Contents

ADR-007: Semantic Caching with ElastiCache for Valkey

Status

Accepted

Context

LLM inference is expensive. At 10MM users generating ~100M queries/day:

Without Caching

```
100M queries/day × $0.002 avg per query = $200,000/day = $6M/month

This is completely unsustainable.
```

Many user queries are semantically similar: - “What’s the weather in NYC?” “NYC weather today?” - “How do I cook pasta?” “Steps to make pasta” - “Explain quantum computing” “What is quantum computing?”

If we can identify similar queries and return cached responses, we dramatically reduce LLM calls.

Target Metrics

Metric	Target	Impact
Cache hit rate	80%	80% fewer LLM calls
Hit latency	< 50ms	88% latency improvement
Similarity threshold	0.95	High precision (few false positives)

Metric	Target	Impact
Cache size	100M entries	Cover common queries

Decision

Implement **semantic caching** using ElastiCache for Valkey with vector search:

Architecture

1. **Query embedding**: Embed user query using small, fast model (all-MiniLM-L6-v2)
2. **Vector search**: Find similar cached queries using HNSW index
3. **Threshold check**: If similarity > 0.95, return cached response
4. **Cache miss**: Call LLM, cache result for future queries

Why Valkey?

- **Vector search**: Native HNSW index support
- **Low latency**: Sub-millisecond lookups
- **Managed**: ElastiCache handles scaling, failover
- **Cost**: ~$2K-10K/month vs. dedicated vector DB

Implementation

Cache Architecture

```
                         User Query
                  "What's the weather in NYC?"




                    Query Embedder
              all-MiniLM-L6-v2 (384 dimensions)
                     Latency: ~10ms





              Valkey Vector Search (HNSW)

  FT.SEARCH semantic_cache_idx "*=>[KNN 1 @embedding $vec]"

  Result: "NYC weather today?" (similarity: 0.97)
                     Latency: ~5ms
```

```
            similarity   0.95   similarity < 0.95



      CACHE HIT                   CACHE MISS

  Return cached response      Call LLM
  Total latency: ~20ms        Cache result
                              Total latency: ~2000ms
```

TypeScript Implementation

```typescript
import Redis from 'ioredis';
import { pipeline } from '@xenova/transformers';

export interface CacheResult {
  hit: boolean;
  response: string | null;
  similarity: number;
  latencyMs: number;
  cacheKey: string | null;
}

export interface CacheConfig {
  redisHost: string;
  redisPort: number;
  similarityThreshold: number;
  ttlHours: number;
  embeddingDim: number;
}

export class SemanticCache {
  private readonly redis: Redis;
  private readonly config: CacheConfig;
  private embedder: any = null;

  constructor(config: Partial<CacheConfig> = {}) {
    this.config = {
      redisHost: config.redisHost || 'localhost',
      redisPort: config.redisPort || 6379,
      similarityThreshold: config.similarityThreshold || 0.95,
      ttlHours: config.ttlHours || 23, // Just under 24h
      embeddingDim: config.embeddingDim || 384
    };

    this.redis = new Redis({
      host: this.config.redisHost,
      port: this.config.redisPort
    });
  }

  async initialize(): Promise<void> {
    // Load embedding model
    this.embedder = await pipeline(
      'feature-extraction',
      'Xenova/all-MiniLM-L6-v2'
    );

    // Create vector search index if not exists
    try {
      await this.redis.call(
        'FT.CREATE', 'semantic_cache_idx',
        'ON', 'HASH',
        'PREFIX', '1', 'cache:',
        'SCHEMA',
        'embedding', 'VECTOR', 'HNSW', '6',
        'TYPE', 'FLOAT32',
        'DIM', this.config.embeddingDim.toString(),
        'DISTANCE_METRIC', 'COSINE',
        'query_text', 'TEXT',
        'response', 'TEXT',
        'timestamp', 'NUMERIC'
      );
    } catch (e: any) {
      if (!e.message?.includes('Index already exists')) {
        throw e;
      }
    }
  }

  async lookup(query: string): Promise<CacheResult> {
    const startTime = Date.now();

    // Embed query
    const embedding = await this.embed(query);
    const embeddingBytes = this.float32ArrayToBuffer(embedding);

    try {
      // Vector search for similar cached queries
      const results = await this.redis.call(
        'FT.SEARCH', 'semantic_cache_idx',
        '*=>[KNN 1 @embedding $vec AS score]',
        'PARAMS', '2', 'vec', embeddingBytes,
        'SORTBY', 'score',
  'RETURN', '3', 'response', 'query_text', 'score',
  'DIALECT', '2'
) as any[];

const latencyMs = Date.now() - startTime;

// No results
if (!results || results[0] === 0) {
  return {
    hit: false,
    response: null,
    similarity: 0,
    latencyMs,
    cacheKey: null
  };
}

// Parse result
const docId = results[1] as string;
const fields = results[2] as string[];

let response: string | null = null;
let score = 0;

for (let i = 0; i < fields.length; i += 2) {
  const key = fields[i];
  const value = fields[i + 1];
  if (key === 'response') {
    response = value;
  } else if (key === 'score') {
    score = parseFloat(value);
  }
}

// Convert distance to similarity (cosine distance → similarity)
const similarity = 1 - score;

if (similarity >= this.config.similarityThreshold) {
  return {
    hit: true,
    response,
    similarity,
    latencyMs,
    cacheKey: docId
  };
}

return {
      hit: false,
      response: null,
      similarity,
      latencyMs,
      cacheKey: null
    };

  } catch (e) {
    // Cache lookup failed - treat as miss
    return {
      hit: false,
      response: null,
      similarity: 0,
      latencyMs: Date.now() - startTime,
      cacheKey: null
    };
  }
}

async store(query: string, response: string): Promise<string> {
  const embedding = await this.embed(query);
  const embeddingBytes = this.float32ArrayToBuffer(embedding);

  // Generate cache key
  const hash = await this.hashString(query);
  const cacheKey = `cache:${hash.slice(0, 16)}`;

  // Store with embedding
  await this.redis.hset(cacheKey, {
    query_text: query,
    response: response,
    embedding: embeddingBytes,
    timestamp: Date.now()
  });

  // Set TTL
  await this.redis.expire(cacheKey, this.config.ttlHours * 3600);

  return cacheKey;
}

async invalidateByDomain(domain: string): Promise<number> {
  // Search for entries mentioning domain
  const results = await this.redis.call(
    'FT.SEARCH', 'semantic_cache_idx',
    `@query_text:${domain}`,
    'RETURN', '0'
  ) as any[];
  if (!results || results[0] === 0) {
    return 0;
  }

  // Delete matching entries
  const keys = [];
  for (let i = 1; i < results.length; i++) {
    keys.push(results[i]);
  }

  if (keys.length > 0) {
    await this.redis.del(...keys);
  }

  return keys.length;
}

async getStats(): Promise<{
  hitRate: number;
  totalHits: number;
  totalMisses: number;
  cacheSize: number;
}> {
  const info = await this.redis.info('stats');
  const keyspaceInfo = await this.redis.info('keyspace');

  // Parse stats
  const hits = parseInt(info.match(/keyspace_hits:(\d+)/)?.[1] || '0');
  const misses = parseInt(info.match(/keyspace_misses:(\d+)/)?.[1] || '0');
  const total = hits + misses;

  // Parse cache size
  const dbMatch = keyspaceInfo.match(/db0:keys=(\d+)/);
  const cacheSize = dbMatch ? parseInt(dbMatch[1]) : 0;

  return {
    hitRate: total > 0 ? hits / total : 0,
    totalHits: hits,
    totalMisses: misses,
    cacheSize
  };
}

private async embed(text: string): Promise<Float32Array> {
  const output = await this.embedder(text, {
    pooling: 'mean',
    normalize: true
    });
    return output.data as Float32Array;
  }

  private float32ArrayToBuffer(arr: Float32Array): Buffer {
    return Buffer.from(arr.buffer);
  }

  private async hashString(str: string): Promise<string> {
    const encoder = new TextEncoder();
    const data = encoder.encode(str);
    const hashBuffer = await crypto.subtle.digest('SHA-256', data);
    const hashArray = Array.from(new Uint8Array(hashBuffer));
    return hashArray.map(b => b.toString(16).padStart(2, '0')).join('');
  }
}
```

Consequences

Positive

- **86% cost reduction**: Cache hits avoid LLM inference
- **88% latency improvement**: ~20ms vs ~2000ms
- **Scalable**: Valkey handles millions of cached entries
- **Automatic eviction**: TTL prevents stale responses

Negative

- **Embedding overhead**: ~10ms per query for embedding
- **Storage cost**: ~$2-10K/month for ElastiCache
- **Cache invalidation**: Must invalidate when Cato learns new information
- **Similarity threshold tuning**: Too low = wrong responses, too high = low hit rate

Cache Invalidation Strategy

When Cato learns new information in a domain: 1. Identify affected domain(s) 2. Invalidate all cache entries related to that domain 3. New queries will be answered with updated knowledge

```
// After learning new facts about "climate change"
await semanticCache.invalidateByDomain('climate change');
```

Scaling

Users	Queries/day	Cache Size	Instances	Cost/month
100K	1M	1M entries	2	$2,000
1M	10M	10M entries	4	$4,000
10M	100M	100M entries	12	$12,000

Terraform Configuration

```
resource "aws_elasticache_replication_group" "semantic_cache" {
  replication_group_id       = "cato-semantic-cache"
  description                = "Semantic cache for Cato LLM responses"
  engine                     = "valkey"
  engine_version             = "7.2"
  node_type                  = "cache.r7g.xlarge"
  num_cache_clusters         = 3
  automatic_failover_enabled = true

  parameter_group_name = aws_elasticache_parameter_group.valkey_vector.name

  security_group_ids = [aws_security_group.cache.id]
  subnet_group_name  = aws_elasticache_subnet_group.main.name
}

resource "aws_elasticache_parameter_group" "valkey_vector" {
  family = "valkey7"
  name   = "cato-valkey-vector"

  # Enable RediSearch module for vector search
  parameter {
    name  = "search-enabled"
    value = "yes"
  }
}
```

References

- ElastiCache for Valkey
- Redis Vector Similarity Search
- Sentence Transformers
- Semantic Caching for LLMs