10.48047/jocaaa.2026.35.03.32

Beyond the Millisecond:Engineering Financial Engines for Single-Digit Latency at eCommerce Scale

Abdul Basit Iqbal
Independent Researcher, USA

Abstract

Modern hyperscale eCommerce demands have rendered traditional database-centric models obsolete. The digital economy now requires deterministic single-digit millisecond response times to maintain conversion rates. Legacy financial systems frequently expose systemic weaknesses, including centralized bottlenecks and I/O-related latencies that risk cascading failures across global infrastructures. This article examines the transition to database-less financial engines that leverage in-memory versioned reference data, shifting the transaction lifecycle from an I/O-bound disk process to a high-velocity, CPU-bound computational process. The architecture does not have a single central primary relational database. It uses distributed caching techniques such as read-through, write-behind and refresh-ahead to achieve data consistency and order-of-magnitude throughput improvements. By using an architecture of separate cells, the workload is split into partitions. These cells contain the effect of a potential failure. The article also leverage a serverless orchestration and state-machine based workflow design pattern that minimizes our overhead to produce a hardened checkout pipeline, where low single digit millisecond latency maps to both our revenue and our systemic high availability.

Keywords: Database-Less Architecture, Distributed Caching, Cell-Based Isolation, Serverless Orchestration, Sub-Millisecond Latency

1. The Performance Imperative in Modern eCommerce Financial Systems

With the maturity of the digital economy, system performance is no longer only a technical specification, but the main driver of consumer behavior. This might explain why financial transaction speed has become a first-class citizen of shopping experiences in the fast-paced world of eCommerce where trust and revenue are at stake. Per the principle of tolerable waiting time, users have specific tolerable waiting limits for a system, and the perceived competence and trustworthiness of a site are strongly diminished if users have to wait longer [1]. As a system approaches single-digit millisecond latency, users are more likely to remain in a flow state—a cognitive condition in which friction and decision fatigue are virtually eliminated [1].

Beyond user experience, controlled experiments on web systems have demonstrated a measurable correlation between response latency and commercial outcomes. Studies indicate that even modest increases in page load time produce statistically significant reductions in user engagement and conversion rates [2]. If 100ms was considered an acceptable performance threshold, hyperscale platforms today are competing in the single digit millisecond range for the marginal improvements that translate into market leadership.

This is most keenly felt at the checkout pipeline where steps such as tax calculations and fraud checks need to be done at single digit millisecond precision or cart abandonment will occur. It is worth noting that although this is the last transactional step, any delay starts to break user momentum. The architectural decisions behind financial engines carry a direct bottom-line impact, since performance degradation during high-traffic periods has been shown to produce substantial revenue losses within

10.48047/jocaaa.2026.35.03.32

compressed timeframes [2]. Moving toward deterministic, single-digit millisecond response times is therefore no longer an optimization—it is a financial imperative.

2. Systemic Vulnerabilities in Database-Dependent Architectures

Financial orchestration systems that rely on a conventional relational database management system tend to be fragile. The availability and performance of the storage layer directly constrain the availability and operational performance of every service that depends on it. This tight coupling creates a single point of failure at the database layer, through which cascading failures can propagate across the entire application regardless of how well-designed individual service components may be.

These systemic vulnerabilities were evident during Amazon's 2018 Prime Day, when error messages became visible to shoppers after the platform was overwhelmed with traffic beginning at 3 p.m. ET on July 16, 2018, and persisting for approximately two hours. Analytics firm One Click Retail estimated losses of $1.2 million per minute during the disruption, even as the full 36-hour Prime Day event generated $3.5 billion in total sales. The incident illustrated how database-intensive architectures can fail under high concurrency, even when substantial infrastructure resources are available [3].

More recent failures confirm that these vulnerabilities persist even in advanced cloud environments. In an October 2025 disruption of AWS US-EAST-1, a DNS misconfiguration triggered cascading failures across a wide range of tightly coupled managed services [4]. The incident lasted approximately 15 hours and affected DynamoDB, EC2, NLB, Lambda, ECS, EKS, and Amazon Connect. The failure unfolded in three waves: the initial DNS record deletion on October 19 at 11:48 p.m. PDT; a manual recovery attempt on October 20 at 2:25 a.m. PDT that caused the DropletWorkflow Manager to fail while reclaiming thousands of leases; and prolonged configuration backlogs in the networking backend that persisted for hours after lease recovery at 5:28 a.m. PDT. This event illustrates a critical architectural risk: when financial engines depend on externally managed services, a failure in any shared underlying component can render the entire application stack unavailable, regardless of how resilient individual services appear in isolation [4].

Both incidents reinforce the central argument of this article. When a financial orchestration engine depends on database round-trips to process transactions, it inherits not only the performance constraints of the database but also its failure modes. Database failure causes all dependent application stacks to fail—an outcome that conventional high-availability solutions such as replication and failover cannot fully address.

10.48047/jocaaa.2026.35.03.32

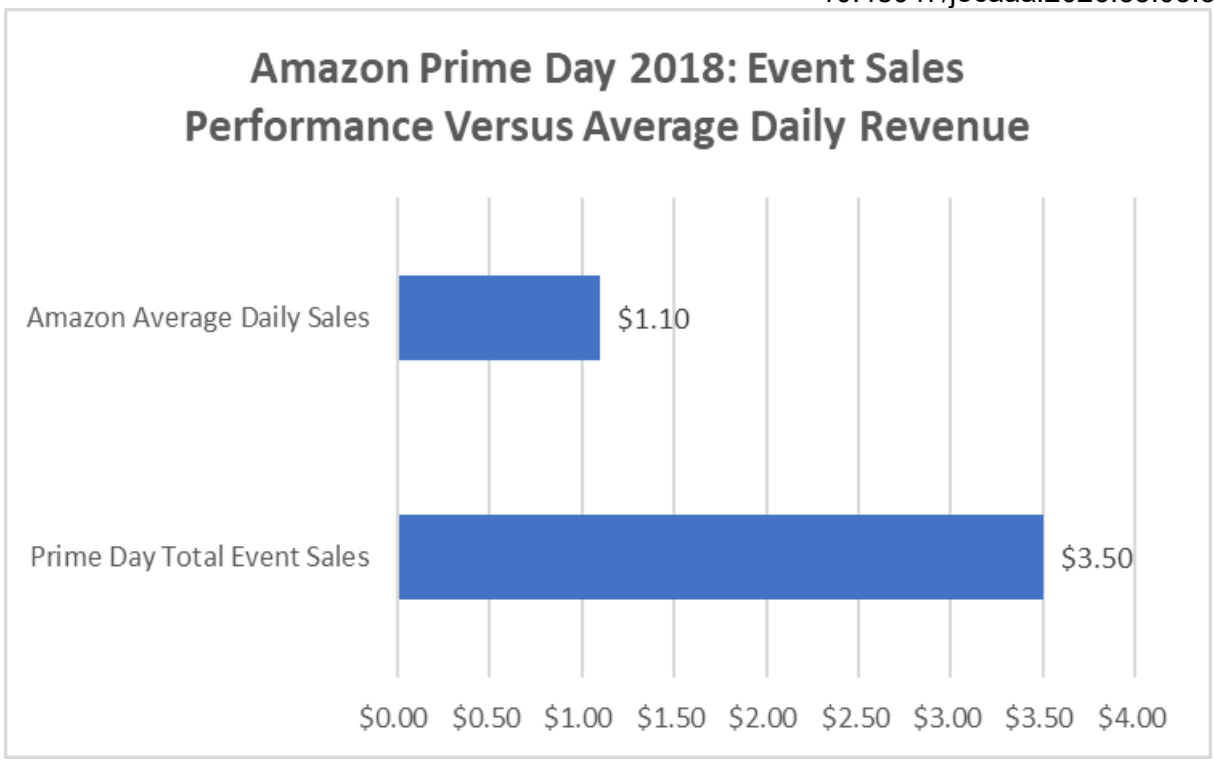

Fig 1: Prime Day Sales Impact: 36-Hour Event Performance Compared to Daily Baseline [3, 4]

3. Methodology

This research employs conceptual and analytical methodology to examine the transition in the architectural design model from database-centric orchestration to modern database-less in-memory solutions. In lieu of primary empirical experimentation, the study relies on synthesis, interpretation, and comparative analysis of publicly available peer-reviewed articles, large-scale industry incidents, and existing aspects of distributed systems research literature. The method consists of a latency measurement framework, a storytelling metaphor for comparison, a latency model based on mathematics, and an architectural evaluation criterion.

The units of measure and the extent of performance measurement in use are the following: end-to-end transaction latency is the sum of the round-trip time of the network request, the disk I/O time if the requested data cannot be fetched from the buffer pool, and the query execution time for query planning and execution. This form of decomposition allows the bottleneck components of the database-centric architecture to be analyzed and acts as a metric to provide understanding into the reduction in latency that occurs when disk IO bound data retrieval is replaced by in-memory cache. Financial processing is considered in three domains, retail payment processing, capital markets execution facilities, and fraud detection pipelines, whose respective requirements are referred to as the thresholds against which the architecture is modeled [5].

Within some idealized assumptions, two architectural classes are compared in the comparative benchmarking story: baseline database-centric architecture and cache-enabled database-less architecture. The baseline database-centric architecture is a conventional architecture in which for every lifecycle event of a transaction, there is at least one synchronous round-trip to the relational data store. In an alternative architecture, the round-trip is replaced with a lookup in a local in-memory cache, which is kept coherent in a distributed way. Each architecture's performance characteristics can be explained in terms of

10.48047/jocaaa.2026.35.03.32

published workload characteristics and systems studies enabling a structured side-by-side comparison that takes into account throughput, cache hit rates and the offload rate from the database. In this way, the contribution of the caching layer can be isolated from variation in other architecture variables in the system.

The mathematical latency model is an expression that defines the relationship between architecture and response time. The total transaction latency in a database-dependent system is equal to the latency of the network plus the retrieval latency of the disk plus the latency of query processing. In a cache architecture, the latency from disk accesses is removed in the event of a cache hit and the latency is therefore determined by fetching and processing time of the object. This model is evaluated for different values of cache hit ratio, the fraction of requests that are served from cache, and determine the end-to-end latency sensitivity to cache performance. This in turn models, much as in the distributed systems literature, how latency improvements relate to measurable architectural parameters [5][6].

The qualitative architectural evaluation criterion gives a qualitative framework for assessing three design patterns that are studied in the following sections: in-memory state, cell-based fault isolation, and serverless workflow orchestration. The four criteria on which the three design patterns are assessed are latency, consistency guarantees, fault isolation, and operation scalability. This criterion-based approach is based on established practices in the evaluation of software architecture and allows to formally derive design recommendations.

Methodology Component	Description	Key Elements
Latency Measurement Framework	Defines the units and boundaries of performance assessment used throughout the study	Network round-trip time; disk I/O time; query execution time (planning and processing); evaluated across retail payment processing, capital markets execution, and fraud detection pipelines
Comparative Benchmarking Narrative	Contrasts two architectural classes under controlled conceptual conditions to isolate the contribution of the caching layer	Baseline database-dependent architecture vs. cache-enabled database-less architecture; throughput capacity; cache hit ratios; database offload rates
Mathematical Latency Model	Formalizes the relationship between architectural choices and observed response times	Total latency = network latency + disk retrieval latency + query processing latency (DB-dependent); Total latency = cache retrieval time + processing time (cache-enabled); hit ratio parameter evaluated across a range of values
Architectural Evaluation Criterion	Provides the qualitative framework for assessing the three design patterns examined in subsequent sections	Four evaluation criteria: latency impact, consistency guarantees, fault containment capability, and operational scalability; applied to in-memory state management, cell-based fault isolation, and serverless workflow orchestration

10.48047/jocaaa.2026.35.03.32

Table 1: Methodology Components and Analytical Framework [5, 6]

4. Deconstructing Relational Bottlenecks Through Latency Modeling

The ACID properties of relational database management systems—atomicity, consistency, isolation, and durability—are achieved through complex locking mechanisms and disk-bound I/O operations that are incompatible with sub-millisecond performance targets. These properties make relational databases suboptimal for high-concurrency financial transaction processing [5]. Real-time financial services operate under strict latency requirements across distinct processing domains, and meeting these requirements demands architectural approaches that eliminate the latency components introduced by synchronous database interaction [5].

Transaction latency in database-dependent systems comprises three measurable segments: the network round-trip to the database, the disk I/O penalty incurred when data is absent from the buffer pool, and the time required to plan and execute the query. Together, these segments render sub-millisecond transaction completion infeasible in conventional database-backed architectures [6]. By contrast, distributed cache systems provide data access at substantially lower latency, representing an order-of-magnitude improvement over disk-backed query execution [6]. Workload studies have found that cache layers provide end-to-end transaction latency improvements, while distributed in-memory architectures provide throughput not matched by architectures bound by databases [5].

Latency is reduced by minimizing database round-trips in critical transaction paths. Highly optimized systems can achieve a very high cache hit ratio, considerably reducing the time spent in the database [6]. Database-less architectures remove all of the network and disk I/O involved in transaction processing by storing versioned, read-only copies of the reference data used in calculations either in application memory or in high performance sidecar processes co-located on the same servers as their respective applications. This transforms financial engines from being I/O-bound processing engines to CPU-bound processing engines, treating reference data (like tax tables and compliance logic) as static resources and assets built into the application rather than mutable state in the database.

Latency Component	Database-Dependent Architecture	Cache-Enabled Architecture	Impact of Elimination
Network Round-Trip	Present on every transaction; latency scales with data store distance and concurrency load	Eliminated on cache hit; local memory access only	Removes the dominant latency contributor in high-concurrency environments
Disk I/O	Incurred when requested data is absent from the buffer pool; compounds under concurrent access	Eliminated; reference data maintained in application memory or local sidecar processes	Removes unpredictable I/O wait times that prevent sub-millisecond completion
Query Planning and Execution	Required for every database interaction; adds processing overhead per transaction	Eliminated; reference data accessed directly as pre-compiled in-memory state	Reduces per-transaction CPU overhead and removes query optimizer dependency
Total Transaction Latency	Sum of all three components; renders sub-millisecond completion	Reduced to cache retrieval time and processing time on cache hit	Order-of-magnitude latency improvement; shifts processing model

10.48047/jocaaa.2026.35.03.32

	infeasible		from I/O-bound to CPU-bound
Throughput Capacity	Constrained by database concurrency limits and disk I/O throughput	Scales with memory capacity and cache hit ratio; sustains substantially higher operations per second	Enables throughput levels that database-bound architectures cannot sustain

Table 2: Latency Component Analysis: Database-Dependent vs. Cache-Enabled Architecture [5, 6]

5. Localized Immutable State and Distributed Caching Architectures

Decoupling the financial engines themselves from the database, while synchronizing reference data across the various active fleets of applications, requires a high degree of coordination. The architecture of the engines themselves makes use of distributed cache implementations that have been designed from the ground up to be massively parallel, with each financial engine maintaining its own local cache of the reference data and propagating changes throughout the fleet. Published performance figures show that effective usage of the distributed cache provides lower latencies than reading from the database, as well as sufficient throughput for high-volume financial transactions [6].

Current caching architectures use several standard cache consistency and freshness patterns. For example, in the read-through pattern, the application accesses the cache directly as a primary data store. On a cache miss, the cache reads the necessary data from the authoritative data store and returns it to the application [7]. Write-through caching is a form of strong consistency, where all data updates to the cache are synchronously replicated to the backing store [7]. This incurs a higher write latency. To reduce write latencies when strong consistency is not required, write-behind caching (also known as write-back caching) does not synchronously write back before returning. Instead, it writes back data in the background. Write-behind caches achieve very high write throughput, but can result in data loss if a cache node fails before writing data to non-volatile storage [7].

Workloads such as financial service quotation engines often use refresh-ahead caches. Refresh-ahead caches preemptively refresh cached entries just before expiration, thereby avoiding cache misses under most workloads [7]. In tax orchestration systems where reference data is updated according to a known schedule, refresh-ahead caches ensure that all calculation engines can work on the latest data without performance penalties. Well-optimized systems with high cache hit rates can reduce pressure on the underlying database [6].

More scalability for advanced distributed systems may come from hardware-aware scheduling and interconnects with high bandwidth, making use of modern optical interconnects to transfer data in a low-latency fashion between distributed financial engines [8]. The hardware and software optimizations enable financial engines to achieve the sub-millisecond latency required for high-volume, high-speed transactions.

10.48047/jocaaa.2026.35.03.32

Attribute	Database-Dependent Model	Distributed Cache-Enabled Model
Data Storage Location	Centralized relational data store; accessed via network round-trip	Local application memory or high-speed sidecar process per engine instance
Data Access Mechanism	Synchronous query execution per transaction	Direct in-memory lookup; cache retrieves from authoritative store only on miss
Reference Data Update Propagation	Updates applied to central store; all engines reflect changes on next query	Updates propagated across engine fleet via distributed cache invalidation or refresh-ahead scheduling
Consistency Guarantee	Strong consistency enforced by ACID properties	Consistency model varies by caching pattern (strong, eventual, or predictive)
Fleet-Wide Synchronization	Implicit; all engines query the same central store	Explicit; requires coordinated cache invalidation or scheduled refresh cycles
Suitability for Periodic Updates	High; all engines receive updates immediately on next query	High for refresh-ahead pattern; reference data refreshed on known schedules without cache miss penalty
Latency Under High Concurrency	Degrades as concurrent queries contend for database resources	Stable; local memory access is unaffected by fleet-wide concurrency

Table 3: Reference Data Management: Database-Dependent vs. Distributed Cache-Enabled Model [6, 7]

6. Cell-based Architecture for Fault Isolation and Availability

At the scale of hyperscale eCommerce, being able to contain the damage of a failure is as important as preventing it. Cell-based architecture (CBA) is where the workload is partitioned in such a way that each part can operate independently of the others, similar to a ship's watertight bulkheads allowing flooding to be isolated to a single area [9]. Each cell encapsulates a complete instance of the application stack, including dependencies, services, and data caching layers, allowing it to be treated independently.

This cellular model has a few architectural pieces. The cell router receives the requests and has some basic routing logic that routes them to the correct cell based on the partition key [10]. Usually either keys (e.g., customer identifiers) or geography (such as city or region) are used as partition keys to deterministically (and consistently) route requests to certain cells to minimize inter-cell communication, and with that, minimize weakening of the isolation [10]. A separate control plane is used for provisioning resources, monitoring capacity of each cell, migrating tenants, etc. The control plane is completely decoupled from the data plane to not interfere with processing of transactions [10].

Because of the way traffic is distributed in a cell-based architecture, it has a good availability model for financial applications, as a single cell failure can only affect at worst a share of traffic corresponding to the affected cell. For example, in the case of deployment on 10 cells, one cell failure hits 10% of users, leaving 90% unaffected [10]. In studying the failure behavior of cloud systems, fault isolation boundaries

10.48047/jocaaa.2026.35.03.32

have been found to be among the most effective mechanisms to contain cascading outages in distributed infrastructure systems [9]. A cell architecture converts a systemic failure into a local event and an existential availability risk into a bounded availability event.

Cell-based architectures allow horizontal scaling, which is less complex than the typical vertical scaling seen in monolithic architectures. Instead of replacing existing systems with larger systems, the operators provision more cells as the capacity demand increases. This constant-complexity scaling is especially well-suited for financial systems that require resilient behavior and high availability under peak loads [9].

7. Serverless Orchestration and Infrastructure Automation at Scale

These database-less architectures on distributed fleets of servers require orchestration and automation for deployment and operations. Major cloud providers offer infrastructure-as-code ecosystems to help organizations manage the infrastructure programmatically. The serverless model is particularly well-suited to financial engines since it abstracts the complexity of the underlying computing architecture, and supports event-driven processing of multi-step workflows [11].

AWS Step Functions orchestrates multiple services into asynchronous, multi-step workflows by using visual workflow definitions that specify the ordering and parallelism of tasks, as well as error handling and retry behavior, across heterogeneous execution environments [12]. Step Functions also enables developers to build distributed applications as state machines, where each state represents a discrete self-contained unit of computation that uses built-in error handling and branching logic [12]. In systems with global engine fleets that propagate reference data updates, Distributed Map is useful for fan-out at scale as child workflows are spawned in parallel when consistent updates are needed across independent cells [11].

To avoid the above limitations of the iterative pattern, the use of Distributed Map state allows for splitting data into sub-units that can be processed in parallel, leading to a scalable and near-linear growth in speedup relative to size [11]. The declarative model also allows for the correct ordering of multiple stages in the event of a complex or multi-step workflow, and automatic retries or compensatory actions in transient failures.

Step Functions provides two types of workflows, standard workflows and express workflows. Standard workflows are designed to have exactly-once execution semantics for long-running tasks, while express workflows are designed for high throughput, at-least-once event processing [12]. This dual model allows organizations to align guarantees around the execution of the workflow with the particular consistency and performance requirements of their use case.

In this context, serverless compute and state machine orchestration together form an 'infrastructure as code' operational model in which the infrastructure building, application deployment, data change detection, and system maintenance workflows are expressed as code. This reduces operational toil, promotes consistency in the way fleet-wide operations are performed, and drives down operational cost for scaling processing capacity [11].

8. Discussion

While the previous section makes the case for database-less financial engines, there are non-trivial trade-offs when implementing such an architecture. Below the article briefly outline the primary pain points of each of the three pillars of this architecture (in memory state management, cell based fault isolation and serverless orchestration), and under what circumstances they do and do not manifest themselves.

10.48047/jocaaa.2026.35.03.32

The major trade-off is between performance and consistency. Moving reference data from a persistent SQL database to distributed in-memory caches can considerably improve performance. However, the strong consistency guarantees present in ACID-compliant databases are no longer guaranteed. Write-behind caching patterns present the lowest write latency, but leave a potential gap window for data loss when a cache node fails before persistence in the background completes [7]. Given the nature of financial systems responsible for tax calculation, compliance logic, etc, even a stale reference data window potentially leads to an erroneous transaction. This requirement can be met through careful caching policies (e.g. write-through or refresh-ahead for regulated data and cache validation for detecting and resolving inconsistencies before they reach downstream components) with the caveat that the consistency-latency trade-off must be made explicitly for each data type rather than at an application-wide level.

Another downside to the cell-based architecture is operational complexity. While the cell's isolation from the rest of the system reduces the blast radius of an instance failing and thereby increases reliability, it also increases the operational surface area of the system. Maintaining, monitoring and provisioning each cell separately may incur a higher infrastructure overhead. Mature automation tooling is required to avoid linear increases in operations costs. Relocating tenants due to load imbalance or the need to decommission an underperforming instance can make workload transfer more complex for the same reason; the state must be moved without interrupting any in-flight transactions, and securely transferred to the other cell. The cell router still acts as a point of concentration, since its failure would prevent routing to the cells, even if they were otherwise deployed in a fault-tolerant manner.

Serverless orchestration has an additional limit, the cold start time when a serverless function is invoked after it has been inactive. The cold start time may violate the sub-millisecond performance goal that induced developers to use serverless in the first place, particularly for workflows that are triggered by sporadic or bursty load [11]. Mitigations for cold start, such as provisioned concurrency, add costs that may offset the economic benefits of serverless. Vendor lock-in is also a concern, with state machine specifications encoded using proprietary workflow languages typically tightly coupled to a particular cloud provider's orchestration layer, making it more difficult for institutions with multi-cloud or hybrid cloud strategies to switch providers.

Cost and infrastructure overhead affect all three pillars. For example, distributing a large amount of reference data across the engine fleet takes up a lot of memory, and in-memory persistence has a far higher per-unit cost than disk storage. As variety and volume of reference data grows (e.g. tax tables, fraud rules, compliance rules, pricing operations, etc.), so does the amount of memory used per engine instance. Organizations need disciplined data lifecycle management processes (cache eviction policies, tiered caching, etc.) to ensure that the memory cost does not negate the economics of the architecture.

In-memory financial data stores also have security or compliance considerations, as data on volatile memory may be more susceptible to memory inspection attack vectors than encrypted disk-based data. Legal and compliance requirements around encryption, access control, and audit logging for financial data (such as PCI-DSS and SOC 2) must also be applied to data in in-memory data stores and distributed cache layers. It may be difficult to consistently implement uniform encryption-in-transit and encryption-at-rest policies across all caches and also consistently apply strict role-based access controls on access to cached reference data like for persistent data stores. Considerations and costs for these concerns should form a part of the initial adoption planning phase.

Taken together, these trade-offs do not invalidate the architectural approach advanced in this article, but instead define the conditions under which it is the right solution. Organizations with productive DevOps practices, a commitment to data governance, and workloads with high transaction loads and well

10.48047/jocaaa.2026.35.03.32

understood reference data update cycles are best positioned to enjoy the latency and availability benefits offered by the architecture, while managing the implementation challenges outlined above.

Conclusion

Database-less financial orchestration represents a significant architectural advance for hyperscale eCommerce environments where relational databases can no longer satisfy performance requirements. By eliminating database round-trips and adopting local immutable state management, financial systems achieve the latency reductions necessary to protect conversion rates and revenue. Distributed caching provides the consistency guarantees and parallel access patterns required to synchronize reference data across globally deployed engine fleets, delivering response latency that is orders of magnitude lower than database-bound solutions. Cell-based architecture improves availability by containing failure blast radius and enabling systems to degrade gracefully rather than fail catastrophically. Serverless orchestration transforms infrastructure management into a deployable software capability that scales elastically without proportional increases in operational complexity. These three architectural models—in-memory state, cellular isolation, and event-driven automation—form the blueprint for mission-critical, low-latency financial systems in which every microsecond of performance improvement translates directly into competitive advantage. As eCommerce latency requirements continue to tighten, database-less architectures will increasingly define the standard for organizations whose business outcomes depend on the speed of financial transactions.

References

[1] Fiona Fui-Hoon Nah, "A Study on Tolerable Waiting Time: How Long Are Web Users Willing to Wait?" Behaviour & Information Technology, 2003 [Online]. Available: https://www.researchgate.net/publication/220893869_A_Study_on_Tolerable_Waiting_Time_How_Long_Are_Web_Users_Willing_to_Wait

[2] Ron Kohavi et al., "Controlled experiments on the web:
survey and practical guide," Data Min Knowl Disc, 2009. [Online]. Available: https://robotics.stanford.edu/~ronnyk/2009controlledExperimentsOnTheWebSurvey.pdf

[3] Marcia Kaplan, "Amazon's 2018 Prime Day Sets Record Despite Glitches," Practical Ecommerce, 2018. [Online]. Available: https://www.practicalecommerce.com/amazons-2018-prime-day-sets-record-despite-glitches

[4] Tahir, "Complete Analysis of the AWS US-EAST-1 Service Disruption and DNS Failure," Medium, 2025. [Online]. Available: https://medium.com/@tahirbalarabe2/complete-analysis-of-the-aws-us-east-1-service-disruption-and-dns-failure-299cbaa1f782

[5] Abdullah Talha Kabakus and Resul Kara, "A performance evaluation of in-memory databases," Journal of King Saud University - Computer and Information Sciences, Volume 29, Issue 4, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1319157816300453

[6] Berk Atikoglu et al., "Workload Analysis of a Large-Scale Key-Value Store," ResearchGate, 2012. [Online]. Available: https://www.researchgate.net/profile/Eitan-Frachtenberg/publication/254461663_Workload_analysis_of_a_large-scale_key-value_store/links/5489e9720cf214269f1abf06/Workload-analysis-of-a-large-scale-key-value-store.pdf

[7] Dan R. K. Ports et al., "Transactional Consistency and Automatic Management in an Application Data Cache,". [Online]. Available: https://www.usenix.org/legacy/event/osdi10/tech/full_papers/Ports.pdf

10.48047/jocaaa.2026.35.03.32

[8] Fei Da et al., "State of the Art in Parallel and Distributed Systems: Emerging Trends and Challenges," Electronics, 2025. [Online]. Available: https://www.mdpi.com/2079-9292/14/4/677

[9] Haryadi S. Gunawi et al., "Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages," 2016. [Online]. Available: https://ucare.cs.uchicago.edu/pdf/socc16-cos.pdf

[10] AWS Documentation, "What is cell-based architecture?" [Online]. Available: https://docs.aws.amazon.com/wellarchitected/latest/reducing-scope-of-impact-with-cell-based-architecture/what-is-a-cell-based-architecture.html

[11] Ioana Baldini et al., "Serverless Computing: Current Trends and Open Problems," arXiv:1706.03178v1, 2017. [Online]. Available: https://arxiv.org/pdf/1706.03178

[12] AWS, "AWS Step Functions—workflow orchestration." [Online]. Available: https://aws.amazon.com/step-functions/