Cache Me If You Can

Decentralize Your Distributed Caches With Hollow

Viswanathan Ranganathan (Vish)

What Made Me Think Of This Talk?

Account Metadata

A lookup given an account id.

Don’t make the accounts service your bottleneck.

What About Redis?

A new cluster to operationalize, manage, and maintain.

The Obvious Plan

Preload at startup. Fetch on miss.

Simple enough in theory.

Two Code Paths. One Problem. Forever.

Bootstrap ↔ Runtime. Same data. Two mechanisms.

Viswanathan (Vish) Ranganathan

Senior Engineer · Netflix Delivery

COME TALK TO ME ABOUT

Java · Scala · Functional Programming · Distributed Systems · Streaming Systems · Data Analytics · Open Source

Why The Obvious Approach Doesn’t Hold Up

Problem A

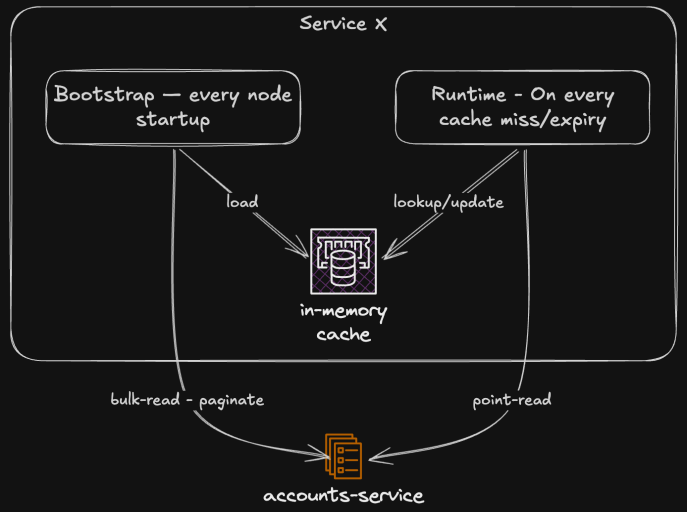
Two code paths, one problem

Preload vs. Refresh

At bootstrap: paginate through accounts list

At runtime: fetch individual entries from the API

Two systems to test, operate, and debug. **Forever.**



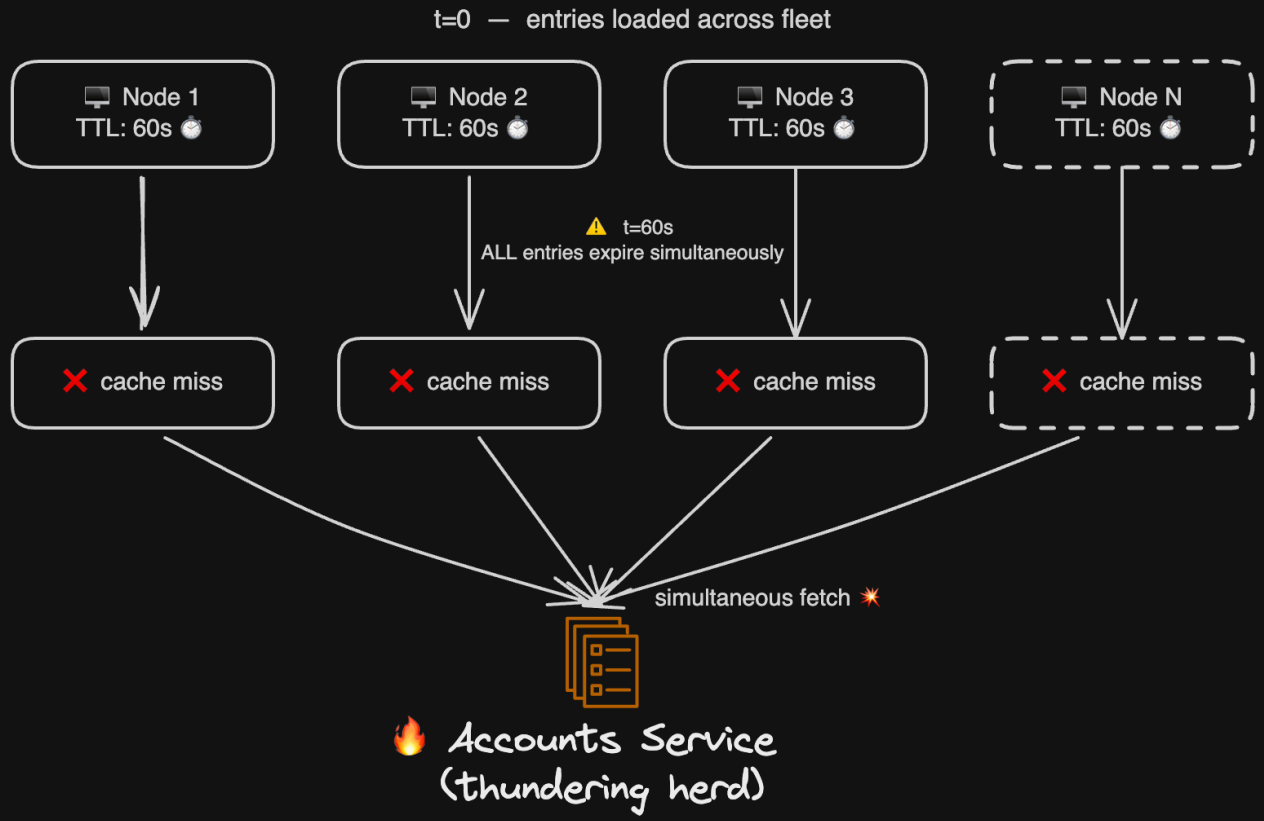
Problem B

The runtime fetch still creates a thundering herd

Individual entries expire. One node fetches. Fine.

But entries loaded at the same time expire at the same time — **across every node, simultaneously.**

Cache Stampede



Patches on a Broken Model

The model is still broken.

One Producer. N Consumers.

Push, not poll.

Consumers never touch the source again. No pagination. No expiry timers. No stampede.

This Is Just Git

Clone once. Pull only the diff.

Producer = the remote

Consumer = your local repo

Hollow: A Data Distribution System

github.com/Netflix/hollow

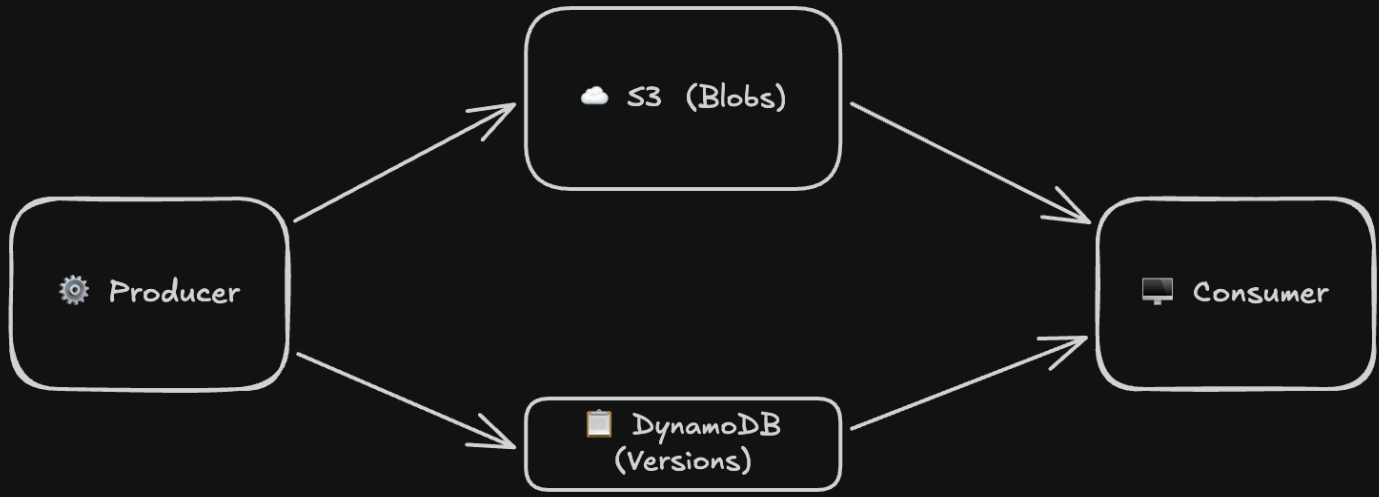
Four Interfaces. That’s the Contract.

Publisher — writes snapshots + deltas to blob storage

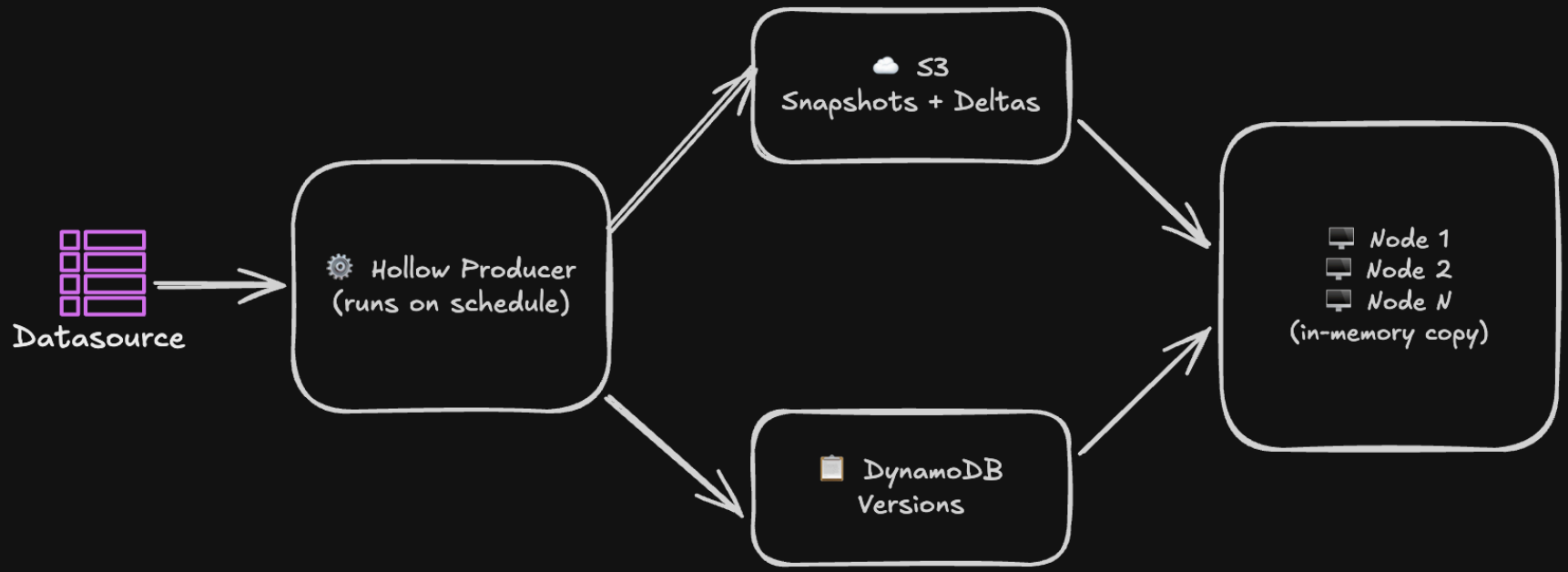
Announcer — writes the new version to a coordination store

BlobRetriever — reads blobs from storage on the consumer side

AnnouncementWatcher — polls the coordination store for new versions



The Architecture



The Example

Two independent datasets. One Spring Boot service.

Dataset	Records	Update cadence
Catalog	50,000 movies	every 7 min
Users	350,000 profiles	every 10 min

One query endpoint: `` `GET /users/{userId}/recently-watched` ``

Define Your Schema

```
// 50,000 records · updates every 7 min
@HollowPrimaryKey(fields = ["id"])
data class Movie(
    val id: Long, val title: String, val genre: String,
    val rating: Double, val releaseYear: Int, val duration: Int,
    val director: String, val cast: List<String>,
    val tags: List<String>, // ...18 fields total
)

// 350,000 profiles · updates every 10 min
@HollowPrimaryKey(fields = ["userId"])
data class User(
    val userId: String, val firstName: String, val lastName: String,
    val planName: String,
    val recentlyWatchedMovieIds: List<Long>
)
```

Generate the Client

```
task generateHollowAPI(type: JavaExec) {
    group = 'hollow'
    mainClass = 'org.vish.hollowdemo.codegen.GenerateMovieAPI'
    classpath = sourceSets.main.runtimeClasspath
    dependsOn classes
}
task generateUserAPI(type: JavaExec) {
    group = 'hollow'
    mainClass = 'org.vish.hollowdemo.codegen.GenerateUserAPI'
    classpath = sourceSets.main.runtimeClasspath
    dependsOn classes
}
task generateAllAPIs {
    group = 'hollow'
    dependsOn generateHollowAPI, generateUserAPI
}
```

```
./gradlew generateAllAPIs
```

```
→ MovieAPI.java
→ UserAPI.java
```

```
public class GenerateMovieAPI {
  public static void main(String[] args)
      throws IOException {
    HollowWriteStateEngine writeEngine =
        new HollowWriteStateEngine();
    HollowObjectMapper mapper =
        new HollowObjectMapper(writeEngine);

    mapper.initializeTypeState(Movie.class);

    HollowAPIGenerator generator =
        new HollowAPIGenerator.Builder()
            .withAPIClassname("MovieAPI")
            .withPackageName("org.vish.hollowdemo.api")
            .withDataModel(writeEngine)
            .build();

    generator.generateFiles("src/main/java");
  }
}
```

Wire the Producer

```
val localDir = Paths.get("data/catalog")

val producer = HollowProducer
    .withPublisher(HollowFilesystemPublisher(localDir))
    .withAnnouncer(HollowFilesystemAnnouncer(localDir))
    .build()
```

Produce a Cycle

```
producer.runCycle { writeState ->
    currentMovies.forEach { movie -> writeState.add(movie) }
    // Hollow computes: snapshot · delta · reverse-delta
}
```

What Gets Published

```
HollowS3Publisher  Publishing SNAPSHOT  → catalog/snapshot-20260519202727001
HollowS3Publisher  Publishing DELTA     → catalog/delta-...-20260519203437003
HollowS3Publisher  Publishing REV-DELTA → catalog/reversedelta-...
DynamoDBAnnouncer  Announcing version 20260519203437003 → hollow-dev-announcements
```

Cycle #2 · delta: 620 KB · snapshot: 10.4 MB · total cycle time: 3.4 s

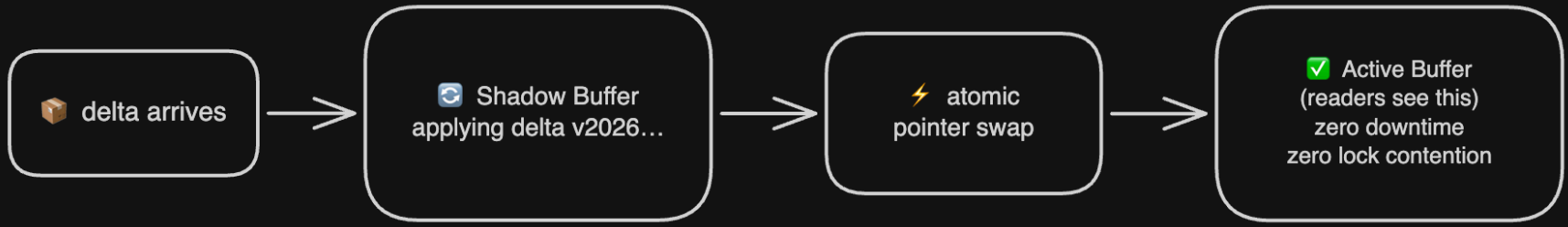
Wire the Consumer

```
val localDir = Paths.get("data/catalog")

val consumer = HollowConsumer
    .withBlobRetriever(HollowFilesystemBlobRetriever(localDir))
    .withAnnouncementWatcher(HollowFilesystemAnnouncementWatcher(localDir))
    .withGeneratedAPIClass(MovieAPI::class.java)
    .build()

consumer.triggerRefresh()  // reads snapshot from disk, blocks until ready
val movieIndex = Movie.uniqueIndex(consumer)
```

Double-Buffer Model



What You Get For Free

Type-safe generated APIs

Memory-optimized encoding

Consumer pinning

The Gap

You Want AWS. Hollow Ships Filesystem.

S3 for blobs. DynamoDB for announcements.

Four Interfaces to Implement

Not hard. **But not nothing.**

The Fork in the Road

Write it yourself. Or go back to Redis.

hollow-infra-adapters

Producer + Consumer Setup

Producer

```
HollowProducer producer = HollowProducer
  .withPublisher(
    HollowS3Publisher.create(config))
  .withAnnouncer(
    HollowDynamoDBAnnouncer.create(config))
  .build();

producer.runCycle(state ->
  state.add(myData));
```

Consumer

```
HollowConsumer consumer = HollowConsumer
  .withBlobRetriever(
    HollowS3BlobRetriever.create(config))
  .withAnnouncementWatcher(
    HollowDynamoDBAnnouncementWatcher
      .create(config))
  .build();
```

Config: 4 Environment Variables

```
HollowAwsConfig config = HollowAwsConfig.fromEnvironment();
```

Multiple Datasets: No Terraform Changes

Same bucket. Same table. Different `datasetId`.

What The Library Handles

You handle your data model.

The Log Walkthrough

Catalog Snapshot

```
13:27:27  CatalogProducer     Generating initial catalog (seed=99)...
13:27:27  CatalogProducer     Catalog prepared: 50,000 movies
13:27:28  HollowS3Publisher   Publishing SNAPSHOT → catalog/snapshot-20260519202727001
13:27:30  DynamoDBAnnouncer   Announcing version 20260519202727001
13:27:30  CatalogProducer     Cycle #1 completed in 3,370ms
```

3.4s

50,000 movies · snapshot: 10.4 MB

Users Snapshot

```
13:27:32  UserProducer       Generating 350,000 users (seed=42)...
13:27:33  UserProducer       User generation complete: 350,000 users in 1,948ms
13:27:34  HollowS3Publisher  Publishing SNAPSHOT → users/snapshot-20260519202732002
13:27:36  DynamoDBAnnouncer  Announcing version 20260519202732002
13:27:37  UserProducer       Cycle #1 completed in 4,605ms
```

4.6s

350,000 profiles · snapshot: 24.7 MB

Consumer Cold Start

```
13:27:42  CatalogConsumer   🔄 Catalog update started: v-MAX → v20260519202727001
13:27:43  CatalogConsumer   ✅ Catalog updated (update #1) · O(1) index ready
13:27:43  UserConsumer      🔄 User dataset update started: v-MAX → v20260519202732002
13:27:45  UserConsumer      ✅ User dataset updated (update #1) · O(1) index ready
13:27:46  Spring Boot       Started HollowdemoApplicationKt in 5.226 seconds
```

5.2s

cold boot · 50K movies + 350K users · both indexed

Delta Cycle

```
13:34:44  CatalogConsumer     🔄 Catalog update started: v20260519202727001 → v20260519203437003
13:34:45  HollowClientUpdater update plan: {DELTA to 20260519203437003}
13:34:45  HollowBlobReader    DELTA COMPLETED IN 28ms
13:34:45  CatalogConsumer     ✅ Catalog updated: v...202727001 → v...203437003 (update #2)
```

28ms

delta · 620 KB · background thread · read path never blocked

The Cross-Dataset Join

`GET /users/{userId}/recently-watched`

1. Look up user in `users` dataset → **O(1), local memory**
2. For each watched ID, look up `catalog` → **O(1), local memory**
3. Return enriched results → **no DB · no API · no network**

When To Use It

Use Hollow When

Read-heavy. Millions of reads, rare writes.

Bounded. Fits in memory.

Eventually consistent is fine.

Skip Hollow When

Sub-second consistency required.

Dataset is unbounded or user-generated.

Random write/read patterns — **use a database.**

Get Started

`github.com/vichu/hollow-infra-adapters`

Come find me after.

Thank You

Every deployment should be boring.

Vish Ranganathan

`github.com/vichu/hollow-infra-adapters`