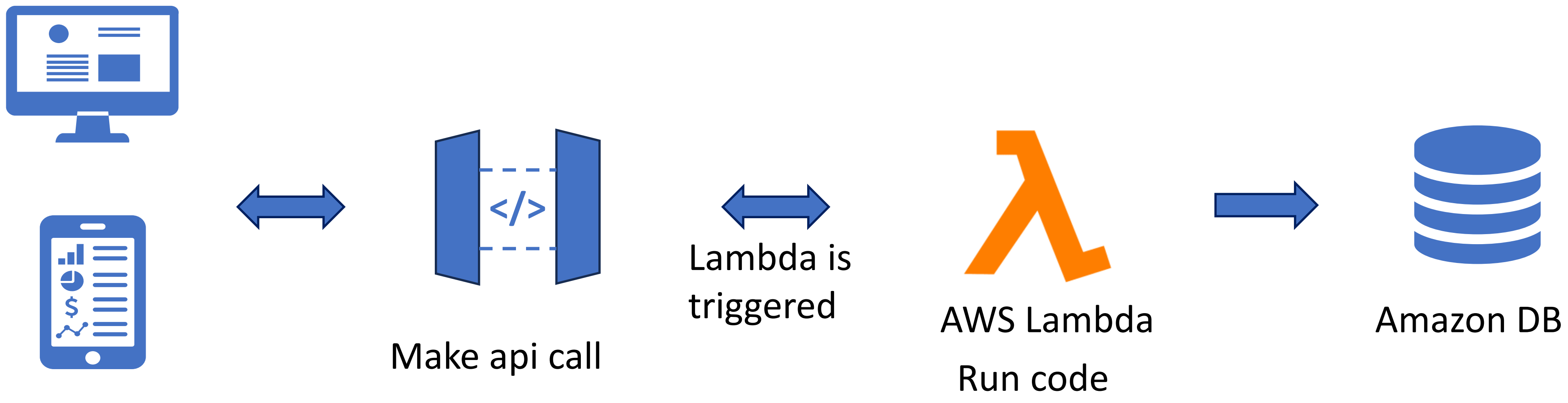
On-demand Container Loading in AWS Lambda

ATC' 23

Background: AWS Lambda

- Provide your code or image, we run it as an event when things happen
- No provisioning or managing servers
- Scale up in milliseconds in response to traffic

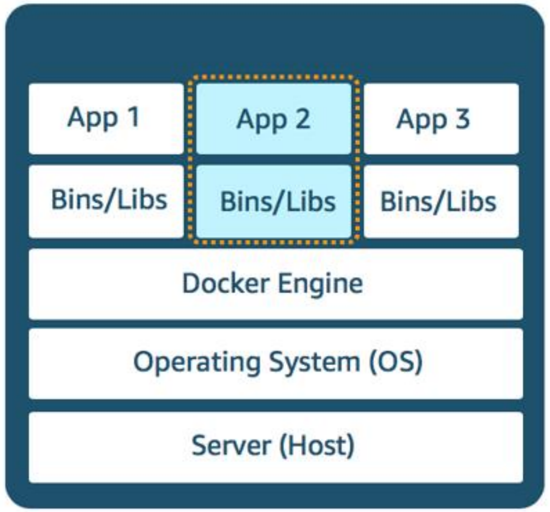


Background: AWS Lambda

- Container: an isolated environment for your code.

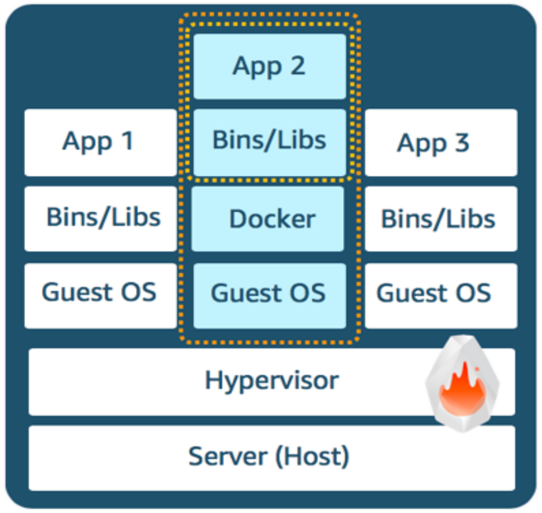
  sharing host operating system
- AWS Lambda: each container or code runs in one MicroVM

  customer code(250 MB) or container image (10 GB)



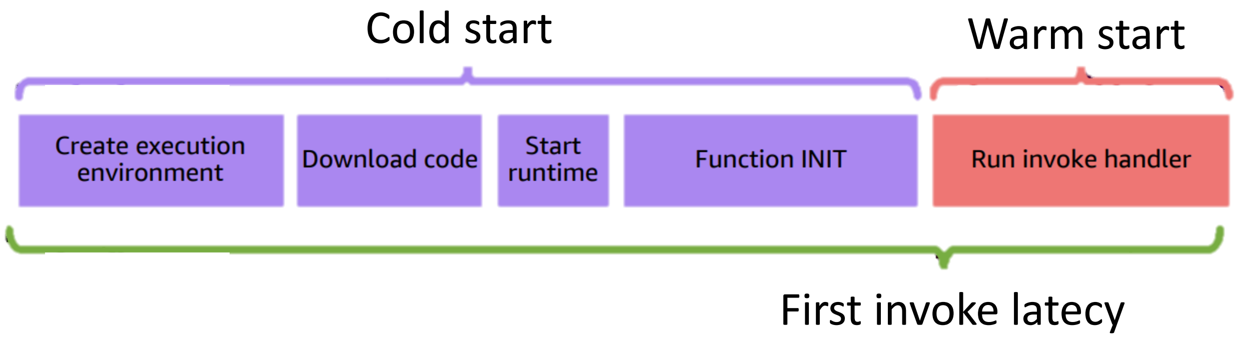
Docker Container

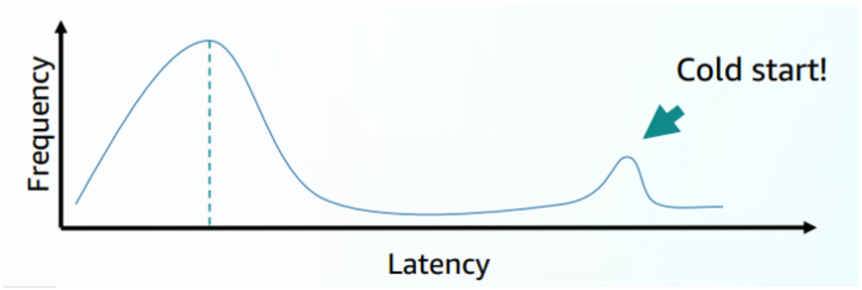
AWS Lambda



Problem

- Adding container support to AWS Lambda without regressing on cold-start time
  - Meeting Lambda's goals of rapid scale, high request rate and low start-uptimes
  - The core challenge is simply one of data movement.





Main idea

- **Sparsity** —block-level demand loading

  Most container images contain a lot of files, but only 6.4% of container data is needed at startup.

- **Commonality** —deduplication

  Many popular container images are based on common base layers

- **Cacheability** —Tiered Caching

  Most of workloads tend to be driven by a smaller number of images

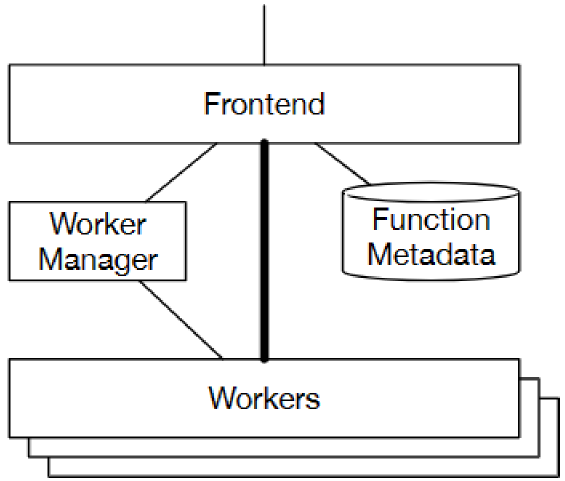
Architecture

- **Worker Manager**: Assignment Service

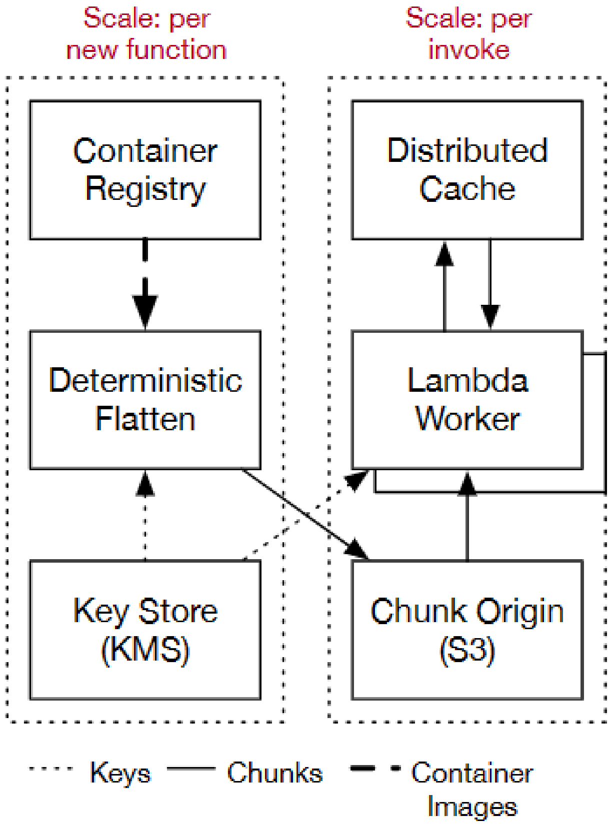
  forward the request to a worker or start a new worker

- **Worker**:

  Lots of independent isolated environments to run customer code



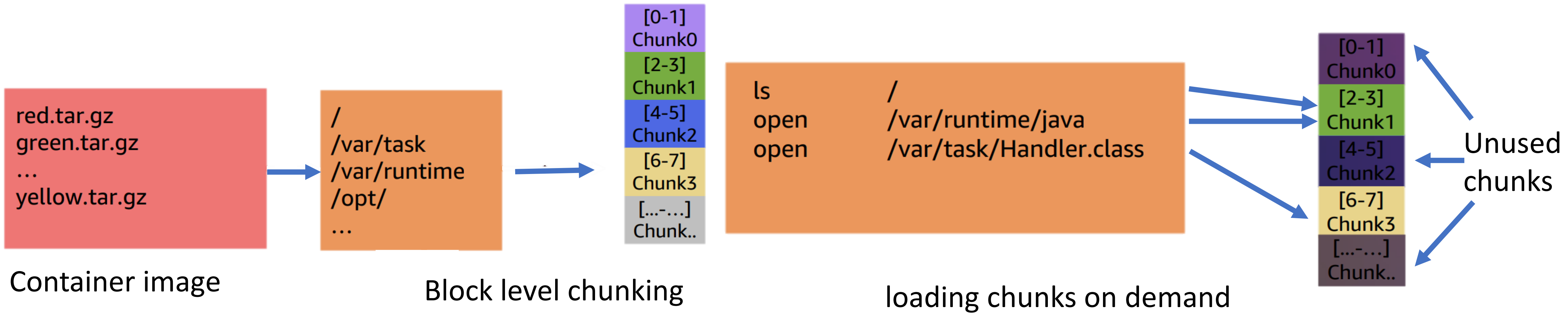
Invoke path



High-level system architecture

Design1: Block-Level Loading

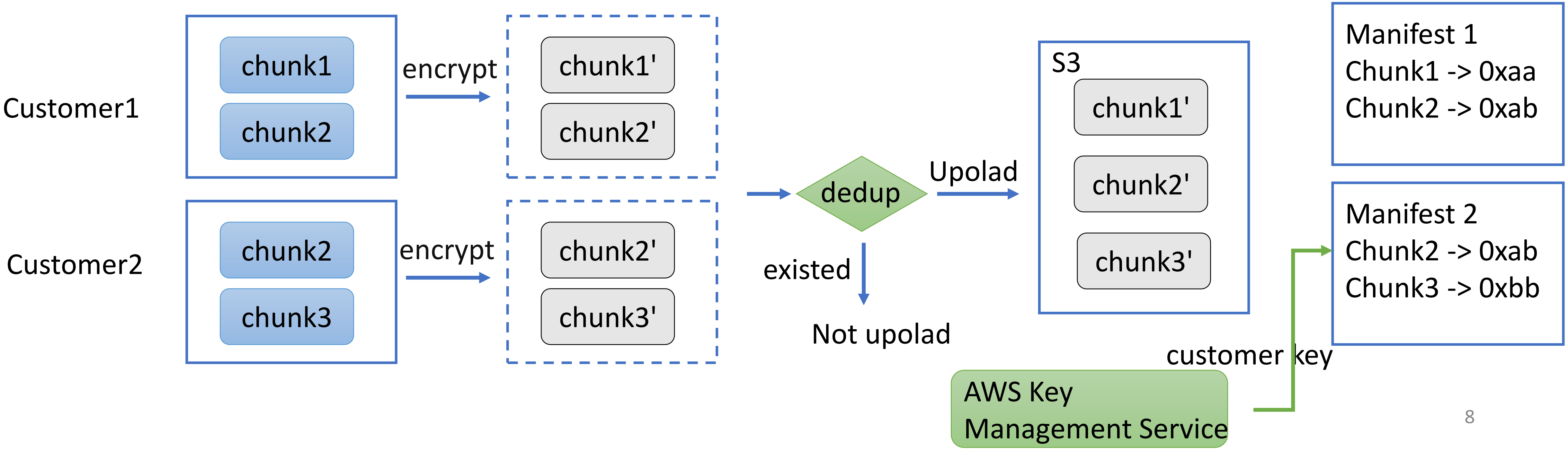
- Collapse the container image into a block device image
  - flattening each tarball to create a single ext4 filesystem
  - overlay a stack of layers using overlayfs.
- Build a filesystem that knows about our chunked container format
  - reads by fetching just the chunks of the container it needs



Design2: Deduplication

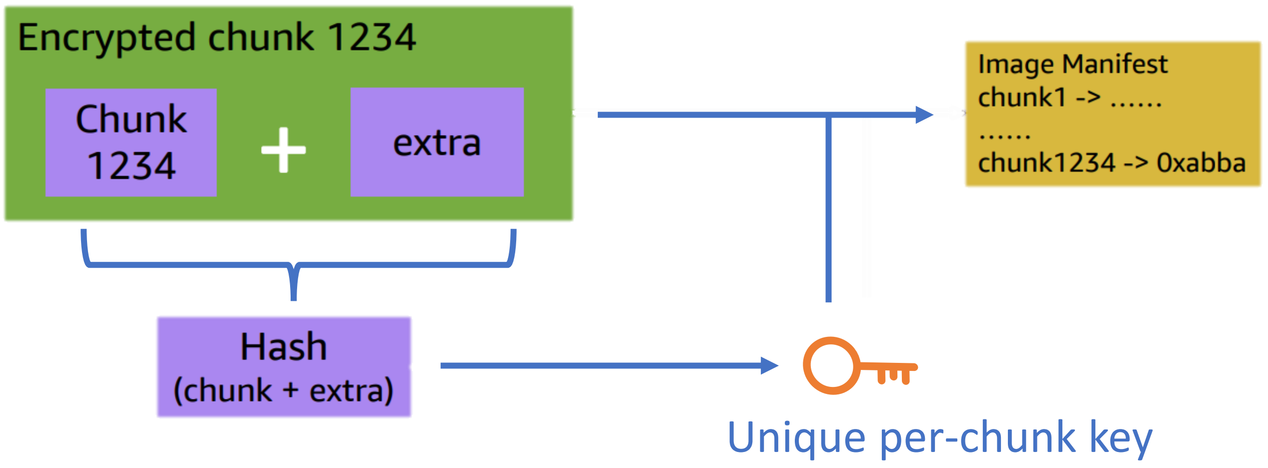
- Deduplication-after-encryption.
- Each Lambda worker host to only being able to access the data that have been sent to it.

Do not encrypt the entire manifest. Only the chunk key table is encrypted.

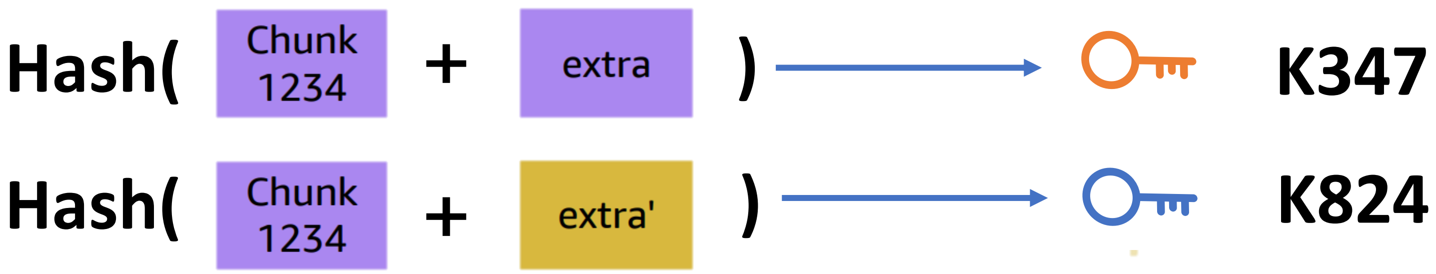


Design2: Convergent encryption

- The same chunk is encrypted by same key.

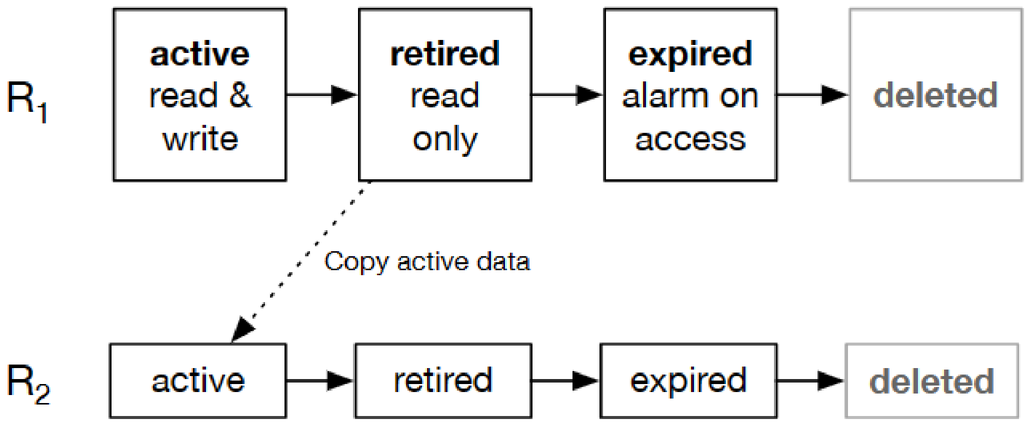


- Using varying salt in the key derivation step to limit Blast Radius.



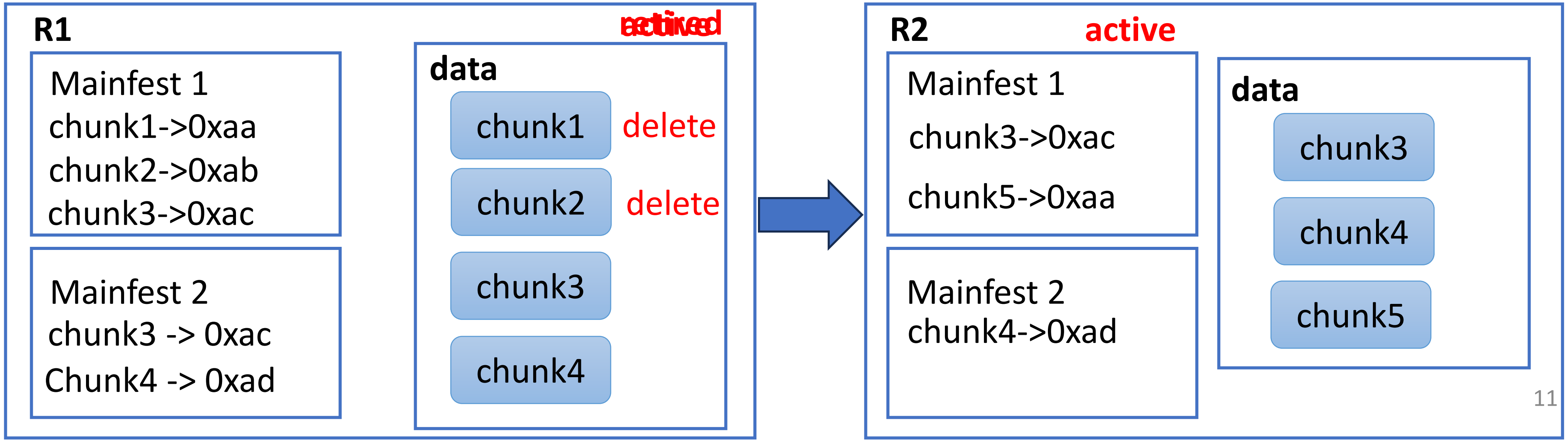
Design2: Garbage Collection

- Removing data from the backing store when it is no longer actively referenced.
- Root: a self-contained manifest and chunk namespace
  - While R1 is retired, any manifest that is still referenced in R1 is migrated to R2.
  - In expired state, data is still allowed to be read, but any attempt to access data leads to an alarm.



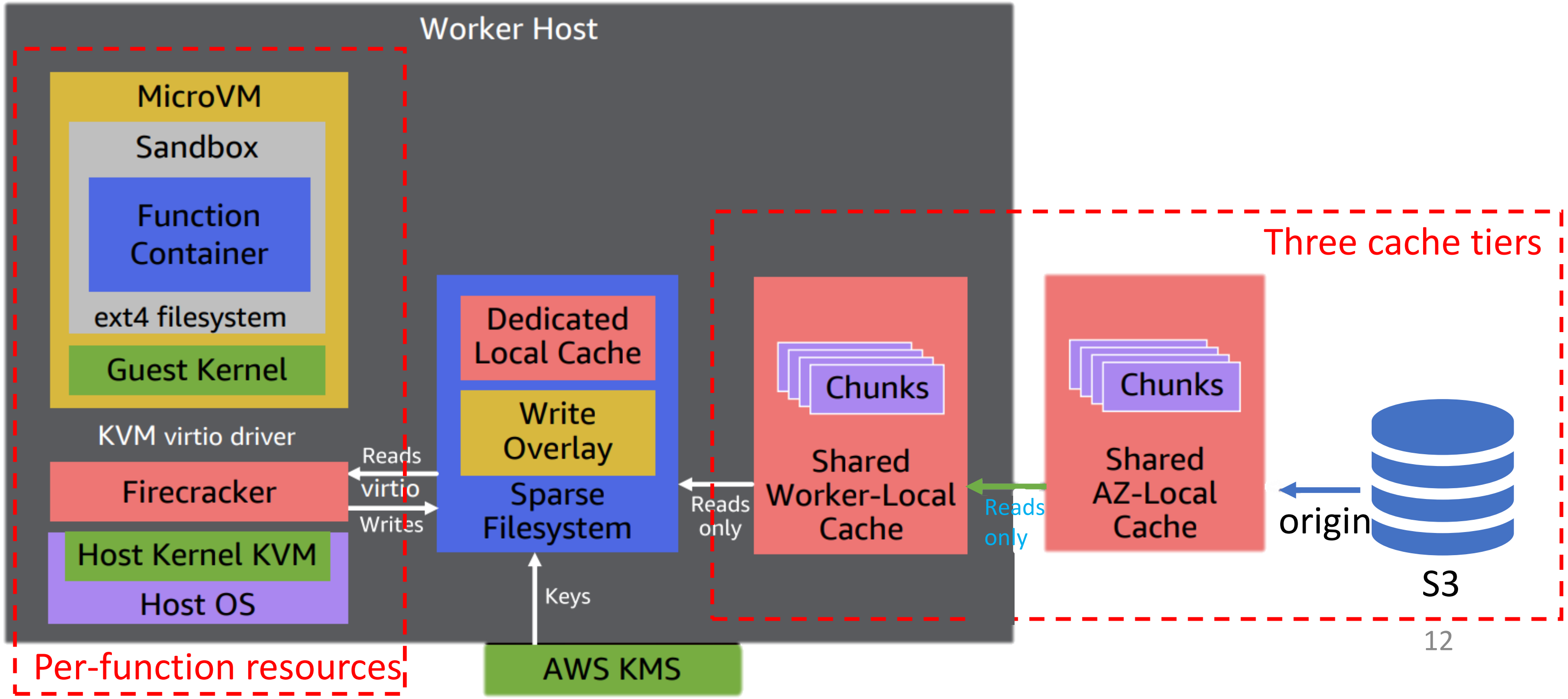
Design2: Garbage Collection

- Removing data from the backing store when it is no longer actively referenced.
- Root: a self-contained manifest and chunk namespace
  - While R1 is retired, any manifest that is still referenced in R1 is migrated to R2.
  - In expired state, data is still allowed to be read, but any attempt to access data leads to an alarm.



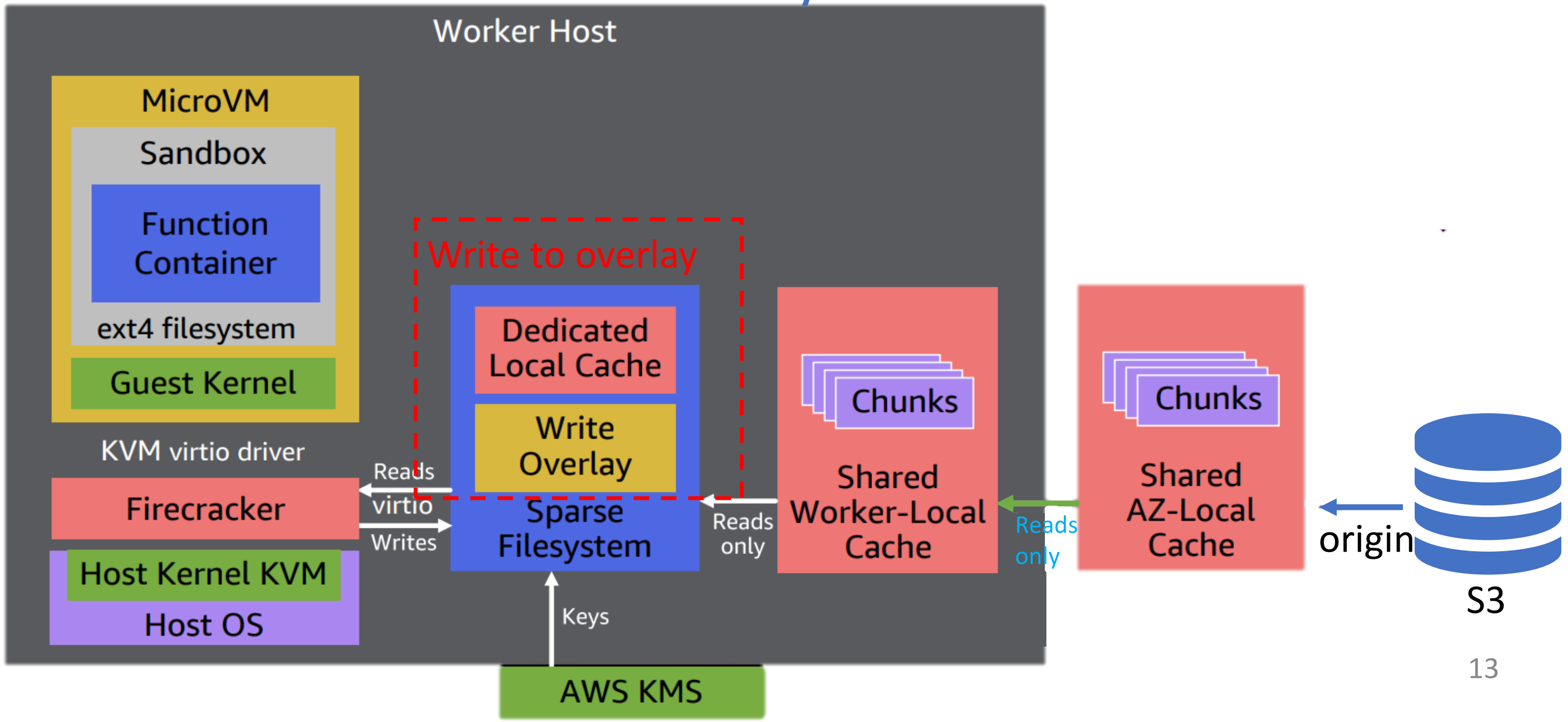
Design3: Tiered Caching

- Three cache tiers
  - S3 cache: origin tier that stored all chunks.
  - Worker Local Cache: caches chunks that are frequently used on a worker.
  - AZ-level cache : caches chunks that are frequently used on workers in availability-zone.



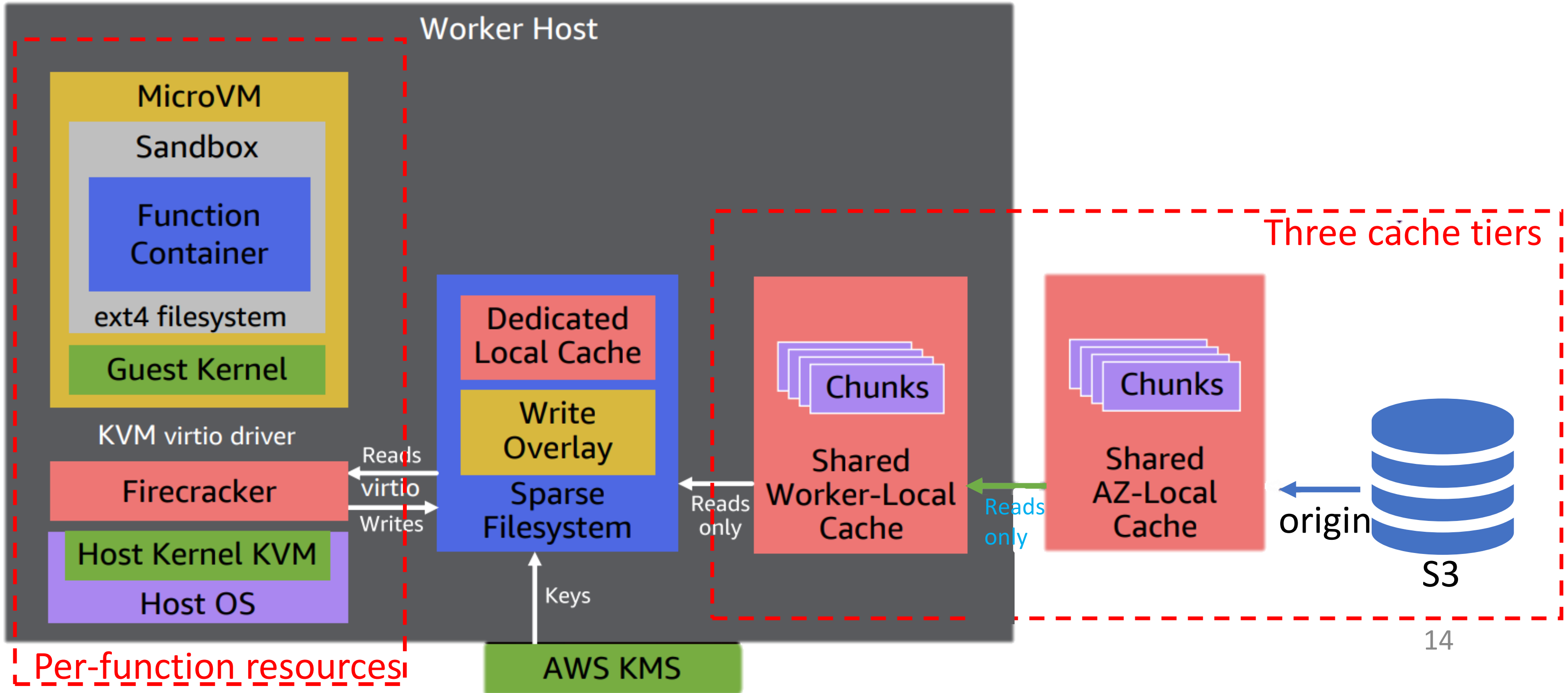
Design3: Tiered Caching

- Read chunk data: reading directly from the local cache firstly
  - If not exists in local cache, the chunk is fetched from the AZ-level cache.
- Write data to block overlay
  - Using a bitmap to check if chunk written to overlay



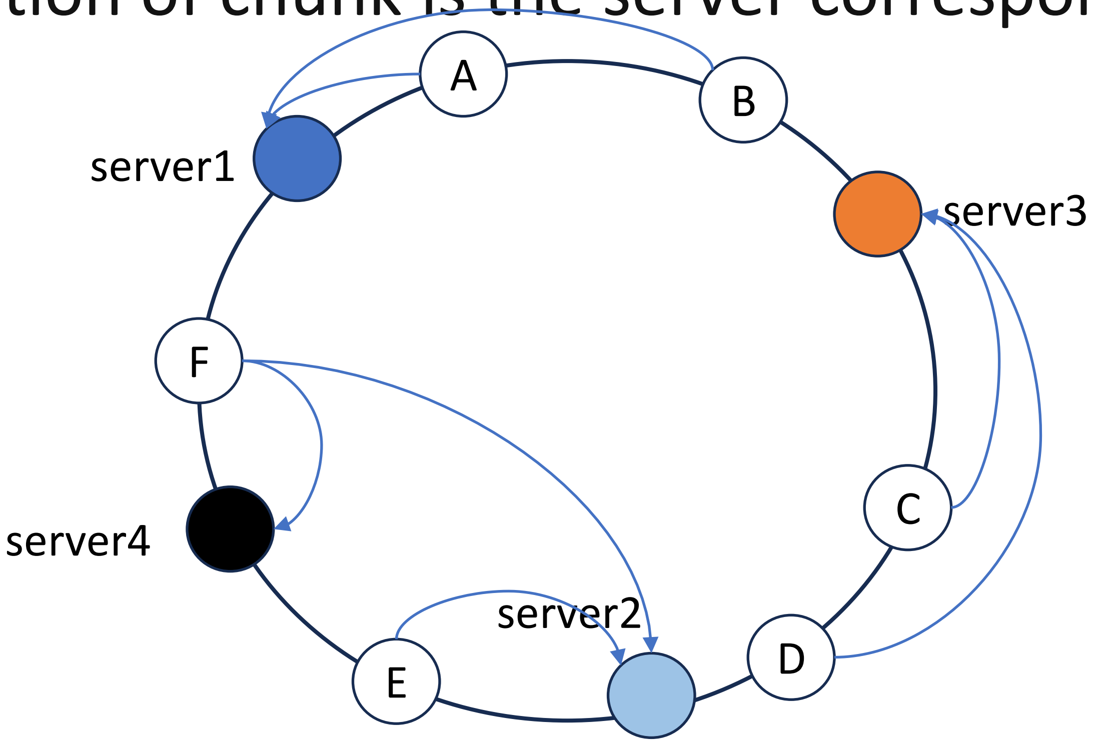
Design3: Tiered Caching

- AZ-level cache: a fairly standard design of distributed cache.
  - An in-memory tier for hot chunks and a flash tier for colder chunks.
  - Evictiontion is LRU-k -a scan-resistant LRU
  - Using a consistent hashing scheme to distribute chunks.
  - Erasure coding to down tail latency.



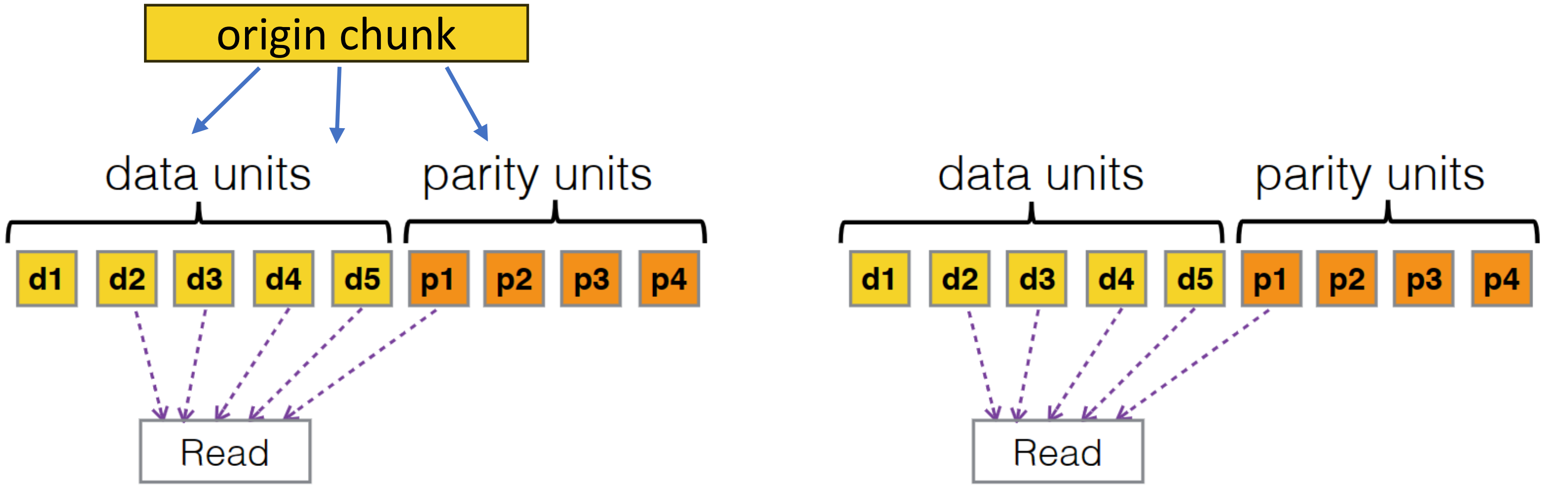
Design3: consistent hashing

- Map the chunk to the hash ring
- Map the server's id to the hash ring
- The first server encountered in a counterclockwise direction from the location of chunk is the server corresponding to the chunk.



Design3: Erasure coding

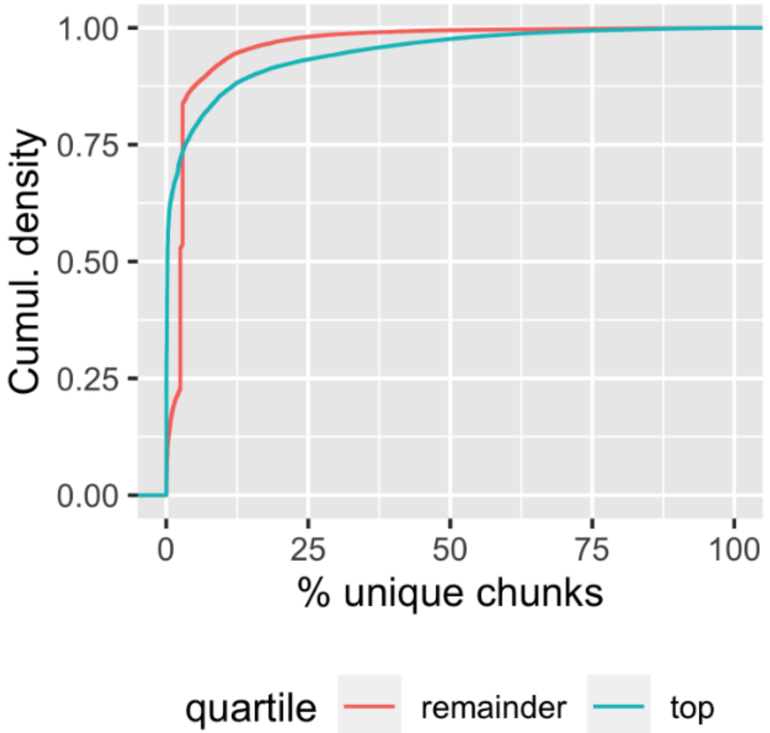
- A single slow cache server can cause wide spread impact because of congestion in the network, or by partial software failure.
- Erasure coding: Any *k* of the (k+r) units are sufficient to decode origin full data.



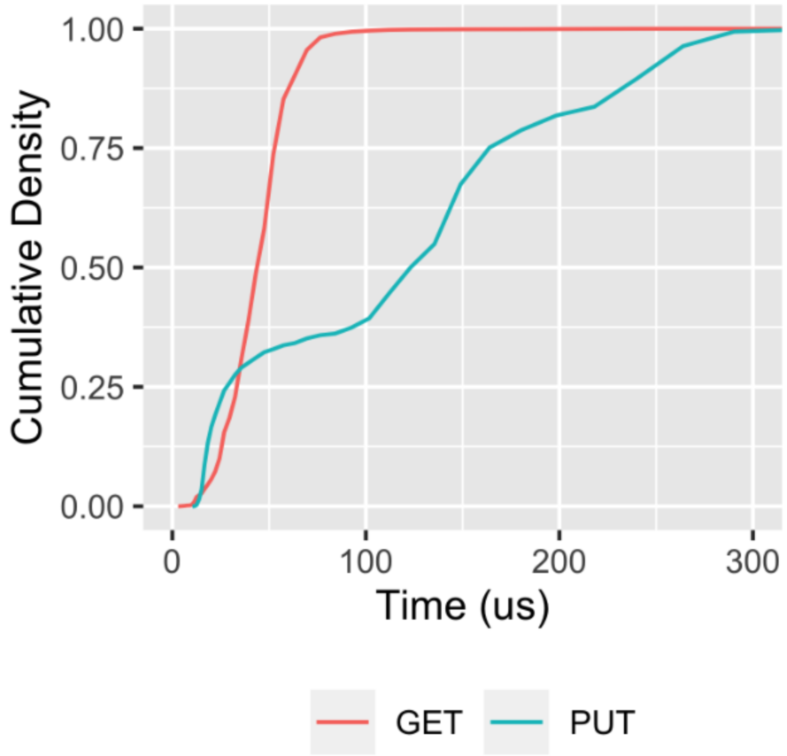
- k = 5
- r = 4

Theorem: k points can determine a curve corresponding to a polynomial of order k-1

Evaluation



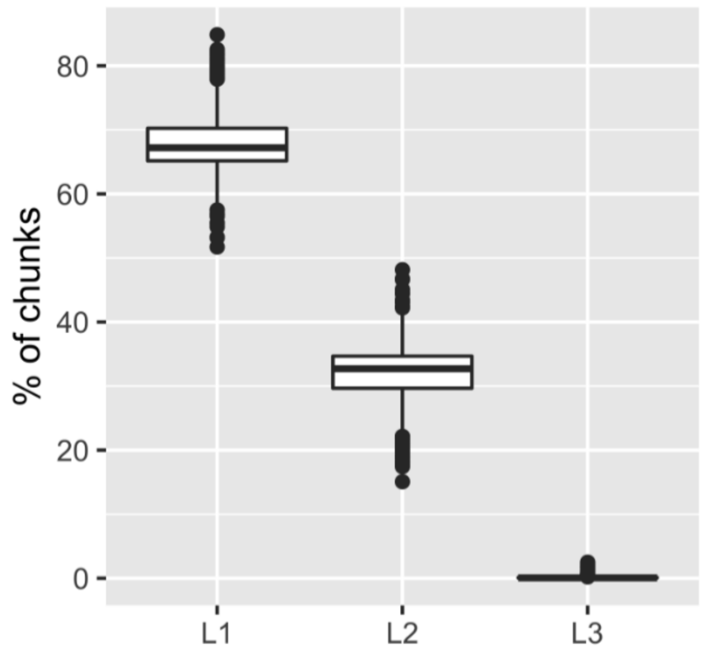
deduplication effectiveness



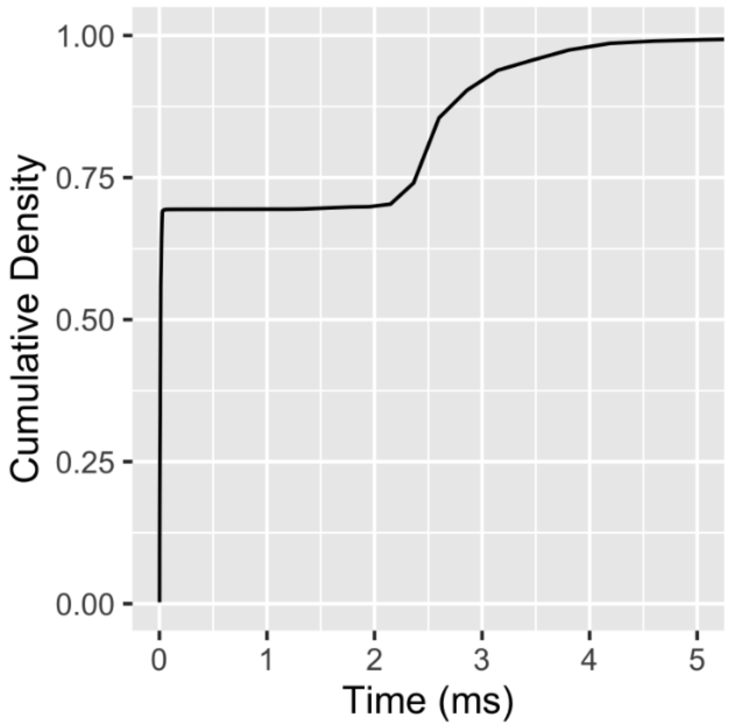
Server-side measured latency ofthe L2 cache server

- The majority of functions of all sizes are heavily deduped.
- GET latency is very consistent, with a median of below 50µs.
- PUT latency is less consistent than GET, but performance is still excellent

Evaluation



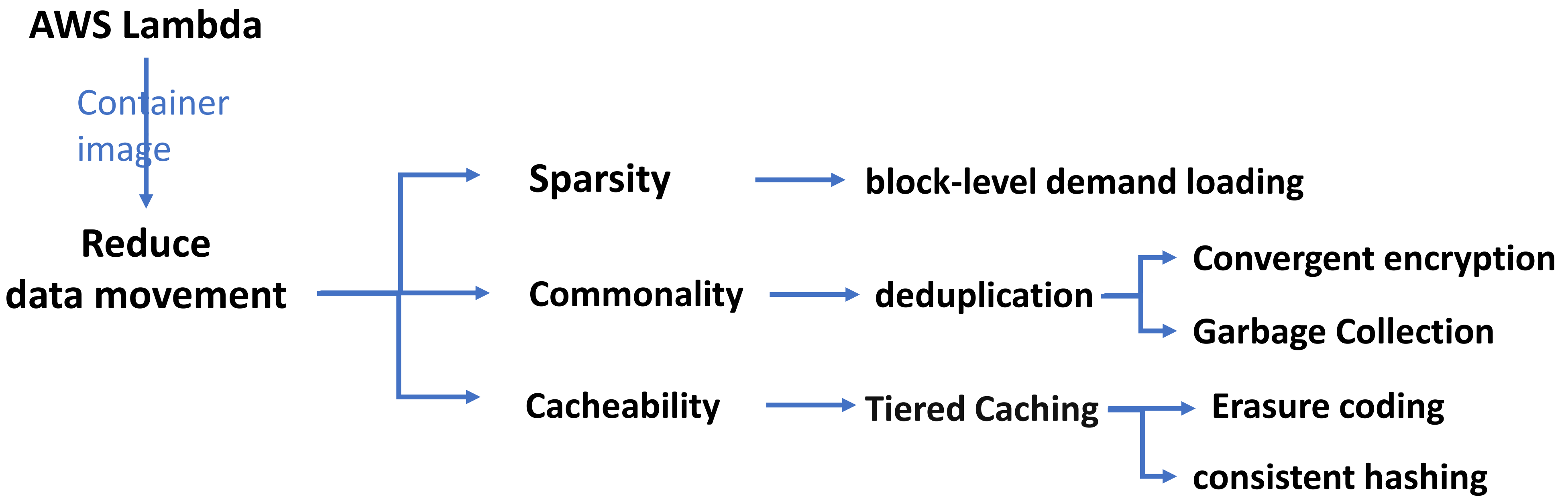
One week of hit rates on each of the cache tiers



Empirical CDF of end-to-end read latency observed at the local agent

- These three cache tiers are efficient
- A mode below 100µs which represent local cache hits, a mode around 2.75ms which represent L2 hits

Summary


AWS Lambda
Container image
Reduce data movement
Sparsity
block-level demand loading
Commonality
deduplication
Convergent encryption
Garbage Collection
Cacheability
Tiered Caching
Erasure coding
consistent hashing