Universal Library of Engineering Technology Research Article

ISSN: 3064-996X | Volume 3, Issue 1

Open Access | PP: 12-21

DOI: https://doi.org/10.70315/uloap.ulete.2026.0301003

Optimization of MLOps Processes for Product Recommendation Systems under High Load

Dmitrii Timoshenko

Applied Scientist, Seattle, USA.

Abstract

This article examines the comprehensive optimization of MLOps processes for high-load product recommendation systems, in which stringent latency SLAs and terabyte-scale embeddings coexist with rapid drift in user preferences and intense business pressure to maximize commercial KPIs. The relevance of the study stems from the fact that classical batch-oriented MLOps practices do not provide the required feature consistency, stable model quality, or predictability of revenue, conversion, and retention metrics under peak loads typical of e-commerce and media services. The study aims to develop a holistic engineering and product-oriented approach to the design of data, inference, and training architectures, encompassing a Feature Store with streaming aggregations, hierarchical parameter servers, algorithmic embedding compression, dynamic batching, and concurrent model execution, vector search over embeddings, as well as drift monitoring loops and continuous (online) training. The scientific contribution lies in integrating hardware-oriented optimizations and process-centric MLOps methodologies into a unified RecSys design standard that simultaneously improves throughput and reduces latency while maintaining recommendation quality and stabilizing key business metrics. The article is intended for data and MLOps engineers, recommendation system architects, and technical leaders of digital products and marketing teams.

Keywords: *MLOps, Recommendation Systems, DLRM, High-Load Systems.*

INTRODUCTION

In the contemporary landscape of e-commerce and online services, recommender systems have evolved from auxiliary navigation tools into a primary driver of revenue and user retention. According to existing studies, personalization based on machine learning algorithms accounts for up to 35% of Amazon's revenue and 80% of Netflix's views [1]. In online retail, recommendations influence not only which items are purchased, but also campaign uplift, average order value, and the distribution of demand across categories and price segments. However, as models become deeper and more complex, transitioning from matrix factorization to transformers and graph neural networks, the requirements for their serving infrastructure increase exponentially.

Many such business applications are particularly tied to commercial processes such as digital merchandising, promo placement, coupon and loyalty mechanics, and remarketing. As a result, any degradation in model performance, feature uniformity, or latency will be immediately reflected in business metrics monitored by the commercial or marketing departments, such as click-through rate (CTR), conversion rate, revenue per session, and inventory turnover. MLOps processes for recommender systems cannot be viewed solely as a technical operations exercise; they must also directly support the reliability and controllability of business experiments.

The specifics of recommender systems under high load are characterized by a unique set of constraints that distinguish them from computer vision or NLP workloads. First, there are stringent latency Service Level Agreements (SLAs). Research indicates that an increase in system response latency of only 100 ms may lead to a 2.4% or more drop in conversion [2]. For systems serving millions of users, this implies that extremely complex computations (nearest-neighbor search, ranking thousands of candidates) must be performed within a 10–50 ms window.

Second, there is the problem of data scale. Modern DLRM architectures operate on categorical features (user IDs, item IDs) with cardinalities that can reach billions. This leads to embedding tables whose size reaches hundreds of gigabytes or even terabytes, exceeding the memory capacity (HBM) of even the most advanced accelerators (GPUs/TPUs) [3].

Third, there is environmental dynamism. Unlike image classification, where the concept of a cat is static, user preferences and item popularity change continuously. The phenomena of data and concept drift require the introduction of continuous training or online learning processes, which pose nontrivial challenges for MLOps engineers in ensuring the stability and reproducibility of pipelines [4].

Classical MLOps practices, developed primarily for batch-processing workloads, prove insufficient for real-time RecSys whose outputs are directly monetized. A standard train once per week – deploy cycle leads to models that fail to account for the latest user interactions, price changes, or campaign configuration, which is critical for session-based recommendation scenarios and short-lived promotions. Furthermore, there is the well-known problem of training-serving skew, in which the feature generation logic in the offline environment (Spark/Pandas) diverges from that in the online environment (C++/Go/Java microservices), resulting in degradation of model quality metrics in production [5]. In commercial systems, this skew manifests as inconsistent prices and availability, incorrect segmentation and promo eligibility, and ultimately as fluctuations in conversion and revenue that business teams cannot explain.

Thus, the relevance of this study lies in the need to systematize disparate engineering and algorithmic optimization methods into a unified, coherent approach to building MLOps processes for high-load recommender systems.

The objective of this work is to develop theoretically grounded, practically applicable recommendations for optimizing the architecture and MLOps processes of product recommendation systems under high load, with an explicit focus on preserving and improving target business metrics (CTR, conversion, revenue per session, and retention) during traffic peaks and intensive marketing activity.

To achieve this objective, the following tasks are addressed:

1. Analysis of modern architectural patterns for managing the lifecycle of recommendation models, including Feature Store and Model Registry.
2. Investigation of inference optimization methods, including quantization, pruning, and dynamic batching, with assessment of their impact on latency and throughput metrics.
3. Comparative analysis of model serving tools (Triton Inference Server, TensorFlow Serving) in the context of working with large embedding tables.
4. Development of strategies to combat data drift and implementation of continuous training pipelines.
5. Assessment of the risks and limitations associated with deploying high-frequency model updates in production environments.

The study's scientific novelty lies in integrating hardware-level optimizations (GPU and memory) with process-oriented MLOps methodologies and explicitly linking them to business outcomes. While most works focus either exclusively on recommendation algorithms or on general MLOps issues, this article examines the synergy between the two domains. It analyzes how architectural decisions propagate to commercial KPIs. It proposes hybrid approaches, such as hierarchical parameter servers (HPS) combined with streaming Feature Stores, to minimize end-to-end latency and to increase the stability and predictability of revenue-related metrics under high load.

MATERIALS AND METHODS

The study is based on a Systematic Literature Review (SLR) conducted according to the PRISMA methodology. Sources were retrieved from the abstract databases Scopus and Web of Science, as well as from the digital libraries IEEE Xplore and ACM Digital Library. The sample included publications from 2020 to 2025 that match the following keywords: Recommender Systems, MLOps, High Load, Embedding Optimization, Concept Drift, and Inference Serving.

The analysis includes works published in peer-reviewed journals and A/A* level conference proceedings (such as KDD, RecSys, SIGMOD, OSDI, VLDB, NeurIPS), as well as technical reports from R&D units of major technology companies, including Meta, Google, NVIDIA, Alibaba, and ByteDance. A key requirement is that these reports describe architectures of real-world production systems, such as Monolith, Ekko, or HugeCTR. Additionally, experimental data or performance metrics are mandatory to enable quantitative evaluation of the proposed solutions.

Blog posts are excluded from consideration, as are works that analyze recommender systems solely from an algorithmic perspective, without addressing engineering aspects related to deployment, operation, and maintenance in production environments. In total, 21 key sources were selected, including fundamental works on DLRM architecture, studies on inference optimization, and concept drift management.

Several complementary analysis methods are applied in this work. Architectural analysis enables us to break down the overall Triton architecture, or a sub-part like the Feature Store, into functional modules to identify performance bottlenecks, and helps us analyze how each module fits into the overall architecture, and how they contribute to the latency, resource consumption, and reliability requirements of the system.

In addition, comparative (benchmarking) analysis is employed, involving the comparison of performance characteristics of different frameworks, for example, TensorFlow Serving and Triton, based on experimental data from the selected sources. This enables objective assessment of the advantages and limitations of alternative solutions in typical operational scenarios.

To describe and understand system organization, modeling is also used. Mermaid is employed as a formal notation to visualize data flows, server architectures, and CI/CD/CT processes. Such diagrams help systematize knowledge of the system's structure and simplify the analysis of its scalability and integration points.

Special attention is paid to trade-off analysis. A balance is assessed between recommendation accuracy, measured by AUC and NDCG metrics, and system-level characteristics such as response latency and memory consumption. This analysis is performed in the context of optimization techniques, including quantization and caching, to understand the degree to which model quality loss is acceptable in exchange for improved performance and infrastructure efficiency.

RESULTS

Data Management Architecture: Feature Store as a Foundation of Consistency

In high-load recommender systems, data quality and access speed are decisive factors. One of the most acute problems is training-serving skew, divergence in feature distributions or feature computation logic between training and inference stages. This often occurs when training features are prepared via batch processes (e.g., Spark). In contrast, inference features are reimplemented in high-performance languages (e.g., C++, Go), which inevitably leads to bugs and inconsistencies [6].

In commercial product recommendation scenarios, this type of inconsistency directly impacts key business indicators. Misaligned feature computation between offline training and online serving manifests as outdated prices in recommendations, incorrect stock availability, or missing promotional attributes at inference time. For a high-traffic storefront, even a small fraction of recommendations built on stale or inconsistent features can lead to measurable drops in click-through rate, conversion, and average order value, as well as an increase in customer support incidents. Centralizing feature computation in a Feature Store, therefore, serves not only as an engineering control to manage technical debt but also as a mechanism to stabilize revenue and marketing campaign performance during peak load.

Dual Feature Store Architecture

To address this issue, the industry widely adopts the Feature Store pattern, which implements data abstraction through two physical stores under a unified logical interface [7]. The first store, the offline store (cold storage), is intended for storing historical data over long periods, up to months and years, and is used primarily for model training. CDP storage backends include a distributed file system (such as the Hadoop Distributed File System (HDFS) or Amazon Simple Storage Service (S3)), a columnar database (such as BigQuery or Snowflake), or a columnar storage format (such as Parquet). These storage backends should be able to scan high-velocity data and query terabytes of data. It is also important that the backend supports time-travel queries, which allow the state of features to be retrieved at arbitrary points in time to avoid future leakage.

The second store, the online store (hot storage), is oriented toward real-time operation and is responsible for providing feature vectors for model serving when handling online requests. In-memory databases such as Redis or RonDB, as well as low-latency NoSQL stores such as DynamoDB or Cassandra, are used at this level. The primary requirement is ultra-low latency for key-based read operations (point lookups).

Latency Optimization in the Online Store

Performance studies conducted on OpenMLDB and Hopsworks demonstrate that the choice of technology for the online store critically influences the overall latency of the recommendation service [8]. Traditional key-value stores (such as Redis) deliver high speed but can become bottlenecks when complex on-the-fly aggregations are required (for example, average order value over the last 10 minutes).

In this regard, a promising direction is the use of specialized engines that support SQL-like queries over streaming data with just-in-time (JIT) compilation of query plans into machine code. For instance, the OpenMLDB architecture achieves latency below 5 ms at a throughput of 17k QPS due to optimized execution plans and caching [8]. This enables moving feature engineering logic out of application code and into the Feature Store itself, guaranteeing identical computations in both offline and online modes. The table 1 shows a comparative analysis of Feature Store architectures for RecSys.

Table. 1. Comparative analysis of Feature Store architectures for RecSys.

Characteristic	**Feast**	**Hopsworks**	**Tecton**	**OpenMLDB**
Type	Open Source (DIY)	Platform (Enterprise/OSS)	Managed Service	Open Source (Optimized)
Offline Store	Pluggable (S3, BQ, etc.)	HopsFS / S3	Snowflake / Databricks	Spark / HDFS
Online Store	Redis, DynamoDB	RonDB (MySQL Cluster)	Dynamo DB / Redis	In-memory optimized DB
Consistency	Eventual	Strong (within transactions)	Eventual	Strong
Streaming Aggregations	Via external engines (Flink)	Built-in	Built-in	Built-in (Optimized SQL)

Figure 1 illustrates the data flow in an MLOps pipeline that ensures feature consistency via a Feature Store.

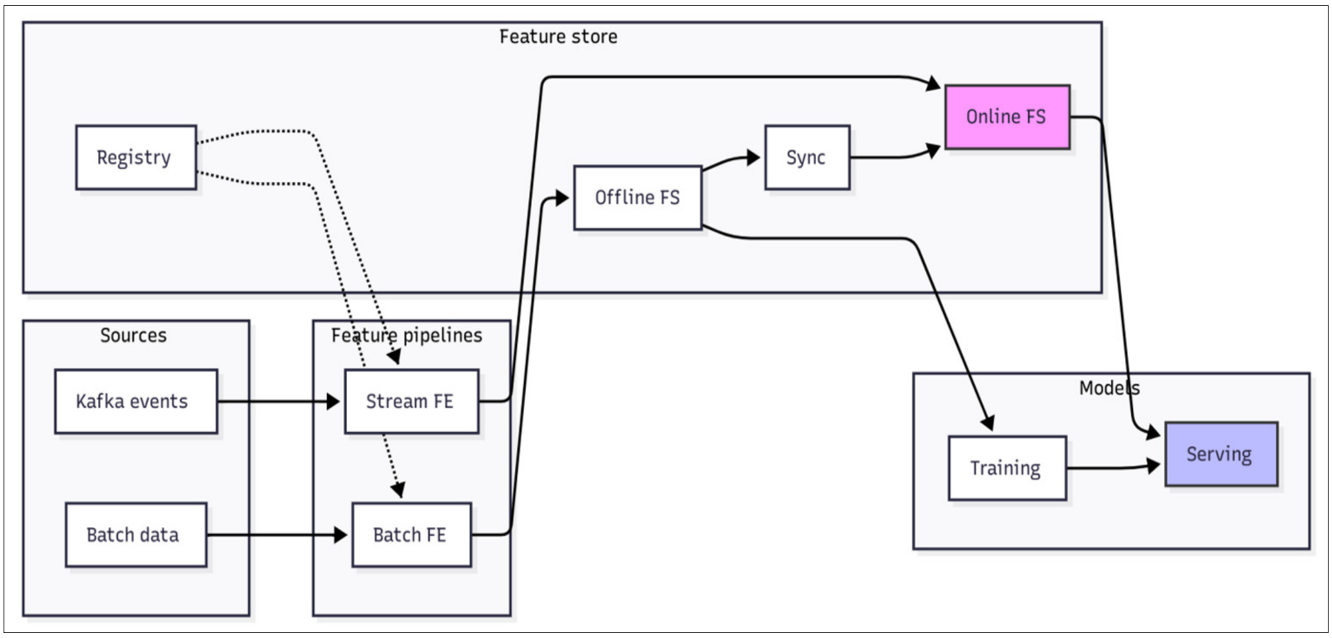

Fig. 1. End-to-End Feature Store Architecture.

The diagram shows how streaming data (RawEvents) are processed in real time, ingested into the online store for immediate use at inference time, and archived in the offline store for future retraining. The feature registry ensures that feature definitions are identical along both processing paths.

From a business perspective, the ability to compute features over streaming data with millisecond-level latency makes it possible to expose near-real-time signals to marketing and merchandising logic. Examples include session-level engagement scores, dynamic price sensitivity indicators, and short-term demand surges driven by campaigns or external events. When these features are consistently available both to training pipelines and to online serving, recommendation and ranking models can react within minutes rather than days to changes in user behavior or campaign configuration. Empirically, this shortens the feedback loop between marketing experiments and observable KPI shifts, enabling more aggressive A/B testing of promotion strategies without sacrificing user experience during traffic spikes.

The Problem of Embedding Scalability and Hierarchical Storage Systems

Modern recommender systems are based on models that use embeddings (dense vector representations) to encode sparse categorical features. Unlike computer vision (CNN) or natural language processing (Transformer) models, where dense computations (GEMM) dominate, DLRM architectures spend a substantial portion of their time on memory lookups [9].

GPU Memory Constraints

The problem arises because the size of embedding tables in industrial systems (e.g., Facebook, Google, Alibaba) can reach hundreds of gigabytes or tens of terabytes. This is due to the enormous number of users and items, as well as the use of combinatorial features (cross-features). Modern GPUs (such as NVIDIA A100 or H100) have limited memory capacity (40–80 GB), making it impossible to fit the entire model on a single device [10].

Traditional model parallelism (distributing a model across multiple GPUs) becomes economically inefficient at terabyte-scale embeddings, since it requires hundreds of GPUs purely for parameter storage, while compute cores may remain underutilized.

Hierarchical Parameter Server (HPS)

A solution to this problem is adopting a hierarchical parameter server architecture. This approach, implemented for example in the NVIDIA Merlin HugeCTR framework, uses a multi-level storage hierarchy analogous to CPU memory hierarchies [11]. In the context of product recommendation, such an architecture is not only a technical optimization, but also a way to control unit economics: it allows keeping the most commercially essential entities (high-margin and high-traffic SKUs, active loyalty users, key promotional inventories) in the fastest memory tier, while offloading the long tail of the catalog to cheaper storage. As a result, the system can sustain the required recommendation quality and latency for revenue-critical segments without linear growth of accelerator count and infrastructure costs.

In the embedding storage hierarchy, the first level, the L1 cache, resides in GPU memory and stores hot embeddings, that is, vectors corresponding to the most frequently requested IDs. Request distributions in recommender systems typically follow a power law: a relatively small fraction of objects, for example, around 20% of items, accounts for up to 80% of all views. Placing the corresponding embeddings in fast GPU memory (HBM) is critical for overall system performance, as it minimizes access latency for the most in-demand data.

The second level, the L2 cache, is located in the host's main memory (CPU RAM) and stores warm embeddings, that is, less popular but still regularly used vectors. Modern servers can be equipped with terabytes of RAM, enabling the storage of tables significantly larger than what fits in GPU memory. Access to these data occurs over PCIe or, in more specialized systems such as DGX, over high-speed interconnects like NVLink, providing an acceptable compromise between capacity and latency.

The third level, L3 storage, is represented by persistent storage based on solid-state drives (SSD), typically NVMe. This level holds cold embeddings, which are rarely accessed, as well as complete copies of feature tables. Although SSD access is substantially slower than GPU or main memory access, use of this level allows scaling the system to huge data volumes while retaining the ability to restore and periodically load rarely used vectors into faster cache levels as needed.

Analysis of the HPS architecture shows that it effectively hides the latency of accessing slower memory tiers through asynchronous prefetching and pipelining [12]. The HugeCTR system, integrated with Triton Inference Server, allows this hierarchy to be used transparently to the developer, automatically managing the movement of embeddings between levels based on access frequency.

For business stakeholders, this means that catalog growth and assortment rotation can proceed without proportional increases in serving costs. The ratio of infrastructure spend to incremental gross merchandise value (GMV) generated by recommendations becomes more favorable, particularly in markets with thin margins and frequent price changes. Moreover, explicit separation of hot and cold segments in the embedding hierarchy simplifies the design of differentiated commercial strategies, for example, more aggressive cross-selling for fast-moving consumer goods versus more conservative exposure of long-tail items.

Algorithmic Optimization: Quantization and Budgeted Embeddings

In addition to hardware solutions, quantization is an effective way to reduce memory pressure. Converting embeddings from FP32 (32-bit floating-point) to FP16 or INT8 reduces memory consumption by a factor of 2–4. Study [9] shows that using mixed precision has a negligible effect on ranking metrics (AUC/NDCG) while substantially increasing memory bandwidth.

A more advanced method is the use of Budgeted Embedding Tables (BET) [13]. Instead of using a fixed vector dimensionality (e.g., d = 64) for all entities, BET dynamically assigns dimensionality depending on feature frequency. Popular items receive higher-dimensional vectors for better representational power, while rare items (the tail of the distribution) are encoded using lower-dimensional vectors. This allows significant model compression without losing information about essential entities.

Optimization of Real-Time Inference Processes

The inference stage has latency requirements, which are the most demanding of all stages since they determine the storefront, search, and personalization widgets that users see. During periods of high load, the server must handle thousands of concurrent requests with the lowest response time possible to avoid degrading click-through rate, conversion rate, and basket completion during peak traffic. In practice, this implies that optimizations at the inference layer must be evaluated not only in terms of queries per second, but also in terms of their impact on abandonment rates, session depth, and the stability of key commercial KPIs during large-scale campaigns.

Dynamic Batching

Processing each request individually (batch size = 1) is highly inefficient for GPUs, which are designed for massive parallel computations. Kernel launch overhead and data transfer costs may exceed the time required for practical computation.

Dynamic batching addresses this issue by aggregating incoming requests into a buffer over a short time window (e.g., 1–5 ms) or until a predefined batch size is reached. The resulting batch is then sent to the GPU as a single tensor [14]. This approach dramatically increases system throughput at the cost of a slight and controllable increase in per-request latency. In e-commerce, dynamic batching is a form of traffic shaping. When the batching window and maximum batch sizes are set correctly, this can reduce latency by several milliseconds, while providing a much larger increase in throughput and temporarily reducing the risk of timeouts or downgrades at campaign start. This allows for more aggressive marketing, such as flash sales and coupon pushes, as well as remarketing banners.

Tool analysis demonstrates that NVIDIA Triton Inference Server provides one of the most advanced implementations of dynamic batching. Unlike TensorFlow Serving, which also supports this feature, Triton enables more flexible configuration of queue priorities and timeout strategies, and supports heterogeneous batches (combining requests for different versions of the same model) [15].

Comparative Analysis of Inference Servers

The choice of inference server is a critical architectural decision. In current practice, several key distinctions among popular model-serving solutions can be identified.

TensorFlow Serving (TFS) is the de facto standard in the TensorFlow ecosystem and provides high performance for TF-based models, including through XLA compilation optimizations [16]. However, support for models developed in other frameworks (PyTorch, ONNX) is limited and often requires additional conversion. The TFS architecture is tightly coupled to TensorFlow's computational graph representation, which may introduce overhead when using nonstandard operations and heterogeneous technology stacks.

TorchServe is a native solution for PyTorch models, co-developed by AWS and Facebook [17]. It is designed to simplify deployment and supports the TorchScript format, which facilitates the transfer of models to production. In practice, TorchServe demonstrates strong performance and ease of operation, but in some benchmarks, it falls short of specialized C++-based solutions in terms of maximum throughput under high load.

NVIDIA Triton Inference Server is a multi-framework server that supports TensorFlow, PyTorch, ONNX, TensorRT, and XGBoost [18]. Its key advantage lies in its ability to concurrently execute multiple models on a single GPU, where different models or numerous instances of the same model simultaneously utilize various sets of streaming multiprocessors or share resources over time. Triton also integrates with HugeCTR, which optimizes the handling of large embedding tables in recommender systems. Comparison of inference servers for high-load RecSys shown in table 2 below.

Table. 2. Comparison of inference servers for high-load RecSys.

Feature	**TensorFlow Serving**	**TorchServe**	**NVIDIA Triton Inference Server**
Backend	TensorFlow (C++)	PyTorch (Java/Python)	C++ (custom backends)
Dynamic Batching	Yes	Yes	Yes (advanced scheduling)
Embedding Cache	No (requires external solution)	No	Yes (via HugeCTR backend)
Concurrent Execution	Limited	Limited	Full support (MIG, streams)
Protocols	gRPC, REST	gRPC, REST	gRPC (optional), REST, C API
Typical Use Case	Pure TF pipeline	Pure PyTorch pipeline	Heterogeneous models, maximum performance

The diagram in Figure 2 illustrates the request processing flow in Triton with dynamic batching and hierarchical embedding caching.

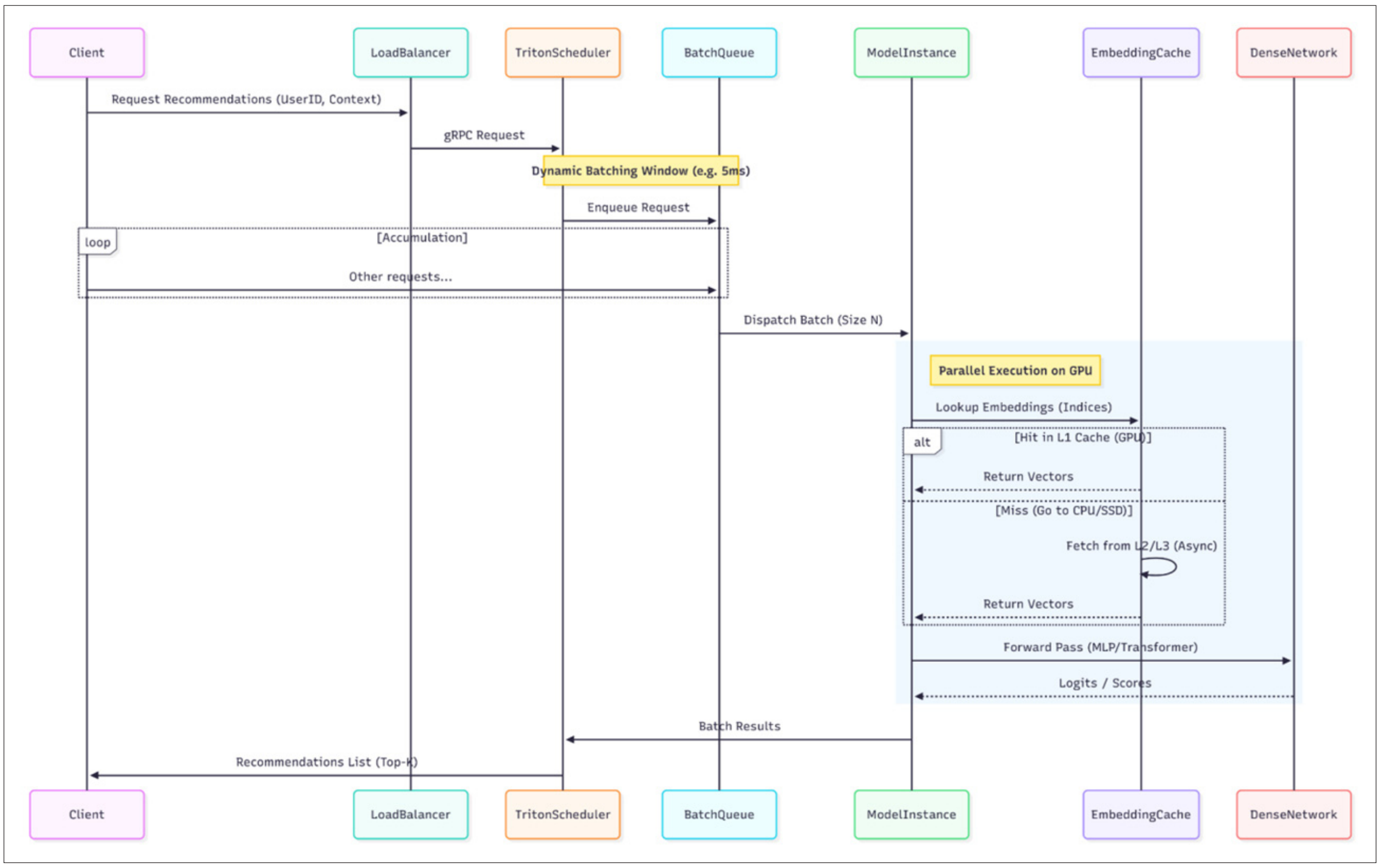

Fig. 2. Triton Inference Flow with Dynamic Batching and Embedding Cache

Adaptivity and Continuous Training: Coping with Drift

The rapid evolution of user preferences in e-commerce and media implies that static recommendation models quickly lose relevance and degrade in quality. This phenomenon is usually described in terms of drift, which is typically classified into several types.

Data drift refers to changes in the statistical distribution of input features P(X). An example might be a new type of product or content category that becomes very popular in a short time. If the datasets and user profiles are not updated to reflect the new product type, the model's old patterns will be outdated and no longer relevant.

Concept drift is the change in the association between a set X of explanatory attributes and the target variable Y, defined as P(Y|X), whereby the explanatory attributes remain unchanged while their dependence on Y varies. Concretely, an individual's purchasing inclination and behavior during holiday sales, pandemics, or other external events may change, thereby altering how a user reacts to the same recommendations.

Drift Monitoring

Timely detection of model degradation and system-level issues requires the implementation of a comprehensive monitoring system that tracks not only infrastructure metrics such as CPU and GPU utilization, response latency, and throughput, but also statistical properties of incoming data and model predictions. This approach enables the detection of changes in traffic patterns and user behavior before they translate into noticeable declines in business metrics.

Key statistical metrics include the Population Stability Index (PSI), which measures differences between feature distributions at training time and at inference time. Significant PSI deviations indicate that the current data stream differs substantially from the conditions under which the model was trained. In addition, KL divergence is used to quantify discrepancies between probability distributions of model predictions, for example, the distribution of click or conversion probability scores.

Online quality metrics also play an essential role, such as realized CTR, conversion rate, revenue per session, and average order value, measured with a time delay after user feedback has been observed. Analyzing the dynamics of these indicators, in conjunction with statistical metrics on the data and predictions, enables not only the detection of drift and model degradation, but also the measurement of the impact of changes in architecture, hyperparameters, or training strategies on commercial outcomes. In practice, tying drift alerts to thresholds on business metrics (for example, a relative drop in CTR or revenue per mille recommendations) allows product and marketing teams to reason about model health in familiar KPI terms and to prioritize remediation according to estimated revenue at risk.

Retraining Strategies

Since retraining under heavy load can take days, retraining a model on historical data cannot be used in real-time systems, and MLOps infrastructure must accommodate more real-time and responsive ways to improve a model. A third approach is incremental learning (fine-tuning), where the weight parameters of a deployed model are updated via new data (e.g., data from the last few hours) to reflect recent user interests to quickly capture changes in trends, user interests, and user behavior while preserving what has been learned about long-term user interests. This can be seen as a warm start, where the model is not retrained from scratch but is refined with each new data point.

A separate direction is represented by architectures such as Ekko [19]. In these systems, a peer-to-peer mechanism for propagating model updates across infrastructure components is proposed. Instead of the traditional process where a checkpoint with updated weights is first saved to disk and then loaded by inference servers (a process that may take minutes), gradient updates are transmitted directly between training and inference workers. This dramatically reduces model update latency, from tens of minutes to seconds, down to 2.4 seconds in experimental studies. Such responsiveness is crucial for news recommendation systems, where content relevance decays rapidly.

The Monolith system by ByteDance advances this approach further by implementing fully streaming, real-time training [20]. Its embedding tables are built on collisionless hash structures and eviction mechanisms for stale entries. This enables the model to train continuously on the Kafka event stream, updating user and item representations as soon as new activity occurs. Consequently, the system can react almost instantaneously to new user actions, minimizing the lag between behavioral changes and corresponding model adaptation.

Vector Search in High-Load Systems

At the candidate generation stage, the system must select a few hundred items from the most relevant millions. For this purpose, Approximate Nearest Neighbor (ANN) search methods are used [21].

Under high load, classical nearest-neighbor approaches such as KD-tree-based structures become ineffective due to the high dimensionality of feature spaces and large data volumes. This has been especially important in recommender systems and embedding-based web search, where millions or billions of vectors must be processed in near real time. In these applications, the industry has increasingly turned to specialized ANN algorithms and libraries, which strike a balance between accuracy and performance.

Hierarchical Navigable Small World (HNSW) is one of the more recent ANN standards, which uses multilayer small-world graphs for efficient search in the embedding space. It has near-logarithmic search time and a high recall (of vectors from the index) for a given accuracy, and has a good trade-off between speed, accuracy, and memory-usage, making it suitable for low-latency systems.

Another key solution is the Faiss (Facebook AI Similarity Search) library, explicitly designed for highly efficient search over extensive collections of vector representations. Faiss implements a variety of indexes, including IVF (Inverted File Index) and PQ (Product Quantization), with GPU support. The combination of IVF and PQ enables aggressive index compression, sometimes by an order of magnitude or more compared to naive vector storage, while simultaneously accelerating search by reducing the amount of data processed per query. At the same time, a satisfactory level of accuracy is preserved, making Faiss an industry standard for scalable embedding search.

Using GPUs for vector search (e.g., via Faiss-GPU or NVIDIA cuVS) enables thousands of queries to be processed in parallel, achieving orders-of-magnitude higher throughput than CPU-only solutions. Integrating these libraries into Triton (through custom backends or the Python backend) enables end-to-end GPU pipelines, avoiding the overhead of data transfers between CPU and GPU.

For product recommendations, the configuration of ANN indexes and the associated latency–accuracy trade-offs directly translate into business impact. A higher recall in the candidate generation stage increases the probability that downstream ranking models will consider high-margin, promotion-relevant, or inventory-critical items. At the same time, stable low latency ensures that these gains are not offset by user frustration or session abandonment. Consequently, vector search should be treated as a controllable lever in the commercial strategy: different index configurations and search budgets can be aligned with the objectives of specific campaigns (e.g., clearance of overstocked items versus maximization of basket value in premium categories) and evaluated through online experiments on standard marketing KPIs.

DISCUSSION

The results obtained show that optimizing MLOps processes for product recommendation systems under high load should be considered a specialized engineering discipline rather than a straightforward transfer of generic machine learning practices into the e-commerce domain. Unlike media or content recommendations, latency and output quality in product recommendations are directly linked to conversion, average order value, bounce rate, and session depth, especially during peak traffic periods such as sales and promotional campaigns. Under these conditions, MLOps becomes a key mechanism for maintaining the stability and predictability not only of models, but also of the business logic of storefronts, search results, dynamic merchandising, and promo placements.

It is shown that transitioning to centralized feature management via a Feature Store with strict consistency between offline and online processing is the foundation of reliable product recommendation operation: it reduces the risk of inconsistencies in prices, stock levels, promo rules, and segmentation, which, in a high-load environment, are instantly amplified across a large user base.

Analysis of embedding storage architectures demonstrates that, for product recommendation systems operating over large, continuously changing catalogs, hierarchical parameter servers and algorithmic representation compression become critically important. The use of multi-level embedding caches, in which hot items and active users are retained in GPU memory, while the warm and cold portions of the catalog are moved to cheaper memory tiers, enables sustaining high RPS without exponential growth in the number of accelerators. At the same time, dynamically allocating higher-dimensional embeddings to the most in-demand and high-margin SKUs yields an additional gain in memory footprint and throughput. However, such an architecture increases the dependency of recommendation quality on the correctness of caching and monitoring policies: any errors in parameter hierarchy management under high load may degrade performance precisely in those catalog segments that generate the most revenue.

The approaches to inference optimization considered here show that, for high-load product recommendation systems, dynamic batching and concurrent model execution on the inference server should be treated as components of traffic management. Batching configuration, request prioritization, and model placement schemes on accelerators must be tuned to the storefront's traffic profile: spikes in requests to the home page, category listings, and the cart; differences between mobile and web traffic; and user behavior at campaign launch time. More aggressive request aggregation and the use of heterogeneous batches can significantly increase throughput, but they demand precise control over latency in critical user journeys. Vector search as an extension of candidate generation models plays an equally central role: the choice of indexes and parameters for approximate embedding search directly affects output relevance and latency stability when working with millions of item vectors. It therefore must be governed by the same MLOps processes as the rest of the recommendation pipeline.

Finally, integrating drift monitoring and continuous training loops specifically in the context of high-load product recommendations shows that the minimal delay between changes in assortment and user behavior and the delivery of an updated model to production determines system resilience. E-commerce event streams, new items, price changes, promotion start and end, seasonal demand spikes, create an extremely dynamic environment in which infrequent batch model updates are insufficient. Incremental or streaming training over real-time event flows, combined with automated monitoring of data, predictions, and business metrics, can substantially reduce this gap but requires mature processes for versioning, testing, and risk management.

Taken together, the results indicate that for product recommendation systems under high load, MLOps must evolve into a comprehensive engineering standard in which architectural decisions on data, embeddings, inference, and training are designed jointly, with explicit consideration of peak loads, infrastructure costs, and target commercial KPIs.

CONCLUSIONS

Optimizing MLOps for high-load product recommendation systems requires shifting from static pipeline paradigms to dynamic, reactive, and hardware-aware architectures. The analysis conducted allows the following key conclusions and recommendations to be formulated.

Data consistency must be ensured at the architectural level by introducing a Feature Store that supports streaming aggregations and by using optimized in-memory stores (Redis, RonDB, OpenMLDB) for inference. The embedding scale problem is effectively addressed by employing hierarchical parameter servers (HPS) that utilize the whole memory hierarchy (GPU HBM → RAM → SSD) in conjunction with quantization techniques and dynamic vector dimensionality selection.

Maximizing inference throughput is achieved through dynamic request batching and the use of specialized servers (NVIDIA Triton) that support concurrent model execution and optimized backends (TensorRT). Adaptivity to change is ensured by transitioning to incremental or online training, backed by architectures for fast model update delivery (Ekko/Monolith-type systems) and continuous concept drift monitoring.

The future development of this area is inextricably linked to the integration of Large Language Models (LLMs) into recommendation pipelines (Generative RecSys). This will require adapting optimization methods developed for LLMs (KV caching, PagedAttention) to the specifics of recommendation tasks, as well as further evolution of hardware accelerators for efficient sparse-data processing.

REFERENCES

1. S. Raza, M. Rahman, S. Kamawal, A. Toroghi, and A. Kazemeini, “A Comprehensive Review of Recommender Systems: Transitioning from Theory to Practice,” *Arxiv*, Jul. 2024, doi: https://doi.org/10.48550/arXiv.2407.13699.
2. M. Basalla, J. Schneider, M. Luksik, R. Jaakonmäki, and J. Vom Brocke, “On Latency of E-Commerce Platforms,” *Journal of Organizational Computing and Electronic Commerce*, vol. 31, no. 1, pp. 1–17, Jan. 2021, doi: https://doi.org/10.1080/10919392.2021.1882240.
3. Q. Wang, B. Sang, H. Zhang, M. Tang, and K. Zhang, “DLRover-RM: Resource Optimization for Deep Recommendation Models Training in the Cloud,” *Arxiv*, Apr. 2023, doi: https://doi.org/10.48550/arxiv.2304.01468.
4. Y. Park, J. Mun, Y. Lee, J. Um, J. Choi, and J. Choi, “Data-Driven Optimization of Healthcare Recommender System Retraining Pipelines in MLOps with Wearable IoT Data,” *Sensors*, vol. 25, no. 20, p. 6369, Oct. 2025, doi: https://doi.org/10.3390/s25206369.
5. K. Shivashankar, A. Hajj, and A. Martini, “Scalability and Maintainability Challenges and Solutions in Machine Learning: Systematic Literature Review,” *Arxiv*, Apr. 2025, doi: https://doi.org/10.48550/arXiv.2504.11079.
6. A. Jose and S. D. Shetty, “DistilledCTR: Accurate and scalable CTR prediction model through model distillation,” *Expert Systems with Applications*, vol. 193, p. 116474, May 2022, doi: https://doi.org/10.1016/j.eswa.2021.116474.
7. J. de la Rúa Martínez *et al.*, “The Hopsworks Feature Store for Machine Learning,” *Companion of the 2024 International Conference on Management of Data*, pp. 135–147, Jun. 2024, doi: https://doi.org/10.1145/3626246.3653389.
8. M. A. Sidiq, A. A. Salih, and S. M. Hassan, “Optimization Techniques For SQL+ML Queries: A Performance Analysis Of Realtime Feature Computation In OpenMLDB,” *International Journal of Database Management Systems*, vol. 17, no. 5, pp. 1–12, Oct. 2025, doi: https://doi.org/10.5121/ijdms.2025.17501.
9. S. Li *et al.*, “Embedding Compression in Recommender Systems: A Survey,” *ACM Computing Surveys*, vol. 56, no. 5, pp. 1–21, Jan. 2024, doi: https://doi.org/10.1145/3637841.
10. Z. Liu, Q. Song, L. Li, S.-H. Choi, R. Chen, and X. Hu, “PME: pruning-based multi-size embedding for recommender systems,” *Frontiers in Big Data*, vol. 6, Jun. 2023, doi: https://doi.org/10.3389/fdata.2023.1195742.
11. Merlin HugeCTR, “Hierarchical Parameter Server Database Backend,” *Github*. https://nvidia-merlin.github.io/HugeCTR/main/hugectr_parameter_server.html (accessed Nov. 10, 2025).
12. Y. Wei *et al.*, “A GPU-specialized Inference Parameter Server for Large-Scale Deep Recommendation Models,” *Sixteenth ACM Conference on Recommender Systems*, Sep. 2022, doi: https://doi.org/10.1145/3523227.3546765.
13. Y. Qu, T. Chen, Q. V. H. Nguyen, and H. Yin, “Budgeted Embedding Table For Recommender Systems,” *Arxiv*, Oct. 2023, doi: https://doi.org/10.48550/arxiv.2310.14884.
14. F. Yu *et al.*, “Characterizing and understanding deep neural network batching systems on GPUs,” *BenchCouncil Transactions on Benchmarks, Standards and Evaluations*, vol. 3, no. 4, p. 100151, Dec. 2023, doi: https://doi.org/10.1016/j.tbench.2024.100151.
15. NVIDIA, “Ragged Batching - NVIDIA Triton Inference Server,” *NVIDIA*. https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/user_guide/ragged_batching.html (accessed Nov. 14, 2025).
16. OpenXLA, “XLA: Optimizing Compiler for Machine Learning,” *OpenXLA*, 2025. https://openxla.org/xla/tf2xla (accessed Nov. 15, 2025).

17. Pytorch, “Performance Guide,” *Pytorch*. https://docs.pytorch.org/serve/performance_guide.html (accessed Nov. 16, 2025).

18. NVIDIA, “Concurrent Model Execution,” *NVIDIA*. https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/user_guide/model_execution.html (accessed Dec. 17, 2025).

19. H. Lee, S. Yoo, D. Lee, and J. Kim, “How Important is Periodic Model update in Recommender System?” *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 2661–2668, Jul. 2023, doi: https://doi.org/10.1145/3539618.3591934.

20. Z. Liu *et al.*, “Monolith: Real Time Recommendation System With Collisionless Embedding Table,” *Arxiv*, Sep. 2022, doi: https://doi.org/10.48550/arxiv.2209.07663.

21. M. Douze *et al.*, “The Faiss library,” *Arxiv*, Jan. 2024, doi: https://doi.org/10.48550/arXiv.2401.08281.