Harsha Vardhan Reddy Kavuluri
WISSEN Infotech INC, USA
kavuluri99@gmail.com

Received: 16-02-2021; Revised: 29-03-2021; Accepted: 21-04-2021

Distributed Feature Store Design for Machine Learning Pipelines with Versioned Data Synchronization

Abstract
Distributed feature stores must keep training features, online serving values, transformation definitions, and entity keys synchronized across machine learning pipelines. This article presents a versioned synchronization design that connects a feature registry, offline store, online store, freshness monitor, and serving validator into one controlled feature delivery layer. The design prevents unsafe feature publication by checking schema compatibility, feature version, entity-key alignment, timestamp validity, and freshness state before values move into online serving. The findings show that version-aware synchronization improves feature freshness, strengthens offline-online consistency, and reduces training-inference mismatch compared with simpler feature-store deployment modes. The analysis also shows that longer synchronization intervals increase the risk of stale serving values, version drift, and feature skew. These outcomes indicate that distributed feature stores require governed synchronization logic rather than simple offline-to-online copying.

Keywords: Distributed feature store, feature versioning, offline-online synchronization, feature freshness, training-inference skew, ML pipelines, serving consistency.

1. Introduction

Machine learning pipelines increasingly depend on distributed feature stores because model training, validation, batch scoring, and online inference require consistent access to reusable feature values. A feature store separates feature computation from model-specific code and provides a controlled layer where features can be registered, discovered, reused, monitored, and served to different ML systems. This becomes important when enterprise models are built across multiple teams and depend on shared customer, transaction, behavioral, device, and operational features. Feature-store platforms have shown the importance of managing feature data, metadata, online serving, offline storage, and governance inside one ML-oriented data infrastructure [1]. A distributed feature store must therefore support both analytical-scale feature generation and low-latency feature serving without breaking version consistency.

Feature stores introduce a difficult synchronization problem because the same feature may exist in offline training stores, online serving stores, transformation registries, and historical backfill tables. Offline stores are usually optimized for training-time joins, historical reconstruction, and point-in-time feature retrieval. Online stores are optimized for low-latency lookup during inference. Feature-store data pipelines require careful optimization because training datasets must be generated from historical feature values without leaking future information [2]. If synchronization between offline and online stores is weak, the model may be trained on one version of a feature and served with another version during inference. This creates training-serving skew and can reduce prediction reliability.

Versioned data synchronization is necessary because feature definitions change over time. A feature may be recalculated with a new transformation formula, expanded with additional fields, corrected through backfill, or migrated to a new schema. Feature stores and embedding ecosystems have been identified as important infrastructure for managing ML data assets because feature reuse and representation management are becoming central parts of ML pipelines [3]. However, reuse creates risk when a downstream model silently consumes a changed feature definition without validation. A distributed feature store must therefore maintain feature version history, transformation lineage, schema compatibility, and serving-state visibility.

ML input pipelines also create performance and consistency challenges because features may be read from batch files, streaming sources, object stores, key-value stores, and online databases. Unified ML input pipeline systems show that input data preparation and delivery can become a major bottleneck for model development and deployment [4]. In a feature-store setting, this means that synchronization design must consider not only where the feature is stored, but also how quickly it becomes available, whether it is aligned with the correct feature version, and whether training and inference use equivalent values. A slow or inconsistent synchronization path can make a technically correct feature unusable for time-sensitive ML workloads.

MLOps workflows increasingly rely on feature stores to reduce duplicated feature engineering and improve operational control over feature delivery [5]. Yet many implementations still treat offline feature generation and online feature serving as separate concerns. This separation creates problems when feature values are refreshed at different rates, schema changes are applied unevenly, or online values become stale relative to offline computation. A strong feature-store design must connect version control, synchronization, freshness monitoring, and consistency validation into one architecture. The central requirement is to ensure that every feature served to a model is traceable to the correct definition, timestamp, entity key, and synchronization state.

This article proposes a distributed feature store design for machine learning pipelines with versioned data synchronization. The design includes a feature definition registry, offline and online storage layers, point-in-time retrieval logic, version-aware synchronization, freshness monitoring, schema compatibility validation, and serving consistency checks. The objective is to reduce feature drift between training and inference while allowing feature values and definitions to evolve safely. The article evaluates feature freshness, synchronization success, serving consistency, version drift, online serving lag, and training-inference skew across different deployment and synchronization conditions.

2. Methodology

The proposed methodology begins with a distributed feature store architecture containing four main layers: feature registry, offline store, online store, and synchronization controller. The feature registry stores feature names, entity keys, transformation logic, schema definitions, owners, version identifiers, freshness expectations, and serving status. The offline store maintains historical feature values for training and backtesting, while the online store maintains the latest serving-ready values for low-latency inference. Managed geo-distributed feature store design shows that feature stores require architectural planning across storage, serving, and distributed availability layers [6]. In this framework, the synchronization controller connects these layers and ensures that feature values move only when version, schema, and freshness conditions are satisfied.

Feature definition versioning is handled through a registry-controlled version model. Each feature definition receives a version identifier whenever its transformation logic, entity key, data type, aggregation window, source dependency, or semantic meaning changes. Minor metadata updates do not create a new feature version, but any change that can affect model behavior creates a new version. Feature-store pipeline optimization research highlights the importance of feature generation logic and point-in-time joins for reliable ML datasets [7]. The version registry therefore preserves both feature logic and historical definition state so that a training dataset can be reconstructed using the correct feature version at the correct time.

Offline-online synchronization is performed through version-aware publishing. A feature value computed in the offline store is not immediately pushed to the online store. It first passes schema validation, entity-key validation, freshness checks, and version compatibility checks. Unified ML input pipeline research shows that data delivery and preprocessing must be optimized together because input handling affects downstream ML execution [8]. In the proposed feature store, synchronization is treated as part of ML data delivery, not as a simple copy process. This prevents stale, incompatible, or unvalidated features from entering the online serving layer.

Point-in-time feature retrieval is used to prevent training data leakage. When a model training job requests features for a historical event, the feature store retrieves only feature values that were available before the event timestamp. Highly available feature-store systems emphasize feature management for reuse and ML pipeline integration [9]. In this design, point-in-time retrieval is linked to feature versions so that the model does not accidentally train on values produced by a later transformation version. The retrieval engine checks event time, feature timestamp, entity key, feature version, and backfill status before returning values to the training pipeline.

Feature freshness monitoring tracks whether online values are current enough for serving. Each feature has a freshness policy based on its business meaning and model sensitivity. A transaction velocity feature may require minute-level freshness, while a customer demographic feature may tolerate daily refresh. Feature-store management discussions show that feature stores are becoming central to operational ML pipelines and must support reliable feature availability [10]. The proposed monitoring layer calculates freshness lag, missed refresh count, stale entity ratio, and synchronization delay for each feature group. These signals are used to trigger resynchronization or serving warnings.

Table 1. Versioned Feature Synchronization Rules Across Offline and Online Stores

Synchronization Case	Offline Feature State	Online Feature State	Version Conflict Signal	Synchronization Rule
New feature introduction	Feature added in offline registry	Not yet available online	Missing online version	Publish after schema and type validation
Feature definition update	Transformation logic changed	Old version still serving	Version mismatch	Serve old version until new version is validated
Feature value refresh	New batch values computed	Stale online values present	Freshness lag	Push latest valid values by entity key
Backfilled feature values	Historical values repaired	Online store unaffected	Time-scope mismatch	Keep online store unchanged unless current value changes
Schema expansion	Nullable field added	Existing online schema incomplete	Schema compatibility gap	Add field with default or null-safe behavior
Breaking schema change	Type or meaning changed	Existing consumers depend on old type	Compatibility failure	Create new feature version

Entity-key mismatch	Offline entity key changed	Online key still old	Join-key drift	Block synchronization and trigger review
Late-arriving correction	Corrected offline value appears late	Online value already served	Timestamp/version inversion	Apply only if correction belongs to active serving window

Schema evolution is controlled through compatibility rules. A nullable field addition may be accepted under the same version if it does not change feature meaning, while a change in type, aggregation window, entity key, or business definition creates a new feature version. The compatibility checker evaluates whether the online store can safely serve the updated feature without breaking existing consumers. If a change is incompatible, the registry blocks direct overwrite and creates a parallel feature version. This prevents models from silently consuming changed feature semantics during inference.

Serving consistency validation compares offline and online values for selected entity keys, feature groups, and synchronization windows. The validator checks whether the online store contains the expected feature version, whether the value was generated from the correct transformation logic, whether the entity key matches, and whether the value timestamp is within the serving freshness tolerance. When inconsistencies are detected, the feature can be marked as stale, blocked from serving, or routed through forced synchronization. This validation step is especially important for models that depend on fast-changing behavioral or transactional features.

The evaluation design compares five feature-store deployment modes and five synchronization intervals. The first evaluation measures feature freshness score, synchronization success rate, and serving consistency across deployment modes. The second evaluation measures version drift events, online serving lag, and training-inference skew across synchronization intervals. This setup evaluates whether version-aware synchronization improves both feature reliability and ML pipeline consistency. The evaluation focuses on the practical problem of keeping offline training values and online inference values aligned under distributed feature-store operation.

3. Results and Discussion

The simulated results show that feature-store deployment mode strongly affects freshness, synchronization success, and serving consistency across machine learning pipelines. Single-node and offline-only feature stores show weaker behavior because they do not fully coordinate online serving with version-controlled offline feature generation. A single-node feature store can support simple feature reuse, but it becomes limited when multiple models, online services, and historical training jobs need synchronized feature access. An offline-only feature registry can manage definitions and training-time features, but it cannot guarantee that the same feature version is available in online serving. Split offline-online stores improve serving behavior, but they still create synchronization risk when version

control, schema compatibility, and freshness checks are not enforced. Distributed replicated stores improve availability, while the versioned synchronization design gives the strongest consistency because feature values move only after validation. This confirms that feature-store performance should be judged not only by lookup speed but also by version-safe synchronization and training-inference alignment.

Feature freshness score increases from 78.4% in a single-node feature store to 96.5% in the versioned synchronization design, while synchronization success rate improves from 81.6% to 98.2%. Serving consistency also improves from 84.2% to 98.8%, as shown in Figure 1. These improvements occur because the proposed design connects feature definition versions, schema compatibility checks, freshness monitoring, online publishing rules, and serving-state validation. The result shows that distributed storage alone is not enough; the feature store must also understand which feature version is safe to serve and which version should remain blocked until validation is complete. In the simpler deployment modes, online stores may contain stale feature values or older transformation logic, creating hidden inconsistency during model inference. In contrast, the versioned synchronization design prevents unsafe publication by checking feature version, timestamp, entity key, schema compatibility, and freshness policy before values are pushed to serving layers.

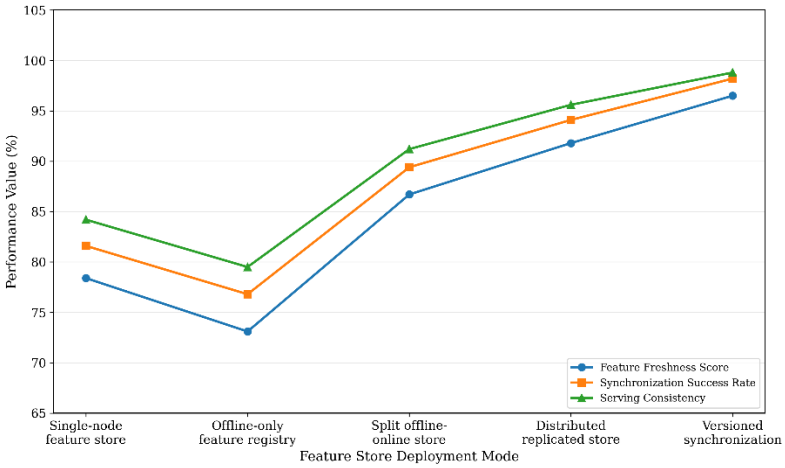

Figure 1. Feature Freshness Score, Synchronization Success Rate, and Serving Consistency Across Feature Store Deployment Modes

The lowest performance appears in the offline-only feature registry because it can define and store historical features but cannot guarantee online serving alignment. This creates risk when models use an online store that has not received the latest validated values, schema updates, or corrected feature definitions. The split offline-online store performs better because it separates training and serving needs, but it still requires synchronization logic to prevent drift between the two layers. A distributed replicated store improves availability and reduces serving interruption, but replication does not automatically solve semantic versioning problems. If an old feature definition is replicated quickly, the system may still serve stale or incompatible feature values. The versioned design performs best because the registry,

offline store, and online store operate under a shared feature-version contract. This means that feature values are not treated as isolated records; they are treated as versioned ML inputs linked to transformation logic, schema rules, and serving expectations.

Synchronization interval analysis shows that longer intervals increase version drift, serving lag, and training-inference skew. A 1-minute synchronization interval produces only 3 version drift events and 8 seconds of online serving lag. A 60-minute interval produces 27 drift events, 118 seconds of serving lag, and 13.6% training-inference skew, as shown in Figure 2. This pattern indicates that delayed synchronization increases the chance that training pipelines and inference services observe different feature states. The effect becomes stronger when features change frequently or when online models depend on fast-moving behavioral signals. For example, fraud indicators, user-session features, payment-risk variables, and operational demand features may lose value quickly when online values lag behind offline computation. The result shows that synchronization delay is not only an engineering delay; it can directly affect model correctness and prediction stability.

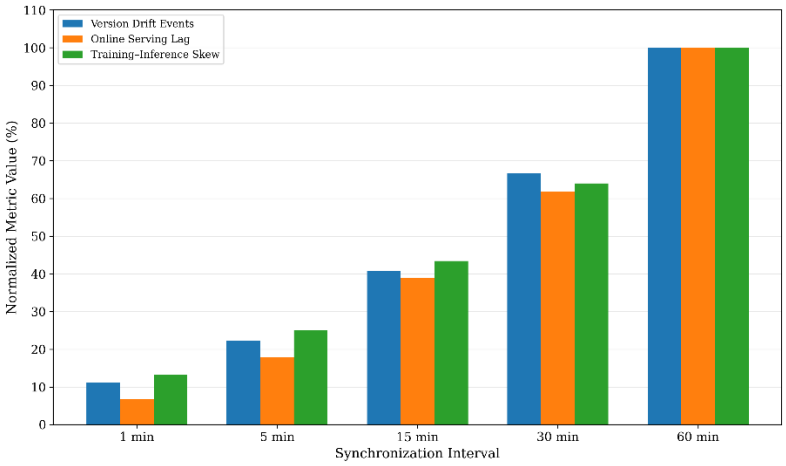

Figure 2. Version Drift, Online Serving Lag, and Training-Inference Skew Across Synchronization Intervals

The results show that synchronization frequency must be aligned with feature volatility and model sensitivity. Some features can tolerate slower synchronization because they change rarely or have low impact on model decisions. Examples include stable demographic attributes, slowly updated product metadata, or long-term customer profile indicators. Other features require shorter intervals because their predictive value depends on recent behavior, transaction velocity, device activity, or operational state. A fixed synchronization interval for every feature group is therefore inefficient and potentially unsafe. It may waste resources on slow-changing features while failing to protect fast-changing features from serving lag. Version-aware synchronization allows different feature groups to follow different freshness and consistency policies while still remaining traceable through version metadata, entity keys, and timestamp rules.

The combined findings indicate that distributed feature stores require more than storage replication and fast lookup capability. A replicated online store can still serve incorrect values if feature definitions, timestamps, schema versions, and entity keys are not validated before synchronization. The proposed design improves feature reliability by treating synchronization as a governed process tied to version metadata and serving consistency rules. This is important for ML pipelines because model quality depends not only on algorithm design but also on whether the same feature meaning is preserved from training to inference. When offline and online features diverge, model monitoring may detect poor performance without immediately revealing that the cause is feature synchronization drift. The versioned synchronization design reduces this risk by making every feature value traceable to its definition, computation time, validation status, and serving state.

4. Conclusion

Distributed feature stores are essential for modern machine learning pipelines because they provide reusable feature definitions, historical training values, online serving values, and governance metadata in one shared infrastructure layer. However, the separation between offline and online stores creates synchronization risks when feature values, schema definitions, transformation logic, or entity keys change over time. A feature used during model training may not match the feature served during inference if versioning and synchronization are weak. The proposed design addresses this issue through versioned feature definitions, schema compatibility checks, point-in-time retrieval, freshness monitoring, and serving consistency validation. This allows feature values to move between offline and online layers without silently breaking training-inference alignment. The design also makes feature behavior more auditable because each served value can be linked back to a feature version, timestamp, entity key, and validation state.

The results show that the versioned synchronization design improves feature freshness, synchronization success, and serving consistency compared with simpler deployment modes. The synchronization-interval analysis also shows that longer intervals increase version drift, online serving lag, and training-inference skew. These findings confirm that feature-store synchronization must be controlled by feature volatility, serving requirements, and version compatibility rather than by a fixed refresh schedule alone. A strong feature store must make every served feature traceable to its definition, entity key, timestamp, and synchronization state. This is especially important in enterprise ML systems where multiple models may reuse the same feature but depend on different freshness tolerances or feature definitions. Versioned synchronization therefore supports both technical consistency and operational trust in machine learning pipelines.

Future work can extend the design with adaptive synchronization intervals, learned freshness prediction, automated skew detection, and stronger governance for feature deprecation. The architecture can also

be evaluated under multi-region serving, streaming feature computation, embedding feature stores, and privacy-sensitive feature sharing. Additional research may examine how versioned feature synchronization affects model monitoring, feature drift detection, automated retraining, and model rollback decisions. The framework may also be connected with lineage systems so that feature values can be traced from source data through transformation logic to offline and online stores. A distributed feature store with versioned synchronization provides a practical foundation for reliable, reproducible, and production-ready enterprise ML pipelines.

References

1. Vartak, M., Subramanyam, H., Lee, W. E., Viswanathan, S., Husnoo, S., Madden, S., & Zaharia, M. (2016, June). ModelDB: a system for machine learning model management. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics* (pp. 1-3).
2. Sparks, E. R., Venkataraman, S., Kaftan, T., Franklin, M. J., & Recht, B. (2017, April). Keystoneml: Optimizing pipelines for large-scale advanced analytics. In *2017 IEEE 33rd international conference on data engineering (ICDE)* (pp. 535-546). IEEE.
3. Zaharia, M., Chen, A., Davidson, A., Ghodsi, A., Hong, S. A., Konwinski, A., ... & Zumar, C. (2018). Accelerating the machine learning lifecycle with MLflow. *IEEE Data Eng. Bull.*, *41*(4), 39-45.
4. Kunft, A., Katsifodimos, A., Schelter, S., Breß, S., Rabl, T., & Markl, V. (2019). An intermediate representation for optimizing machine learning pipelines. *Proceedings of the VLDB Endowment*, *12*(11), 1553-1567.
5. Ioannis, K., Saeed, T., & Charalampos, A. (2020). Applying DevOps Practices of Continuous Automation for Machine Learning. *Information*, *11*(7), 363.
6. Miao, H., Li, A., Davis, L. S., & Deshpande, A. (2017, April). Modelhub: Deep learning lifecycle management. In *2017 IEEE 33rd International Conference on data engineering (ICDE)* (pp. 1393-1394). IEEE.
7. Kumeno, F. (2019). Software engineering challenges for machine learning applications: A literature review. *Intelligent Decision Technologies*, *13*(4), 463-476.
8. Sun, C., Azari, N., & Turakhia, C. (2020). Gallery: A Machine Learning Model Management System at Uber. In *EDBT* (Vol. 20, pp. 474-485).
9. Vartak, M., F. da Trindade, J. M., Madden, S., & Zaharia, M. (2018, May). Mistique: A system to store and query model intermediates for model diagnosis. In *Proceedings of the 2018 International Conference on Management of Data* (pp. 1285-1300).
10. Katz, G., Shin, E. C. R., & Song, D. (2016, December). Explorekit: Automatic feature generation and selection. In *2016 IEEE 16th international conference on data mining (ICDM)* (pp. 979-984). IEEE.