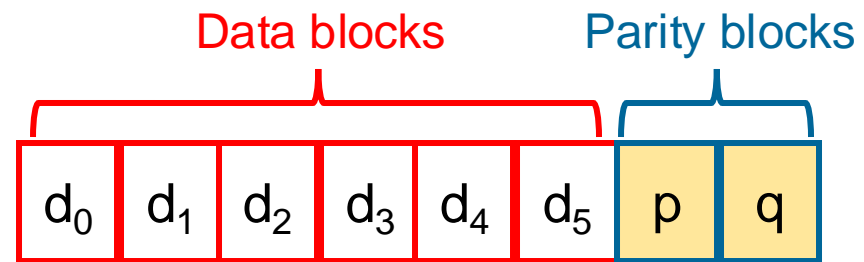


(f) RAID 5: block-interleaved distributed parity.



(g) RAID 6: P + Q redundancy.

RAID-6



RAID-6 can fail at most 2 disks at a time.

Encoding

1	0	0	0	0	0
0	1	0	0	0	0
0	0	1	0	0	0
0	0	0	1	0	0
0	0	0	0	1	0
0	0	0	0	0	1
1	1	1	1	1	1
32	16	8	4	2	1

Generator matrix

d₀

d₁

d₂

d₃

d₄

d₅

*

=

d₀

d₁

d₂

d₃

d₄

d₅

p

q

The product vector is called a codeword

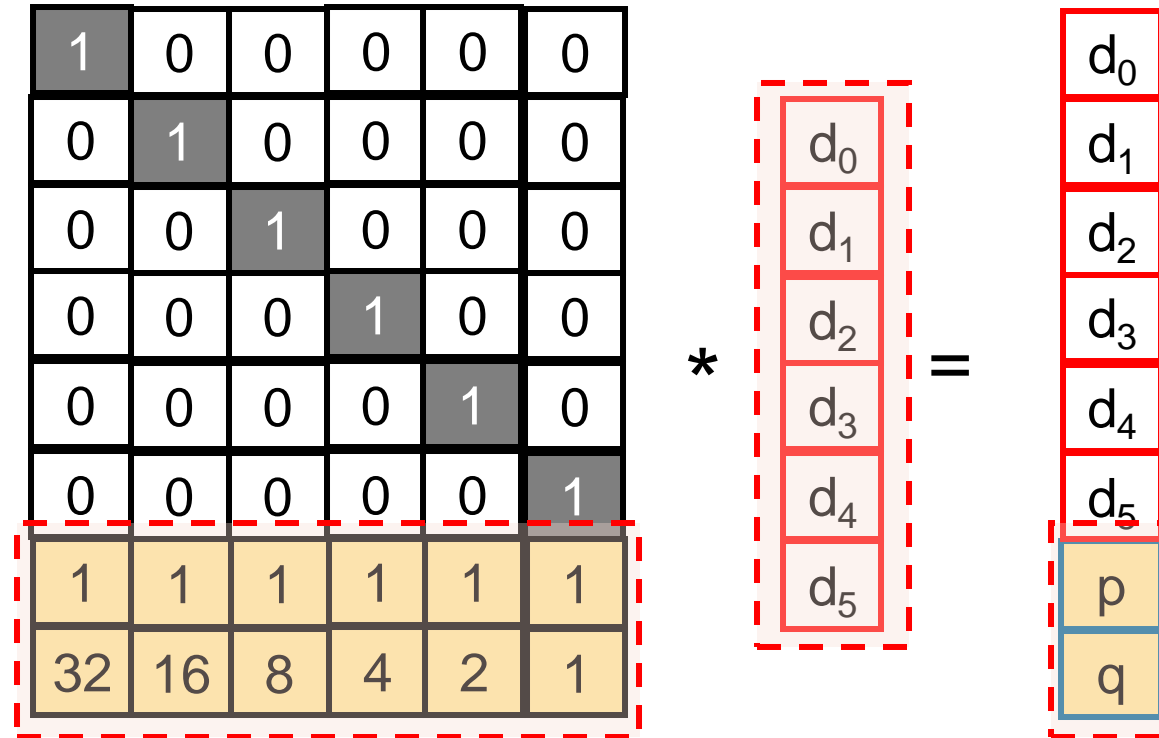
$$[8 \times 6] * [6 \times 1] = [8 \times 1]$$

Yue Cheng

CS4740 Fall '24

79

Encoding



$$\begin{aligned}
 d_0 \oplus d_1 \oplus d_2 \oplus d_3 \oplus d_4 \oplus d_5 &\longrightarrow p \\
 32d_0 \oplus 16d_1 \oplus 8d_2 \oplus 4d_3 \oplus 2d_4 \oplus d_5 &\longrightarrow q
 \end{aligned}$$

Decoding with a parity check matrix

Parity check matrix

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 32 & 16 & 8 & 4 & 2 & 1 & 0 & 1 \end{bmatrix} * \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ p \\ q \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\begin{aligned}
 d_0 \oplus d_1 \oplus d_2 \oplus d_3 \oplus d_4 \oplus d_5 \oplus p &= 0 \\
 32d_0 \oplus 16d_1 \oplus 8d_2 \oplus 4d_3 \oplus 2d_4 \oplus d_5 \oplus q &= 0
 \end{aligned}$$

Handling failures with decoding

$$\begin{array}{ccccccccc} d_0 & \oplus & d_1 & \oplus & d_2 & \oplus & d_3 & \oplus & d_4 & \oplus & d_5 & \oplus & p & = & 0 \\ 32d_0 & \oplus & 16d_1 & \oplus & 8d_2 & \oplus & 4d_3 & \oplus & 2d_4 & \oplus & d_5 & \oplus & q & = & 0 \end{array}$$

Suppose disk1 (d_1) and disk4 (d_4) fail

Handling failures with decoding

$$\begin{array}{ccccccccc} d_0 & \oplus & d_1 & \oplus & d_2 & \oplus & d_3 & \oplus & d_4 & \oplus & d_5 & \oplus & p & = & 0 \\ 32d_0 & \oplus & 16d_1 & \oplus & 8d_2 & \oplus & 4d_3 & \oplus & 2d_4 & \oplus & d_5 & \oplus & q & = & 0 \end{array}$$

Suppose disk1 (d_1) and disk4 (d_4) fail

Step 1: Put the failed data on the right of the equations.

$$\begin{array}{lcl} d_0 \oplus d_2 \oplus d_3 \oplus d_5 \oplus p & = & d_1 \oplus d_4 \\ 32d_0 \oplus 8d_2 \oplus 4d_3 \oplus d_5 \oplus q & = & 16d_1 \oplus 2d_4 \end{array}$$

Handling failures with decoding

$$\begin{array}{ccccccccc} d_0 & \oplus & d_1 & \oplus & d_2 & \oplus & d_3 & \oplus & d_4 & \oplus & d_5 & \oplus & p & = & 0 \\ 32d_0 & \oplus & 16d_1 & \oplus & 8d_2 & \oplus & 4d_3 & \oplus & 2d_4 & \oplus & d_5 & \oplus & q & = & 0 \end{array}$$

Suppose disk1 (d_1) and disk4 (d_4) fail

Step 2: Calculate the left sides, since those all exist.

$$\begin{array}{ccccccccccc} d_0 & \oplus & d_2 & \oplus & d_3 & \oplus & d_5 & \oplus & p & = & S_0 & = & d_1 & \oplus & d_4 \\ 32d_0 & \oplus & 8d_2 & \oplus & 4d_3 & \oplus & d_5 & \oplus & q & = & S_1 & = & 16d_1 & \oplus & 2d_4 \end{array}$$

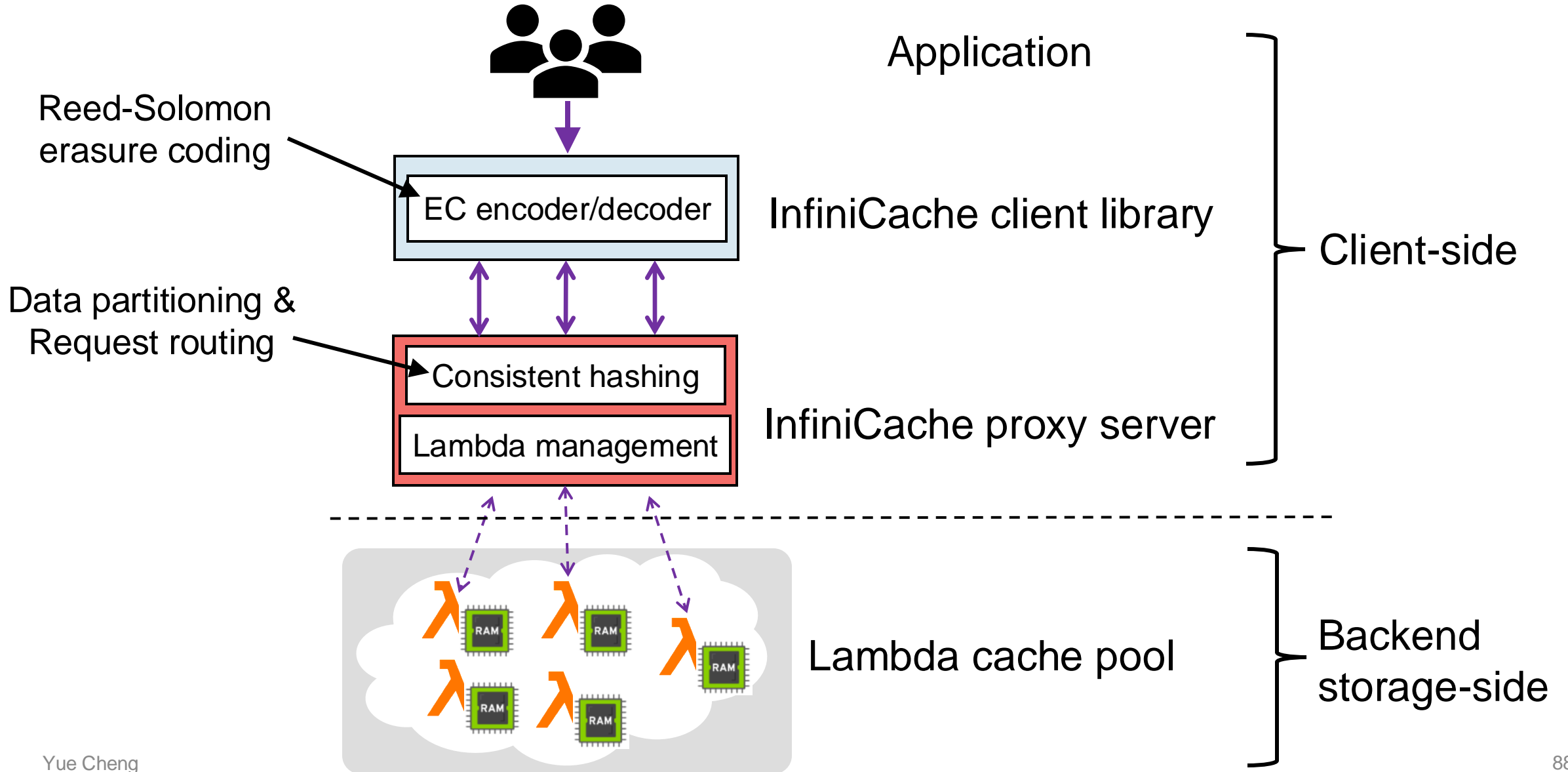
RAID-6 Analysis

Assuming a RS configuration of 6+2

1. What is capacity? $(N-2) * C$ where $N = 8$
2. How many disks can fail? 2

Switching back to InfiniCache

InfiniCache bird's eye view



InfiniCache: PUT path



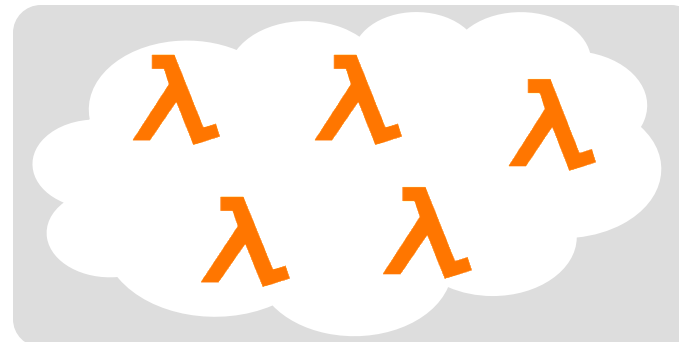
Application

EC encoder

InfiniCache client library

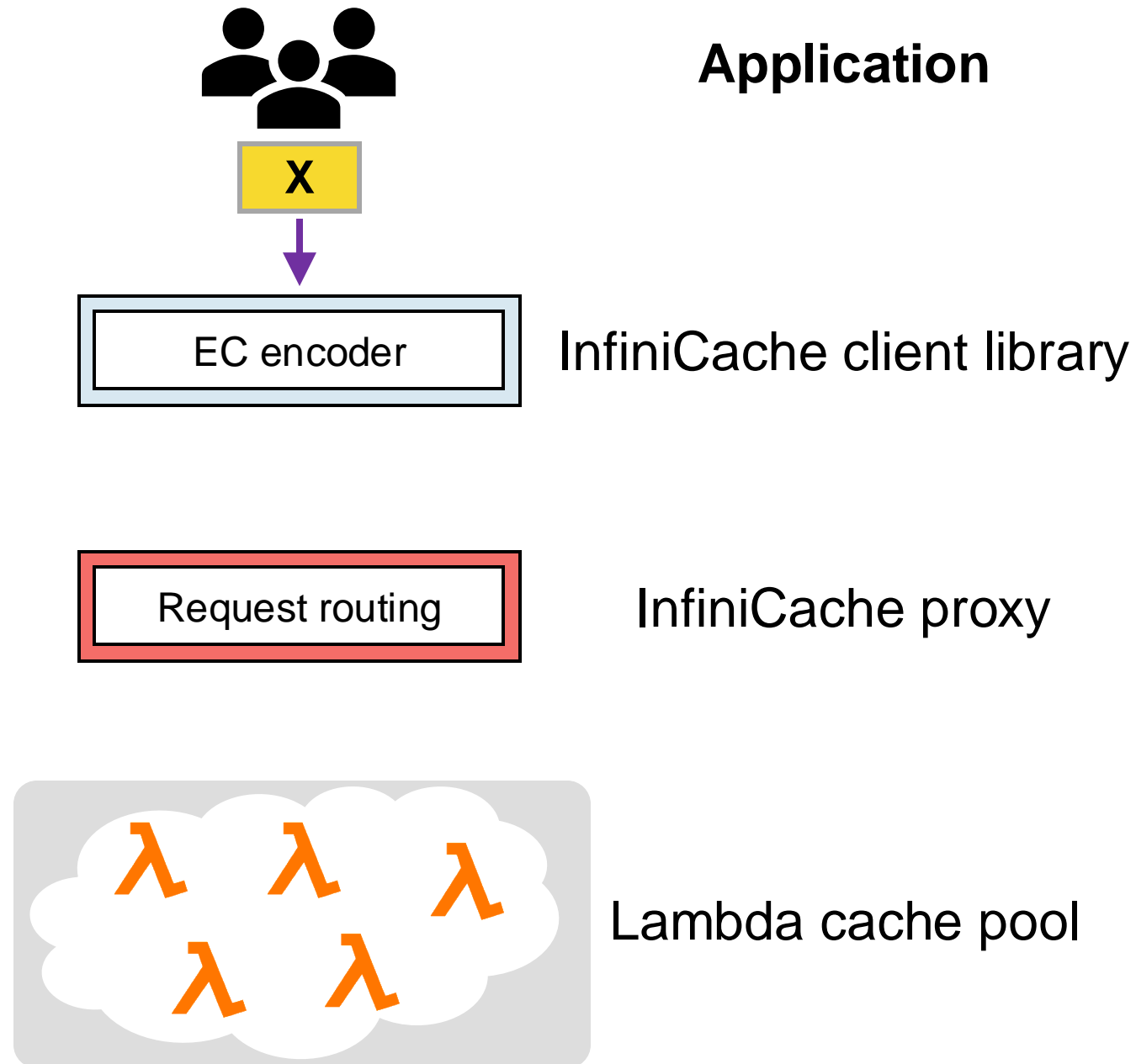
Request routing

InfiniCache proxy



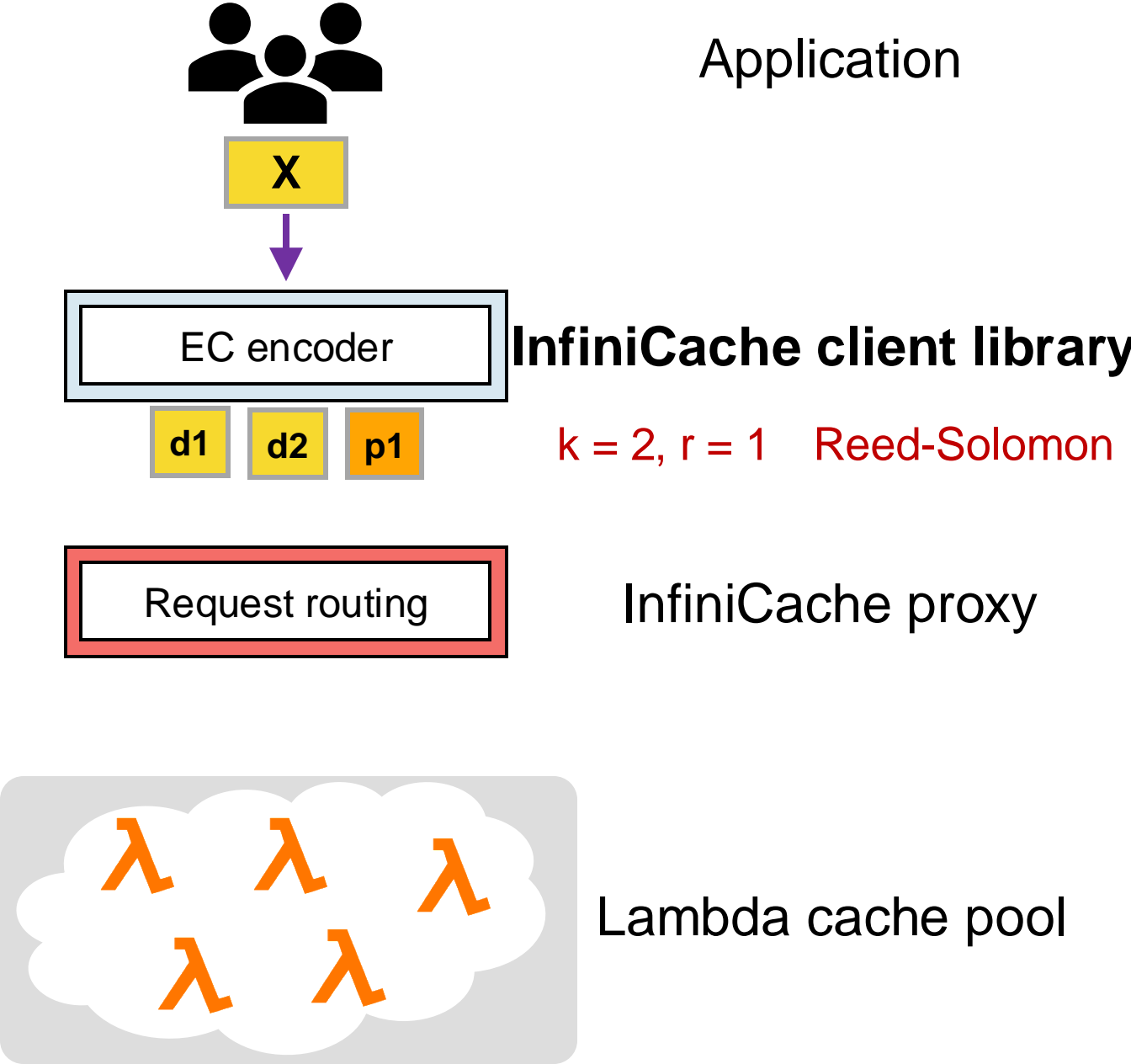
Lambda cache pool

InfiniCache: PUT path



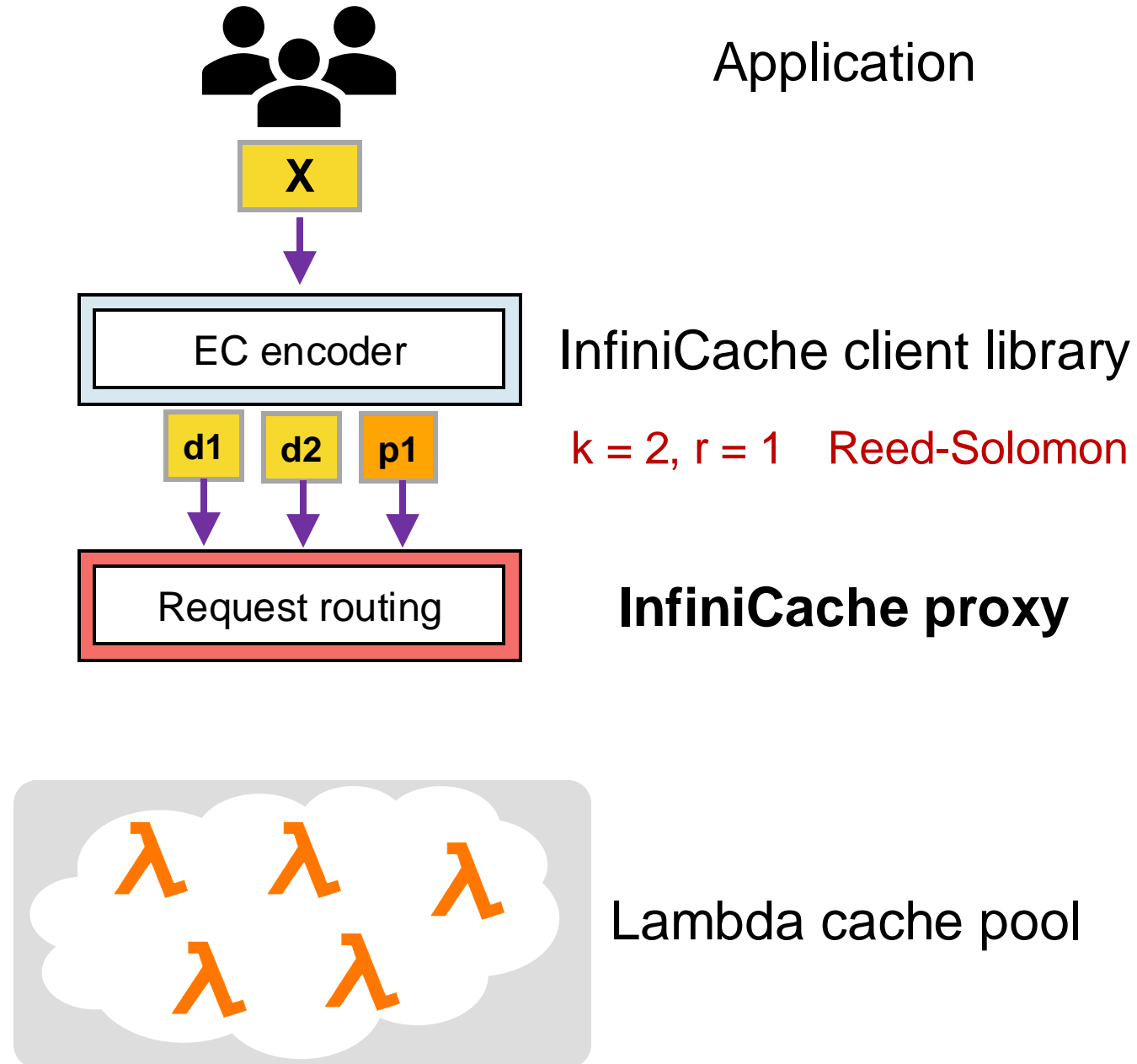
InfiniCache: PUT path

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



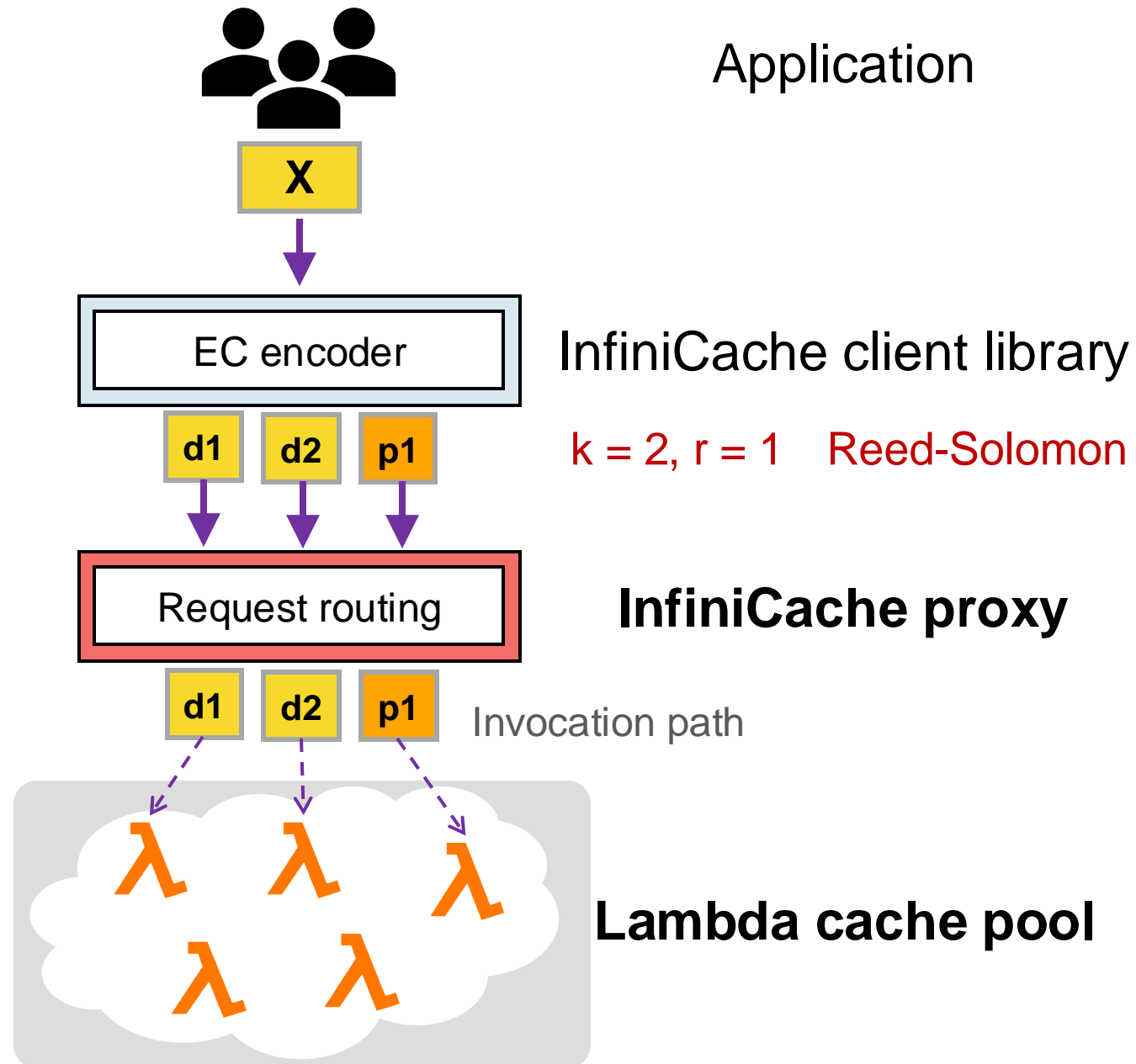
InfiniCache: PUT path

1. Object is split and encoded into $k+r$ chunks
2. Object chunks are sent to the proxy in parallel



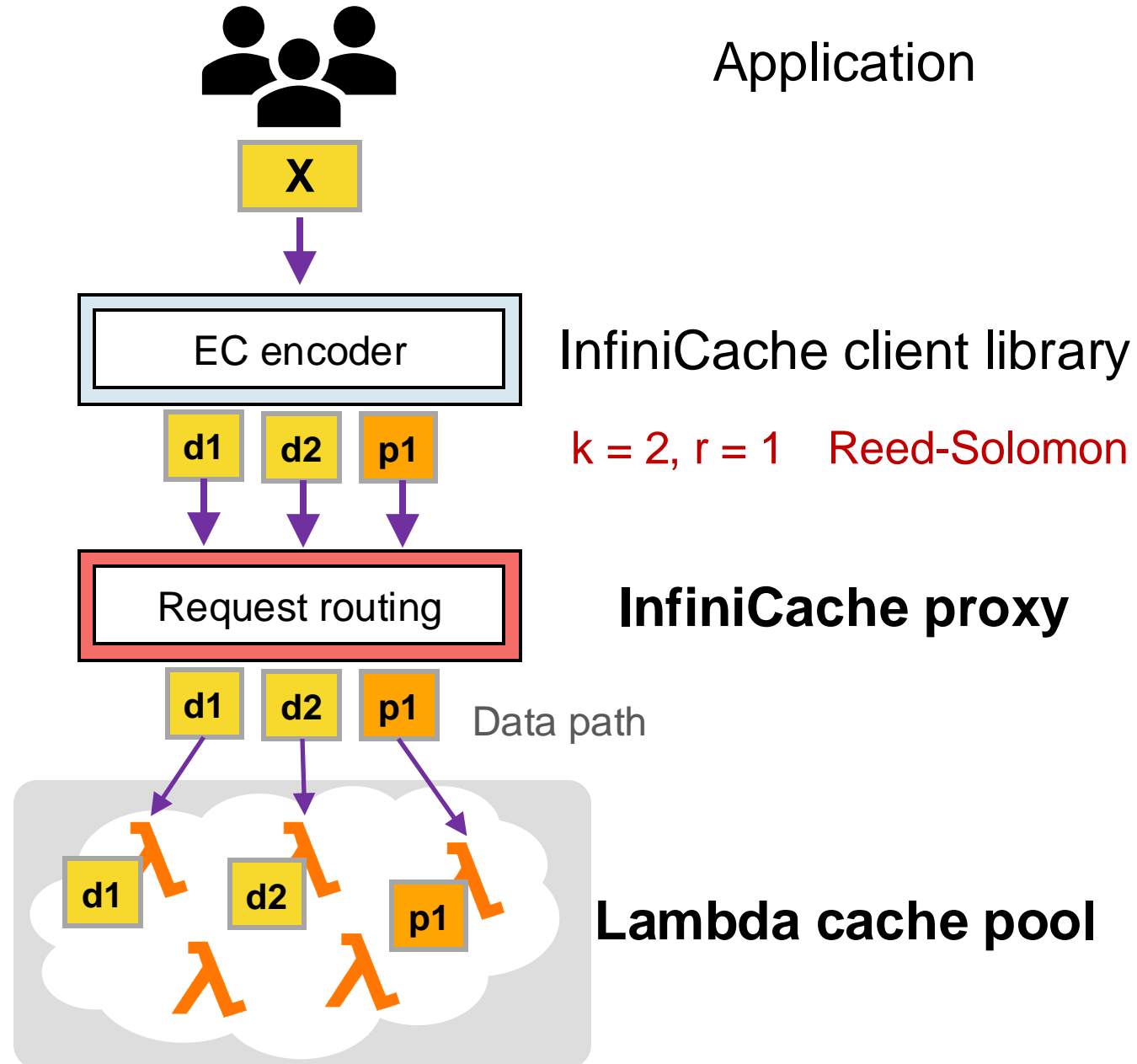
InfiniCache: PUT path

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

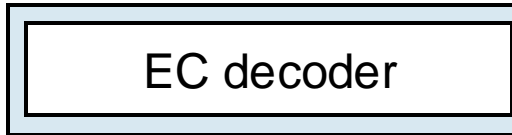
1. Object is split and encoded into $k+r$ chunks
2. Object chunks are sent to the proxy in parallel
3. Proxy invokes Lambda cache nodes
4. Proxy streams object chunks to Lambda cache nodes



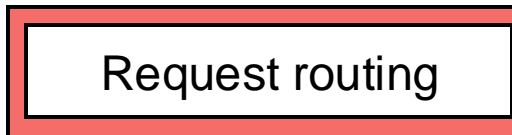
InfiniCache: GET path



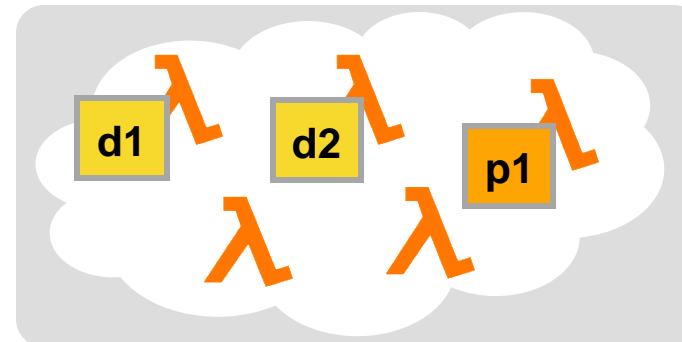
Application



InfiniCache client library



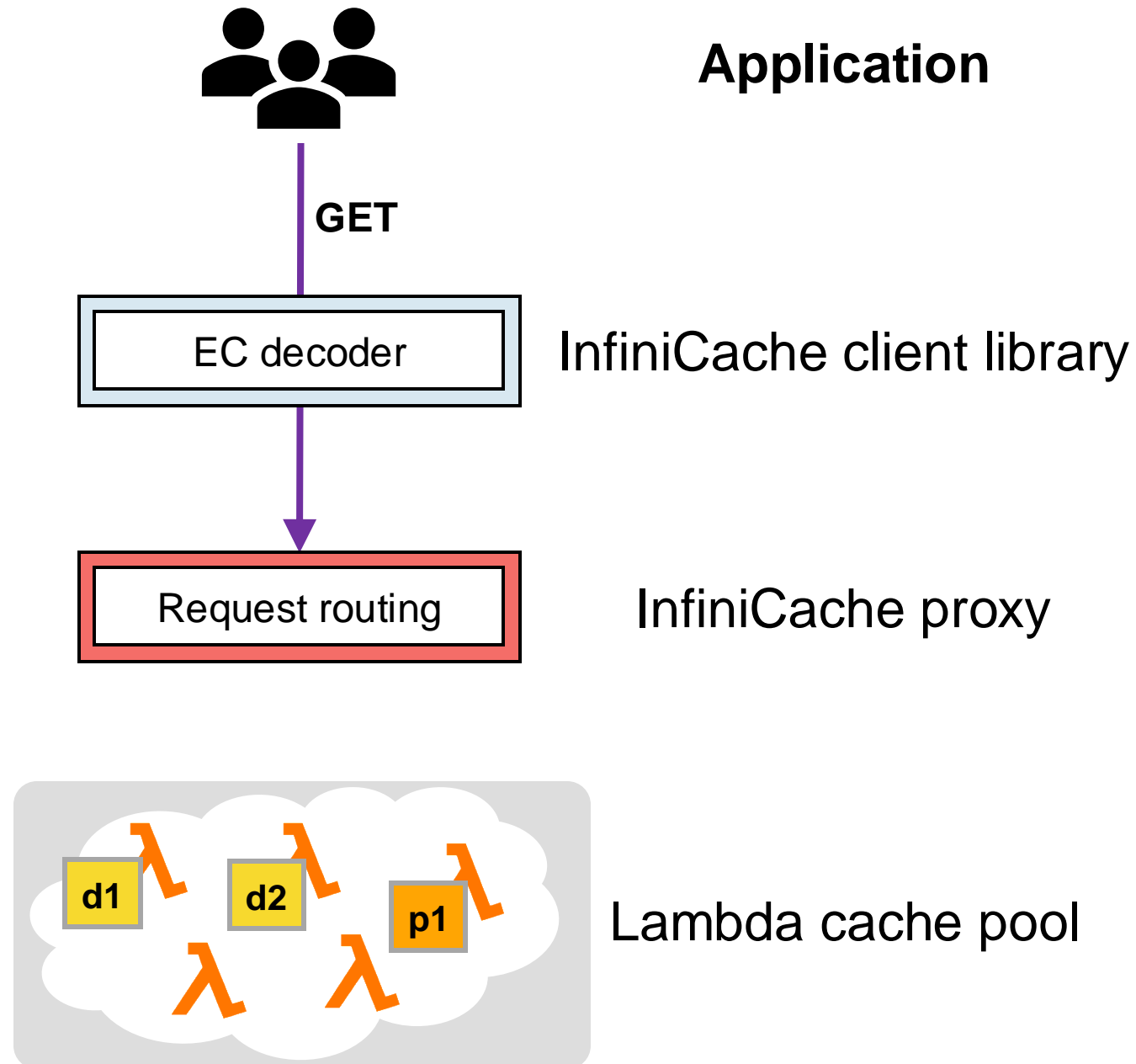
InfiniCache proxy



Lambda cache pool

InfiniCache: GET path

1. Client sends GET request

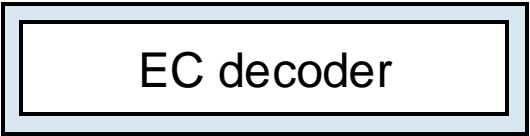


InfiniCache: GET path



Application

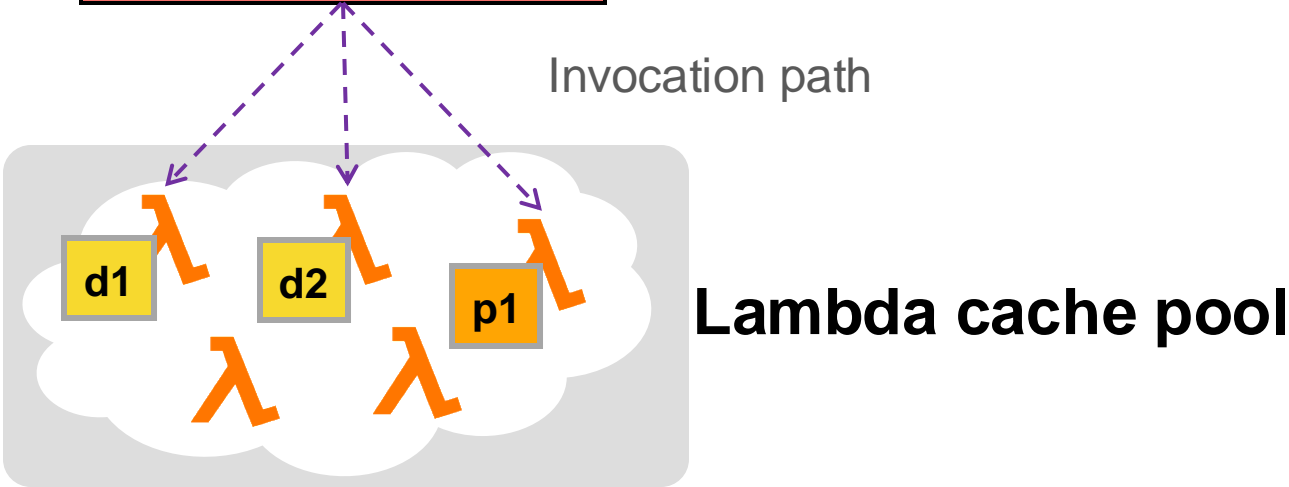
- 1. Client sends GET request
- 2. Proxy invokes associated Lambda cache nodes



InfiniCache client library



InfiniCache proxy

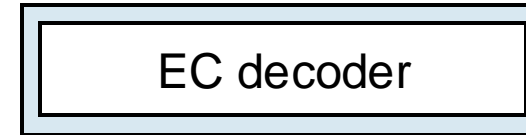


InfiniCache: GET path



Application

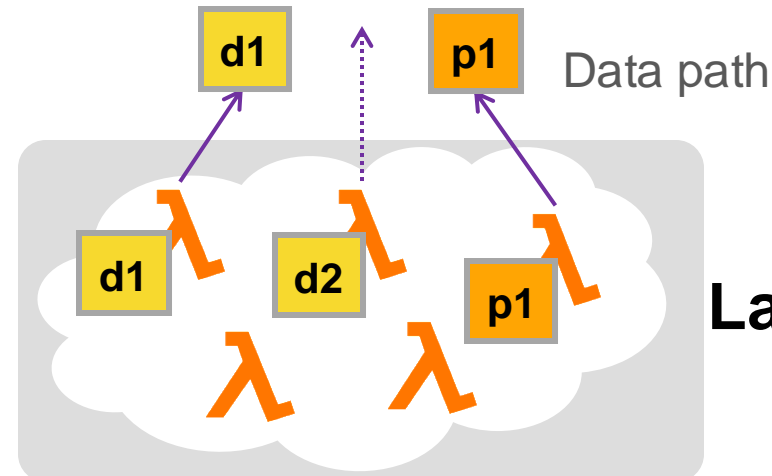
1. Client sends GET request
2. Proxy invokes associated Lambda cache nodes
3. Lambda cache nodes transfer object chunks to proxy



InfiniCache client library



InfiniCache proxy



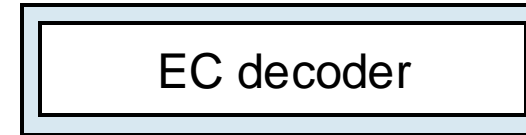
Lambda cache pool

InfiniCache: GET path



Application

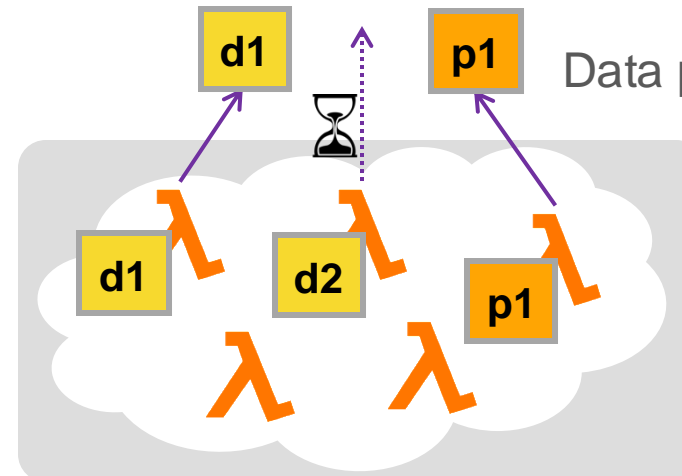
1. Client sends GET request
2. Proxy invokes associated Lambda cache nodes
3. Lambda cache nodes transfer object chunks to proxy
 - **First-d optimization:** Proxy drops **straggler** Lambda



InfiniCache client library



InfiniCache proxy



Data path $k = 2, r = 1$
 $d2$ is straggling...

Lambda cache pool

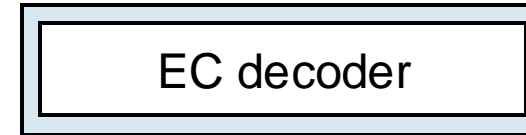
InfiniCache: GET path



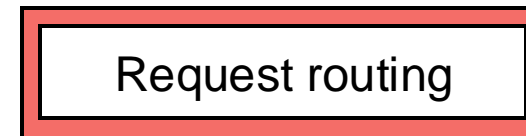
Application

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

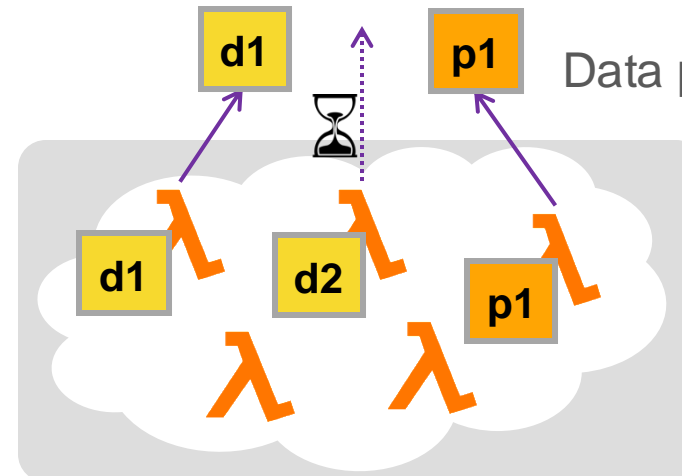
1. Client sends request
2. Proxy invokes associated Lambda cache nodes
3. Lambda cache nodes transfer object chunks to proxy
 - **First-d optimization:** Proxy drops **straggler** Lambda



InfiniCache client library



InfiniCache proxy



Data path $k = 2, r = 1$
d2 is straggling...

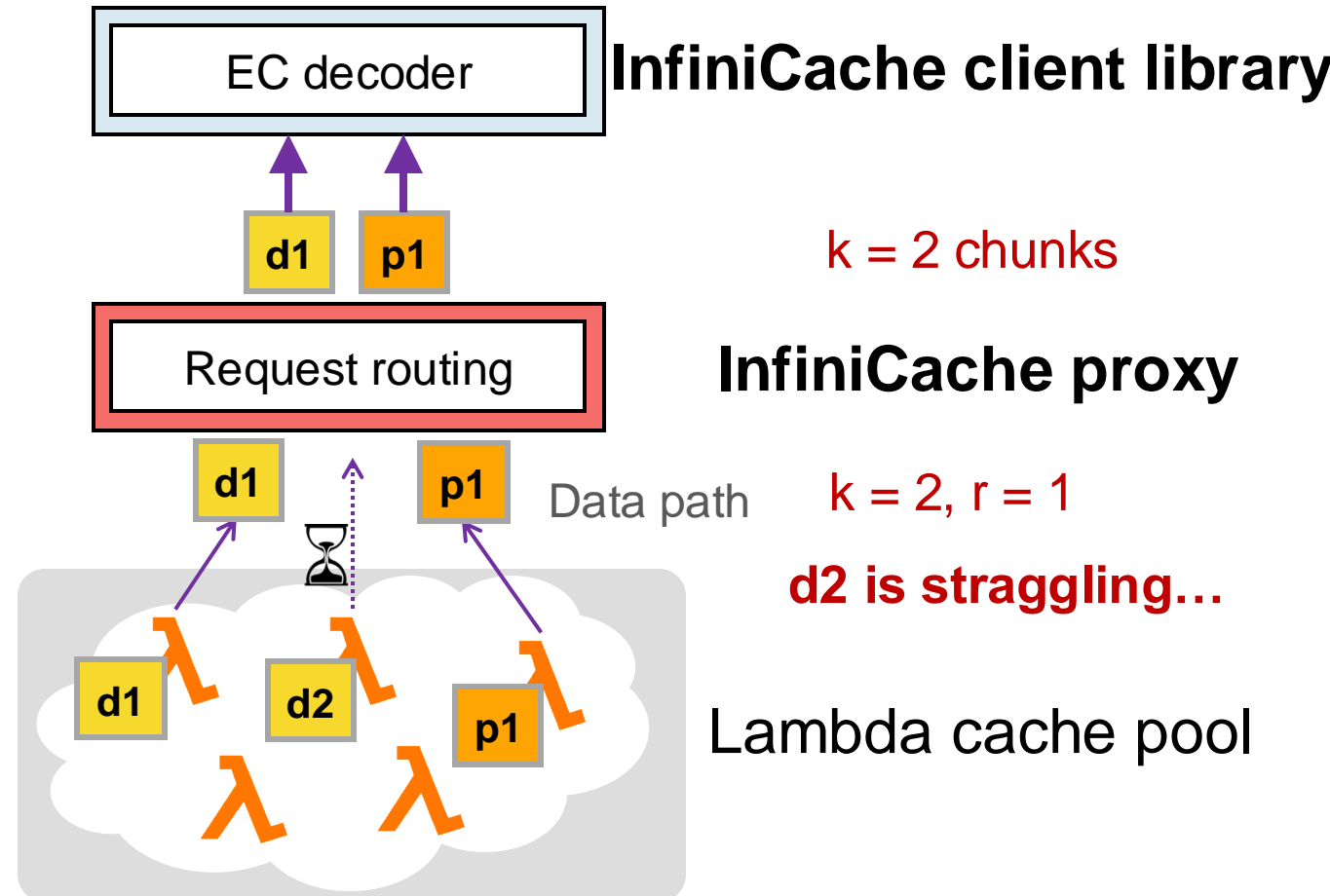
Lambda cache pool

InfiniCache: GET path



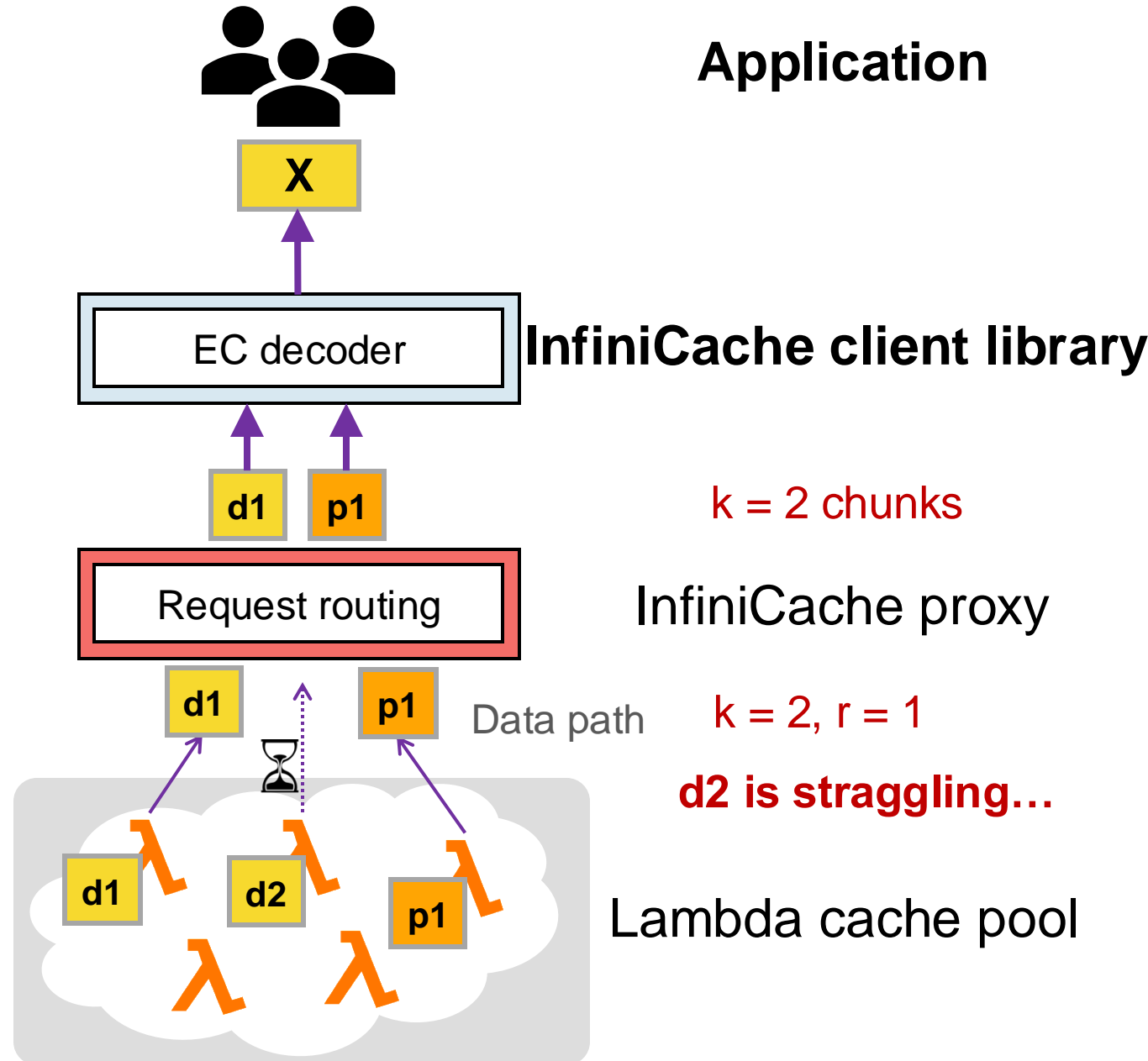
Application

1. Client sends GET request
2. Proxy invokes associated Lambda cache nodes
3. Lambda cache nodes transfer object chunks to proxy
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



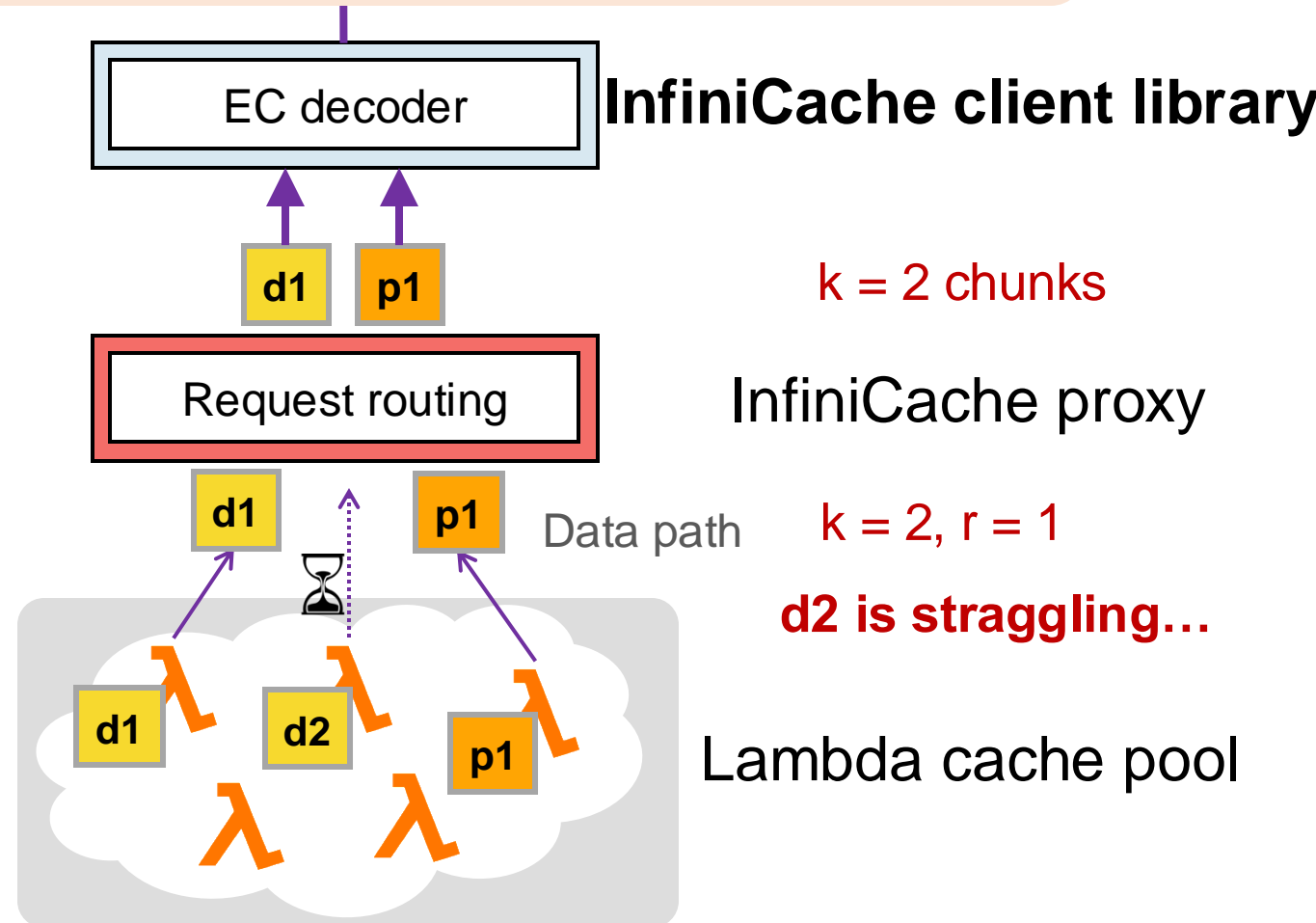
InfiniCache: GET path



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 sends request
2. Proxy invokes Lambda cache
3. Lambda cache transfers object chunks to proxy
4. Proxy streams $k=2$ chunks in parallel to client
5. Client library **decodes** k chunks



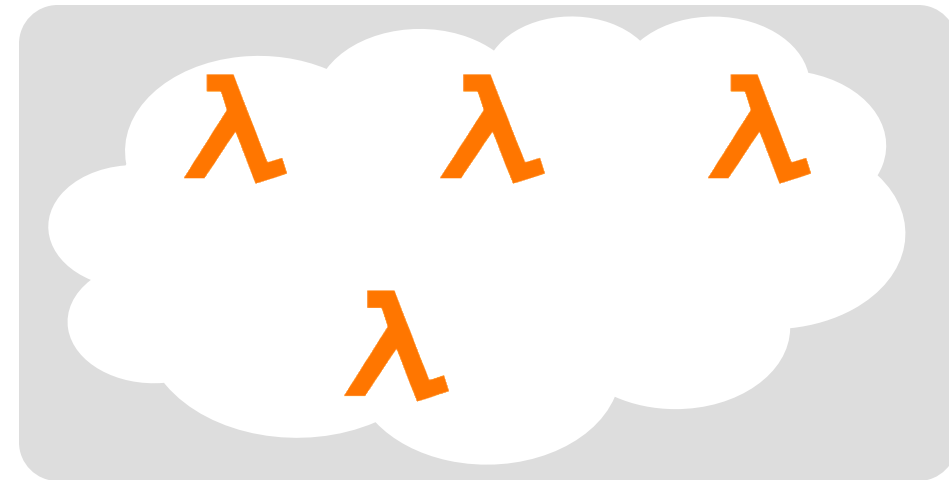
Maximizing data availability

- 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

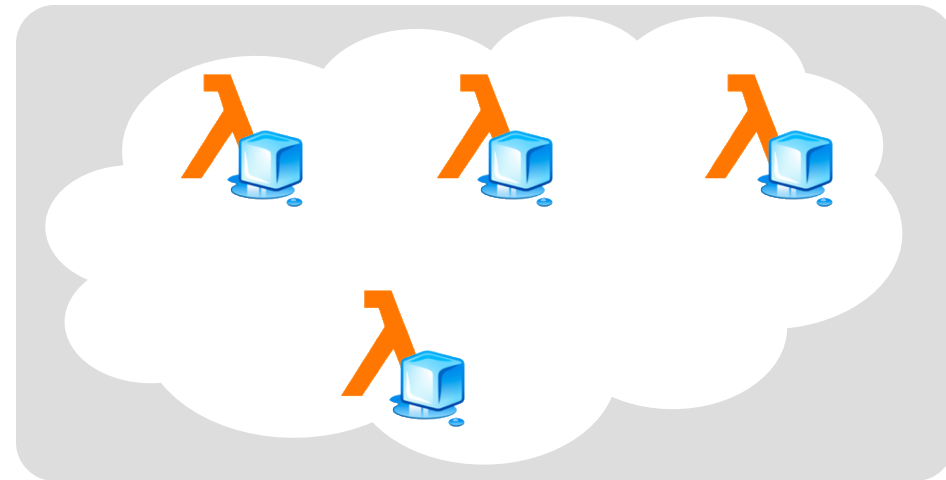
Proxy



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

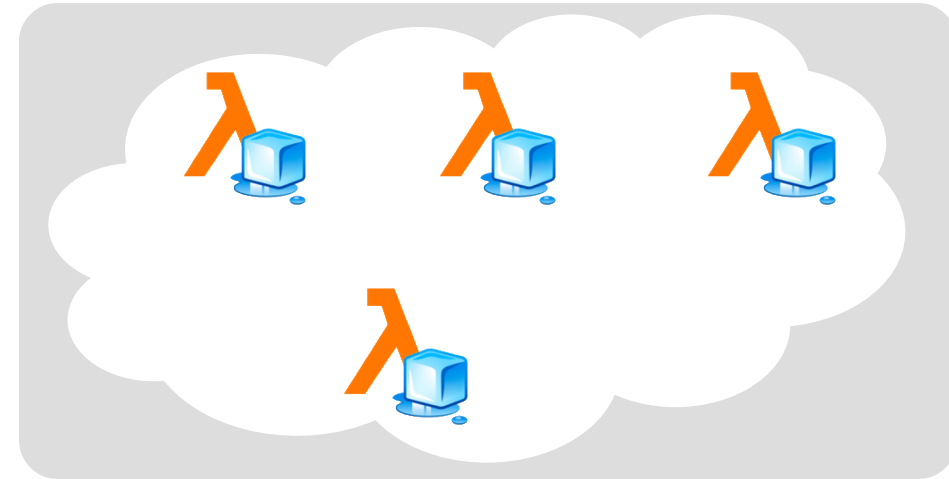
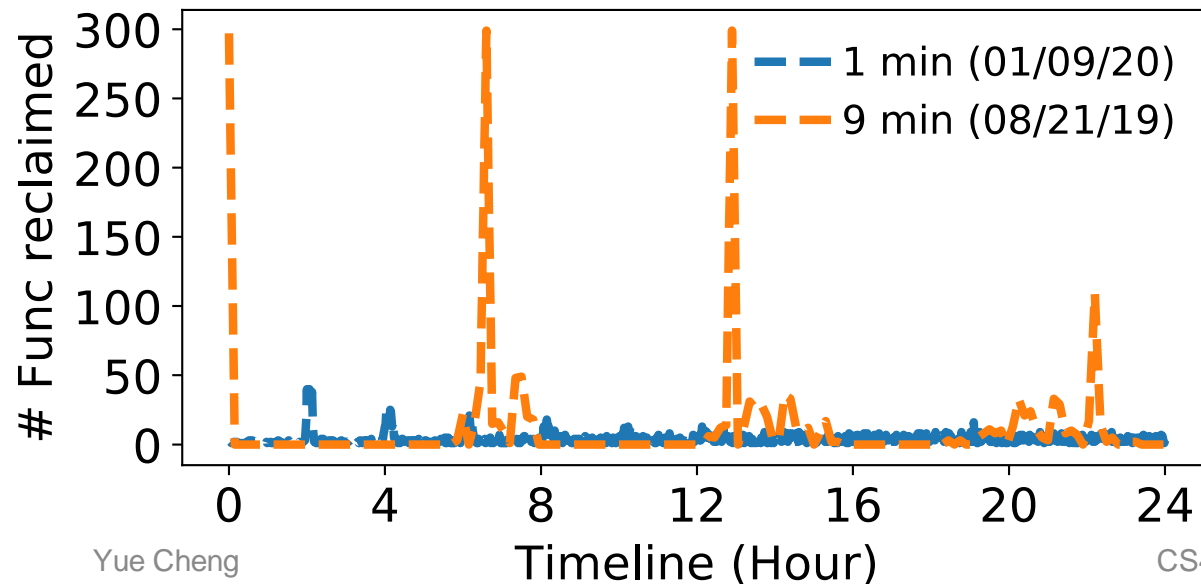
Proxy



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

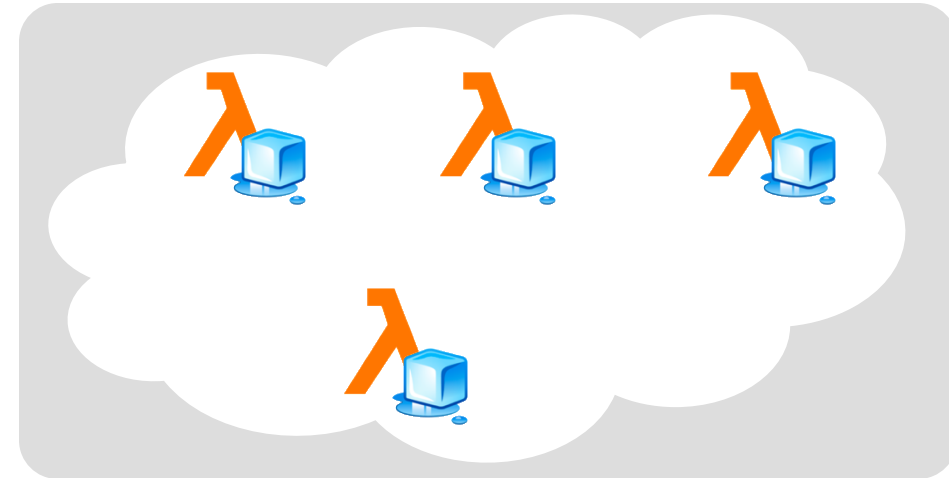
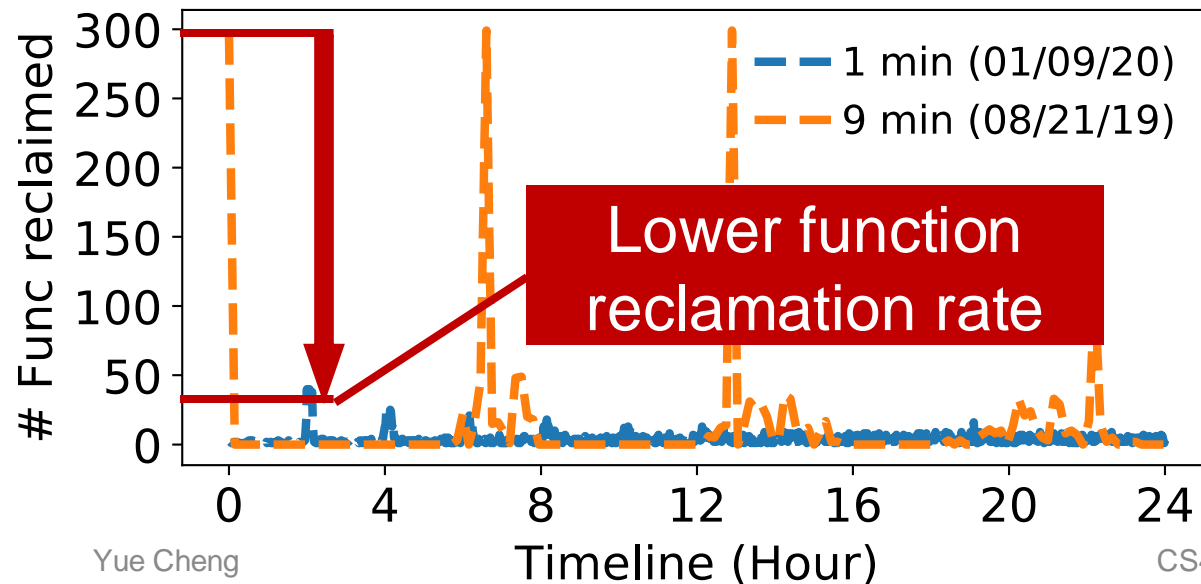
Proxy



Maximizing data availability: Periodic warm-up

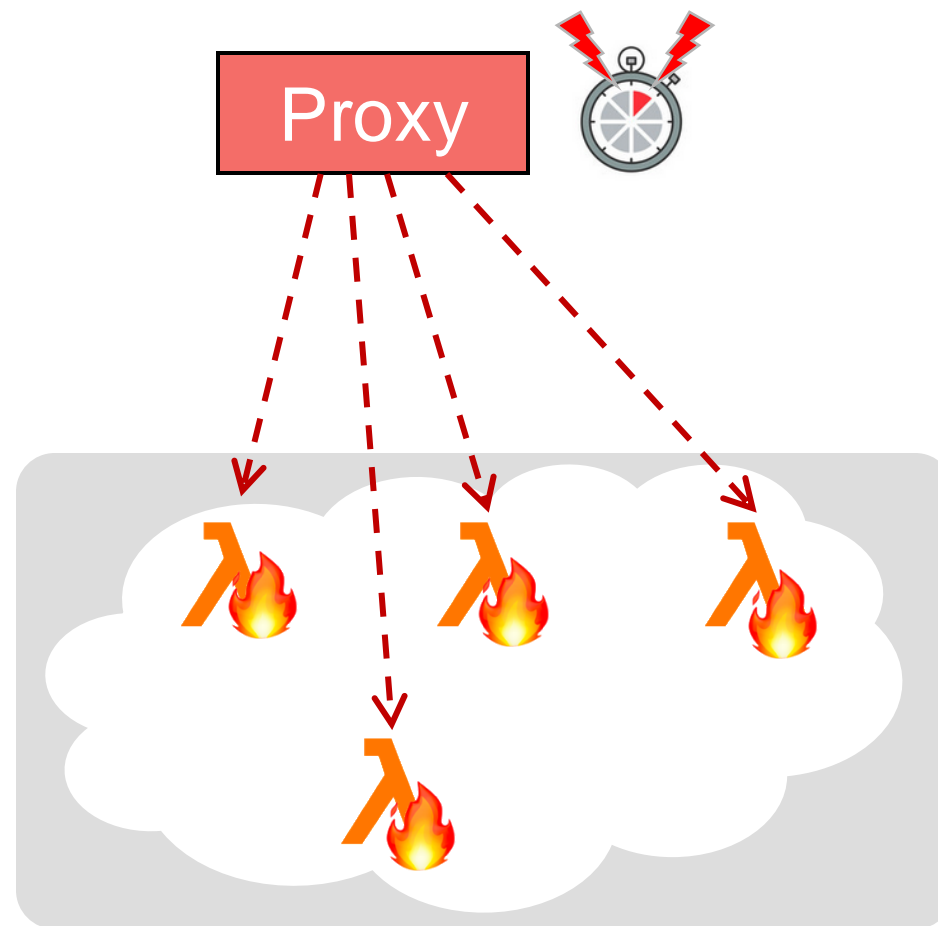
1. Lambda nodes are cached by AWS when not running
 - AWS may reclaim cold Lambda functions after they are idling for a period

Proxy

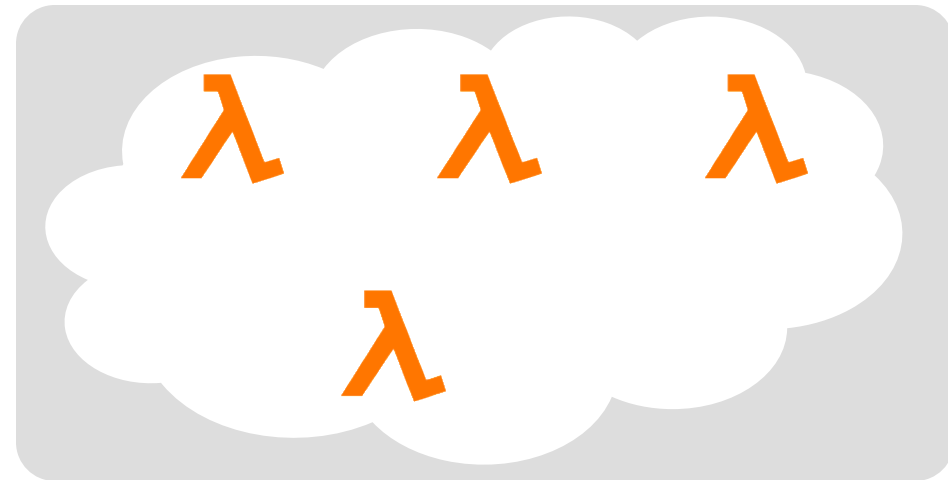


Maximizing data availability: Periodic warm-up

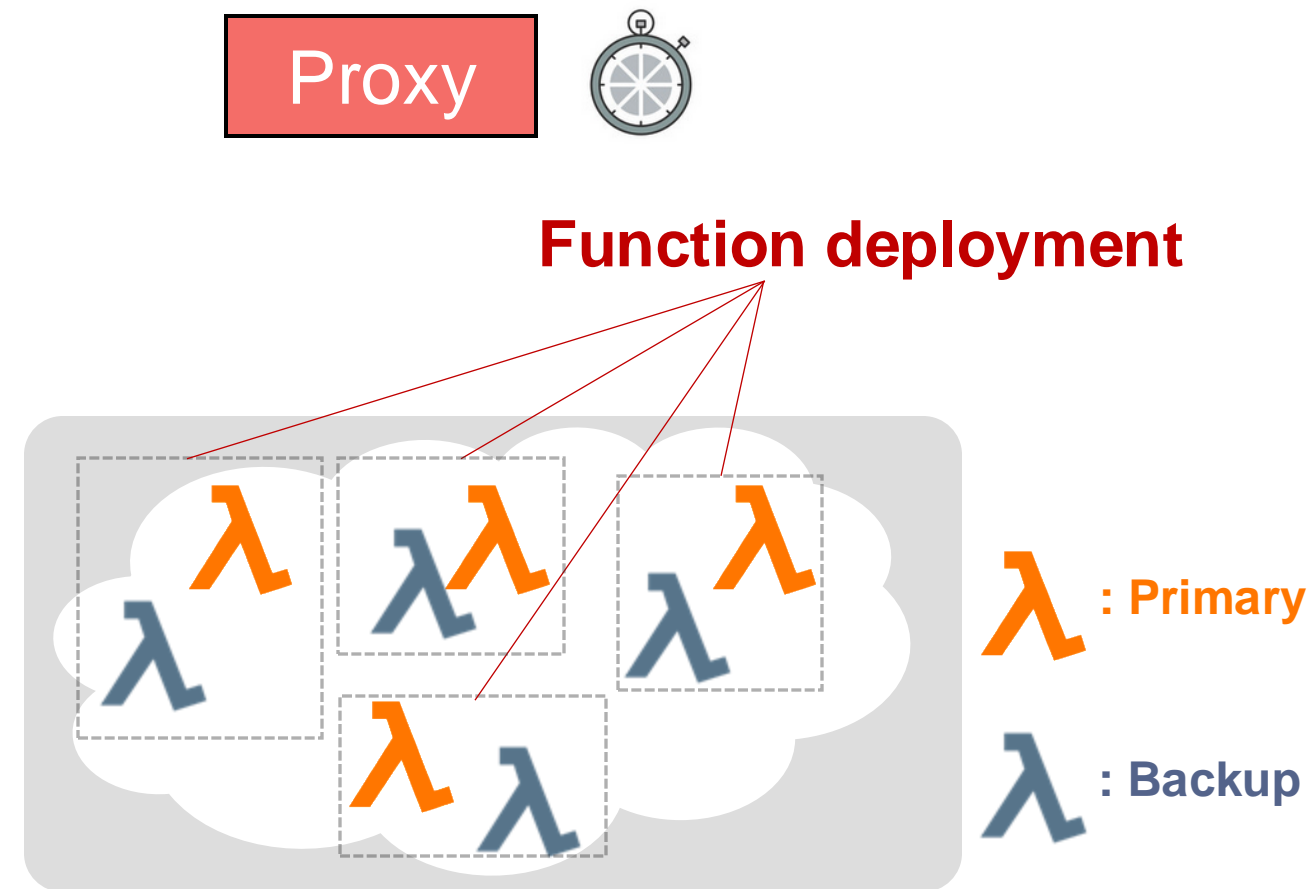
1. Lambda nodes are cached by AWS when not running
2. Proxy periodically invokes sleeping Lambda cache nodes to extend their lifespan



Maximizing data availability: Periodic backup

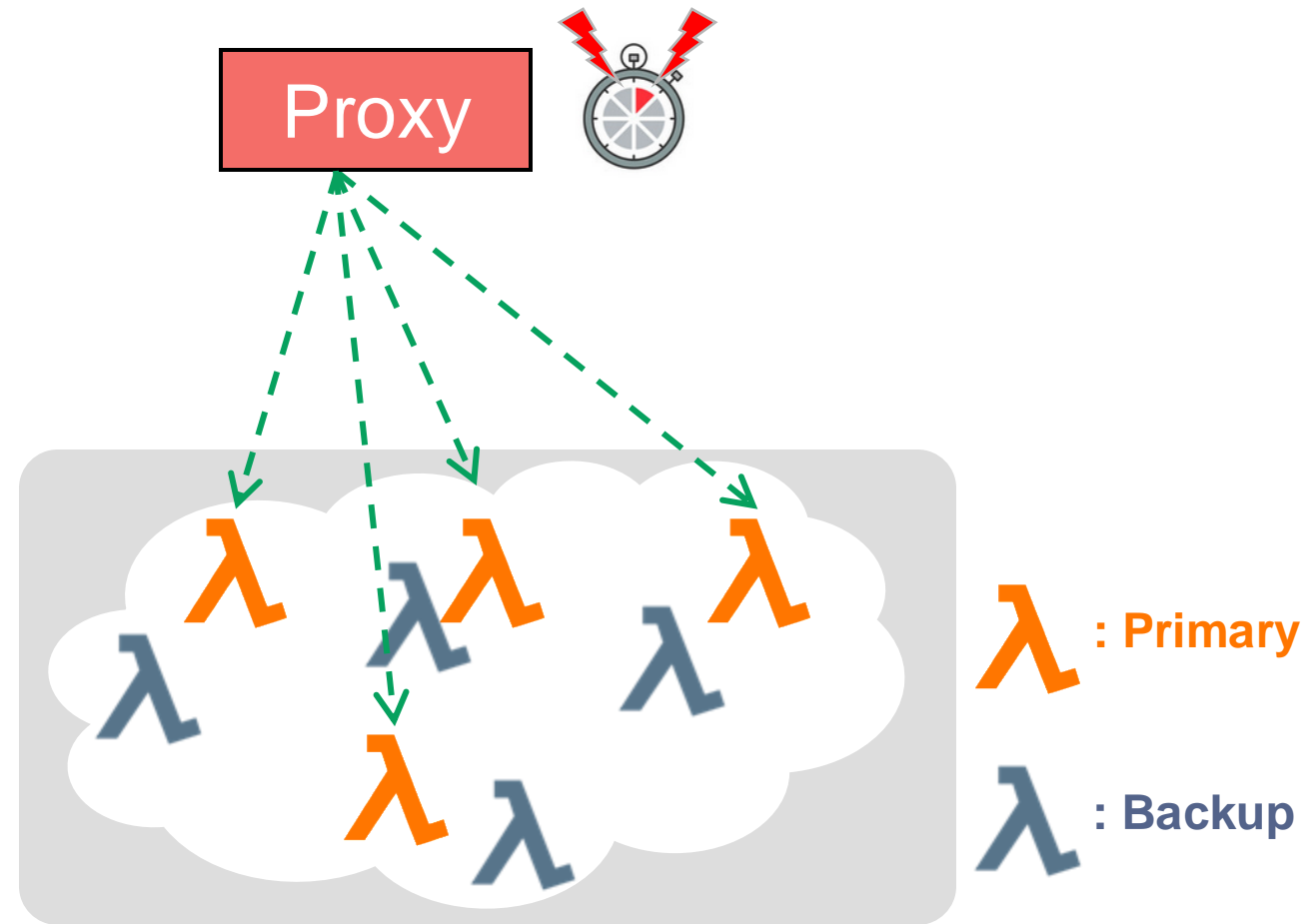


Maximizing data availability: Periodic backup



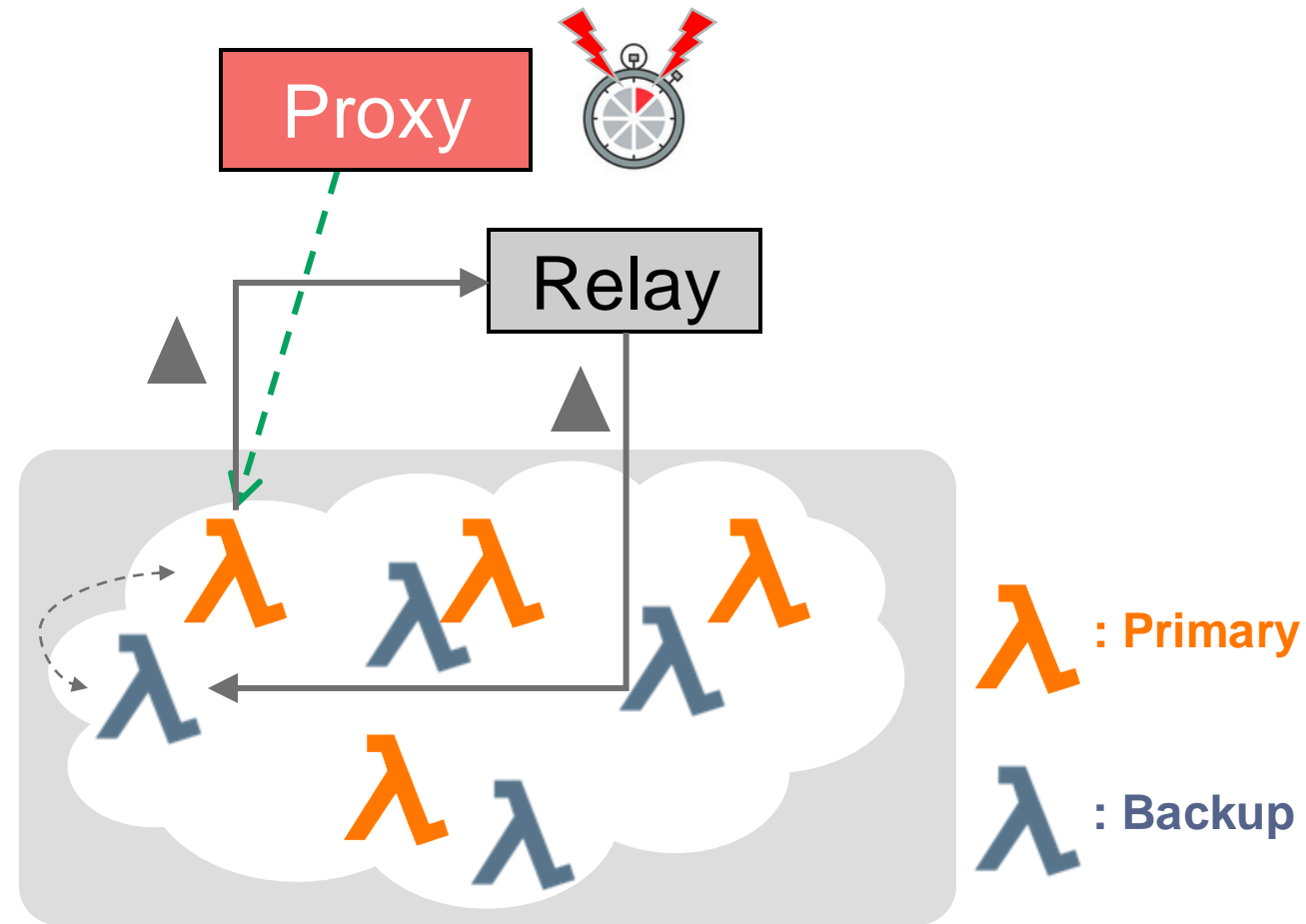
Maximizing data availability: Periodic backup

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

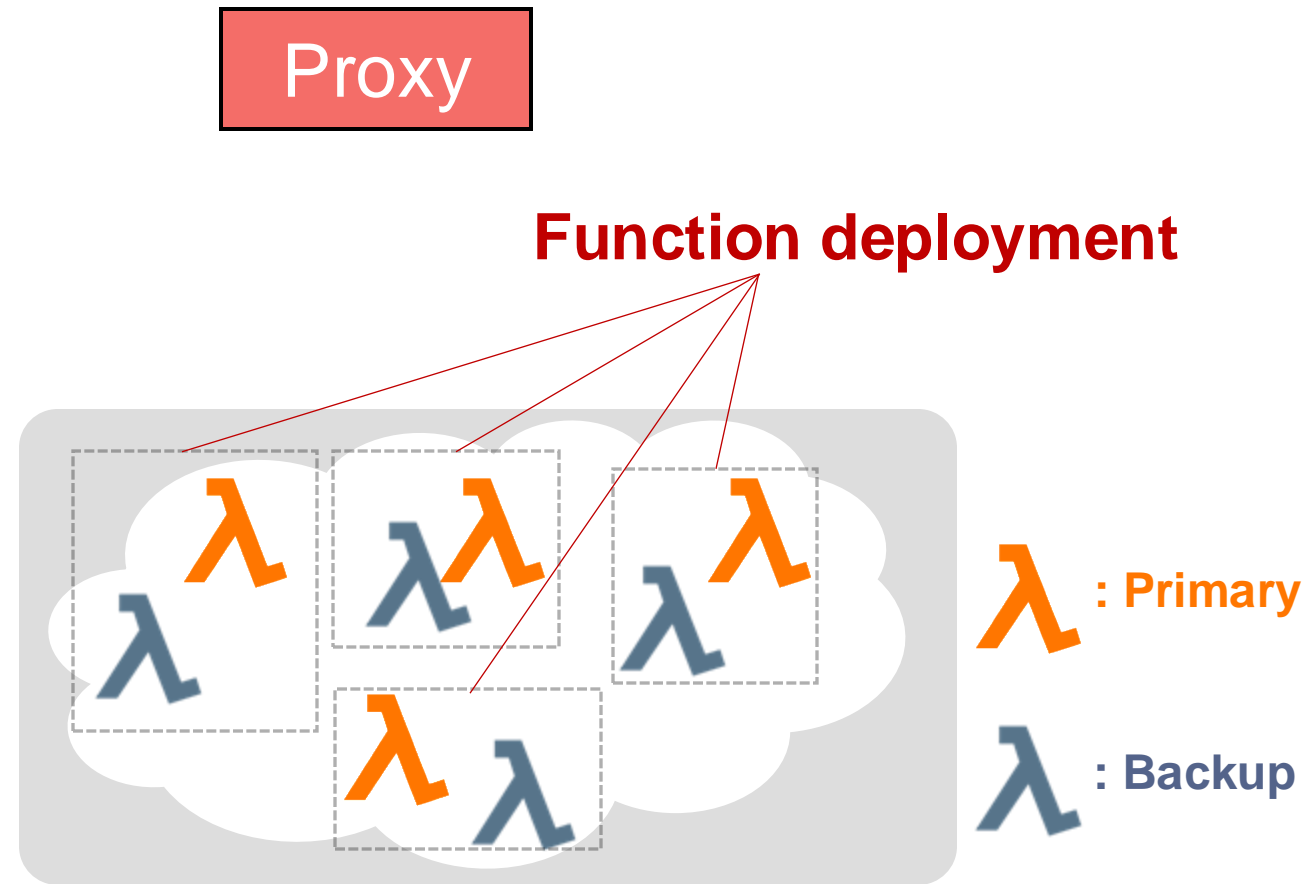


Maximizing data availability: Periodic backup

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

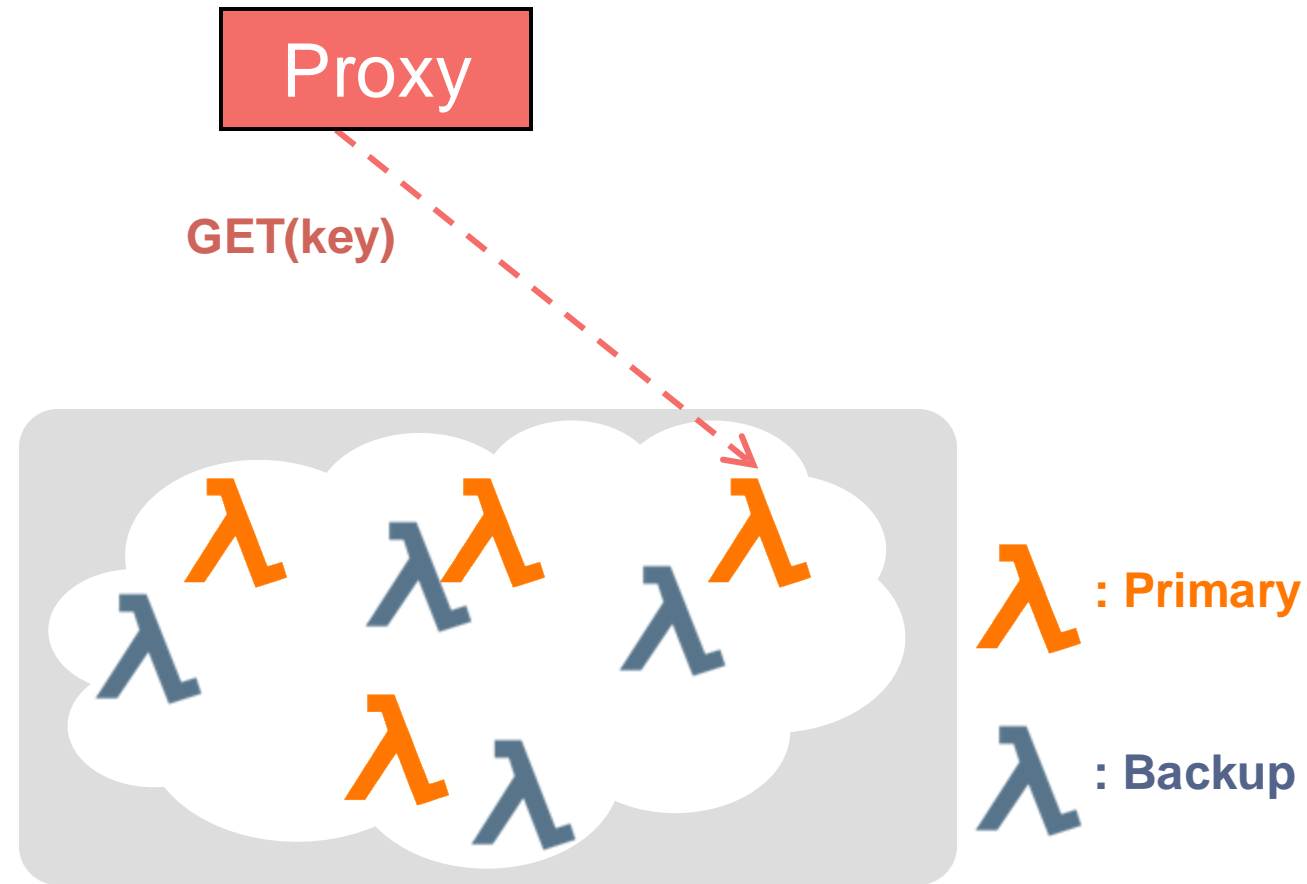


Seamless failover



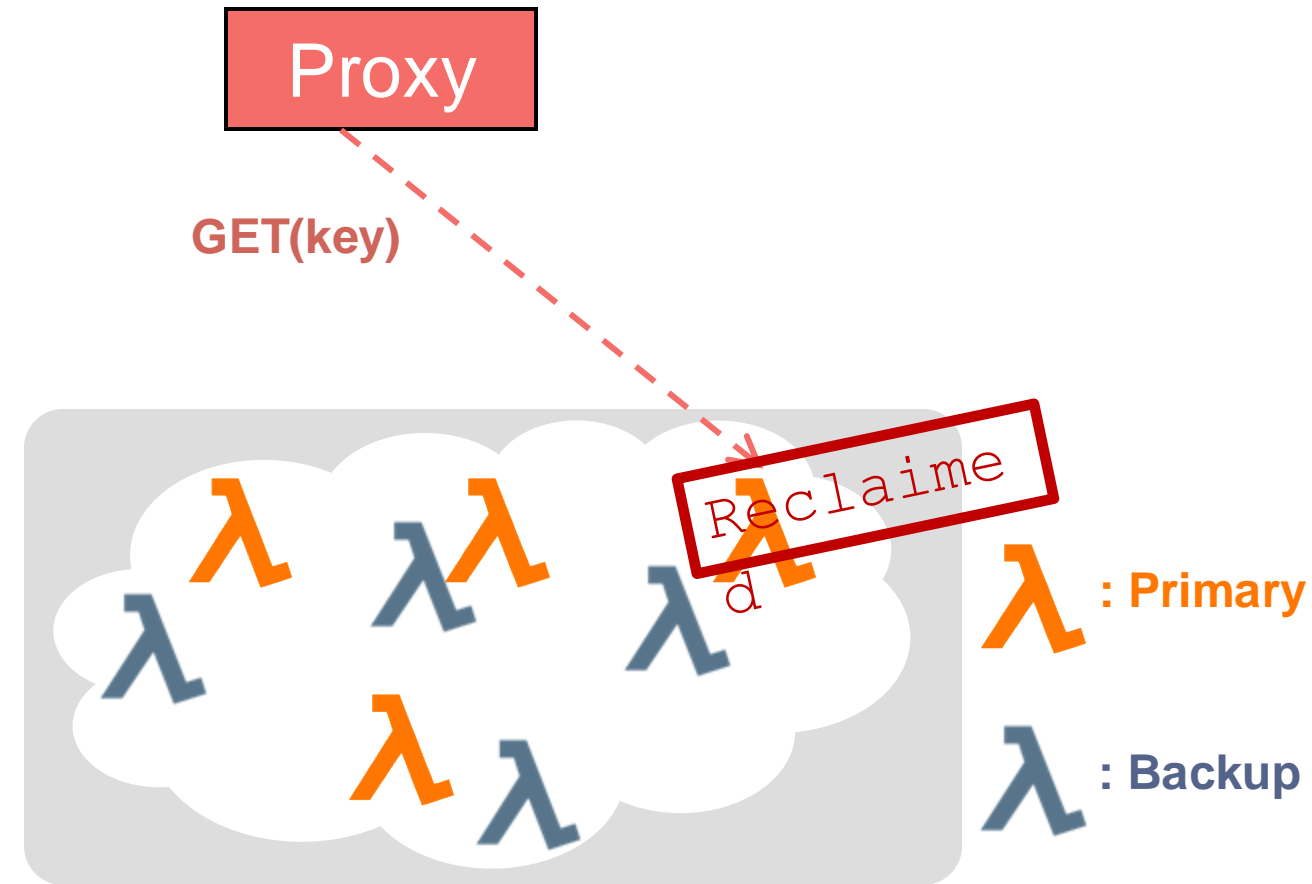
Maximizing data availability: Seamless failover

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



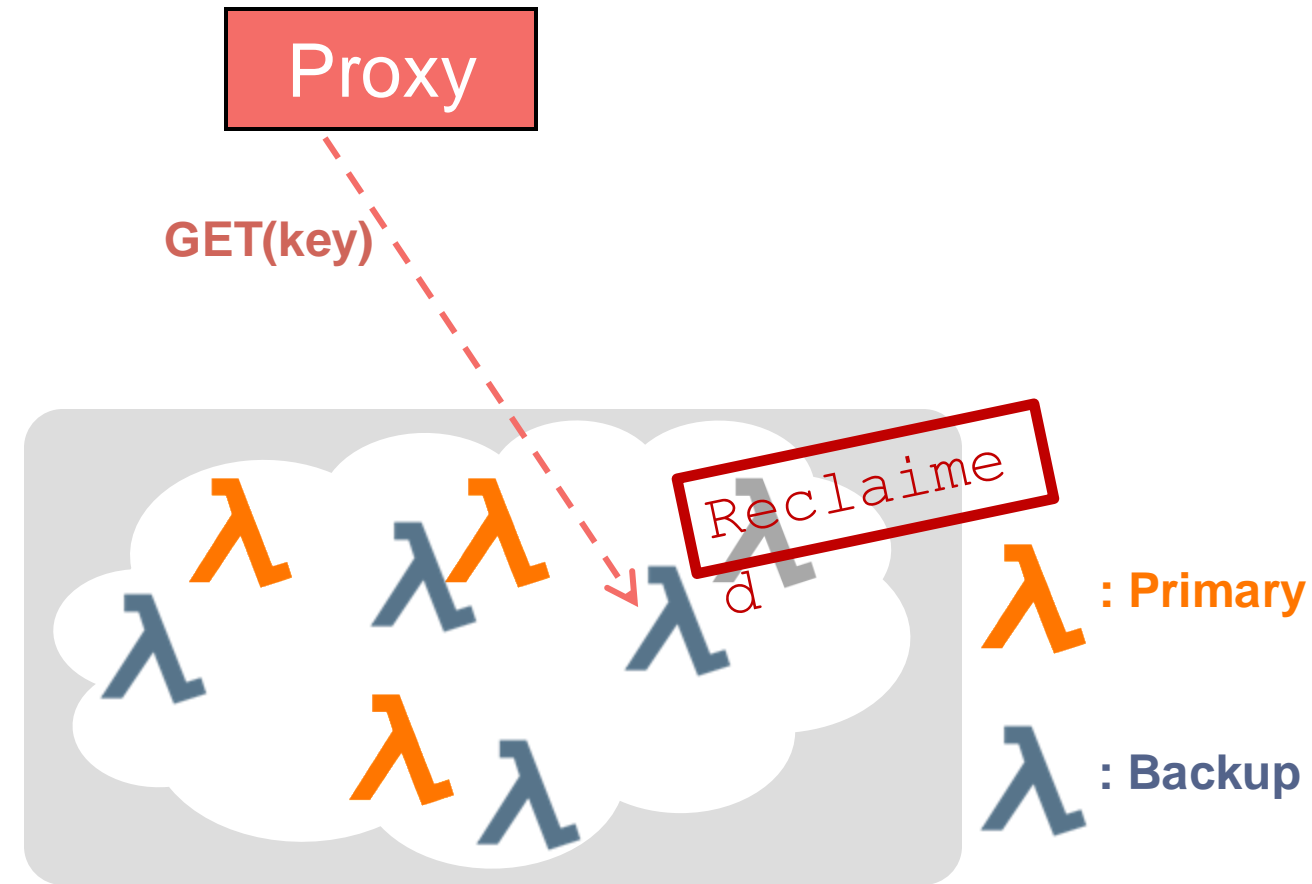
Maximizing data availability: Seamless failover

1. Proxy invokes a Lambda cache node with a GET request
2. Source Lambda gets reclaimed



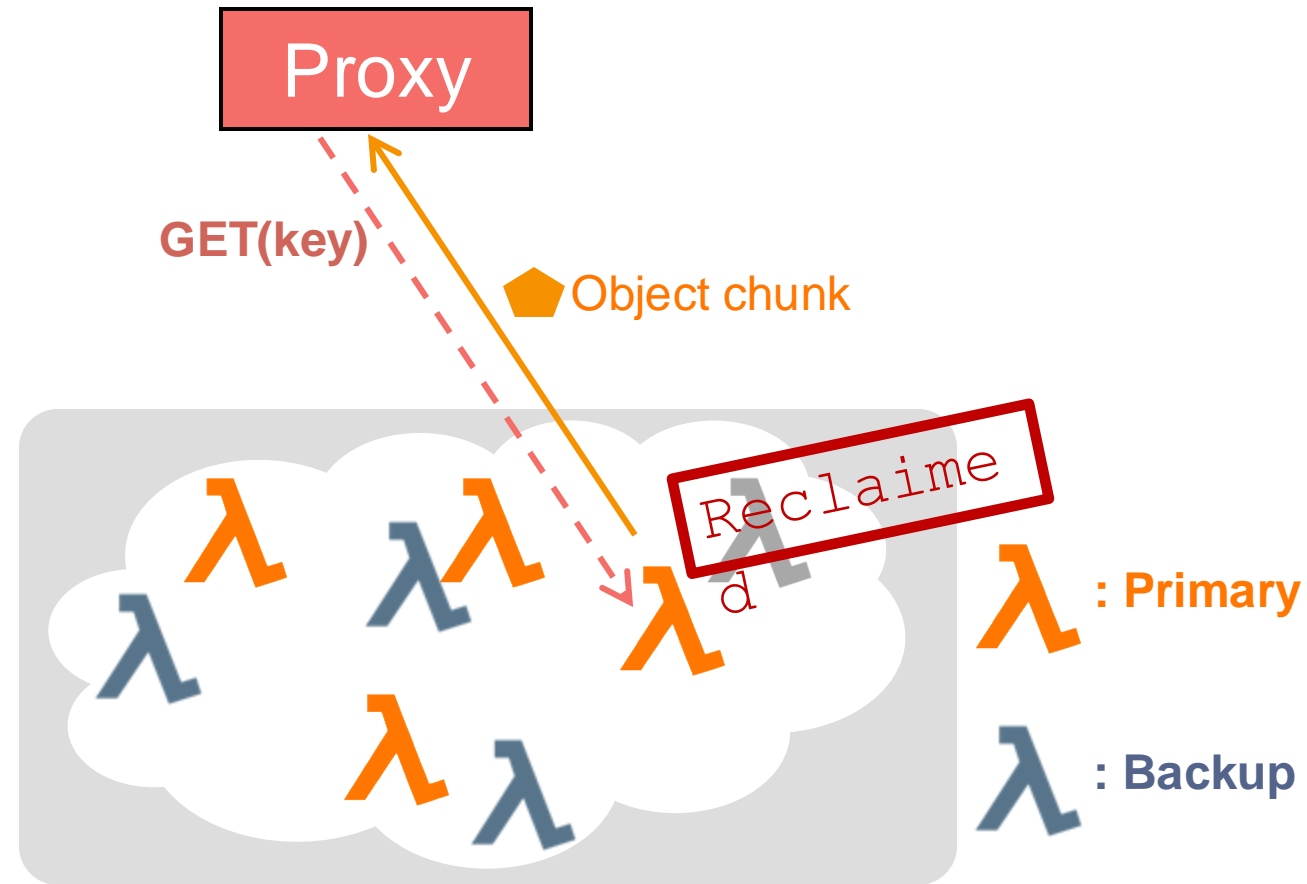
Maximizing data availability: Seamless failover

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

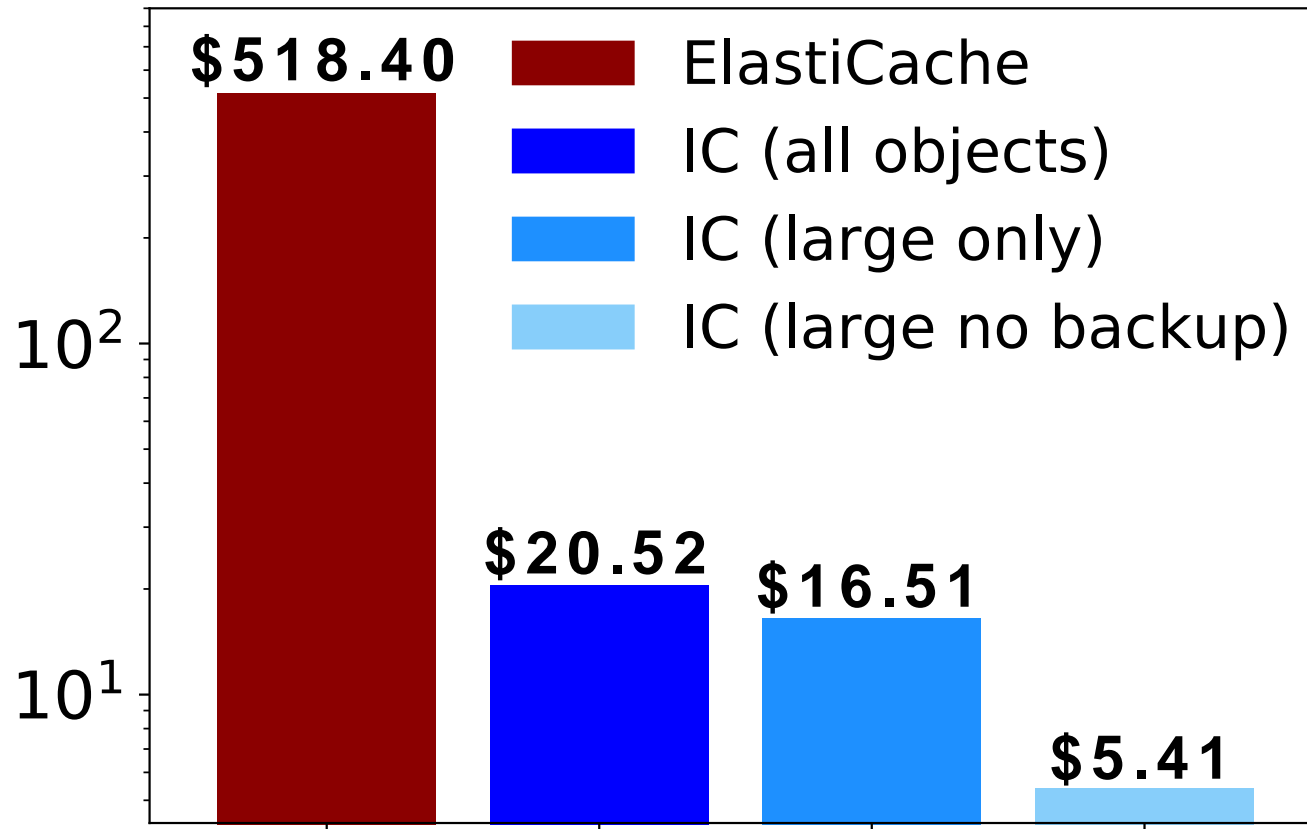


Maximizing data availability: Seamless failover

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



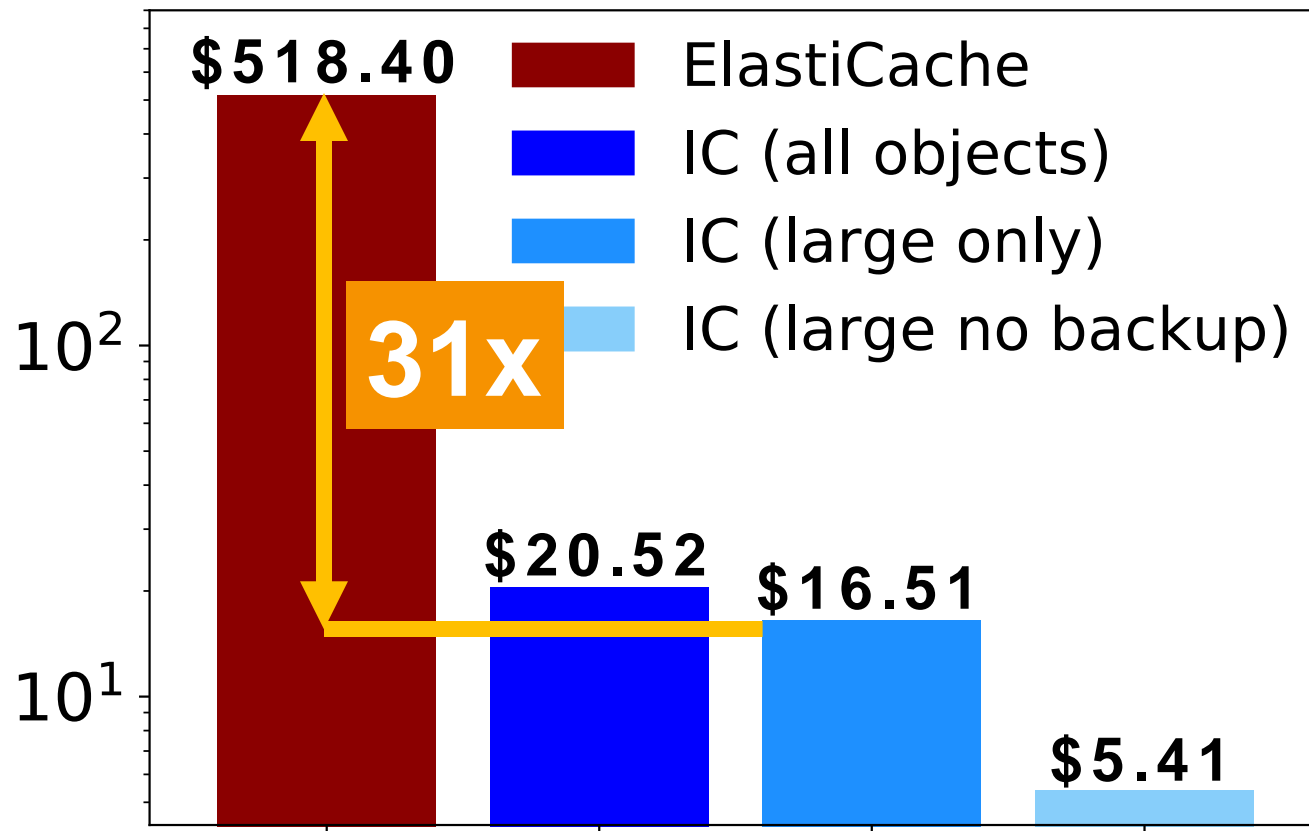
Cost effectiveness of InfiniCache



Workload setup

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

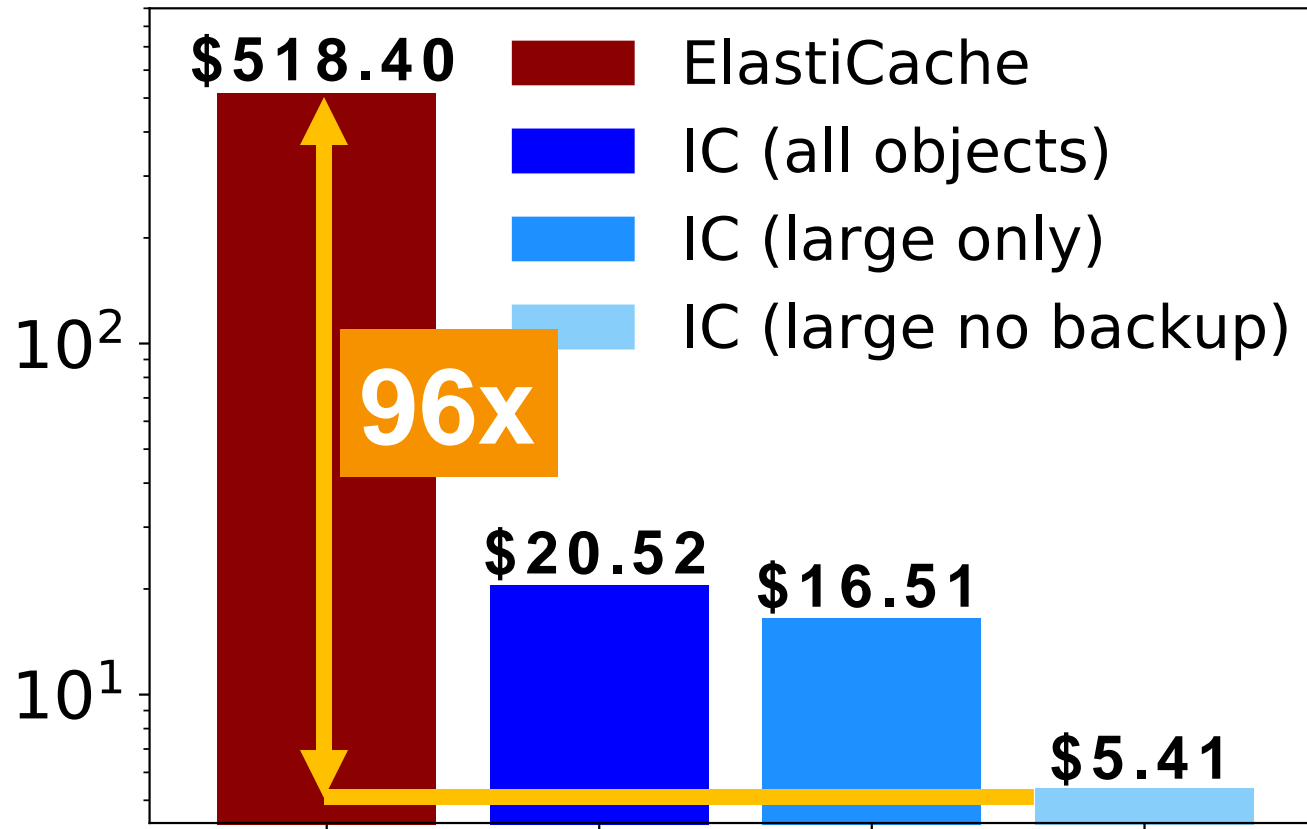
Cost effectiveness of InfiniCache



Workload setup

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

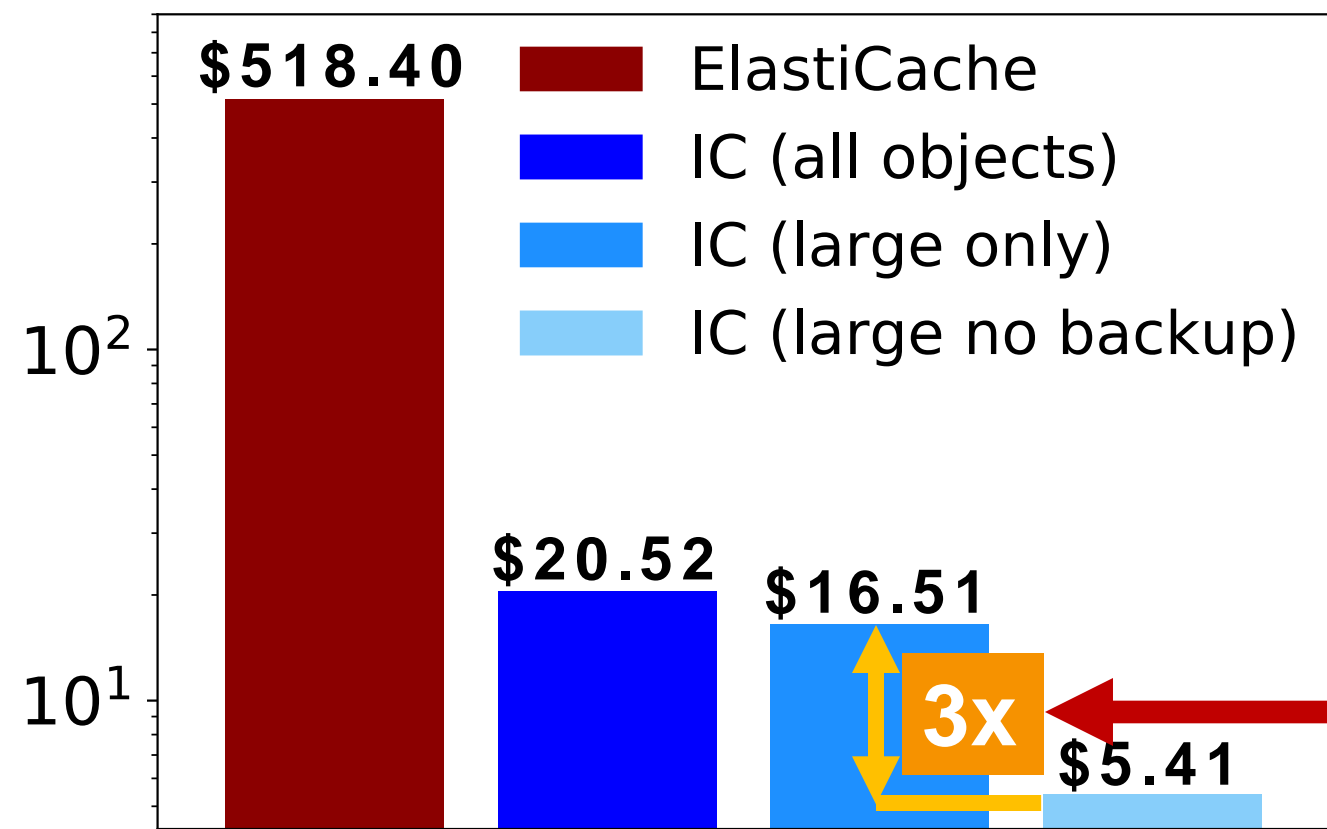
Cost effectiveness of InfiniCache



Workload setup

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

Cost effectiveness of InfiniCache



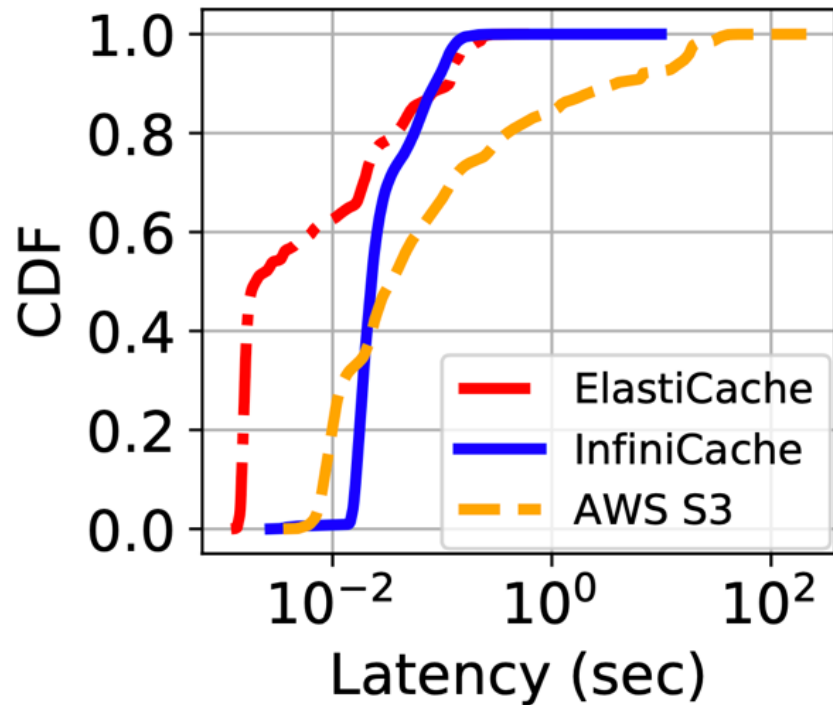
Workload setup

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

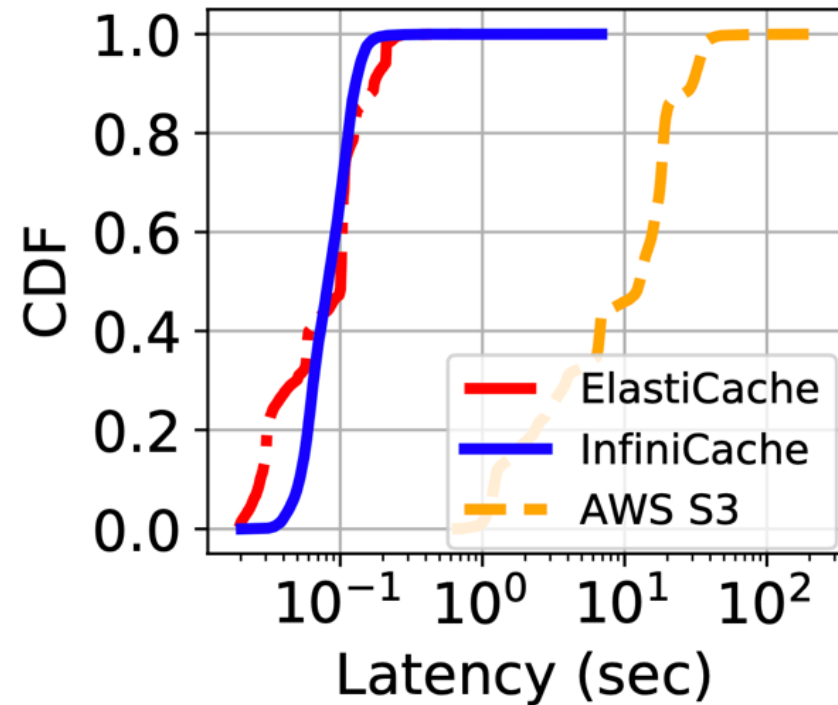
Hit ratio and \$\$ cost tradeoff

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

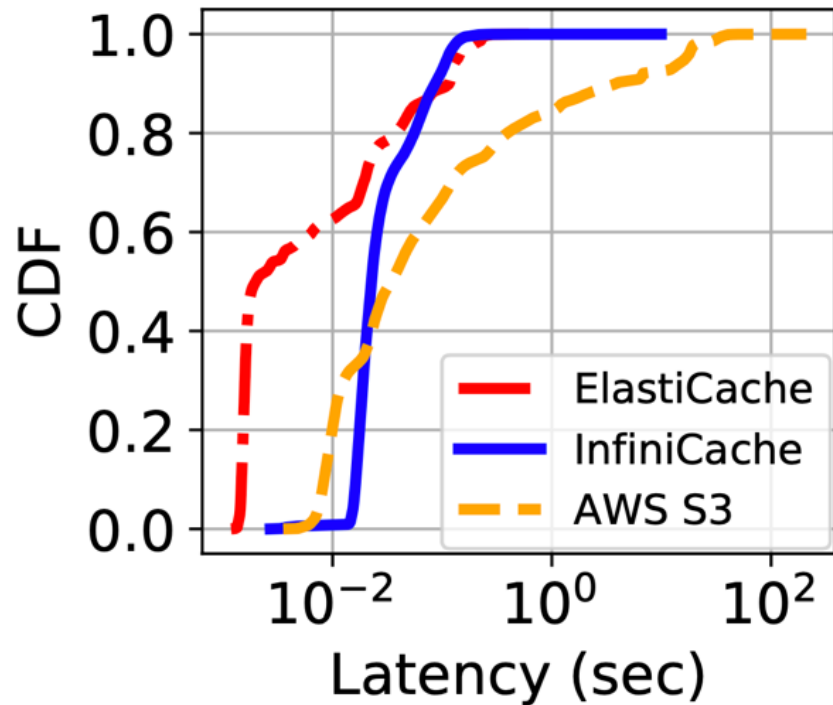


All objects

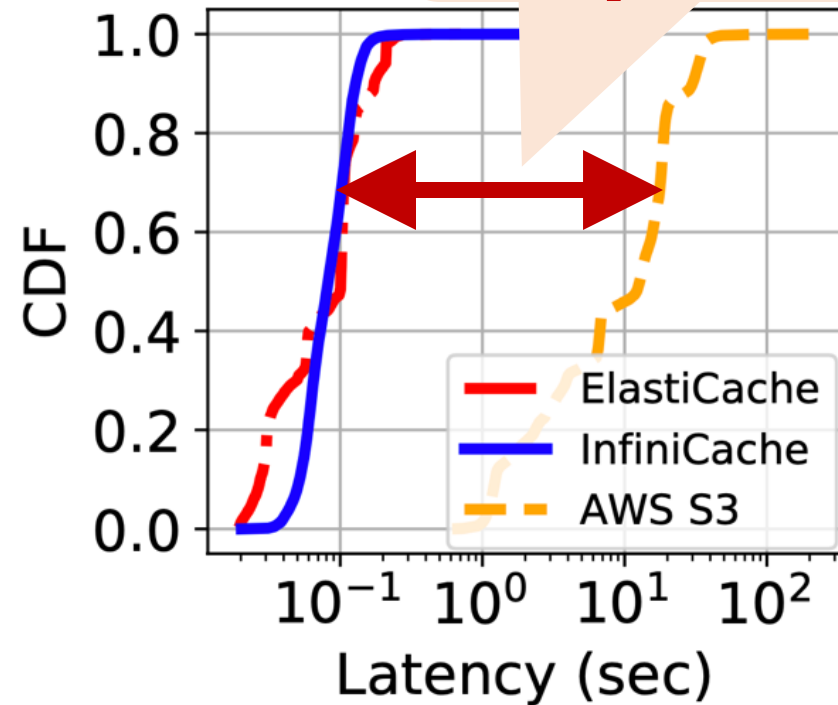


Large objects only

Performance of InfiniCache



All objects



Large objects only

> 100 times improvement

Discussion

- InfiniCache's cost saving benefits have conditions
 - The same condition holds for many different types of serverless/FaaS apps
- Unit time \$ cost increases with the access rate

