InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache

Ao Wang*, **Jingyuan Zhang***, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, Yue Cheng

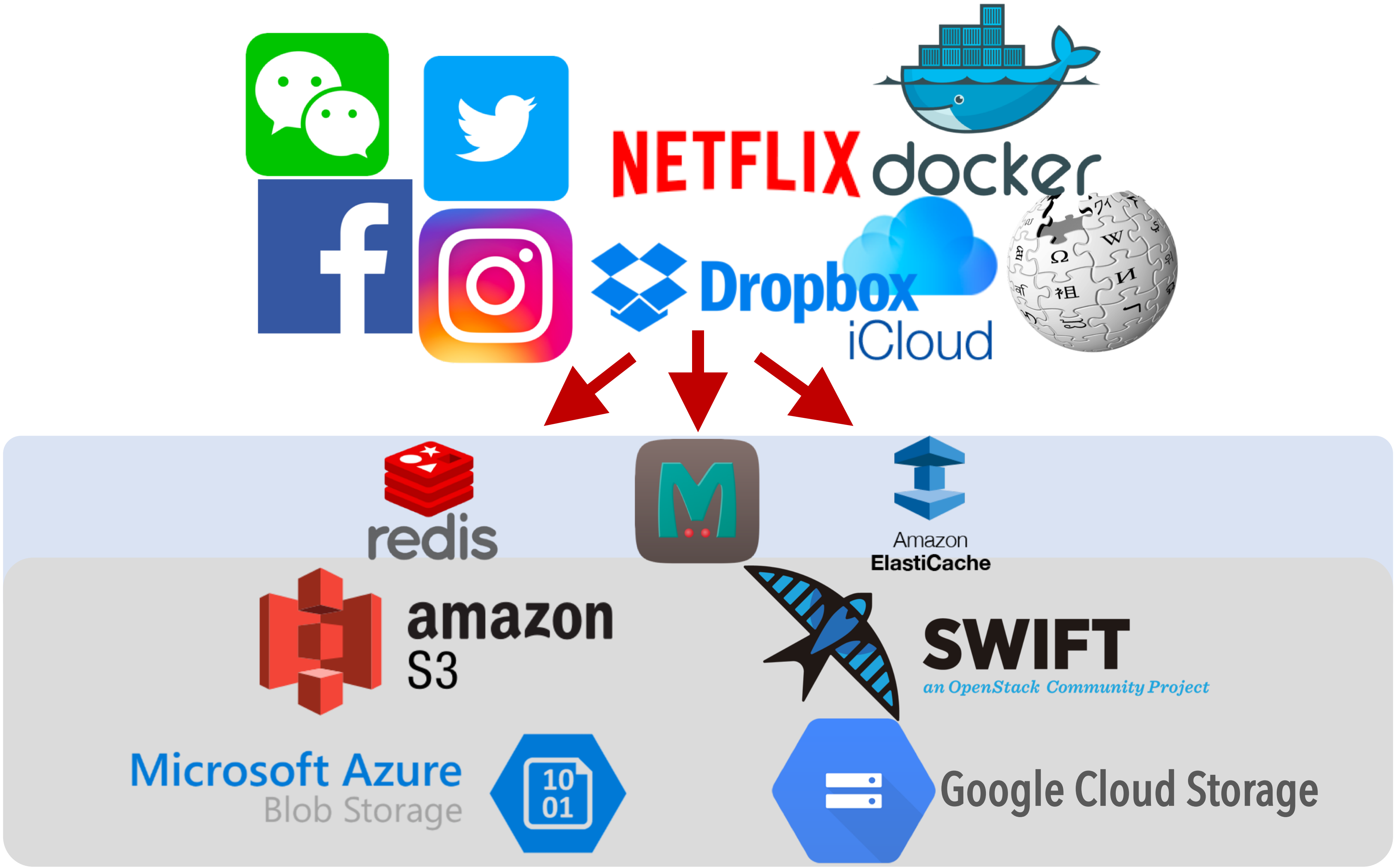
George Mason University

University of Nevada, Reno

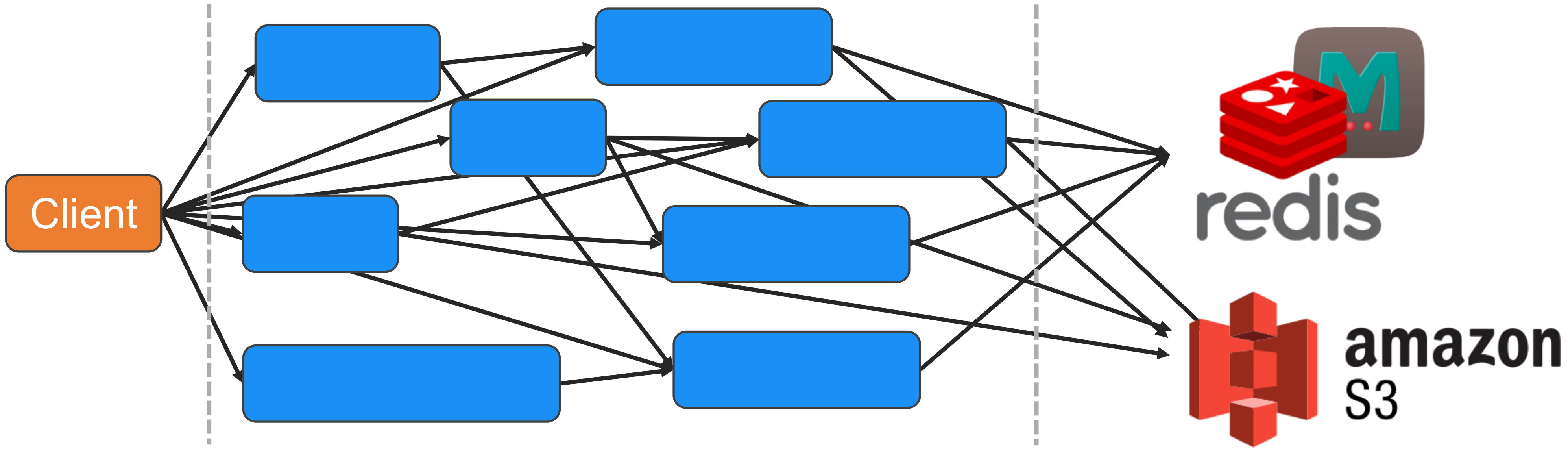
IBM

* These authors contributed equally to this work

Web applications are storage-intensive



Web applications – heterogeneous I/O



Microservices

Case study: IBM Docker registry workloads

- IBM Cloud container registry service across 75 days during 2017
- Selected data centers: Dallas & London

Case study: IBM Docker registry workloads

- Object size distribution
- Large object reuse patterns
- Storage footprint

Case study: IBM Docker registry workloads

- Object size distribution
- Large object reuse patterns
- Storage footprint

Extreme variability in object sizes:

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

Case study: IBM Docker registry workloads

- Object size distribution
- Large object reuse patterns
- Storage footprint

Caching large objects is beneficial:

- > 30% large object (>10MB) access 10+ times
- Around 45% of them got reused within 1 hour

Case study: IBM Docker registry workloads

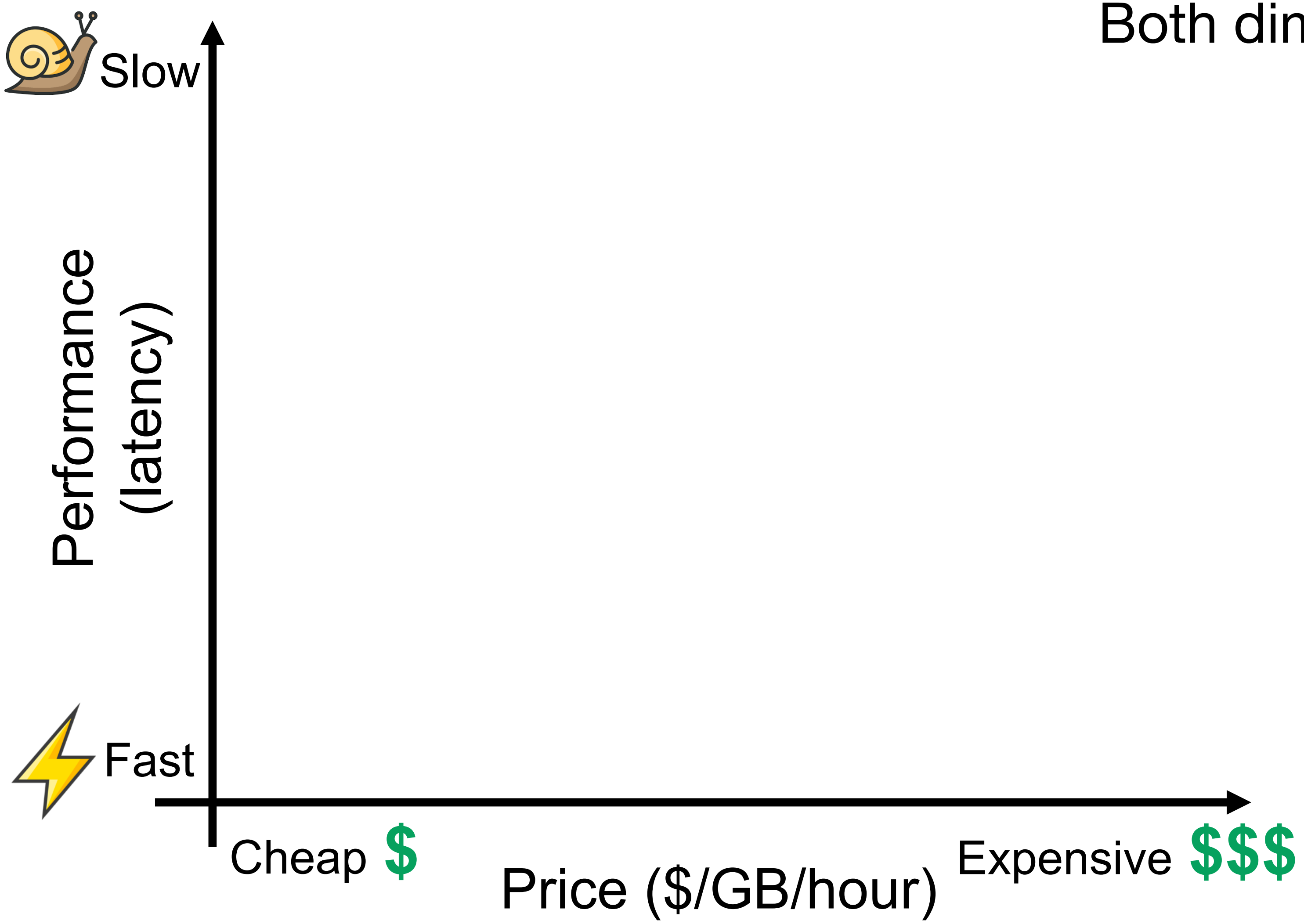
- Object size distribution
- Large object reuse patterns
- Storage footprint

Extreme tension between small and large objects:

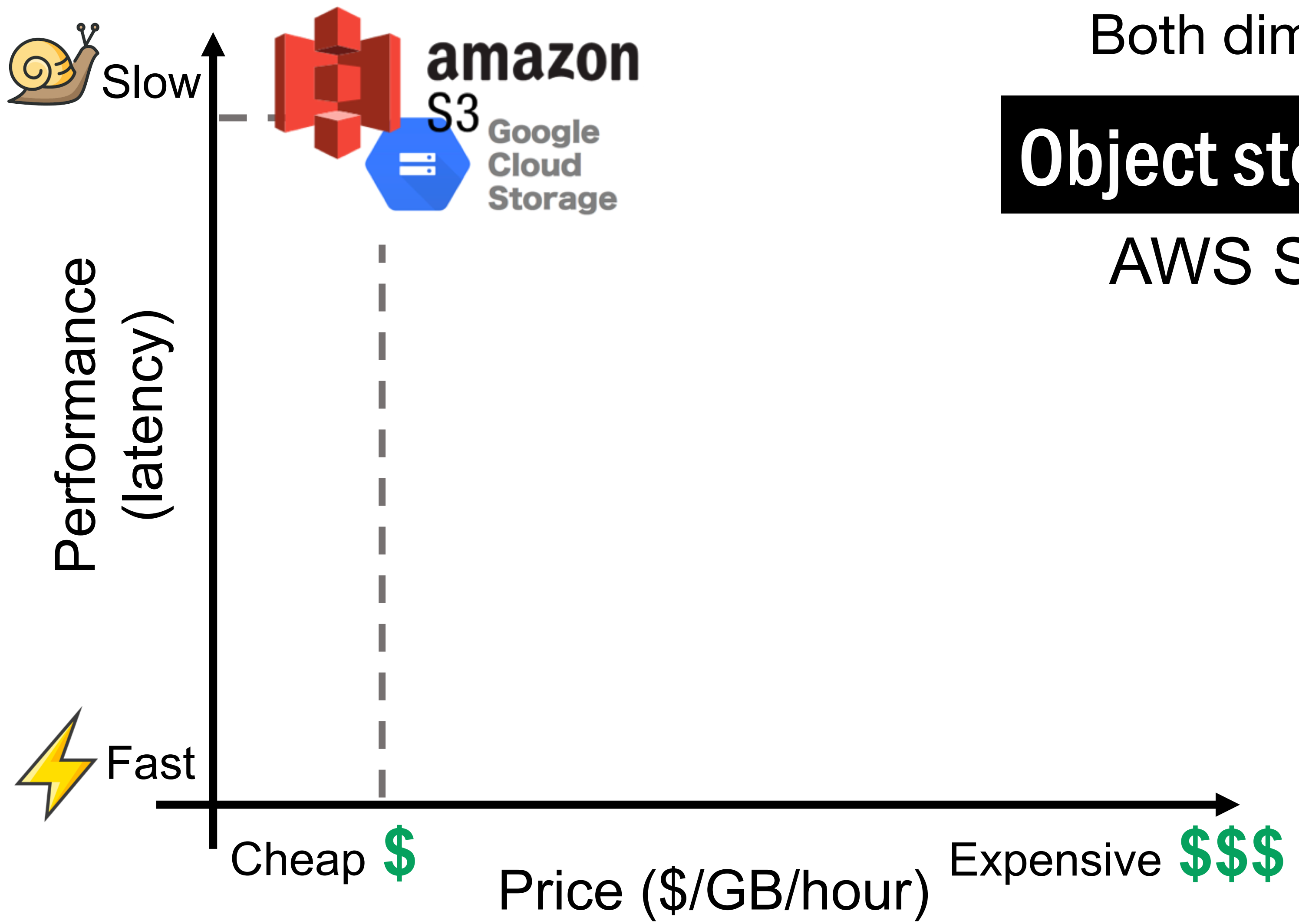
➢ Large objects (>10MB) occupy 95% storage footprint

Existing cloud storage solutions

Both dimensions: the lower the better



Large objects managed by cloud object stores

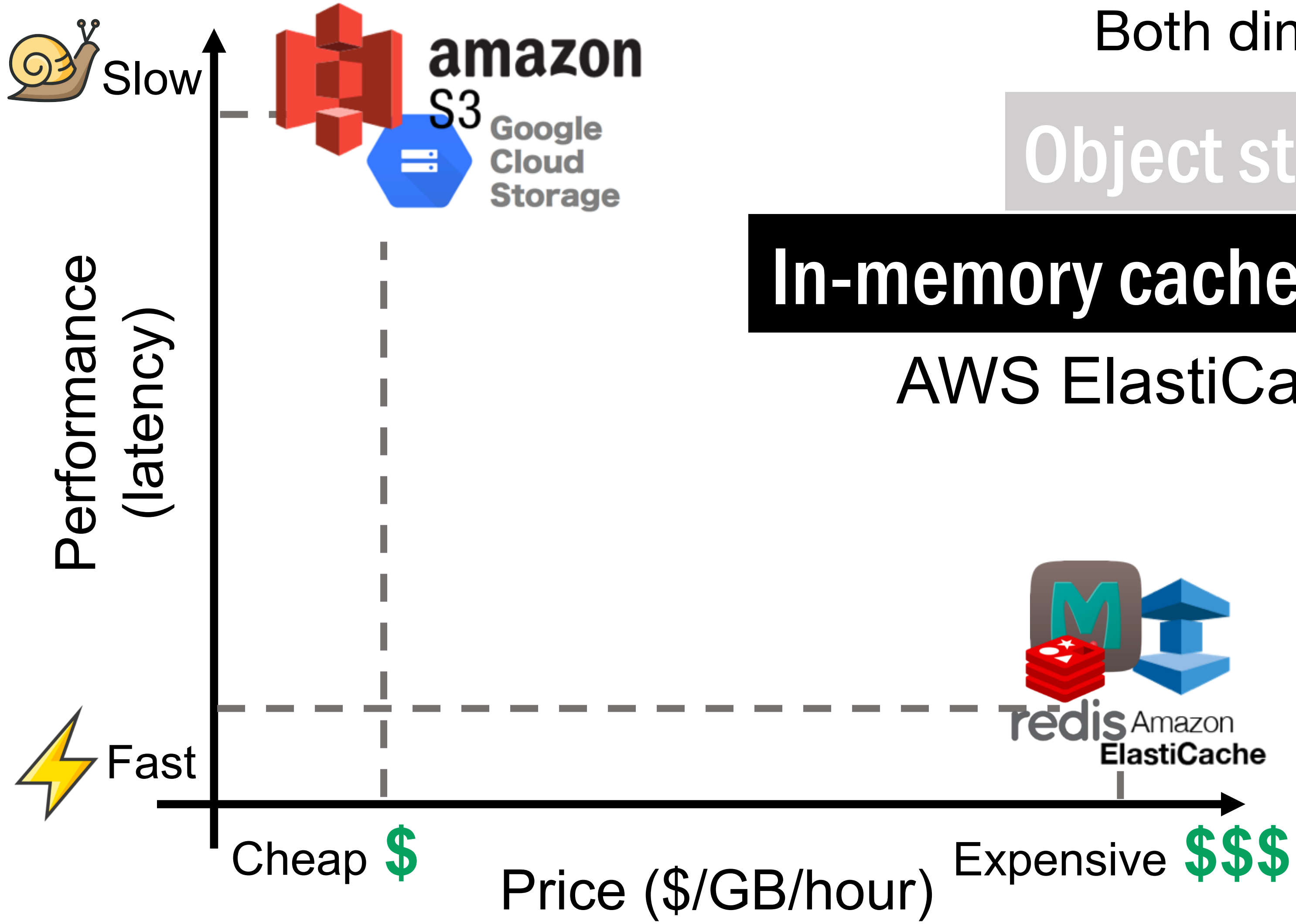


Both dimensions: the lower the better

Object stores are cheap but too slow

AWS S3: $0.023 per GB per month

Small objects accelerated by in-memory caches



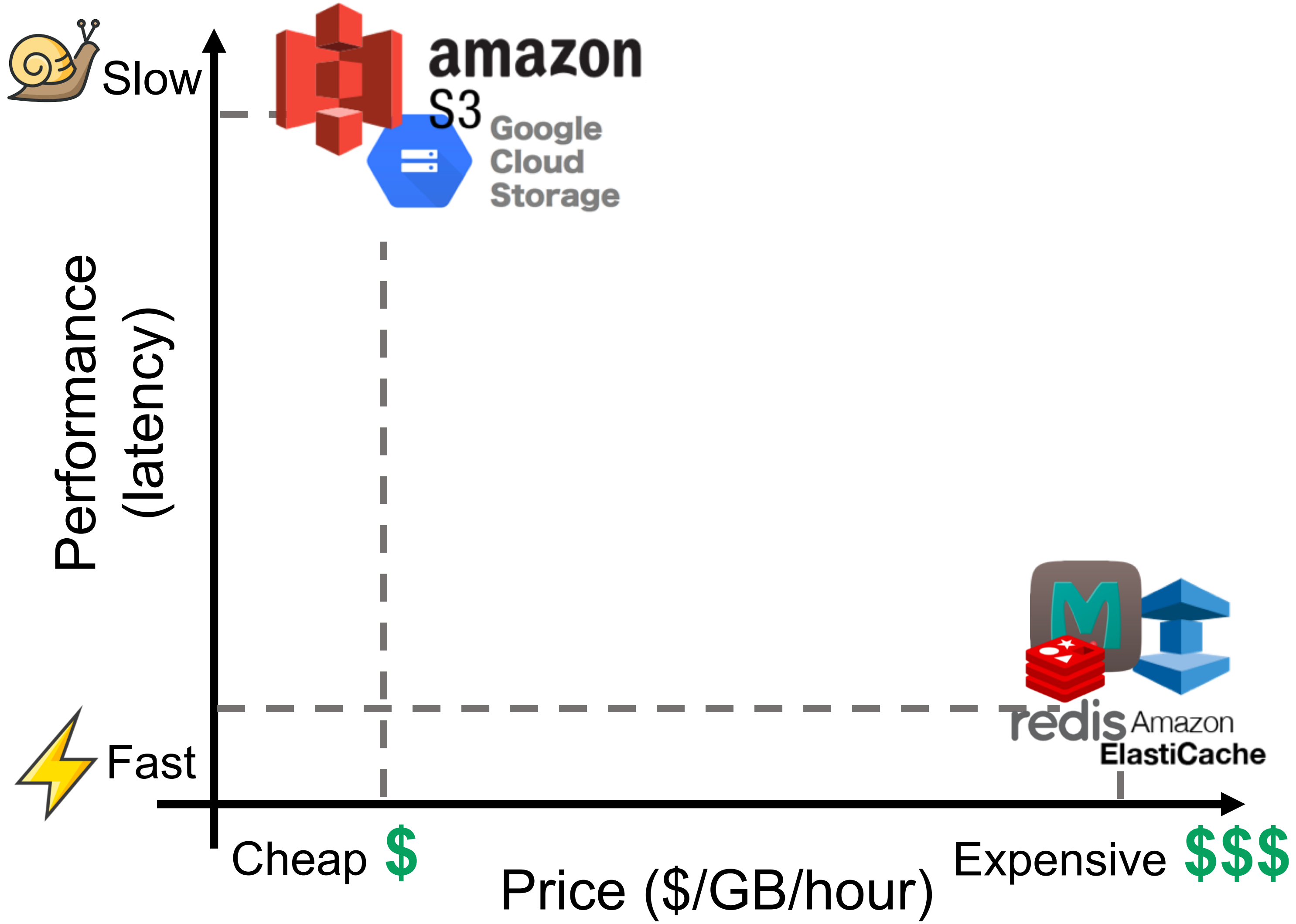
Both dimensions: the lower the better

Object stores are cheap but too slow

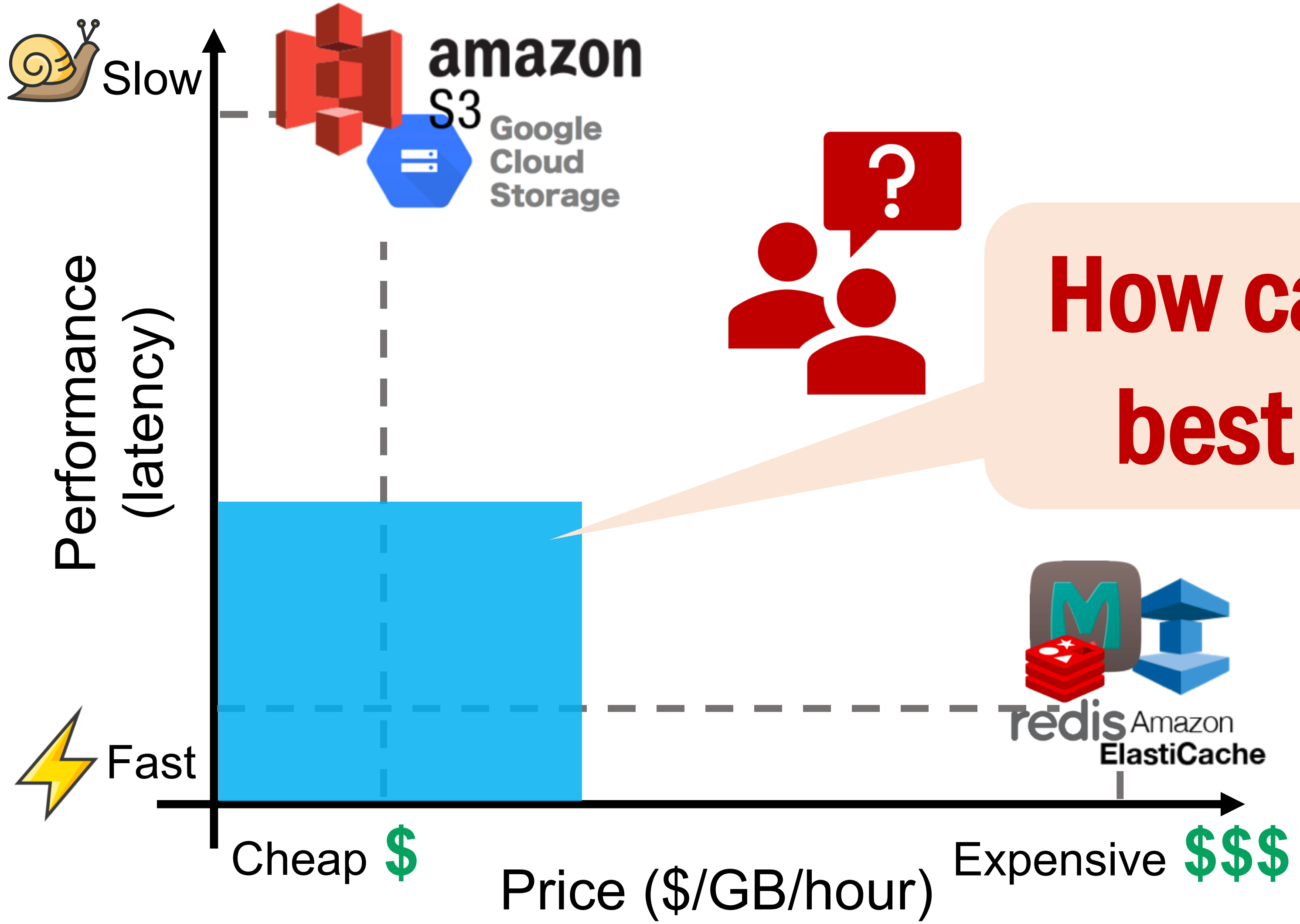
In-memory caches are fast but too expensive

AWS ElastiCache: $0.016 per GB per hour

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



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



How can we achieve the best of both worlds?

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

Requires rethinking about a new cloud cache/storage model that achieves both cost effectiveness and high-performance!

InfiniCache: A cost-effective and high-performance in-memory caching solution atop Serverless Computing platform

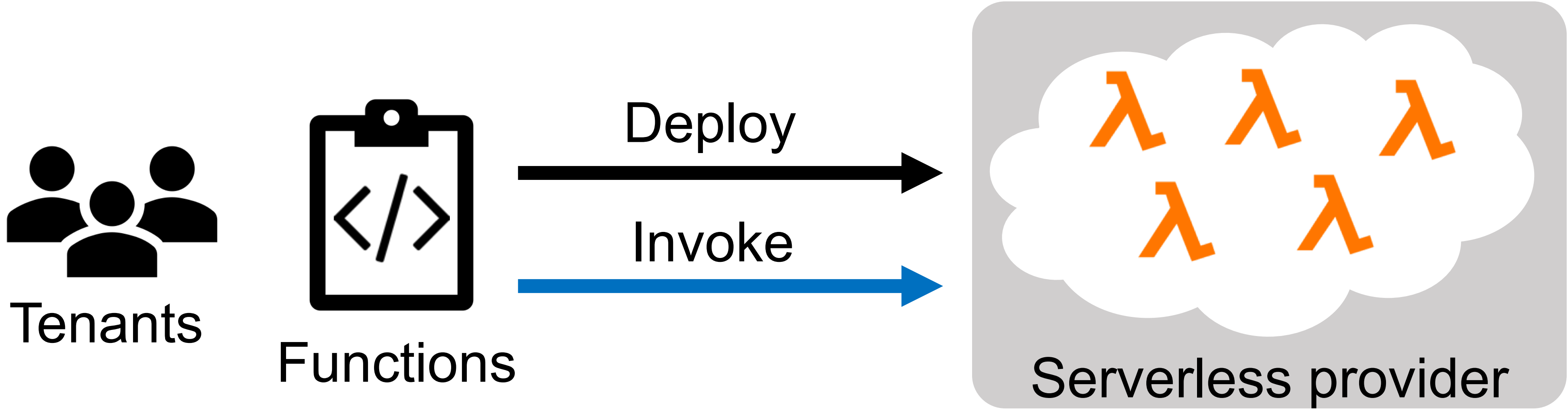
- **Insight #1:** **Serverless functions' <CPU, Mem> resources are pay-per-use**
- **Insight #2:** **Serverless providers offer "free" function caching for tenants**

InfiniCache: A cost-effective and high-performance in-memory caching solution atop Serverless Computing platform

- **Insight #1:** Serverless functions' <CPU, Mem> resources are pay-per-use → **Cost-effectiveness**
- **Insight #2:** Serverless providers offer "free" function caching for tenants → **High-performance**

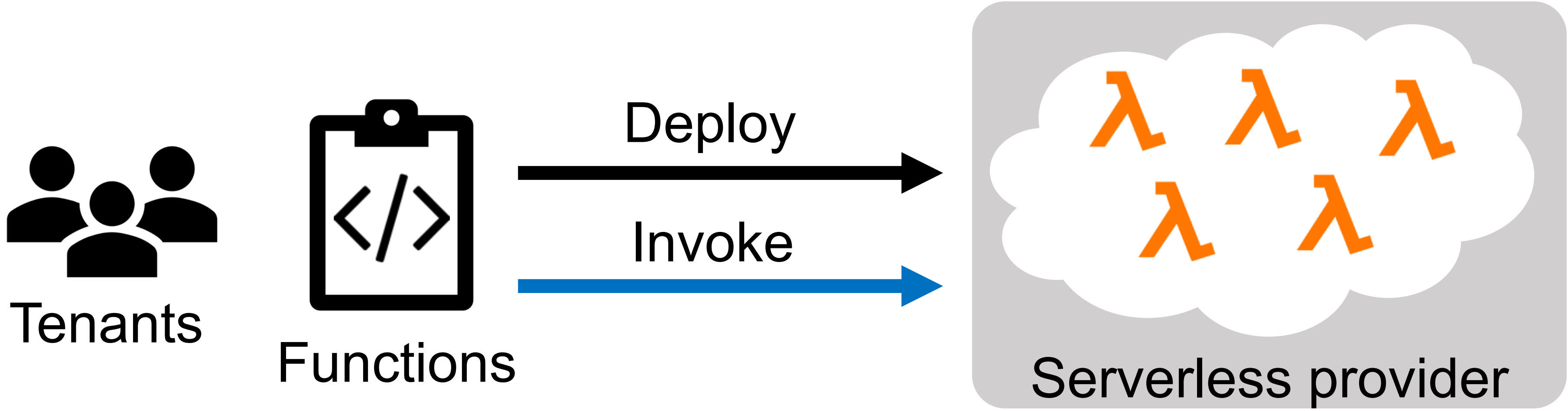
A primer on Serverless Computing

- Serverless computing enables cloud tenants to launch short-lived tasks (i.e., Lambda functions) with high elasticity and fine-grained resource billing



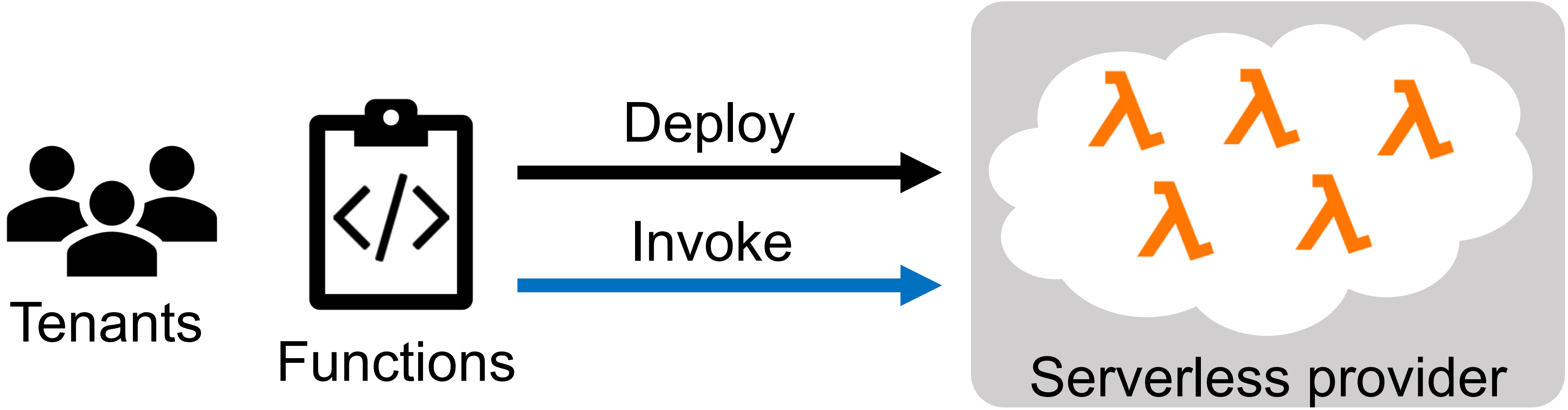
A primer on Serverless Computing

- Serverless computing enables cloud tenants to launch short-lived tasks (i.e., Lambda functions) with high elasticity and fine-grained resource billing
- Function: basic unit of deployment. Application consists of multiple serverless functions



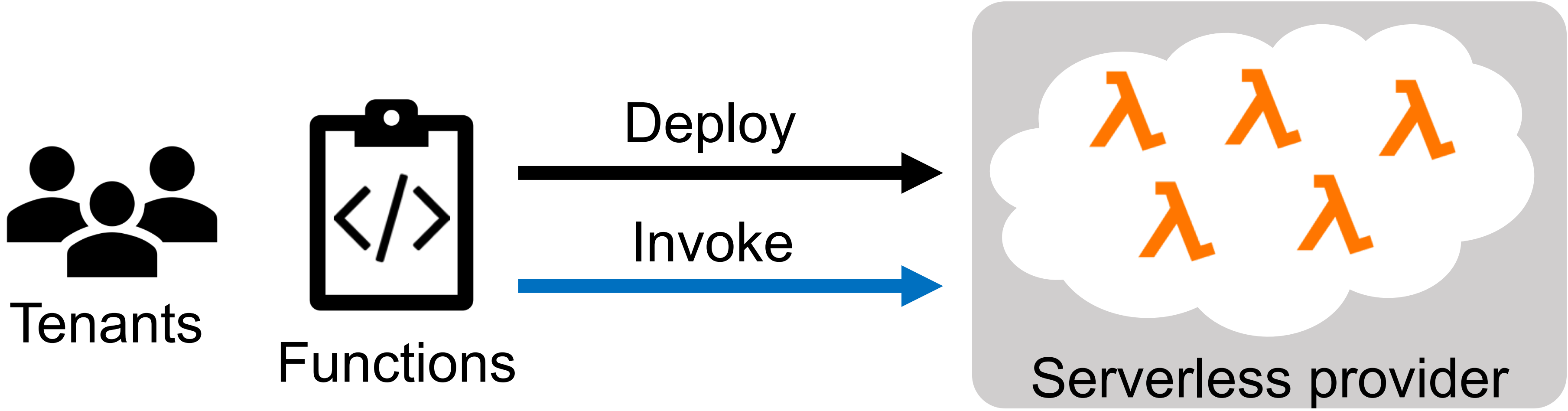
A primer on Serverless Computing

- Serverless computing enables cloud tenants to launch short-lived tasks (i.e., Lambda functions) with high elasticity and fine-grained resource billing
- Function: basic unit of deployment. Application consists of multiple serverless functions
- Popular use cases: Backend APIs, data processing…



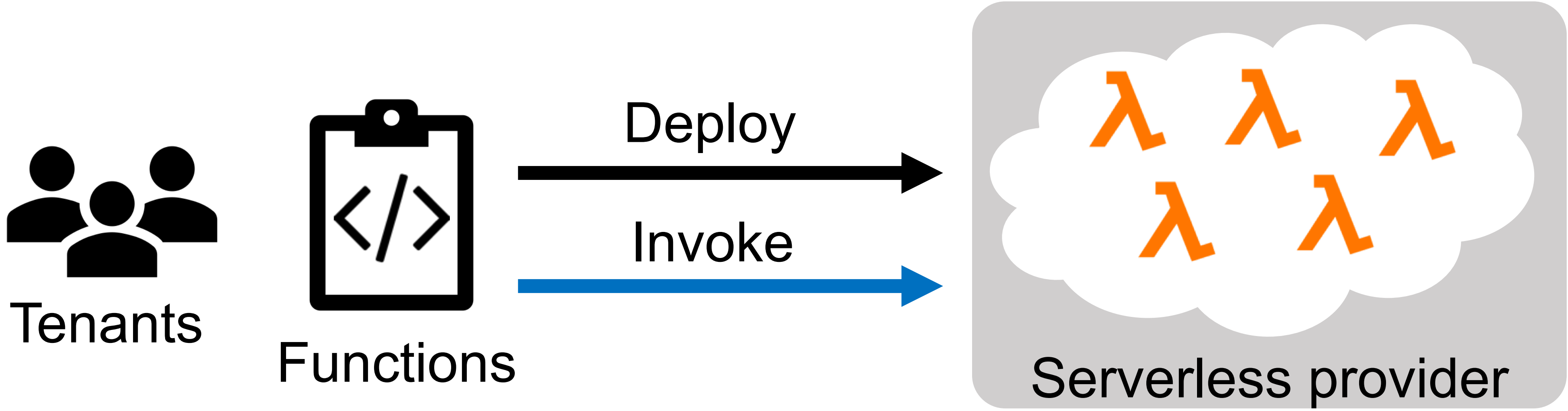
Serverless Computing is desirable

- Pay-per-use pricing model
  - AWS Lambda: $0.2 per 1M invocations
    $0.00001667 for every GB-sec



Serverless Computing is desirable

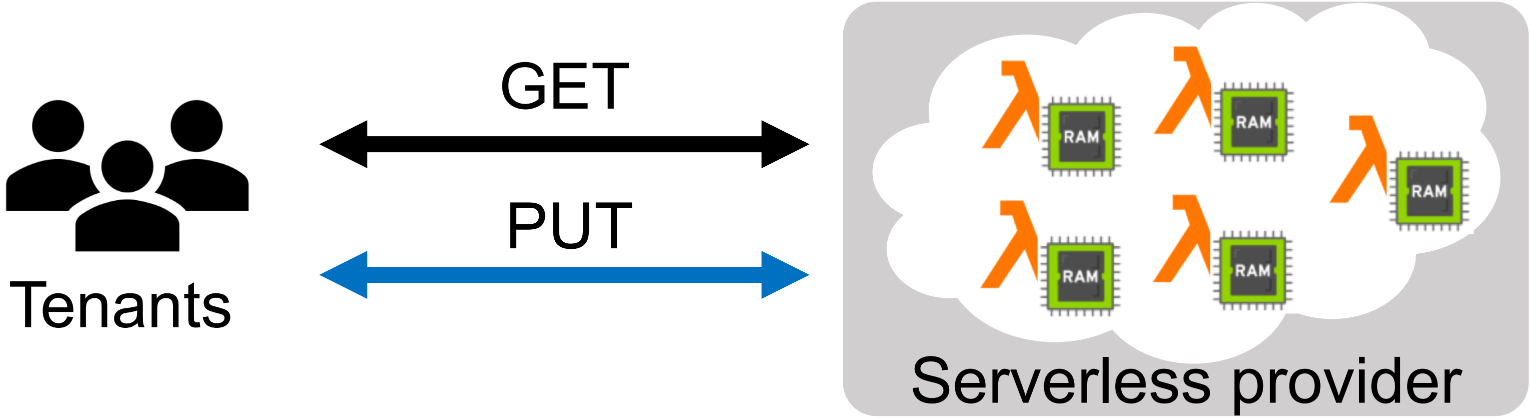
- Pay-per-use pricing model
  - AWS Lambda: $0.2 per 1M invocations
    $0.00001667 for every GB-sec
- Short-term function caching
  - Provider caches triggered functions in memory without charging tenants



Serverless Computing is desirable

- Pay-per-use pricing model
  - AWS Lambda: $0.2 per 1M invocations
    $0.00001667 for every GB-sec
- Short-term function caching
  - Provider caches triggered functions in memory without charging tenants

Goal: Exploit the serverless computing model to build a cost-effective, high-performance in-memory cache



Challenges: to build a memory cache with serverless functions

- A strawman proposal
  - Directly cache the objects in serverless functions' memory?
- No data availability guarantee
- Banned inbound network
- Limited per-function resources

Challenges: to build a memory cache with serverless functions

- A strawman proposal
  - Directly cache the objects in serverless functions' memory?

- No data availability guarantee

⚠️ Serverless functions could be reclaimed any time

⚠️ In-memory state is lost

- Banned inbound network

- Limited per-function resources

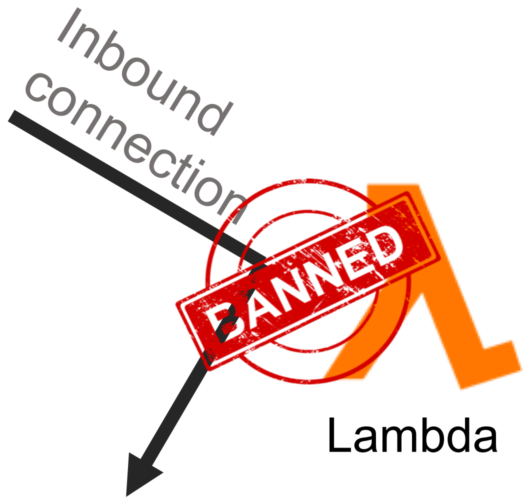
Challenges: to build a memory cache with serverless functions

- A strawman proposal
  - Directly cache the objects in serverless functions' memory?

⚠️ Serverless functions cannot run as a server

- No data availability guarantee

- Banned inbound network



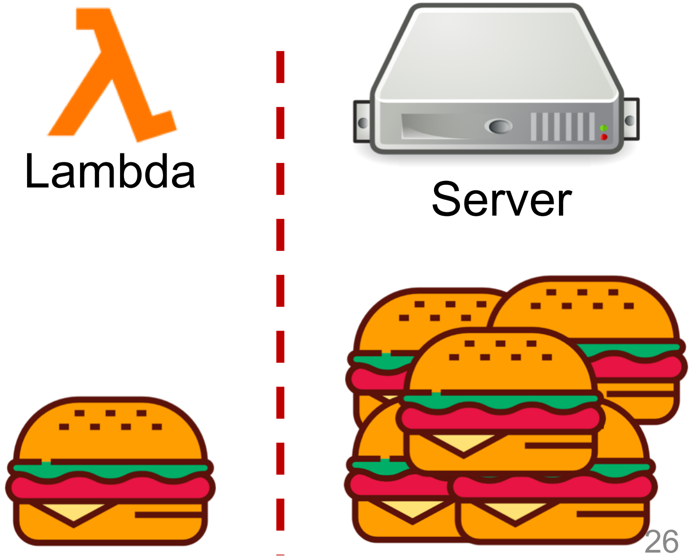
- Limited per-function resources

Challenges: to build a memory cache with serverless functions

- A strawman proposal
  - Directly cache the objects in serverless functions' memory?
- No data availability guarantee
- Banned inbound network
- Limited per-function resources

⚠️ Memory up to 3 GB

⚠️ CPU up to 2 cores



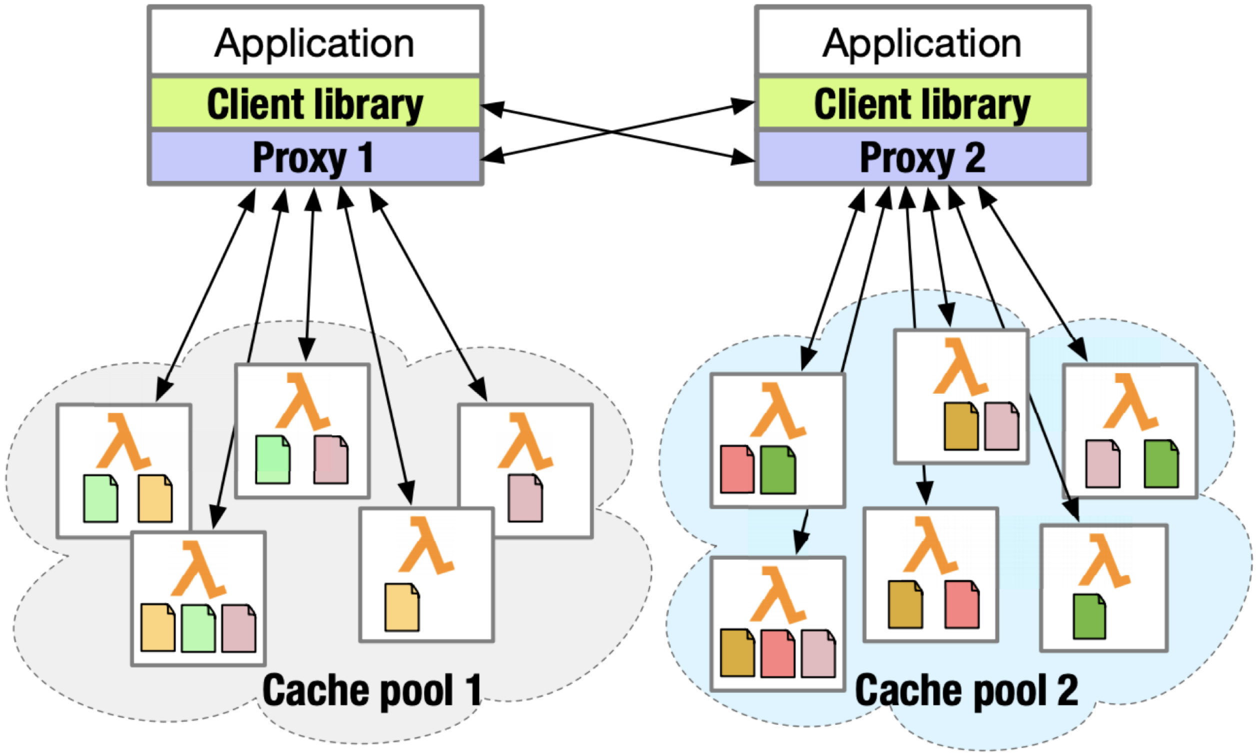
Our contribution: InfiniCache

- **The first in-memory cache system built atop serverless functions**
- InfiniCache achieves high data availability by leveraging erasure coding and delta-sync periodic data backup across functions
- InfiniCache achieves high performance by utilizing the aggregated network bandwidth of multiple functions in parallel
- InfiniCache achieves similar performance to AWS ElastiCache, while improving the cost-effectiveness by 31—96X

Outline

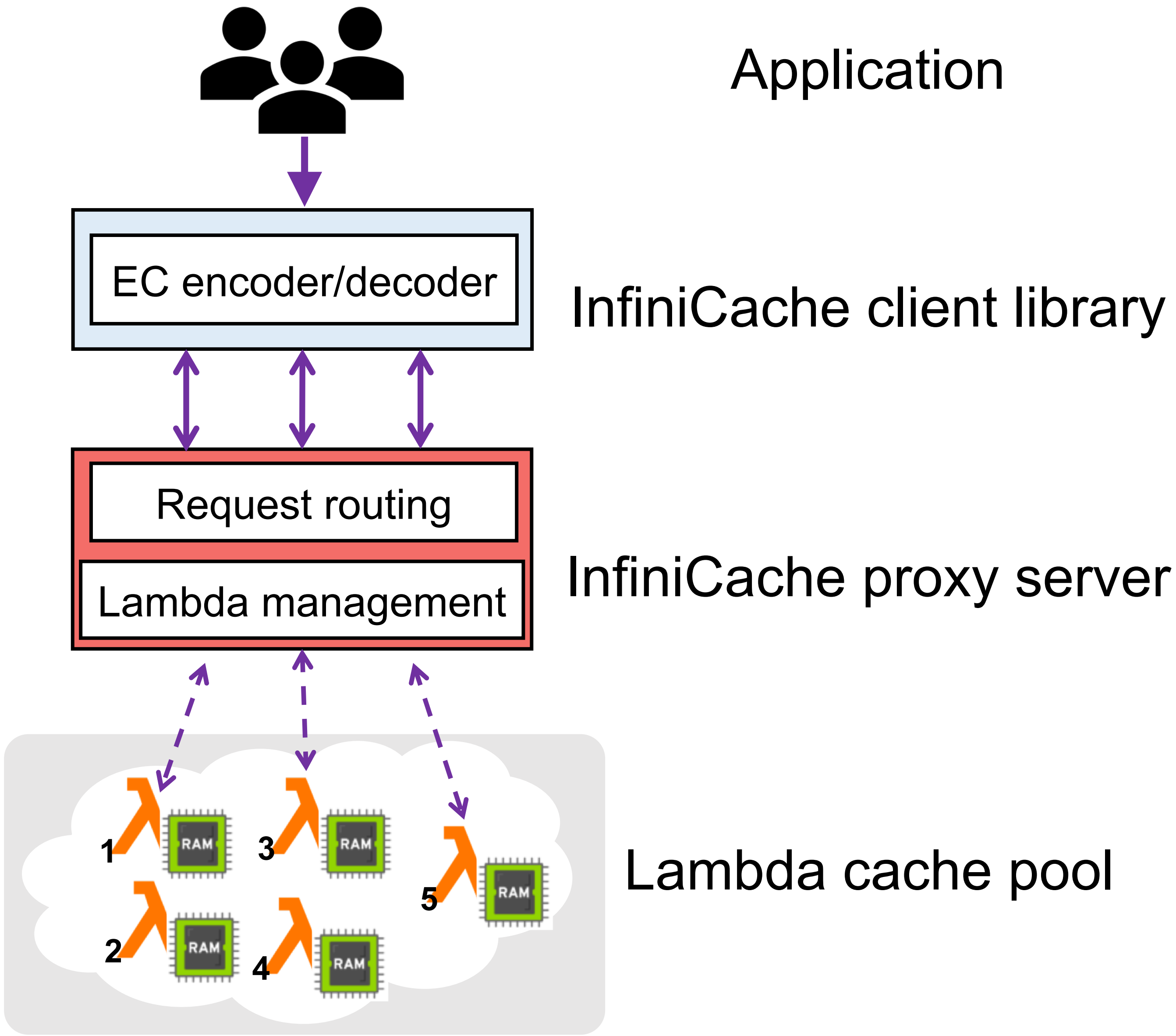
- InfiniCache Design
- Evaluation
- Conclusion

InfiniCache bird's eye view – Multi proxy

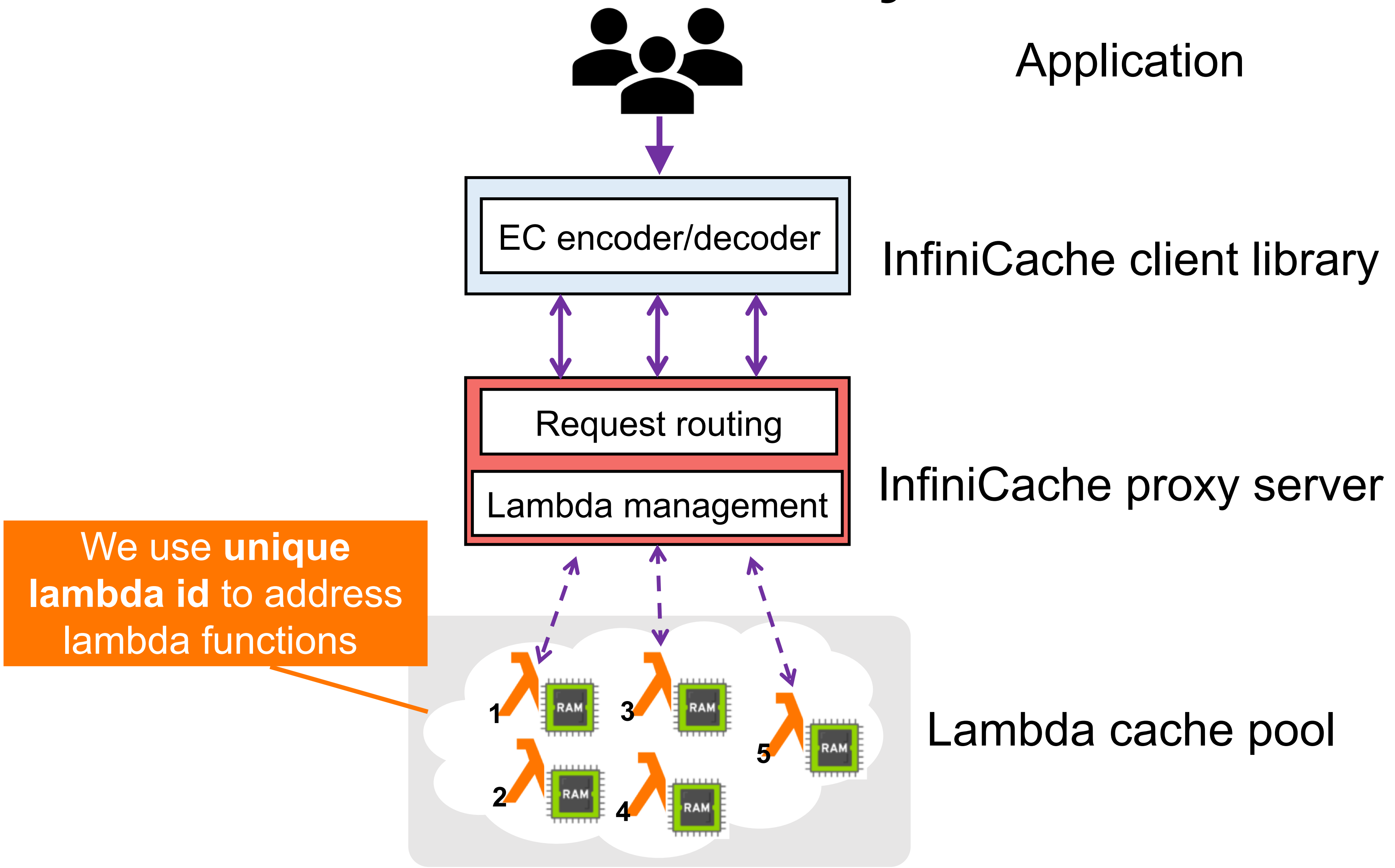


- Each application and each proxy will be fully connected
- No intersection between different lambda cache pools

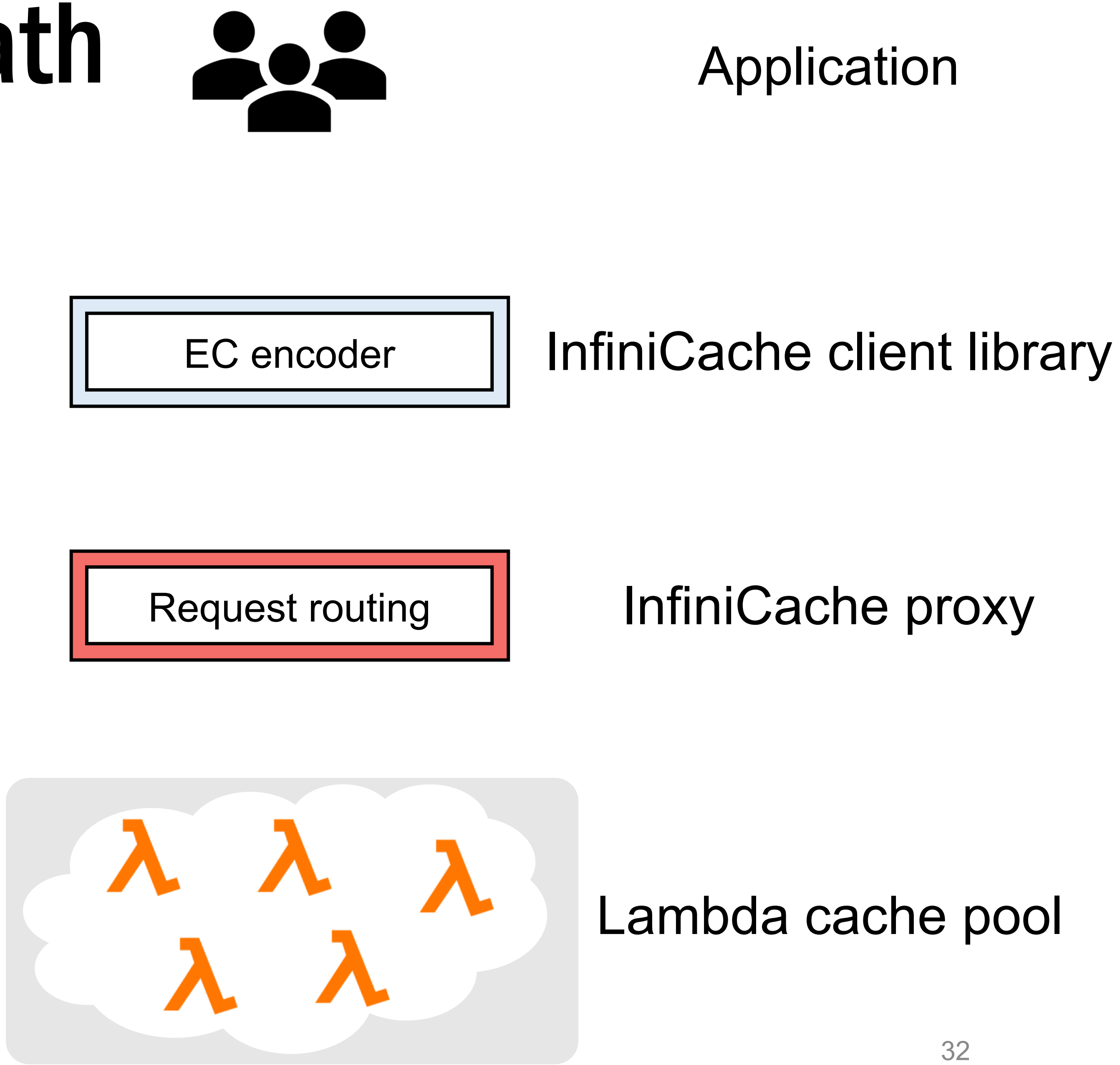
InfiniCache bird's eye view – zoom in (single proxy)



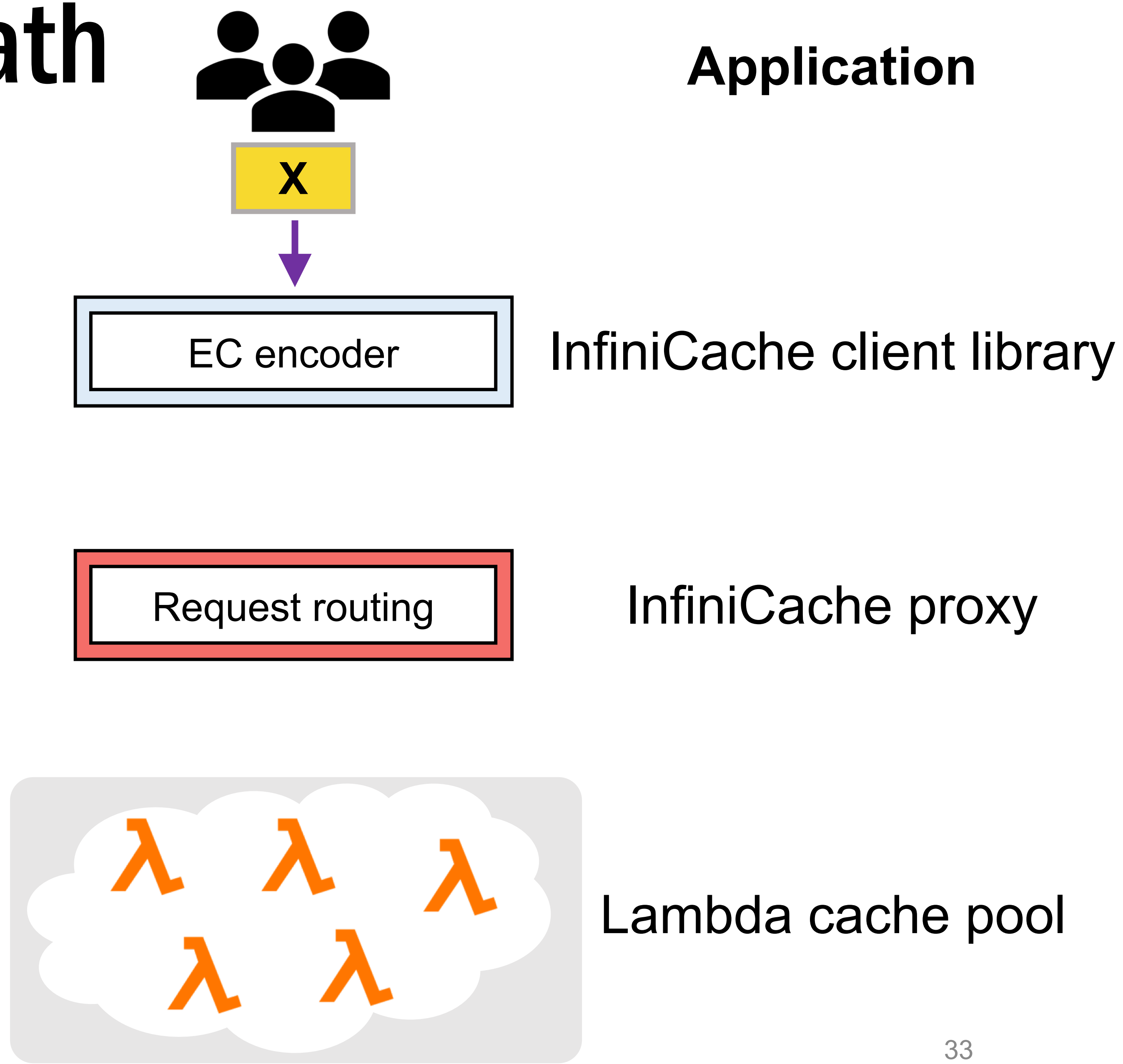
InfiniCache bird's eye view



InfiniCache: PUT path

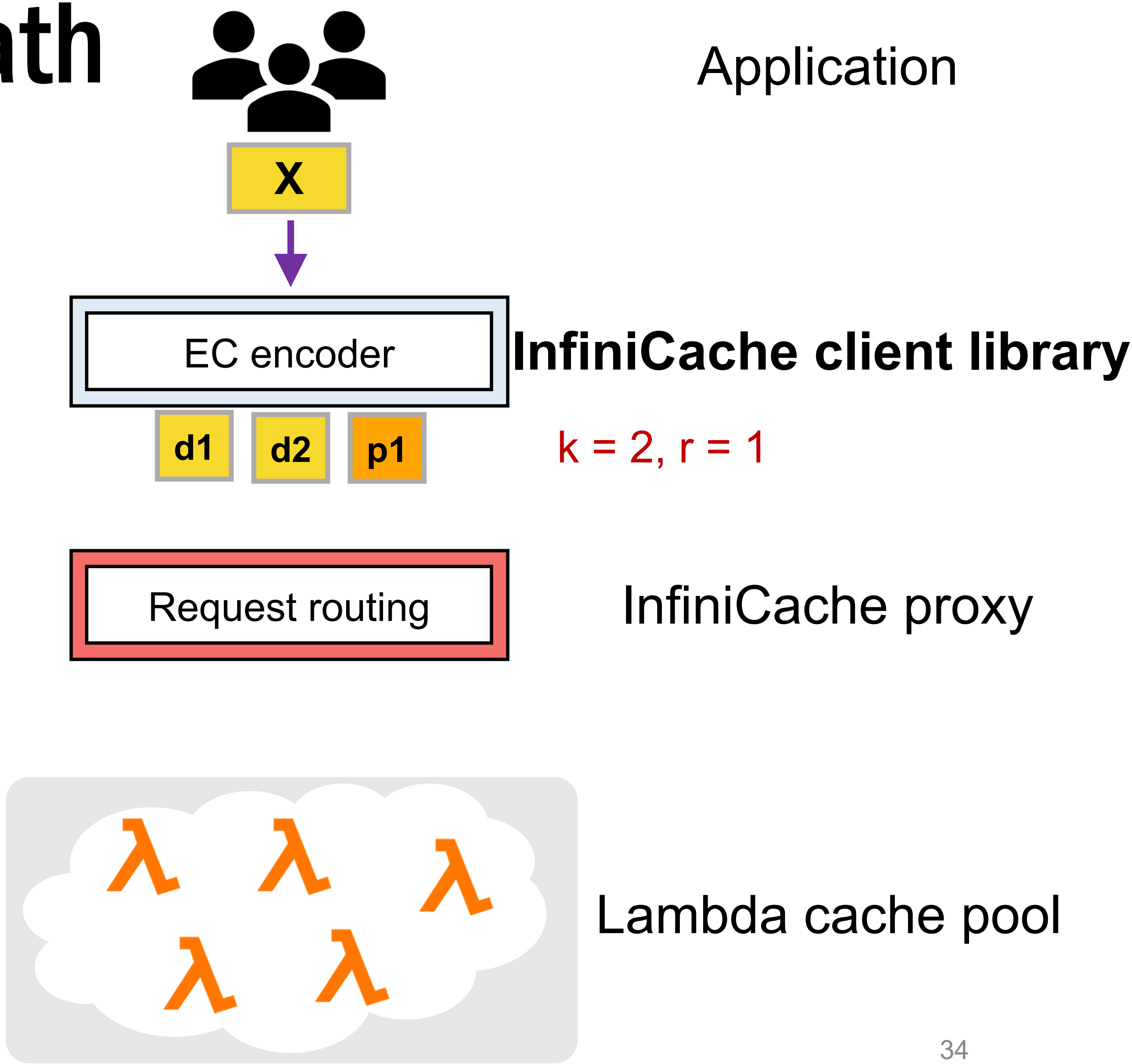


InfiniCache: PUT path



InfiniCache: PUT path

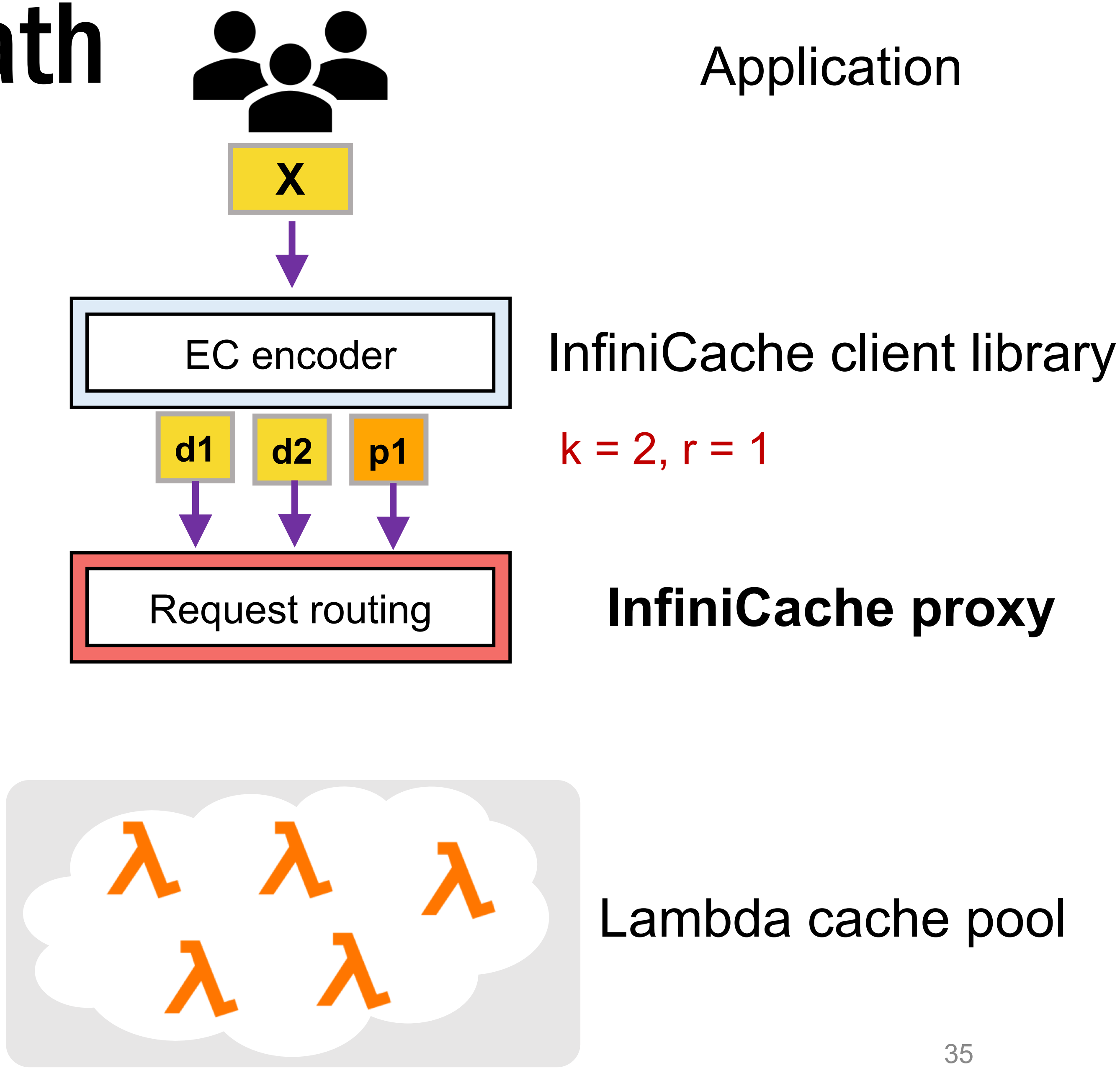
1. Object is split and encoded into k+r chunks



InfiniCache: PUT path

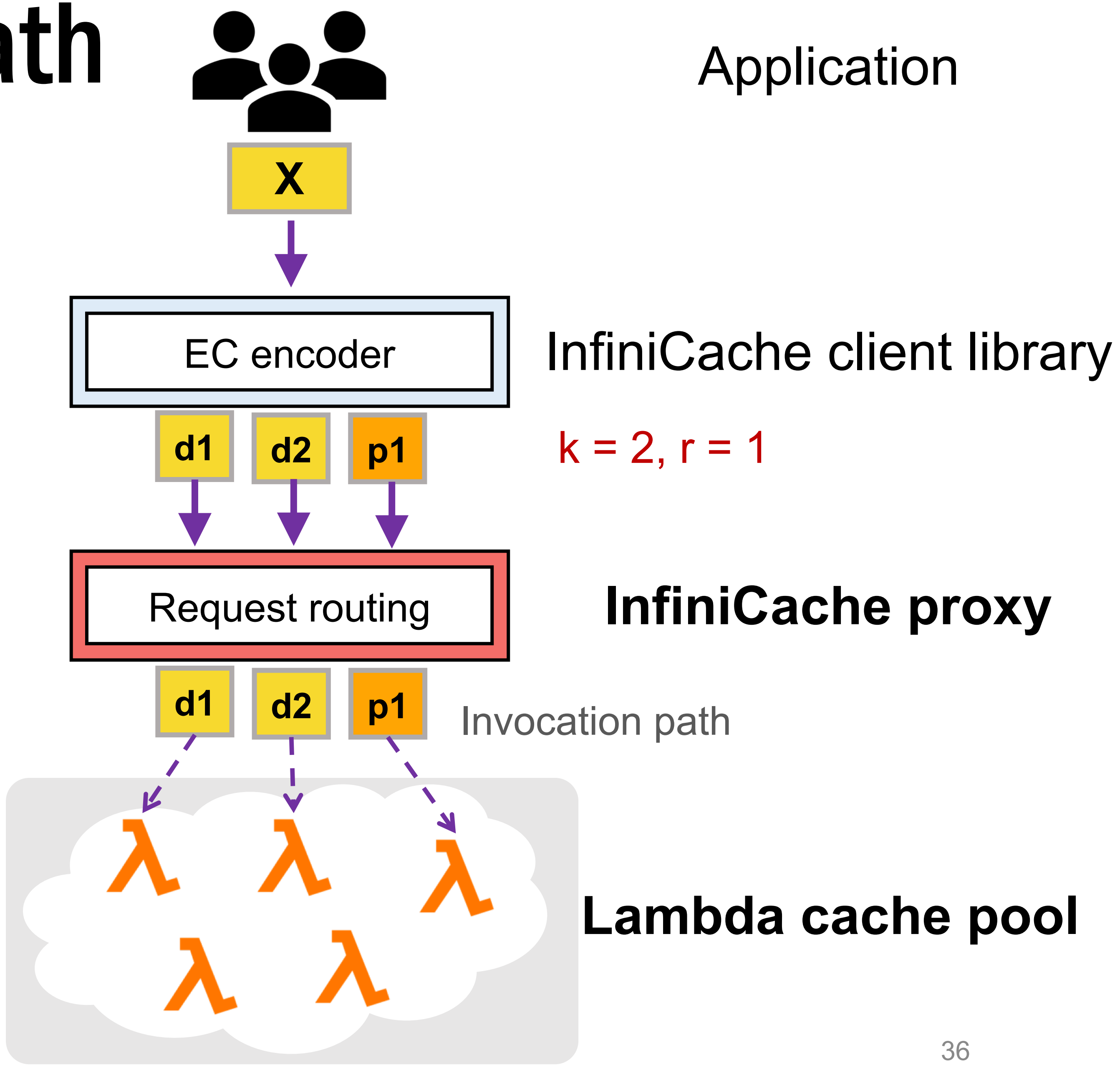
1. Object split and encode into k+r chunks

2. Object chunks are sent to the proxy in parallel



InfiniCache: PUT path

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



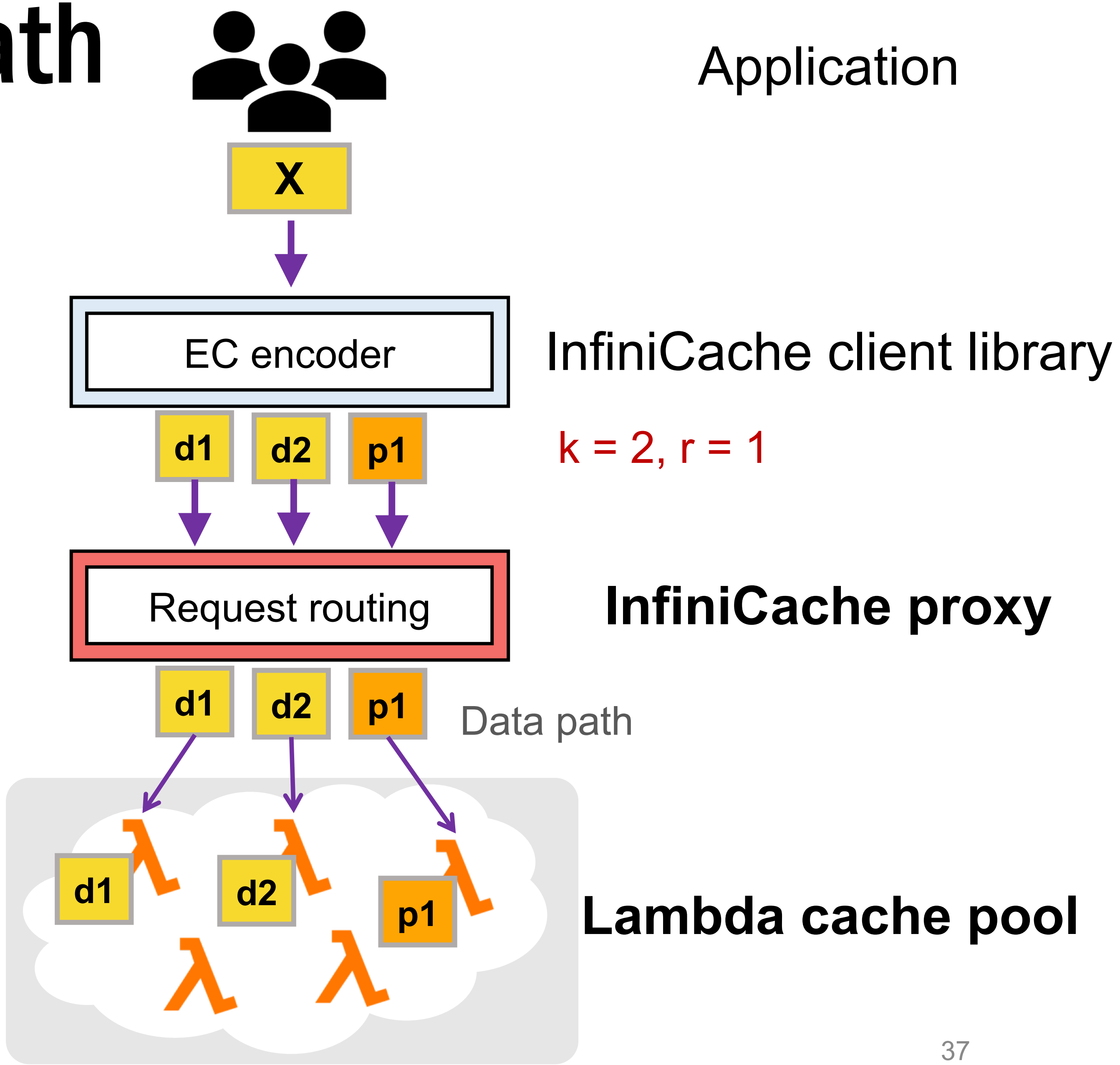
InfiniCache: PUT path

1. Object split and encode into k+r chunks

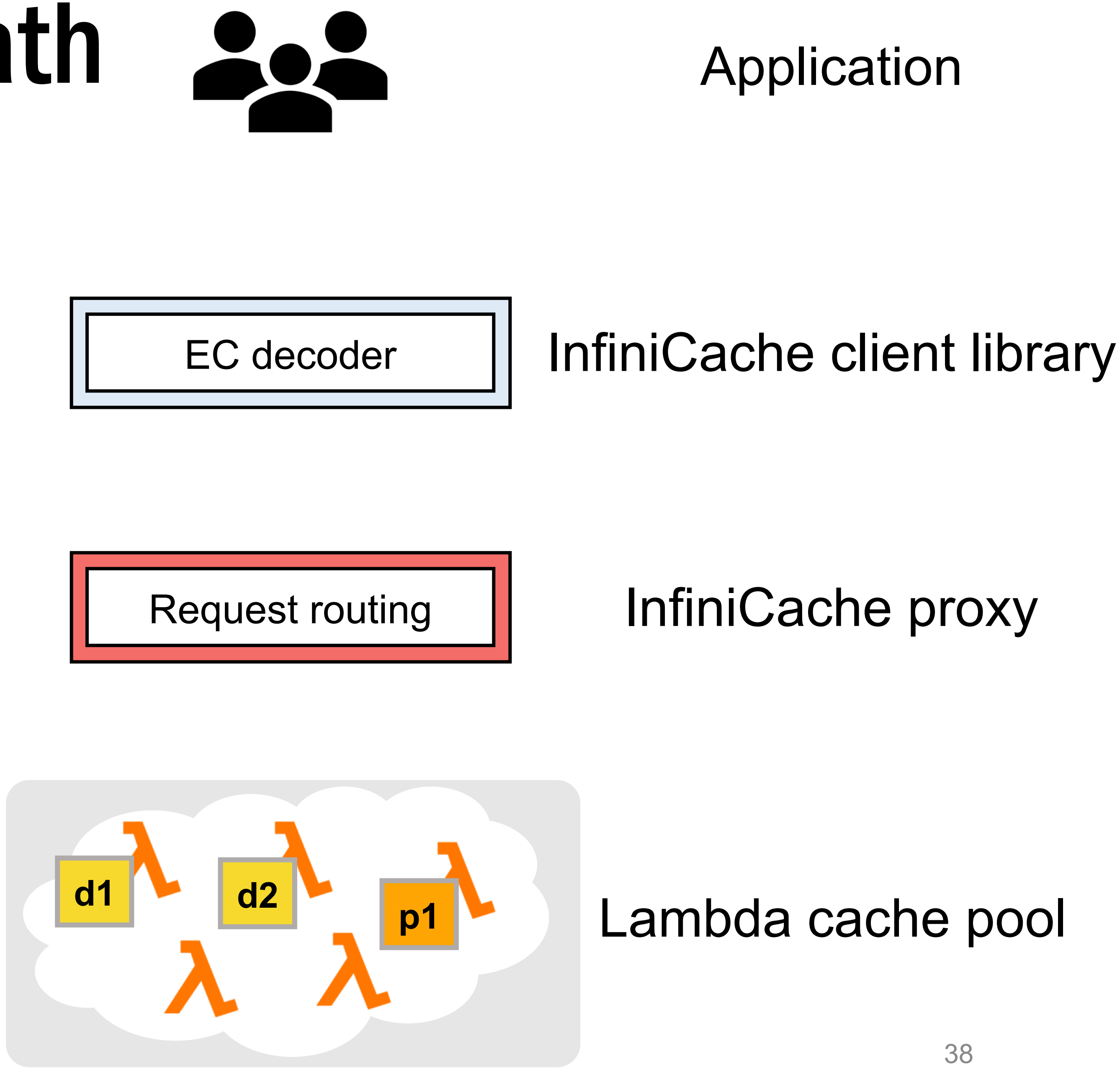
2. Object chunks are sent to the proxy in parallel

3. Proxy invoke Lambda cache nodes

4. Proxy streams object chunks to Lambda cache nodes

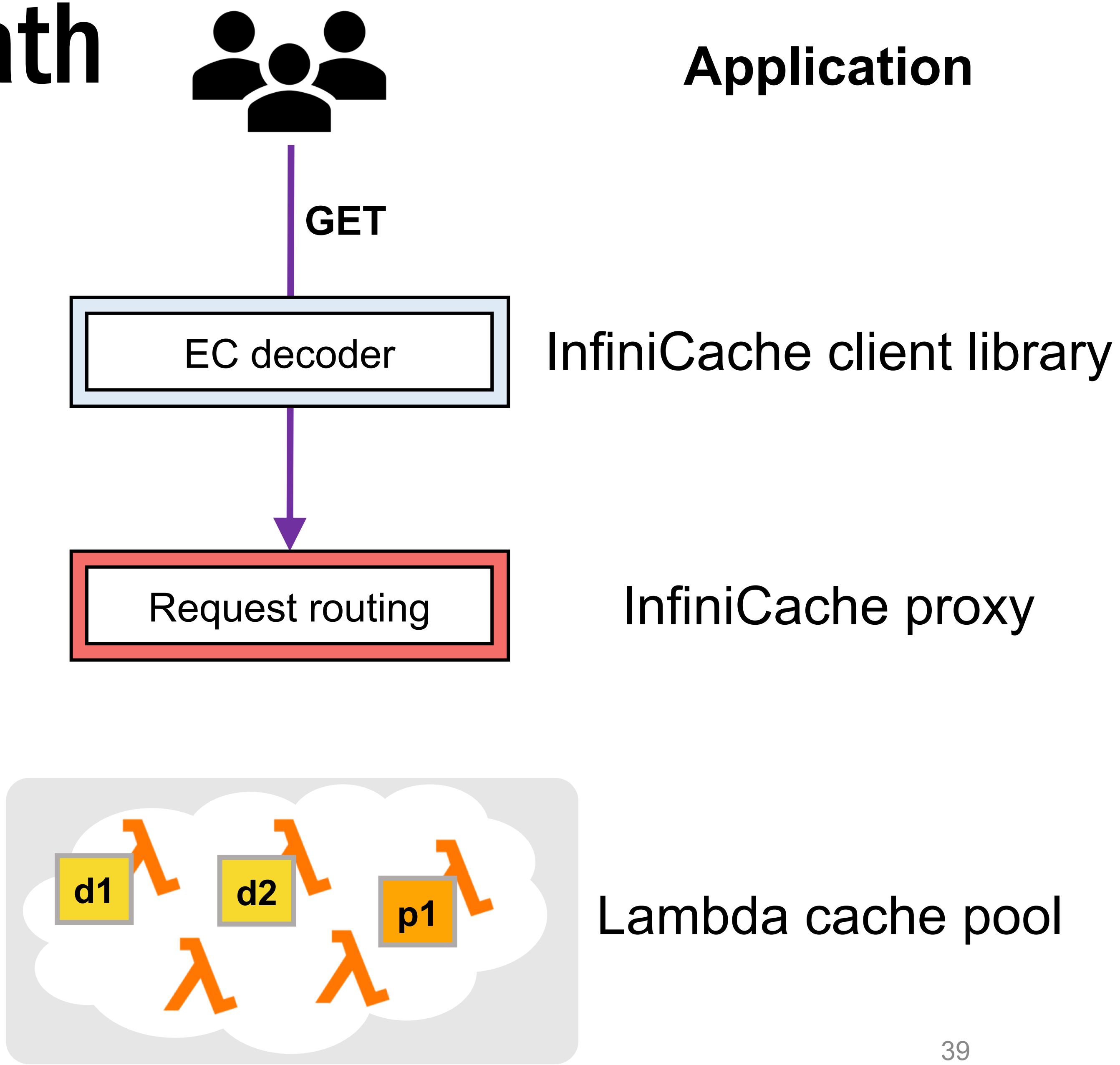


InfiniCache: GET path



InfiniCache: GET path

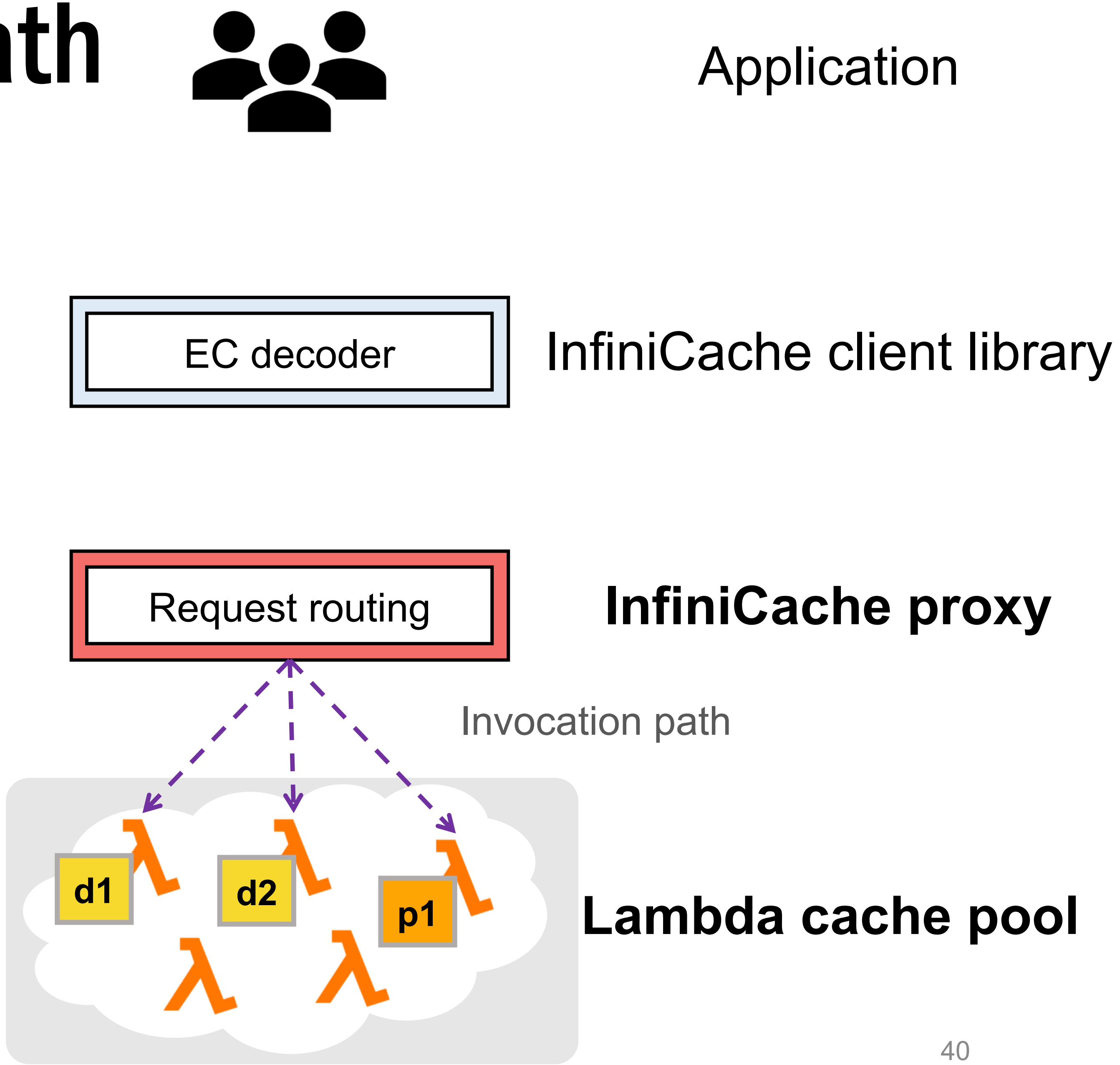
1. Client sends GET request



InfiniCache: GET path

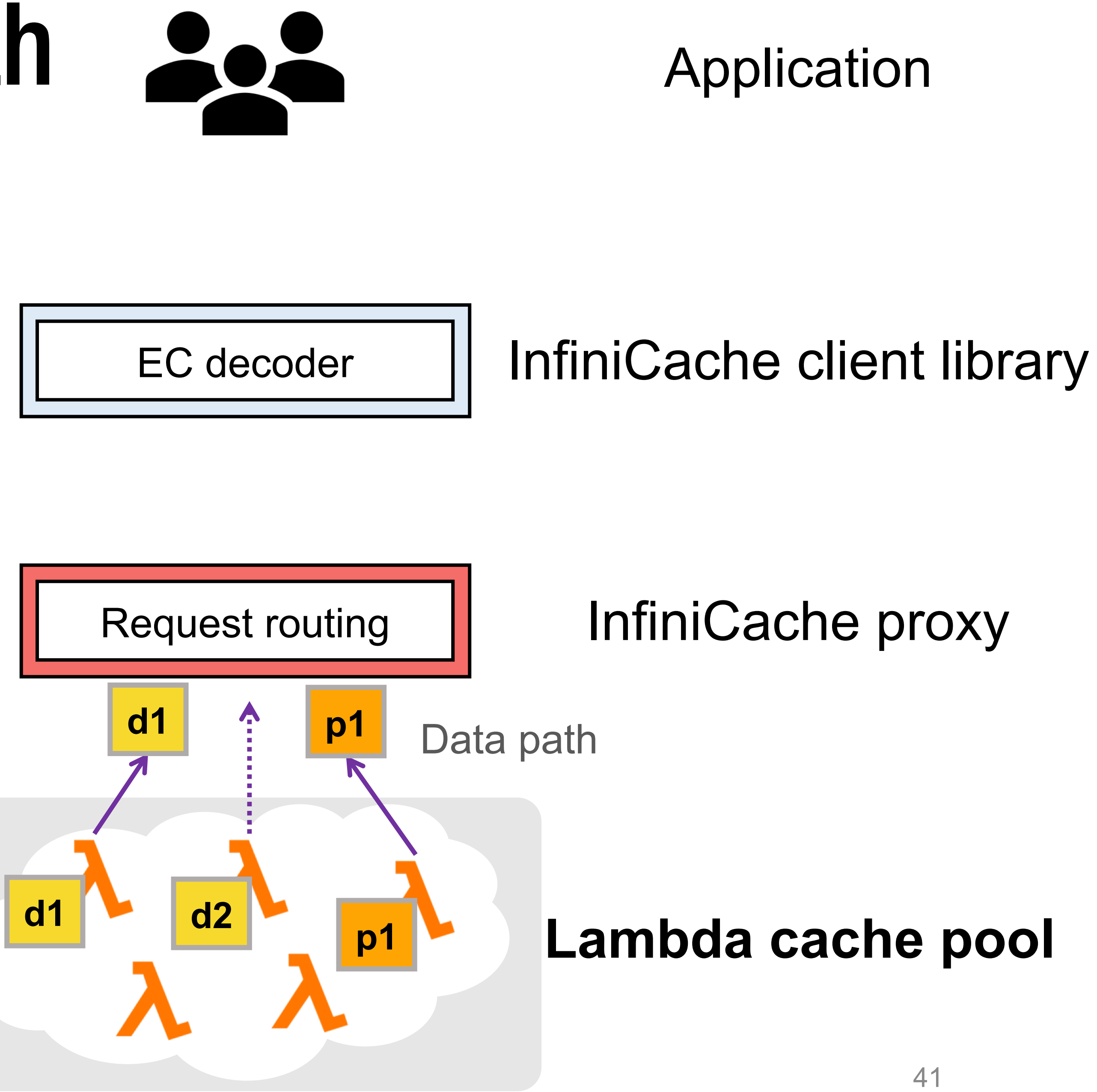
1. Client sends GET request

2. Proxy invokes associated Lambda cache nodes



InfiniCache: GET path

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

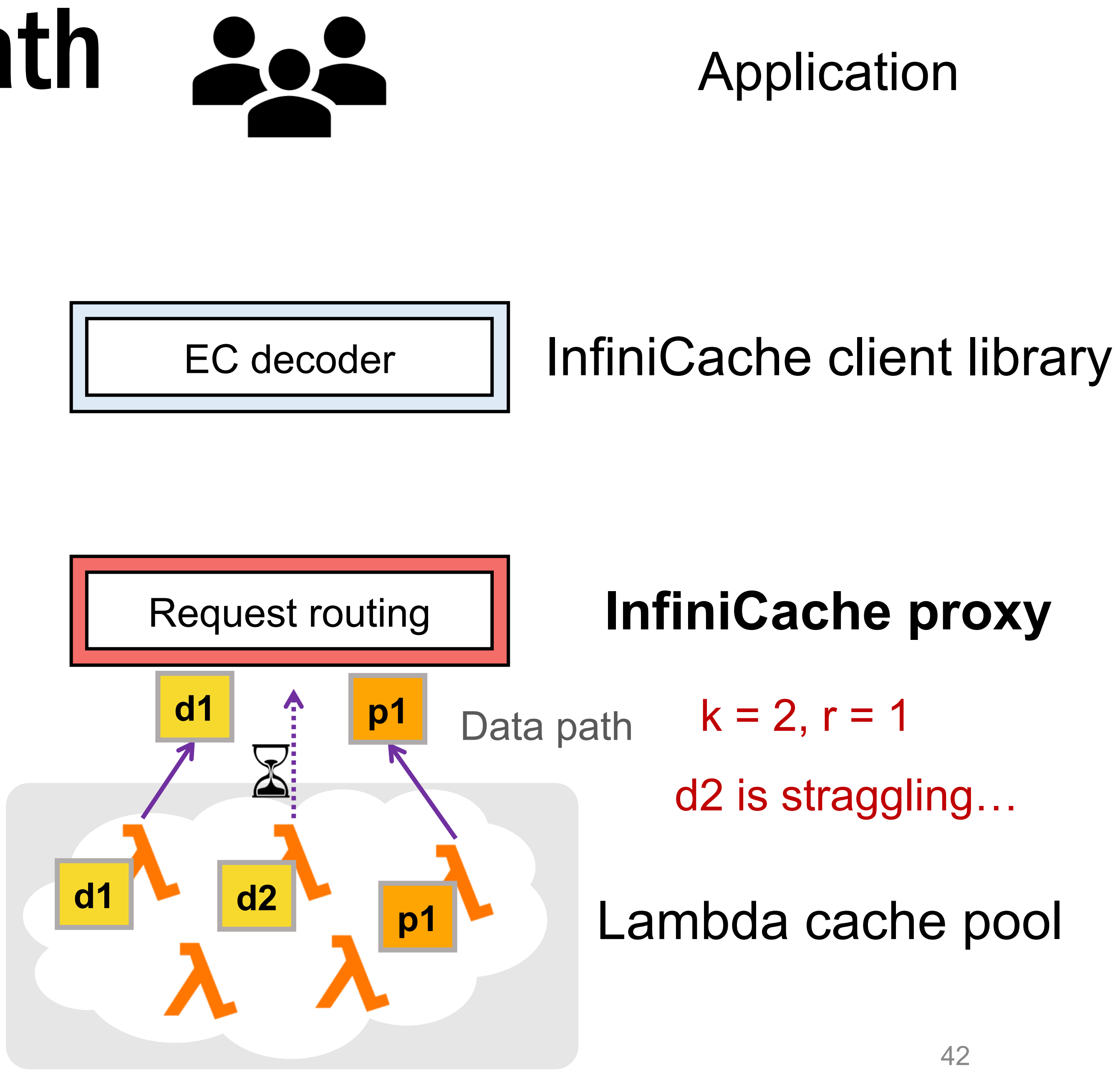


InfiniCache: GET path

1. Client sends GET request

2. Proxy invokes associated Lambda cache nodes

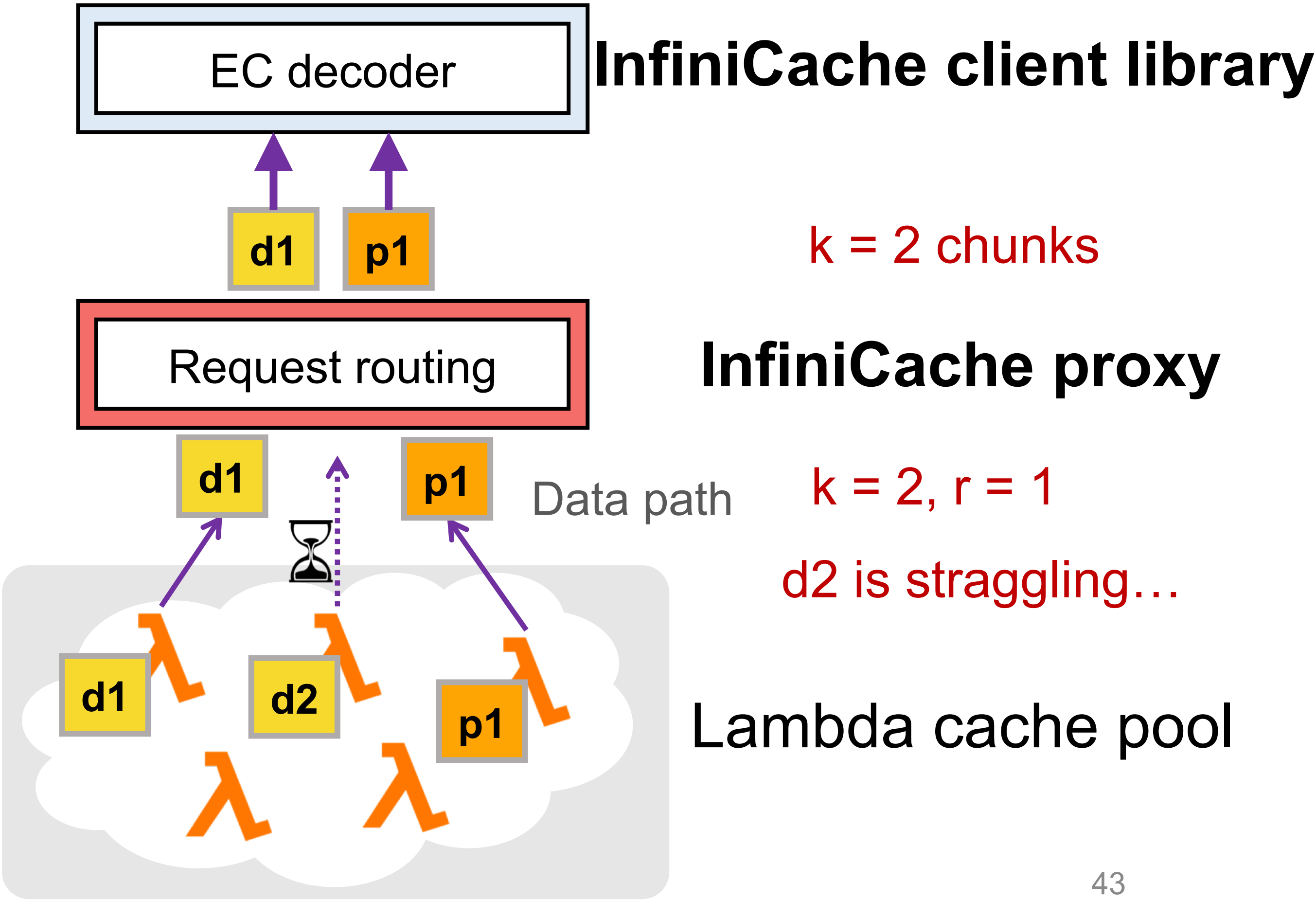
3. Lambda cache nodes transfer object chunks to proxy
    - **First-d optimization:** Proxy drops straggler Lambda



InfiniCache: GET path

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 chunks in parallel to client



InfiniCache: GET path

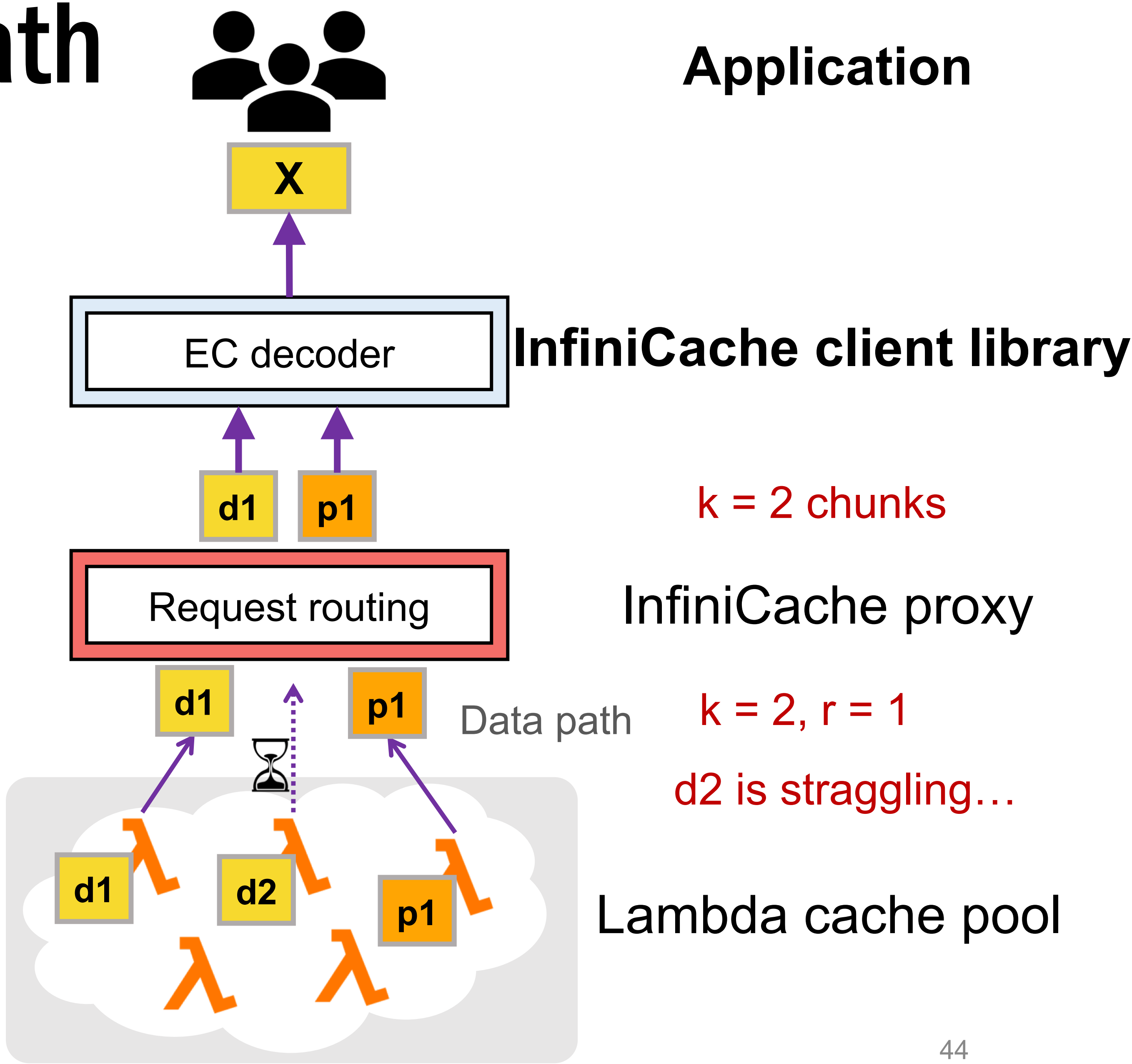
1. Client sends GET request

2. Proxy invokes associated Lambda cache nodes

3. Lambda cache nodes transfer object chunks to proxy

4. Proxy streams k chunks in parallel to client

5. Client library decodes k chunks



Maximizing data availability

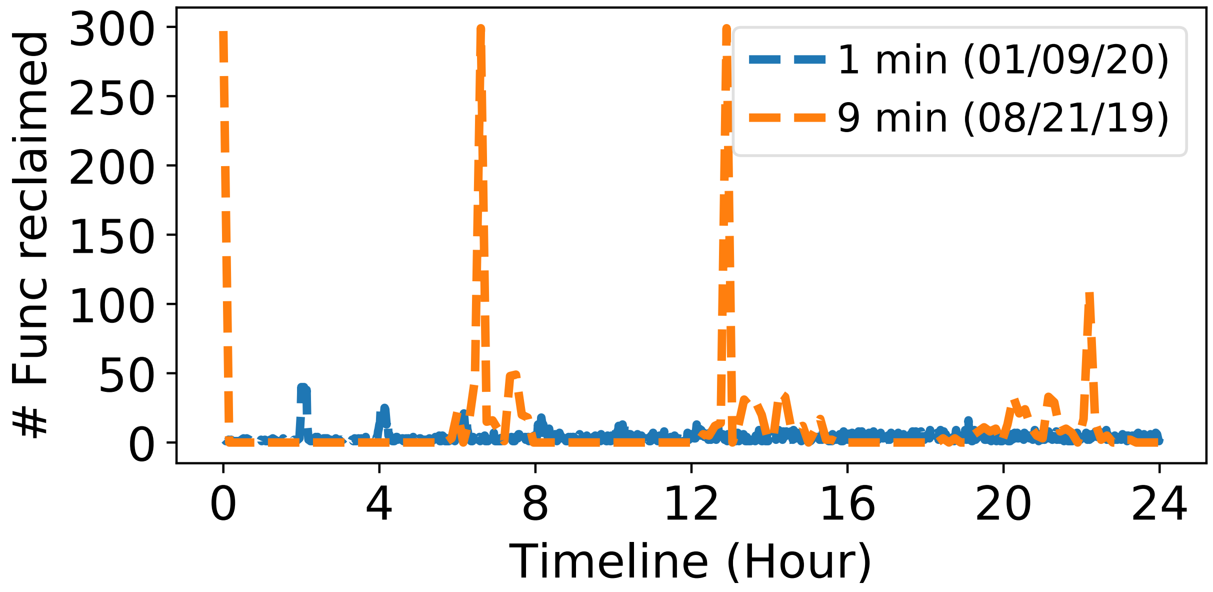
- Erasure-coding
- Periodic warm-up
- Periodic delta-sync backup

Maximizing data availability

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

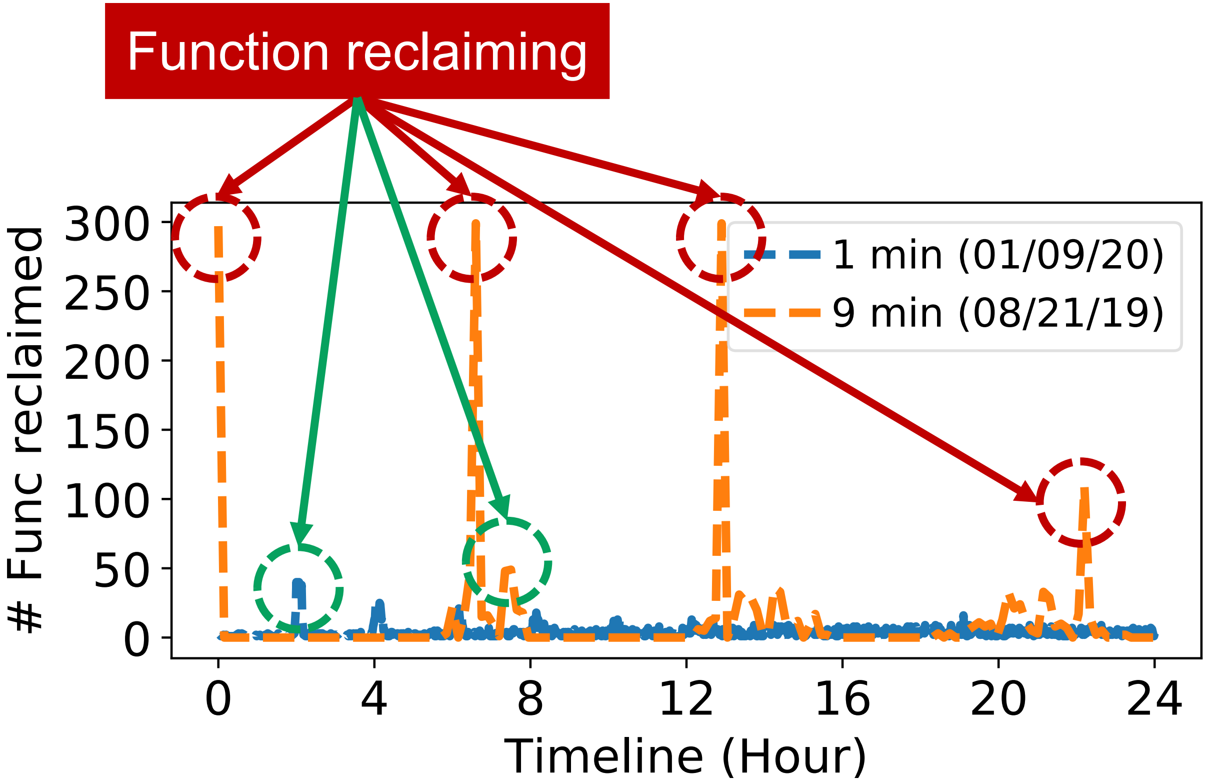
Maximizing data availability: Periodic warm-up

AWS Lambda reclaiming policy



Maximizing data availability: Periodic warm-up

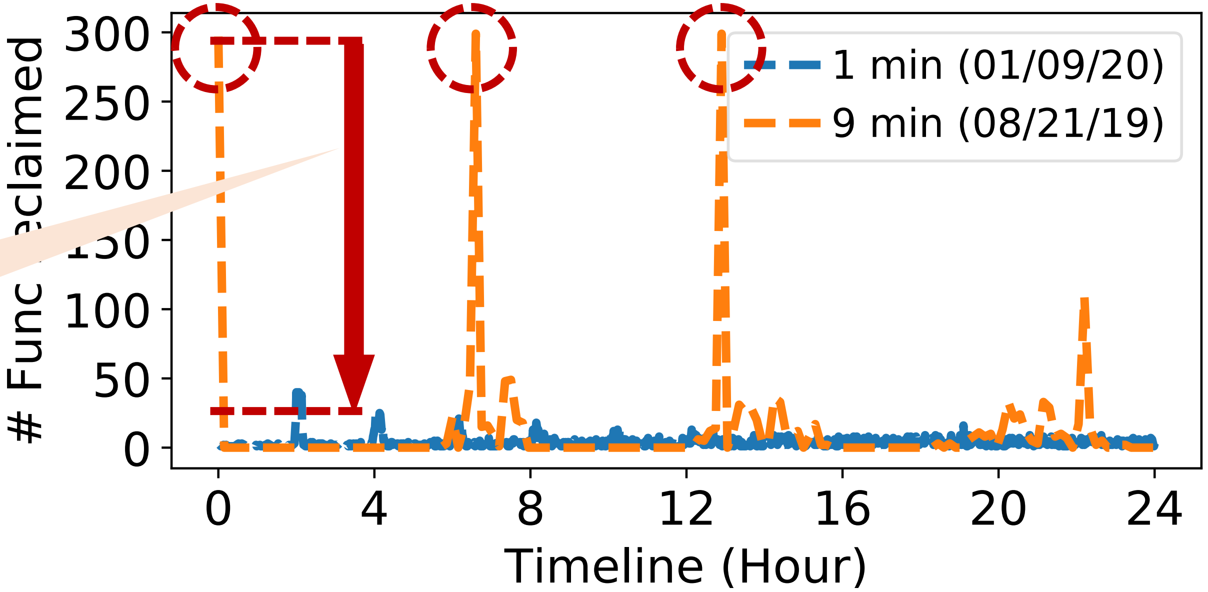
AWS Lambda reclaiming policy



Maximizing data availability: Periodic warm-up

AWS Lambda reclaiming policy

- Shorter triggering interval will lower the function reclaiming rate



1min interval significantly reduce function reclaiming rate

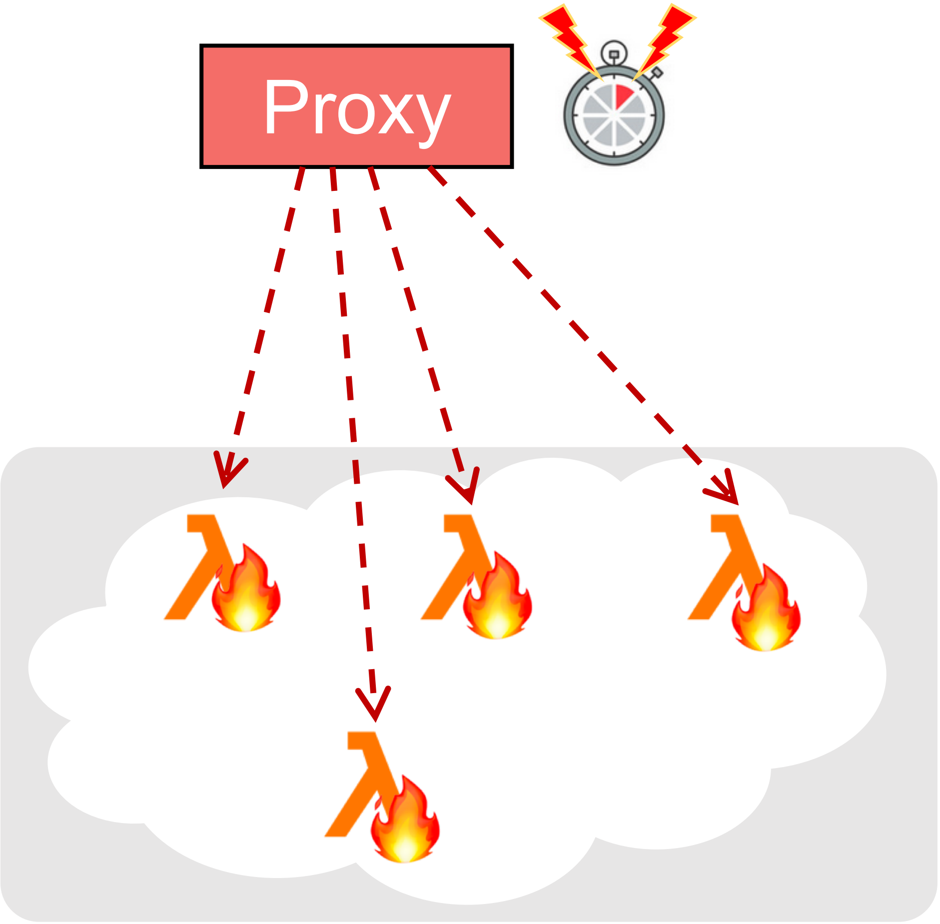
Maximizing data availability: Periodic warm-up

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

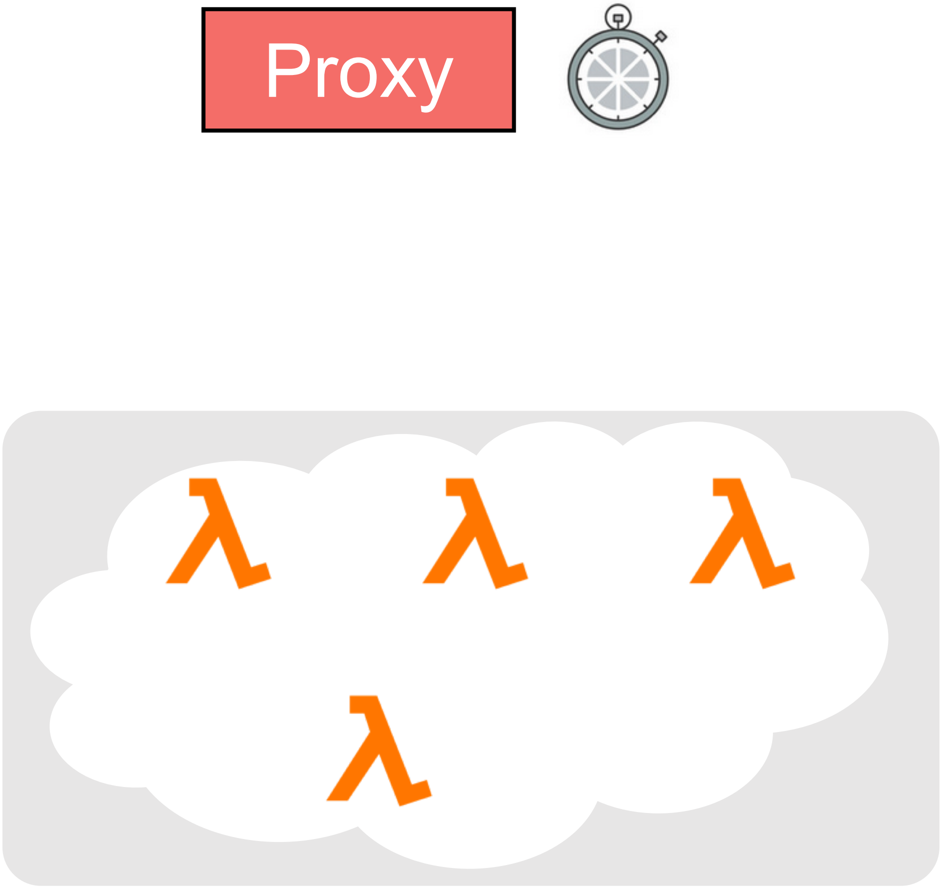
Maximizing data availability: Periodic warm-up

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

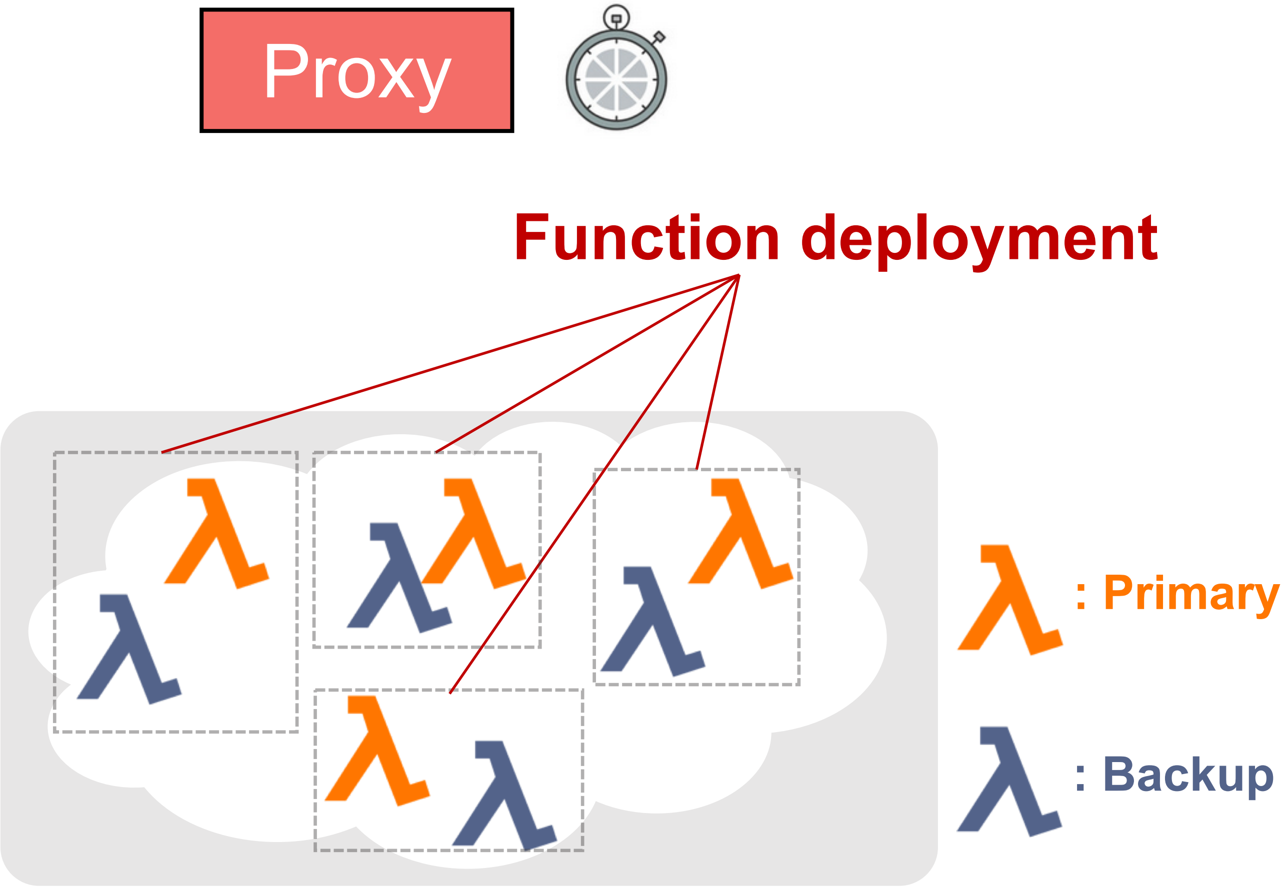
2. Proxy periodically invokes sleeping Lambda cache nodes to extend their lifespan



Maximizing data availability: Periodic backup

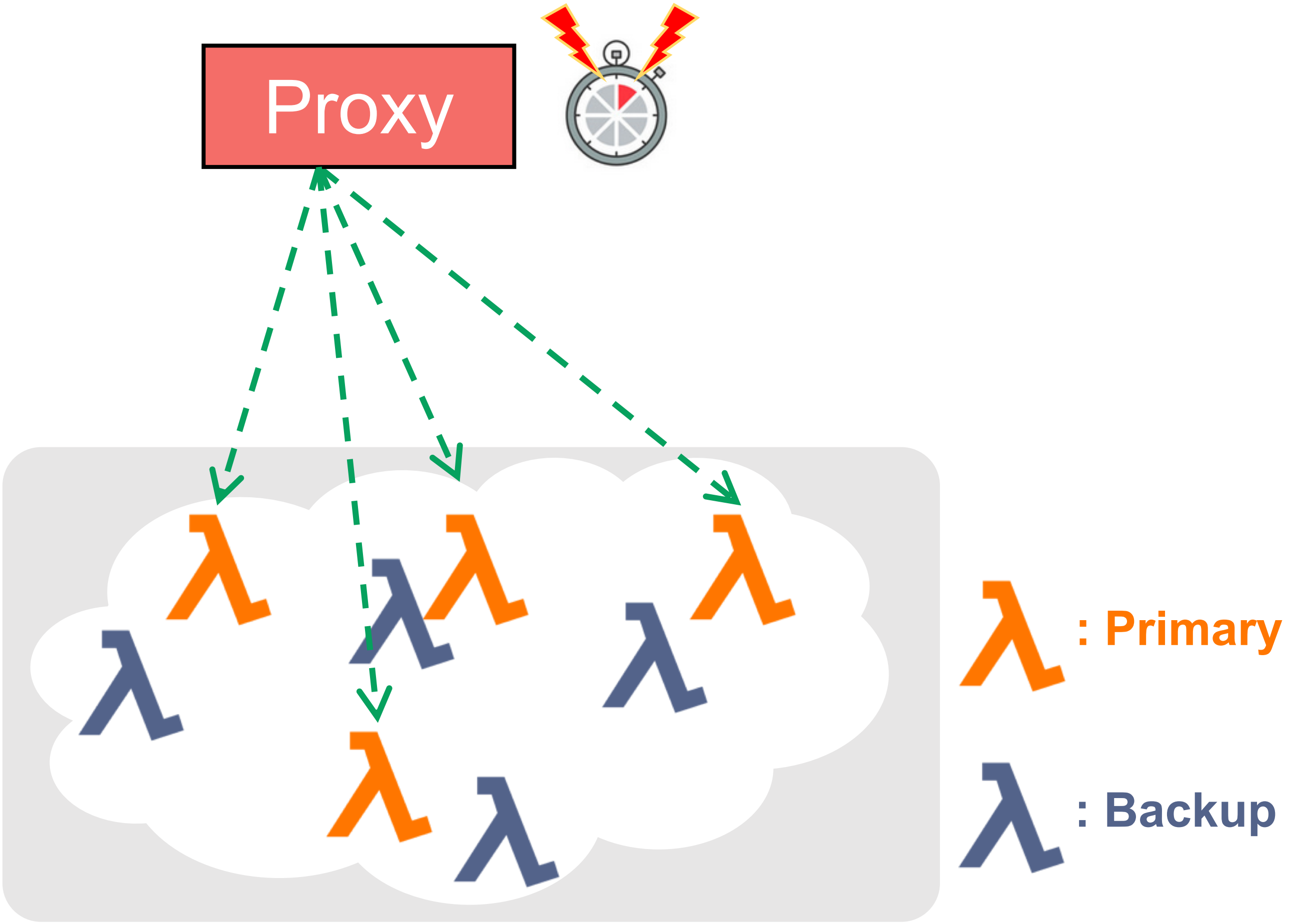


Maximizing data availability: Periodic backup



Maximizing data availability: Periodic backup

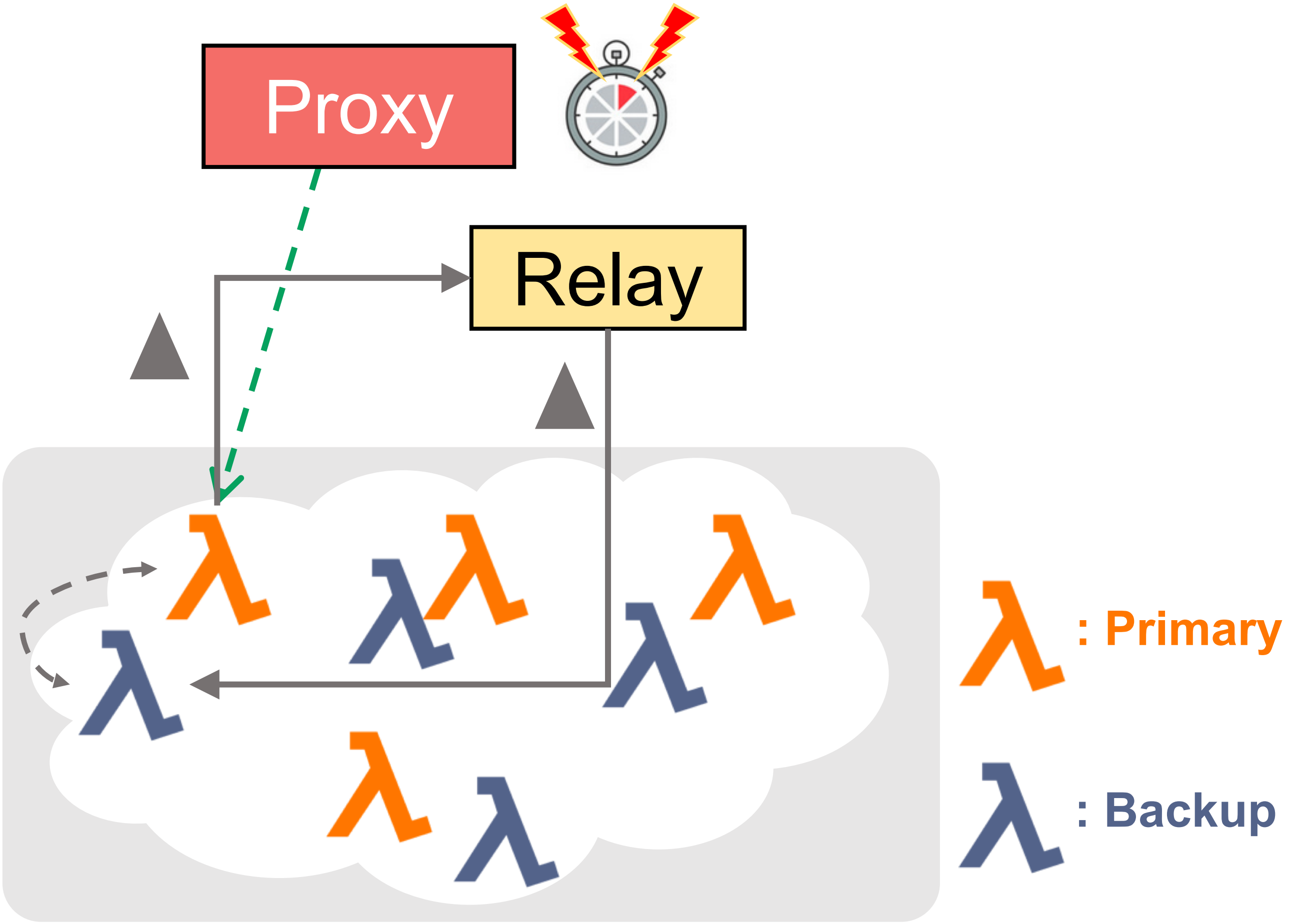
1. Proxy periodically sends out backup commands to Lambda cache nodes



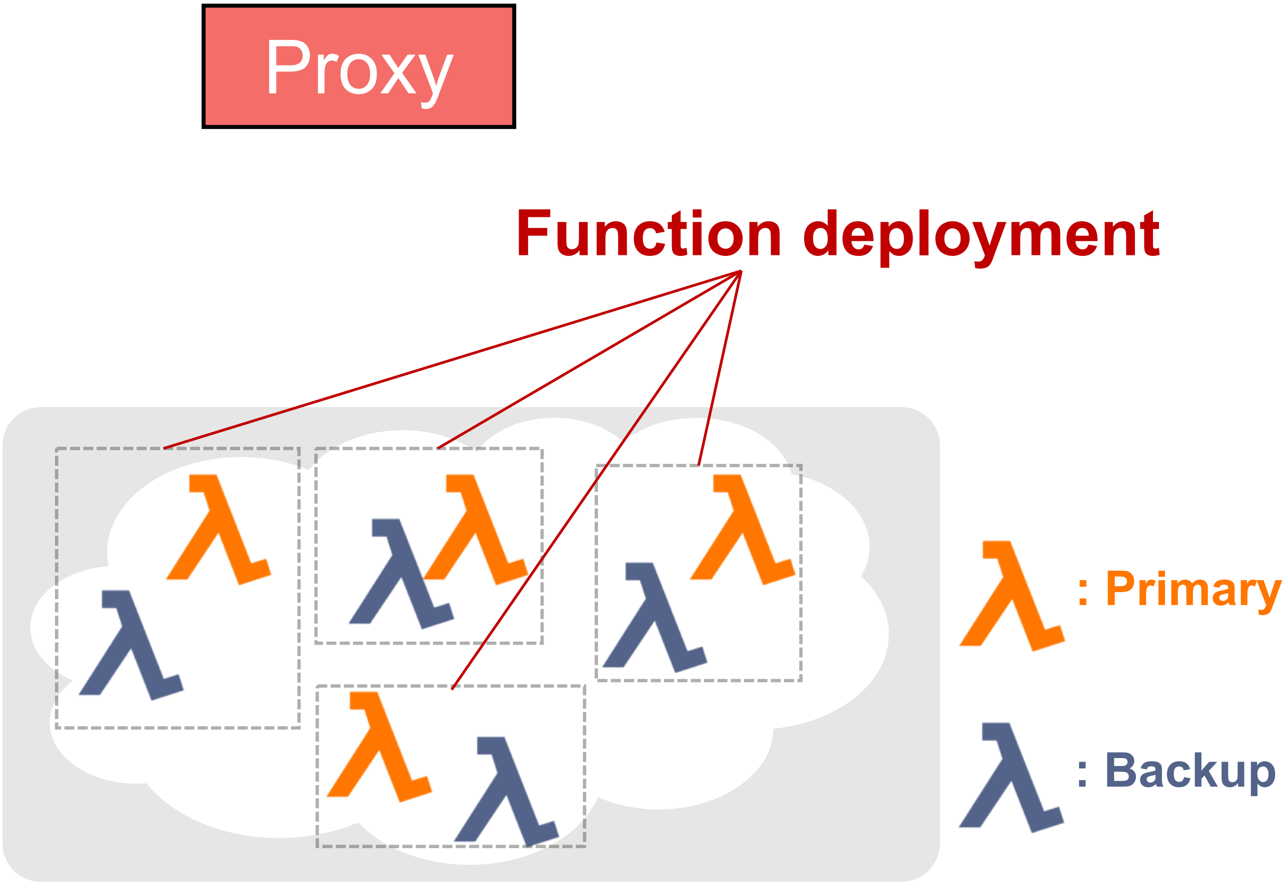
Maximizing data availability: Periodic backup

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

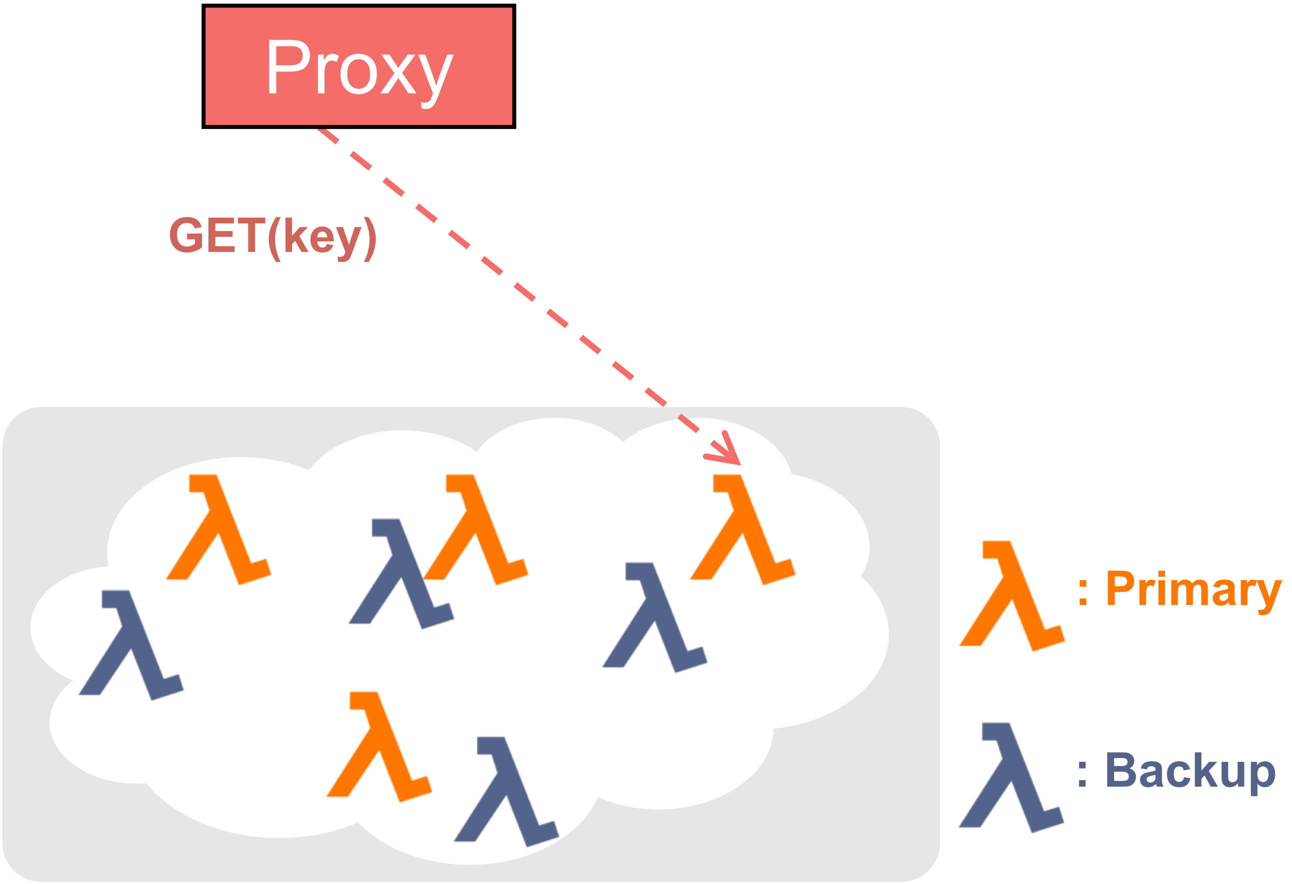


Seamless failover



Maximizing data availability: Seamless failover

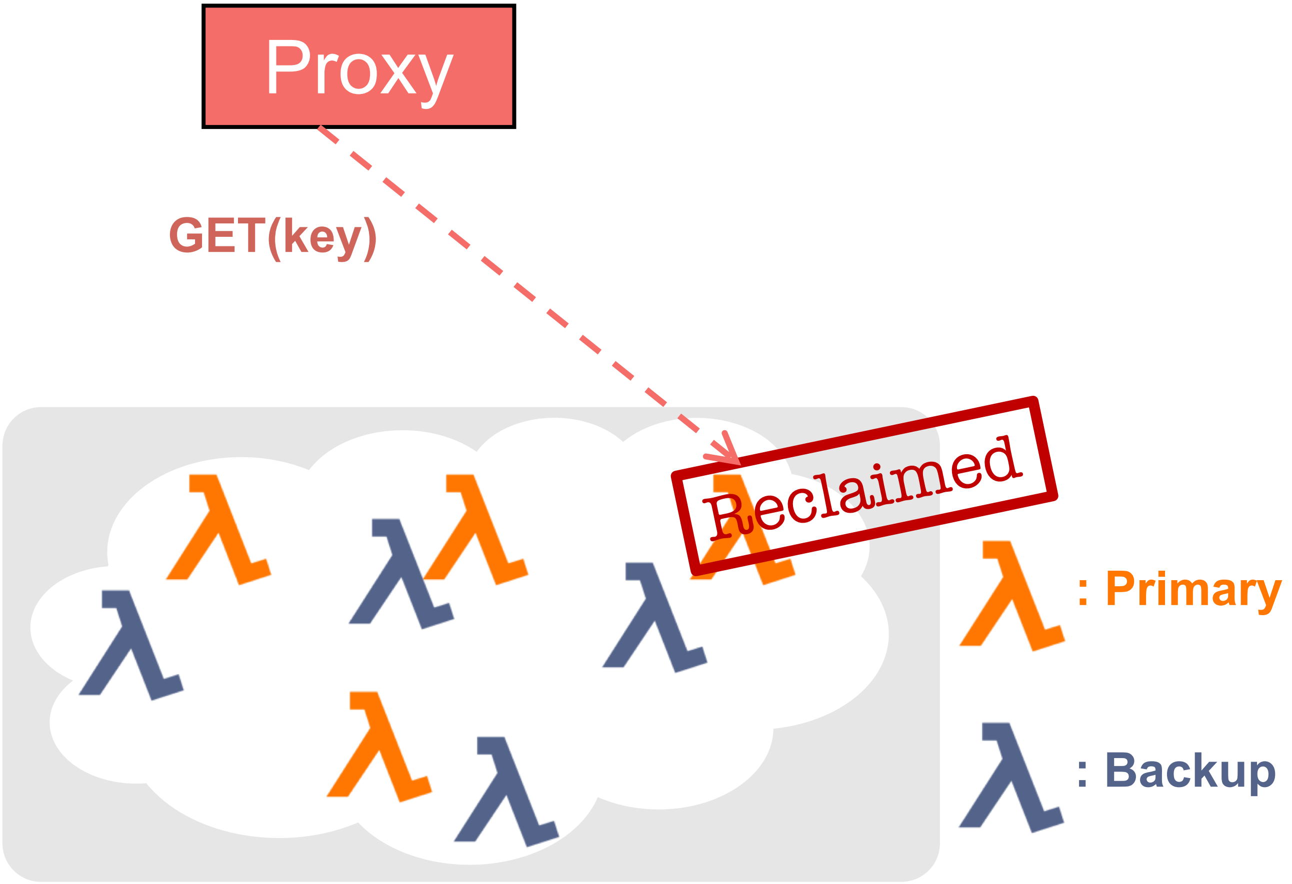
1. Proxy invokes a Lambda cache node with a GET request



Maximizing data availability: Seamless failover

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

2. Primary Lambda gets reclaimed

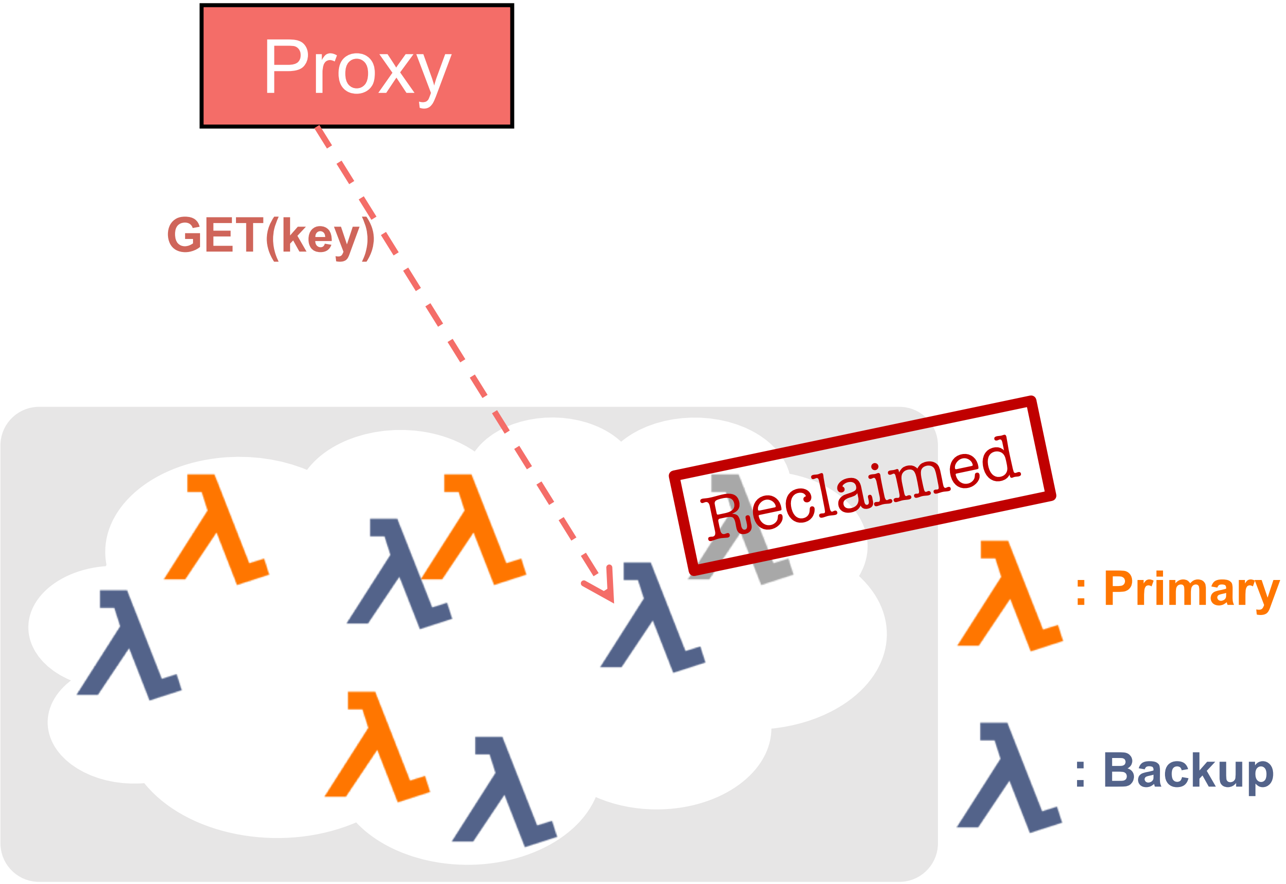


Maximizing data availability: Seamless failover

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

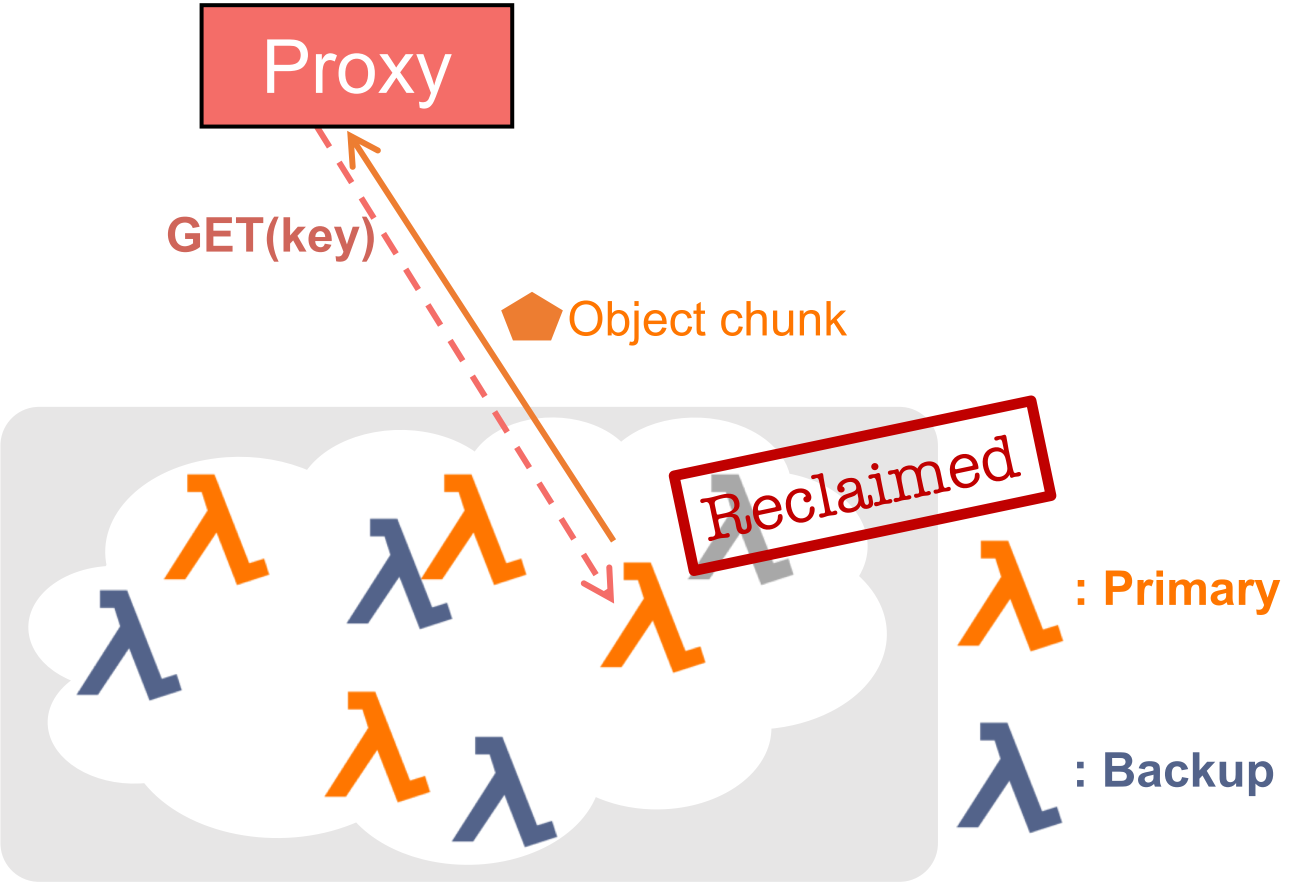
2. Primary Lambda gets reclaimed

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



Outline

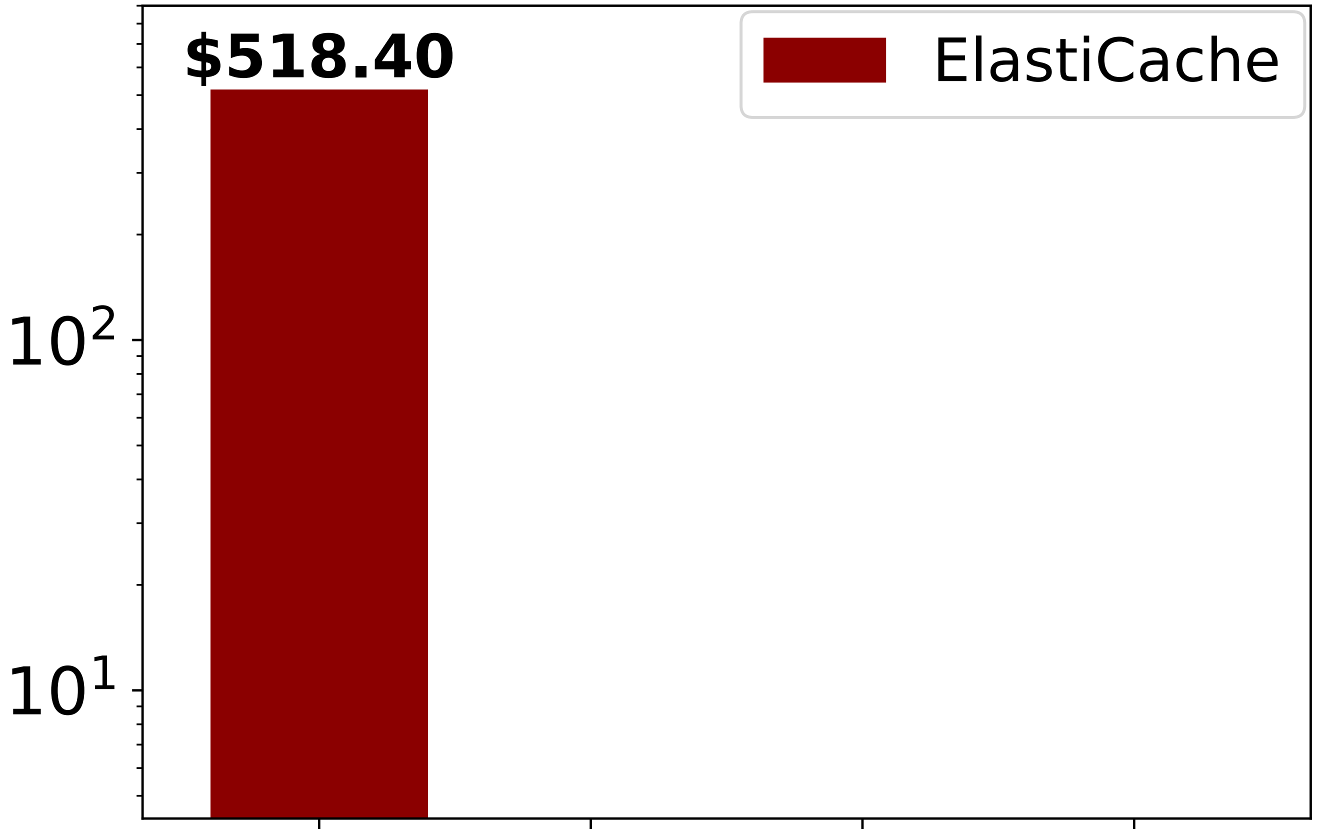
- InfiniCache Design
- Evaluation
- Conclusion

Experimental setup

- InfiniCache
  - 400 1.5GB Lambda cache nodes
  - Client running on one `c5n.4xlarge` EC2 VM
  - Warm-up interval: 1 minute; backup interval: 5 minutes
  - Under one AWS VPC

- Production workloads
  - The first 50 hours of the Dallas datacenter traces from IBM Docker registry workloads
  - All objects: including small and large objects
  - Large object only: objects > 10MB

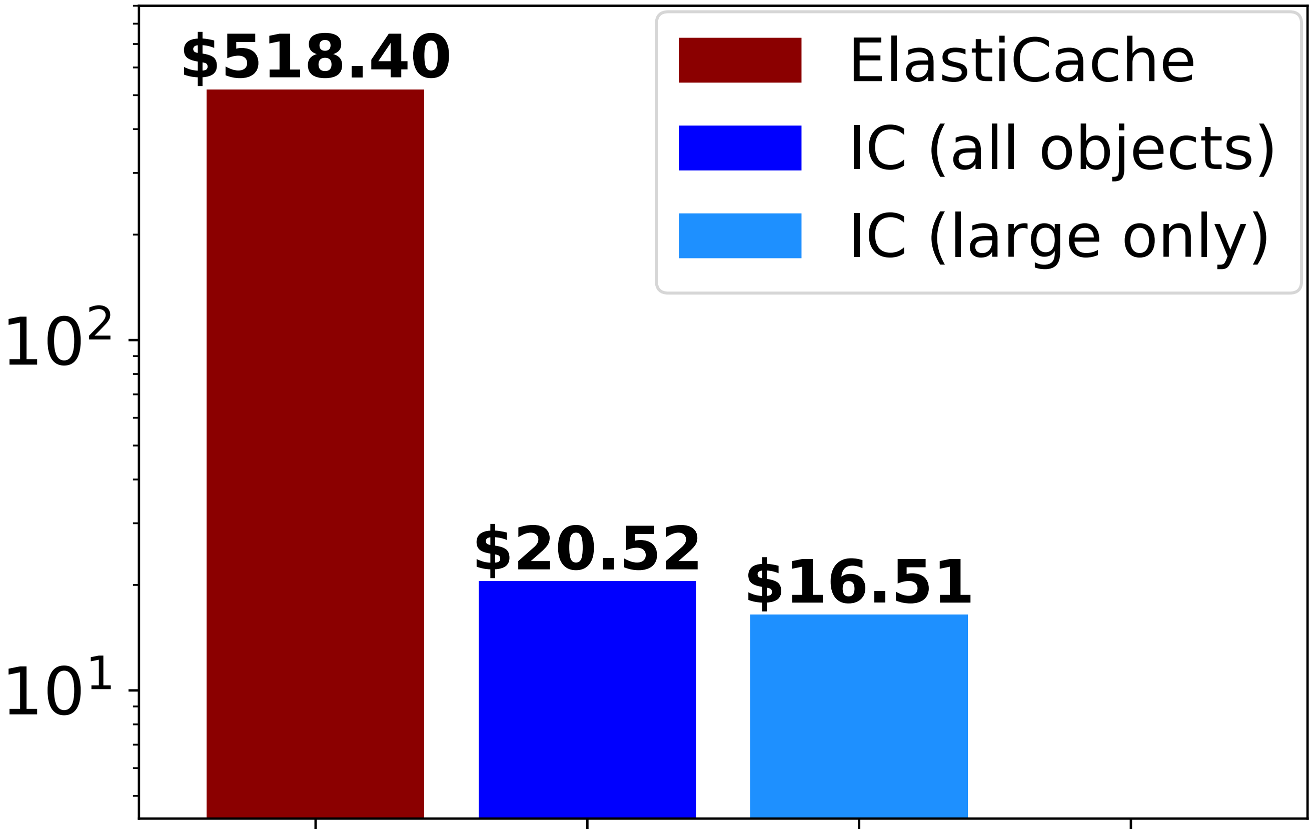
Cost effectiveness of InfiniCache



AWS ElastiCache

- One `cache.r5.24xlarge` with 600GB memory
- $10.368 per hour

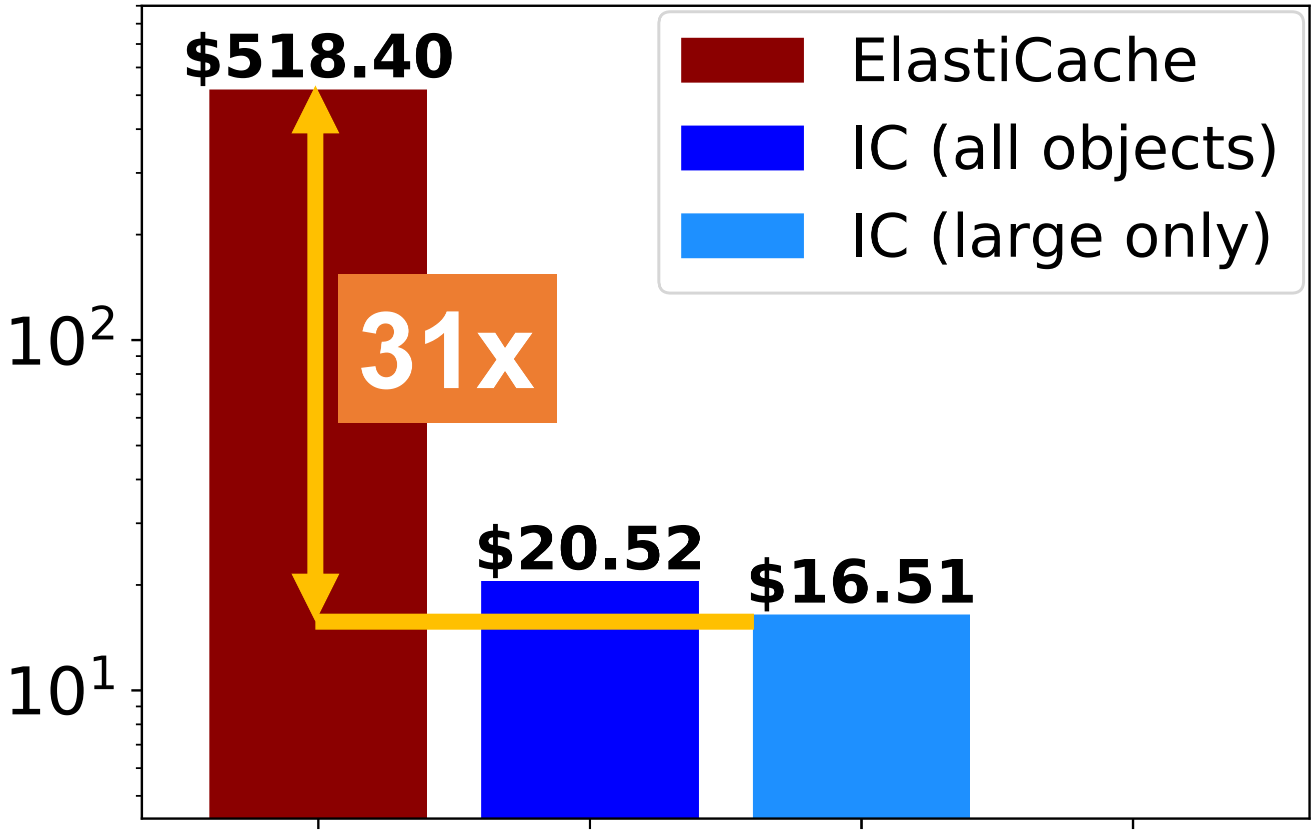
Cost effectiveness of InfiniCache



Workload setup

- All objects
- Large object only
  - Object larger than 10MB

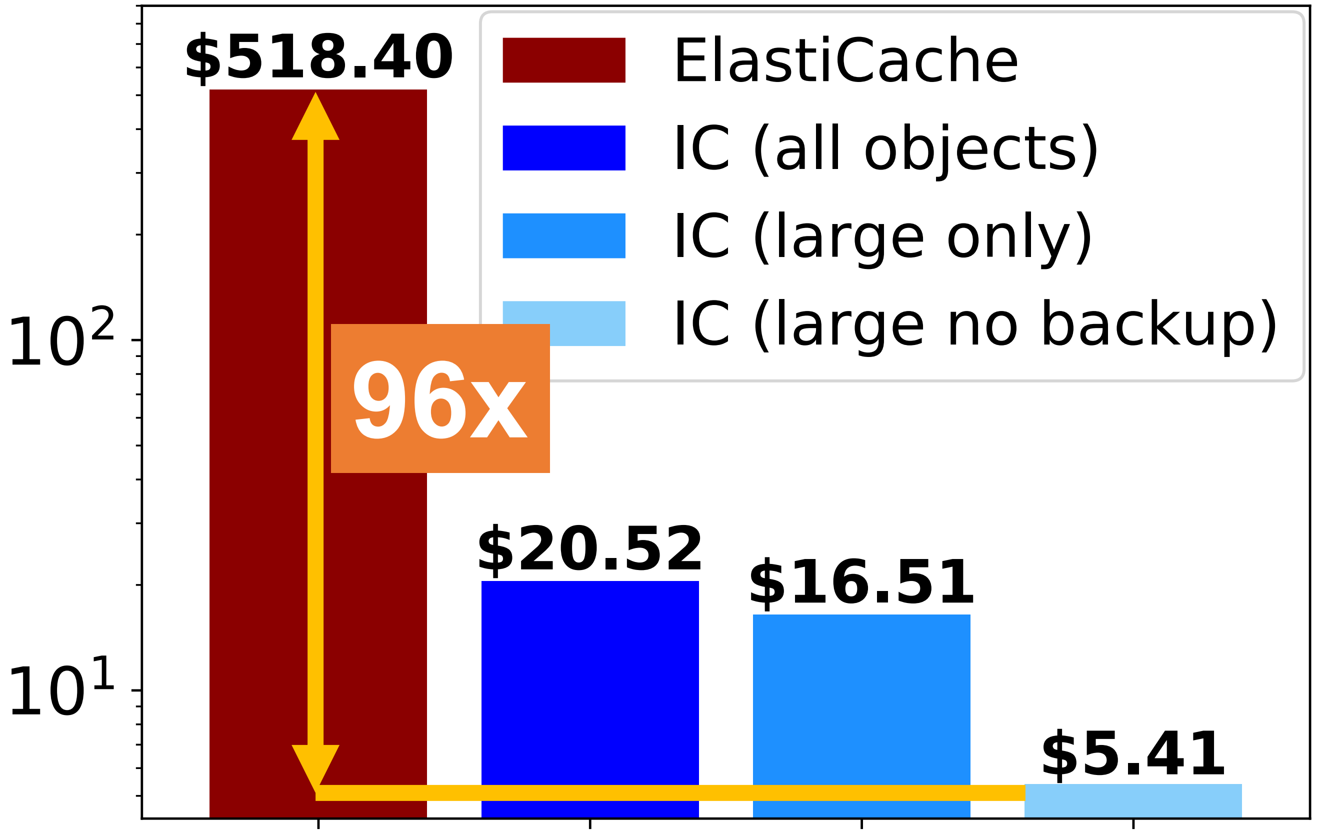
Cost effectiveness of InfiniCache



Workload setup

- All objects
- Large object only
  - Object larger than 10MB

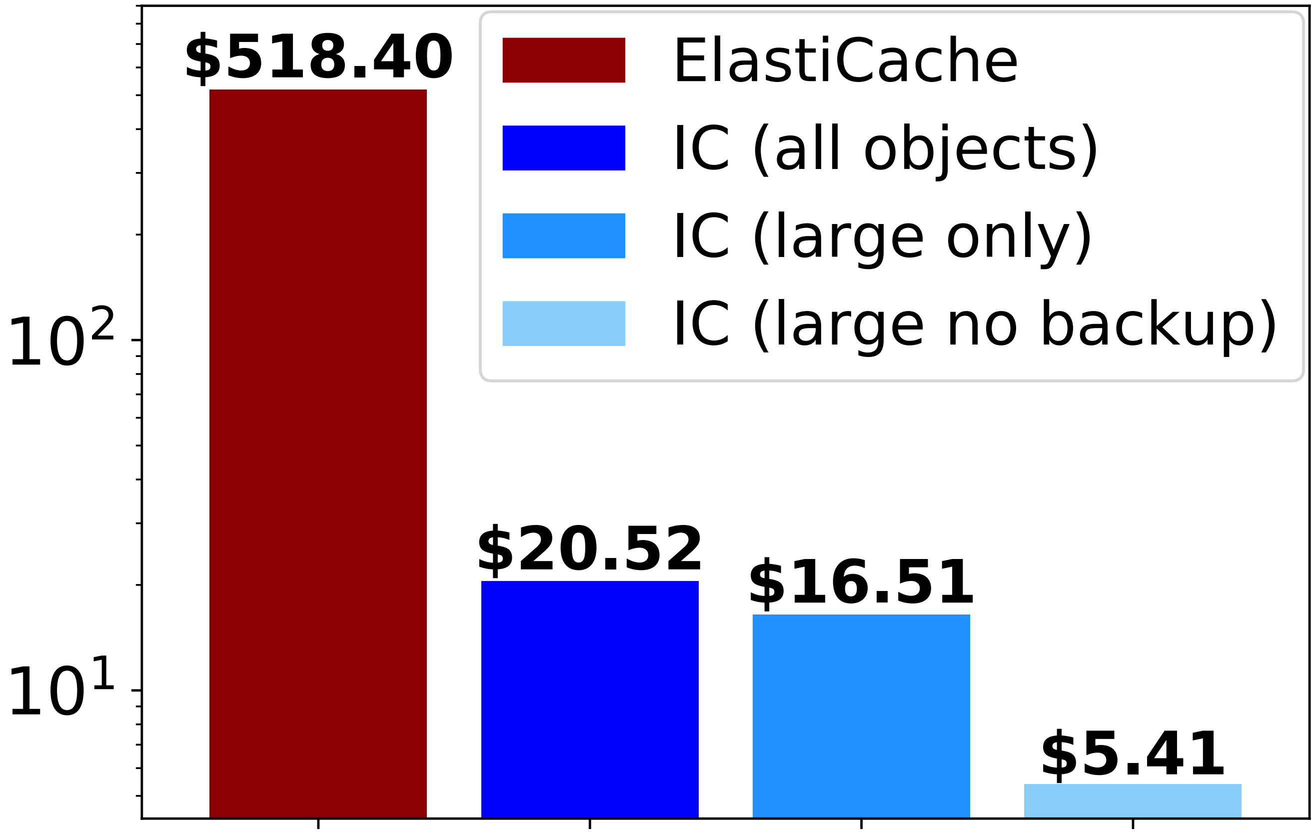
Cost effectiveness of InfiniCache



Workload setup

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

Cost effectiveness of InfiniCache



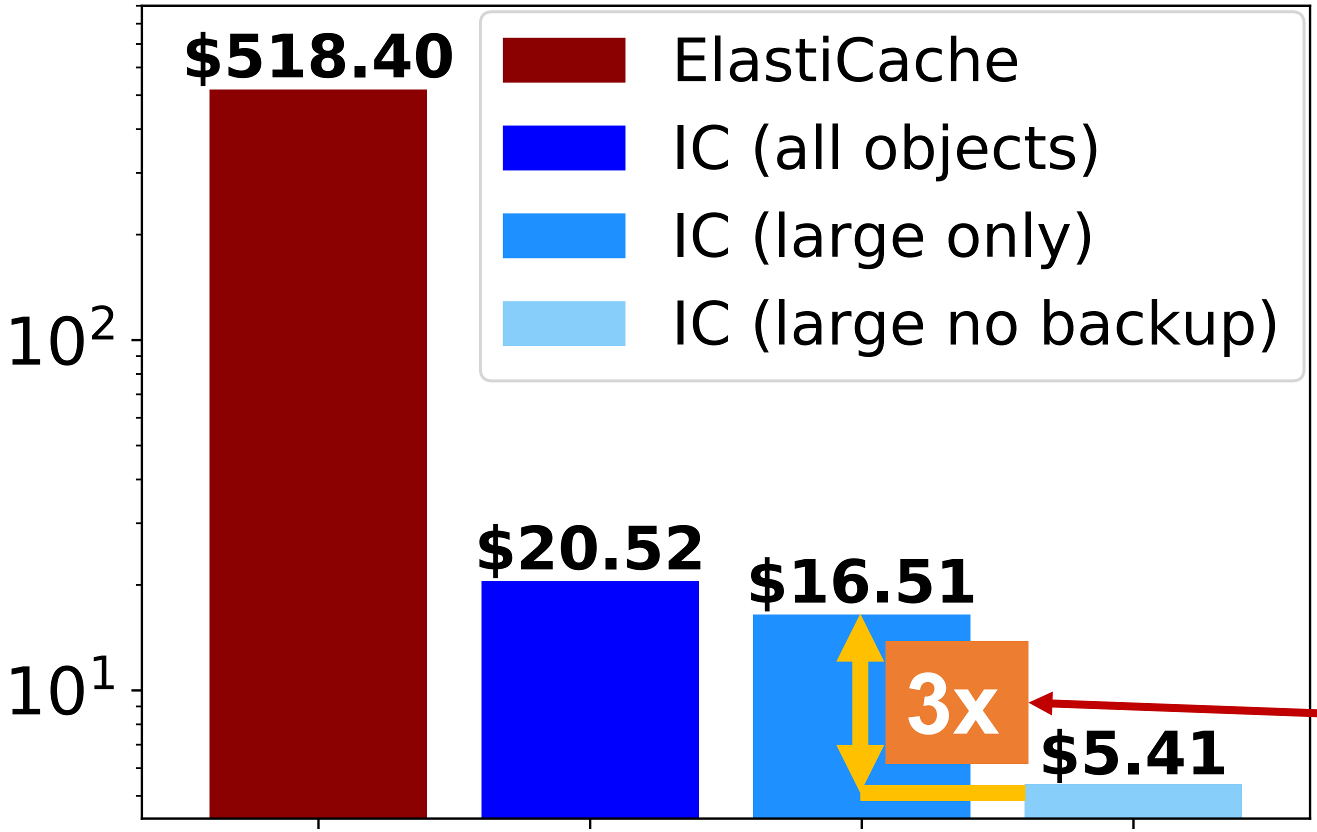
Workload setup

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

Hit ratio

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

Cost effectiveness of InfiniCache



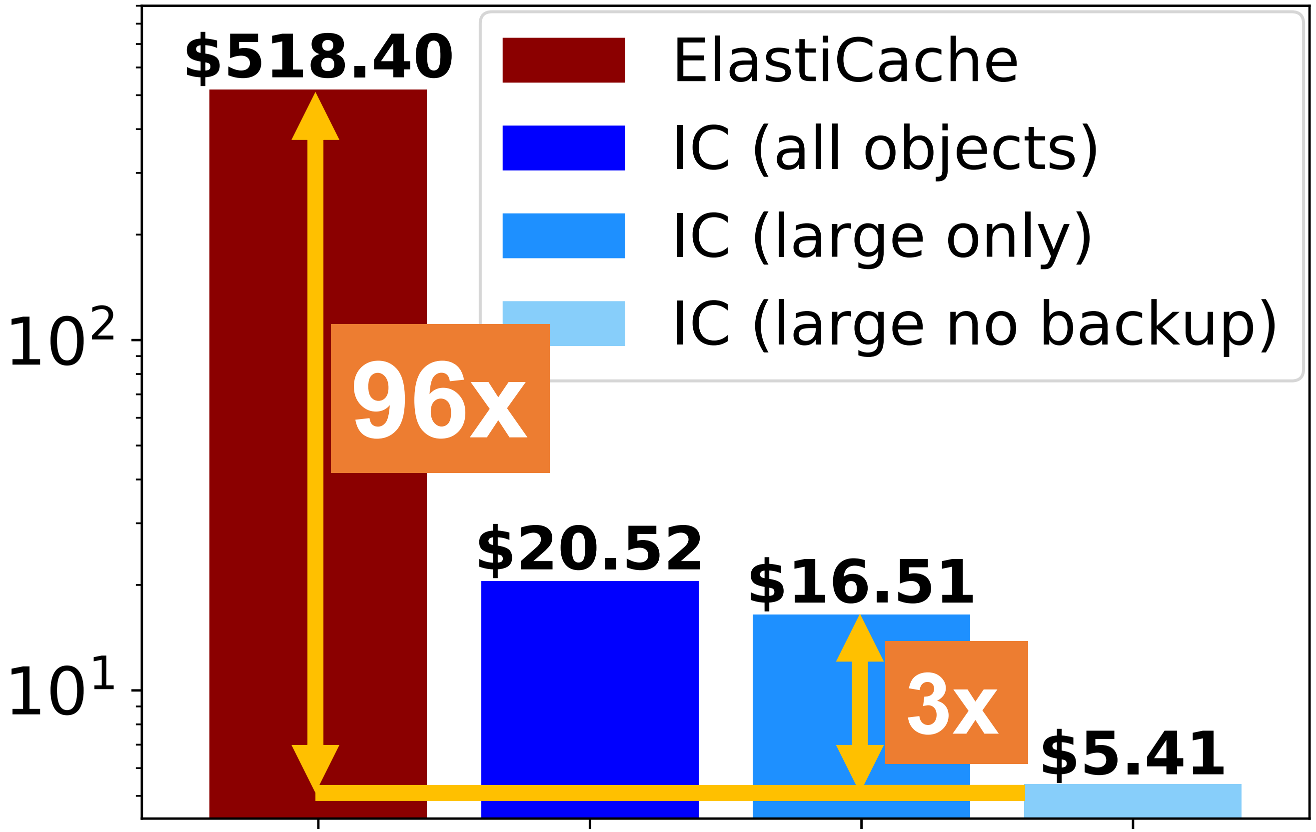
Workload setup

- 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%

Cost effectiveness of InfiniCache

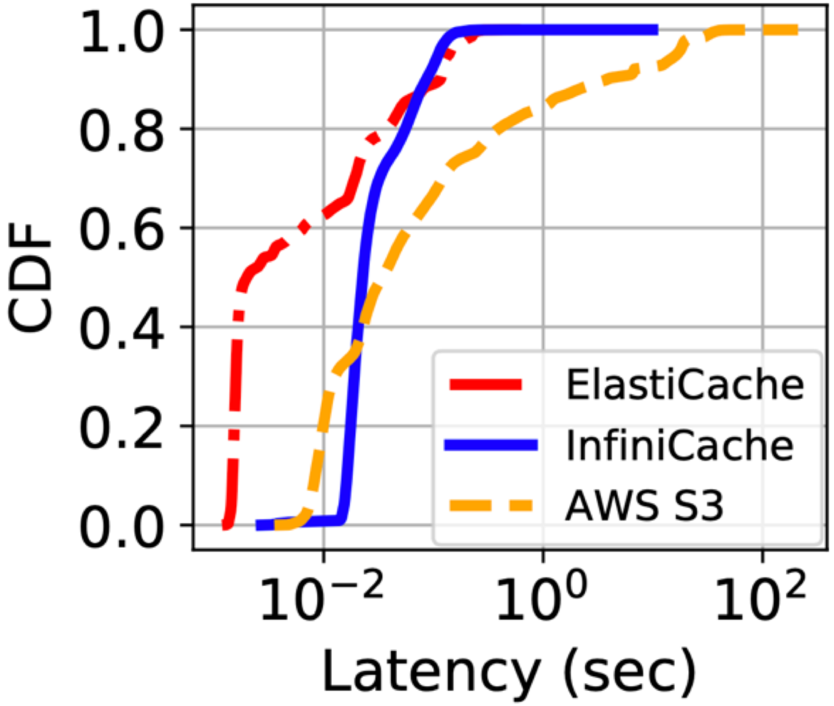


Workload setup

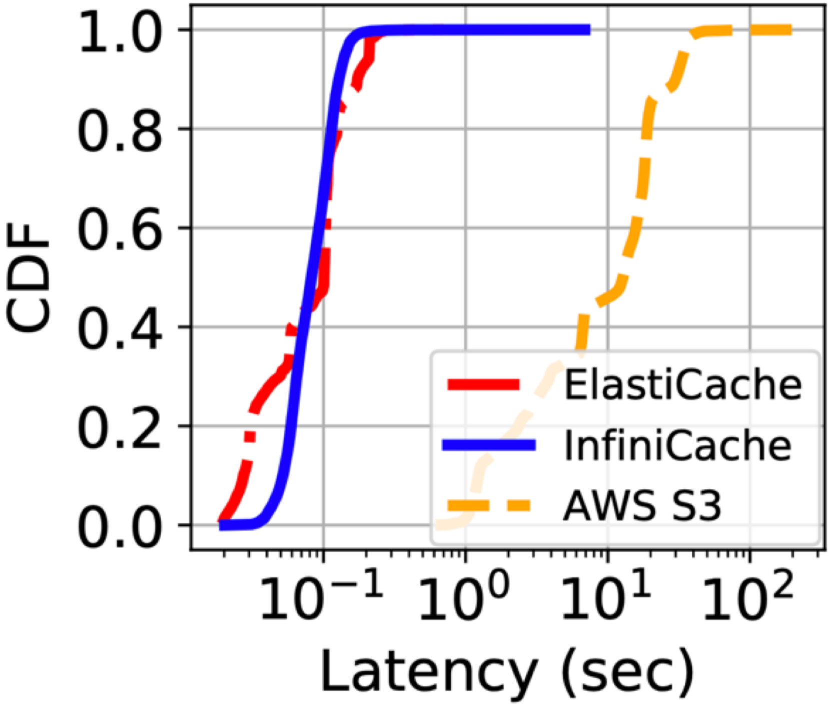
- All objects
- Large object only
  - Object larger than 10MB
- Large object w/o backup

InfiniCache is 31 – 96x cheaper than ElastiCache because tenant does not pay when Lambdas are not running

Performance of InfiniCache



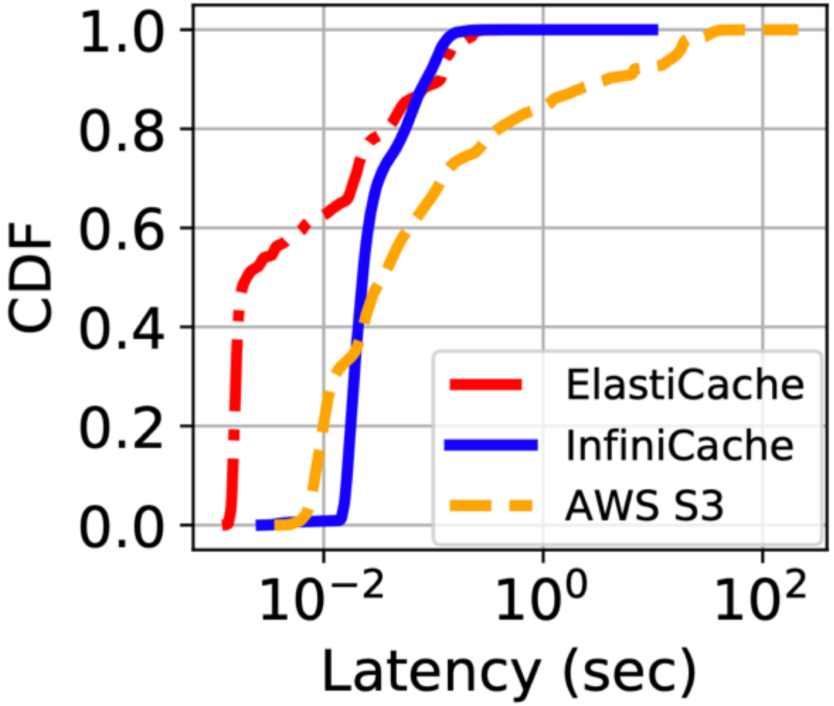
All objects



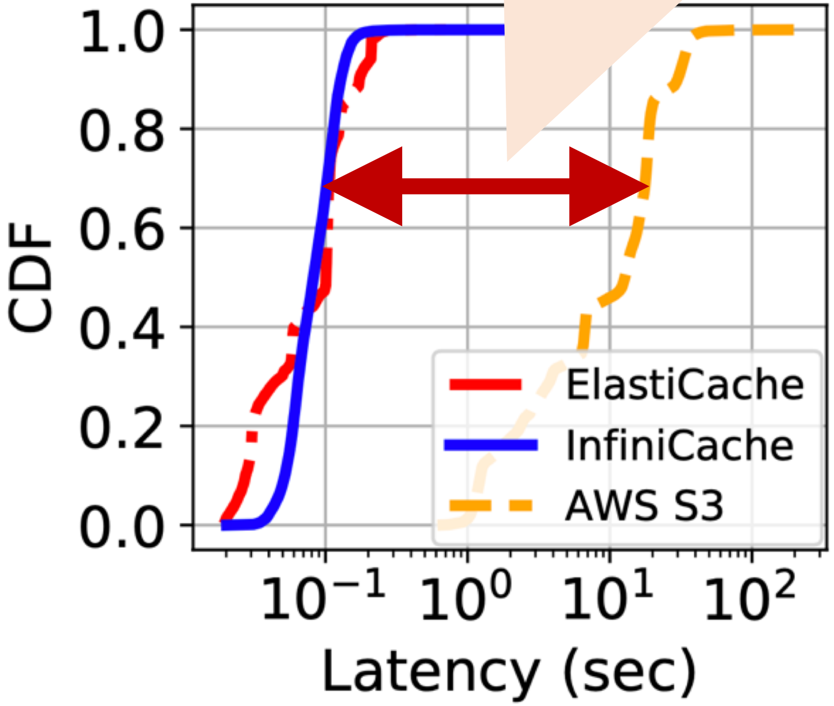
Large objects only

Performance of InfiniCache

> 100 times improvement

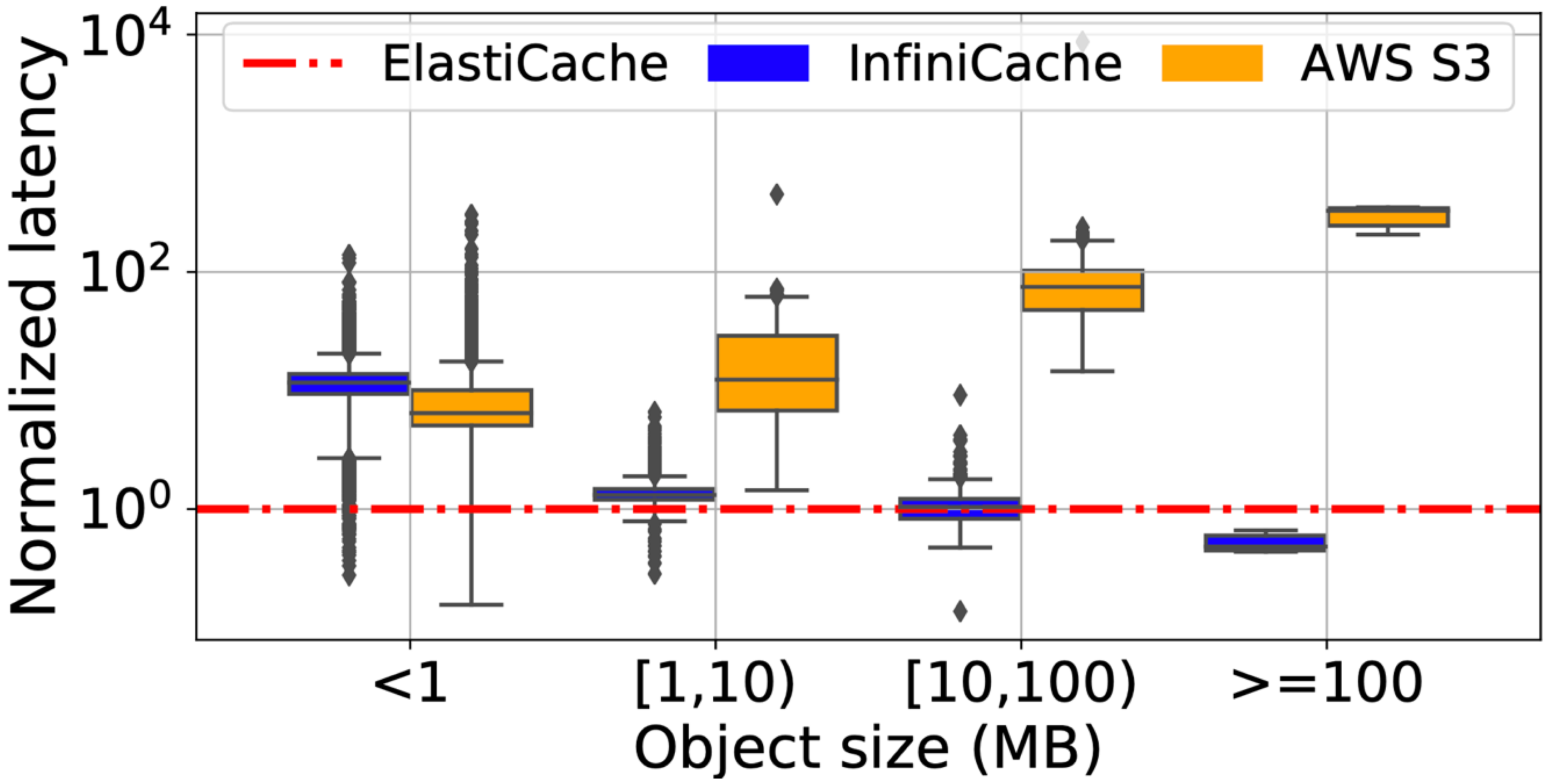


All objects

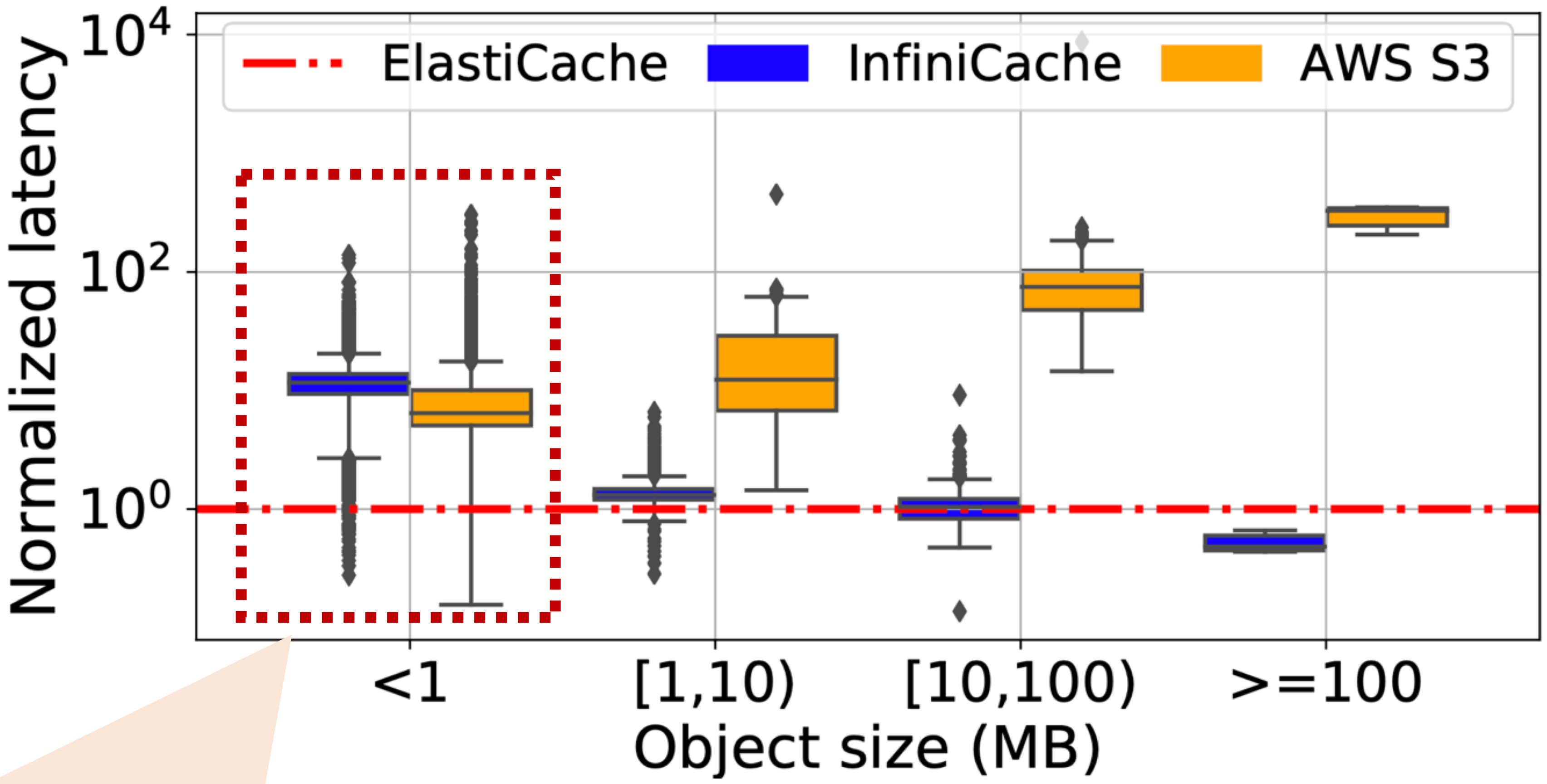


Large objects only

Performance of InfiniCache

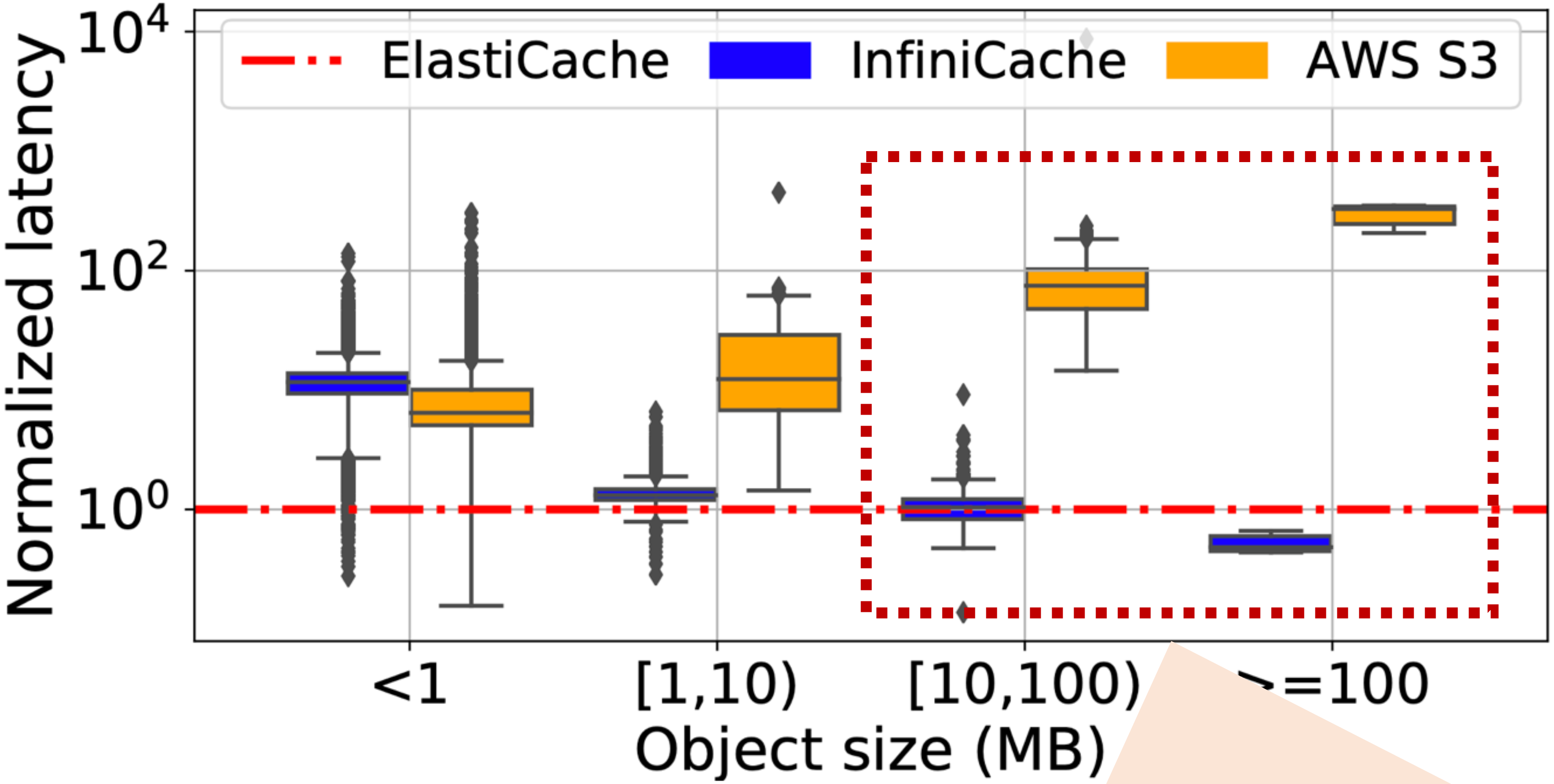


Performance of InfiniCache



Lambda invocation overhead (~13ms) dominates when fetching small objects

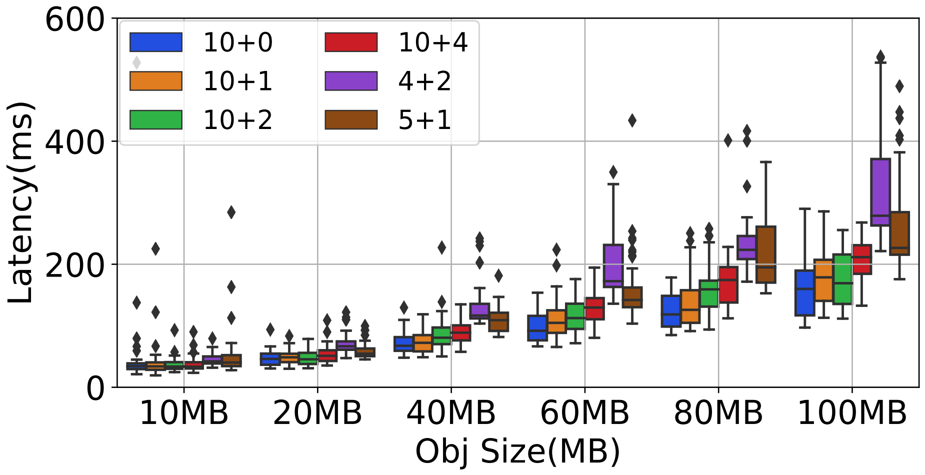
Performance of InfiniCache



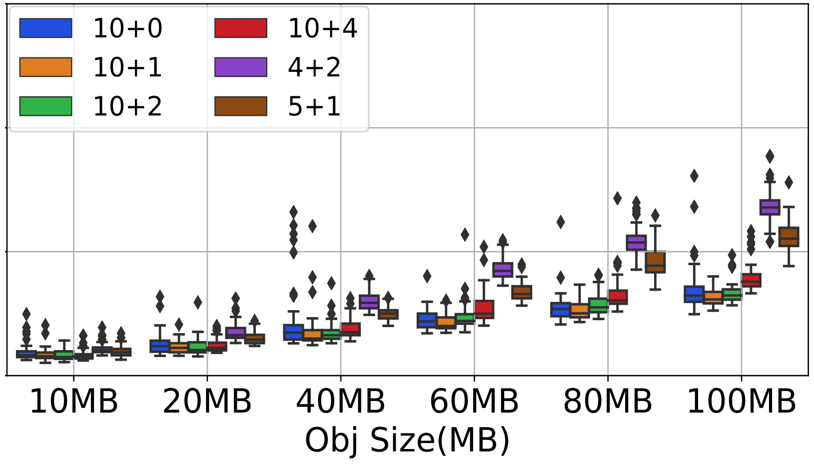
InfiniCache achieves same or higher performance than ElastiCache for large objects

Evaluation

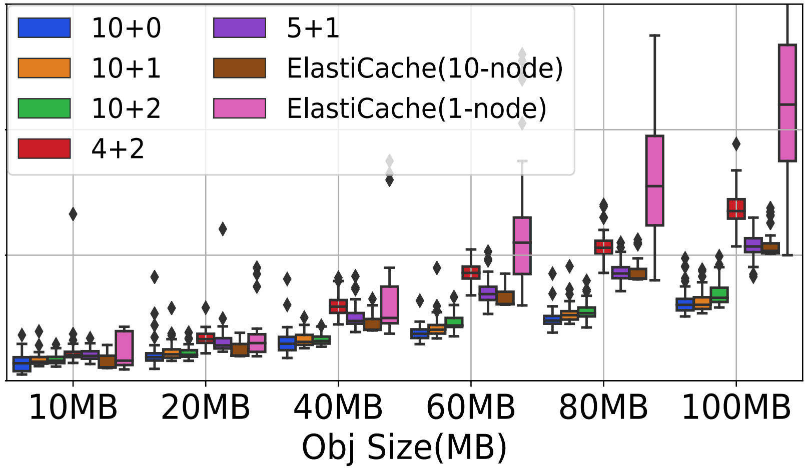
- Microbenchmark



512MB Lambda memory



2048MB Lambda memory

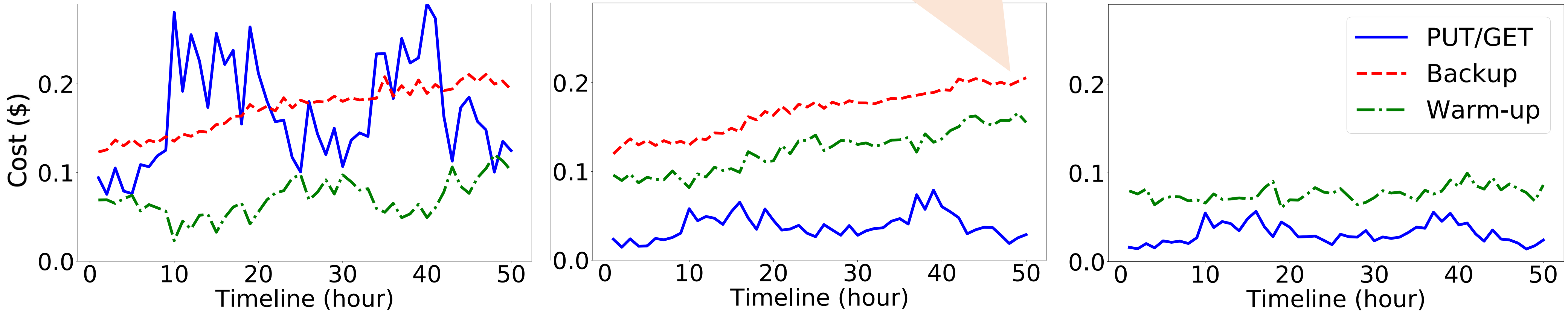


3008MB Lambda memory

Evaluation – Production Workloads

- Cost Breakdown
  - Warm-up cost
  - Backup cost
  - PUT/SET cost

Backup and Warm-up cost dominate total cost



All objects | Large objects only | Large objects only w/o backup

Conclusion

- InfiniCache is the first in-memory cache system built atop a serverless computing platform (AWS λ)
- InfiniCache synthesizes a series of techniques to achieve high performance while maintaining good data availability
- InfiniCache improves the cost-effectiveness by 31-96x compared to AWS ElastiCache

Thank you!

FORK ME ON GITHUB!

- Contact: Ao Wang – awang24@gmu.edu,

  Jingyuan Zhang – jzhang33@gmu.edu

- https://github.com/mason-leap-lab/infinicache

Supplementary Topics

- Keep Lambdas alive
- Advanced proxy-lambda interaction
- How to use InfiniCache?
    1. Storage for machine learning applications.
    2. Client in the Lambda, a P2P approach

Keep Lambdas Alive - Problem

- What we knew?
  - Lambda instances can be reclaimed any time.
  - If invoked periodically every 60s, the lifetime ranges from 1 minute to 8.3 hours, with median instance lifetime ... is 6.2 hours.
  - If idle, the instance will be reclamied within 27 minutes. [Wang ATC'18]

- Problem?
  - We have N Lambda functions, 1 instance per function, how to avoid data loss?

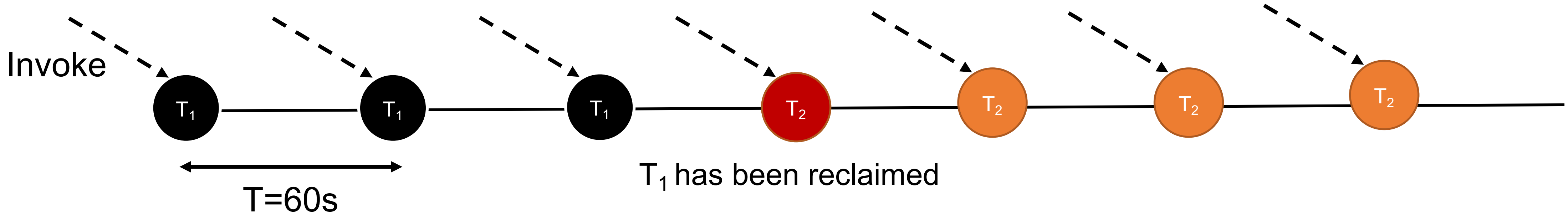
Keep Lambdas Alive - Idea

- Idea?
  - Invoking Lambda instances every 60s, chances are N instances will not all be reclaimed at any moment given the lifetime various.
  - With erasure coding, data are stored in D+P Lambda instances. If more than D instances survive on requesting, the data is recoverable.

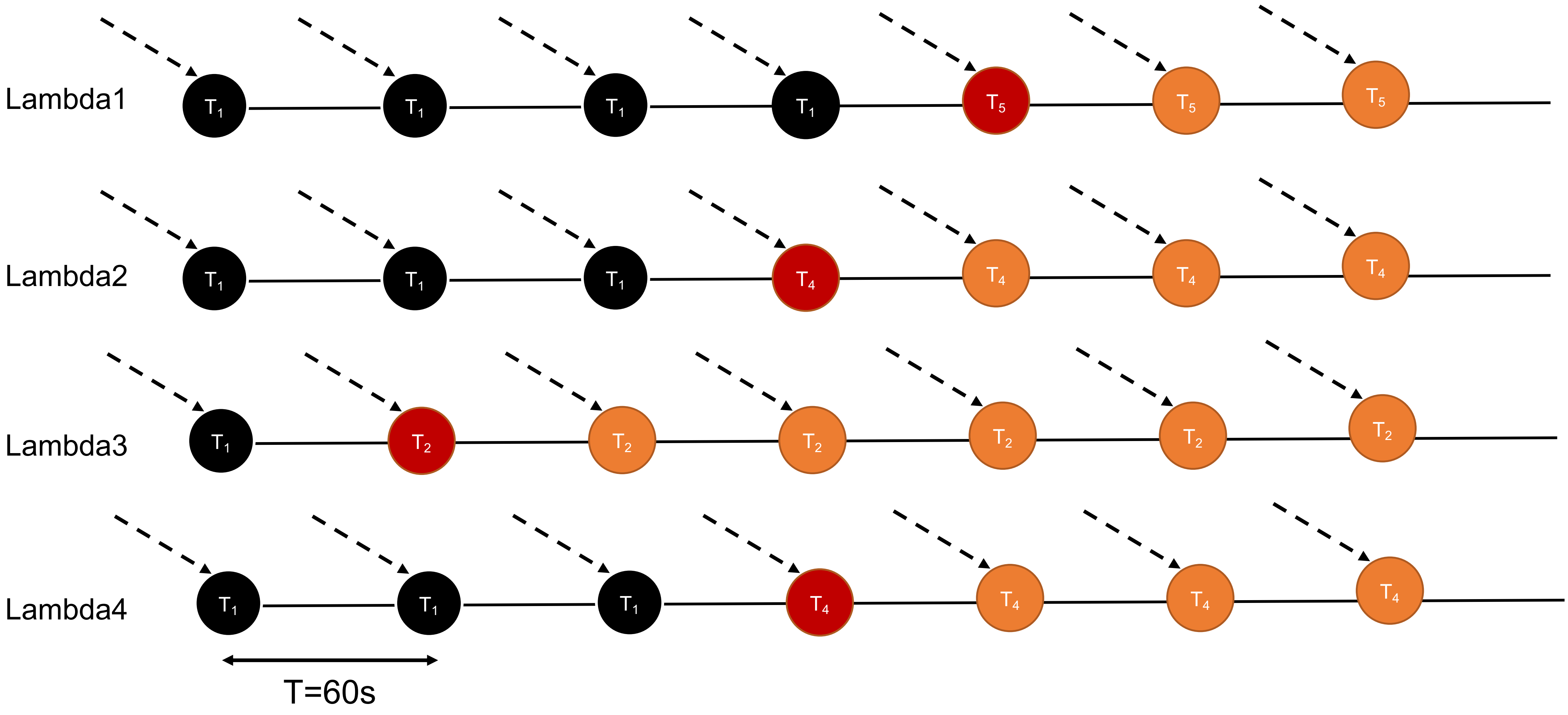
- Challenge?
  - If N instances get reclaimed at the same time, data can't be preserved.
  - If the chance of losing P instances out of any D+P instances is high enough, data can't be preserved.
  - Can we invoke instances with longer interval, how about 9 minutes?

Keep Lambdas Alive - Experiment

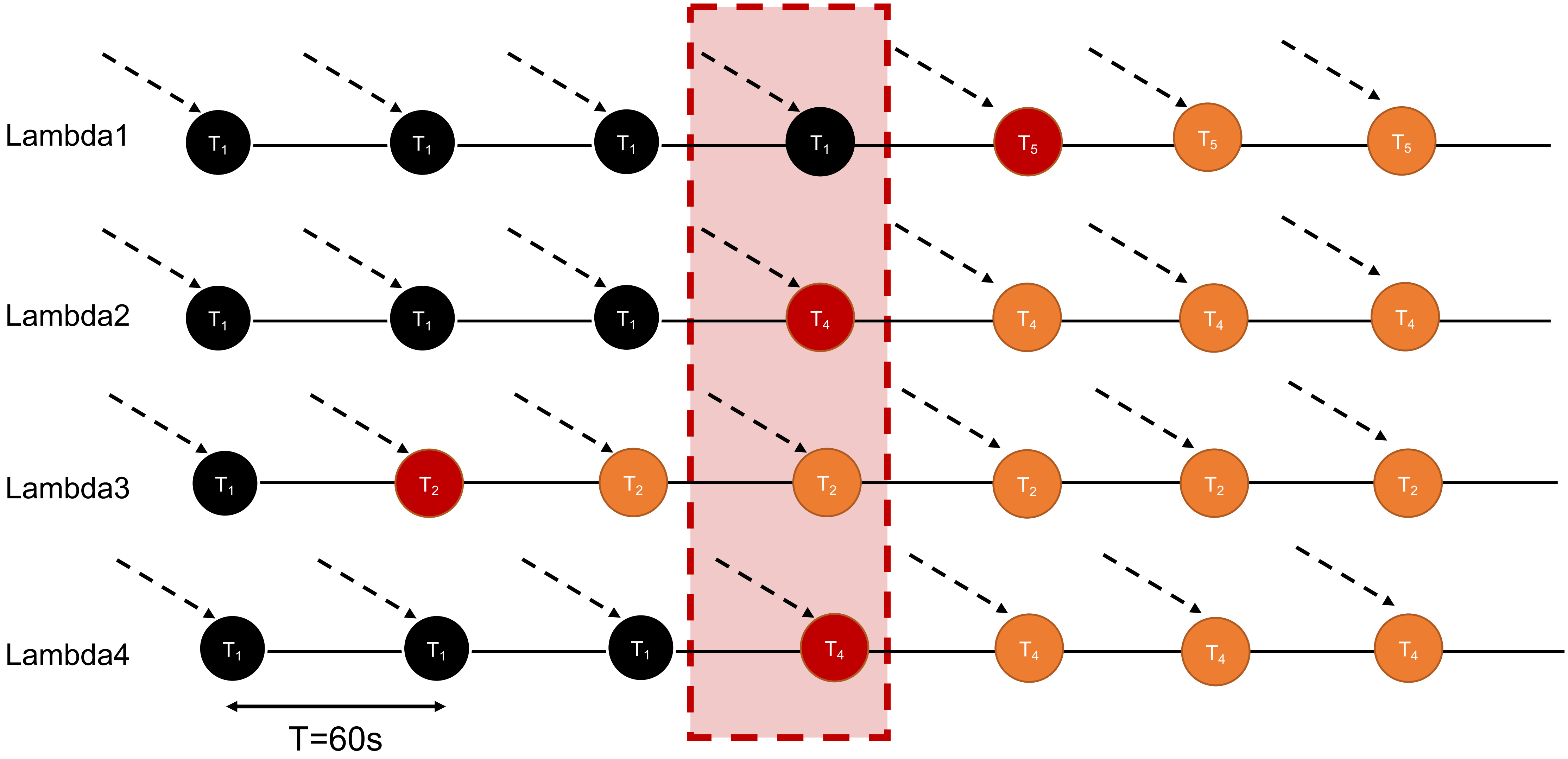
- Solution: Experiment
  - N = 400 Lambda functions was deployed. 1 instance per function.
  - Instances are invoked every T=60s and T=540s.
  - Every invocation, the start time of the instance is recorded. So a finding of new start timestamp indicates the old instance is reclaimed.
  - Every T interval, the number of new instances is reported.



Keep Lambdas Alive - Experiment



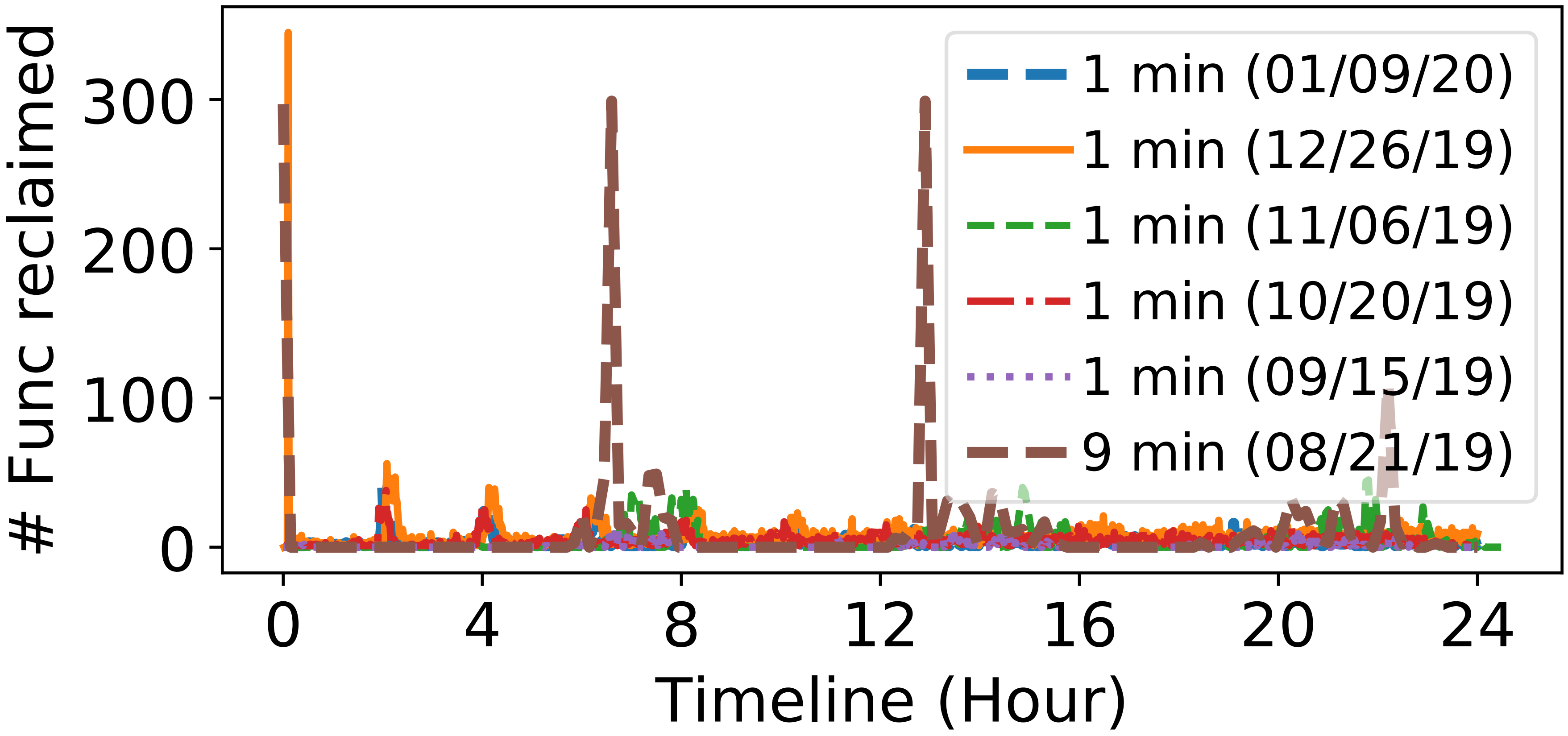
Keep Lambdas Alive - Experiment



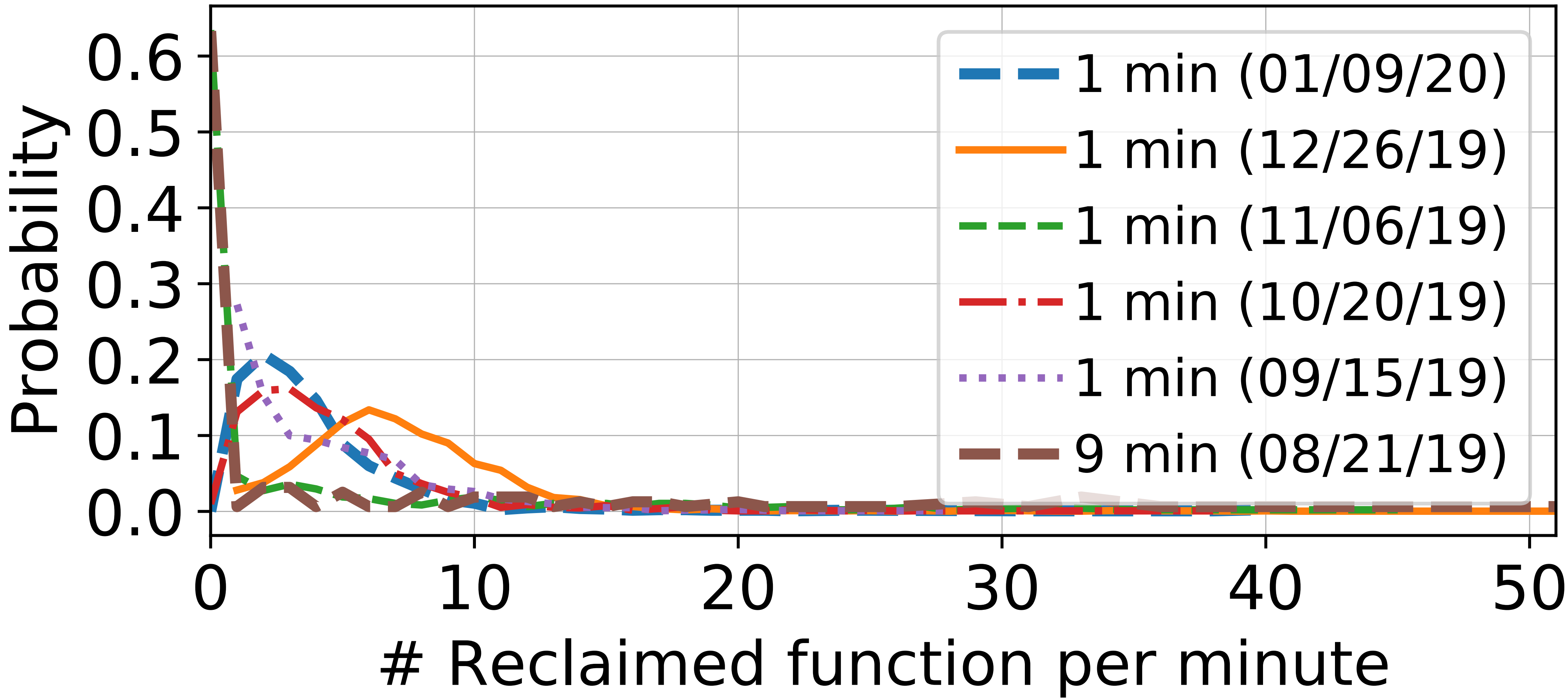
At T_4 , 2 out of 4 Lambda instances have been reclaimed

Keep Lambdas Alive - Result

- The experiment had been carried for 6 months to study policy changes of AWS Lambda.



Keep Lambdas Alive - Distribution



Keep Lambdas Alive - Observation

- In Sep 2019, if we invoke Lambda instances every 60s:
  - We observed 10+ out of 400 Lambda instances get reclaimed within one-minute interval for 2 out of 1440 samples (24 hours)
  - 87% of samples loss no more than 2 instances within one-minute interval
- Later experiments observed policy changes, but trends hold.

With erasure coding, can we recover data from this loss?

Keep Lambdas Alive - Calculation

- Assuming a configuration of erasure coding **n** = d + p
  - If **i** (i > p) chunks are lost, data are unrecoverable.
- Assuming for **N** Lambda instances
  - **r** instances are reclaimed within one-minute interval.
- The chance P_i the data are lost because i chunks are lost is:

$$P_i = \frac{C(r,i)C(N-r,n-i)}{C(N,n)}$$

- The aggregated chance P(r) the data are lost is:

$$P(r) = \sum_{i=p+1}^{n} P_i \cong Pp_{+1}$$

Keep Lambdas Alive – Calculation cont'd

- The chance P of losing any data within one-minute interval is:

$$P = \sum_{r=p+1}^{N} P(r)p_d(r)$$

$$P \cong \sum_{r=p+1}^{N} \frac{C(r, p+1)C(N-r, n-p-1)}{C(N, n)} p_d(r)$$

While $p_d(r)$ is the chance of reclaiming r instances within that on—miniute interval.

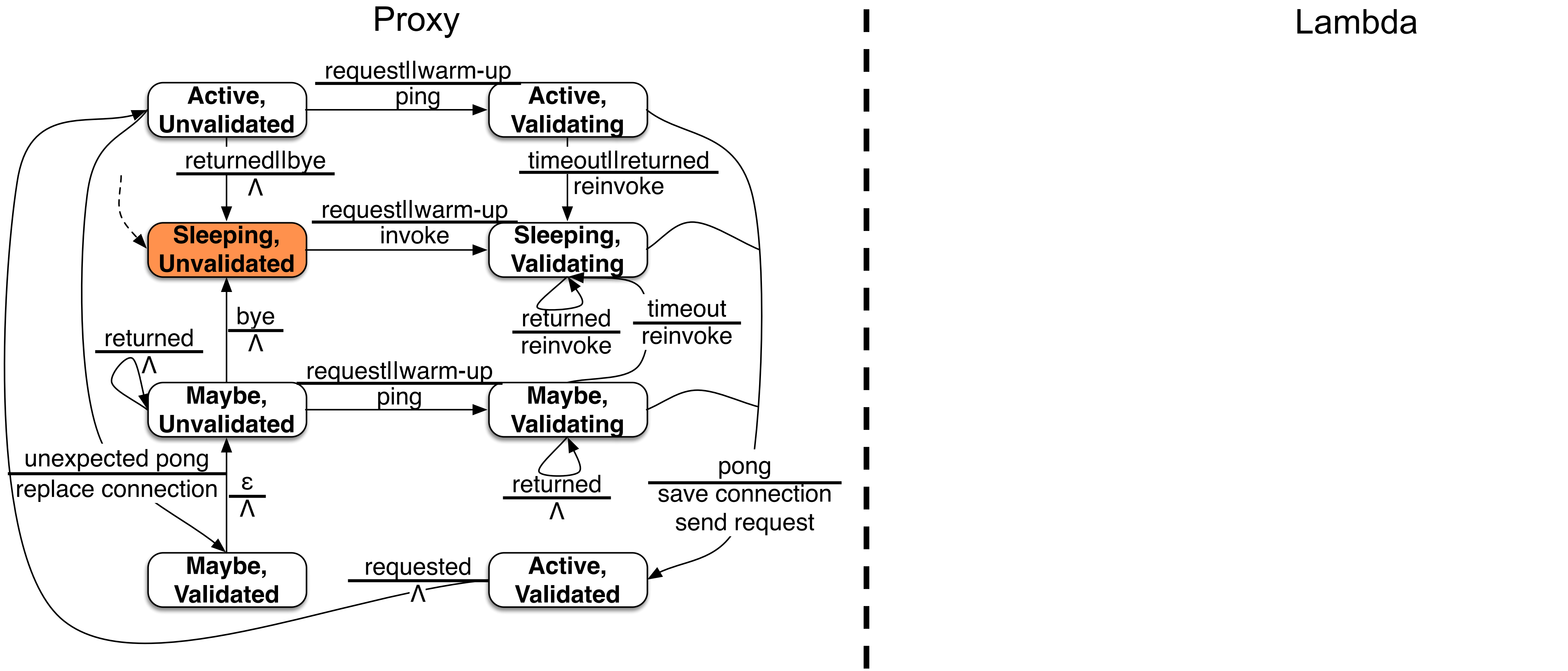
- The result shows P = 0.0039% in September, and at most 0.11% in later months.

Keep Lambdas Alive - Conclusion

- Combine following techniques, we can hold data in Lambdas instances for sufficient long time:
  - Erasure coding
  - Invoke instances every fixed interval of 60s (Periodical warm-up)

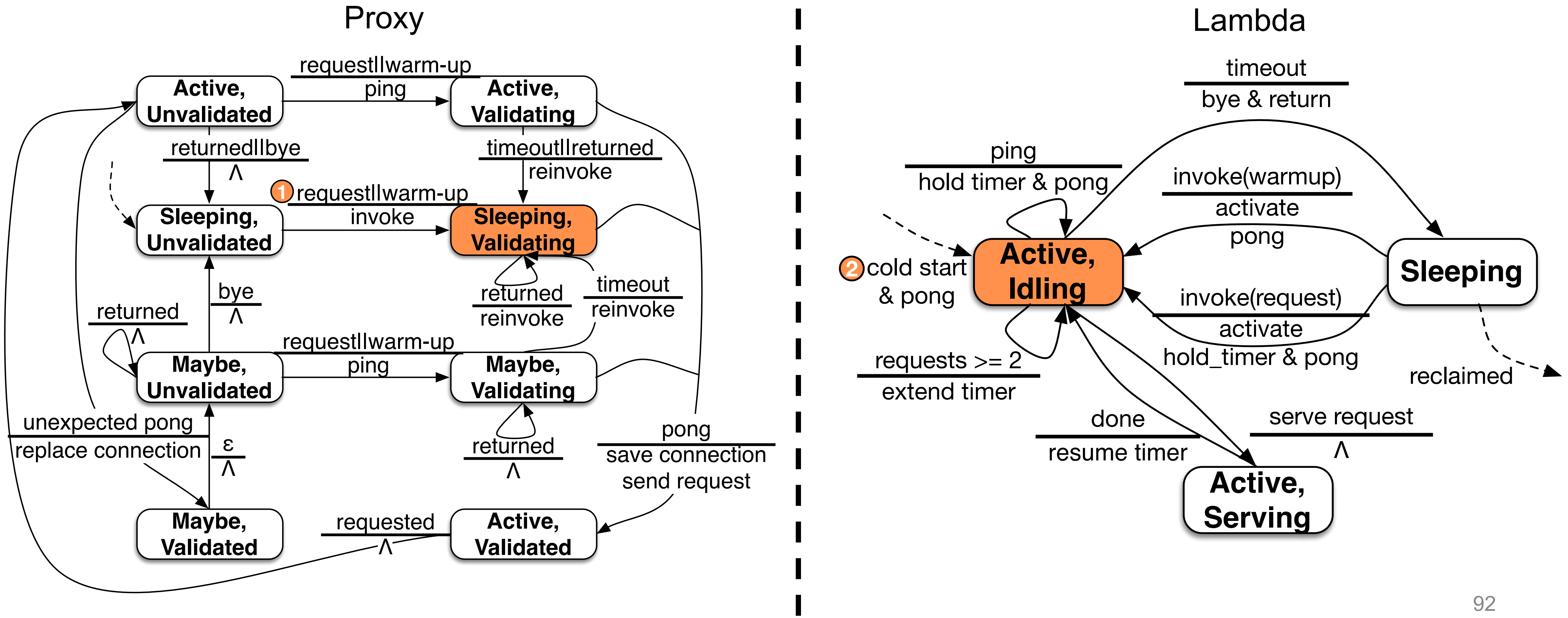
Advanced proxy-lambda interaction

- Very first request



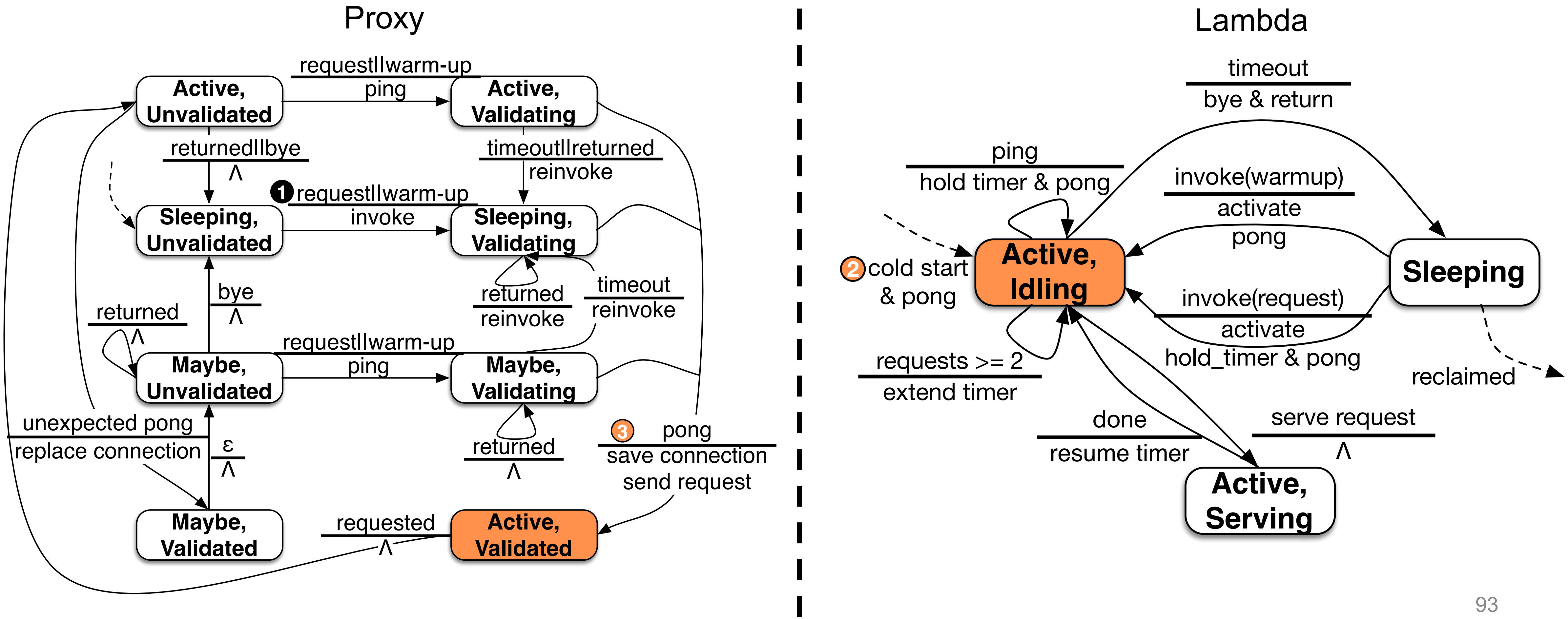
Advanced proxy-lambda interaction

- Very first request



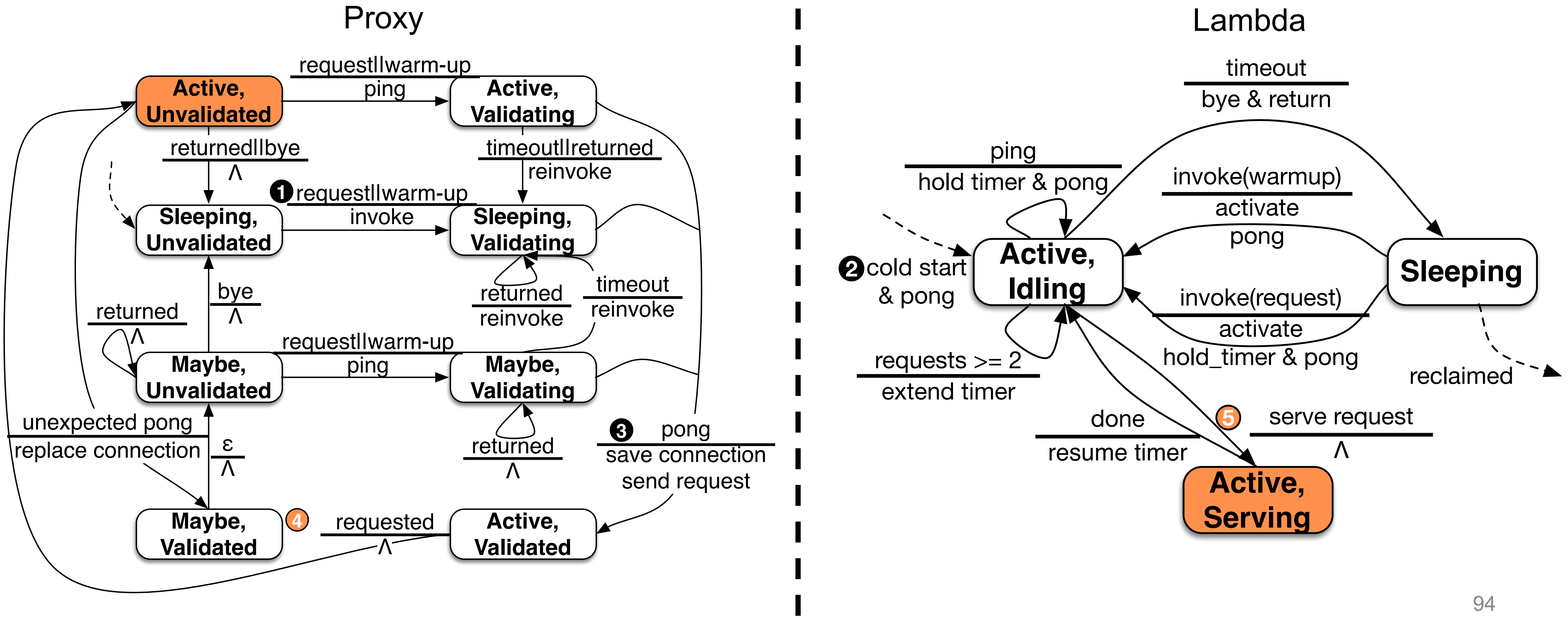
Advanced proxy-lambda interaction

- Very first request



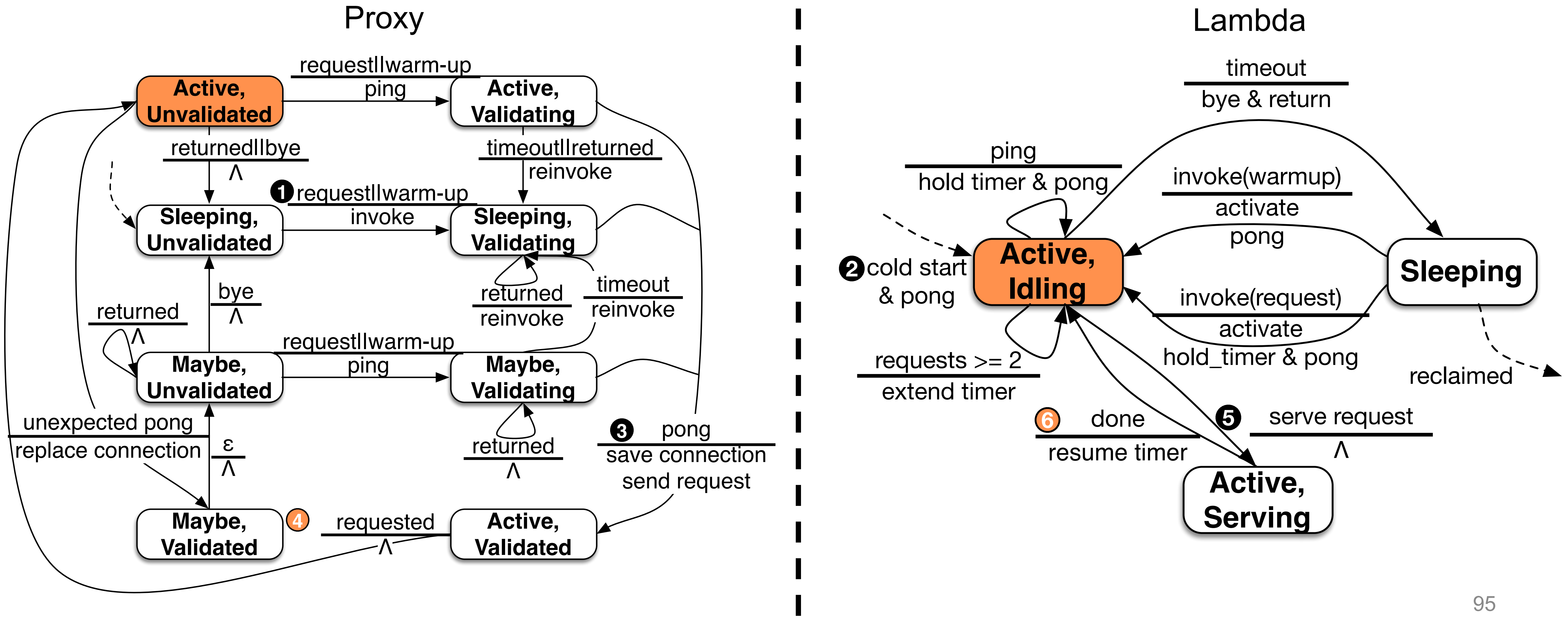
Advanced proxy-lambda interaction

- Very first request



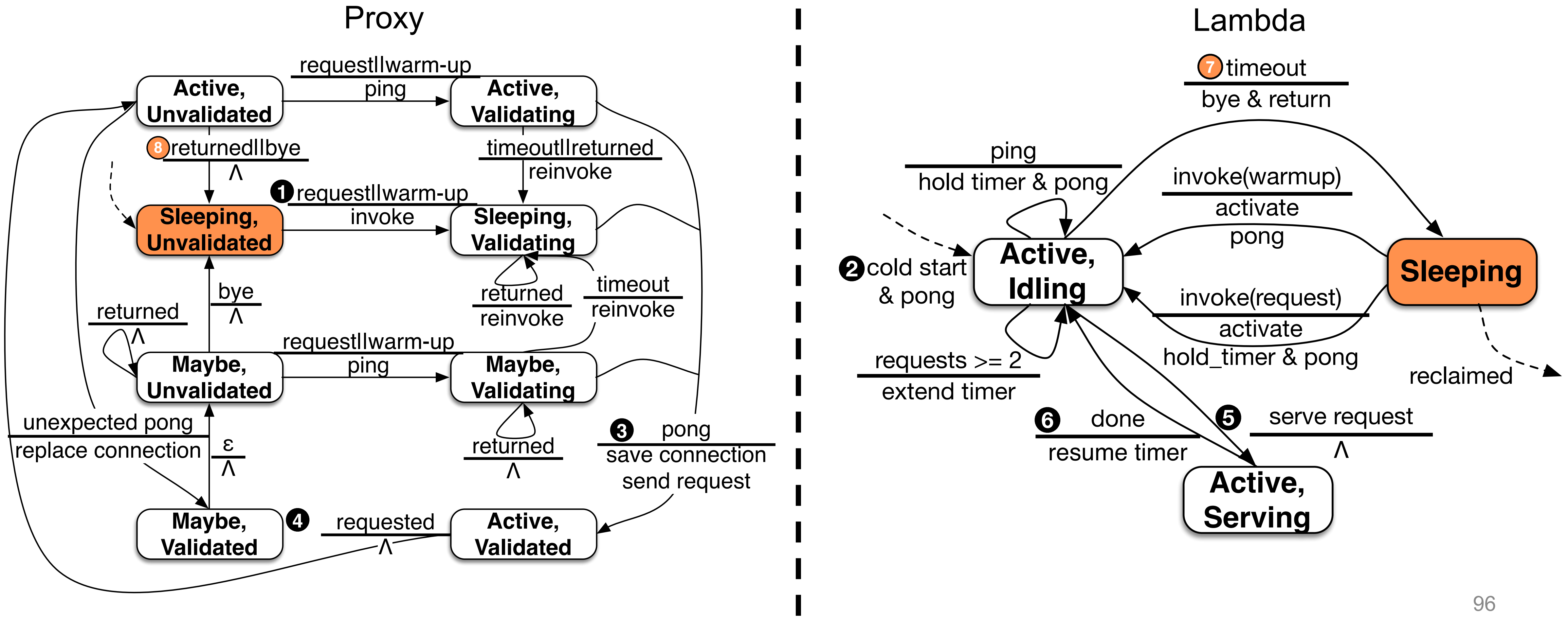
Advanced proxy-lambda interaction

- Very first request



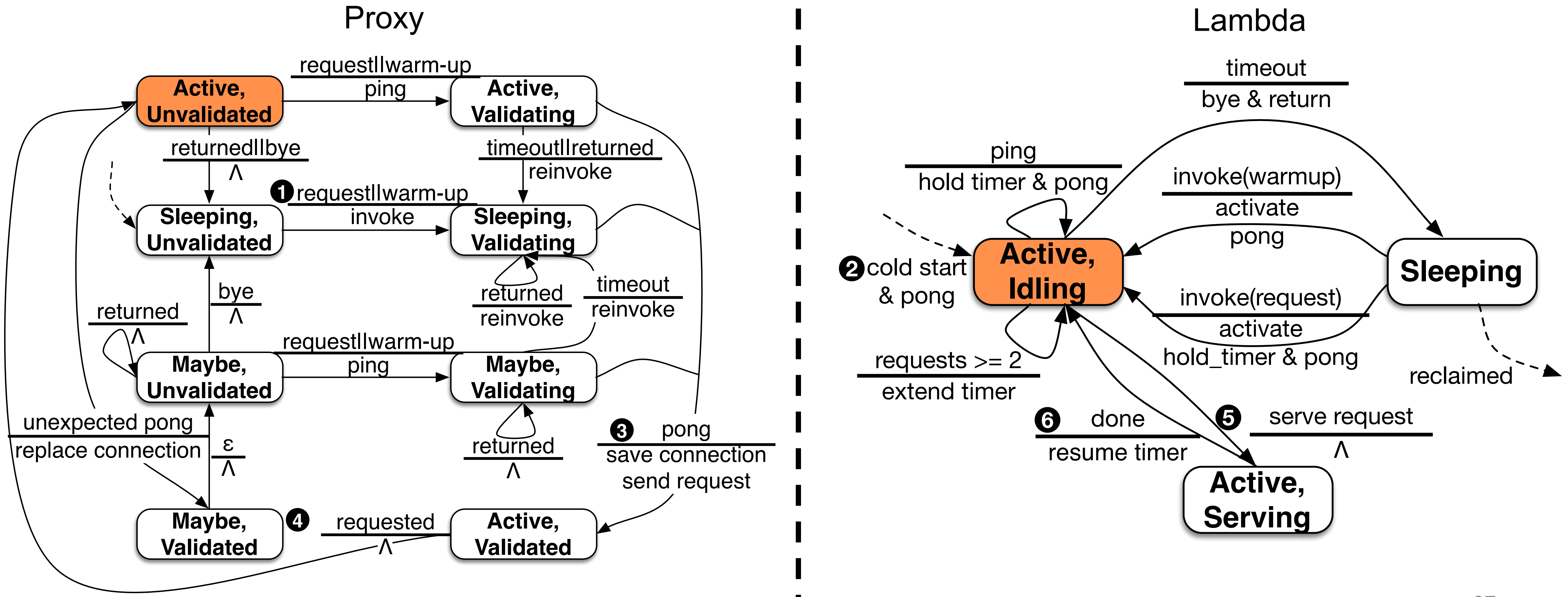
Advanced proxy-lambda interaction

- Very first request



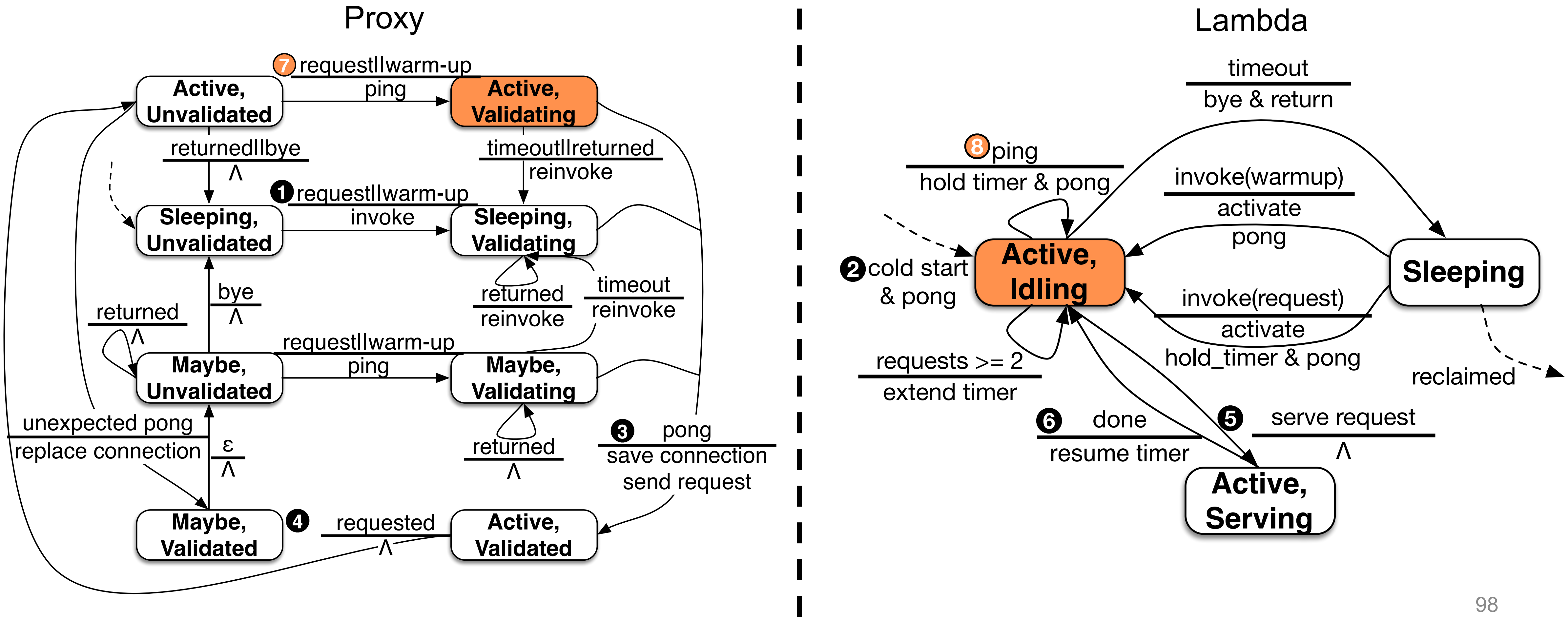
Advanced proxy-lambda interaction

- Second request in the same session



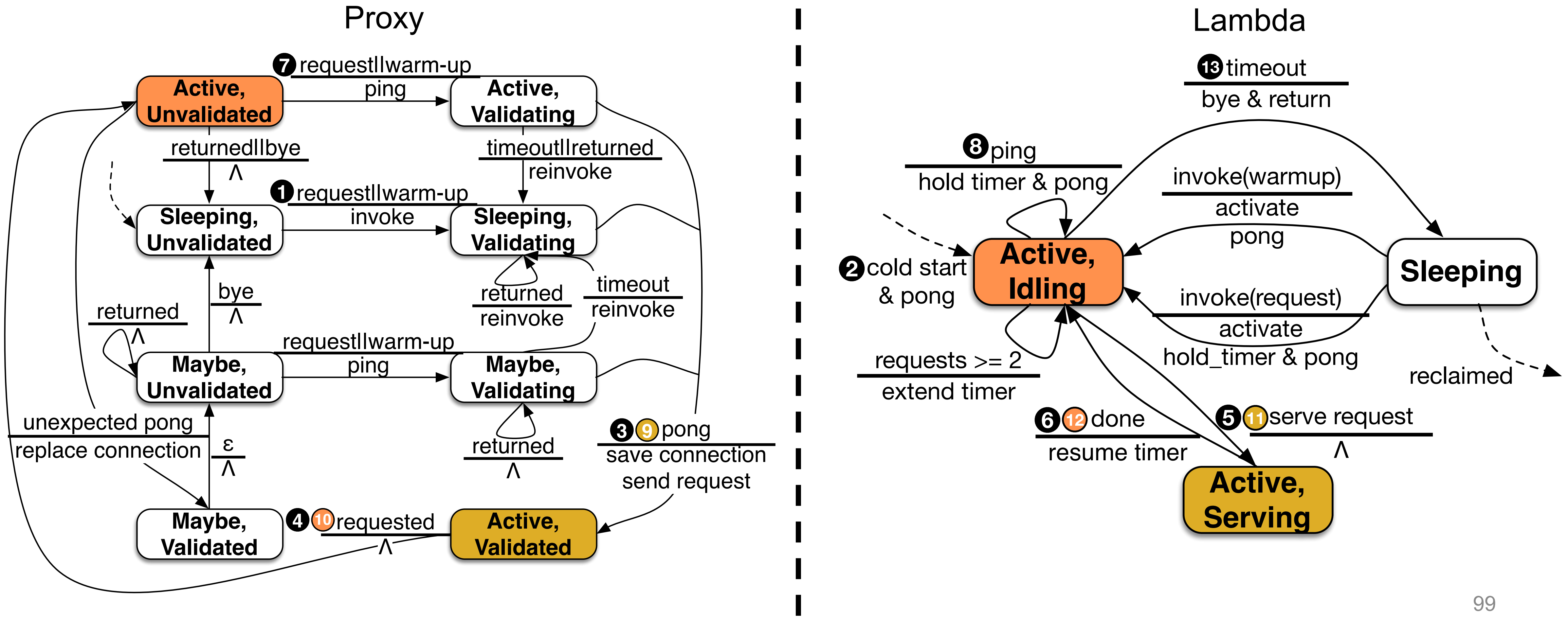
Advanced proxy-lambda interaction

- Second request in the same session



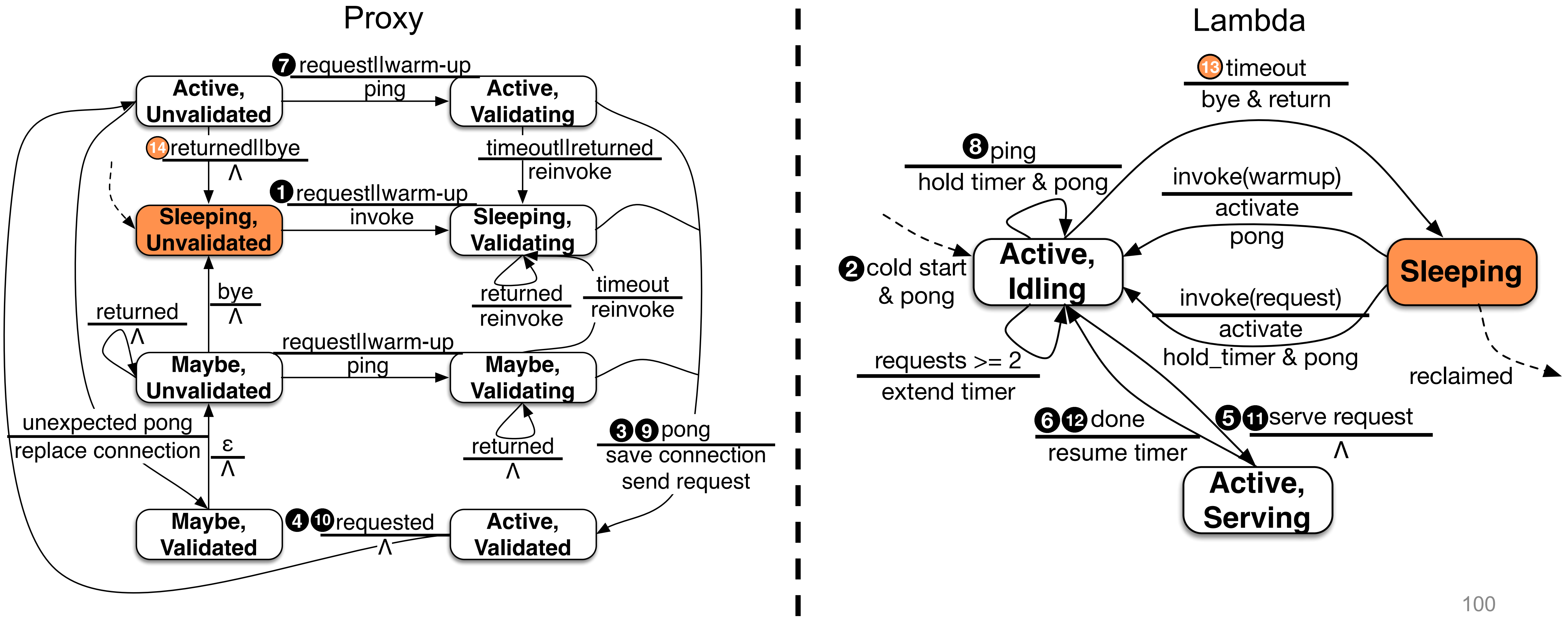
Advanced proxy-lambda interaction

- Second request in the same session

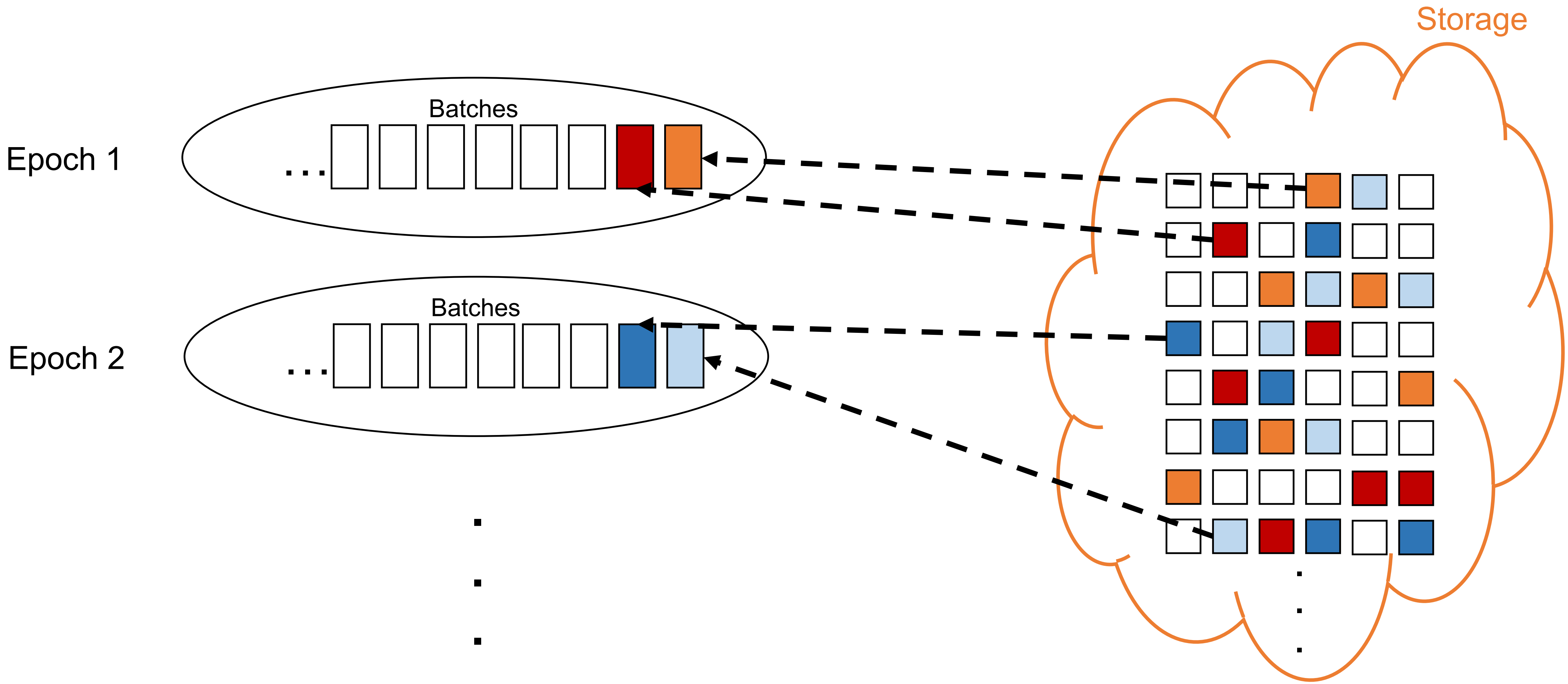


Advanced proxy-lambda interaction

- Second request in the same session



Storage for Machine Learning Applications



Storage for Machine Learning Applications

- S3 as storage
  - Pros: cheap
  - Cons: slow
- ElasticCache as storage
  - Pros: quick
  - Cons: expensive, slow to launch and shutdown.

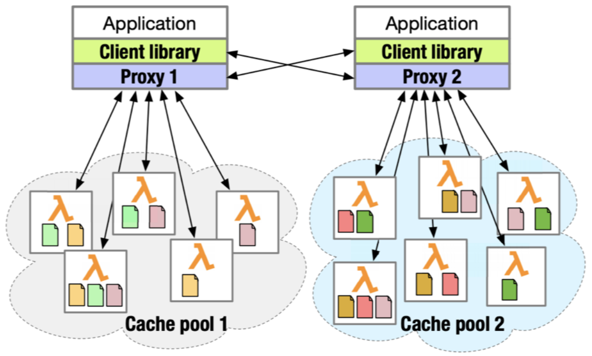
Storage for Machine Learning Applications

- Challenges to use InfiniCache as storage
  - Most of ML frameworks are Python based.
  - Must load data from S3, and set to the InfiniCache in epoch 1.

Is it worthy?

Client in the Lambda, a P2P approach

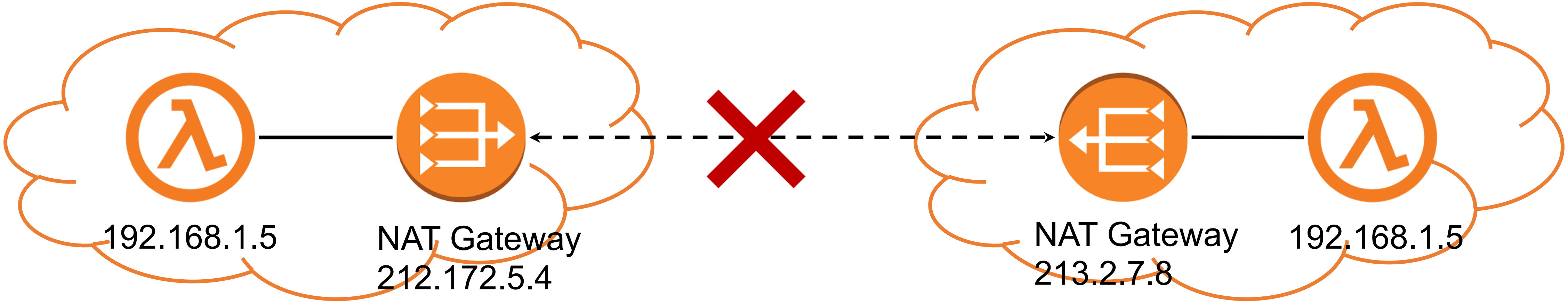
- In original InfiniCache design, the proxy is co-located with client.
  - The expense of the proxy is covered by the client.
  - A client must allow inbound connection.



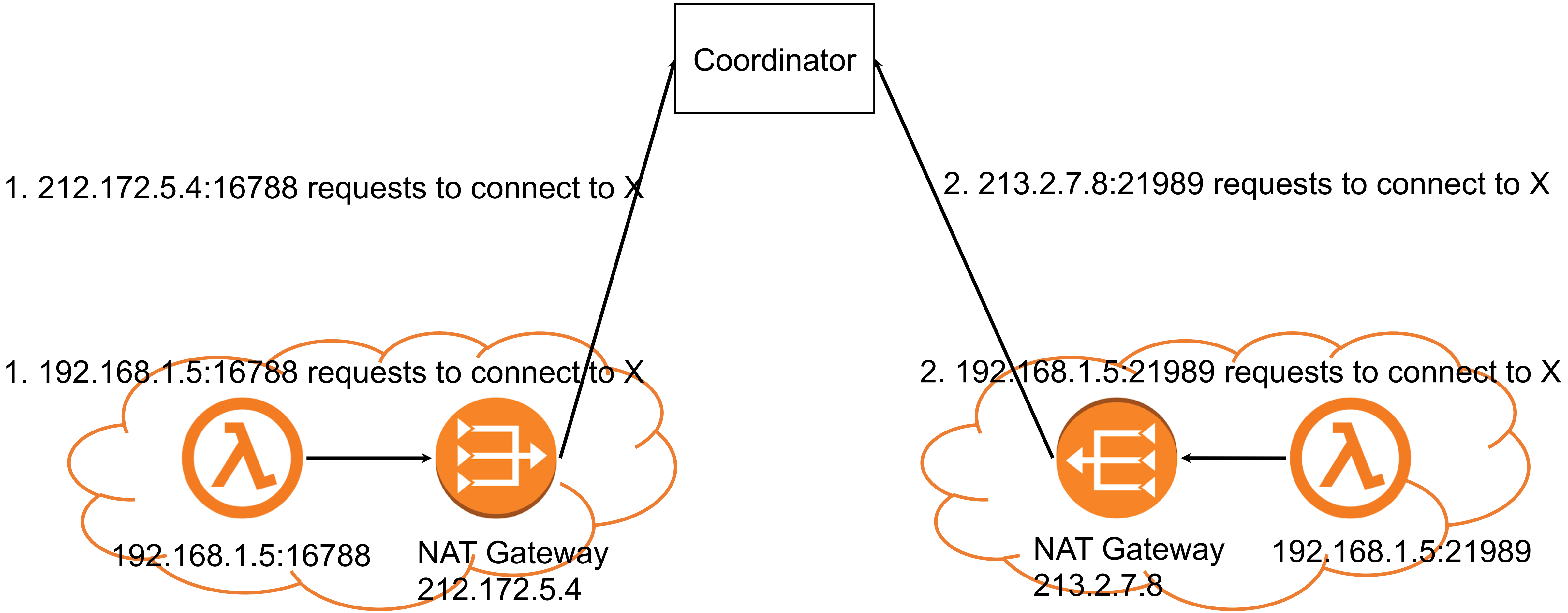
How Lambda functions benefit from the InfiniCache?

Client in the Lambda – P2P network

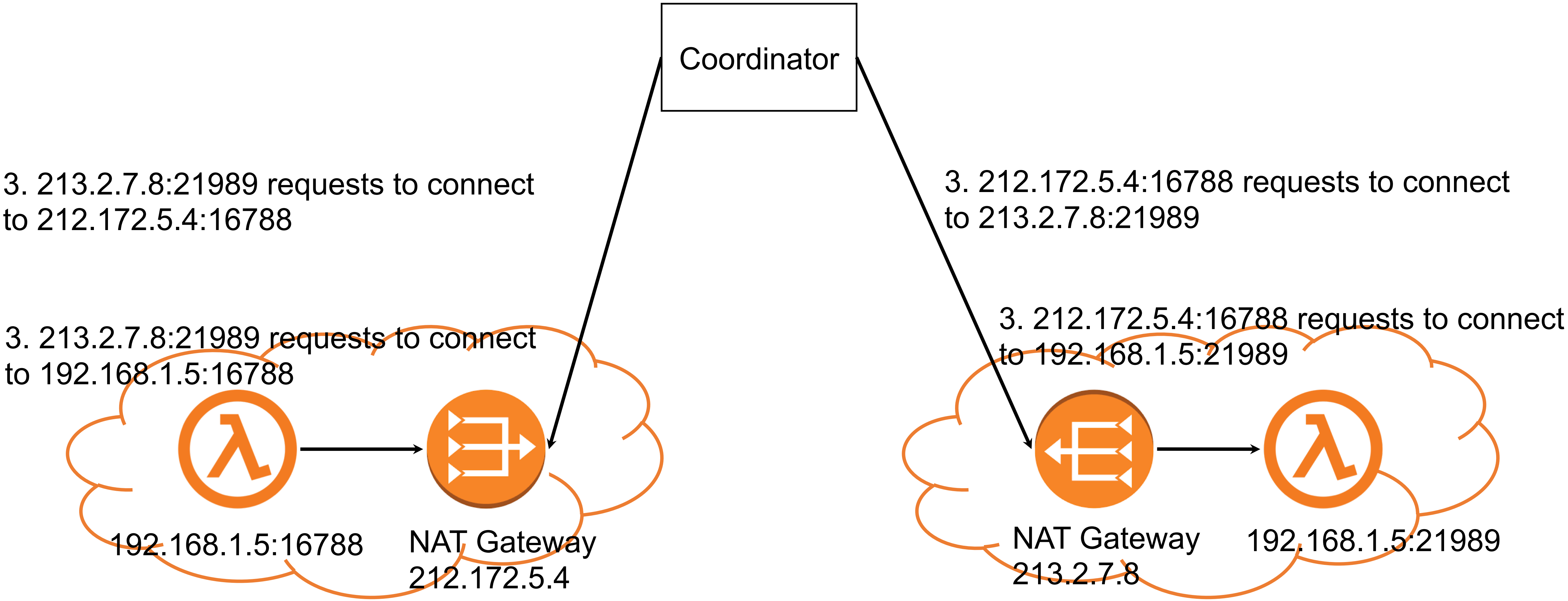
- Lambdas can connect with each other by leverage UDP hole punching
  - https://networkingclients.serverlesstech.net/getting_started.html



Client in the Lambda – Hole Punching



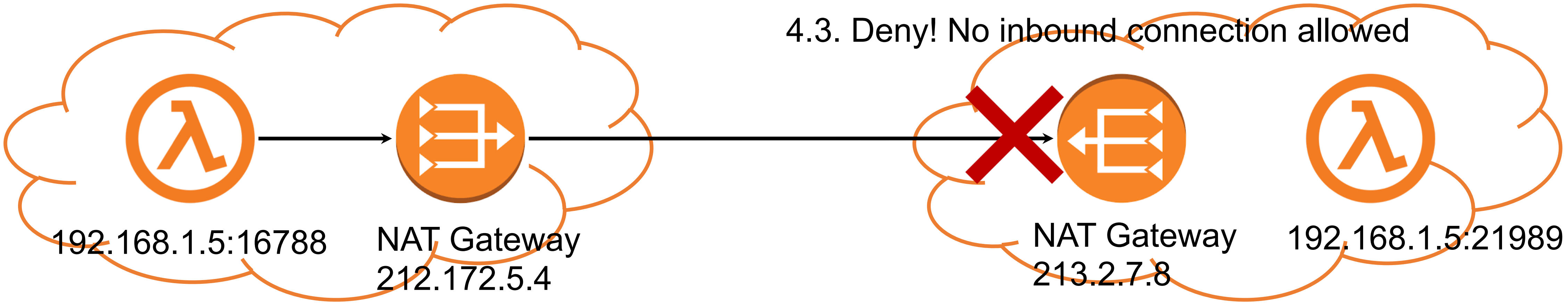
Client in the Lambda – Hole Punching



Client in the Lambda – Hole Punching

4. 192.168.1.5:16788 requests to connect to 213.2.7.8:21989

    4.1. 212.172.5.4:16788 requests to connect to 213.2.7.8:21989
    4.2. Waiting for acknowledgement from 213.2.7.8:21989



Client in the Lambda – Hole Punching

Coordinator

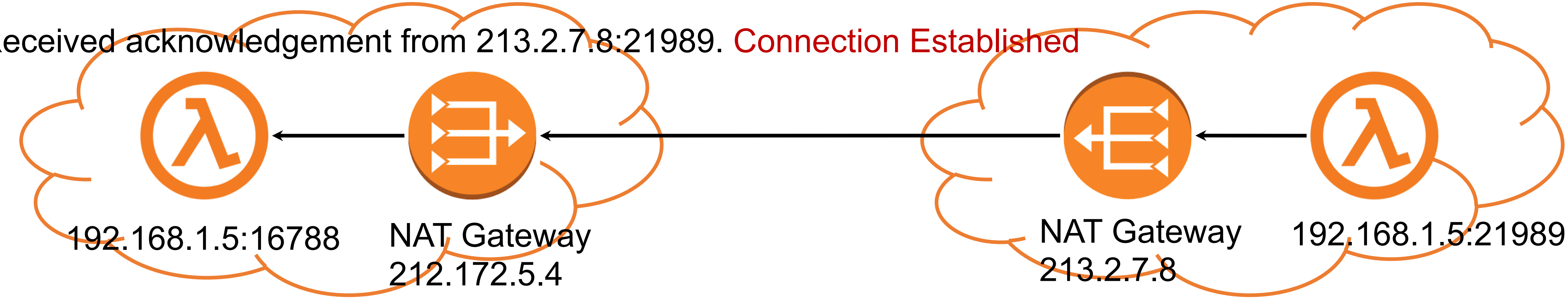
4.4. 192.168.1.5:21989 requests to connect to 212.172.5.4:16788

4.5. 213.2.7.8:21989 requests to connect to 212.172.5.4:16788

4.6. Waiting for acknowledgement from 212.172.5.4:16788

4.7. Received acknowledgement from 213.2.7.8:21989. Pass!

4.8. Received acknowledgement from 213.2.7.8:21989. Connection Established



Client in the Lambda

- Idea
  - Using coordinator as the proxy

- Challenge?
  - Now the coordinator is another service, is the idea still cost effective?
  - How the proxy owning global meta information, so the proxy can schedule and balance the workload, given a client can connect to Lambda instances of the InfiniCache directly?

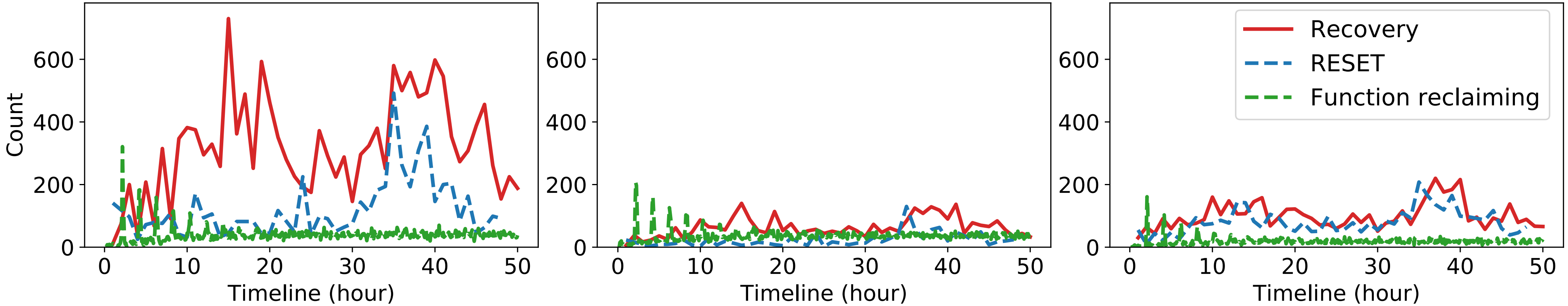
Client in the Lambda

- Possible solution
  - Clients make request to the proxy (control path), and accept data from Lambda instances of the InfiniCache directly (data path).
  - Since the proxy is not on data path, cheaper ec2 can be used to provide coordination, hence may justify the cost effectiveness.

Backup

Evaluation – Production Workloads

- Fault tolerance activities
  - Recovery: erasure-coding recovery
  - RESET: GET miss
  - Function reclaiming



All objects

Large objects only

Large objects only w/o backup

Evaluation

- Scalability