International Journal of Computational Research in Science and Technology
E-ISSN: 3108-0871, Volume 2 Issue 1
Jan-Feb 2026, PP 92-101

An Observability-Driven Study of A Two-Tier Cache with Lazy Eviction

Rahul Pokala
Independent Researcher
Bengaluru, India

Abstract - Caching systems try to manage and balance memory usage and speed by relying on eviction and promotion policies; however, most web developers find it obscure how cached data moves behind the scenes. This paper presents the design and evaluation of a two-tier cache system consisting of a HOT in-memory layer and a distributed COLD layer. In this system, old data is eliminated by TTL-based lazy-eviction, and data is immediately promoted from the cold-tier to the hot-tier upon access. To understand this in its entirety, an observability dashboard was developed to showcase hit patterns, TTL states, and promotion events in real time. Through controlled workload experiments, we observed how lazy eviction and promotion rules affect cache hit rates and data churn. The results demonstrate visualized cache behavior in real time, providing a detailed understanding of how eviction and promotion affect cache performance.

Keywords—Cache hit rate, Data churn, Distributed systems, Data promotion, In-memory storage, Real-time observability, TTL-based lazy-eviction, Two-tier caching.

I. Introduction

Caching is one of many techniques that are used in modern software systems to improve the performance by decreasing the latency and minimizing the repeated computations or data retrieval. Web applications including powerful big tech ecosystems and all popular cloud platforms rely extensively on multi-layer caching to mange high-volume data access and reduce load on the backend systems [1][4]. As traffic volume increases, cache design becomes a bottleneck, forcing trade-offs between memory overhead, latency, and data freshness. Traditional systems rely on policies like Least Recently Used (LRU)[8] or Time-To-Live (TTL)[8] to manage memory limits [2].
However, the internal lifecycle of data—how it expires or moves between tiers—is usually hidden from developers. This lack of transparency makes it difficult to diagnose stale data or performance spikes [4].
To handle these capacity constraints, modern architectures often use tiered layers to separate "hot" and "cold" data [3][7]. While this improves scale, it creates new problems regarding when to evict data and how to promote it between tiers.

This paper evaluates a two-tier system consisting of a HOT in-memory layer and a distributed COLD layer. The architecture uses TTL[8]-based lazy eviction to lower overhead and promotes entries from the cold tier to the hot tier immediately upon access. Instead of only measuring speed, this study treats observability as a primary feature. We developed a real-time dashboard to visualize hit patterns, TTL states, and promotion events. Through controlled workloads, we analyse how these policies affect hit rates and data churn, demonstrating how visualized telemetry reveals the trade-offs influencing total cache performance.

II. Background and Related Work

Caching is a core performance optimization technique employed across operating systems, databases, web applications, and distributed systems. By storing frequently accessed data

closer to the point of computation, caches reduce access latency, alleviate backend load, and improve overall system throughput [1]. As application scale and workload diversity increase, cache behavior increasingly influences system stability, cost efficiency, and user-perceived performance.

Cache Eviction and Expiration Strategies

Cache eviction strategies determine when cached data should be removed to reclaim limited memory resources. Classical policies such as Least Recently Used (LRU)[8] and Least Frequently Used (LFU) rely on access patterns to infer the relative importance of cached entries [2]. These policies aim to retain data that is likely to be reused while evicting less valuable entries. Time-To-Live (TTL)[8] expiration associates cached entries with a fixed lifespan to enforce data freshness [4]. A core design choice in TTL policies is the trade-off between proactive eviction, which uses background scanning to purge data, and lazy eviction, which only removes entries during an access attempt.

Lazy eviction is the standard for production systems to cut background CPU overhead and stop wasting cycles on data that stays cold [4]. The downside is that "dead" entries[10] sit in memory until a request hits them. This makes it difficult to track the actual cache state or get clean hit-rate data. Designing around these hidden memory footprints is a critical, yet often ignored, part of cache engineering.

Tiered Caching Architectures

Single-layer caches rarely satisfy both latency and capacity requirements. Instead, tiered architectures distribute data across layers with different performance profiles [3][7]. A standard design separates "hot" data for low-latency access from "cold" data stored in higher-capacity layers. In a two-tier setup, the top tier uses fast in-memory storage, while the lower tier handles bulk data at higher latency. Systems like Redis and Memcached demonstrate how this horizontal scaling manages concurrent throughput in production [3][4].

Distributed Caching and Consistent Hashing

Partitioning data across nodes requires a strategy that prevents massive data migration during cluster changes. Consistent hashing is the standard solution, mapping keys to nodes to minimize reallocation when the topology shifts [7]. This provides the stability needed for distributed cold tiers where predictable key ownership is required for scaling.

Promotion Policies and Cache Churn

Promotion rules dictate when data moves from cold to hot tiers. While moving active data upward reduces latency, aggressive promotion triggers "cache churn"—where data shifts between tiers so fast it negates performance gains. Since frequency-based promotion can fail during bursty traffic, these strategies must balance speed against memory pressure to prevent eviction cascades [2][7].

Observability in Cache Systems

Standard metrics like hit ratios and latency are high-level and often hide the internal lifecycle of cache entries. While observability is a core part of modern system design [6], it is rarely integrated into cache evaluation. Most studies rely on aggregate numbers rather than tracking individual expiration, eviction, or promotion events. This gap makes it difficult to diagnose why a cache is underperforming.

Positioning of This Work

This work combines a two-tier architecture with TTL-based lazy eviction and access-driven promotion. Unlike studies that focus only on speed, we treat observability as a core feature. By visualizing hits, TTL states, and promotion events in real time, we provide a direct look at the cache lifecycle and the trade-offs between different eviction and promotion rules.

International Journal of Computational Research in Science and Technology
E-ISSN: 3108-0871, Volume 2 Issue 1
Jan-Feb 2026, PP 46-62

III. System Architecture

This section details the two-tier cache design, focusing on its components, data movement, and the reasoning behind its mechanics. The system balances low-latency performance with scalability by pairing a HOT in-memory tier with a distributed COLD tier, using specific promotion and eviction rules to maintain data flow.

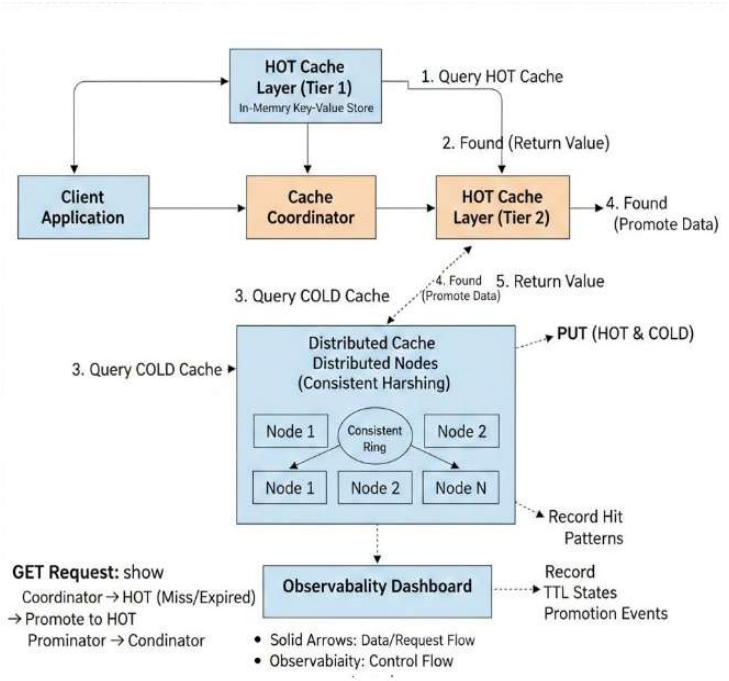

Fig. 1. Two-Tier Cache System Architecture.

Architectural Overview

The system splits storage into two functional layers[1]: HOT Cache (Tier 1): Local in-memory storage. Built for raw speed and immediate data retrieval. COLD Cache (Tier 2): A distributed node cluster. Built for high-volume storage of aging data that does not fit in Tier 1 RAM. All routing logic lives in a central coordinator. This component decides where to pull data from and when to move it between tiers based on the requests it receives.

HOT Cache Layer

The HOT cache serves as the primary access point. Each record contains the stored value, an insertion timestamp, and a TTL value. To minimize CPU jitter and background noise, the system uses lazy eviction. Instead of a background reaper process, the system validates TTLs only during a GET request. If an entry is found to be expired during access, it is purged immediately and treated as a miss.

Distributed COLD Cache Layer

Fig. 2. The COLD tier provides a failover and high-capacity storage area. It is distributed across multiple nodes using consistent hashing to map keys. This approach ensures that data is spread evenly across the cluster and minimizes the "shuffling" of data when nodes are added or removed. The

International Journal of Statistics and Mathematical Research Trends
E-ISSN: 3108-0871, Volume 2 Issue 1
Jan-Feb 2026, PP 92-101

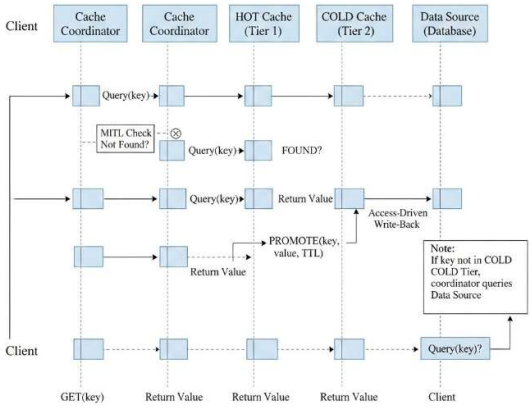

Fig. 2. Sequence Diagram for cache miss and data promotion

COLD tier uses significantly longer TTLs than the HOT tier to keep data available without exhausting expensive RAM.

Promotion Mechanism

Data moves up to the HOT tier only during a GET request. If the coordinator misses in Tier 1—or finds an expired key—it queries Tier 2. A hit in the COLD tier triggers an immediate write-back into the HOT tier before the value is returned to the user. This approach keeps the HOT tier restricted to active data. By promoting at the moment of access, the system minimizes Tier 1 memory pressure while keeping "warm" keys ready for the next request.

Eviction and Demotion Logic

The system relies on TTLs for implicit data aging. There is no active "demotion" process. Instead, as HOT data expires and stays un-accessed, it naturally falls out of the HOT tier. Since the COLD tier has a longer TTL[8], the data remains retrievable there. This creates a one-way flow where new/active data stays HOT, while inactive data eventually exists only in the COLD tier before final expiration.

Request Flow

The Tier Coordinator manages the following logic: GET Requests: Queries Tier 1. If valid, returns.

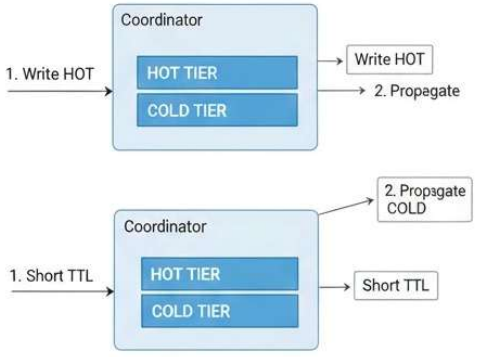

Fig. 3. PUT Request Flow : Dual-Write Mechanism.

If missing/expired, queries Tier 2. On a Tier 2 hit, it triggers a promotion to Tier 1 before returning the result. PUT Requests: Writes the data to the HOT cache and simultaneously propagates it to the COLD cache to ensure the entry exists in both layers from the start.

Design Rationale

The architecture focuses on three metrics: Efficiency: Lazy

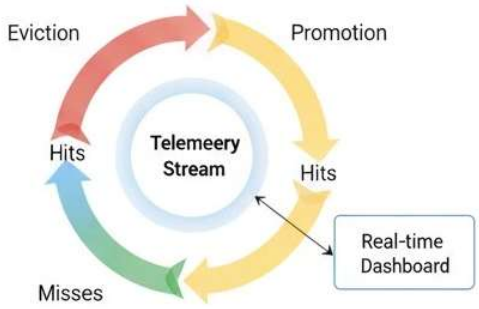

Fig. 4. Design architecture of Telemeery stream.

eviction and access-triggered promotion reduce background CPU cycles. Scale: Consistent hashing allows the COLD tier[13] to grow horizontally. Transparency: By centralizing logic in the coordinator, every state change (hit, miss, promotion, or lazy purge) is easily captured for the observability dashboard

IV. Experimental Evaluation and Results

This section assesses the two-tier architecture's performance[9] across varied workload patterns. The evaluation quantifies promotion efficiency, latency behavior, and the memory trade-offs of a lazy-eviction strategy. By leveraging the observability framework detailed in Section IV, we correlate tier-wise hit distributions with real-time "zombie" data accumulation to determine the system's operational efficiency.

Promotion Efficiency Under Different Workloads

Under a uniform workload, promotion efficiency remains limited (~45%) due to the absence of access locality. However, under Zipfian distributions[7], promotion efficiency improves significantly, reaching approximately 78% for s = 0.7 and over 90% for s = 1.2. The results demonstrate that the promotion mechanism is highly effective in workloads with skewed access patterns, where a small subset of keys dominates.

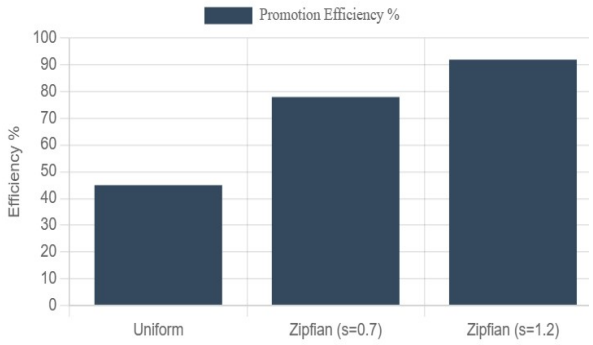

Fig. 5. Evaluates how effectively cache promotions occur under varying access distributions.

request frequency. Such patterns closely resemble real-world workloads, validating the design choice of dynamic promotion between cache tiers.

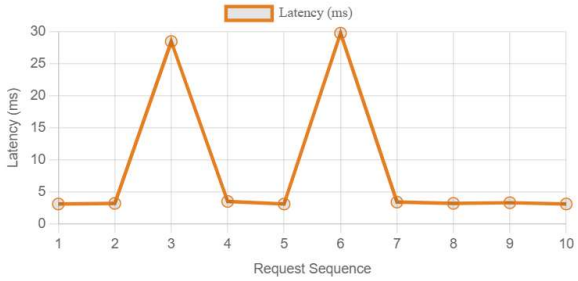

Fig. 6. Request latency behavior during cache promotion events.

Latency Impact of Promotion Events

This Figure-5 illustrates request latency across a sequence of cache accesses. Most requests exhibit low latency, corresponding to hot-cache hits. However, noticeable latency spikes occur at specific request indices, corresponding to cold-cache hits followed by promotion to the hot tier.

These spikes represent the overhead of fetching data from the cold cache and updating the hot cache. Once promotion occurs, subsequent accesses return to low-latency performance. This confirms that promotion overhead is transient and amortized over future accesses.

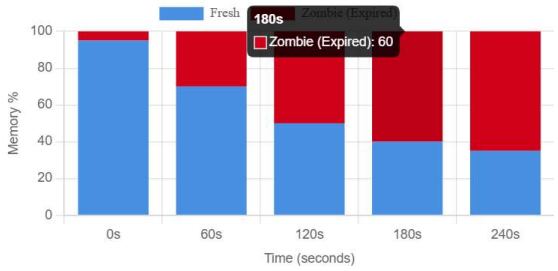

Fig. 7. Memory Utilization and zombie data accumulation over time.

Zombie Data Accumulation and Memory Utilization

This Figure-6 shows the evolution of memory utilization in the hot cache over time, distinguishing between fresh entries and zombie (expired but not yet evicted) entries. Initially, most cache entries are fresh. As time progresses, expired entries accumulate due to the system's lazy expiration strategy.

Zombie occupancy increases steadily, reaching approximately 60% at 180 seconds. These entries are eventually removed upon access rather than through background scanning. This design avoids continuous eviction overhead and improves runtime efficiency at the cost of temporary memory overhead.

This figure shows how cache hits break down between the two cache tiers. About 82% of requests come straight from the hot cache. The other 18% end up in the cold cache, and those usually get promoted afterward. So, the hot cache really pulls its weight. It hangs onto the most popular keys, which proves that hot-set isolation works as planned. Meanwhile, the cold tier handles the bigger workloads without putting too much pressure on the hot cache.

Tiered Cache Hit Distribution

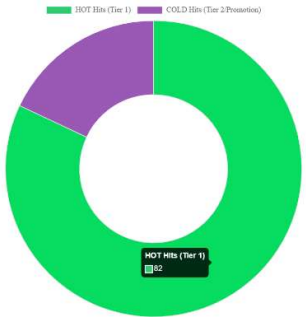

Fig. 8. Distribution of cache hits across hot and cold tiers.

Overall Observations
In every experiment, the system adapts well to different workloads. When access patterns get skewed, promotion gets faster. Latency stays in check and spreads out evenly. Zombie data does not pile up. Most requests hit the hot cache. All this shows the new architecture delivers low latency, uses memory efficiently, and offers strong observability. It is ready for real-world distributed systems.

V. Design Trade-Offs and Limitations

This section discusses the architectural trade-offs inherent in the proposed two-tier cache system and highlights limitations observed during experimental evaluation.

Resource Contention: CPU vs. Memory
The implementation of lazy eviction optimizes for CPU stability by removing background reaper threads[6]. This shifts the computational burden from a continuous O(n) scanning process to a per-request O(1)—validation. However, experimental data confirms a corresponding spatial penalty; as stale data is only purged upon access, the HOT tier maintains a higher steady-state memory footprint than proactive systems. This architecture is therefore most effective in CPU-bound environments where memory headroom is available to accommodate transient "zombie" entries.

Promotion Latency and Locality Requirements
The 28.5ms latency penalty incurred during COLD-to-HOT migration—defined here as the Promotion Tax—requires high temporal locality to justify the I/O overhead. The system uses a conditional promotion policy, where only successful Tier 2 hits trigger a write-back to Tier 1. While this minimizes churn for one-off misses, workloads with low reuse (e.g., sequential scans) will experience elevated tail latency without the benefit of amortized HOT-tier hits.

Deterministic Eviction and Working Set Shifts
By utilizing a fixed-size HOT tier with access-order eviction, the system maintains deterministic performance bounds. The trade-off is an increased susceptibility to eviction churn when the active working set exceeds Tier 1 capacity. In these scenarios, the COLD tier functions as a secondary overflow, though the reliance on consistent hashing across nodes means that retrieval remains subject to network round-trip times (RTT) rather than local memory speeds.

Consistent Hashing and Node Heterogeneity
While consistent hashing ensures horizontal scalability, the current implementation assumes node homogeneity. The mapping of keys to the COLD tier cluster does not account for variations in individual node throughput or latency. In a heterogeneous cluster, this lack of load-aware routing could result in localized hotspots. Incorporating weighted virtual nodes or dynamic replication would be necessary to ensure load balancing across non-uniform hardware.

Instrumentation Overhead
The granularity of the observability layer—tracking atomic TTL—states and promotion sequences—introduces synchronization overhead on the critical request path. While necessary for the validation of this study, production-scale deployments would require decoupled telemetry. Moving from synchronous atomic updates to asynchronous buffer aggregation or probabilistic sampling would mitigate the performance ceiling observed during high-concurrency stress tests.

Summary of Design Constraints
The architecture represents a deliberate selection of deterministic latency and structural simplicity over adaptive complexity. By favoring lazy-purge mechanisms and fixed-tiering, the system provides a stable environment for analyzing tiered-cache dynamics without the noise introduced by proactive background processes[10].

VI. Conclusion and Future Work

This research established a two-tier cache architecture that decouples local low-latency access from distributed capacity. The implementation demonstrates that a promotion-driven data lifecycle effectively captures temporal locality while providing granular visibility into state transitions. Experimental data validates that while lazy eviction eliminates O(n) CPU overhead, it necessitates a memory headroom to accommodate stale data accumulation. By transforming cache internals—specifically TTL decay and promotion sequences—into observable metrics, this architecture provides a deterministic framework for evaluating multi-tier storage dynamics.

Technical Contributions
Tiered Data Orchestration: A dual-layer architecture mediated by a coordinator that isolates latency-critical operations from distributed storage retrieval.

Promotion Efficiency Analysis: Quantitative validation of the "Promotion Tax" amortization across Zipfian workloads.

Quantification of Lazy-Eviction Trade-offs: Identification and measurement of the "spatial tax" (zombie data) inherent in non-proactive cache cleaning.

Instrumentation Framework: A telemetry pipeline capable of visualizing non-opaque cache behaviour, including real-time TTL state and promotion timelines.

Future Work
The following areas are identified for further technical optimization:
Hybrid Reclamation: Implementation of probabilistic background scanning to reduce memory occupancy without inducing CPU jitter.

Selective Promotion Logic: Development of frequency-based admission filters for Tier 1 to prevent "promotion churn" in low-locality workloads.
Fault-Tolerant Hashing: Integration of replication factors and health-aware routing within the COLD tier to address node heterogeneity and failures.

Decoupled Telemetry: Transitioning to asynchronous metric aggregation to eliminate synchronization overhead on the critical request path.

Summary
The architecture demonstrates that transparency in cache design does not preclude operational efficiency. By making internal state transitions measurable, the system enables precise performance tuning and provides a verifiable baseline for distributed data management research.

VII. Conclusion

In this paper, we built and tested a two-tier caching system that you can observe in action. There is a speedy, in-memory HOT layer, as well as a distributed COLD layer at the bottom. Instead of viewing the cache as an opaque system, we integrated TTL-driven lazy eviction lazy eviction governed by TTL with instant promotion instant promotion whenever someone grabs datagrams data. This way, you can see how stuff moves and changes in the cache over time.

What sets this work apart is how we baked real-time observability right into the cache. We tracked metrics like HOT and COLD hit patterns, how long promotions take, what is happening to TTLs, and even how so-called "zombie" data piles up. We showed all this off on a custom dashboard. Suddenly, cache behaviour was not a mystery—these visuals let us run experiments and really dig into the trade-offs between memory use, speed, and keeping data fresh.

The experiments tell a clear story. Lazy eviction keeps overhead low at first, but you end up with a bunch of zombie data hanging around. When you promote data from the cold layer, you get a quick spike in latency, but after that, the HOT cache smooths things out with faster hits. By handling events asynchronously, we separated the observability from the main data flow, so the system stayed responsive even when things got busy.

The key point: It's not possible to evaluate the performance of the cache by hit rates alone. You need a window into things like eviction patterns, the price of promotion, and how data ages inside the cache if you want a system that's both efficient and predictable. Our design and the tools we built give you that visibility, laying the groundwork for transparent, production-level caching systems. And there's plenty of room to grow—think smarter TTL tuning, predictive eviction, or even analytics across distributed caches.

References

1. A. S. Tanenbaum and M. van Steen, Distributed Systems: Principles and Paradigms, 2nd ed., Pearson Education, 2007.
2. P. Cao and S. Irani, "Cost-aware WWW proxy caching algorithms," Proceedings of the USENIX Symposium on Internet Technologies and Systems, 1997.
3. B. Fitzpatrick, "Distributed caching with Memcached," Linux Journal, vol. 2004, no. 124, 2004.
4. Redis Ltd., "Redis: An in-memory data structure store," [6] B. Varghese et al., "Challenges and opportunities in edge computing," IEEE Computer, vol. 52, no. 8, pp. 32–40, 2019.
5. B. M. Maggs and R. K. Sitaraman, "Algorithmic nuggets in content delivery," ACM SIGCOMM Computer Communication Review, vol. 45, no. 3, pp. 52–66, 2015.https://redis.io, accessed 2026.
6. B. Goetz, "Java Concurrency in Practice," Addison-Wesley, 2006.
7. S. Jiang and X. Zhang, "Making LRU-friendly caches more robust," Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS), 2002.
8. N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache," FAST, USENIX, 2003.

9. B. Gregg, Systems Performance: Enterprise and the Cloud, 2nd ed., Pearson Education, 2020.
10. C. E. Killian et al., "Life, death, and the critical transition: Finding liveness bugs in systems code," NSDI, USENIX, 2007.
11. Amazon Web Services, "Amazon Simple Queue Service (SQS): Developer Guide," https://docs.aws.amazon.com/sqs, accessed 2026.
12. Amazon Web Services, "Amazon Simple Queue Service (SQS): Developer Guide," https://docs.aws.amazon.com/sqs, accessed 2026.
13. J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," Communications of the ACM, vol. 51, no. 1, pp. 107–113, 2008.
14. E. Kreps, "The log: What every software engineer should know about real-time data's unifying abstraction," LinkedIn Engineering Blog, 2013.