



# Architecting backends to serve millions of RPS

Conor Gallagher



What problem is being solved and why?

# Problem Statement

Zalando adopted streaming architectures in the mid 2010s

Pushing Product data into an event bus did not make it easy to interact with for our various business units

Would it be possible to serve Product data centrally via API and make it so performant that the distributed Product datastores become redundant?

# Requirements

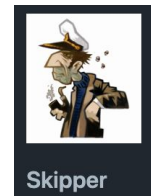
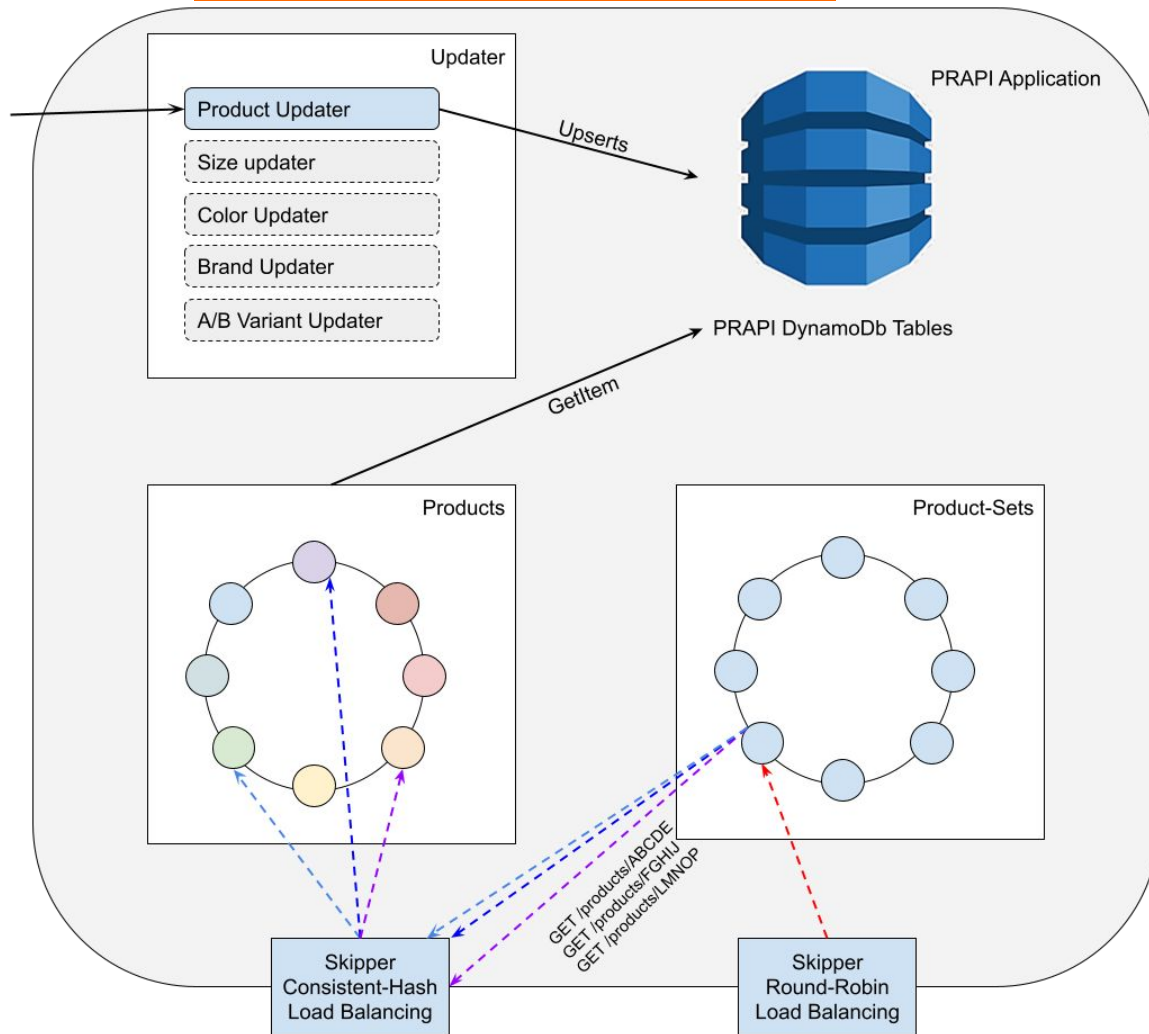
New Product Read API will be a tier one service serving Product data to Zalando's Fashion Stores and internal systems across Europe.

- Low Latency < 50ms p99 per single-get
- High Throughput: Millions requests per second
- High Availability
- Support for Batch Retrieval
- Handle Hot Products

Hot Product



# Product Read API (PRAPI)

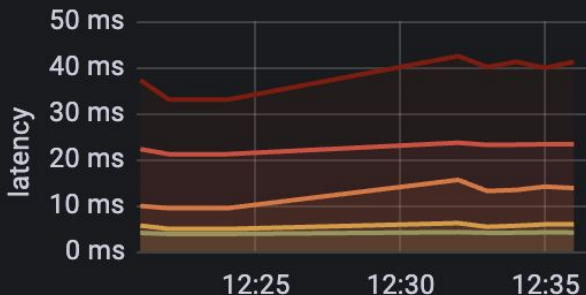


# Single GET Performance

Average ...

4.43

Response Latency: \* spp-product-read/api/products/{product\_id} ▾

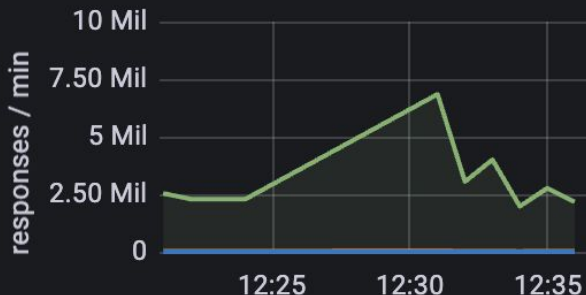


	max	avg	current
p50	4.16 ms	3.97 ms	4.09 ms
p75	6.12 ms	5.42 ms	5.85 ms
p95	15.5 ms	12.0 ms	13.7 ms
p99	23.6 ms	22.4 ms	23.2 ms

Last Succ...

98.9

Requests (per min): \* spp-product-read/api/products/{product\_id}



	min	max	avg	curr
2xx	1.98 Mil	6.84 Mil	3.02 Mil	2.1
3xx				
4xx	17.8 K	43.4 K	25.3 K	2
5xx	0	0	0	

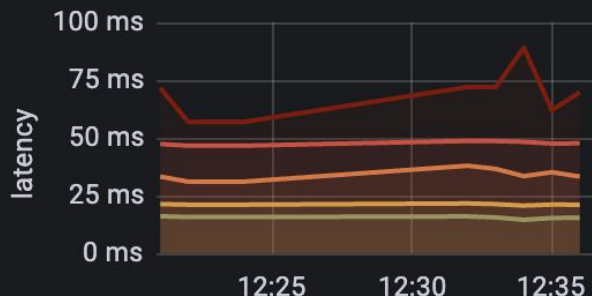


# Batch GET Performance

Average ...

14.9

Response Latency: \* spp-product-read/api/product-sets

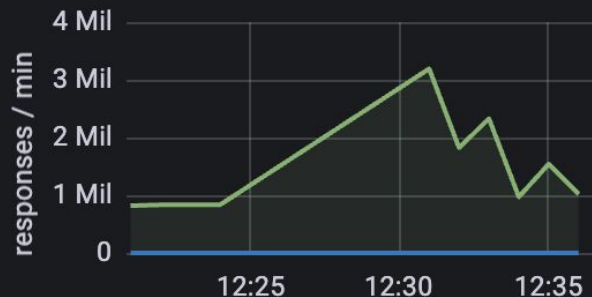


	max	avg	current
p50	16.0 ms	15.5 ms	15.4 ms
p75	21.6 ms	21.1 ms	21.0 ms
p95	37.9 ms	33.6 ms	33.2 ms
p99	48.7 ms	47.5 ms	47.6 ms

Last Succ...

100

Requests (per min): \* spp-product-read/api/product-sets



	min	max	avg	current
2xx	819 K	3.19 Mil	1.42 Mil	1.02 M
3xx				
4xx				
5xx				



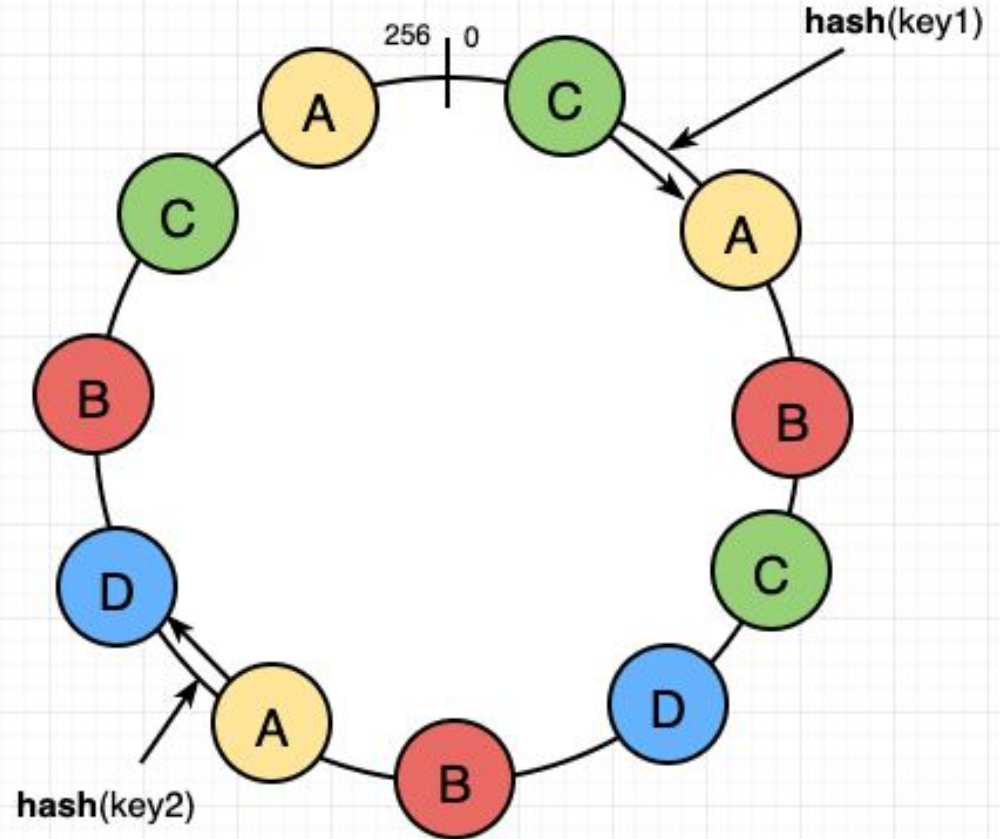
How do we achieve this performance?



# Load Balancing

Hash some part of the incoming request to determine its location on the ring.

LB will always send traffic to the closest POD to the right on the ring.



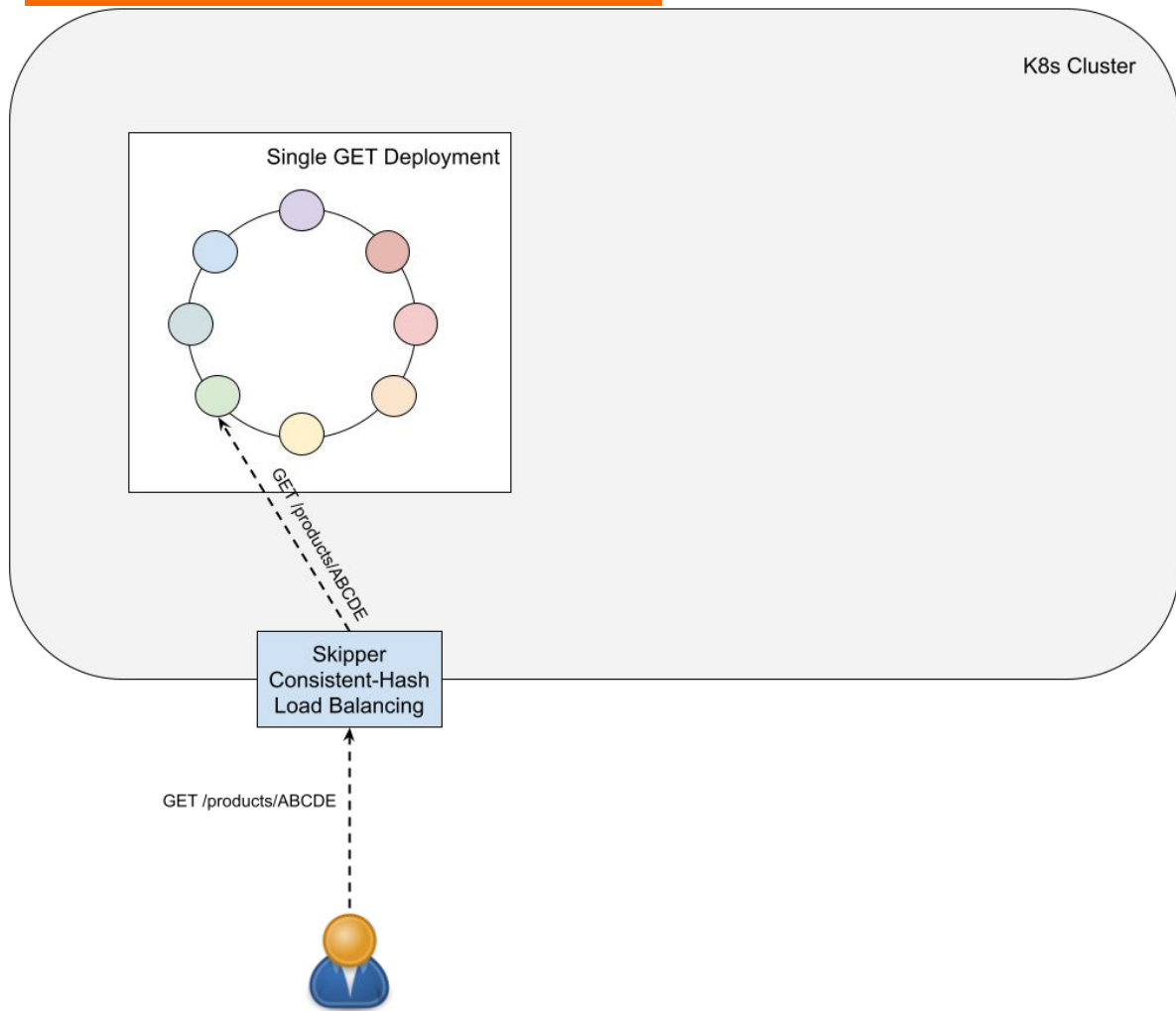
# Consistent Hash Load Balancing for Products

Use the product-id from the request URL as input to the LB strategy to consistently route product requests to particular pods.

As Product Catalogs are only ever partially hot, a small bounded cache on each pod with a short TTL would have a huge impact. By hot, we mean popular products, think of your basic white t-shirts or new Nike shoes under campaign.

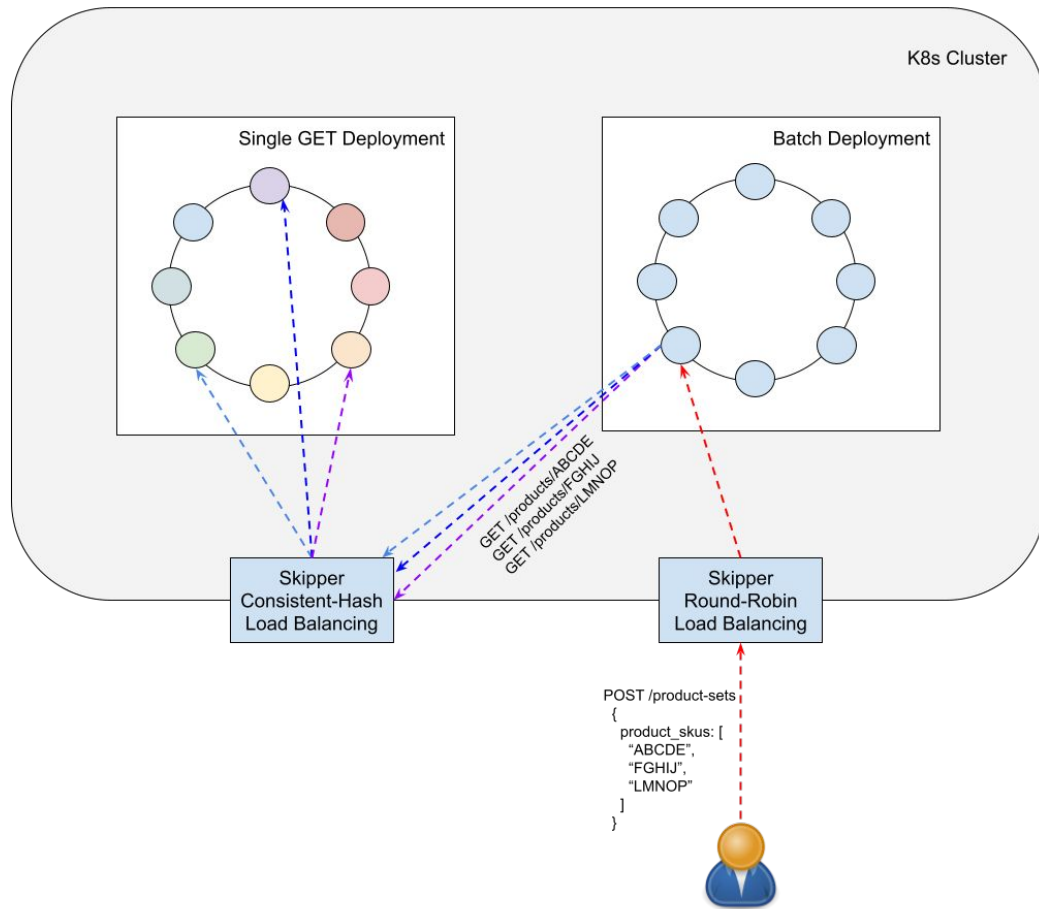
Extract the  
product-id from the  
path of the incoming  
request

Hash the product-id  
to determine its  
location on the ring  
and the POD that  
will get the traffic



Skipper is configured to round-robin batch requests across a dedicated Batch deployment.

This deployment makes N parallel consistently-routed requests to the Single Get deployment



# Consistent Hash Load Balancing for Products

Scaling Activities should not cause Mass Cache Invalidation

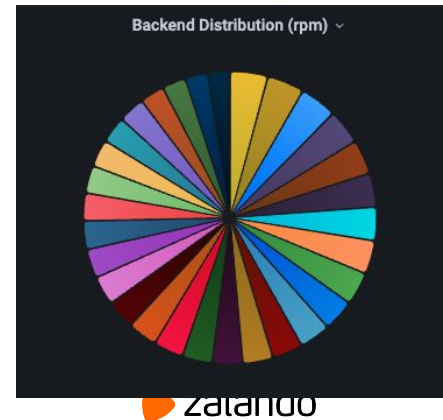
<https://github.com/zalando/skipper/issues/1712>

- Enter each pod at 100 random locations on the ring.
- Results in  $1/N$  cache invalidations, where  $N$  is the total number of pods


Avoid Overloading a single POD. Requests should spill-over consistently into neighbouring PODs:

<https://github.com/zalando/skipper/issues/1769>

- Introduce Bounded Load, with configurable loading factor
- A single pod can only ever serve  $N (1.5)$  times more requests than the average







# Async vs Non Blocking What's the difference?

# Non Blocking IO

```
val apacheHttpAsyncClient = HttpAsyncClientBuilder.create()
    .setDefaultIOReactorConfig(ioReactorConfig)
    .setThreadFactory(threadFactory)
    .setConnectionManager(connectionManager)
    .build()

val builder = ResteasyClientBuilderImpl()
    .httpEngine(ApacheHttpAsyncClient4Engine(apacheHttpAsyncClient, closeHttpClient: true))
```

## DynamoDB Client (Async NIO using Netty)

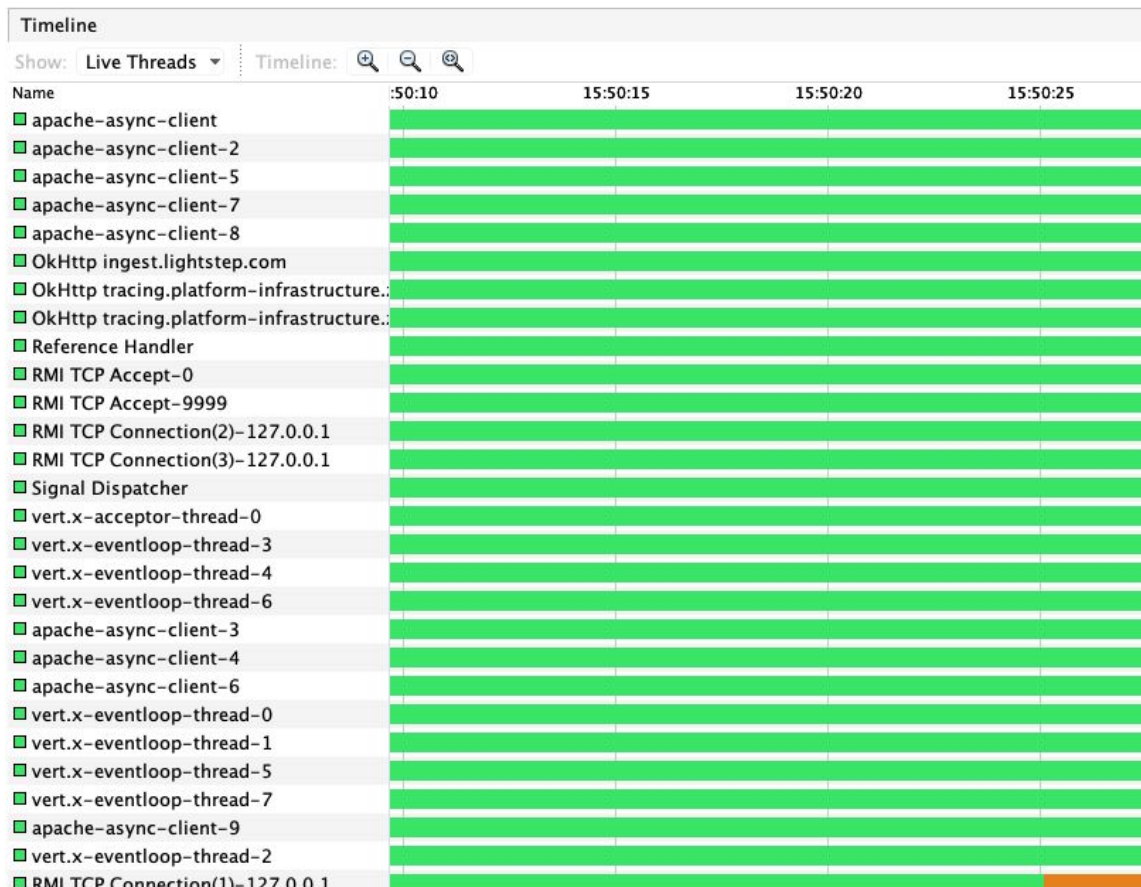
```
val nettyClientBuilder = NettyNioAsyncHttpClient.builder()
config.maxConcurrency().ifPresent { nettyClientBuilder.maxConcurrency(it) }
```

```
val clientBuilder = DynamoDbAsyncClient.builder()
    .httpClientBuilder(nettyClientBuilder)
    .asyncConfiguration { b ->
        b.advancedOption(FUTURE_COMPLETION_EXECUTOR, vertx.nettyEventLoopGroup())
    }
```

# NIO - Resource Utilisation Under Load

10,000 Outbound requests per second:

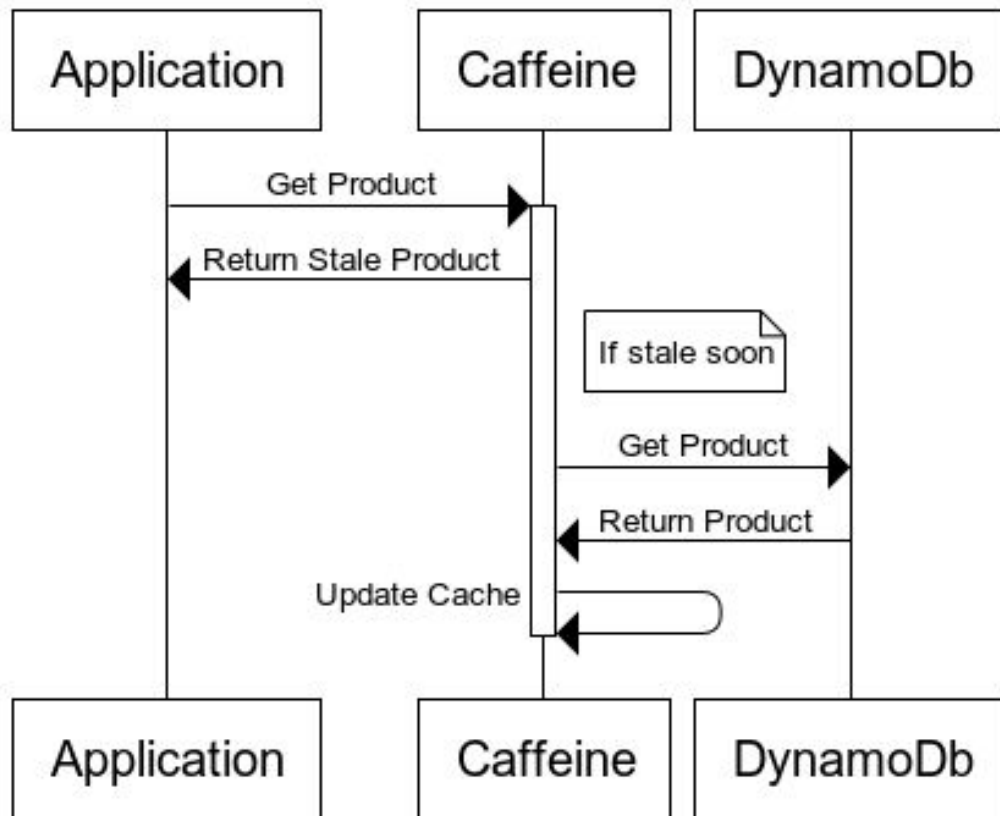
- 4 CPU cores request limit
- 16 Active Threads
- 20% CPU Utilization



# Caffeine Async Loading Cache

<https://github.com/ben-manes/caffeine>

## Product Read: Caffeine Async Loading Cache





# Garbage Collection (GC) Tuning

# GC Tuning - Before



# GC Tuning - Fix

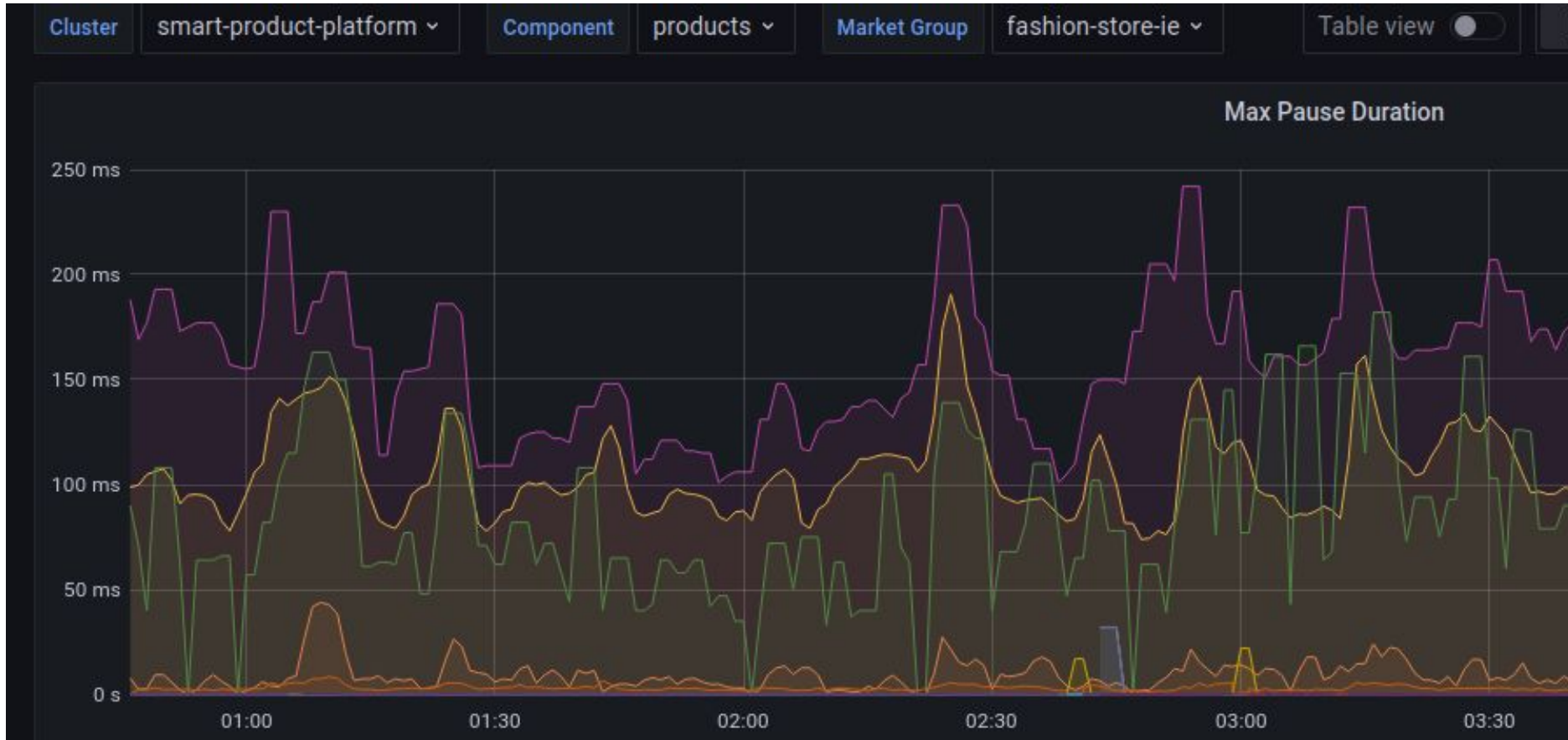
```
- import com.fasterxml.jackson.databind.node.ObjectNode
import com.github.benmanes.caffeine.cache.AsyncLoadingCache
import ie.zalando.spp.product.config.CacheConfig
import ie.zalando.spp.product.config.CaffeineConfig

@@ -18,9 +17,9 @@ class CacheStore(
    private val cacheBuilder: CacheBuilder,
    private val cacheConfig: CacheConfig,
) {
-     private val cacheStore = mutableMapOf<Cache, AsyncLoadingCache<String, Optional<ObjectNode>>>>()
+     private val cacheStore = mutableMapOf<Cache, AsyncLoadingCache<String, Optional<ByteArray>>>>()

-     private fun getCacheReloader(cache: Cache): (String, Span) -> CompletableFuture<Optional<ObjectNode>> {
+     private fun getCacheReloader(cache: Cache): (String, Span) -> CompletableFuture<Optional<ByteArray>> {
```



# GC Tuning - After





Questions?