Vol. 3, No. 2, pp.: 1-4, 2025

Building Real-Time Feature Stores: Architecture Patterns and Trade-offs at Scale

Yuki Nomura[1,*], **Danielle Okafor**[2], **Luca Ferraro**[1]

[1] Platform ML Engineering, Rakuten Group, Inc., Tokyo, Japan, 158-0094

[2] Department of Computer Science, University of Nairobi, Nairobi, Kenya, 00100

[*] *yuki.nomura@rakuten.com*

Article Information	
Received	14 January 2025
Accepted	22 June 2025
DOI	https://doi.org/10.63646/datamind.2025.030201

Abstract

Feature stores have emerged as a critical component of production machine learning infrastructure, providing a layer of abstraction between raw data and model training and serving that addresses the dual challenges of feature reuse and online/offline consistency. Despite their growing adoption, the design space for feature stores is poorly mapped in the academic literature, with most detailed discussions confined to engineering blog posts and conference talks from technology companies. This technical communication systematises the feature store design space through a taxonomy of five architectural patterns — Lambda architecture, Kappa architecture, pre-computed offline-only, on-demand online-only, and hybrid push-pull — and provides quantitative comparisons of each pattern across four operational dimensions: latency, throughput, freshness, and operational complexity. We describe our experience migrating from a Lambda-based feature store to a hybrid push-pull architecture serving over 800 models and 15,000 feature definitions at Rakuten Group, report on the specific engineering trade-offs encountered, and provide a decision framework for architecture selection based on feature type and serving latency requirements.

Keywords: ***feature store; MLOps; real-time ML; Lambda architecture; Kappa architecture; feature serving; ML infrastructure***

1. Introduction

The feature store problem is deceptively simple to state: you want to compute features from raw data, store them efficiently, and serve them consistently for both model training and online inference. The practice is considerably more complex. Training and serving environments have different computational constraints, different latency budgets, and different freshness requirements. Features that are trivially computed offline (seven-day rolling purchase value, account age in days) require real-time event streaming to serve freshly at inference time. And the operational overhead of maintaining consistency between the feature values used in training and those seen at serving time — training-serving skew — is one of the most common sources of silent production ML failures.

The term 'feature store' has been applied to an unusually wide range of systems, from simple Parquet file directories with a metadata layer to fully managed streaming pipelines with online serving at sub-millisecond P99 latency. This variety makes it difficult to compare approaches or make

principled architectural choices. Our goal in this technical communication is to reduce that confusion by providing a systematic taxonomy of architectural patterns and a quantitative framework for evaluating them.

2. Architectural Patterns

2.1 Lambda Architecture

The Lambda architecture partitions feature computation into a batch layer (producing accurate historical features from complete data, typically daily or hourly) and a speed layer (producing approximate but low-latency features from recent events). Both layers write to a serving layer. At training time, the batch layer provides the feature values; at serving time, recent speed layer values are merged with batch values to provide a complete, approximately fresh feature vector. This architecture is conceptually clean and operationally mature — it is the dominant pattern in legacy feature store deployments — but its dual-layer maintenance overhead is significant, and the merge semantics at serving time are a common source of subtle inconsistencies.

2.2 Kappa Architecture

The Kappa architecture eliminates the batch layer entirely, treating all feature computation as stream processing. Historical feature recomputation is handled by reprocessing the event log, typically stored in a durable message queue such as Apache Kafka. Kappa simplifies the operational surface but demands a streaming compute platform (Apache Flink, Spark Streaming) capable of handling both low-latency serving and high-throughput historical reprocessing. The key insight is that if your streaming platform is fast enough, you do not need a separate batch layer. The qualification is significant: Kappa works well for relatively simple aggregations over recent windows but struggles with complex historical aggregations that require full dataset passes.

2.3 Hybrid Push-Pull

The hybrid push-pull pattern separates feature definitions into two categories based on their freshness requirements. Static and slowly-changing features (user demographics, account-level aggregations with daily granularity) are pre-computed in batch and pushed to the online store on a schedule. Dynamic features with latency requirements below 100ms are computed on-demand from a combination of pre-aggregated state and real-time event data. This pattern requires maintaining two feature computation paths but allows each path to be optimised for its specific characteristics, avoiding the overhead of running a full streaming pipeline for features that do not require it.

Table 1. *Architectural pattern comparison across four operational dimensions. Scores from 1 (poor) to 5 (excellent). Scores reflect typical deployments; actual performance depends heavily on implementation choices.*

Pattern	Latency (P99)	Throughput	Freshness	Ops. Complexity
Lambda	5	4	3	2
Kappa	4	5	5	3
Pre-computed offline	5	5	1	5
On-demand online	3	2	5	2
Hybrid push-pull (our	4	4	4	3

Pattern	Latency (P99)	Throughput	Freshness	Ops. Complexity

approach)

Latency and throughput scores reflect typical configurations. Ops. Complexity is inverse: 5 = simpler to operate. Hybrid push-pull balances all dimensions better than pure approaches for diverse feature portfolios.

3. Migration Experience

3.1 From Lambda to Hybrid Push-Pull

Our migration at Rakuten Group was motivated by two specific operational failures of the Lambda architecture at scale. First, the dual-layer consistency problem: after our online catalogue expanded to 45 million products, the merge semantics between batch and speed layer values for product-level features produced sporadic inconsistencies that took six months to diagnose fully. Second, the batch layer lag: our daily batch computation cycle was taking 14 hours on the data volumes we were processing, meaning that features were frequently more than 24 hours stale during the cycle overlap period.

The migration to hybrid push-pull took 14 months and involved re-classifying 15,000 feature definitions into static (9,200 features) and dynamic (5,800 features) categories based on their staleness tolerance and query frequency. The classification was done through a combination of staleness sensitivity analysis (how much does model performance degrade if this feature is 6 hours stale?) and query profiling (how many inference requests per second does this feature serve?). The most surprising finding from this analysis was that approximately 40% of features classified as 'real-time' by their original authors were in practice insensitive to staleness up to 6 hours, allowing them to be reclassified to the lower-cost pre-computed path.

4. Decision Framework

Based on our experience and the architectural analysis above, we propose the following decision framework. If your feature portfolio is dominated by slow-changing or daily-granularity features, and your model inventory does not require sub-second serving latency, pre-computed offline is operationally simpler and should be your default. If sub-second serving latency is required for more than 20% of your features, and your team has streaming infrastructure expertise, Kappa provides the cleanest semantic model. For large, diverse feature portfolios like ours, hybrid push-pull provides the best balance of operational manageability and serving performance, at the cost of maintaining two feature computation paths.

5. Conclusion

The feature store is not a commodity component that can be selected from a catalogue without careful attention to the specific characteristics of your feature portfolio and serving requirements. The architectural patterns we have described occupy meaningfully different positions in the latency-freshness-complexity space, and the right choice depends on use case characteristics that are often not assessed systematically. We hope the taxonomy and decision framework provided here give ML platform engineers a more principled basis for feature store architecture decisions.

References

Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., & Xin, R. (2016). MLlib: Machine learning in Apache Spark. Journal of Machine Learning Research, 17(34), 1–7. https://jmlr.org/papers/v17/15-237.html

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., & Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. NSDI 2012, 2–2. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

Kreps, J., Narkhede, N., & Rao, J. (2011). Kafka: A distributed messaging system for log processing. NetDB Workshop, 1–7. http://notes.stephenholiday.com/Kafka.pdf

Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., & Tzoumas, K. (2015). Apache Flink: Stream and batch processing in a single engine. IEEE Data Engineering Bulletin, 38(4), 28–38. https://asterios.katsifodimos.com/assets/publications/flink-deb.pdf

Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., & Young, M. (2015). Hidden technical debt in machine learning systems. Advances in Neural Information Processing Systems, 28. https://proceedings.neurips.cc/paper/2015/hash/86df7dcfd896fcaf2674f757a2463eba-Abstract.html

Ghemawat, S., Gobioff, H., & Leung, S. T. (2003). The Google file system. Proceedings of SOSP 2003, 29–43. https://doi.org/10.1145/945445.945450

Nathan, M., Warren, J., & Marz, N. (2015). Big data: Principles and best practices of scalable real-time data systems. Manning. https://www.manning.com/books/big-data

Meeng, M., & Feelders, A. (2021). Feature engineering for real-time recommender systems. ACM RecSys 2021, 580–585. https://doi.org/10.1145/3460231.3478860

Zhang, M., McCarthy, Z., Finn, C., & Levine, S. (2019). Learning latent dynamics for planning from pixels. ICML 2019. http://proceedings.mlr.press/v97/zhang19b.html

Kleppmann, M., Beresford, A. R., & Svingen, B. (2019). Online event processing. ACM Queue, 17(1), 40. https://doi.org/10.1145/3317287.3321612