

Latency-Consistency Tradeoffs in Real-Time Personalization Serving: A Comparative Architectural Study

Niharika Srivastav

Submitted:22/08/2024

Revised:01/10/2024

Accepted:09/10/2024

Abstract: Personalization systems must continually provide and display features that are dynamic: user, product, inventory, pricing and contextual. With e-consumers demanding a lightning-fast response, it's not a simple task to get a site to respond quickly, while maintaining the desired consistency and data freshness of the business. This research paper will compare the different approaches to the design of personalization-serving architectures, distributed caching, cache invalidation strategies, the pros and cons of using either eventual consistency or strong consistency, and the way in which features can be cached as individual features or as sequences. These are compared to other important characteristics of the service (such as serving latency, throughput, feature freshness, stale read rate, scalability, cache efficiency, and infrastructure requirements). Three axis classification are provided, as a function of freshness requirement, read/written ratio and consistency tolerance to consider the implications of workload and feature properties on architectural choices. The framework is based on differentiating features based on their need for very up-to-the-moment information, read-heavy workload, availability of limited staleness periods. The analysis shows that there isn't a single consistency and serving architecture for all personalization workloads. Rather, architectural decisions must be made based on the operational needs of individual features and their workloads characteristics. The proposed framework provides suggestions of what actionable steps should be taken by the back-end developers and system architects of personalization systems to design infrastructure that can balance latency, freshness, consistency, scalability, and resource efficiency.

Keywords: Real-Time Personalization; Feature Stores; Data Consistency; Cache Invalidation; Serving Latency

1. Introduction

With the prevalence of large-scale e-commerce systems, real-time personalization has emerged as a crucial part of online platforms that dynamically fine-tune their recommendations and user experience with the ever-evolving data. Personalization systems have increasingly been integrated with the product attributes, inventory, pricing, and context, in order to produce relevant content at runtime while the user is making a request. The systems need to be able to retrieve features quickly as the delay in getting up-to-date information has a direct impact on serving. On the other hand, personalization infrastructure needs to deal with feature freshness, feature consistency,

infrastructure availability and infrastructure cost. This is more difficult as there is more interaction with the user and more state changes in the business. This makes it crucial to have serving architectures that will be able to efficiently retrieve the features with a balance of performance, freshness and consistency.

Different problem with personalization serving is that there is a limitation between being low latency and good consistency. While eventual consistency will decrease the coordination overhead and boost serving performance, it may introduce stale user, inventory, or pricing features. A higher Consistency level might give more recent state but could also add coordination overhead, network communication overhead and infrastructure overhead. This is a difficult choice since personalisation workloads will have different requirements for freshness, read/writes ratio, and supporting stale information. For inventory or pricing features, a consistency strategy that is well

*Senior Software Engineer, Independent Research,
Walmart Global Tech*

Email: Niharika.belle@gmail.com

ORCID ID: 0009-0009-2346-1954

Santa Clara, California

suited for recommendation may not be the same for these features. Outdated information might have varying operational implications. It necessitates approaches being evaluated with different workloads, spanning different conditions instead of expecting there to be one single approach for all personalization workloads.

Current research tends to focus on feature stores, distributed caching, and consistency models separately from each other as infrastructure problems. These areas have been deeply researched, but only a little bit of comparative research has been done that maps freshness requirements, read/writes, consistency tolerance and serving latency in the same personalization-serving framework. As a result, there is still limited practical advice to provide on how to separate the recommendation ranking, inventory-aware personalization and dynamic pricing architectures. This gap poses a challenge for system designers to know how the workload characteristics and feature requirements should affect the architecture, especially in systems with strict latency requirements and a need for sufficiently fresh and reliable data.

Against this background, the study aims to classify personalization-serving architectures according to freshness requirements, read/write ratios, and consistency tolerance; compare strong, eventual, and hybrid consistency approaches; measure latency, throughput, freshness, stale-read rate, cache performance, and infrastructure utilization; examine the effects of workload characteristics on architectural performance; and develop a practical architecture-selection framework for real-time personalization systems. The study therefore investigates the impact of consistency strategy on p50-, p95- and p99-latencies, the impact of feature update frequency on stale read rates, the impact of read/write ratios on architectural performance, the impact of cache TTL on latency and freshness, and which architectural configurations provide a good trade-off between serving performance, freshness, consistency, scalability and infrastructure requirements.

It is important that the study offers a practical basis for the understanding of how real-time personalization systems can be used to deliver balance between serving latency, feature freshness and consistency requirement. The findings give measurable recommendations for backend engineers to choose their consistency model (strong consistency, eventual consistency or hybrid

consistency) depending on workload rather than by consistency model. The analysis of p50, p95, and p99 latency, stale-read rates, cache-hit rates, and throughput and resource utilization can help design and evaluate feature-serving infrastructure. For system architects, the study provides a structured framework for relating freshness requirements, read/write ratios, and consistency tolerance to architectural decisions. It can help select the right combination of online feature stores, distributed caches, update mechanisms, and consistency strategies for various workloads, especially when there are varying consistency requirements for recommendation, inventory, and pricing features in terms of data freshness and correctness. For e-commerce organizations, the study offers practical insight into managing rapidly changing user and business-state information without unnecessarily increasing infrastructure overhead. The knowledge of how the cache TTL, update frequency, stale reads, and serving latency relate can help in setting reasonable performance and freshness expectations and help allocate computing, memory, network, and storage resources more efficiently.

2. Literature Review

2.1 Real-Time Personalization and Feature Stores

Real-time personalization systems leverage user, product, behavioral, contextual, inventory, and pricing features to deliver suitable, personalized recommendations in real-time. These systems pull relevant features as needed from the client's query, and perform model inference during the user's request, which requires low latency of data access. With the help of a feature store, this process can be carried out offline and online. Offline stores have the model data available from historical data, while online stores have the data values available for production inference. The choice of "hybrid architectures" between batch and streaming updates for improving feature availability. The selection of "hybrid architectures" between batch and streaming updates for improving feature availability. But it is possible there is a time lag between offline and online space which could lead to features that are no longer available. The architecture of the feature-store thus affects the following characteristics, serving latency, relevant predictions, and freshness of the data [1].

Figure 1 illustrates the role of feature stores in supporting real-time data processing and

personalization by connecting historical and current feature data. It underscores the effectiveness of feature management to facilitate prompt decision-making and enhance the relevance of personalized services.

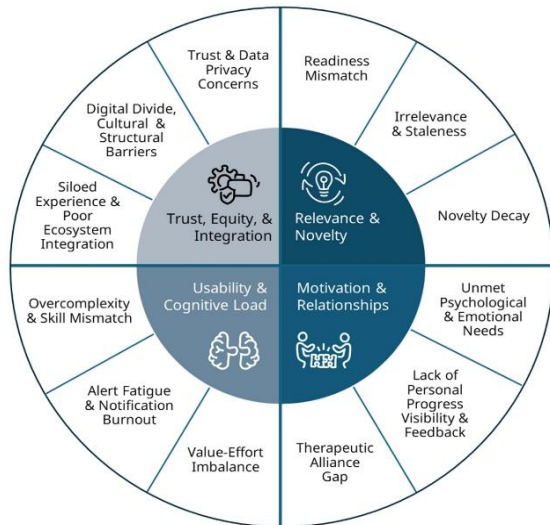


Figure 1: Achieving clinically meaningful outcomes in digital health

2.2 Distributed Caching and Feature Freshness

Distributed caching is designed to decrease the latency when fetching a feature, by placing it closer to the application's serving process. Popular strategies include cache-aside, read-thru, write-thru, and write-behind caching. Local caching minimizes network traffic, but could cause varying data copies for each instance of the application, while distributed caching gives the benefit of sharing but there will be more network overhead. Determine the speed of availability of updated features by using cache invalidation. There are various ways to ensure that top-fuelled, expiration, event-driven and version-driven invalidation, and stale-while-revalidate strategies are set up for satisfaction. Longer TTLs lead to better cache hit rates but may induce a higher level of staleness; shorter TTLs may lead to more requests going to the backend. Therefore, caching performance depends on balancing retrieval speed, invalidation overhead, and feature freshness.

2.4 Count of the elements in the mailbox

Consistency models classify how consistent the retrieved features are with respect to authoritative data. Strong consistency does value consistency, but can be coordination overhead. Eventual consistency allows for temporary inconsistencies, bounded-staleness does not allow

for data that is too old to be used [2]. The decisions made here have an impact on availability, scalability and latency. Other factors in serving latency include cache misses, feature aggregation, serialization, model inference, communication to the network, and database lookups. Latency measurements including average, p95 and p99 are complementary. The nature of the workloads also affects the efficiency of caching. Recommendation systems that are read-intensive can leverage the ability to cache, whereas price and stock information can leverage the ability to be faster updated. This study considers read/write ratios of 10:1, 100:1, 1,000:1, and 10,000:1 to examine these differences [3].

Figure 2 depicts two major consistency models, Strong Consistency and Eventually Consistent, and Bounded Staleness; its relation to distributed capabilities — data freshness, availability, and data serving — and the impact of the models on data serving..

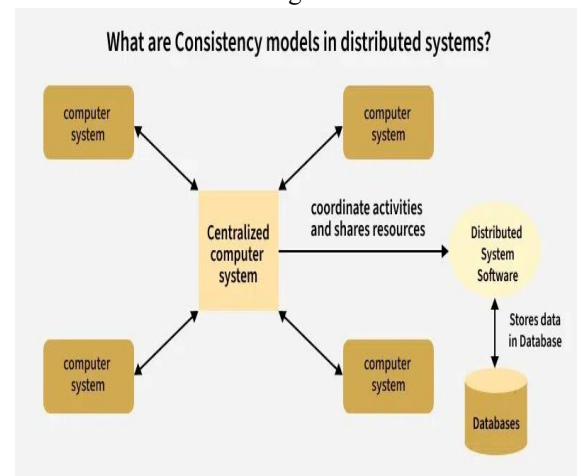


Figure 2: Consistency Model in Distributed System

2.4 Industry Architectures and Research Gap

Archetypes like Uber's Michelangelo or Feast have all been publicized to illustrate the online feature store, offline processing, streaming updates, and precomputed features approaches for supporting low latency inference. However, publicly available documentation does not establish the complete proprietary configurations or performance characteristics of these systems. The current methods of architecture have provided good ideas for designing and creating users, but do not completely determine the most appropriate caching strategy for any specific workload. A few points must be noted that scale the research gap on the limited integration of latency, freshness, consistency and read/write workloads into the context of a

unified comparative framework [4]. This study fills that gap by investigating the behavior of caching strategies on various workload ratios and freshness requirements in terms of mean latency, tail latency, stale-feature frequency and the propagation of updates.

3. Conceptual Framework

3.1 Architecture Classification Framework

The conceptual architecture categorizes real-time personalization architectures in terms of three main factors: freshness requirement, read/write ratio, and consistency tolerance. The concept of 'freshness requirement' refers to the maximum possible age of a feature once retrieved for online inference. Read Write ratio is the ratio of feature reads to feature updates and reflects the application workloads that the serving architecture needs to support. Consistency tolerance represents the degree to which temporary differences between cached or served values and the authoritative data source can be accepted. These are dimensions which give a structured way to compare various feature-serving and caching architectures [5].

Figure 3 illustrates a conceptual framework for evaluating system performance across different operational dimensions. This work adapts the framework to classify real-time personalization architectures as per the freshness required, Read/Write ratio and the consistency tolerance required.

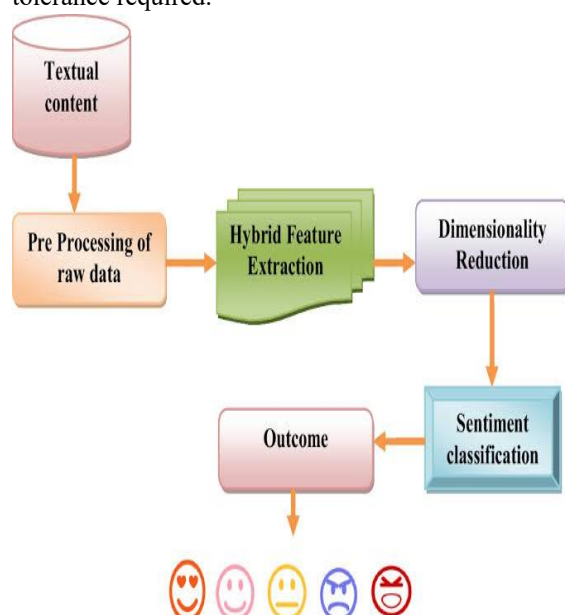


Figure 3: Improving the deep learning monitoring system for CollegeEnglish reading efficiency guided by neural networks

3.2 Freshness Requirement

The freshness of features is defined experientially as very high (<1 s), high (1–30 s), moderate (30 s to 5 min) and low (>5 min). These thresholds are NOT industry standards, but experimental categories created for the current study. Very-high freshness is for features the values from which can be instantly changed, like operational information that changes quickly [6]. Short propagation intervals are possible when the freshness is very high, but moderate freshness allows for updates to be performed in a few minutes. Low freshness indicate features that can have longer update time. The classification makes it possible to remove and then investigate the behavior of the caching and synchronization mechanisms in progressively different freshness requirements all the way up to Internet latency [7].

3.3 Read/Write Ratio

The framework considers four workload ratios: 10:1, 100:1, 1,000:1, and 10,000:1. These are ratios of increasing degree of read dominance. A 10:1 workload includes a greater number of reads and somewhat more reads/writes in a single instance than would a 10,000:1 workload. The higher the number, the more potential caching has to be able to satisfy repeat service requests without accessing the underlying feature store. But advantages of caching need to be weighed against the number of updates, and the potential for stale values. The ratios thus allow for controlled experimental conditions of evaluating the change in architecture performance as a function of the workload intensity [8;9].

3.4 Consistency Tolerance

Consistency tolerance comes in the various forms of strong consistency, bounded staleness, eventual consistency and hybrid consistency [10]. A high consistency implies a high degree of authoritativeness of values whereas a bounded staleness allows values to vary in a designated interval of freshness. Eventual consistency means that there will be temporary differences as long as updates trickle down eventually as well. Hybrid consistency is the use of a combination of consistency modes for various categories of features. Depending on feature characteristics and business criticality, the appropriate tolerance is used. Certain features may need greater consistency because of the more volatile behavior or price, e.g. more volatile inventory.

3.5 Latency–Consistency Trade-off Model

This framework measures the performance of architecture based on features propagation delay, stale-reads, cache-hit rates, throughput, infrastructure utilization and latency (p50, p95 and p99) [11]. These measures provide "normal performance" and details about how quickly updates come in and how often they contain data that is missing or incorrect. The framework assumes that improving one performance dimension may affect another; for example, stronger consistency or faster propagation may increase processing or communication overhead.

3.6 Proposed Architecture Matrix

The authors propose a comparison of the following different forms of online consistency: "Direct strongly consistent reads," cache-based eventual consistency, streaming feature stores, strongly consistent online stores, and a hybrid serving architecture [12]. To evaluate each architecture, the relative values of each of the categories of freshness, read/write ratio, and consistency tolerated are established. The matrix defines a methodology that allows to identify measurable differences of latency, freshness, stability, throughput, and resource usage under controlled load conditions. The framework doesn't presume that all architectures are equal; instead it compares their performance for various operational conditions.

4. Methodology

4.1 Research Design and Experimental Environment

This study will use comparative experiments to examine different architectures designed to serve personalization under controlled circumstances [13]. We will use the following ranges of workload intensities, updates frequencies, cache configurations and consistency requirements through repeated benchmarking experiments. The simulated environment will be made up of 3-5 application nodes, 2-3 cache or feature-store nodes, an event-streaming layer, a persistent database, a load-generation layer, and a monitoring and metrics-collection layer. The same workloads will be continually applied to the same environments being compared.

4.2 Dataset, Workloads, and Experimental Architectures

A synthetic dataset will be created to contain around 1 million users, 100,000 products and 10-50 million feature records [14]. The number of requests per second can be from 1,000 to 20,000, and the number of feature updates per second can be from 10 to 1,000. Three architectures will be considered: Strong Consistency architecture, eventual consistency architecture and hybrid consistency architecture. The ranking against the workload of a read is recommended, and is composed of three types of data: read-heavy workload, recent clicks and product affinity, session behavior, and freshness requirements of 30–300 seconds, with a read/write ratio of about 1,000:1. The approximative read-write ratios the inventory-aware workload will use are around 100:1, updates are 10-500 per second and freshness ranges from 1-10 seconds. Approximate DPO and price updates will be 10:1 – 100:1, 100 updates to 1,000 updates per second will be used and the freshness requirement will be sub-second up until around 5 seconds.

Realistic synthetic data sets were used for test performance, as shown in Figure 4. A similar strategy is used to create the personalization workloads of different types of controlled workloads for this study: user, product, feature record, request rate, and update frequency.

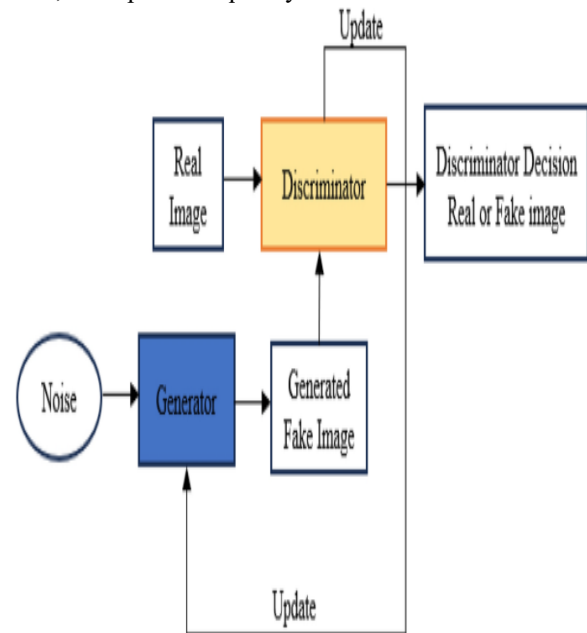


Figure 4: Realistic synthetic dataset generation for cyber-physical systems: a performance evaluation

4.3 Cache, Consistency, and Update Experiments

The different TTLs that will be used to evaluate cache behaviour 1 second, 5 seconds, 30 seconds, 60 second and 300 seconds. These configurations will be tested with various workload intensities, measuring Cache-Hit rate, Cache-Miss rate, Stale Read rate and p95 / p99 latency. Consistency experiments will compare synchronous updates, asynchronous updates, and bounded-staleness propagation under low, moderate, and high update frequencies. Special focus will be given to feature propagation delay and stale-state exposure, particularly when it comes to inventory and pricing workloads where stale data might influence the decision on what's served [15]

4.4 Performance Metrics and Statistical Analysis

The following latency, requests per second and error, cache-hit, stale-read percentage, feature propagation delay will be measured and utilized as data: CPU usage, memory usage, network usage, storage usage, cost-per-million requests. Means and standard deviations, median, percentile distribution, 95% confidence interval, percentage difference and coefficient of variation are used to summarize the resulting data. Latency will be compared using appropriate statistical tests between the different architectures and effect sizes will be reported where applicable [16]. Results will be only reported on statistical levels when validated by the experimental results.

Table 1: Summary of the Experimental Methodology and Evaluation Parameters

Methodology Area	Details
Research Design	Comparative experimental study using controlled benchmarking tests.
System Setup	3–5 application servers, 2–3 cache/feature-store servers, streaming system, database, load generator, and monitoring tools.
Dataset	Synthetic dataset with about 1 million users, 100,000 products, and 10–50 million feature records.
Request Rate	1,000–20,000 requests per second.

Methodology Area	Details
Update Rate	10–1,000 feature updates per second.
Architectures Tested	Strong consistency, eventual consistency, and hybrid consistency.
Recommendation Workload	Read/write ratio of about 1,000:1; freshness requirement of 30–300 seconds.
Inventory Workload	Read/write ratio of about 100:1; updates of 10–500 per second; freshness requirement of 1–10 seconds.
Pricing Workload	Read/write ratio of 10:1–100:1; updates of 100–1,000 per second; freshness requirement of below 1–5 seconds.
Cache Testing	TTL values of 1, 5, 30, 60, and 300 seconds.
Consistency Testing	Synchronous, asynchronous, and bounded-staleness updates.
Main Metrics	p50, p95, p99 latency, throughput, error rate, cache-hit rate, stale-read rate, and feature propagation delay.
Resource Metrics	CPU, memory, network, storage use, and cost per million requests.
Data Analysis	Means, medians, standard deviations, percentage differences, confidence intervals, and appropriate statistical tests.

5. Experimental Results

5.1 Serving Latency, Throughput, and Scalability Results

Latency will be measured using the p50 and p95 values and the p99 value, to reflect typical service performance and tail latency. The request rates will be ramped up from 1,000 to 20,000 requests per second to see how each architecture performs when the system is tested. Changes in p95 and p99 latency with high system stress will be of special concern because tail latency can grow

significantly as infrastructure hits the saturation point. Also, throughput will be measured by the maximum number of possible requests per second for each architecture at varying rates of feature updates. In addition to throughput, error rates will be measured and reported, and the point to which increases in workload cause instability to the system will be determined. The results will demonstrate if each architecture will yield consistent results as requests and updates grow.

5.2 Feature Freshness and Stale-Read Results

For Feature freshness, the amount of time for a user update to the source system to take effect in the freshness feature will be used to measure how fresh the value is [18]. For each architecture and workload, propagation delay statistics will be computed: mean, median, p95 and maximum propagation delays. Stale-read percentage will also be measured under different feature-update frequencies to determine how often requests receive values that are older than the defined freshness requirement. This strong consistency architecture will benefit from a comparison with eventual and hybrid consistency architecture in terms of propagation behavior and inability to see stale states. Special focus will be paid on inventory and pricing load because their freshness's are shorter than the recommended feature. The findings will also include the correlation between stale reads and cache TTL, highlighting the tradeoff between wait time and the longer caching can be set to allow serving stale feature values.

5.3 Cache Performance, Read/Write Workload, and Resource Utilization

Cache performance will be measured by the cache-hit rate, cache-miss rate, cache-eviction rate and invalidation frequency [19]. Different TTLs (1 second, 5 seconds, 30 seconds, 60 seconds, 300 seconds) will be used to identify any impact that expiration policies have on the efficiency of serving data and the presence of stale data. The following read/write ratios will be evaluated: 10:1, 100:1, 1,000:1, and 10,000:1. These experiments will show how increasing read dominance influences cache effectiveness and overall system performance. Resources will also be monitored such as CPU usage, memory usage, network usage, storage operations, and overall infrastructure usage. These will be used to determine the amount of computing power and infrastructure needed to reduce the

serving latency of these measurements, or to ensure that they are fresh.

5.4 Workload-Specific and Statistical Results

The final results will compare architectural behavior across the three workload scenarios: recommendation ranking, inventory-aware personalization, and dynamic pricing [20]. Differences will be discussed based on the time taken, throughput, freshness, and stale-read exposure of each workload. Recommendation ranking is likely to be more about high reads and more on speeding up of features and freshness of inventory and prices. The 95% confidence intervals, percentage differences, effect sizes and variance measures of repeated experimental measurements will be summarized by statistical analysis. Statistically appropriate test is used to decide if observed differences in the architectures are backed up by the experimental data. Statistical significance will only be reported when supported by the repeated measurements and distribution

Table 2: Summary of Experimental Results and Evaluation Measures

Result Area	Key Measures / Tests
Latency, Throughput & Scalability	p50, p95, p99 latency, requests per second, error rate, and performance at 1,000–20,000 requests/second.
Feature Freshness	Propagation delay, stale-read percentage, and freshness under different update rates.
Cache Performance	Cache-hit rate, cache-miss rate, eviction rate, and invalidation frequency using TTLs of 1–300 seconds.
Workload & Resources	Performance under 10:1–10,000:1 read/write ratios; CPU, memory, network, and storage use.
Workload Comparison	Comparison of recommendation, inventory, and pricing workloads based on latency, throughput, freshness, and consistency.
Statistical Analysis	95% confidence intervals, percentage differences, effect sizes, and appropriate statistical tests.

6. Discussion

6.1 Interpretation of Major Findings

The experimental results show that feature-serving architecture has a significant impact on latency, consistency, freshness, and system performance. Architectures react in different ways when the number of requests and feature-updates grows, as shown by different p50, p95, and p99 latencies. Higher serving latency for architectures that guarantee higher consistency of the architecture may be due to additional serialization necessary to deliver the latest feature values to clients. Architectures that fit in the eventual consistency modality, in contrast, can significantly decrease coordination overhead and potentially return other responses sooner, but might return lesser values. The measured freshness and stale-reads are then important to help determine if improvements in performance are gained while incurring a viable consistency cost.

6.2 There are various tradeoffs between Latency, Consistency, and Freshness.

The findings also show that consistency and latency should not be treated separately in the design of personalization systems in real-time systems [21]. The consistency semantics are more important for less time-sensitive recommendation features, but many time-sensitive features need a strong consistency semantics, because the data is made available to the system later and is crucial to making decisions in that system. The importance of stale data will vary according to feature and how fast the state of the feature changes. The information about inventory and pricing is typically subject to tight freshness requirements compared to the behavioral recommendations that can accept short stales periods. Thus, it is important to consider the desired freshness of the architectural elements, and not seek uniformity on all elements under the guise of a consistency model.

6.3 Caching, TTL, and Workload Characteristics

Keeping data in the network cache can significantly enhance the performance of read-based applications, as it decreases the need for multiple accesses to the distributed feature stores and minimizes serving timeouts [22]. A tradeoff of cache efficiency and freshness is present when selecting TTL, however. By keeping the time-to-live short, the stale values are available for a smaller amount of time but miss rates and backend traffic are higher. A higher TTL value helps improve the efficiency of the

cache but can lead to stale-read rates. However, event-driven invalidation can bring quicker updates and increase infrastructure and operational complexity. Likewise, if the number of feature updates is large, this can make caching less effective, and may increase the demands on the network, CPU and storage. Thus, the optimal architecture will depend on the nature/read vs write ratio of the workload and the frequency of updates to the workload.

6.4 Infrastructure, Cost, and Consistency Budget

The variations in performance between architectures also have direct conclusions about infrastructure, cost implications. A higher request rate and frequent changes of features lead to higher use of CPU, memory, network and storage. A consistency budget provides a practical way of defining the maximum acceptable staleness for individual features. The consistency objective for Recommendation features may be as high as say 50% or more, while some features such as inventory and pricing might need more comparable low limits [23]. Raising the consistency demands can then lead to higher infrastructure costs, e.g., due to more coordination, data replication, and network communication. When considering architectural alternatives, cost per million requests and cost per million feature updates may also be taken into consideration, in addition to latency and freshness.

6.6 Literature Comparison and Applications.

The results are similar to prior works and industry designs that focus on the low latency serving, consistency and scalability/operational costs performance trade-offs [24]. Differences between the experiments and past research can be attributed to differences in workload intensity, infrastructural configuration, caching strategies, and system size. The results also show the relevance of setting Latency SLOs and Freshness SLOs based on criticality of features for backend engineers [25]. Using online feature stores, streaming systems along with event driven updates, combined with monitoring stale-reads and graceful degradation, can help reduce infrastructure needs while ensuring performance.

7. Conclusion

It is evaluated different real-time feature serving architectures according to: latency of the serving - freshness of the data served - consistency

of the data served - scalability - infrastructure requirements. The experimental results indicated that the p50, as well as the p95 and p99 latency, could vary measurably based on the architecture chosen, as feature-updates and request-rates grew. Read-intensive workloads have seen improvements in terms of caching and the optimized path to serves while feature changes have introduced additional strings of coordination, network and processing demands. The results also showed that higher stale-read rates are possible with weakening consistency models if the changes are in the serving performance.

These findings validate that latency, freshness, consistency and scalability are not independent properties of a system. The research also offers a three-axis classification criterion based on the level of freshness of the featured elements, the ratio of read and write operations and level of the consistency that can be achieved. This can be a structure that enables to go out it to determine how the various features are and really are not implied in the selection from the supporting architectures. For features that have to be fresh, and that are frequently updated, such as updates, there are more ways to ensure they're fresh at runtime than there are for features that do not have either of those requirements. The framework thus goes beyond just conducting an architectural analysis based only on latency, also covering workload and data usage characteristics. In reality, the framework can be used for the design and optimization of real-world personalization infrastructure by allowing the backend engineers to categorize its features based on their freshness requirement, update frequency, read/write ratio, and acceptable degree of staleness to determine an architecture for serving the features. This can further influence aspects of the feature and role selection, caching, TTL, event driven invalidation, streaming pipelines, replication, and online feature stores. The method can also be employed in organization to establish service level targets on latency and freshness, which allow them to find the right equilibrium between performance, reliability, scalability and infrastructure costs.

A larger scale experimental testing trace and real production traces are needed to evaluate the framework and to collect new results in the future; Study the generalizability of results obtained for different workloads used and infrastructure conditions. A future investigation could look into adaptive cache TTLs that adapt to feature update

patterns or at dynamic consistency policies that adapt to the features' criticality and workload conditions. Computerized architecture-selection mechanisms are also a worthwhile path of research to explore given the idea of using the characteristics of a workload to recommend the best configuration of the architecture. Finally, future research should integrate personalization quality measures (e.g., relevance of recommendations, conversion measures) and infrastructure quality measures (e.g., latency, freshness, consistency, throughput, and cost). This integration could provide a more comprehensive understanding of the real-time personalization system and how the choices of infrastructure can translate into better personalization service and value at the end point.

References;

- [1] Vanama, S. K. R. (2023). Integrating Site Reliability Engineering SRE Principles into Enterprise Architecture for Predictive Resilience. *International Journal of Emerging Trends in Computer Science and Information Technology*, 4(3), 164-170. <https://www.ijetcsit.org/index.php/ijetcsit/article/download/514/462>
- [2] Ubale, A. (2023). Beyond Telematics: Leveraging Generative AI for Synthetic Accident Reconstruction and Liability Attribution in Autonomous Vehicle Claims. *International Journal of AI, BigData, Computational and Management Studies*, 4(4), 119-124. <https://ijaibdcms.org/index.php/ijaibdcms/article/download/356/352>
- [3] Sayyed, Z. (2024). *Implementing automation with BPMN for margin call workflow*. IRJERNET. <https://irjernet.com/index.php/fecsit/article/view/171>
- [4] Hariharan, R. (2024). *API gateway threat prevention in large-scale applications*. SciPubHouse. https://scipubhouse.com/wp-content/uploads/2025/10/011-API_gateway_threat_prevention_in_large-scale_applications.pdf
- [5] Gundla, S. R. (2024). *AI-optimized Kubernetes scheduling: Node affinity for Java microservices*. SciPubHouse. <https://scipubhouse.com/home/international-journal-of-sustainability-and-innovation-in-engineering-ijsie/content/ijsie-2024/ai->

optimized-kubernetes-scheduling-node-affinity-for-java-microservices/

- [6] Nagaraj, V. (2024). *Addressing power efficiency challenges in AI hardware through verification*. SciPubHouse.
<https://scipubhouse.com/home/international-journal-of-sustainability-and-innovation-in-engineering-ijisie/content/ijisie-2024/addressing-power-efficiency-challenges-in-ai-hardware-through-verification/>
- [7] Samala, S. (2024). *Real-time Jira analytics: Integrating JQL with Power BI/Snowflake for predictive agile metrics*. SciPubHouse.
<https://scipubhouse.com/home/international-journal-of-sustainability-and-innovation-in-engineering-ijisie/content/ijisie-2024/real-time-jira-analytics-integrating-jql-with-power-bi-snowflake-for-predictive-agile-metrics/>
- [8] Ubale, A. (2024). From Detect and Repair to Predict and Prevent: Assessing the Viability of Real-Time AI Nudges in Reducing Fleet Accident Rates. *International Journal of Emerging Research in Engineering and Technology*, 5(2), 115-123.
<https://ijeret.org/index.php/ijeret/article/download/408/389>
- [9] RaoVanama, S. K. (2024). AI-Augmented CI/CD Pipeline Optimization for Scalable Cloud-Native Deployment. *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, 5(4), 175-187.
<https://ijaidsmi.org/index.php/ijaidsmi/article/download/368/338>
- [10] Diogo, M., Cabral, B., & Bernardino, J. (2019). Consistency Models of NoSQL Databases. *Future Internet*, 11(2), 43.
- [11] Marchisio, A., Massa, A., Mrazek, V., Bussolino, B., Martina, M., & Shafique, M. (2020, November). Nascaps: A framework for neural architecture search to optimize the accuracy and hardware efficiency of convolutional capsule networks. In *Proceedings of the 39th International Conference on Computer-Aided Design* (pp. 1-9).
- [12] Jin, H., Wei, S., Sha, Y., Ye, C., Liu, H., & Liao, X. (2022). PMLiteDB: Streamlining access paths for high-performance persistent memory document database systems. *IEEE Transactions on Computers*, 72(6), 1778-1791.
- [13] Felizardo, K. (2021). Development of an Ontology-based Approach for Knowledge Management in Software Testing. *Journal of Software Engineering Research and Development*.
- [14] Ma, R., Chen, X., & Zhai, R. (2023). A DDoS attack detection method based on natural selection of features and models. *Electronics*, 12(4), 1059.
- [15] Matos, M., & Greve, F. (2021). *Distributed applications and interoperable systems*. Springer International Publishing.
- [16] Zerouali, A., Mens, T., Decan, A., Gonzalez-Barahona, J., & Robles, G. (2021). A multi-dimensional analysis of technical lag in Debian-based Docker images. *Empirical Software Engineering*, 26(2), 19.
- [17] Qiu, K., Huang, S., Xu, Q., Zhao, J., Wang, X., & Secci, S. (2017). ParaCon: A parallel control plane for scaling up path computation in SDN. *IEEE Transactions on Network and Service Management*, 14(4), 978-990.
- [18] Sun, Y., Uysal-Biyikoglu, E., Yates, R. D., Koksall, C. E., & Shroff, N. B. (2017). Update or wait: How to keep your data fresh. *IEEE Transactions on Information Theory*, 63(11), 7492-7508.
- [19] Zheng, Q., Yang, T., Kan, Y., Tan, X., Yang, J., & Jiang, X. (2021). On the analysis of cache invalidation with LRU replacement. *IEEE Transactions on Parallel and Distributed Systems*, 33(3), 654-666.
- [20] Panchetti, T., Pietrantoni, L., Puzzo, G., Gualtieri, L., & Fraboni, F. (2023). Assessing the relationship between cognitive workload, workstation design, user acceptance and trust in collaborative robots. *Applied sciences*, 13(3), 1720.
- [21] Lv, Z., Zhang, W., Zhang, S., Kuang, K., Wang, F., Wang, Y., ... & Wu, F. (2023, April). Duet: A tuning-free device-cloud collaborative parameters generation framework for efficient device model generalization. In *Proceedings of the ACM Web Conference 2023* (pp. 3077-3085).
- [22] Gessert, F., Schaarschmidt, M., Wingerath, W., Witt, E., Yoneki, E., & Ritter, N. (2017). Quaestor: Query web caching for database-as-a-service providers. *Proceedings of the VLDB Endowment*, 10(12), 1670-1681.
- [23] Makreshanski, D., Giceva, J., Barthels, C., & Alonso, G. (2017, May). BatchDB: Efficient isolated execution of hybrid OLTP+ OLAP workloads for interactive applications. In *Proceedings of the 2017 ACM International*

Conference on Management of Data (pp. 37-50).

- [24] Emily, H., & Oliver, B. (2020). Event-driven architectures in modern systems: designing scalable, resilient, and real-time solutions. *International Journal of Trend in Scientific Research and Development*, 4(6), 1958-1976.
- [25] Kogias, E. M. (2020). *Operating System and Network Co-Design for Latency-Critical Datacenter Applications* (Doctoral dissertation, EPFL).