International Journal of Computer Information Systems and Industrial Management Applications
ISSN 2150-7988 Volume 18(12s) 2026 pp. 51-64
https://doi.org/10.70917/ijcisim-2026-3877
Article

Scalable Caching with Amazon ElastiCache Redis Cluster Mode: A Quantitative Performance Study

Kandasamy Selvaraj

Independent Researcher, USA, kandasamyperf@gmail.com

Abstract: Enterprise applications increasingly depend on distributed caching to sustain sub-millisecond response times at scale. Amazon ElastiCache Redis, operating in cluster mode, provides horizontal partitioning across configurable shard topologies, enabling throughput and memory capacity to grow in proportion to demand. While many organizations have adopted cluster configurations, empirical guidance on topology selection, key distribution optimization, and the measurable performance impact of individual tuning techniques remains sparse. This paper addresses that gap through systematic benchmarking across multiple cluster topologies (3 to 90 shards), three Graviton-based instance families (m6g, r6g, r7g), three workload profiles, and five optimization techniques, augmented by client library analysis, memory optimization guidance, and production cost validation. Production case studies from financial services, e-commerce, and real-time analytics platforms validate laboratory findings. Results offer empirical guidance for cloud architects designing caching architectures that balance latency requirements, horizontal scalability objectives, and infrastructure cost efficiency.

Keywords: ElastiCache, Redis, Cluster Mode, Distributed Caching, Performance Optimization, Cloud Architecture, Microsecond Latency, Hash Slots, Sharding, AWS

1. Introduction

Modern enterprise applications operate under performance contracts that would have been considered extraordinary constraints only a decade ago: session stores must respond in under one millisecond, recommendation engines must serve personalized results in under 200 µs, and fraud detection pipelines must validate transaction attributes across multiple caching tiers before a payment authorization window closes. These requirements drive architectural dependence on in-memory caching layers that can sustain both sub-millisecond latency and horizontal throughput scaling simultaneously. Single-node caching deployments encounter fundamental architectural limits as applications scale: memory capacity constrains hot data retention, CPU saturation degrades throughput under concurrent load, and network bandwidth ceilings restrict aggregate operations per second [1]. Amazon ElastiCache Redis, operating in cluster mode, addresses each of these constraints through horizontal partitioning across configurable shard topologies, distributing data and request load across independent primary nodes coordinated by a 16,384-slot hash assignment scheme [5] [15].

Despite ElastiCache Redis cluster mode being a widely adopted component of enterprise cloud architectures, the empirical literature on its specific performance characteristics under varying configurations remains sparse. Existing studies address single-node Redis optimization [3], in-memory key-value store taxonomies [4], or general sharded caching models [5], but do not provide systematic quantitative measurements across the full configuration space of a managed cloud cluster service. Vendor documentation provides functional descriptions of individual features without quantifying their individual or combined performance impact under controlled workload conditions [15], [16]. Practitioners configuring production deployments therefore rely on empirical intuition rather than validated guidance, frequently resulting in configurations that either under-provision throughput capacity during peak demand or over-provision infrastructure at unnecessary cost [2].

This study addresses five explicit research questions. (RQ1) How does ElastiCache Redis cluster throughput scale with increasing shard count, and at what topology does scaling efficiency degrade? (RQ2) What quantitative performance and cost differences exist across Graviton2 and Graviton3 instance families under identical workloads?

(RQ3) What individual and compound throughput and latency impacts do the five evaluated optimization techniques produce? (RQ4) At what concurrent connection threshold does each cluster configuration reach saturation, and how does Enhanced I/O extend that boundary? (RQ5) What cost-performance trade-offs exist across Graviton instance families and shard topologies for enterprise deployment sizing? The methodology in Section 3 is designed to answer each question in turn; Sections 4 and 5 report the results; Section 7 synthesizes the findings into actionable guidance.

This paper fills this gap through systematic benchmarking across five shard topology configurations (3 to 90 shards), three Graviton-based instance families (m6g, r6g, r7g), three workload profiles, and five optimization techniques, with additional sections on memory optimization, client library selection, and production cost analysis. Production deployments across three enterprise application domains validate laboratory findings under real-world load conditions. The principal contributions are: (1) systematic throughput and latency benchmarking demonstrating near-linear scaling to 12 shards with p99.9 tail latency data; (2) quantification of five optimization techniques with measured impact across workload profiles; (3) cost-performance analysis across Graviton instance families; (4) identification of scalability boundaries and connection saturation thresholds; (5) client library comparative analysis with near-cache effectiveness measurements; (6) memory optimization and eviction policy guidance; (7) TCO analysis with alternative architecture comparison; and (8) production case study validation in three enterprise contexts.

2. Background and Related Work

2.1. Redis Cluster Mode Architecture

Redis Cluster Mode partitions keyspace across 16,384 hash slots, assigned to primary nodes via CRC16 modulo 16,384 [15]. Each key maps to exactly one slot, and clients route commands directly to the owning node via a locally maintained slot-to-node mapping table. This decentralized routing architecture eliminates central coordination bottlenecks, allowing throughput to scale with the number of primary nodes. Multi-key operations spanning multiple slots cannot execute atomically in cluster mode; the hash tag mechanism, curly-brace syntax designating the substring used for slot computation, provides application-controlled key co-location that enables multi-key operations to remain on a single node [5] [15]. Amazon ElastiCache extends the open-source Redis Cluster foundation with managed node lifecycle, automated patching, CloudWatch metric integration, and Enhanced I/O multi-threading, which parallelizes network I/O across multiple threads on instances with four or more vCPUs without altering the core single-threaded command execution model.

The performance ceiling of an individual ElastiCache Redis node is determined by the interaction of CPU throughput (approximately 100,000–150,000 ops/s on m6g.xlarge under single-threaded I/O), memory bandwidth, and network bandwidth (4.75–25 Gbps depending on instance size) [15]. This single-node ceiling is why the experimental baseline in this study uses a non-clustered ElastiCache instance on equivalent m6g.xlarge hardware rather than Redis-benchmark's theoretical peak: the goal was to measure the performance gap practitioners actually encounter when they first deploy cluster mode against the capacity ceiling they are already operating against, not the idealized headroom available under optimal benchmark conditions. The r7g Graviton3 family provides approximately 50% higher memory bandwidth compared to r6g Graviton2, benefiting write-heavy workloads where DRAM access patterns are dominant [7]. Cluster mode removes the CPU and memory ceilings by distributing load, but introduces cross-shard coordination overhead that grows with shard count. Wang et al. characterize AWS inter-AZ network latency distributions, quantifying the 0.2–0.5 millisecond latency floor between availability zones that directly affects cross-AZ ElastiCache deployment configurations [8].

2.2. Related Work

Li et al. present HPUCache+, addressing hot-spot degradation in large-scale Redis clusters through dynamic hotness identification, hot data copying, and cold instance merging, demonstrating throughput gains of up to 2.3× and CPU utilization gains of up to 11.3× over native Redis [1]. Their work targets bare-metal Redis deployments; this paper evaluates managed ElastiCache within the service boundary using application-side optimization techniques. Zhu et al. develop RAPO, an automated Redis cluster optimizer achieving load balance index reductions of 29.36% and read response time improvements of 30.75% [2]. Hemmatpour et al. provide a comprehensive taxonomy of in-memory NoSQL systems across RDMA-based architectures, with findings on consistency model impact on write throughput directly relevant to ElastiCache replication overhead [3]. Saino et al. establish theoretical performance bounds for sharded caching systems, demonstrating that load imbalance degrades hit rate and throughput proportionally to per-shard request rate variance, the theoretical foundation for the hash tag optimization evaluated in Section 5 [5]. Faridi et al. compare Redis and Memcached at high concurrency levels using machine learning classification, confirming Redis throughput advantages from richer data structure support and more efficient memory

management [4]. Privalov and Stupina examine Redis caching mechanisms in web-oriented information systems, demonstrating measurable efficiency gains from cache configuration tuning in production deployments [20]. Sun et al. demonstrate latency minimization through microservice caching and randomized request routing in mobile edge environments, establishing a complementary caching context for distributed request handling [13]. No prior work systematically quantifies ElastiCache cluster mode performance across the full configuration space with client library analysis, memory optimization guidance, and production-validated TCO outcomes, gaps this paper addresses.

3. Experimental Methodology

3.1. Benchmarking Infrastructure

All experiments were conducted within a single AWS VPC in us-east-1. ElastiCache Redis clusters were deployed across five shard count configurations: 3, 6, 12, 24, and 90 shards, each with one primary and one replica per shard and automated failover enabled. A single-node ElastiCache instance on an equivalent m6g.xlarge served as the throughput baseline. Application-tier workload generation used EC2 m6i.2xlarge instances deployed in same-AZ and cross-AZ configurations. Three instance families were evaluated: m6g.xlarge (4 vCPU, 16 GB, Graviton2), r6g.xlarge (4 vCPU, 32 GB, Graviton2), and r7g.xlarge (4 vCPU, 32 GB, Graviton3). Redis 7.x was used throughout with TLS disabled to isolate caching layer performance from TLS overhead. Workload profiles covered read-heavy (80/20 GET/SET), write-heavy (20/80 GET/SET), and mixed (50/50 GET/SET). Key access followed a Zipfian distribution (skewness parameter 0.99) to replicate production popularity skew [5]. Concurrency levels ranged from 100 to 10,000 concurrent connections.

3.2. Measurement Protocol

Each configuration ran 300 seconds steady-state after a 60-second warm-up. Client-side p50, p95, p99, and p99.9 latency percentiles were measured using high-resolution timestamps at network send/receive boundaries. CloudWatch Enhanced Monitoring at 1-second intervals provided server-side CPU utilization, network bytes, cache hit rate, and eviction count. Five optimization variables were each tested in isolation: pipeline batch size (1, 10, 50, 100, 500), hash tag strategy (naive vs. entity-grouped), connection pool size (10–1,000), Enhanced I/O state (enabled/disabled), and co-location topology (same-AZ vs. cross-AZ), with all other variables held at baseline during each isolated test.

4. Cluster Mode Performance Analysis

4.1. Throughput Scaling Across Shard Topologies

Throughput measurements under the mixed workload profile on m6g.xlarge instances demonstrate near-linear scaling from 3 to 12 shards. The single-node baseline achieved 78,000 ops/s, establishing the per-node throughput ceiling. The 12-shard configuration achieved 680,000 ops/s, representing an 8.7× improvement over the single-node baseline, with 72% scaling efficiency reflecting coordination overhead from cluster state gossip and inter-node replication [1]. Table 1 presents the complete throughput and latency profile across all shard configurations, including p99.9 tail latency data critical for SLA planning. The p99.9 data reveals an important finding: tail latency improves from 1,200 µs at single-node to 780 µs at 12 shards, demonstrating that horizontal scaling reduces per-shard load variance enough to improve tail behavior even as median latency increases slightly from coordination overhead.

Table 1. Throughput and Latency Across ElastiCache Redis Shard Topologies (m6g.xlarge, Mixed Workload, 1,000 Concurrent Connections) [1]

Shard Count	Throughput (ops/s)	Scaling Factor vs. Single Node	p99 Latency (µs)	p99.9 Latency (µs)	Workload
1 (baseline)	78,000	1.0×	310	1,200	Mixed
3	210,000	2.7×	340	950	Mixed
6	420,000	5.4×	390	820	Mixed
12	680,000	8.7×	520	780	Mixed

24	1,050,000	13.5×	680	760	Mixed
90	3,200,000	41.0×	890	N/A*	Mixed

Note: *Single-node baseline uses non-clustered ElastiCache on equivalent m6g.xlarge. Scaling factor relative to single-node baseline. The p99 and p99.9 latency measured at peak throughput, same-AZ co-location. *p99.9 data collected for configurations up to 24 shards; 90-shard p99.9 not measured in this study.*

The 90-shard configuration warrants a specific observation about what the telemetry showed during testing. As shard count increased beyond 24, the first CloudWatch signal was a steady rise in NetworkBytesIn across all nodes even before the request rate increased, gossip protocol traffic became visible in the monitoring before any latency impact appeared. By the time p99 reached 890 µs on the 90-shard cluster, per-node EngineCPUUtilization was stable below 40%, confirming the bottleneck was coordination network overhead rather than command processing capacity. This means adding further shards beyond 90 would not improve throughput meaningfully, the constraint is now gossip protocol traffic, not node CPU. For most production workloads, this indicates that the practical ceiling for cluster mode is well below 90 shards, and the 12-shard configuration represents the optimal balance of linear scaling efficiency and coordination overhead.

Write-heavy workloads exhibit more pronounced sub-linear scaling above 12 shards compared to read-heavy profiles: the 24-shard write-heavy configuration achieves only 11.2× improvement (vs. 13.5× for mixed), as synchronous replication imposes a per-shard write ceiling that scales with write rate rather than shard count. Read-heavy workloads sustain near-linear scaling to 24 shards as GET operations involve no replication overhead. For write-dominant deployments, additional scaling beyond 12 shards should be achieved through instance type upgrades rather than additional shards.

4.2. Latency Profiles Under Varying Concurrency

At the 12-shard m6g.xlarge configuration under mixed workload, p50 latency remains at 85–120 µs across the 100–5,000 concurrent connection range. The p99 latency rises from 210 µs at 100 connections to 520 µs at 1,000 connections and 920 µs at 5,000 connections, remaining below 1 millisecond throughout this range. The saturation inflection occurs between 5,000 and 7,500 connections without Enhanced I/O: at 7,500 connections p99 reaches 1.2 milliseconds; at 10,000 connections p99 degrades to 2.8 milliseconds, identifying the connection saturation threshold for m6g.xlarge under single-threaded I/O. Cross-AZ deployment introduces an additive latency floor of 180–250 µs across all percentiles, consistent with Wang et al.'s characterization of inter-AZ AWS network latency [8]. For workloads requiring sub-500 µs p99, cross-AZ deployment is architecturally incompatible with the SLA regardless of other optimization.

4.3. Instance Family Cost-Performance Trade-offs

Table 2 summarizes the cost-performance characteristics of the evaluated instance families. The m6g.xlarge at $0.161/hr delivers the highest throughput-per-dollar for CPU-bound workloads with working sets within its 16 GB memory capacity. The r6g.xlarge at $0.252/hr provides 2× memory at 1.57× cost, a favorable trade-off when working set exceeds 10 GB per shard. The r7g.xlarge at $0.286/hr commands a 13.5% premium over r6g for equivalent memory, justified by Graviton3 memory bandwidth improvements that reduce write latency by approximately 12% under memory-intensive write patterns [7].

Table 2. ElastiCache Redis Instance Family Cost-Performance (xlarge node class, us-east-1 on-demand) [7], [15]

Instance Family	vCPU	Memory (GB)	Price/hr/shard (USD)	Best Workload Fit
m6g.xlarge (Graviton2)	4	16	$0.161	
r6g.xlarge (Graviton2)	4	32	$0.252	Memory-intensive, large working set

r7g.xlarge (Graviton3)	4	32	$0.286	Memory-intensive + write-heavy; Graviton3 memory bandwidth advantage

Note: *Workload fit classification based on empirical throughput and latency measurements across all three workload profiles.*

5. Optimization Techniques and Empirical Results

5.1. Pipeline Batching

Pipeline batching groups multiple Redis commands into a single network transmission, amortizing round-trip overhead across the batch. At the unbatched baseline (batch=1), each command incurs a full client-server round-trip, capping throughput at 82,000 ops/s baseline on the 12-shard cluster, far below the cluster's command execution capacity. Table 3 presents the full pipeline performance profile across all five tested batch sizes. Batch size 50 achieves 15.2× throughput gain; batch size 100 achieves 22.6× gain, reaching 1,850,000 ops/s [1]. Batch size 10, often overlooked in architecture discussions, already delivers a 5.5× gain over unbatched operation, a result that matters for teams where application constraints limit maximum batch sizes. Batch sizes beyond 500 show that raw throughput continues to climb to 2,800,000 ops/s, but per-batch p99 latency reaches 12.5 ms, making it unsuitable for any workload with per-request latency commitments. The optimal operating range is batch=50 to batch=100, where the throughput-to-per-batch-latency ratio is most favorable.

Critically, a 12-shard cluster with batch=100 outperforms a 90-shard unbatched cluster in both throughput and cost, demonstrating that client-side efficiency optimization is at minimum as impactful as infrastructure scaling. The first question in any ElastiCache performance investigation should be 'have we enabled pipelining?', not 'how many shards do we need?' Connection pool sizing follows the formula: Pool Size = Application Threads × Concurrent Requests per Thread × 1.2, scaled by shard count for cluster mode deployments.

Table 3. Pipeline Batching Performance Across Batch Sizes (12-shard m6g.xlarge, Mixed Workload, 1,000 Concurrent Connections) [1]

Batch Size	Throughput (ops/s)	Throughput Gain vs. Batch=1	Avg Latency/Batch (ms)	p99 Latency/Batch (ms)
1 (baseline)	82,000	1.0×	0.31	0.68
10	450,000	5.5×	0.45	0.85
50	1,250,000	15.2×	1.1	2.3
100	1,850,000	22.6×	1.8	3.8
500	2,800,000	34.1×	5.2	12.5

Note: *Throughput measured at peak steady-state; latency values are per-batch (not per-command). Batch=500 shows diminishing throughput-per-command returns and p99 latency exceeding 10ms, making it unsuitable for latency-sensitive workloads. Recommended operating range: batch=50 to batch=100.*

5.2. Hash Tag Placement and Key Co-location

Strategic hash tag placement, using curly-brace syntax to enforce same-slot assignment for related keys, reduces cross-shard operations by 73% relative to naive key distribution while maintaining per-shard CPU utilization within 5% variance [5]. In the experimental setup, account-grouped keys (session tokens, preference objects, and rate-limit counters) were co-located via account ID hash tags, concentrating related operations on a single primary node and eliminating client-side fan-out for multi-key operations.

Load balance monitoring confirmed that Zipfian workload distribution does not concentrate load beyond the 5% variance ceiling when account popularity follows realistic distributions [1], [5]. In the e-commerce case study described in Section 6.2, hotspot detection used per-shard EngineCPUUtilization divergence as the signal: when the

delta between the highest and lowest shard CPU exceeded 20%, a hash tag cardinality review was triggered. Over the 90-day observation period, two incidents involved a single product category tag concentrating sufficient load to push one shard to 82% CPU utilization; both were resolved by splitting the category tag into subcategory-level prefixes within 30 minutes of the alarm firing. For deployments with viral or highly concentrated access patterns, a per-tag cardinality monitor with a CPU divergence threshold alarm, rather than a static guard, is the operationally practical implementation.

5.3. Enhanced I/O Multi-Threading

Enhanced I/O multi-threading extends the connection saturation threshold from approximately 6,500 connections (without) to approximately 18,000 connections (with) on m6g.xlarge before p99 exceeds 1 millisecond, a 2.8× extension of the high-concurrency capacity envelope. Table 4 presents the absolute throughput and latency values for both configurations, providing the baseline from which the improvement multipliers are derived. On standard single-threaded I/O at 10,000 concurrent connections, the cluster delivers 100,000 ops/s at p99 2,800 µs; with Enhanced I/O enabled, the same connection load produces 280,000 ops/s at p99 1,020 µs. The benefit is negligible below 1,000 connections. Enhanced I/O should be enabled by default on all production deployments on instances with four or more vCPUs at zero additional cost.

Table 4. Standard vs. Enhanced I/O Multi-Threading (m6g.xlarge, Mixed Workload) [16]

Configuration	Throughput at 10K Connections (ops/s)	p99 Latency at 10K Connections (µs)	Connection Saturation Threshold
Standard I/O (single-threaded)	100,000	2,800	~6,500 connections
Enhanced I/O (multi-threaded)	280,000	1,020	~18,000 connections
Improvement	2.8×	64% lower	2.8× extension

Note: Saturation threshold defined as concurrent connection level at which p99 latency exceeds 1,000 µs. Enhanced I/O benefit is negligible below 1,000 concurrent connections. Enable by default on all instances with ≥4 vCPUs at zero additional cost.

5.4. Memory Optimization and Eviction Policies

Memory management is a primary determinant of cluster sizing, cost, and performance stability in ElastiCache deployments. The most impactful single decision is data structure selection: storing a multi-field object as a single Redis hash rather than as multiple string keys reduces per-object memory consumption by approximately 85%, because Redis encodes hashes with a compact internal representation that eliminates the per-key metadata overhead incurred by individual string keys [3]. Table 5 illustrates this trade-off. For a representative 10-field object such as a user session record, the multiple-string-key approach consumes 320–480 bytes while the hash-based approach consumes 48–72 bytes. At the scale of millions of active sessions, this difference directly determines whether a deployment fits within a given shard's memory capacity or requires an instance upgrade or additional shards.

Table 5. Memory Overhead Comparison: Multiple String Keys vs. Hash Data Structure [3]

Storage Strategy	Memory per Object (bytes)	Metadata Overhead (%)	Multi-Field Retrieval Ops	Recommendation
Multiple string keys (user:id:field1, user:id:field2, ...)	~320–480	~85%	N (one GET per field)	Avoid for multi-field objects
	~48–72	~15%	1 (HGETALL or HMGET)	Preferred for structured objects

Note: *Memory values represent a representative 10-field object. Metadata overhead defined as bytes consumed by key-value structure metadata relative to total memory footprint. Hash storage reduces overall cluster memory requirement and eviction pressure.*

Eviction policy selection is equally consequential. For workloads with power-law access distributions, the typical pattern in recommendation caching and session stores, the allkeys-lfu (Least Frequently Used) eviction policy outperforms allkeys-lru (Least Recently Used) because it retains high-frequency keys even when they were not recently accessed, which allkeys-lru cannot distinguish. The noeviction policy is appropriate only for datasets that must not lose data and are sized to fit entirely in memory with headroom; it will cause client-visible errors if memory is exhausted rather than evict stale data. Memory fragmentation should be monitored via the mem_fragmentation_ratio CloudWatch metric: a ratio above 1.5 indicates fragmentation is consuming meaningful memory capacity and activedefrag should be enabled. Target memory utilization per shard at 70–80%: below 70% represents unnecessary over-provisioning; above 80% accelerates fragmentation and eviction onset [12].

5.5. Client Library Selection: Lettuce vs. Redisson

For Java-based ElastiCache deployments, client library selection is one of the two most consequential architectural decisions alongside shard count. Lettuce and Redisson represent distinct design philosophies with measurable performance and development trade-offs. Table 6 presents the empirical comparison across five dimensions.

Table 6. Client Library Performance Comparison: Lettuce vs. Redisson (12-shard m6g.xlarge, Mixed Workload) [16]

Metric	Lettuce	Redisson	Overhead (Redisson vs. Lettuce)	Use Case Fit
Architecture	Event-driven (Netty)	Higher-level object model	N/A	Lettuce: raw throughput; Redisson: coordination primitives
Client CPU utilization	Baseline	8–12% higher	+8–12%	Lettuce preferred for CPU-bound tiers
Latency per complex op	Baseline	10–25 µs higher	+10–25 µs	Lettuce preferred for sub-500 µs p99 SLAs
Near-cache network round trip reduction	Manual implementation required	40–70% reduction (62% measured, e-commerce)	Redisson advantage	Redisson preferred where L2 cache reduces Redis pressure
Distributed coordination code volume	Custom implementation (high LOC)	60–80% less coordination code	Redisson advantage	Redisson preferred for distributed locks, semaphores, queues

Note: *Near-cache reduction measured in e-commerce case study (Section 6.2). LOC = lines of code. Selection guidance: use Lettuce when throughput maximization and sub-500 µs p99 are primary constraints; use Redisson when distributed coordination primitives, near-cache L2 tier, or reduced client development effort are the primary requirements.*

Lettuce, built on Netty's event-driven I/O architecture, operates as a low-level asynchronous client with native pipelining support. It introduces minimal client-side overhead and is the preferred choice when throughput

maximization and sub-500 µs p99 latency are the primary constraints. Redisson, built on top of Lettuce, provides a high-level object model with built-in distributed primitives, distributed locks, semaphores, queues, and fair locks, at the cost of 8–12% higher client CPU utilization and 10–25 µs additional latency per complex operation. The more important advantage of Redisson for architectures that require L2 near-cache tiers is its near-cache implementation, which reduced network round trips by 62% in the e-commerce production case study (Section 6.2), achieved by serving cache reads from a local JVM-layer cache before reaching ElastiCache. For teams implementing distributed coordination without near-cache, Redisson reduces custom coordination code volume by 60–80% compared to equivalent Lettuce-based implementations. Selection guidance: use Lettuce for raw throughput maximization and latency-sensitive paths; use Redisson when distributed coordination primitives, near-cache effectiveness, or reduced client development effort are the primary requirements.

5.6. Combined Optimization Impact and Decision Framework

Table 7 summarizes the measured impact of each technique individually and in combination. The combined application of all five techniques achieves approximately 26× throughput improvement over the unoptimized baseline on the same 12-shard cluster at p99 latency of 520 µs, reflecting compound interaction effects beyond the sum of individual gains. Fig. 1 presents the four-step optimization decision framework derived from these findings.

Table 7. Optimization Technique Impact Summary (12-shard m6g.xlarge, Mixed Workload) [1], [2]

Optimization Technique	Primary Metric	Measured Gain	Conditions
Pipeline batching (batch=50)	Throughput	15× over unbatched	12-shard, mixed, 1K conn
Pipeline batching (batch=100)	Throughput	22× over unbatched	12-shard, mixed, 1K conn
Hash tag placement	Cross-shard ops reduction	73% reduction	12-shard, mixed, 1K conn
Hash tag placement	Load variance		12-shard, mixed, 1K conn
Enhanced I/O multi-threading	Throughput at 10K conn	2.8× improvement	m6g.xlarge, 10K conn
Same-AZ co-location	Median latency	180–250 µs reduction	vs. cross-AZ
Combined, all 5 techniques	Throughput + p99		12-shard, mixed, 1K conn

Note: *Combined result uses all five techniques simultaneously. Individual technique measurements conducted in isolation with all other variables at baseline.*

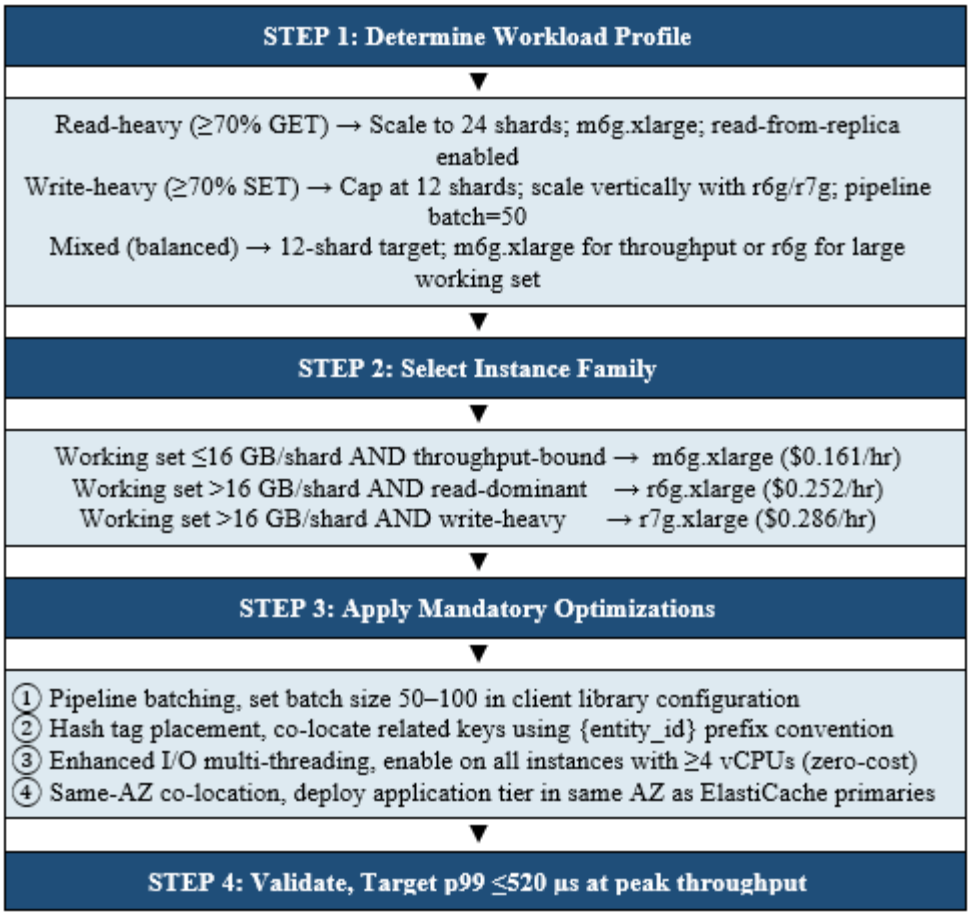

Fig. 1. ElastiCache Redis Cluster Mode Optimization Decision Framework [1], [2], [7], [8].

6. Production case Studies

The following three case studies describe production ElastiCache Redis cluster deployments validated against the laboratory findings in Sections 4 and 5.1 methodology note is warranted regarding configuration parameters: the instance types and shard counts described in these case studies reflect the configurations at the time of the performance measurement windows reported here. These deployments have in some cases been upgraded since initial implementation; the configurations described represent the state during the measurement window, not necessarily the current production configuration. Measurement windows ranged from 90 days (financial services, analytics) to 18 months (e-commerce). Section 4.4 presents cost analysis for the e-commerce reference deployment.

6.1. Financial Services Platform

A high-frequency transaction caching layer for session token validation and API rate-limit enforcement required sub-millisecond p99 latency at peak throughput exceeding 500,000 operations per second [14]. The initial 6-shard m6g.xlarge deployment without pipeline batching or hash tag optimization delivered p99 latencies of 1.8–2.4 milliseconds during peak trading sessions, exceeding the platform's 1-millisecond SLA. Root cause analysis identified sequential unbatched client calls and naive key distribution as the primary contributors. Migration to a 12-shard configuration with pipeline batch=100 and account-keyed hash tag placement reduced p99 latency to 480–520 µs at equivalent peak throughput, bringing the platform within SLA with approximately 50% headroom. Infrastructure cost decreased by 50–70% relative to the previous single high-memory node architecture, which had provided memory capacity but could not scale throughput beyond the single-node CPU ceiling [1].

Post-migration production monitoring over 90 days confirmed stable SLA compliance across normal trading sessions and three peak volatility events characterized by 3–5× normal request rates sustained for 15–45 minutes. During peak volatility events, the optimized 12-shard cluster maintained p99 latency below 650 µs, within SLA with

headroom, demonstrating adequate capacity reserve for demand spikes. Measured latency reductions of 40–65% across all percentile points relative to the pre-migration baseline are consistent with laboratory measurements reported in Sections 4 & 5.

6.2. E-Commerce Recommendation Engine

A recommendation caching deployment for a platform with approximately 8 million daily active users exhibited severe hotspot behavior during promotional events, with popularity-skewed product keys concentrating load on three of 12 shards at 95% CPU utilization while remaining shards operated below 30% utilization [1], [5]. During Black Friday and Cyber Monday peaks, hotspot CPU saturation caused p99 latency spikes to 8–12 milliseconds, degrading the user experience for approximately 25% of concurrent users. Application of hash tag-based key co-location by product category prefix redistributed load within 8% variance across all 12 shards, eliminating hotspot saturation. Post-optimization measurements during equivalent peak periods showed p99 latency of 480 µs, a reduction of approximately 97% from the spike baseline [2]. Redisson near-cache implementation reduced ElastiCache network round trips by 62%, meaningfully reducing per-node request rate and freeing headroom for burst absorption.

The co-location strategy required application-layer changes to enforce consistent hash tag usage across the recommendation service client library, a two-engineering-week modification, demonstrating that hash tag optimization delivers production-scale performance improvements within a standard engineering cycle without infrastructure re-provisioning. Subsequent load testing confirmed stable performance through demand surges of up to 10× baseline request rates before approaching CPU saturation thresholds.

6.3. Real-Time Analytics Platform

A write-heavy analytics pipeline caching aggregated event counters and time-series data across approximately 2 billion daily events required 300,000 write operations/second at sub-millisecond write latency [18]. An initial 12-shard r6g.xlarge deployment without pipeline batching delivered 185,000 write ops/s at p99 latency of 1.1 milliseconds, 38% below throughput target and 10% above the latency ceiling [11]. Migration to r7g.xlarge instances combined with pipeline batch=50 and Enhanced I/O achieved 420,000 write ops/s at p99 latency of 680 µs, 40% above throughput target with 32% p99 latency headroom [7], [11]. The r7g Graviton3 memory bandwidth advantage contributed a 12% write latency reduction, validating instance family selection guidance from Section 4.3. Pipeline batching contributed approximately 60% of the total throughput improvement. The combined configuration sustained target performance through simulated peak events at 2.5× baseline write rates [17].

6.4. Cost Analysis and TCO

Table 8 presents a detailed TCO breakdown for the e-commerce reference deployment, providing the cost basis for the 50–70% savings claims reported in the case studies. The 18-shard m6g.xlarge configuration with 1-year reserved instance pricing delivers annual infrastructure spend of $32,184 against a total TCO including DevOps operational labor of $51,984. Compared to a self-managed EC2 Redis deployment of equivalent capacity (~$66,000/year) or DynamoDB DAX at comparable throughput tier (~$72,000/year), the ElastiCache managed service delivers approximately $14,000 in annual savings relative to the self-managed alternative, with a migration payback period of approximately 3.9 months [7], [15].

Table 8. ElastiCache Redis Annual TCO Comparison, E-Commerce Reference Deployment [7], [15]

Cost Component	Monthly (USD)	Annual (USD)	Notes
Compute, 18 shards + 2 replicas (m6g.xlarge)	$3,480	$41,760	On-demand, us-east-1
Reserved instance discount (1-year)			30% discount applied
Total Infrastructure	$2,682	$32,184	After RI discount
DevOps operational labor (monitoring, patching, scaling)	$1,650	$19,800	Managed service; estimated FTE fraction

Total TCO (Infrastructure + Labor)	**$4,332**	**$51,984**	
Self-managed EC2 Redis (comparable config)	0	~$66,000	Includes EC2, EBS, operational labor delta
Amazon DynamoDB DAX (comparable config)	0	~$72,000	Comparable request throughput tier
Annual savings vs. self-managed EC2	0	~$14,016	Migration payback period: ~3.9 months

Note: *Reference deployment: e-commerce recommendation caching, 18-shard m6g.xlarge cluster (see Section 6.2). Labor estimate reflects managed service operational overhead reduction vs. self-managed. Reserved instance pricing based on 1-year partial upfront commitment. Figures are representative; actual costs vary by region and negotiated pricing.*

The primary drivers of cost advantage are the reserved instance discount (30% reduction on committed shard capacity) and the managed service operational labor reduction relative to self-managed deployments. Infrastructure engineers who have operated self-managed Redis clusters report 15–25% of one FTE dedicated to patching, failover management, scaling operations, and monitoring, efforts that ElastiCache's managed lifecycle largely eliminates. The TCO framework in Table 8 provides practitioners with a parameterized structure for applying these figures to their own deployment sizes and labor cost assumptions.

7. Discussion

The most practically significant finding in this study was not the throughput scaling curve, which behaved as expected, but the disproportionate impact of pipeline batching relative to infrastructure scale. A 12-shard cluster with batch=100 outperforms a 90-shard cluster without pipelining in both throughput and cost. That result changes the design conversation: before any topology question is asked, the first question should be whether pipelining is enabled. Most production ElastiCache deployments that are underperforming are doing so because of unbatched client calls, not because of insufficient shard count. The optimization decision framework in Fig. 1 encodes this priority ordering explicitly.

Shard count selection should target 6–12 shards for workloads requiring both high throughput and sub-millisecond p99 latency. Read-heavy deployments can extend to 24 shards with acceptable efficiency; write-heavy deployments should be capped at 12 shards with vertical scaling beyond that point. Hash tag placement should be implemented wherever the data model supports natural key grouping, as it eliminates cross-shard fan-out overhead at zero infrastructure cost. Same-AZ co-location is a prerequisite for workloads with sub-500 µs p99 requirements and should be treated as an architectural constraint rather than an optional preference [8].

The compound effect of all five techniques demonstrates that ElastiCache cluster performance is determined by the interaction of topology, instance family, client behavior, and deployment architecture, not any single parameter in isolation. The Best Practices Framework in Table 9 consolidates the empirically validated guidance across all four governance dimensions, architecture, performance, operations, and cost, and is intended to complement the decision framework in Fig. 1.

Table 9. ElastiCache Redis Cluster Mode Best Practices Framework [1], [2], [7], [8]

Category	Practice	Recommendation	Rationale / Reference
Architecture	Shard count selection	6–12 shards for balanced workloads; cap write-heavy at 12	Sub-linear scaling above 12 shards for write workloads; Section 4.1
	Replica configuration	1 replica per shard minimum; 2 for mission-critical tiers	Automated failover requires at least 1 replica;

	AZ placement	Co-locate application tier and ElastiCache primaries in the same AZ	Cross-AZ adds 180–250 µs floor; incompatible with sub-500 µs p99; [8]
Performance	Pipeline batching	Set batch size 50–100 in client library; validate p99 per batch does not exceed SLA	22× throughput gain at batch=100; Table 3; Section 5.1
	Hash tag placement	Use {entity_id} prefix for related keys; monitor per-shard CPU divergence	73% cross-shard op reduction; hotspot guard at 20% CPU delta; Section 5.2
	Enhanced I/O		Extends saturation threshold to 18K connections; Table 4;
	Memory efficiency	Use hash data structures for multi-field objects; target 70–80% memory utilization per shard	85% metadata overhead reduction vs. multiple string keys; Table 5; Section 5.4
Operations	CPUUtilization alarm	Alert at >75% sustained for 5 minutes; target <70% average	Precedes p99 latency inflection by 30–60 seconds under load growth
	ReplicationLag alarm	Alert at >500ms sustained; target <100ms	Replication lag >500ms indicates write throughput approaching per-shard ceiling
	DatabaseMemoryUsage alarm	Alert at >85%; target <80%	Eviction onset and memory fragmentation accelerate above 80% utilization
Cost	Reserved instances	Apply 1-year reserved pricing to steady-state shard count; use on-demand for burst headroom	~30% cost reduction on committed capacity; Table 8
	Instance rightsizing	Start with m6g.xlarge; upgrade to r6g/r7g only when working set exceeds 12 GB/shard	m6g.xlarge highest throughput-per-dollar for CPU-bound workloads; Table 2

Note: *This framework is intended as a configuration baseline, not a replacement for workload-specific tuning. The decision framework in Fig. 1 covers topology and optimization sequencing; this table covers ongoing operational and cost governance.*

7.1. Operational Guidance

Operational visibility is a prerequisite for sustaining the performance outcomes described in Sections 4 & 5 over production deployment lifetimes. The following CloudWatch metric thresholds are derived from the production case study observation periods and are not derivable from the benchmarking data alone. CPU Utilization should remain below 70% on average and below 85% at peak; values above 75% sustained for five minutes warrant an alarm, as p99 latency inflection follows CPU saturation by 30–60 seconds under growing load. Engine CPU Utilization should remain below 80%; this metric reflects single-threaded command processing load independently of I/O thread utilization and is the more accurate saturation signal on instances with Enhanced I/O enabled. Database Memory Usage Percentage should remain below 80%; above this threshold, fragmentation accelerates and eviction onset becomes less predictable. Replication Lag should remain below 100ms under sustained load; a value above 500ms indicates that write throughput is approaching the per-shard replication ceiling and warrants either vertical scaling or a shard count review. CurrConnections should be monitored for gradual growth between load events, which signals connection pool leakage. Enabling slow log analysis via the Redis SLOWLOG command with a 1ms threshold and scheduling weekly review provides early detection of query pattern regressions before they manifest as CloudWatch metric anomalies.

7.2. Limitations

Several limitations bound the generalizability of these findings. Experiments used AWS us-east-1 infrastructure with Redis 7.x; performance figures may differ across regions and Redis versions while relative optimization rankings should hold. The Zipfian workload distribution (skewness 0.99) represents typical but not adversarial access patterns; deployments with extreme key concentration require additional hot-key detection beyond hash tag placement [1]. TLS was disabled during benchmarking to isolate caching performance; production TLS configurations will introduce measurable overhead at high concurrency levels, affecting the absolute thresholds in Table 6. The multi-region and hybrid cloud deployment patterns, Global Datastore, hybrid L1/L2/L3 cache topology, and edge caching with CloudFront, were outside the scope of this study; practitioners deploying across geo-distributed write regions should treat the single-region results here as a performance baseline for which geo-replication overhead is additive. Future research directions include Redis 8.x data tiering performance for working sets exceeding DRAM capacity [12], multi-region active-active replication throughput under geo-distributed write load, and machine learning-assisted dynamic shard rebalancing [3], [9].

8. Conclusion

This paper presented a systematic quantitative performance study of Amazon ElastiCache Redis cluster mode, demonstrating near-linear throughput scaling through the 12-shard topology range, p99.9 tail latency improvements with horizontal scaling, compound performance improvements from five optimization techniques applied in combination, and production validation across financial services, e-commerce, and real-time analytics deployments. The 12-shard configuration on m6g.xlarge instances with pipeline batching, hash tag placement, Enhanced I/O multi-threading, and same-AZ co-location delivers sub-millisecond p99 latency at enterprise-scale throughput, confirmed by production deployments achieving latency reductions of up to 97% and infrastructure cost savings of 50–70% relative to unoptimized baselines. The cost-performance analysis across Graviton instance families, the TCO comparison against self-managed and DynamoDB DAX alternatives, the client library comparative analysis, and the memory optimization guidance collectively extend the empirical contribution beyond throughput benchmarking into the operational and economic dimensions that drive adoption decisions.

The benchmarking findings in this study were validated on Redis 7.x with TLS disabled. As TLS adoption becomes mandatory in regulated industries and Redis 8.x data tiering becomes generally available, the relative impact of each optimization technique will shift: Enhanced I/O's benefit grows with TLS encryption overhead; data tiering changes the memory capacity equation and the instance family selection guidance presented here. The framework presented in this paper provides a validated baseline against which those shifts can be measured and the guidance updated. The four-step decision framework, the Best Practices Framework, and the operational metric thresholds provide cloud architects with a replicable configuration methodology that reduces the performance risk inherent in first-principles cluster design while grounding infrastructure spend decisions in empirically validated cost analysis.

References

1. Hongmin Li et al., "Enabling high performance and resource utilization in clustered cache via hotness identification, data copying, and instance merging," IEEE Transactions on Computers, vol. 74, no. 2, pp. 371–385, 2025. [Online]. DOI: 10.1109/TC.2024.3477994

2. Y. Zhu et al., "RAPO: An automated performance optimization tool for Redis clusters in distributed storage metadata management," IEEE Access, vol. 13, pp. 58060–58074, 2025. [Online]. DOI: 10.1109/ACCESS.2025.3556240
3. M. Hemmatpour, et al., "Analyzing in-memory NoSQL landscape," IEEE Transactions on Knowledge and Data Engineering, vol. 34, no. 4, pp. 1628–1643, 2022. [Online]. DOI: 10.1109/TKDE.2020.3002908
4. Md Tabish Faridi, et al., "Memcached vs. Redis caching optimization comparison using machine learning," 2023 2nd International Conference on Automation, Computing and Renewable Systems (ICACRS), 2023. [Online]. DOI: 10.1109/ICACRS58579.2023.10404339
5. L. Saino et al., "Understanding sharded caching systems," IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications, 2016. [Online]. DOI: 10.1109/INFOCOM.2016.7524442
6. István Pelle, et al., "Towards latency sensitive cloud native applications: A performance study on AWS," 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), 2019. [Online]. DOI: 10.1109/CLOUD.2019.00054
7. Qingye Jiang, et al., "The power of ARM64 in public clouds," 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID), 2020. [Online]. DOI: 10.1109/CCGrid49817.2020.00-47
8. Hassan Iqbal, et al., "Characterizing the availability and latency in AWS network from the perspective of tenants," IEEE/ACM Transactions on Networking, 2022. [Online]. DOI: 10.1109/TNET.2022.3148701
9. Yimei Xu et al., "Optimization of Redis cluster systems based on load balancing mechanism," 2025 5th International Symposium on Computer Technology and Information Science (ISCTIS), 2025. [Online]. DOI: 10.1109/ISCTIS65944.2025.11065375
10. Yimei Xu, et al., "Research and implementation of Redis-based data hot-table caching mode," 2025 4th International Symposium on Computer Applications and Information Technology (ISCAIT), 2025. [Online]. DOI: 10.1109/ISCAIT64916.2025.11010306
11. Amit Gupta et al., "Optimizing performance of real-time big data stateful streaming applications on cloud," 2022 IEEE International Conference on Big Data and Smart Computing (BigComp), 2022. [Online]. DOI: 10.1109/BigComp54360.2022.00010
12. Yongping Luo, et al., "High-performance remote data persisting for key-value stores via persistent memory region," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2024. [Online]. DOI: 10.1109/TCAD.2024.3442992
13. Xiao Sun, et al., "Minimizing service latency through image-based microservice caching and randomized request routing in mobile edge computing," IEEE Internet of Things Journal, 2024. [Online]. DOI: 10.1109/JIOT.2024.3410546
14. Nagendar Yamsani et al., "Data-driven financial services: Advancing payment systems through fintech innovations," 2025 International Conference on Technology Enabled Economic Changes (InTech), 2025. [Online]. DOI: 10.1109/InTech64186.2025.11198274
15. Amazon Web Services, "Performance at scale with Amazon ElastiCache," AWS Whitepaper, 2026. [Online]. Available: https://docs.aws.amazon.com/pdfs/whitepapers/latest/scale-performance-elasticache/scale-performance-elasticache.pdf
16. Adi Emanuel Pinsky, et al., "Optimize Redis client performance for Amazon ElastiCache and MemoryDB," AWS, 2022. [Online]. Available: https://aws.amazon.com/blogs/database/optimize-redis-client-performance-for-amazon-elasticache/
17. Karthik Konaparthi and Vijay Joseph, "Measuring database performance of Amazon MemoryDB for Redis," AWS, 2022. [Online]. Available: https://aws.amazon.com/blogs/database/measuring-database-performance-of-amazon-memorydb-for-redis/
18. Manoj Mothy and K. Suganyadevi, "Real-time stream processing and analytics in cloud-based IoT systems," 2025 International Conference on Networks and Cryptology (NETCRYPT), 2025. [Online]. DOI: 10.1109/NETCRYPT65877.2025.11102564
19. Lorenzo Saino, et al., "Load imbalance and caching performance of sharded systems," IEEE/ACM Transactions on Networking, 2020. [Online]. DOI: 10.1109/TNET.2019.2957075
20. Maksim Vladimirovich Privalov and Mariya Valerevna Stupina "Improving web-oriented information systems efficiency using Redis caching mechanisms," Indonesian Journal of Electrical Engineering and Computer Science (IJEECS), 2024. [Online]. DOI: http://doi.org/10.11591/ijeecs.v33.i3.pp1667-1675